DEEP LEARNING

Lecture notes of course CS-E4890 at the Aalto University

Alexander Ilin

2022
Aalto university

Contents

1 Introduction **1**
- 1.1 What is deep learning . . . 1
  - 1.1.1 Learning hierarchical representations . . . 1
  - 1.1.2 Artificial neural networks . . . 3
- 1.2 Logistic regression classifier . . . 5
  - 1.2.1 A historical note on early models of neurons . . . 7
- 1.3 Multilayer perceptron . . . 8
- 1.4 The backpropagation algorithm . . . 10
- 1.5 Deep vs shallow networks . . . 13

2 Optimization **17**
- 2.1 Loss functions . . . 18
  - 2.1.1 Classification . . . 18
  - 2.1.2 Regression . . . 19
- 2.2 Analysis of convergence of gradient descent . . . 19
- 2.3 Input normalization . . . 24
- 2.4 Weight initialization . . . 25
- 2.5 Mini-batch training (stochastic gradient descent) . . . 27
- 2.6 Batch normalization . . . 29
- 2.7 Improved optimization algorithms . . . 34
  - 2.7.1 Momentum method . . . 35
  - 2.7.2 Rprop . . . 36
  - 2.7.3 RMSProp . . . 36
  - 2.7.4 Adam . . . 37

3 Convolutional neural networks **39**
- 3.1 Motivation . . . 39
- 3.2 Convolutional layer . . . 42
  - 3.2.1 Inputs with 1d structure . . . 42
  - 3.2.2 Inputs with 2d structure . . . 44
  - 3.2.3 2d convolutional layer in PyTorch . . . 46
  - 3.2.4 Translation equivariance . . . 47
- 3.3 Construction of a convolutional network . . . 48
  - 3.3.1 Pooling layer . . . 48

3.3.2 Stacking more layers . . . 49
3.3.3 Backpropagation through a convolutional layer . . . 50
3.4 Popular convolutional networks . . . 51
3.4.1 AlexNet . . . 52
3.4.2 VGG network . . . 52
3.4.3 ResNet . . . 53
3.4.4 Batch normalization in convolutional networks . . . 54
3.5 Applications of convolutional networks . . . 55
3.5.1 Processing speech with temporal convolutions . . . 55
3.5.2 Convolutional models for semantic segmentation . . . 56
3.5.3 Convolutional model for statistical machine translation . 57
3.5.4 Convolutional networks in reinforcement learning . . . 57
3.5.5 Protein folding . . . 58

4 Regularization 61
4.1 Overfitting . . . 61
4.2 Limit model capacity . . . 63
4.2.1 Reducing the model size . . . 63
4.2.2 L_2 regularization . . . 63
4.3 Early stopping . . . 65
4.4 Ensemble methods . . . 68
4.4.1 Dropout . . . 68
4.4.2 Probabilistic treatment: Bayesian neural networks . . . 70
4.4.3 Weight decay as Bayesian prior . . . 71
4.4.4 Variational approximation of the posterior distribution . . 72
4.4.5 Sampling approach: Stein variational gradient descent . . 73
4.4.6 Uncertainty estimation with Bayesian neural networks . . 74
4.5 Data augmentation . . . 75
4.5.1 Image transformations . . . 75
4.5.2 *mixup* . . . 76
4.5.3 Adversarial examples . . . 76
4.5.4 Adversarial training . . . 77
4.6 Rethinking generalization . . . 79
4.7 Hyperparameter search . . . 80
4.7.1 Grid search . . . 80
4.7.2 Random search . . . 82

5 Recurrent neural networks 83
5.1 Training recurrent neural networks . . . 85
5.2 Problems with RNN training . . . 87
5.2.1 Are there similar problems in backward computations? . . 89
5.2.2 Vanishing gradients . . . 90
5.2.3 Historical note on RNNs . . . 91
5.3 Gated recurrent unit (GRU) . . . 91
5.3.1 Connection to probabilistic graphical models . . . 93
5.4 Long short-term memory (LSTM) . . . 96

5.5 A sequence-to-sequence model for statistical machine translation 97

6 Attention-based models 99
6.1 Sequence-to-sequence models with attention 99
6.2 Convolutional sequence-to-sequence models 104
6.3 Transformer 109
6.3.1 Multi-head attention mechanism 109
6.3.2 Transformer encoder 112
6.3.3 Transformer decoder 113
6.3.4 Positional encoding of transformer 115
6.4 BERT: Transformer-based language model 117
6.4.1 Vision Transformer (ViT) 120

7 Graph neural networks 121
7.1 Motivation 121
7.2 Requirements for graph neural networks 123
7.3 Neural fingerprint networks 125
7.4 Interaction networks 127
7.5 Relational network for visual scene understanding 131
7.6 Graph Convolutional Networks 133
7.7 Recurrent Relational Networks 135
7.8 General algorithms for graph neural networks (GNNs) 137

8 Deep autoencoders 141
8.1 Motivation 141
8.2 Principal component analysis 142
8.3 Bottleneck autoencoders 144
8.4 Denoising autoencoders 145
8.5 Converting autoencoders into generative models 147
8.5.1 Generative model 147
8.5.2 Learning the model parameters with the EM algorithm . 148
8.5.3 Variational approximation 151
8.5.4 Variational autoencoder 154
8.6 VAE vs bottleneck autoencoder 157
8.7 Home assignment 158

9 Generative adversarial networks 161
9.1 Classical generative adversarial networks 161
9.1.1 Deep convolutional GAN 167
9.2 Improving stability of GAN training 167
9.2.1 Wasserstein GAN 167
9.2.2 Spectral Normalization 171
9.2.3 Zero-centered gradient penalties 173
9.3 Improved network architectures 176
9.3.1 Progressive growing (ProGAN) 176
9.3.2 Self-Attention GAN (SAGAN) 177

9.3.3 Style-Based Generators (StyleGAN) 178
9.4 Image-to-Image translation with conditional GANs 179

10 Autoregressive models 183
10.1 Convolutional autoregressive models 183
10.1.1 Autoregressive modeling of sequential data 183
10.1.2 PixelCNN: Autoregressive model of images 185
10.1.3 VQ-VAE 188
10.2 Transformer-based autoregressive models 192
10.2.1 Language models by OpenAI 192
10.2.2 Sparse transformer 193
10.2.3 DALL·E: Generating images from text 195

11 Flow-based generative models 197
11.1 Invertible generative process 197
11.2 Real NVP 199
11.2.1 Affine coupling layer 199
11.2.2 Squeeze operation 201
11.2.3 Split operation 202
11.2.4 Full model 202
11.3 Glow 204
11.3.1 Actnorm layer 204
11.3.2 Invertible 1×1 convolution 204

12 Learning with few labeled examples 205
12.1 Motivation 205
12.2 Transfer learning 206
12.3 Semi-supervised learning 206
12.3.1 Ladder networks 208
12.3.2 Π-model 209
12.3.3 Mean Teacher 210
12.3.4 Evolution of semi-supervised methods 211
12.4 Self-supervised learning 212
12.4.1 Contrastive Predictive Coding 213
12.4.2 A Simple Framework for Contrastive Learning (SimCLR) 216
12.4.3 Bootstrap your own latent (BYOL) 218
12.5 Few-shot learning 219
12.5.1 Siamese networks for one-shot learning 221
12.5.2 Matching networks 221
12.5.3 Prototypical networks 223
12.5.4 Model-Agnostic Meta-Learning 224
12.5.5 Reptile 225

Chapter 1

Introduction

1.1 What is deep learning

1.1.1 Learning hierarchical representations

Suppose that you work as a data scientist in a company and you need to solve a custom machine learning problem. For example, you need to design a classifier that detects spam emails or you need to extract line item information from scanned invoices. In many machine learning tasks, you can design a set of features that could be good input features for your machine learning model.

$$\text{Data} \rightarrow \text{Feature engineering} \rightarrow \text{Machine learning (e.g. classification)}$$

For example, a useful feature for the spam detector could be counts of certain words that may appear in an email. To extract line item information, we may classify each number that appears in a scanned invoice using as features the position of the number on the invoice and the words that appear in the proximity.

The benefit of feature engineering is that one can use the knowledge about a particular problem to design features that work well even for cases not seen in the training data. For example, these features can be designed to be invariant to various distortions.

What are the problems with feature engineering? The first problem is that for many tasks, it is difficult to come up with good features. Suppose, for example, that we want to classify images into classes 'churches' and 'other buildings' using as inputs two-dimensional maps of RGB values. What are useful features in this task?

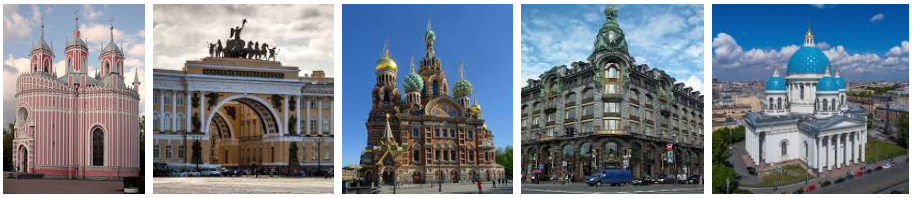
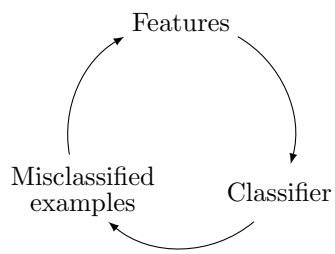


Figure 1.1: Example image classification problem. What are good features to detect images of churches?

Manually designing features for a complex task requires a great deal of human time and effort. It can take decades for an entire community of researchers to design good features. An example of manually designed features are SIFT features for image classification.

The second problem with feature engineering is that handcrafted features are not perfect. There are always examples that are not processed correctly by a machine learning model, which motivates engineering of new features. A typical design loop looks like this. We come up with some features, train a classifier and



check how the classifier works on a test set. We analyze misclassified examples and try to change our features so that we can reduce the number of mistakes. We train our classifier again using a new set of features, test the classifier and notice that it is not yet perfect. And we continue this process over and over again. During this process, features typically get very complex and at some point they become very difficult to maintain. It can be difficult to understand why certain features have been created and how important they are.

These problems can be overcome with *representation learning*.

$$\text{Data} \rightarrow \text{Features (representation)} \rightarrow \text{Classifier}$$

We can use machine learning not only to discover the mapping from the features to the output but also to discover the features (representations) themselves. The benefit of this approach is that a representation learning algorithm can discover a good set of features much faster: in days instead of decades of efforts of an entire research community.

Another advantage is that learned representations often result in much better performance compared to manually designed representations, especially in complex machine learning problems. With learned representations, an artificial intelligence system can rapidly adapt to new tasks, with minimal human intervention.

Deep learning is about learning hierarchical representations. Deep learning does representation learning by expressing more complex representations as combination of simpler representations. For example, we want to classify an object depicted on the image in Fig. 1.2. In the lowest level, we learn to detect simple features such as edges with different orientations. On the next level, we combine the edge feature into more complex concepts such as corners, semicircles and so on. Next, we combine those concepts into more complex features that represent, for example, wheels or windows of a car.

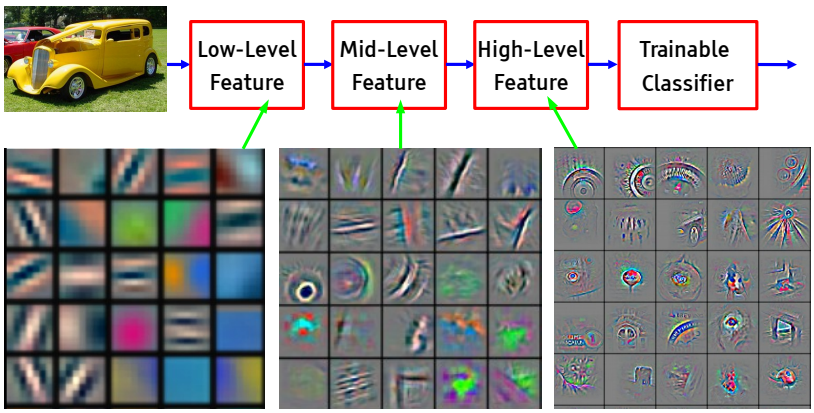


Figure 1.2: A hierarchy of features developed in an image classifcation problem (Zeiler and Fergus, 2013).

1.1.2 Artificial neural networks

Deep learning is also a field that studies artificial neural networks. Many ideas in deep learning models have been inspired by neuroscience. For example, the basic idea of having many computational units that become intelligent only via their interactions with each other is inspired by the brain. Another example is the neocognitron model (Fukushima, 1980) which was inspired by the structure of the mammalian visual system and later became the basis for the modern convolutional networks.

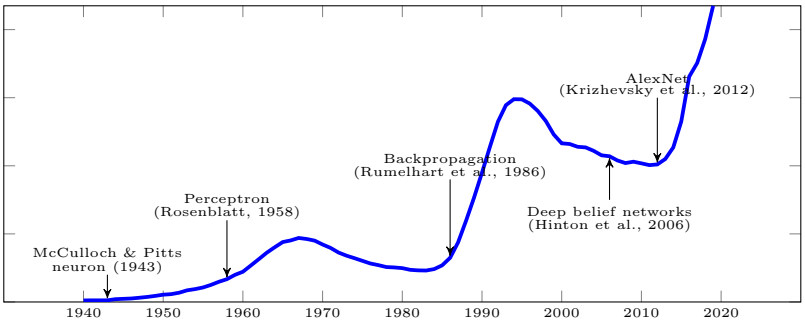


Figure 1.3: Frequency of phrases "cybernetics", "neural networks" and "deep learning" according to Google books.

There have been three waves of interest in artificial neural networks (see Fig. 1.3). The first wave started when the first models of the brain neuron function were proposed. Those models were linear classifiers and training was done using algorithms inspired by neuroscience.

The second wave of interest started in the mid-1980s largely inspired by the introduction of the backpropagation algorithm for training neural networks with multiple layers. Several influential neural models were proposed at that time including Hopfield networks, Boltzmann machines and self-organizing maps.

By the mid-1990s, the interest in neural networks had dropped. New methods showed same or better performance in supervised learning tasks (such as support vector machines, random forests). Deep networks (with more than

two-three hidden layers) did not provide much better results. Artificial neural networks became unpopular in early 2000s and researchers even avoided to use terms "neural network" or "multilayer perceptron" in research proposals.

The third wave started in 2006 when Hinton et al. (2006) demonstrated a better performance of a deep neural network over shallow networks on the MNIST dataset. At that time, the name "deep learning" was invented to rebrand artificial neural networks. The deep learning boom started in 2012 when researchers from Geoff Hinton's group won the ImageNet competition in computer vision by a large margin. The task in the competition was to train an image classification model using a training set with millions of labeled images. Prior to 2012, all winning solutions used handcrafted features which were traditional for computer vision. The winning solution of 2012, that is often called AlexNet by the name of the first author, was a deep convolutional neural network. After that, the majority of the proposed models have been based on neural networks.

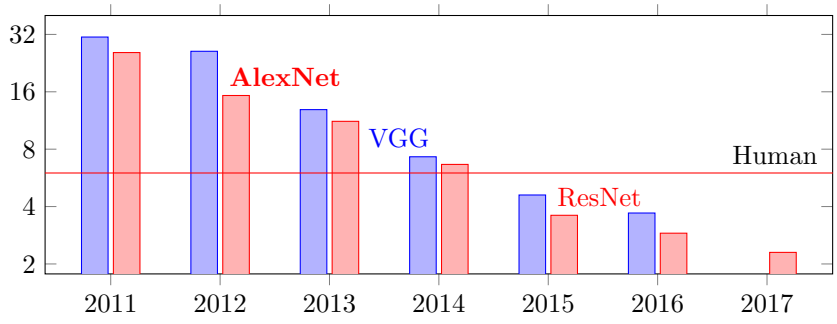


Figure 1.4: Classification errors in the winning solutions (red) and the runners-up (blue) in the yearly ImageNet competitions.

Since 2012, deep learning has entered multiple domains causing a major boost in performance. Examples include speech recognition (Fig. 1.5) or statistical machine translation (Fig. 1.6) where the state-of-the-art models are based on neural networks.

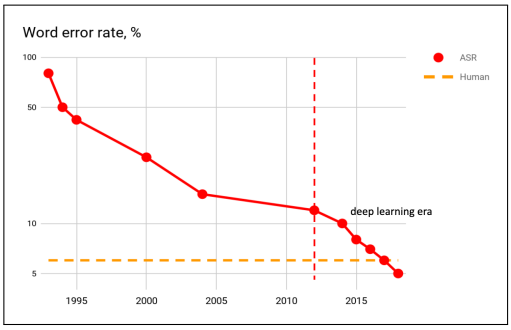


Figure 1.5: Progress in speech recognition. One of the first deep learning solution was presented in (Graves and Jaitly, 2014).

Although many ideas in artificial neural networks have been inspired by neuroscience, modern deep learning is a more general principle of learning multiple

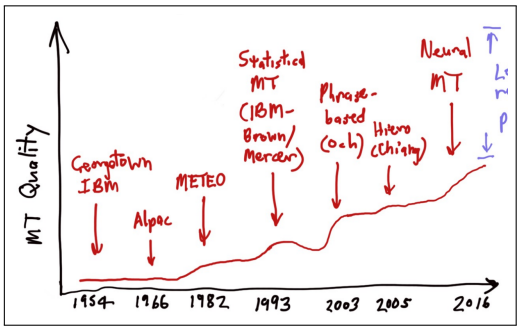


Figure 1.6: Progress in statistical machine translation. One of the first deep learning solution was presented in (Cho et al., 2014).

levels of composition, which can be applied in machine learning frameworks that are not necessarily neurally inspired.

1.2 Logistic regression classifier

Linear classifiers are the simplest machine learning models and can be viewed as elementary components of deep learning models.

Consider a binary classification problem (a classification problem with two classes). We have a data set which consists of pairs of inputs $\mathbf{x}^{(i)}$ and targets $y^{(i)}$:

$$(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(n)}, y^{(n)})$$

The inputs $\mathbf{x}^{(i)}$ are vectors with m elements and the targets $y^{(i)}$ are binary variables with two possible values 0 and 1 (which correspond to the two classes).

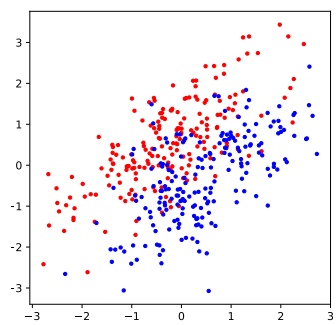


Figure 1.7: Training examples in a toy binary classification problem. The different colors represent the two classes.

We can use the training data to build a linear classifier

$$f(\mathbf{x}) = \sigma\left(\sum_{j=1}^{m} w_j x_j + b\right) = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

which contains a linear combination of the elements of the input vector and a nonlinearity σ that makes the output between 0 and 1. In the popular logistic regression model, the nonlinearity σ is a logistic function shown in Fig. 1.8. The

output of the logistic function can be viewed as the probability $p(y = 1 \mid \mathbf{x})$ that input example $\mathbf{x}$ belongs to class 1.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

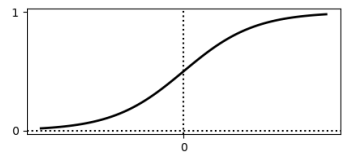


Figure 1.8: Logistic function.

We can tune our linear classifier assuming that the labels are distributed according to the Bernoulli distribution

$$p(y \mid \mathbf{x}, \mathbf{w}, b) = f(\mathbf{x})^y (1 - f(\mathbf{x}))^{1-y}$$

where $f(\mathbf{x})$ is the output of the model. For n training examples, we can write the joint probability of all the labels which gives the likelihood function:

$$p(\text{data} \mid \mathbf{w}, b) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}, b)$$

The likelihood is a function of the model parameters $\mathbf{w}$ and b. To find the best values of the model parameters, we can maximize the likelihood function or minimize minus the logarithm of that:

$$\mathcal{L}(\mathbf{w}, b) = -\log p(\text{data} \mid \mathbf{w}, b) = -\sum_{i=1}^{n} y^{(i)} \log f(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - f(\mathbf{x}^{(i)})).$$

This loss function is often called *binary cross entropy*.

Let us consider a toy binary classification problem with the training set shown in Fig. 1.7. For visualization purposes, let us use a logistic regression model with only two parameters w_1 and w_2 and without the bias term b:

$$f(\mathbf{x}) = \sigma\left(w_1 x_1 + w_2 x_2\right).$$

The binary cross-entropy loss is a function of two parameters and it can be visualized using a contour plot in Fig. 1.9. Each line on this plot connect points with the same value of the loss. The optimal solution for our toy problem is located in the middle of the plot at the center of the ellipse-like isolines. In every location of the parameter space, we can compute the *gradient* which is a vector of partial derivatives:

$$\mathbf{g}(\mathbf{w}) = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_m} \end{pmatrix}$$

The gradient $\mathbf{g}(\mathbf{w})$ points in the direction of the greatest rate of increase of $\mathcal{L}$. We visualize the gradients in Fig. 1.9 with arrows. The magnitude of the

gradient is the slope of the graph of function $\mathcal{L}$ in that direction. Note that the gradient is orthogonal to the isolines of the contour plot. Note also that the magnitudes of the gradient are smaller when we get closer to the optimal solution.

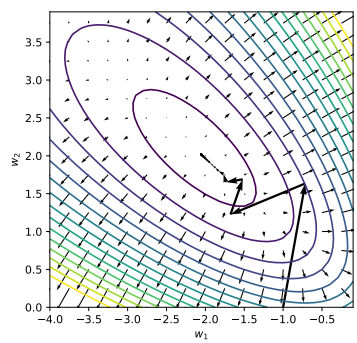


Figure 1.9: Contour plot of the binary cross entropy loss in the toy classification problem. Small black arrows represent the gradient of the loss. The large black arrows illustrate a trajectory obtained with the gradient descent algorithm.

Since the gradient points in the direction in which the function grows with the highest rate and we want to minimize it, we can update the parameters $\mathbf{w}$ in the direction which is opposite to the direction of the gradient:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{g}(\mathbf{w})$$

We take a step of size η in that direction. Unfortunately, we do not end up at the optimal solution and we need to iterate this process multiple times

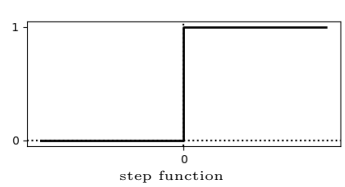
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}(\mathbf{w}_t)$$

until we converge to the optimum. This optimization algorithm is called *gradient descent*. The step size η is often called the *learning rate*. In Fig. 1.9, we illustrate the trajectory of the gradient descent algorithm when it is applied to tune the linear classifier in our toy binary classification problem.

1.2.1 A historical note on early models of neurons

The first linear classifier was proposed as a model of brain function by McCulloch and Pitts (1943). McCulloch-Pitts neuron was a linear binary classifier for binary inputs $x_j \in \{0, 1\}$

$$y = \phi\left(\sum_{j=1}^{m} w_j x_j + b\right)$$



where $\phi(\cdot)$ was a step function. There was no training procedure: to produce the desired output, the parameters w_j, b were set by a human operator.

The first training algorithm for linear classifiers was proposed by Rosenblatt (1958). He designed a machine (not a program) called *perceptron* that implemented a binary classifier

$$\hat{y} = \text{sign}(\mathbf{w}^\top \mathbf{x})$$

The classifier was trained using examples $(\mathbf{x}^{(i)}, y^{(i)})$ with $\mathbf{x}^{(i)} \in \mathbb{R}^m, \quad y^{(i)} \in \{-1, +1\}$. The training algorithm was inspired by a neuroscientific theory of Donald Hebb. The training examples were processed one at a time and if example $\mathbf{x}^{(i)}$ was misclassified, the classifier was adapted to increase the weights between neurons whose activities were positively correlated:

$$\mathbf{w} \leftarrow \mathbf{w} + y^{(i)}\mathbf{x}^{(i)}.$$

The first models were very limited because they were linear classifiers and could not solve complex classification problems. They could not separate linearly inseparable classes, for example, they could not solve the famous XOR problem. In the XOR classification problem, there are two binary inputs and we want to train a classifier to implement the XOR function shown on this plot. This is not possible with a linear model while modeling functions OR or AND is possible with a linear model.

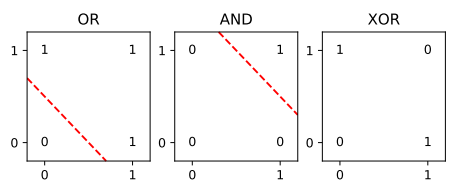
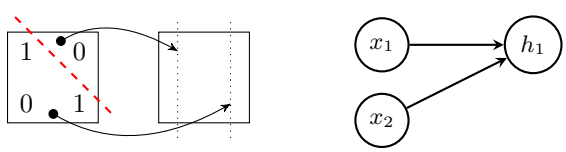


Figure 1.10: Classical examples of binary classification problems. While functions OR and AND can be implemented by linear models, function XOR cannot be implemented by a linear model.

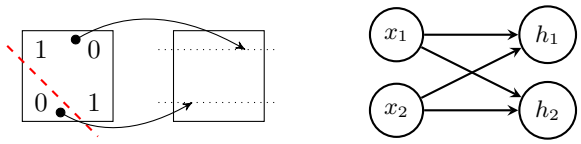
This problem was emphasized in the influential book "Perceptrons" by Minsky and Papert (1969). They argued that more complex (nonlinear) problems have to be solved with multiple layers of perceptrons (what we now call multilayer neural networks).

1.3 Multilayer perceptron

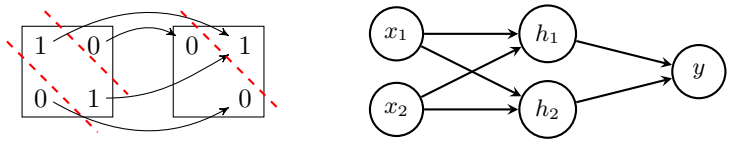
Let us see how we can solve the XOR problem with a network of perceptrons. We can construct the model in the following way. First, we introduce a neuron that can linearly separate the input space as shown below:



Then, we can add another neuron h_2 which can do another kind of separation of the input space:



Now we mapped the original two-dimensional data into a new two-dimensional space where linear separation is possible:



Finally, we add neuron y on top of neurons h_1 and h_2 and we solve the classification problem.

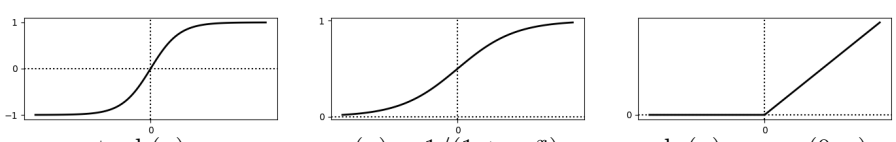
Now we have a network with two layers of neurons. The first layer contains neurons h_1, h_2. Its outputs are not visible from outside the model and therefore this layer is called a *hidden layer*. The second layer consists of one neuron y and this layer is called the *output layer*. A neural network with this architecture is called a *multilayer perceptron* (MLP). The inputs of the MLP are often called the *input layer*.

Of course, a multilayer perceptron can have more layers and many more neurons. Each neuron of an MLP implements a function

$$y = \phi\left(\sum_{j=1}^{m} w_j x_j + b\right) = \phi\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

which resembles a simple linear classifier that we considered in Section 1.2. The layers in an MLP are called *fully-connected* because each neuron is connected to each neuron in the previous layer. We will often use a more compact representation of a multilayer network in which one node corresponds to an entire layer (see Fig. 1.11). In that case we can stack the weight vectors $\mathbf{w}$ of all nodes in the layer into a matrix $\mathbf{W}$ and summarise the operation performed by a layer as the nonlinearity applied to a matrix-vector product.

Nonlinearities ϕ used after an affine transformation of a neuron's inputs are often called *activation functions*. Popular nonlinearities include



$\tanh(x)$ $\sigma(x) = 1/(1 + e^{-x})$ $\mathrm{relu}(x) = \max(0, x)$

In modern deep learning models, $\mathrm{relu}(z)$ is the most popular choice for multilayer networks without recurrence, such as MLPs. The two other activation functions $\tanh(x)$, $\sigma(x) = 1/(1 + e^{-x})$ were more popular in the second wave of artificial

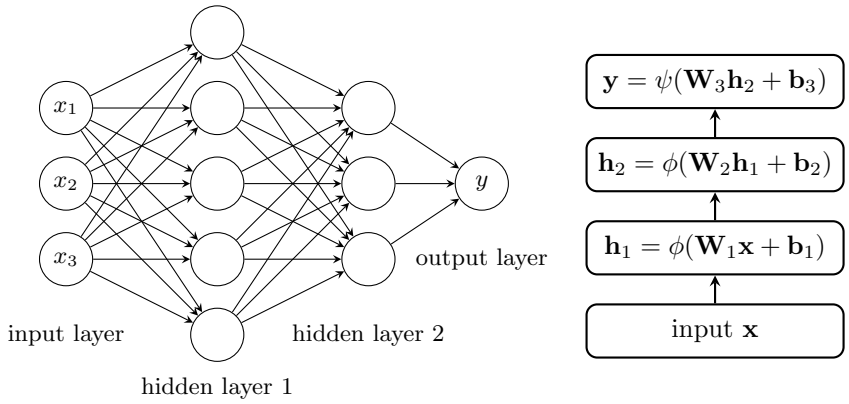


Figure 1.11: Architecture of a multilayer perceptron. Left: A diagram with individual neurons. Right: A more compact diagram with one node representing one layer of neurons.

neural networks but they are still widely used in deep learning models, for example, in recurrent neural networks (see Chapter 5).

What if we do not use any nonlinearity? Then, we simply get a linear model:

$$\begin{aligned}\mathbf{h}_2 &= \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \\ &= (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}'\mathbf{x} + \mathbf{b}'\end{aligned}$$

Therefore, using a stack of linear layers without activation functions does not make much sense: using a single linear layer would have the same representational power.

If we train a neural network to solve a binary classification problem, then we can use the same loss function that we used for training a linear binary classifier in Section 1.2:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} y^{(i)} \log f(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - f(\mathbf{x}^{(i)}))$$

where function f is composed of several functions f_i implemented by the different layers of the network. For a three-layer network shown in Fig. 1.11,

$$f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x}, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2), \boldsymbol{\theta}_3)\,,$$

where $\boldsymbol{\theta}_1 = (\mathbf{W}_1, \mathbf{b}_1)$, $\boldsymbol{\theta}_2 = (\mathbf{W}_2, \mathbf{b}_2)$, $\boldsymbol{\theta}_3 = (\mathbf{W}_3, \mathbf{b}_3)$. Again, we can tune the parameters $\boldsymbol{\theta}_k$ of the network by maximizing the log-likelihood, for example, using gradient descent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{g}(\boldsymbol{\theta}_t)$$

In order to do that, we need to compute the gradient $\mathbf{g}(\boldsymbol{\theta}) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. Backpropagation is an algorithm that allows to compute the gradient efficiently for a multilayer neural network.

1.4 The backpropagation algorithm

Even though the idea of using multilayer perceptrons for solving nonlinear classification problems existed already in the 1960s (Minsky and Papert, 1969), no

one knew how to train multilayer perceptrons. Rosenblatt's learning algorithm did not work for multiple layers because a training example $(\mathbf{x}^{(i)}, y^{(i)})$ only specifies the correct output for the final output layer. Thus, there was no way to know how to adjust the weights of the perceptrons in layers before the last one.

How to train MLP networks was well understood only in the mid 1980s after an influential paper by Rumelhart, Hinton and Williams (1986). In that paper, they showed how to compute the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ efficiently using the backpropagation algorithm. Backpropagation is basically the application of the chain rule of differentiation to models with multiple layers. It was proposed by several researchers even earlier (Linnainmaa, 1970; Werbos, 1982) but became popular and well understood after 1986.

Let us review the chain rule to compute the derivative of a composite function. Suppose that function F is a scalar function of one variable x and it is a composite of functions f and g. Then, its derivative of F is given by the product of the derivatives of f and g.

$$\begin{aligned} F(x) &= f(g(x)) \\ F'(x) &= f'(g(x))g'(x) \end{aligned}$$

Now suppose that we have a model with two layers that operates only with scalar signals.

$$\begin{aligned} \mathcal{L} &= \mathcal{L}(y) \\ y &= f_2(h, \theta) \\ h &= f_1(x, w) \end{aligned}$$

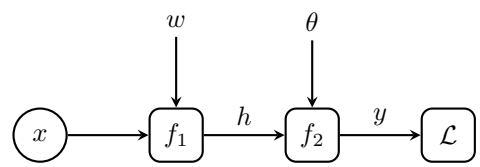


Figure 1.12: A two-layer network operating with scalar signals.

We can compute the derivatives wrt the model parameters θ, w by applying the chain rule.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w} \end{aligned}$$

We can compute the derivatives efficiently by storing intermediate results. We start from the end of the network and compute $\frac{\partial \mathcal{L}}{\partial y}$. Then, we use the chain rule to compute

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial h} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h} \end{aligned}$$

Then, we apply the chain rule again and compute

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial h}\frac{\partial h}{\partial w}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta}$$
$$\frac{\partial \mathcal{L}}{\partial w} = \underbrace{\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}}\frac{\partial h}{\partial w}$$

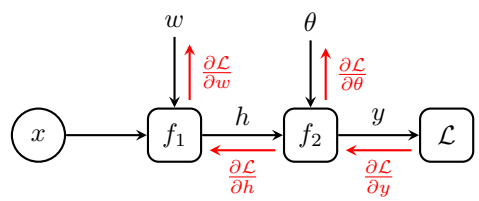


Figure 1.13: Backpropagation in a two-layer network operating with scalar signals.

This is illustrated in Fig. 1.13.

To extend these computations to vector signals, we need to use the chain rule for multi-variable functions. Suppose that we have a multi-variable function F that is a composite of two functions f and g:

$$\mathbf{y} = F(\mathbf{x}) = f(\mathbf{u}), \quad \mathbf{u} = g(\mathbf{x}), \qquad \mathbf{y} \in \mathbb{R}^M,\ \mathbf{u} \in \mathbb{R}^K,\ \mathbf{x} \in \mathbb{R}^N$$

Then, the chain rule is written in terms of Jacobian matrices. The Jacobian matrix of function F is a matrix of the partial derivatives of each output wrt each input:

$$\mathbf{J}_F = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}$$

Then, the chain rule is

$$\mathbf{J}_F(\mathbf{x}) = \mathbf{J}_f(\mathbf{u})\mathbf{J}_g(\mathbf{x})$$

where $\mathbf{J}_f$, $\mathbf{J}$ are the Jacobian matrices of functions f and g respectively. Each element of the Jacobian $\mathbf{J}_F$ can be computed by the rules of matrix multiplication:

$$\frac{\partial y_j}{\partial x_i} = \sum_{k=1}^{K} \frac{\partial y_j}{\partial u_k}\frac{\partial u_k}{\partial x_i}$$

Now let us consider a two-layer model which operates with vector signals (Fig. 1.14).

$$\mathcal{L} = \mathcal{L}(\mathbf{y})$$
$$\mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta})$$
$$\mathbf{h} = f_1(\mathbf{x}, \mathbf{w})$$
$$\mathbf{y} \in \mathbb{R}^K, \mathbf{h} \in \mathbb{R}^L,\ \mathbf{x} \in \mathbb{R}^N$$

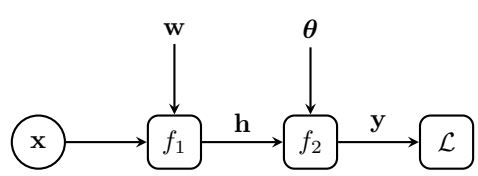


Figure 1.14: A two-layer network operating with multidimensional signals.

We apply the chain rule to compute the derivatives wrt the model parameters. We can compute the derivatives sequentially going from the outputs of the network towards the inputs. We start with computing $\frac{\partial \mathcal{L}}{\partial y_k}$ and use them to

compute

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

Finally, we propagate the derivatives through the first layer:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^{L} \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}.$$

This algorithm for computing derivatives in a multilayer network is called *back-propagation* because we propagate the derivatives starting from the outputs of the network towards the inputs. The computations performed during backpropagation are illustrated in Fig. 1.15.

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^{L} \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}$$

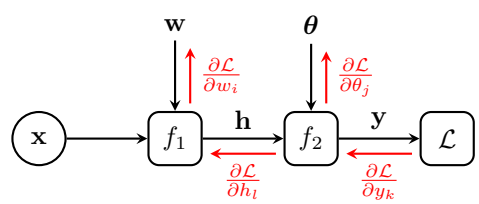


Figure 1.15: Backpropagation in a two-layer network operating with multidimensional signals.

When we implement backpropagation in software, it is convenient to write code for each block of a neural network such that it contains

- forward computations $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta})$
- backward computations that transform the derivatives wrt the block's outputs $\frac{\partial \mathcal{L}}{\partial y_k}$ into the derivatives wrt all its inputs: $\frac{\partial \mathcal{L}}{\partial x_l}$, $\frac{\partial \mathcal{L}}{\partial \theta_j}$,

which is illustrated in Fig. 1.16. We will practice implementing forward and backward computations in the home assignment.

1.5 Deep vs shallow networks

Multilayer neural networks were invented long time ago but they became the state of the art in many applications only recently. One characteristic property of modern neural networks is their depth. For example, modern convolutional neural networks for computer vision tasks can have more than 100 layers.

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial x_l} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial x_l}$$

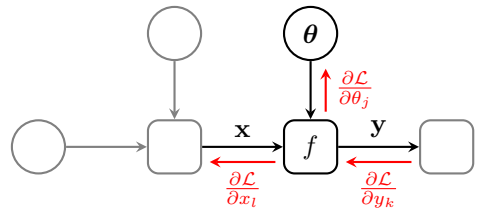


Figure 1.16: Implementation of backpropagation in software.

During the second wave (Fig. 1.3), neural networks were not very deep, only with two-three hidden layers. Deeper networks did not provide better performance, largely because training of deeper networks is a harder optimization problem (as we discuss in Chapter 2). There were no theoretical results that deep networks had better representational power. Shallow networks were known to be universal approximators. The classical universal approximation theorem (Cybenko, 1989) says that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate any well-behaved function with any given accuracy.

Why is the depth important and what is the advantage of deep neural networks compared to shallow ones? Today there are several theoretical results which suggest that deep networks have greater representational power.

Suppose that we have a multilayer perceptron network with the relu activation function in every layer. Such a network implements a piece-wise linear function. For example, the plot in Fig. 1.17 presents an example of a function with one input and one output which is implemented by an MLP with one hidden layer with three hidden neurons. This function contains four linear regions.

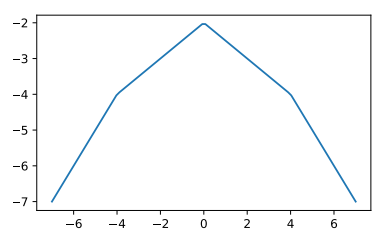


Figure 1.17: An example piecewise linear function implemented by a multilayer network with one hidden layer with three neurons activated with relu.

The number of linear regions that the model can have reflects its representational power. A more flexible model can have a greater number of linear regions. There are theoretical results which estimate the number of possible linear regions for an MLP with relu nonlinearities.

Suppose that the network has n inputs. If the network contains a single layer model with Lm hidden units, then the number of linear regions behaves as $O(L^n m^n)$ (Pascanu et al., 2013). This results says that the number of linear regions has polynomial growth wrt to the number of neurons. However, if the network contains L hidden layers of width $m \geq n$, then the model can compute

functions that have $\Omega((m/n)^{(L-1)n} m^n)$ linear regions (Montúfar et al., 2014). This means that that number of linear regions grows exponentially with the number of layers.

Another theoretical result says that both shallow and deep networks can approximate arbitrarily well any continuous function $\mathbb{R}^n \to \mathbb{R}$ on a compact domain but the number of required parameters grows exponentially wrt the number of inputs n to achieve a given accuracy. However, if the approximated function f is a hierarchical composition of local functions (for example, the function can be computed as shown in Fig. 1.18), then deep networks of the

$$f(x_1, \ldots, x_8) = \\ h_3(h_{21}(h_{11}(x_1, x_2), h_{12}(x_3, x_4)), \\ h_{22}(h_{13}(x_5, x_6), h_{14}(x_7, x_8)))$$

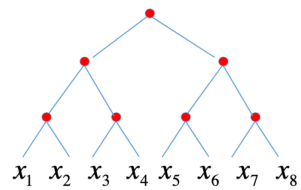


Figure 1.18: Assumed compositional structure of the approximated function in the analysis by Poggio et al., (2019).

convolutional type can have a linear dependence on n unlike shallow networks which have exponential dependence (Poggio et al., 2019).

This results suggests that deep networks with a much smaller number of parameters can achieve equivalent approximation accuracy compared to shallow networks, provided that the approximated function has certain (composite) structure.

Experimentally, it has been found that deeper models perform better. For example, increasing the number of parameters in layers without increasing their depth is not as effective at increasing test set performance. Fig. 1.19 presents an example of accuracies achieved for a Street View House Numbers dataset using deep convolutional neural networks (Goodfellow et al., 2014). Shallow models

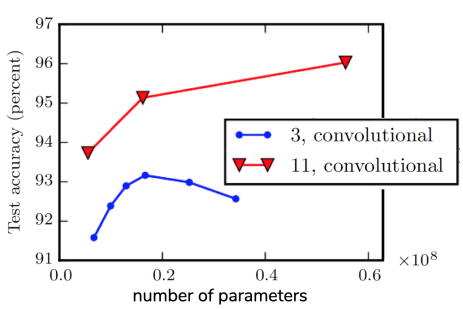


Figure 1.19: Accuracies achieved for a Street View House Numbers dataset using deep convolutional neural networks

overfit at around 20 million parameters while deep models benefit from having over 60 million parameters.

Deep models have a built-in assumption (often called an inductive bias) that a modeled function should consist of many simple functions composed together. This assumption turns out to work very well in many real-world tasks.

Chapter 2

Optimization

Many components of deep learning have been invented long time ago. The first linear model was invented in the 1950s, the training algorithm for training deep networks was well understood in the 1980s, convolutional neural networks were invented in the 1980s as well. Why did deep learning start only 25 years later? Geoff Hinton gave four reasons for that:

- Our labeled datasets were thousands of times too small.
- Our computers were millions of times too slow.
- We initialized the weights in a stupid way.
- We used the wrong type of non-linearity.

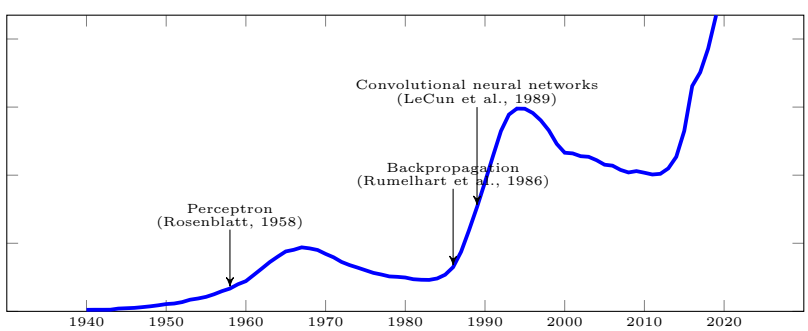


Figure 2.1: Frequency of phrases "cybernetics", "neural networks" and "deep learning" according to Google books.

The last two reasons indicate that training of a deep neural network is a difficult optimization problem. In order to get full benefit of deep architectures (compared to shallow ones used in the second wave of neural networks), one has to use several tricks including input normalization, weight initialization, mini-batch training (stochastic gradient descent), improved optimizers, batch normalization. We study these techniques in this chapter.

2.1 Loss functions

In this chapter, we assume that our model is a multilayer neural network which we train to solve a supervised learning task. Given a set of training examples

$$\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\},$$

we train the network to produce desired output $\mathbf{y}$ for given input $\mathbf{x}$. The two most common tasks of supervised learning are classification and regression.

2.1.1 Classification

In classification tasks, the output is a discrete variable (the label of the predicted class). The desired output (target) can be represented as a one-hot vector $\mathbf{y}$, which is a binary vector with only one non-zero element:

$$y_j \in \{0, 1\} \qquad \sum_{j=1}^{K} y_j = 1$$

For example, in the case of $K = 3$ output classes, one-hot representation is given by:

$$\text{class 1: } y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{class 2: } y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{class 3: } y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

We want our neural network network to produce vector $\mathbf{f}$ whose j-th element f_j is the probability that input $\mathbf{x}$ belongs to class j. Therefore, we need to guarantee that the output satisfies the following condition

$$0 \leq f_j \leq 1 \qquad \sum_{j=1}^{K} f_j = 1$$

We can guarantee that by applying the following transformation to the outputs $\mathbf{h}$ of the last layer of the network:

$$f_j = \frac{\exp h_j}{\sum_{j'=1}^{K} \exp h_{j'}}$$

where h_j is the j-th element of vector $\mathbf{h}$. This function is called *softmax* because if one of the elements h_j is much larger than the rest of the elements

$$h_j \gg h_i, \quad i \neq j,$$

then $\mathbf{f} \approx [0, ..., 0, 1, 0, ..., 0]$ which is a one-hot representation of j, the index of the maximum element of $\mathbf{h}$.

To solve a classification problem, it is common to tune parameters $\boldsymbol{\theta}$ of a neural network $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$ by minimizing the following loss function:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{j=1}^{K} y_j^{(n)} \log f_j(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \tag{2.1}$$

which is the extension of the binary cross-entropy loss to the case of multiple classes. This loss is the negative log-likelihood for a probabilistic model with a categorical distribution (also called multinoulli distribution) for labels $\mathbf{y}$:

$$p(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) = \mathrm{Cat}(\mathbf{y} \mid \mathbf{f}(\mathbf{x}, \boldsymbol{\theta})) = \prod_{j=1}^{K} f_j^{y_j}$$

The parameters of the categorical distribution are given by the output $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$ of the network.

This loss in (2.1) is often called the *cross-entropy loss*. The cross-entropy between two discrete probability distributions p and q is defined as

$$\mathcal{H}(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

If you compare this expression with the loss in (2.1), you can see that the loss $\mathcal{L}(\boldsymbol{\theta})$ can be seen as the cross-entropy between the distribution defined by targets $\mathbf{y}^{(n)}$ and the distribution $\mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta})$ defined by the output of the network.

2.1.2 Regression

Regression is the second common task of supervised learning. In regression problems, the targets are real-valued vectors $\mathbf{y}^{(n)} \in \mathbb{R}^K$. In this case, we can tune the parameters of the network by minimizing the mean-squared error (MSE):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right\|^2 .$$

This loss function also has a probabilistic interpretation. Assuming that the conditional distribution of the targets is Gaussian

$$p(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y} \mid \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})$$

leads to the mean-squared error loss.

2.2 Analysis of convergence of gradient descent

Gradient descent is the simplest optimization method that one can use for tuning the parameters of a neural network. Its update rule

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}(\boldsymbol{\theta}_t)$$

has the learning rate parameter η which has a major effect on the convergence of the gradient descent. With a small learning rate, one may have to take too many small steps and thus the convergence can be slow (see Fig. 2.2). With a large learning rate, the optimization trajectory can be zigzaging. When the learning rate is too high, the optimization procedure can even diverge.

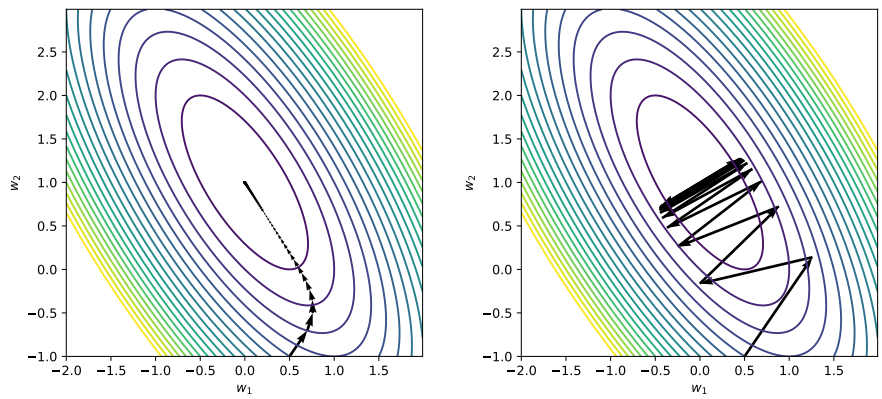


Figure 2.2: The effect of the learning rate on the convergence speed. Left: Small η may lead to slow convergence. Right: Large η may lead to oscillations and even divergence.

What is the optimal value of the learning rate? And if we select the learning rate optimally, what affects the rate of convergence of the gradient descent? Let us analyze the convergence of gradient descent for a quadratic function (Goh, 2017)

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w} - \mathbf{b}^\top \mathbf{w}.$$

For this function, we can compute the optimal solution $\mathbf{w}_*$ analytically:

$$\mathbf{w}_* = \mathbf{A}^{-1}\mathbf{b}$$

The gradient of the quadratic loss is equal to $\mathbf{A}\mathbf{w}_t - \mathbf{b}$ and therefore the gradient descent iteration is given by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta(\mathbf{A}\mathbf{w}_t - \mathbf{b})$$

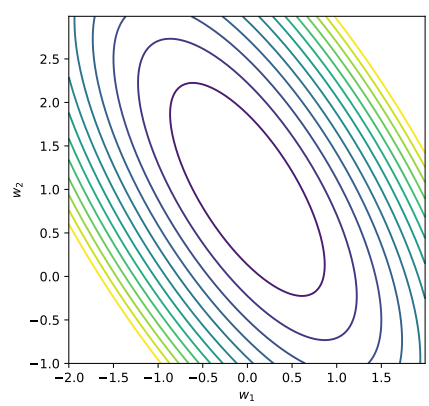


Figure 2.3: An example contour plot of a quadratic loss function.

If vector $\mathbf{w}$ has two elements, we can visualize the quadratic loss with a contour plot. The contour plot may look like the one shown in Fig. 2.3. The isolines of the plot have the shape of ellipses concentrated around the minimum

of the loss function. The axes of the ellipses are defined by the eigenvectors of matrix $\mathbf{A}$ and the eigenvalues λ_m of $\mathbf{A}$ determine the curvature of the loss function in the corresponding direction: larger values of λ_m correspond to higher curvatures.

For the convergence analysis, it is convenient to change the coordinate system such that the optimal solution $\mathbf{w}_*$ is located at the origin of the new coordinate system and the new axes are aligned with the eigenvectors of $\mathbf{A}$. We compute the eigenvalue decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{Q}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_M)\mathbf{Q}^\top$$

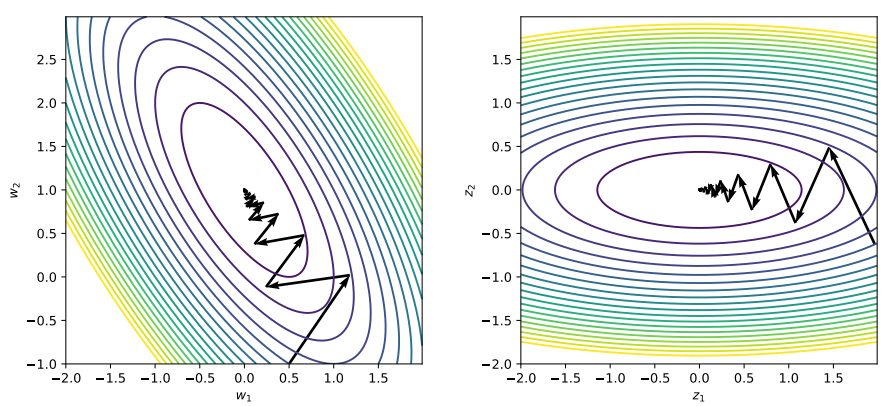
where $\mathbf{Q}$ is an orthogonal matrix and λ_m are ordered eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_M$. Then, we subtract the optimal solution $\mathbf{w}_*$ and rotate the coordinate system using the orthogonal matrix $\mathbf{Q}$:

$$\mathbf{z} = \mathbf{Q}^\top(\mathbf{w} - \mathbf{w}_*) \tag{2.2}$$

To go back from the new coordinates $\mathbf{z}$ to the original coordinates $\mathbf{w}$, we can use the inverse transformation:

$$\mathbf{w} = \mathbf{w}_* + \mathbf{Q}\mathbf{z}$$

The original and new coordinate systems are illustrated in Fig. 2.4.



original coordinate system $\mathbf{w}$ new coordinate system $\mathbf{z}$

Figure 2.4: Original and transformed coordinate systems for a quadratic loss function.

Now we can re-write the gradient-descent update rule in the new coordinate system.

$$\begin{aligned}
\mathbf{z}_{t+1} &= \mathbf{Q}^\top(\mathbf{w}_{t+1} - \mathbf{w}_*) = \mathbf{Q}^\top(\mathbf{w}_t - \eta(\mathbf{A}\mathbf{w}_t - \mathbf{b}) - \mathbf{w}_*) \\
&= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t - \eta(\mathbf{A}(\mathbf{w}_* + \mathbf{Q}\mathbf{z}_t) - \mathbf{b})) \\
&= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t - \eta(\mathbf{b} + \mathbf{A}\mathbf{Q}\mathbf{z}_t - \mathbf{b})) \\
&= \mathbf{z}_t - \eta\mathbf{Q}^\top\mathbf{A}\mathbf{Q}\mathbf{z}_t \\
&= \mathbf{z}_t - \eta\,\mathrm{diag}(\lambda_1, \ldots, \lambda_M)\mathbf{z}_t \\
&= (\mathbf{I} - \eta\,\mathrm{diag}(\lambda_1, \ldots, \lambda_M))\mathbf{z}_t
\end{aligned} \tag{2.3}$$

As we can see from (2.3), the gradient-descent update rule in the new coordinate system can be written separately for each element of $\mathbf{z}$:

$$(z_m)_{t+1} = (1 - \eta\lambda_m)(z_m)_t$$

where $(z_m)_t$ is the m-th element of $\mathbf{z}$ after iteration t.

In the new coordinate system, the optimum is located at the origin $\mathbf{z}_* = 0$ and therefore the distance to the optimum in the m-th coordinate is given by $|z_m|$. Then, we can write the rate of convergence as

$$\text{rate}(\eta) = \frac{|(z_m)_{t+1}|}{|(z_m)_t|} = |1 - \eta\lambda_m|$$

It is the ratio of the distances to the optimum after the next iteration $t + 1$ and the current iteration t. If this ratio is small, we converge quickly to the optimum. For convergence, the ratio has to be smaller than 1, which means that the following condition should hold

$$|1 - \eta\lambda_m| < 1.$$

In the ideal case, we jump to the optimal solution after one iteration, which happens if

$$|1 - \eta\lambda_m| = 0.$$

Since we need to optimize all the coordinates of $\mathbf{z}$, the overall convergence rate is determined by the slowest component.

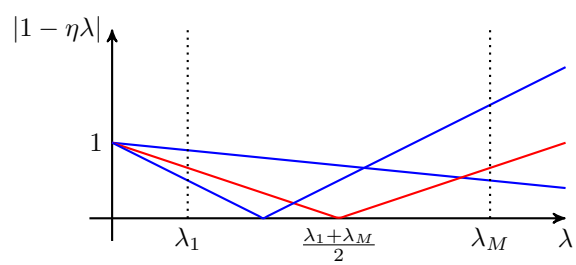


Figure 2.5: The rate of convergence as a function of the eigenvalues of $\mathbf{A}$ for three values of the learning rate η. The red line corresponds to the optimal learning rate in which case the rate of convergence for the largest and the smallest eigenvalues is the same.

Fig. 2.5 illustrates the rate of convergence depending on the value of λ. You can see that the worst case (the slowest convergence) is obtained either for the smallest eigenvalue λ_1 or for the largest eigenvalue λ_M. The overall convergence rate is determined by the slowest component:

$$\text{rate}(\eta) = \max_m |1 - \eta\lambda_m| = \max\{|1 - \eta\lambda_1|, |1 - \eta\lambda_M|\}$$

The worst-case convergence rate is minimized when the rates for λ_1 and λ_M are equal, which suggests the optimal value of the learning rate:

$$\eta_* = \left(\frac{\lambda_1 + \lambda_M}{2}\right)^{-1}.$$

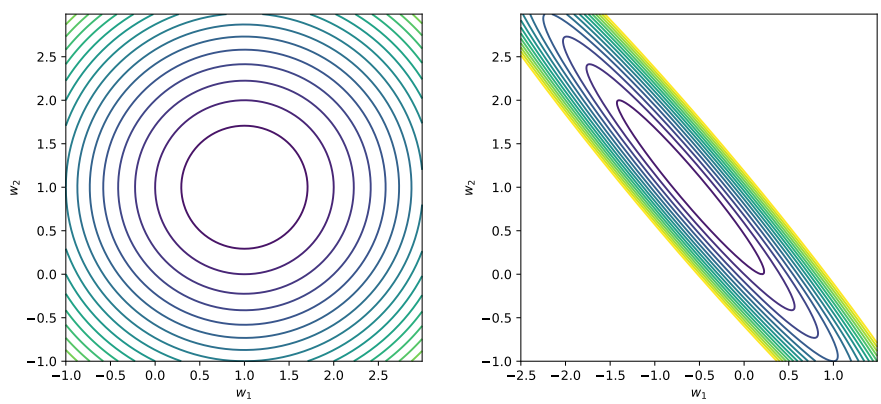
With the optimal learning rate, the rate of convergence is

$$\begin{aligned}\text{rate}(\eta_*) &= \left|1 - \left(\frac{\lambda_1 + \lambda_M}{2}\right)^{-1} \lambda_1\right| = \left|\frac{\lambda_1 + \lambda_M - 2\lambda_1}{\lambda_1 + \lambda_M}\right| = \frac{\lambda_M - \lambda_1}{\lambda_M + \lambda_1} \\ &= \frac{\lambda_M/\lambda_1 - 1}{\lambda_M/\lambda_1 + 1} = \frac{\kappa(\mathbf{A}) - 1}{\kappa(\mathbf{A}) + 1}\end{aligned}$$

where $\kappa(\mathbf{A}) = \frac{\lambda_M}{\lambda_1}$ is the condition number of matrix $\mathbf{A}$. We know from linear algebra that $\kappa(\mathbf{A})$ is a measure of how close to singular matrix $\mathbf{A}$ is. For the analysis of convergence of gradient descent, $\kappa(\mathbf{A})$ is a measure of how poorly the gradient descent will perform:

- in the ideal situation $\kappa(\mathbf{A}) = 1$, we get the fastest convergence;
- for large $\kappa(\mathbf{A})$ the convergence can be slow.

This is illustrated in Fig. 2.6. In the lhs plot, the condition number $\kappa(\mathbf{A}) = 1$ and we can converge to the optimal solution in one iteration from any starting point if we select the optimal learning rate. In the second case, matrix $\mathbf{A}$ is ill-conditioned, which causes zigzaging of the optimization trajectory and slow convergence.



$\kappa(\mathbf{A}) = 1$ large $\kappa(\mathbf{A})$

Figure 2.6: Contour plots of a quadratic loss function for different condition numbers of $\mathbf{A}$. Left: For $\kappa(\mathbf{A}) = 1$, gradient descent can converge in one iteration. Right: Large $\kappa(\mathbf{A})$ causes slow convergence.

We performed the analysis for a quadratic objective function. For non-quadratic functions, the error surface is well approximated locally by a second-order Taylor series expansion

$$\mathcal{L}(\mathbf{w}) \approx \mathcal{L}(\mathbf{w}_t) + \mathbf{g}^\top(\mathbf{w} - \mathbf{w}_t) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_t)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_t) \tag{2.4}$$

which is a a quadratic function (see Fig. 2.7). Here, $\mathbf{g}$ is the gradient (the vector of the first-order partial derivatives) and $\mathbf{H}$ is the *Hessian* matrix, which is the

matrix of the second-order derivatives

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_M} \end{pmatrix}$$

For the quadratic loss $\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w} - \mathbf{b}^\top \mathbf{w}$, the Hessian matrix is equal to matrix $\mathbf{A}$, which suggests that the convergence of the gradient descent is affected by the properties of the Hessian. If the Hessian matrix is ill-conditioned, the gradient descent algorithm converges slowly.

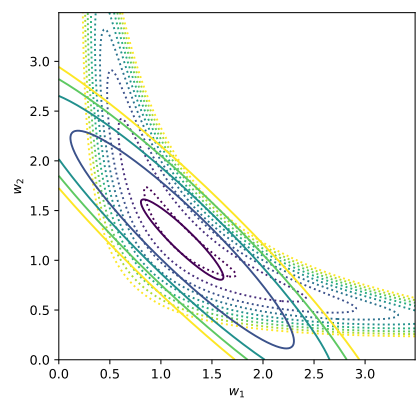


Figure 2.7: A quadratic approximation of a non-quadratic loss function.

The properties of the Hessian matrix affect the landscape of the optimization problem. The eigenvalues of $\mathbf{H}$ determine the curvature of the objective function. Larger λ correspond to higher curvatures in the corresponding direction. If we arrive in a *critical point* $\mathbf{w}_*$ (which is a point with a zero gradient), we can check whether the critical point is a maximum, a minimum or a saddle point:

- if all eigenvalues of $\mathbf{H}$ are positive: $\mathbf{w}_*$ is a local minimum,
- if all eigenvalues of $\mathbf{H}$ are negative: $\mathbf{w}_*$ is a local maximum,
- if $\mathbf{H}$ has both positive and negative eigenvalues: $\mathbf{w}_*$ is a saddle point.

2.3 Input normalization

Let us now consider a linear regression problem in which we are given a set of training examples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ and we model the data with a linear function:

$$\mathbf{f}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$$

For simplicity, we assume that our model does not have a bias term. We tune the parameters $\mathbf{w}$ by minimizing the mean squared error

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)^2$$

and suppose that we use gradient descent to minimize the loss function.

We know that the convergence of the gradient descent is determined by the properties of the Hessian matrix. Let us compute the Hessian matrix for our problem:

$$\mathbf{g} = \frac{2}{2N}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top\mathbf{x}_n\right)(-\mathbf{x}_n) = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top\mathbf{w} - \frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n$$
$$\mathbf{H} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top = \mathbf{C}_\mathbf{x}$$

We can see that the Hessian is equal to the second order moment of the data (which is equal to the covariance matrix of the inputs if inputs have zero mean).

We know from the convergence analysis for the gradient descent that the fastest convergence happens when the Hessian matrix has the condition number of 1. This can be achieved if we transform the inputs such that $\mathbf{C}_\mathbf{x} = \mathbf{I}$. This can be done by decorrelating the input components using principal component analysis (PCA):

$$\mathbf{x}_{\text{PCA}} = \mathbf{D}^{-1/2}\mathbf{E}^\top(\mathbf{x} - \boldsymbol{\mu})$$

where $\mathbf{EDE}^\top$ is the eigenvalue decomposition of the covariance matrix of $\mathbf{x}$. Here, we center the data by subtracting the mean, then we multiply the centered data by the matrix of the eigenvectors $\mathbf{E}^\top$, which gives the principal components of the data. Finally, we divide the principal components by the square root of the corresponding eigenvalues to obtain data that have unit variance when projected onto any unit-norm vector. Optionally, the dimensionality of the data can be reduced by dropping the principal components corresponding to the smallest eigenvalues. This preprocessing procedure is often called *whitening*.

Even though multilayer neural networks are nonlinear models, normalizing the inputs typically has positive effect on the convergence speed. The simplest transformation is to normalize the data to zero mean and unit variance. A more advanced procedure is to perform whitening of the data. In some applications (for example, image processing), we may want the whitened signals to be close to the original ones. This can be done by multiplying the scaled principal components by the matrix of eigenvectors:

$$\mathbf{x}_{\text{ZCA}} = \mathbf{ED}^{-1/2}\mathbf{E}^\top(\mathbf{x} - \boldsymbol{\mu})$$

This procedure is commonly known as ZCA.

2.4 Weight initialization

Initialization of the parameters of a neural network has a significant effect on the optimization procedure. With poor initialization, the optimization procedure can find a bad solution which results in poor model performance.

Let us consider the initialization of the weights of a linear fully connected layer. The layer has N_x inputs and N_y outputs and for simplicity, we assume that the layer does not have the bias terms.

$$y_i = \sum_{j=1}^{N_x} w_{ij} x_j$$

It makes sense to initialize weights with random values. For example, we can draw the initial values of the weights from some distribution $p(w)$ with zero mean $\langle w \rangle = 0$. Suppose that the inputs x_j are normalized to have zero mean and unit variance and they are also uncorrelated. Then, the variance of the output signals is

$$\operatorname{var} y_i = \sum_{j=1}^{N_x} w_{ij}^2 \operatorname{var} x_j = \sum_{j=1}^{N_x} w_{ij}^2$$

Its expectation under the distribution $p(w)$ of the initial values of the weight is then

$$\langle \operatorname{var} y_i \rangle = \sum_{j=1}^{N_x} \langle w_{ij}^2 \rangle = N_x \operatorname{var} w$$

where $\operatorname{var} w$ is the variance of the initial weight values.

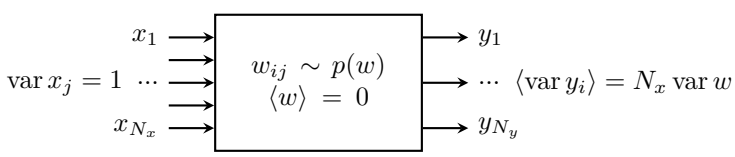


Figure 2.8: Change of the signal variance in the forward propagation through a linear layer.

We can see that if either N_x or $\operatorname{var} w$ is large, the variance of the output signal can get higher than the variance of the inputs. If we stack multiple linear layers on top of each other and use, for example, large $\operatorname{var} w$ for each layer, then the variance of the signals can quickly grows producing very large values at the output. This can cause problems for the optimization procedure. On the other hand, if the variance of the weights is chosen to be too small, then the signals in the network can quickly decay to zero, which is again a problem for the optimization.

To prevent explosions or decay of the signals, it is a good idea to keep the variance at a constant level: $\langle \operatorname{var} y_i \rangle = \langle \operatorname{var} x_j \rangle = 1$, which means that we should select the distribution $p(w)$ such that the variance $\operatorname{var} w$ of the weights is equal to

$$v_{\mathrm{f}} = \frac{1}{N_x}.$$

What about the variance of signals in the backpropagation phase? Let us assume that the inputs $\frac{\partial \mathcal{L}}{\partial y_i}$ of the linear layer in the backward phase are also

uncorrrelated and have unit variance: $\operatorname{var} \frac{\partial \mathcal{L}}{\partial y_i} = 1$. With similar arguments, the expected variance of the outputs is

$$\left\langle \operatorname{var} \frac{\partial \mathcal{L}}{\partial x_j} \right\rangle = N_y \operatorname{var} w$$

and if we want to keep the variance at a constant level, the initial distribution of the weights $p(w)$ should be such that the variance $\operatorname{var} w$ of the initial weight values is equal to

$$v_{\mathrm{b}} = \frac{1}{N_y}.$$

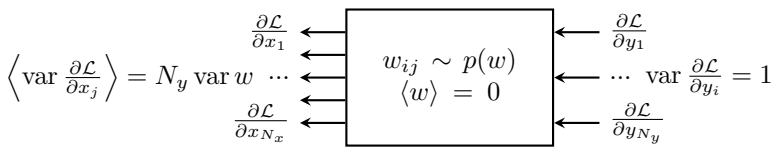


Figure 2.9: Change of the signal variance in the backward propagation through a linear layer.

To keep the balance between the forward and backward variances, Glorot and Bengio (2010) proposed to use weight distribution $p(w)$ such that $\operatorname{var} w$ is the harmonic mean of v_{a} and v_{b}:

$$\operatorname{var} w = \left(\frac{1/v_{\mathrm{f}} + 1/v_{\mathrm{b}}}{2} \right)^{-1} = \frac{2}{N_x + N_y}$$

If $p(w)$ is a uniform distribution $w_{ij} \sim \mathcal{U}[-\Delta, \Delta]$, then the variance of the weights is

$$\operatorname{var} w = \langle w_{ij}^2 \rangle = \int_{-\Delta}^{\Delta} w_{ij}^2 p(w_{ij}) dw_{ij} = \int_{-\Delta}^{\Delta} w_{ij}^2 \frac{1}{2\Delta} dw_{ij} = 2 \frac{\Delta^3}{3} \frac{1}{2\Delta} = \frac{\Delta^2}{3}$$

Then, we can initialize the weights as

$$w_{ij} \sim \mathcal{U} \left[-\frac{\sqrt{6}}{\sqrt{N_x + N_y}}, \frac{\sqrt{6}}{\sqrt{N_x + N_y}} \right]$$

which is perhaps the most popular intialization scheme called *Xavier's initialization.*

2.5 Mini-batch training (stochastic gradient descent)

The loss function that we minimize during training is usually a sum of terms which correspond to individual training examples. For example, for a regression problem with N training example, we minimize the mean-squared error loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right\|^2.$$

Training is often done using very large data sets, nowadays the data sets are so large that they cannot fit into the RAM memory. In the standard gradient descent, we need to go through all the training examples to compute the loss and its gradient before we update the model parameters. But does it make sense for large data sets?

Large data sets are often redundant, which means that different parts of the data set may contain similar information. The effect of that is that the gradients computed on different parts of the data are likely to be similar. Then, why to process the whole data set before updating the model parameters? This seems like a waste of computations.

The idea of mini-batch training is that we compute the gradient using only a small fraction of the training data (called a *mini-batch*):

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \approx \frac{1}{|\mathcal{B}_j|} \sum_{n \in \mathcal{B}_j} \frac{\partial}{\partial \boldsymbol{\theta}} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right\|^2$$

and use that gradient estimate $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ to update the model parameters. Here, $\mathcal{B}_j$ denotes one mini-batch. When using mini-batches, we never use the true gradient which should be computed using the whole data set. We always use its "noisy" estimate caused by the limited size of the mini-batch. Therefore, this method is often called *stochastic gradient descent*.

When doing mini-batch training, at every iteration we construct a mini-batch of examples sampled from the training set. It makes sense to use all training examples the same number of times and therefore the training examples are usually processed in an order. One loop of going through all the training examples is usually called a training *epoch*.

In every training epoch, it is a good idea to use a different split of the training set into mini-batches, otherwise gradient estimates are biased towards a particular mini-batch split. It is also important that mini-batches do not contain too similar examples. In classification problems, for example, mini-batches need to be balanced for classes.

The recent trend is to use as large batches as possible, for example, use the maximum size allowed by the GPU memory. There are two main reasons for that. First, using larger batch sizes reduces the amount of noise in the gradient estimates. Second, computing the gradient for multiple samples at the same time is computationally efficient. It typically requires matrix-matrix multiplications which are efficient, especially on GPUs.

Since in mini-batch training we always use noisy estimates of the gradient, the magnitude of the gradient estimate can be non-zero even when we are close to the optimum (see Fig. 2.10). One way to reduce this effect is to reduce the learning rate η_t towards the end of training. The simplest schedule is to decrease the learning rate after every n updates.

One trick to reduce the amount of noise in the weight estimates caused by stochastic gradient descent is to use exponential moving average of the model parameters. The parameters of the final model are computed as

$$\boldsymbol{\theta}'_t = \gamma \boldsymbol{\theta}'_{t-1} + (1 - \gamma) \boldsymbol{\theta}_t$$

where $\boldsymbol{\theta}_t$ are the parameters of the model trained with stochastic gradient descent and γ is a positive constant which is close to 1, for example, 0.9. By doing exponential moving, we essentially use the average of the weight values obtained in a moving window over the recent updates. This reduces the noise in the parameter values which is illustrated in Fig. 2.10.

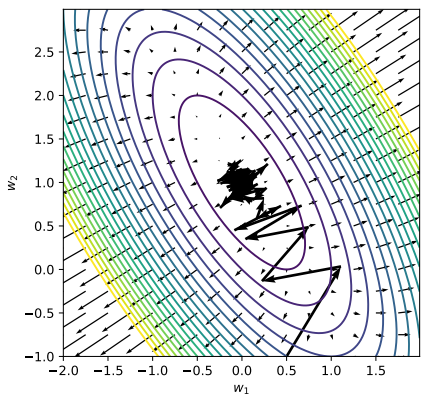


Figure 2.10: An optimization trajectory of stochastic gradient descent. The fluctuations are caused by using gradient estimates from a mini-batch instead of the true gradient.

2.6 Batch normalization

Now we seem to know everything that we need to optimize a neural network. We know how to compute the gradient efficiently using backpropagation, we know that it is better to normalize the inputs of a neural network to zero mean and unit variance, we also know how to initialize the weights so that the signals in the neural network neither vanish nor explode. However, does the fact that our model is a stack of multiple layers cause any problems for optimization?

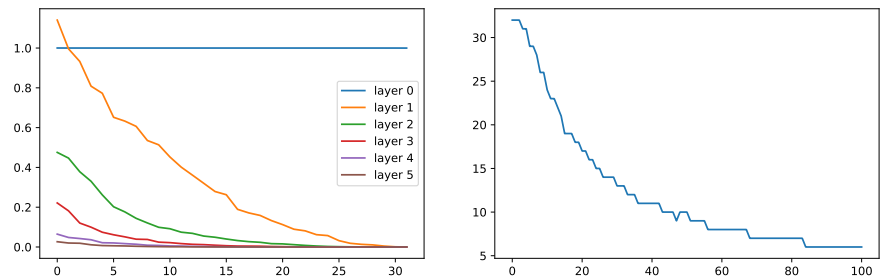
Suppose that we have a deep neural network with 32 inputs and 32 features in each hidden layer. For simplicity, let us assume that the network does not have nonlinearities: it is a stack of linear layers. We do the tricks that we mentioned before:

- whiten the inputs
- initialize the weights with Xavier's initialization

and we initialize bias terms with zeros. In Fig. 2.11a, we show the eigenvalues of the covariance matrices of the intermediate signals after each of the first five layers. We can see that the covariance matrix of the intermediate signals quickly becomes ill-conditioned: the dominant eigenvalues become much larger than the smaller ones. And this happens in spite of the fact that the inputs of the network are whitened, which means that all the eigenvalues at the input layer are equal to one.

Now suppose that we fix the first layers of the network and optimize only the last layers. The fact that the covariance matrix of the intermediate signals

is ill-conditioned means that the convergence of the gradient descent will be extremely slow.



(a) Eigenvalues of covariance matrices (b) Matrix rank as a function of layer

Figure 2.11: The structure of the intermediate signals in a stack of linear layers with 32 features in each layer. (a): The eigenvalues of the covariance matrices of the intermediate signals after each of the first five layers. (b): The rank of the matrix of the intermediate signals as a function of the layer.

In fact, the problem is even more severe. Fig. 2.11b shows the ranks of the matrices (as implemented in `torch.matrix_rank()`) containing the intermediate signals in the network as a function of the layer. We can see that the rank of the intermediate representations decays very quickly with the number of layers. This means that we effectively lose information on the way: some projections of the original data do not influence the output signal at all. As noted by Daneshmand et al. (2020), the rank collapse indicates that the direction of the output vector becomes independent of the actual input. Bjorck et al. (2018) report that a standard neural network initialized normally consistently predicts one specific class, irrespective of the input (see Fig. 2.14). This is also an indicator of the rank collapse.

Why does the rank collapse happen? The intuition is that the dominant eigenvectors of the weight matrices make the input vectors rotate more in the same direction. Recall the power iteration method which is a method to find the dominant eigenvector of a matrix $\mathbf{W}$:

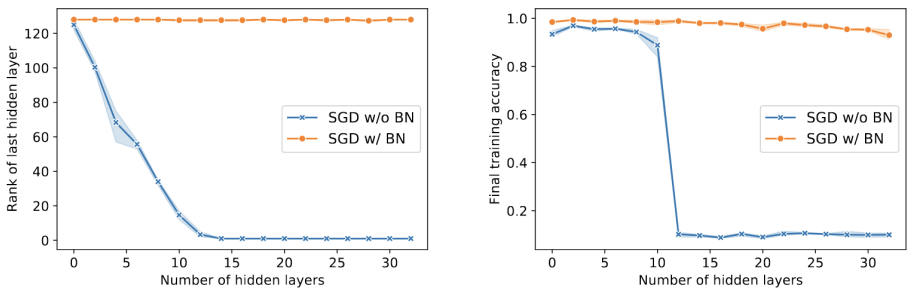
$$\mathbf{v} \leftarrow \frac{\mathbf{W}\mathbf{v}}{\|\mathbf{W}\mathbf{v}\|} .$$

We multiply a randomly initialized vector $\mathbf{v}$ by $\mathbf{W}$, normalize the resulting vector to unit length and use this vector in the next iteration. The power iteration method converges to the desired solution because the matrix multiplication rotates the input vector towards the direction of the dominant eigenvector.

In our deep linear network

$$\mathbf{y} = \mathbf{W}_n ... \mathbf{W}_3 \mathbf{W}_2 \mathbf{W}_1 \mathbf{x}$$

one layer $\mathbf{W}_1\mathbf{x}$ can be viewed as one iteration of the power method without the normalization step. Thus, even after the first layer, the intermediate signals $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x}$ become (slightly) correlated even if the inputs $\mathbf{x}$ have been whitened. When multiple layers are stacked together, the effect becomes very prominent: outputs $\mathbf{y}$ are more determined by the spectral structure of the weight matrices $\mathbf{W}_i$ rather then inputs $\mathbf{x}$. Some data projections simply become invisible in the outputs. Applying intermediate nonlinearities does not fix this problem.



(a) rank in the last hidden layer after random initialization

(b) training accuracy after 75 epochs

Figure 2.12: Effect of depth on rank and learning for an MLP network with 128 hidden units in each hidden layer and ReLU nonlinearities (Daneshmand et al., 2020). The network is trained on the Fashion-MNIST dataset.

The rank collapse problem has a severe negative effect on the training procedure. Fig. 2.12 presents the effect of depth on rank and learning for an MLP network with ReLU nonlinearities (Daneshmand et al., 2020). We can see that when the network has more than 12 layers, the rank collapses to one and the network does not train. Bjorck et al. (2018) report that a deep network with standard initialization can have very large gradient magnitudes (see Fig. 2.14a), which can cause divergence of the training procedure.

The rank collapse problem is diminished by the trick called *batch normalization*. The motivation behind batch normalization is the following: if input normalization has positive effect on training, it is probably a good idea to normalize the intermediate signals as well. The problem is that the intermediate signals change during training and we cannot perform normalization before the training. The solution is to normalize intermediate signals to zero mean and unit variance in *each training mini-batch*. This yields the batch normalization layer:

1. Compute the means and variances of the intermediate signals $\mathbf{x}$ from the current mini-batch $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$:

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}^{(i)} \qquad \boldsymbol{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}^{(i)} - \boldsymbol{\mu})^2 \,.$$

2. Normalize signals to zero mean and unit variance:

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}} \,.$$

3. Scale and shift the signals with trainable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$:

$$\mathbf{y} = \boldsymbol{\gamma} \odot \tilde{\mathbf{x}} + \boldsymbol{\beta} \,.$$

Daneshmand et al. (2020) show that batch normalization has a positive effect on the rank of the intermediate representations (see Fig. 2.13). For a linear model, the rank quickly drops with and without batch normalization. Without

batch normalization, the rank goes to one but with batch normalization the rank stabilizes at a larger value. For an MLP with ReLU activations, the rank almost does not drop at all.

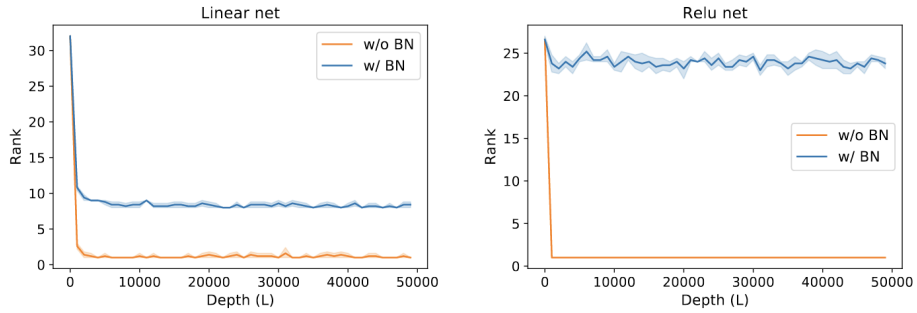


Figure 2.13: The rank of the last hidden layer's activation as a function of the total number of layers.

Due to fixing the rank collapse problem, batch normalization has positive effect on the magnitudes of the gradients inside the network. Bjorck et al. (2018) present the heat maps of the output gradients in the final classification layer after initialization in a deep neural network (see Fig. 2.14). The columns in these plots correspond to classes and the rows correspond to samples in the mini-batch. The unnormalized network consistently predicts one specific class (which corresponds to the very right column) and because of that the gradients are highly correlated. For a batch normalized network, the dependence upon the input is much larger and the gradients become less correlated.

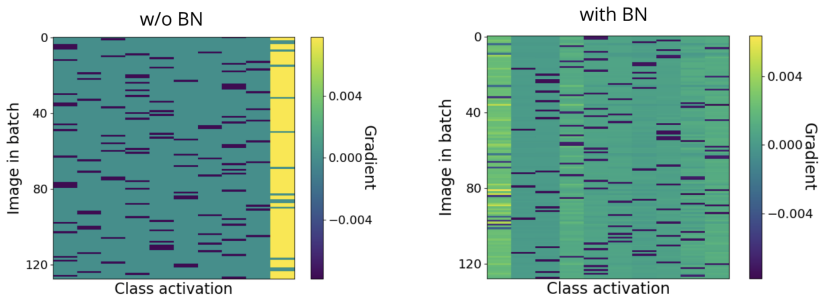


Figure 2.14: A heat map of the output gradients in the final classification layer after initialization (Bjorck et al., 2018). The columns correspond to classes and the rows to samples in the mini-batch.

Bjorck et al. (2018) also compute the histograms over the gradients at initialization for a midpoint layer of a deep neural network (see Fig. 2.15). For the unnormalized network (Fig. 2.15a), the gradients are distributed with heavy tails. The large magnitudes of the gradients can cause divergence of the gradient descent especially for large learning rates. Therefore, one has to use very small learning rates, which slows down training. For the normalized networks, the gradients are concentrated around the mean and the optimization procedure can converge even with larger learning rates. Thus, batch normalization enables training with larger learning rates, which is the cause for faster convergence and better generalization.

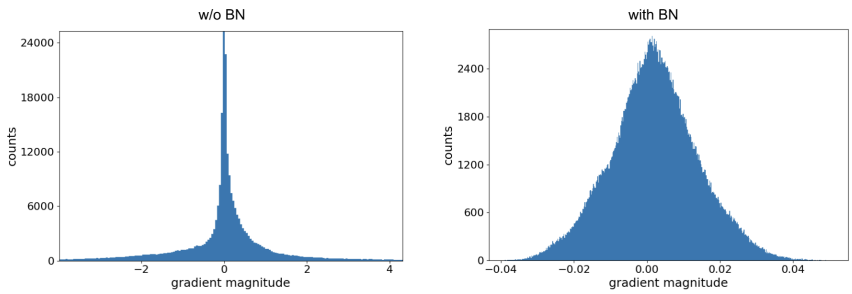


Figure 2.15: Histograms over the gradients at initialization for a midpoint layer of a deep neural network.

In a batch normalization layer, the mean and standard deviation are computed for each mini-batch. What statistics (mean and standard deviation) should be used then at test time for a trained network? Our test batch can contain a single example for which we cannot compute the statistics. The solution is to keep track of the batch statistics (mean and standard deviation) during training:

$$\overline{\boldsymbol{\mu}} \leftarrow (1-\alpha)\overline{\boldsymbol{\mu}} + \alpha \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}^{(i)}$$

$$\overline{\boldsymbol{\sigma}^2} \leftarrow (1-\alpha)\overline{\boldsymbol{\sigma}^2} + \alpha \frac{1}{N}\sum_{i=1}^{N} (\mathbf{x}^{(i)} - \overline{\boldsymbol{\mu}})^2$$

where α is the momentum hyperparameter. It is the running statistics $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$ that are used at test time. For that reason, the behavior of the network with batch normalization layers is different in the training and evaluation modes:

- In the training mode, we compute statistics from a mini-batch and update running statistics $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$.
- In the evaluation mode, we use running statistics $\overline{\boldsymbol{\mu}}$, $\overline{\boldsymbol{\sigma}^2}$ and keep them fixed.

A common mistake is not to set the network into the evaluation mode in which case the running statistics keep on changing.

In PyTorch, the model is switched to the training mode with `.train()` and into the evaluation mode with `.eval()`, like in the code below.

```
model = nn.Sequential(
    nn.Linear(1, 100),
    nn.BatchNorm1d(100),
    nn.ReLU(),
    nn.Linear(100, 1),
)

# Switch to training mode
model.train()
```

```
# train the model
...
# Switch to evaluation mode
model.eval()
# test the model
```

Remember that batch normalization introduces dependencies between samples in a mini-batch in the computational graph.

2.7 Improved optimization algorithms

The gradient descent algorithm is conceptually very simple but unfortunately it has problems. When we analyzed the convergence of the gradient descent, we saw that the convergence can be slow when the curvature of the objective function substantially varies in different directions. The problem is that the gradient magnitudes can be large in the directions in which we need to travel little and they can be small in the direction in which we need to travel much. This fact results in zigzaging of the optimization trajectory.

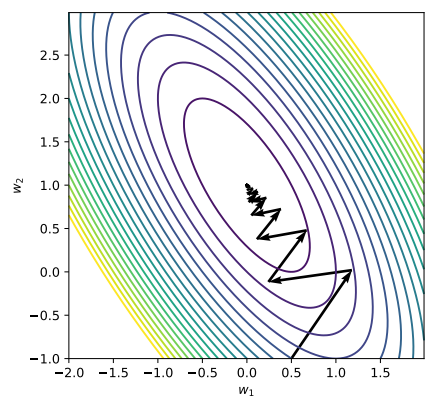


Figure 2.16: Gradient magnitudes can be large in the directions in which we need to travel little and they can be small in the direction in which we need to travel much. This fact results in zigzaging of the optimization trajectory.

We know that the information about the curvature is contained in the Hessian matrix and, in principle, we could try to use that information. This is done in *Newton's method*. In one iteration of this method, we find a quadratic approximation of the loss function, as given in (2.4), and find the minimum of the quadratic approximation. This results in the following update rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{H}_t^{-1}\mathbf{g}_t$$

This optimization method can be efficient but it is not practical for large neural networks which may contain millions of parameters. The problem is the high computational complexity of computing $\mathbf{H}_t^{-1}\mathbf{g}_t$ when the number of parameters is large.

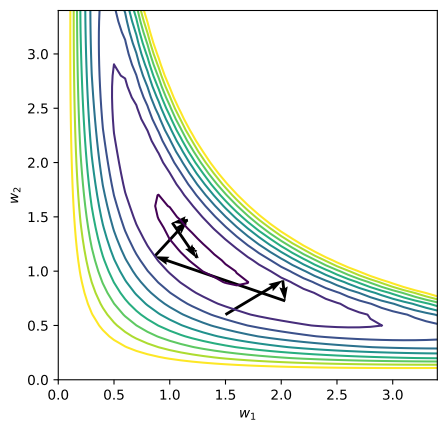


Figure 2.17: An optimization trajectory obtained with Newton's method.

2.7.1 Momentum method

A more practical method is called a *momentum method* (Polyak, 1964). The idea of the method is that

- we would like to move faster in directions with small but consistent gradients;
- we would like to move slower in directions with big but inconsistent gradients.

The method is implemented by aggregating negative gradients in the momentum:

$$\mathbf{m}_{t+1} = \alpha \mathbf{m}_t - \eta_t \mathbf{g}_t$$

and using the momentum to update the parameters:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{m}_{t+1}$$

The first equation can be viewed as applying exponential moving average to $\frac{\eta_t}{(1-\alpha)}\mathbf{g}_t$ which is the gradient scaled by the learning rate:

$$\mathbf{m}_{t+1} = \alpha \mathbf{m}_t - (1-\alpha)\frac{\eta_t}{(1-\alpha)}\mathbf{g}_t.$$

Parameter α is often chosen to be close to one, for example, 0.9.

Hinton (2012) gives a intuitive explanation of the momentum method. Imagine a ball moving on the error surface. The location of the ball represents the value of the parameters (w_1, w_2). At time $t = 0$, the ball follows the gradient. Once it gains velocity, it no longer does steepest descent: its momentum makes it keep going in the previous direction. The momentum damps oscillations in directions of high curvature (by combining gradients with opposite signs) and it builds up speed in directions with a gentle but consistent gradient. You can find the convergence analysis for the momentum method in (Goh, 2017).

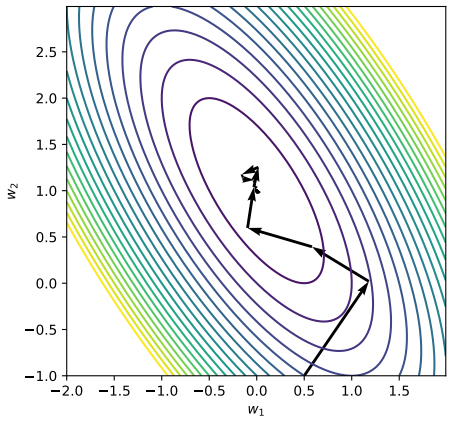


Figure 2.18: An optimization trajectory obtained with the momentum method.

2.7.2 Rprop

The algorithm called Rprop (Reidmiller and Brau, 1992) is the basis of modern optimizers used for tuning neural networks. The authors note that the magnitude of the gradient can be very different for different weights and can change during learning. This makes it hard to choose a single global learning rate. For that reason, Rprop uses only the sign of the gradient but ignores the gradient magnitude. The update rule is

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{\eta}_t \odot \frac{\mathbf{g}_t}{\sqrt{\mathbf{g}_t^2 + \epsilon}}$$

where $\mathbf{g}^2$ is a vector of squared partial derivatives and $\odot$, $\frac{\mathbf{a}}{\mathbf{b}}$ denote elementwise multiplication and division, respectively. The algorithm uses different learning rates for individual parameters, thus $\boldsymbol{\eta}_t$ is a vector in which each element corresponds to one of the parameters. The learning rates $\boldsymbol{\eta}_t$ are adapted to increase the step size for parameters that are consistently changed in the same direction:

- Increase the step size for a weight multiplicatively (e.g., $\times 1.2$) if the signs of its last two gradients agree. Do not exceed the maximum step size.
- Otherwise decrease the step size multiplicatively (e.g., $\times 0.5$).

The Rprop algorithm allows escaping quickly from plateaus with tiny gradients.

2.7.3 RMSProp

Rprop was designed for full-batch training and it does not work well for mini-batch training. Hinton (2012) gives an intuitive example. Consider a weight that gets a gradient of +0.1 on nine mini-batches and a gradient of -0.9 on the tenth mini-batch: We want this weight to stay roughly where it is because if we combine the gradients from the ten mini-batches, the aggregate gradient will be close to zero. However, Rprop would increment the weight nine times and

decrement it once by about the same amount (if we ignore the adaptation of the step sizes). So the weight would grow a lot.

Hinton (2012) proposed to modify Rprop by essentially using the exponential moving average of the squared gradients $\mathbf{g}_t^2$ in the denominator:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{v}_t + \epsilon}}$$
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t^2$$

This algorithm is called RMSprop. Suppose that we use $\beta = 0.9$ in the toy example, that we considered above. Then, for the tenth mini-batch we have $\mathbf{v}_t = 0.09$ and the effective step size is $\eta_t \frac{0.9}{\sqrt{0.09}} = 3\eta_t$, which means that we diminished the problem.

2.7.4 Adam

Adam (Kingma and Ba, 2014) is perhaps the most popular optimization algorithm. It can be viewed as RMSProp with the exponential moving average of the gradient in the numerator:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon}$$
$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$$
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$$

The estimates of the momenta $\mathbf{m}_t$, $\mathbf{v}_t$ start from zero, which creates a bias in favor of small values (which is less negligible at the beginning of training). The authors propose to correct this bias using

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{(1-\beta_1^t)}$$
$$\widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{(1-\beta_2^t)}$$

where β_1^t, β_2^t denote β_1, β_2 in the power of t. These momenta corrected for bias are used in the Adam update rule.

Adam is perhaps the most popular optimization algorithm among deep learning practitioners. Why does Adam work so well? One reason is that in Adam, the effective step size $|\Delta_t|$ is bounded. In the most common case:

$$|\Delta_t| = \left| \eta \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t}} \right| \approx \left| \eta \frac{E[g]}{\sqrt{E[g^2]}} \right| \leq \eta$$

because

$$E[g^2] = E[g]^2 + E[(g - E[g])^2].$$

Thus, we never take too big steps which can happen in the standard gradient descent because the gradient magnitudes can get large (this property is inherited

from Rprop). We go with the maximum speed (taking steps of size η) only if g is the same between updates (mini-batches), that is when the gradients are consistent.

Another reason is that Adam has a mechanism for automatic annealing of the learning rate. At convergence, when we start fluctuating around the optimum, $E[g] \approx 0$ and $E[g^2] > 0$, which means that the effective step size gets smaller.

Chapter 3

Convolutional neural networks

3.1 Motivation

In this chaper, we consider supervised learning problems in which inputs either have specific order such as one- or two-dimensional spatial structure. As an example, let us consider a classification task in which we need to classify images of handwritten digits from the MNIST dataset (see Fig. 3.1). The inputs of a classifier are images of 28×28 pixels. The task is to classify an image into 10 classes.

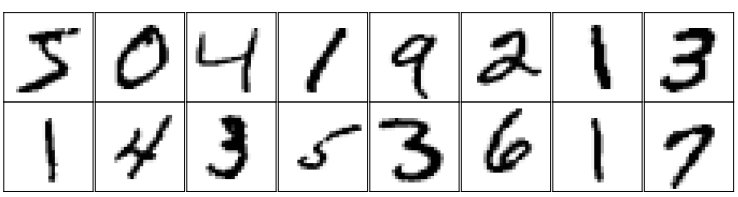


Figure 3.1: Example classification problem: images of handwritten digits from the MNIST dataset. Inputs $\mathbf{x}^{(n)}$ are images 28×28 pixels. Targets $y^{(n)}$: One of the 10 classes.

In this task, spatial information plays an important role. In Fig. 3.2, you can see the same images from Fig. 3.1 with the pixels shuffled in the same way for all the images. Even though the information is preserved in the shuffled

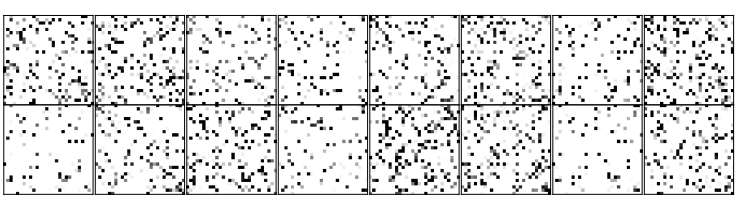


Figure 3.2: If we change the order of the pixels (in the same way for all images), the classification task becomes much harder for a human. This suggests that the classifier can benefit from using spatial information.

images, the digits become non-recognizable for a human. This example shows that the classifier should benefit from using spatial information.

Let us first try to solve our image classification problem using a multilayer perceptron network that we discussed in Section 1.3. We can flatten images (for example, by stacking the columns of the images into one vector) and feed the resulting vector into an MLP model. In Fig. 3.3, we can see a 3×3 image of digit 4 that is converted into a 9-dimensional vector and fed to an MLP. For this simple dataset, the MLP model will actually be able to solve the classification task on a reasonably good level.

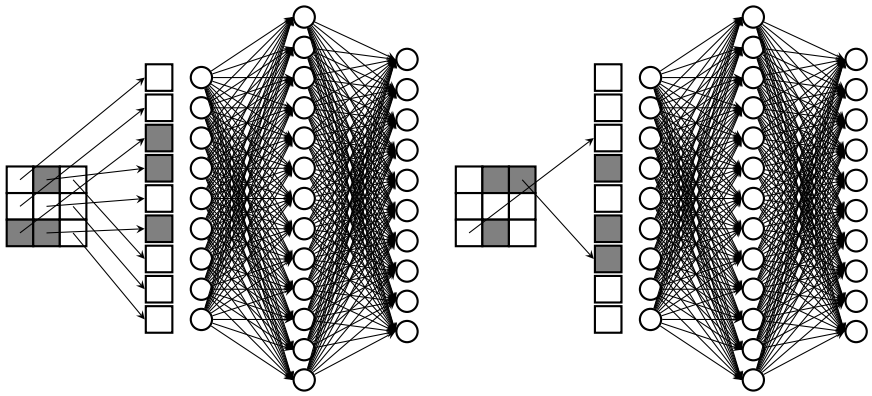


Figure 3.3: Solving an image classification task using a multilayer perceptron. Images are flattened and fed to the MLP. Right: When two pixels are swapped, we simply feed the pixels into different inputs of the MLP.

However, there are several problems with using the MLP model in this task. The first problem is that the model does not use the spatial structure (or order) of the inputs. Suppose that we shuffle the pixels of the input images. In Fig. 3.3, we swap two pixels in the top right corner and the bottom left corner. The swapped pixels will be fed to different inputs of the network, which means that after training the weights of the corresponding neurons are likely to be swapped too. However, swapping of the pixels does not affect the difficulty of the learning problem for the MLP. Training the MLP model using the original images will be as hard as using the shuffled inputs. This means that the MLP ignores the spatial structure and essentially tries to solve a more difficult classification problem.

The problem of ignoring the spatial structure can be demonstrated using the following example. Let us take an image of digit 0 which is resized to 8×8 pixels for demonstration purposes. And let us do small translations of the image by shifting the image one pixel to the left and to the right, up and down (see Fig. 3.4). We can easily see that the translated images contain the same digit in slightly different locations. However, the flattened versions of these five images look substantially different, as the activations happen in different positions. Feeding such translated images to an (untrained) MLP model will result at very different activations inside the network and the output of the network will change in an unpredictable way. During training, the MLP has to learn to be invariant to such transformations, which may require a considerable amount of data and training iterations.

The second problem of the MLP is a large number of parameters. Suppose that for solving the MNIST dataset, we use an MLP network with the following

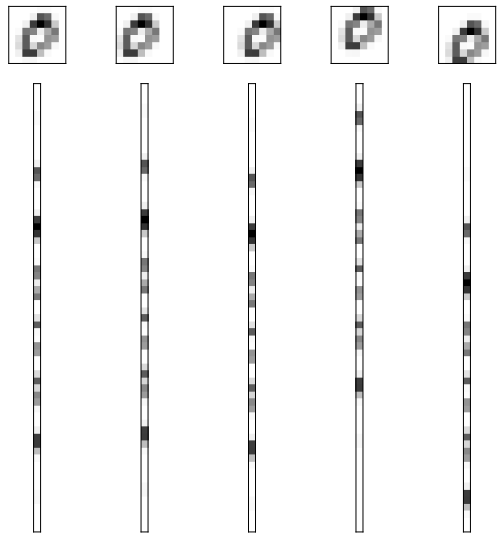


Figure 3.4: Small translations of the input image (for example, shifting the image one pixel to the left/right/top/bottom) result in significant changes of the MLP inputs, therefore the outputs of the MLP will change in an unpredictable way. The MLP has to learn to be invariant to such transformations, which may require a considerable amount of training.

architecture: two hidden layers, the first layer has 225 units, the second layer has 144 units (see Fig. 3.5). The output layer has 10 outputs because we have 10 classes in the MNIST classification problem. Let us count the number of

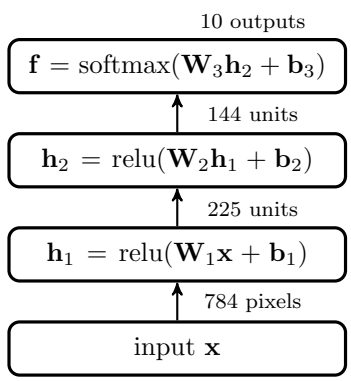


Figure 3.5: An example MLP network for solving the MNIST classification task.

parameters in our MLP network ignoring the bias terms $\mathbf{b}$ for simplicity. The first layer has $28 \times 28 \times 225$ parameters because the input contains 28×28 pixels and the output consists of 225 elements. The second layer has 225×144 parameters and the output layer contains 144×10 parameters. In total, we have

$$28 \times 28 \times 225 + 225 \times 144 + 144 \times 10 = 210240$$

more than 200 thousand parameters for our small classification problem. If we want to process images that contain millions of pixels, the number of parameters would be several orders of magnitude larger.

Therefore, we would like to design an alternative to the fully-connected layer that would address these problems. Our new layer should

- take into account the order of the inputs,
- have a smaller number of parameters compared to the fully-connected layer,
- change the outputs in a predictable way for simple transformations such as translation.

3.2 Convolutional layer

3.2.1 Inputs with 1d structure

Let us consider inputs with one-dimensional structure. For example, we process one-dimensional time series and the order of the inputs is determined by the order in which the observations arrive. Suppose that our time series has length 5 and we process it with a fully-connected layer with 5 outputs (see Fig. 3.6). The layer has $5 \times 5 = 25$ parameters if we neglect the bias terms.

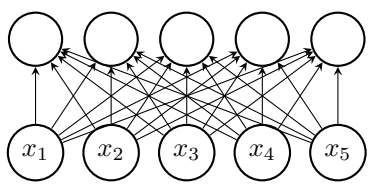


Figure 3.6: Processing of input with five ordered elements by a fully-connected layer.

We can reduce the number of parameters by using only local connections. We can connect each output only to inputs in a small neighborhood. For example, only to three inputs that are closest to the location of the output (see Fig. 3.7). Now the outputs also have an order because each output corresponds to a particular location. The new layer has 13 parameters instead of 25 parameters in the fully-connected layer.

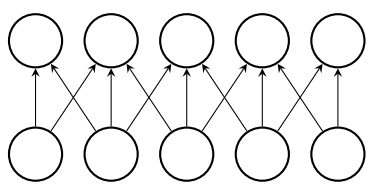


Figure 3.7: Processing of input with five ordered elements by a layer with local connections.

We can further reduce the number of parameters by using weight sharing. In Fig. 3.8, arrows with the same color represent shared weights. We can see that the inputs that come before the output neuron are processed by the same weight (represented by the red arrows). The inputs that come after the current position are processed by another weight (represented by the blue arrows). The black arrows represent the third parameter. By sharing weights in this way, we have reduced the number of parameters to 3.

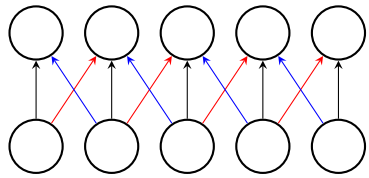


Figure 3.8: Processing of input with five ordered elements by a layer with local connections and shared weights.

Shared parameters bring us the following benefit. A pattern that appears at the beginning of the input sequence may activate a neuron at the beginning of the output sequence. When the same pattern appears at the end of the input sequence, it will activate another neuron that is close to the end of the output sequence. This happens due to parameter sharing. That means that our neurons act as detectors of features at different spatial locations.

The computations performed in such a layer are

$$y_i = \sum_{\Delta i=-1,0,1} w_{\Delta i} x_{i+\Delta i} + b$$

We compute a linear combination of the inputs in a moving window and add a bias term. This layer is called a (one-dimensional) *convolutional* layer because the computations are closely related to the convolution operation:

$$(w * x)[t] = \sum_a w[a]x[t-a].$$

Inputs and outputs of a convolutional layer usually contain multiple elements (usually called *channels*). Each output channel is produced with the same formula:

$$y_{i,o} = \sum_{\Delta i} \sum_c w_{\Delta i,o,c} x_{\Delta i,c} + b_o.$$

Note that we need to combine all the channels in the input and therefore the number of weights w is proportional to the number of the input and output channels. Weights $w_{\Delta i,o,c}$ are usually called *kernel*.

The inputs at the borders can be processed in two different ways. In the first approach, we compute only the outputs for which we have all the required elements in the inputs (see Fig. 3.9). As you can see, this results in the output that contains fewer elements than the input.

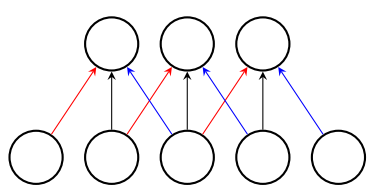


Figure 3.9: A one-dimensional convolutional layer without padding.

In many situations, however, we want to produce an output of the same size as the input. We can achieve that by assuming that the input has extra elements at the borders (see Fig. 3.10). Filling inputs with extra elements at the borders is typically called padding. Inputs are usually padded with zeros.

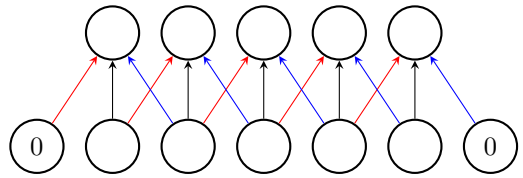


Figure 3.10: A one-dimensional convolutional layer with padding.

3.2.2 Inputs with 2d structure

The same ideas can be used for processing inputs with two-dimensional spatial structure such as images. We can use local connections such that an output is affected only by the inputs in its neighborhood (see Fig. 3.11). We can process different parts of the inputs with shared parameters. In Fig. 3.11, links with the same colors represent shared weights.

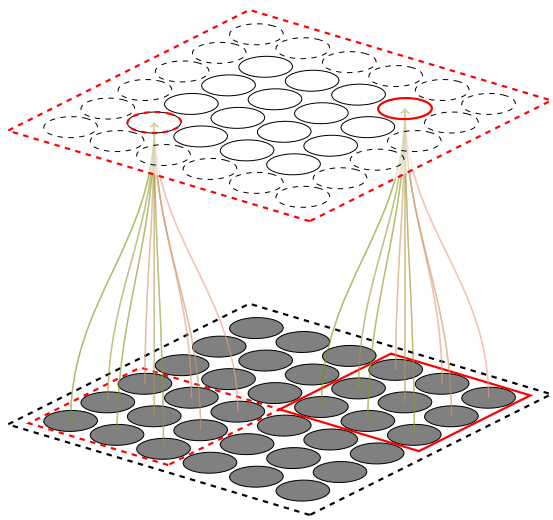


Figure 3.11: A two-dimensional convolutional layer. Links with same colors represent shared weights.

Let us take a closer look at the computations in a two-dimensional convolutional layer. We will use a simplified example in which the input image contains only one channel of binary (black-and-white) pixels (see Fig. 3.12). For simplicity, we assume that all the elements of the kernel are also binary. The output of the layer also contains only one channel and we do not use padding. To compute the output of the layer, we slide the filter across the entire input and compute the dot products between the input entries and the filter weights. In Fig. 3.12, you can see how the output map is populated. Note that the computations can be parallelized: we can compute the outputs at all locations in parallel. The parallelization can be very effective when the computations are done on graphical processing units.

The filter acts as at feature detector. Note that the filter in the example in Fig. 3.12 has the shape of a corner and the output has the maximum value of 3 at the position where this corner is present in the input image.

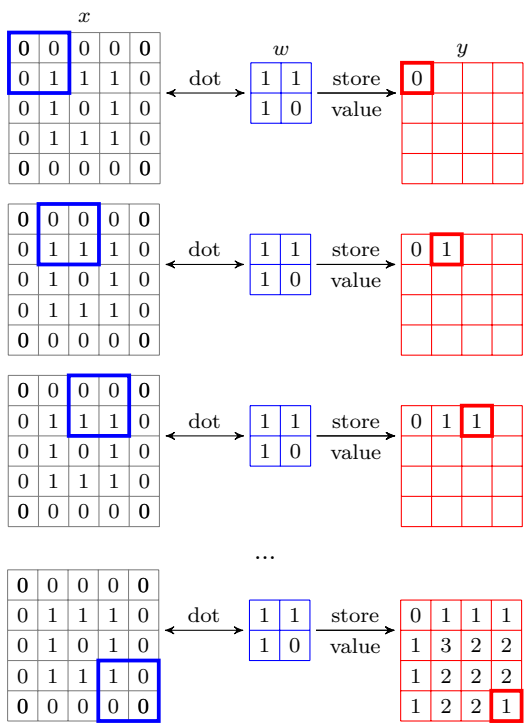


Figure 3.12: Illustration of computations in a two-dimensional convolutional layer. The kernel which contains the layer parameters is shown in the middle. We slide the filter across the entire input and compute dot products between input elements marked with blue squares and the kernel. The result is stored in the position marked with the red square.

In general, inputs of a convolutional layer may contain multiple channels. For example, color images are usually represented as two-dimensional maps with three (RGB) channels. We want to detect multiple features in the input images and therefore we typically use multiple filters. Therefore, the convolutional layer produces multiple channels o and we have a separate set of parameters w for each output channel.

$$y_{i,j,o} = \sum_{\Delta i} \sum_{\Delta j} \sum_{c} w_{\Delta i, \Delta j, o, c} x_{i+\Delta i, j+\Delta j, c} + b_o$$

Just like in multilayer perceptrons, the output of a convolutional layer is usually transformed by a nonlinear activation function, such as ReLU:

$$y'_{i,j,o} = \mathrm{relu}(y_{i,j,o}).$$

One filter learns to detect one feature in the input. Consider an example in Fig. 3.13. We process a grayscale image with two convolutional filters. The weights of the first filter change in the vertical direction. Therefore, it produces high activations at locations where the color changes in the direction from top to bottom. Similarly, the second filter detects color changes in the horizontal direction. We observe high values in the locations where the input resembles the kernel. Low values are observed in the locations where the color gradient is opposite compared to the color gradient in the kernel.

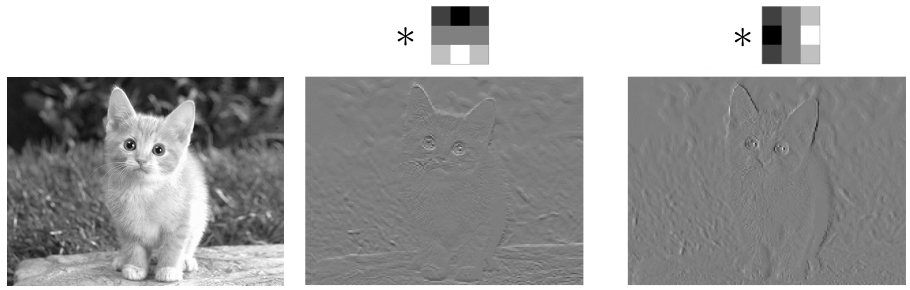


Figure 3.13: Each filter learns to detect one feature throughout the input.

3.2.3 2d convolutional layer in PyTorch

A two-dimensional convolutional layer in PyTorch is implemented in class

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

One needs to specify the number of input and output channels, the kernel size which is the number of elements in the kernel in both dimensions. Padding defines the number of padded elements at the borders. Parameter `stride` specifies how many pixels we skip when we slide over the input image. Skipping pixels results in reducing the dimensionality of the output image. This can be useful in practice because two neighboring windows of pixels in the input image often contain similar information. By skipping pixels with stride, we can reduce the resolution of the output without losing much information. An interactive visualization of the convolutional layer can be found on this web page. Fig. 3.14 explains the meaning of the parameters of the layer.

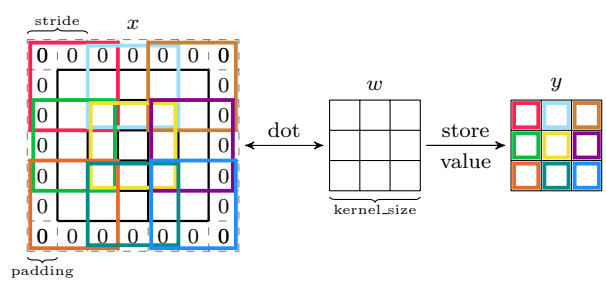


Figure 3.14: Visualization of the parameters of a two-dimensional convolutional layer. The input image is shown with a black square on the left hand side. The colors represent receptive fields of all pixels in the output map.

Note that the size n_o of the output map is generally different from the size n_i of the input. It can be computed using the following formula

$$n_o = \frac{1}{s}(n_i + 2p - k) + 1,$$

where k is the kernel size, p is the number of padded elements and s is the stride value.

Why do we need padding?

– With padding, the output of a convolutional layer can have the same size as the input, which makes it is easier to design deep networks. For

example, in the network called ResNet (Section 3.4.3), the output of a layer is summed with its input

$$\mathbf{x} + \mathrm{conv}(\mathbf{x}),$$

which means the input and output dimensions have to match.

– With padding, we can construct deeper networks. Without padding, the size would reduce quickly with adding new layers.

– Padding improves the performance by keeping information at the borders.

3.2.4 Translation equivariance

Let us now conduct the following experiment. We take an image and compute the output of a convolutional layer for this image (see the top row in Fig. 3.15). Then, we translate the image by shifting it one pixel to the right and feed the transformed image to the same convolutional layer. Note that the output of the layer will be a shifted version of the layer output for the original image (the bottom row in Fig. 3.15). This happens because during sliding over the transformed image, we will be getting the same values but in locations shifted by one pixel to the right compared to the original output. This property is known as *translation equivariance*.

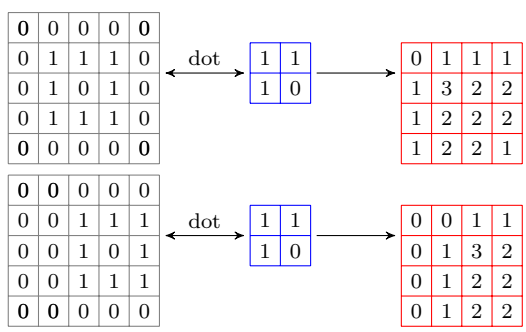


Figure 3.15: Shifting the input image by one pixel to the right changes the output in the same way: it is shifted by one pixel to the right.

Function f is *equivariant* with respect to transformation T if applying the transformation to input x and feeding it to the function is equivalent to applying the transformation T to the output $f(x)$:

$$f(T(x)) = T(f(x)).$$

Function f is *invariant* with respect to a transformation T if the result does not change when you apply the transformation to the input:

$$f(T(x)) = f(x).$$

Thus, the convolutional layer is equivariant to translations but not invariant.

3.3 Construction of a convolutional network

Let us build a convolutional neural network to solve the MNIST classification task. The input is an image with 28×28 pixels with 1 channel. As the first layer, we use a convolutional layer with a 5×5 kernel, 9 channels and padding with two pixels on each side. The output of the first layer contains 28×28 pixels

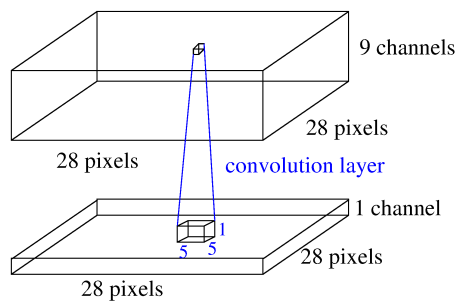


Figure 3.16: Constructing a convolutional network to solve the MNIST classification task. The first layer has a 5×5 kernel, 9 channels and padding with two pixels on each side.

with 9 channels. The number of parameters in the first layer (ignoring the bias terms) is

$$5 \times 5 \times 9 = 225\,.$$

Compare this number with the number of parameters in the first fully-connected layer of the MLP that we described in Section 3.1:

$$28 \times 28 \times 225 = 176400\,.$$

Let us now count the number of signals in the first layer:

$$28 \times 28 \times 9 = 7056\,.$$

Compare this number with 225 which was the number of signals in the first layer of the MLP from Section 3.1. We can see that the number of intermediate signals is much larger in the convolutional layer. To process such a high-dimensional signal, we need to perform a significant amount of computations in the rest of the network. In order to decrease the amount of computations, it makes sense to reduce the number of intermediate signals. We can do so by a *pooling layer*.

3.3.1 Pooling layer

A common way to reduce the number of intermediate signals is to keep the maximum values in a small window moving over the input. Consider as an example an output of a convolutional layer that is 4×4 in Fig. 3.17. We slide a 2×2 window over the image and select the maximum value inside that window. We can see that by using a filter of size 2×2 we discard 75 percent of the values.

The motivation behind pooling is that the outputs of a convolutional layer in two neighboring locations are likely to be similar because they have been computed in overlapping windows. By using max pooling, we take the most

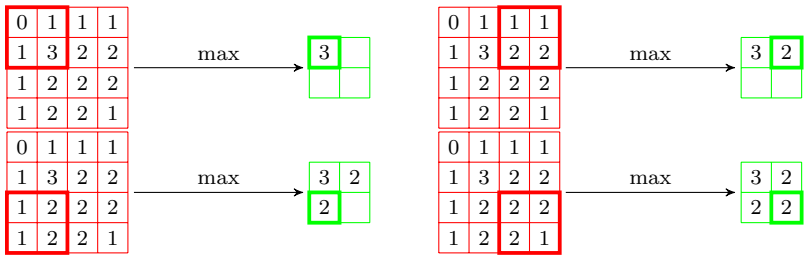


Figure 3.17: Populating the output of a max pooling layer.

pronounced presence of a given feature in the input map. Pooling also makes intermediate representations approximately invariant to small translations of the input.

Let us add a 2×2 pooling layer on top of the first convolutional layer from Fig. 3.16. The output of the pooling layer is a 14×14 map with 9 channels, the same number of channels as in the output of the first convolutional layer.

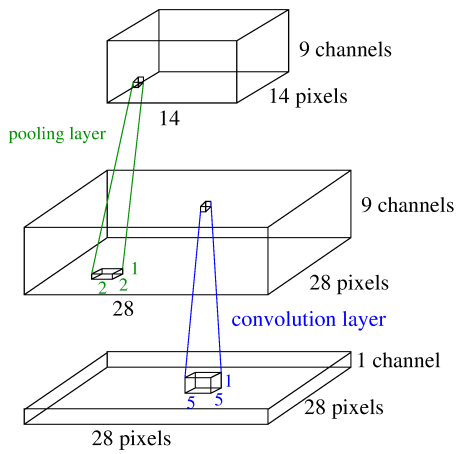


Figure 3.18: Constructing a convolutional network to solve the MNIST classification task. Adding a 2×2 pooling layer.

3.3.2 Stacking more layers

We continue by stacking another convolutional layer with a 5×5 kernel, padding of the inputs and 16 output channels (see Fig. 3.19) Note that each unit looks at all the channels of the previous layer. The resulting two-dimensional map is 14×14 pixels with 16 channels. Finally, we flatten the outputs of the last convolutional layer and feed them to a fully-connected layer with the softmax nonlinearity.

The network can be trained by a gradient-based optimization procedure, for example, Adam. The gradients are computed by backpropagation just as in the multilayer perceptron. The biggest difference is that we need to take into account parameter sharing inside the convolutional layers.

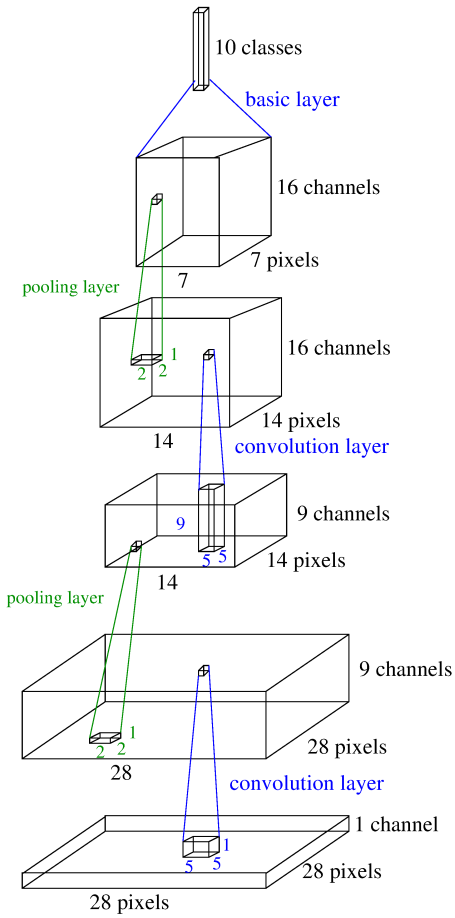


Figure 3.19: A convolutional network to solve the MNIST classification task.

3.3.3 Backpropagation through a convolutional layer

Let us take a look at the computations that happen in a convolutional layer

$$y_{i,j,o} = \sum_{\Delta i} \sum_{\Delta j} \sum_{c} w_{\Delta i, \Delta j, o, c} x_{i+\Delta i, j+\Delta j, c} + b_o$$

in the backpropagation phase. To compute the derivative with respect to parameters $w_{\Delta i, \Delta j, o, c}$, we need to compute a sum over all locations i and j (due to parameter sharing):

$$\frac{\partial \mathcal{L}}{\partial w_{\Delta i, \Delta j, o, c}} = \sum_{i} \sum_{j} \frac{\partial \mathcal{L}}{\partial y_{i,j,o}} x_{i+\Delta i, j+\Delta j, c}.$$

To compute the derivatives wrt the inputs of the layer, we need to combine

the partial derivatives wrt to the outputs with the layer parameters.

$$\frac{\partial \mathcal{L}}{\partial x_{i,j,c}} = \sum_{\Delta i} \sum_{\Delta j} \sum_{o} \frac{\partial \mathcal{L}}{\partial y_{i-\Delta i, j-\Delta j, o}} w_{\Delta i, \Delta j, o, c} \tag{3.1}$$

Note that the resolution of the outputs in the backpropagation phase increases. The operation that processes inputs $\frac{\partial \mathcal{L}}{\partial y_{i-\Delta i, j-\Delta j, o}}$ as written in (3.1) is often called *transposed convolution*.

$$\frac{\partial \mathcal{L}}{\partial x_{i,j,c}} = \sum_{\Delta i} \sum_{\Delta j} \sum_{o} \frac{\partial \mathcal{L}}{\partial y_{i-\Delta i, j-\Delta j, o}} w_{\Delta i, \Delta j, o, c}$$

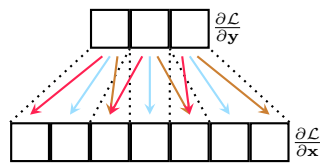


Figure 3.20: The operation performed during backpropagation through a convolutional layer is often called *transposed convolution*.

3.4 Popular convolutional networks

The architecture that resembles modern convolutional networks was first proposed by Fukushima (1980). The model was called neocognitron, it was a neural network architecture with multiple layers of local feature detectors and weight sharing (see Fig. 3.21).

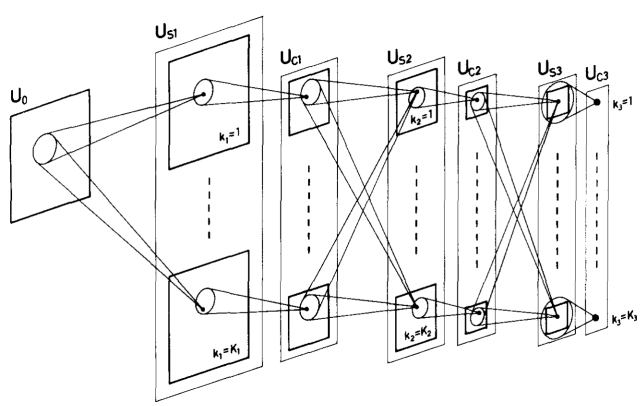


Figure 3.21: Fukushima's Neocognitron

The neocognitron architecture inspired many researchers. LeCun et al. (1998) used that architecture as inspiration for building the first convolutional networks to solve the task of handwritten character recognition. A time-delay neural network was similar to convolutional networks but applied to audio (Waibel et al., 1989).

Convolutional neural networks became hugely popular after the ImageNet competition in 2012. In this section, we will look at the architectures of three notable networks that were successful in the ImageNet competition: AlexNet, VGG network and ResNet.

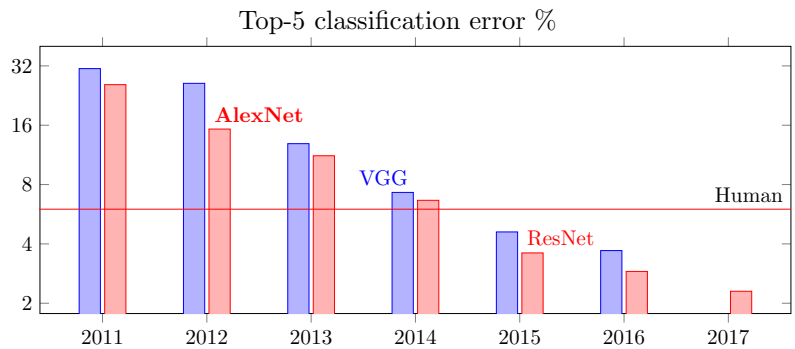


Figure 3.22: Top-5 classification error in the ImageNet competition.

3.4.1 AlexNet

The network (Krizhevsky, 2012) contains five convolutional layers and the first layer has pretty large filters 11×11. The network has three maximum pooling layers and three fully-connected layers at the end of the network. The network has ReLU nonlinearities after the convolutional layers. To reduce overfitting, the authors used a regularization technique called dropout (see Section 4.4.1).

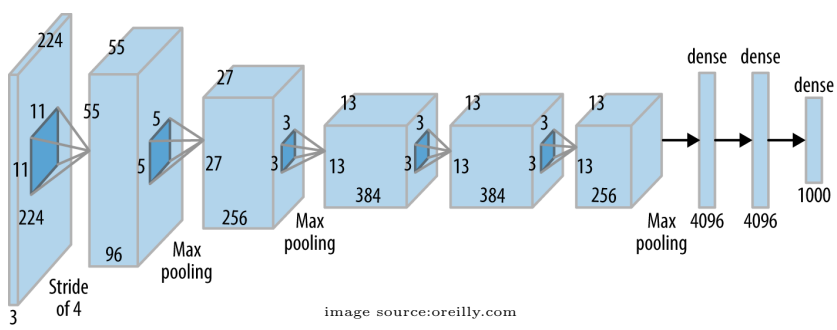


Figure 3.23: The architecture of AlexNet (Krizhevsky, 2012).

3.4.2 VGG network

Simonyan and Zisserman (2015) give the following motivation behind the VGG architecture. Suppose that we have a block that processes a two-dimensional map with c input channels to produce a two-dimensional map with the same number of channels c. If we implement this block as a single convolutional layer with 7×7 filters, we have $49c^2$ parameters (see Fig. 3.24).

Now suppose that we stack three convolutional layers with 3×3 filters (see the rhs plot in Fig. 3.24). As you can see, the value of a pixel in the third layer is affected by a window of size 7 in the input. That means that the receptive field of a neuron in the third layer is 7×7. However, the number of parameters in the block is $27c^2$, which is about half of the parameters of one convolutional layer with the same receptive field. Therefore, it makes to use smaller kernels but deeper networks.

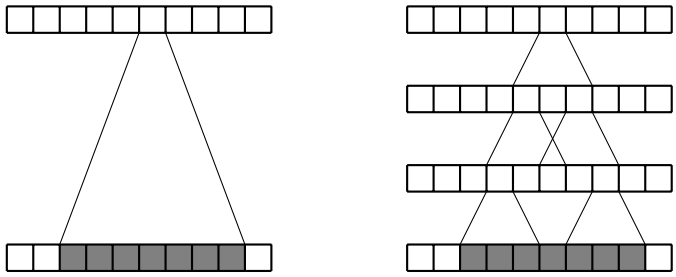


Figure 3.24: Motivation behind using small kernels: by using a stack of three layers with kernel size 3, we get a receptive field of 7 with a smaller number of parameters compared to a single layer.

Using deeper networks with small kernels has been the trend since 2012. In the VGG-19 network, there are 19 convolutional layers with 3×3 kernels (see Fig. 3.25).

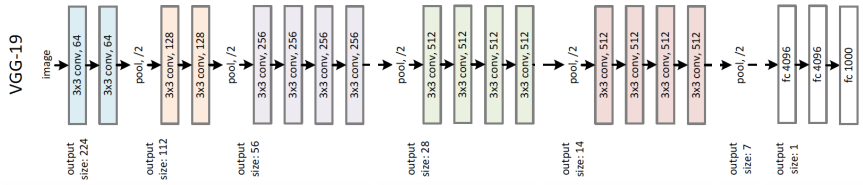


Figure 3.25: The architecture of the VGG-19 network.

3.4.3 ResNet

ResNet (He et al, 2016) is probably the most popular convolutional architecture. The authors noticed that training a convolutional network with a large number of layers is a difficult problem. In Fig. 3.26, you can see the learning curves for two convolutional networks with 20 and 56 layers. The performance of the deeper network is worse than of the more shallow one. This is somewhat counter-intuitive because the extra layers of the deeper network could simply learn an identity mapping if the shallow architecture is more preferable. However, this does not happen in practice.

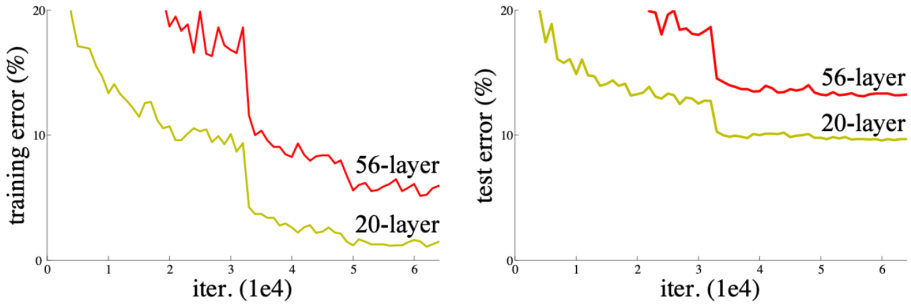


Figure 3.26: Learning curves for two convolutional networks without residual connections. Training of a convolutional network with more layers is a harder optimiation problem.

The authors proposed an architecture in which instead of learning function $f(\mathbf{x})$, layers learn function $\mathbf{x} + h(\mathbf{x})$. The motivation is that if an identity mapping is optimal for a layer, it is easier to push the residual $h(\mathbf{x})$ to zero than to

learn an identity mapping with $f(\mathbf{x})$. The proposed architecture contains residual connections that skip convolutional layers (see Fig. 3.27). The architecture is called *ResNet*.

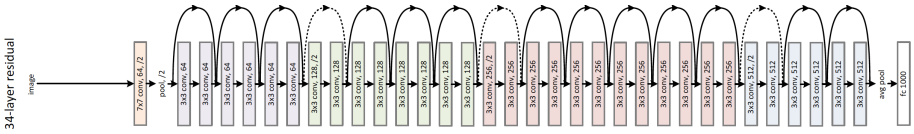
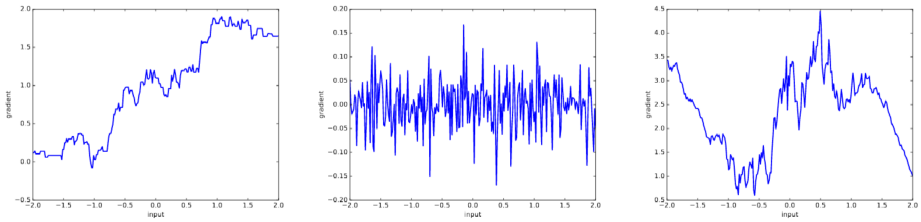


Figure 3.27: The architecture of ResNet (He et al, 2016).

Later, Balduzzi et al. (2017) gave a different perspective into why residual connections help training. They looked at a randomly initialized MLP network with scalar inputs and outputs $f : \mathbb{R} \to \mathbb{R}$. The network had 200 neurons and ReLU activations in each layer. They visualized the derivative of the output as a function of the input x (see Fig. 3.28). One can see that in a shallow network with a single layer the derivative changes "slowly" wrt to the input. On the contrary, the derivative changes rapidly in a 24-layer network. Note that



1-layer feedforward 24-layer feedforward 50-layer ResNet

Figure 3.28: Gradients $\frac{\partial f}{\partial x}(x)$ as a function of the input.

the sign of the gradient may change from positive to negative even for a small change of the input. The authors call this phenomenon *shattered gradients*.

Why does this matter? When we do backpropagation, we need to compute the partial derivatives of the output of a block wrt to its input. In a deep network without skip connections, these derivatives may change significantly even for a small perturbation of the input. The fact that the sign of the derivative changes means that instead of increasing the value of a weight, we can start decreasing the value just because of a slightly perturbed input. This causes problems for the optimization procedure. As we can see from Fig. 3.28, adding skip connections fixes this problem. The gradients start behaving similarly to the shallow network, which has a positive effect on optimization.

3.4.4 Batch normalization in convolutional networks

To improve convergence, the ResNet architecture contains batch normalization layers. The batch statistics in a batch norm convolutional layer are computed across all examples in a mini-batch and all pixels.

$$\boldsymbol{\mu} = \frac{1}{NWH} \sum_{n=1}^{N} \sum_{i=1}^{W} \sum_{j=1}^{H} \mathbf{x}_{ij}^{(n)} \qquad \boldsymbol{\sigma}^2 = \frac{1}{NWH} \sum_{n=1}^{N} \sum_{i=1}^{W} \sum_{j=1}^{H} (\mathbf{x}_{ij}^{(n)} - \boldsymbol{\mu})^2$$

The statistics are computed separately for each channel, therefore vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ have as many elements as there are channels in the input $\mathbf{x}$. The batch statistics are used to normalize the signals in each channel. After that, the signals are scaled and shifted with parameters γ_c and β_c:

$$y_{ijc}^{(n)} = \gamma_c \frac{z_{ijc}^{(n)} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c \,.$$

3.5 Applications of convolutional networks

Convolutional networks have been used for processing images, text data, speech, analyzing positions in board games and even for predicting protein folding. The main advantages of the convolutional networks are:

- They take into account the order of the inputs.
- They can process input sequences with varying lengths (due to parameter sharing).
- The computations can be effectively parallelized.

3.5.1 Processing speech with temporal convolutions

WaveNet (van den Oord et al., 2016) is an autoregressive model of speech that predicts the next value of the speech signal given a window of the previous values.

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1})$$

The model is implemented as a stack of one-dimensional convolutional layers.

One challenge in the speech domain is that one typically needs to model dependencies between very distant signals. The sampling rate of the CD quality audio is 44 kHz which means that one second of audio contains 44 thousand samples. If we want to model dependencies within one second with WaveNet, the receptive field of the last convolutional layer should be 44 thousand. To obtain such a large receptive field with standard convolutional layers, one would have to stack a huge number of them (see Fig. 3.29).

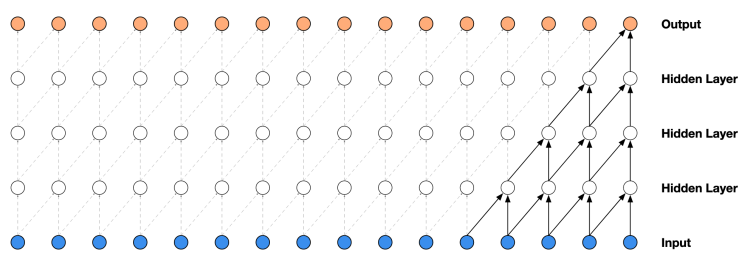


Figure 3.29: In a stack of standard convolutional layers, the receptive field grows slowly. The marked pixel has a receptive field of size 5 in a stack of four layers.

For that reason, WaveNet uses a stack of *dilated* convolutional layers whose architecture is shown in Fig. 3.30. Dilated convolutions allow fast growth of the

receptive field, which means that we can obtain the same size of receptive fields with fewer convolutional layers. WaveNet is still the state-of-the-art model in speech synthesis.

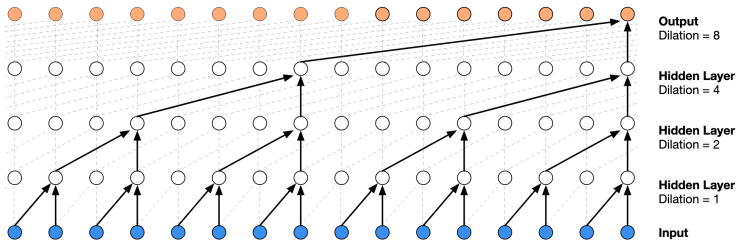
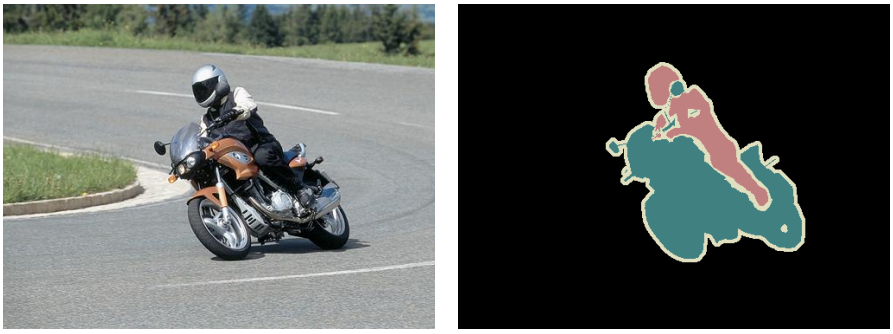


Figure 3.30: In a stack of dilated convolutional layers, the receptive field grows faster. The marked pixel has a receptive field of size 16 in a stack of four layers.

3.5.2 Convolutional models for semantic segmentation

Semantic segmentation is a popular application for convolutional neural networks. The task is to generate pixel-wise segmentations of images, that is to predict the class of the object visible in each pixel. This is a harder classification problem compared to producing one label for the whole image.



input image output segmentation map

Figure 3.31: The task of semantic segmentation.

U-net is a popular architecture for solving the semantic segmentation task (see Fig. 3.32). The model contains two paths: the contracting path from the high-resolution input to a low-resolution representation and the expansive path from the low-resolution representation to high-resolution outputs. The purpose of the contracting path is to extract high-level features, that is we want to detect high-level objects such as a human or a motorcycle in Fig. 3.31. The purpose of the expansive path is to make the classification decisions on the pixel level. The expansive path uses representations developed in the contracting path via skip connections. The representations from the skip connections are concatenated with the signals computed in the expansive path and fed as the input of the next layer.

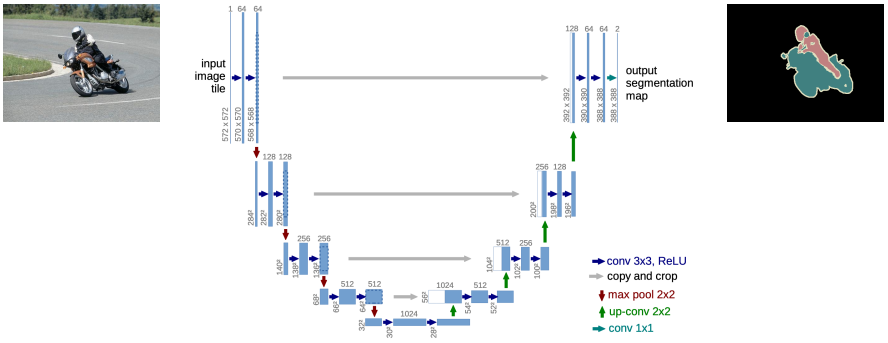


Figure 3.32: The architecture of the U-net.

3.5.3 Convolutional model for statistical machine translation

Convolutional networks have been also used in the task of statistical machine translation. The task is to translate a sentence in one language into a sentence in another language. Sentences can be seen as sequences of words and they can also be processed by a convolutional neural network. Gehring et al., (2017) used convolutional networks to encode the source sequence and to decode the developed representations into the output sequence (Fig. 3.33). We will have a closer look at this model in Chapter 6.

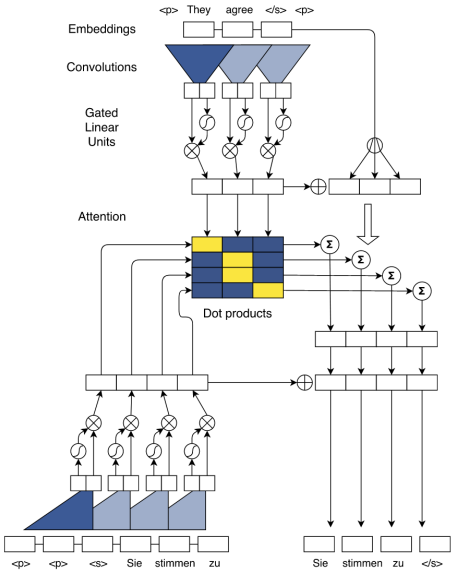


Figure 3.33: Convolutional model for statistical machine translation Gehring et al., (2017).

3.5.4 Convolutional networks in reinforcement learning

In reinforcement learning tasks, convolutional networks are used to process sensory inputs with two-dimensional structure. For example, the famous AlphaZero

algorithm (Silver et al, 2017), that achieves superhuman performance in the games of Go, chess and shogi, uses convolutional neural networks to analyze game positions. Deep convolutional networks are used to compute the probability of the next move and the probability that the player wins the game from the current position.

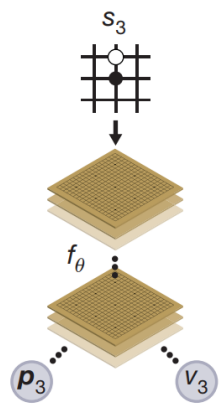


Figure 3.34: A convolutional network to analyze the board position in the AlphaZero algorithm (Silver et al, 2017).

3.5.5 Protein folding

Convolutional neural networks were also used to solve the protein folding problem. Proteins are comprised of chains of amino acids and the information about the sequence of amino acids is contained in DNA. The protein folding problem is predicting how the chains of amino acids will fold into the 3D structure of a protein.

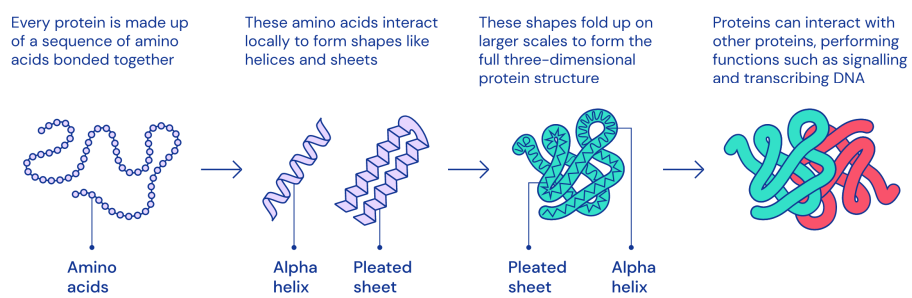


Figure 3.35: Complex 3D shapes emerge from a string of amino acids. The image is from Deepmind's blog post.

The standard for assessing techniques in the protein prediction problem is a biennial global competition called the Critical Assessment of protein Structure Prediction (CASP) which runs since 1994. In 2020, DeepMind's model called AlphaFold 2 (Senior et al., 2020) has achieved "unprecedented progress in the ability of computational methods to predict protein structure" (see Fig. 3.36).

The backbone of the AlphaFold 2 model is a convolutional neural network (Fig. 3.37). The input of the model is a sequence of amino acids. Each sequence

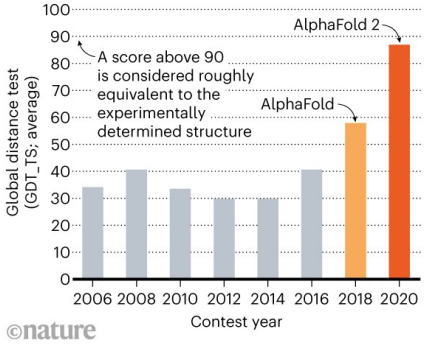


Figure 3.36: Progress in the Critical Assessment of protein Structure Prediction (CASP).

is represented as a two-dimensional matrix in which each element corresponds to one pair of amino acids. The features (channels) of each pixel are produced using an external model. The outputs are the distances between the C_β atoms of pairs of amino acid residues of a protein. It is interesting that even though the input has one-dimensional structure, it is processed by a convolutional network with two-dimensional convolutions and the results are outstanding.

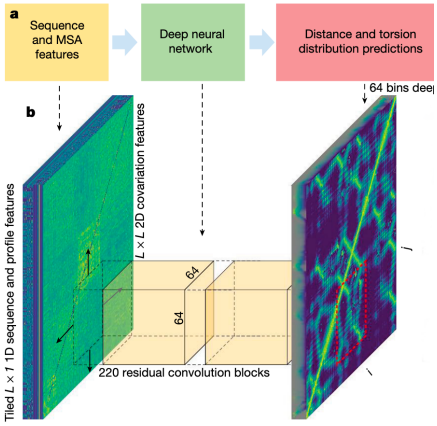


Figure 3.37: A convolutional network is the backbone of the AlphaFold 2 model for protein structure prediction.

Chapter 4

Regularization

4.1 Overfitting

Overfitting happens when a trained model follows training data too closely but it fails to fit test data reliably. In Fig. 4.1, we show an example of fitting a polynomial function of order 12 to the data points shown with the blue circles. The model trained on these data are shown with the red line. We can see that the model fits the training data very well but it generalizes poorly between the training examples.

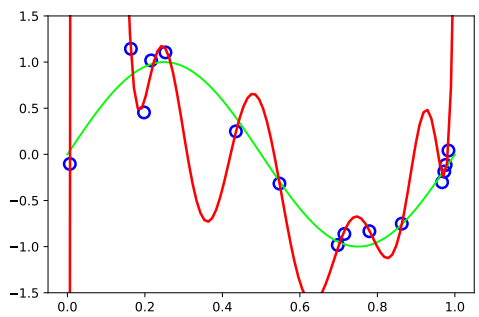


Figure 4.1: Example of overfitting with a polynomial model of order $M = 12$. The training samples are shown with the blue circles. The red line is the fitted model. The green line is the correct function.

The conventional wisdom says that overfitting happens when "the model is too flexible for the amount of training data". The flexibility of the model is traditionally assumed to be connected to the number of model parameters. This is a quote from Wikipedia which says that "an overfitted model is a statistical model that contains more parameters than can be justified by the data". There is a well-known rule of thumb for the logistic regression model: to keep the risk of overfitting low, the number of examples should be ten times larger than the number of parameters.

The common way to detect overfitting is to use a validation set to evaluate

the performance of a trained model. In Fig. 4.2, the black circles represent the validation set. We can measure that the error on that validation set is much larger that the error on the training set. This is a sign of overfitting.

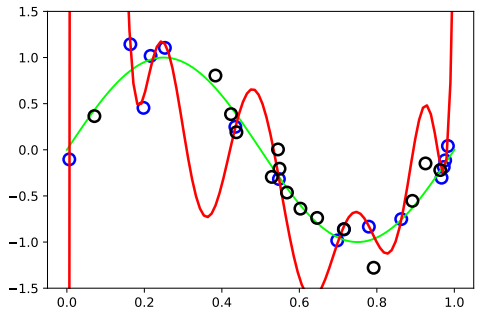


Figure 4.2: Overfitting can be detected by using a validation set (black dots).

Regularization is any modification to the learning algorithm that is done to reduce the overfitting problem. As we discussed in Chapter 1, neural networks are universal approximators, which means that they are very flexible models and they can also overfit. In Fig. 4.3, we see an MLP model fitted to the same training data as in Fig. 4.1. We can clearly see that the model overfits.

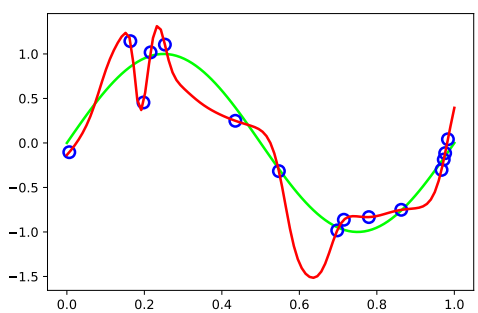


Figure 4.3: Fitting a toy data set with an MLP network. The red line is the fitted model. The green line is the correct function.

In this chapter, we study common methods to prevent overfitting of neural networks. They are:

1. limiting model capacity (by reducing network size, weight decay, parameter sharing)

2. early stopping

3. ensemble methods (dropout, Bayesian neural networks)

4. data augmentation (noise injection, transformations, adversarial training).

4.2 Limit model capacity

4.2.1 Reducing the model size

Recall the conventional wisdom: overfitting is likely to happen when a model contains more parameters than can be justified by the data. Then the obvious solution is to reduce the number of parameters. For example, we can vary the number of neurons, the number of layers and find the architecture that generalizes best in our learning problem. The advantage of this approach is that it is conceptually easy. The disadvantage is that other regularization methods often give better results.

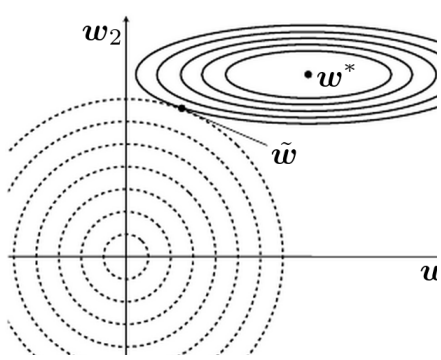
4.2.2 L_2 regularization

Another easy way to reduce overfitting is a penalty term that penalizes large values of the model parameters to add to the loss function (Tikhonov, 1943).

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \Omega(\mathbf{w})\,.$$
$$\Omega(\mathbf{w}) = \frac{\alpha}{2}\left\|\mathbf{w}\right\|^2 = \frac{\alpha}{2}\sum_i w_i^2$$

where α is a hyperparameter that controls the strength of the regularization. The penalty term $\Omega(\mathbf{w})$ is simply a sum of the squared values of the model parameters.

This regularization method is known as Tikhonov regularization or L_2 regularization. It pushes the solution towards zero. In Fig. 4.4, we see example contour plots of the unregularized loss $\mathcal{L}$ and the contour plots of the penalty term. The solution that minimizes the original loss is located here. The solution that minimizes the penalty term is located at $\mathbf{w}^*$. The regularized solution is found somewhere in between the unregularized solution and the origin $\mathbf{w} = 0$. The regularized solution is a compromise between $\mathbf{w}^*$ and $\mathbf{w} = 0$



w^* unregularized solution
$\tilde{w}$ regularized solution

Figure 4.4: L_2 regularization pushes the solution towards zero.

L_2 regularization is often called *weight decay*. For example, the Adam optimizer in PyTorch has the option of using L_2 regularization.

```
torch.optim.Adam(params, lr, betas, eps, weight_decay)
```

The strength of the regularization is controlled by the parameter `weight_decay` which sets the value of the hyperparameter α.

However, using term *weight decay* for L_2-regularization may cause confusion. In the early paper by Hanson and Pratt (1988), weight decay was implemented by the following update rule

$$\mathbf{w}_{t+1} = (1-\lambda)\mathbf{w}_t - \eta\nabla\mathcal{L} \tag{4.1}$$

in which the weights $\mathbf{w}$ decay exponentially with factor $(1-\lambda)$. We see that in order to compute the updated values of the weights, we multiply the previous values of the weights by factor $(1-\lambda)$ which is smaller than 1. That means that the contribution of the weight $\mathbf{w}_t$ will decay exponentially when the training progresses.

It is easy to see that weight decay is equivalent to L_2 regularization for standard stochastic gradient descent.

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\alpha}{2}\|\mathbf{w}\|^2$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\nabla\mathcal{L}_{\text{reg}} = \mathbf{w}_t - \eta\nabla\mathcal{L} - \eta\alpha\mathbf{w}_t = (1-\eta\alpha)\mathbf{w}_t - \eta\nabla\mathcal{L}$$

If we set $\eta\alpha = \lambda$, we get the same update rule as in (4.1).

However, for algorithms like Adam, weight decay, as defined by (4.1), is not equivalent to L_2 regularization. This was noted by Loshchilov and Hutter (2017). They proposed a regularized version of Adam which tries to follow the early-days definition of weight decay. That optimization algorithm is available in PyTorch as `torch.optim.AdamW`.

It seems that many students are confused about this topic and the confusion seems to be caused by a popular blog post which says "that the way weight decay is implemented in Adam in every library seems to be wrong". They call that implementation "old, broken Adam". What is implemented in PyTorch under the name of Adam with weight decay is simply Adam with L_2 regularization. It is a valid regularization technique and it is not broken in any way. AdamW implements a different regularization technique. It may work better or worse but both are valid regularization methods.

Why does L_2 regularization reduce overfitting? One intuition is that smaller weights usually produce smoother functions, functions with smaller magnitudes of derivatives. A more formal analysis of weight decay can be done for a linear regression problem (Goodfellow et al., 2016). Consider a linear regression problem in which we minimize the mean-squared error loss and let us drop the bias terms for simplicity:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top\mathbf{x}_n\right)^2 + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}.$$

We can find the minimum of the regularized loss by computing the gradient

and equating it to zero:

$$\nabla_{\mathbf{w}}\mathcal{L} = \frac{2}{2N}\sum_{n=1}^{N}\left(y_n - \mathbf{w}^\top\mathbf{x}_n\right)(-\mathbf{x}_n) + \frac{\alpha}{2}\mathbf{w} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top\mathbf{w} - \frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n + \frac{\alpha}{2}\mathbf{w}$$
$$= \left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top + \alpha\mathbf{I}\right)\mathbf{w} - \frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n = 0\,,$$

which yields

$$\mathbf{w} = \left(\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^\top + \alpha\mathbf{I}\right)^{-1}\left(\frac{1}{N}\sum_{n=1}^{N}y_n\mathbf{x}_n\right).$$

Let us take a closer look at this expression. We can see that the first term is the sample covariance matrix plus an additional term which is a diagonal matrix with the same diagonal elements α. That means that we simply add α to the diagonal elements of the sample covariance matrix.

Therefore, L_2 regularization causes the learning algorithm to "perceive" the input as having higher variance. This makes the weights shrink (become smaller). The regularization effect is larger for the weight values determined by the minor components of the data (those components that have small variance). Although this analysis was done for a linear model, it gives us some intuition why this regularization method works.

4.3 Early stopping

When we apply early stopping, we monitor the performance of the model on the validation set during training. Typical learning curves look like the ones shown in Fig. 4.5. First, both the training and validation errors decrease. Then, at some point, the training error continues to decrease while the validation error starts increasing. This is a sign of overlearning.

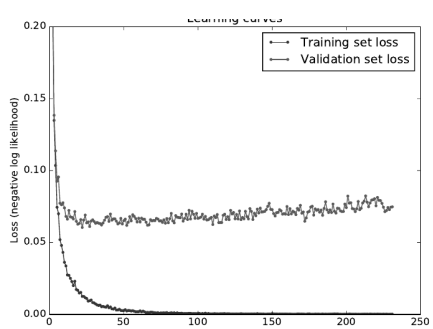


Figure 4.5: Typical learning curves during training.

The idea of early stopping is to stop training when the validation performance starts deteriorating. By doing that, we keeps the solution close to the

initialization. Note that when other regularization techniques are used, the deterioration of performance may never start and this technique does not help.

Goodfellow et al. (2016) give an intuition why early stopping works. They perform analysis for a quadratic loss function with optimal solution $\mathbf{w}_*$

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}(\mathbf{w}_*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_*)^\top \mathbf{A}(\mathbf{w} - \mathbf{w}_*).$$

Suppose that we minimize the loss using gradient descent. For this analysis, it is convenient to use the same transformation of the coordinate system that we did in Section 2.2. We compute the eigenvalue decomposition of matrix $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ and transform the coordinate system as in (2.2):

$$\mathbf{z} = \mathbf{Q}^\top(\mathbf{w} - \mathbf{w}_*). \tag{4.2}$$

The gradient descent update rule in the new coordinate system is given by (2.3):

$$\mathbf{z}_t = (\mathbf{I} - \eta\mathbf{\Lambda})\mathbf{z}_{t-1}$$

which can be written as

$$\mathbf{z}_t = (\mathbf{I} - \eta\mathbf{\Lambda})\mathbf{z}_{t-1} = (\mathbf{I} - \eta\mathbf{\Lambda})^2\mathbf{z}_{t-2} = ... = (\mathbf{I} - \eta\mathbf{\Lambda})^t\mathbf{z}_0$$

Now using (4.2) yields

$$\mathbf{Q}^\top(\mathbf{w}_t - \mathbf{w}_*) = (\mathbf{I} - \eta\mathbf{\Lambda})^t\mathbf{Q}^\top(\mathbf{w}_0 - \mathbf{w}_*)$$

If we assume that we start with $\mathbf{w}_0 = 0$, we get

$$\begin{aligned}\mathbf{Q}^\top(\mathbf{w}_t - \mathbf{w}_*) &= -(\mathbf{I} - \eta\mathbf{\Lambda})^t\mathbf{Q}^\top\mathbf{w}_* \\ \mathbf{Q}^\top\mathbf{w}_t &= \mathbf{Q}^\top\mathbf{w}_* - (\mathbf{I} - \eta\mathbf{\Lambda})^t\mathbf{Q}^\top\mathbf{w}_* = [\mathbf{I} - (\mathbf{I} - \eta\mathbf{\Lambda})^t]\mathbf{Q}^\top\mathbf{w}_*\,.\end{aligned} \tag{4.3}$$

Now consider minimizing the same loss with a weight decay penalty.

$$\mathcal{L}_\alpha(\mathbf{w}) = \mathcal{L}(\mathbf{w}_*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_*)^\top\mathbf{A}(\mathbf{w} - \mathbf{w}_*) + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}$$

We can find the optimal solution $\tilde{\mathbf{w}}$ by taking the gradient and equating it to zero.

$$\nabla\mathcal{L}_\alpha = \mathbf{A}(\mathbf{w} - \mathbf{w}_*) + \alpha\mathbf{w} = 0$$

The optimal solution $\tilde{\mathbf{w}}$ is given by

$$\tilde{\mathbf{w}} = (\mathbf{A} + \alpha\mathbf{I})^{-1}\mathbf{A}\mathbf{w}_*$$

In the coordinate system rotated by $\mathbf{Q}^\top$, the solution is given by

$$\begin{aligned}\mathbf{Q}^\top\tilde{\mathbf{w}} &= \mathbf{Q}^\top(\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top + \alpha\mathbf{I})^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top\mathbf{w}_* \\ &= \mathbf{Q}^\top\left[\mathbf{Q}(\mathbf{\Lambda} + \alpha\mathbf{I})\mathbf{Q}^\top\right]^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}\mathbf{w}_* \\ &= (\mathbf{\Lambda} + \alpha\mathbf{I})^{-1}\mathbf{\Lambda}\mathbf{Q}^\top\mathbf{w}_*\end{aligned} \tag{4.4}$$

Now let us compare the solutions found with the L_2 regularization (4.4) and with early stopping after iteration t (4.3):

$$\begin{aligned} L_2 \text{ regularization:} \quad & \mathbf{Q}^\top \tilde{\mathbf{w}} = (\mathbf{\Lambda} + \alpha \mathbf{I})^{-1} \mathbf{\Lambda} \mathbf{Q}^\top \mathbf{w}_* \\ \text{early stopping:} \quad & \mathbf{Q}^\top \mathbf{w}_t = [\mathbf{I} - (\mathbf{I} - \eta \mathbf{\Lambda})^t] \mathbf{Q}^\top \mathbf{w}_* \end{aligned}$$

If the hyperparameters η, α and t are chosen such that

$$(\mathbf{\Lambda} + \alpha \mathbf{I})^{-1} \mathbf{\Lambda} = [\mathbf{I} - (\mathbf{I} - \eta \mathbf{\Lambda})^t]$$

then L_2 regularization and early stopping can be seen as equivalent.

This analysis suggests that if we start with small values of the weights, the effect of early stopping is similar to the effect of the L_2 regularization. We find a solution that is somewhere between the origin where we started from and the unregularized solution. This is illustrated in Fig. 4.6.

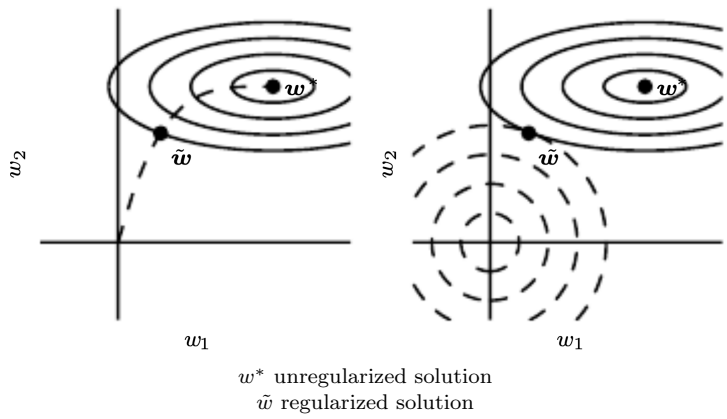


Figure 4.6: Left: In early stopping, the solution is between the initialization $\mathbf{w} = 0$ and the unregularized solution. Right: A similar effect is achieved with L_2 regularization.

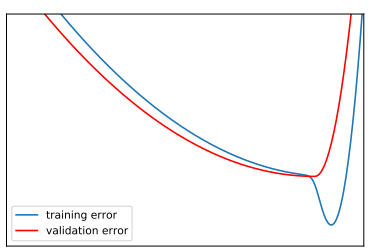


Figure 4.7: Early stopping stops training before we go to a narrow hole in which the model may generalize poorly.

For nonlinear optimization problems, the intuition why early stopping helps can be given by Fig. 4.7. At the beginning of training, we start moving in the areas of the parameter space where the gradients are small and small changes of the parameters cause small changes of both training and validation error. When we get closer to the optimum, we may start going into a narrow hole of the training loss where small changes of the weights can cause large changes of the validation loss. In early stopping, we stop training before we start going into that narrow hole.

4.4 Ensemble methods

The idea of ensemble methods is to train several models and take the average of their outputs. This approach is also known as *bagging* or *model averaging*. There are several ways how one can create an ensemble of models. For example, we can

- vary the types of models and training algorithms
- we can vary the hyperparameters of a single model
- we can vary data, for example, dropping some examples or dimensions
- we can vary model initializations.

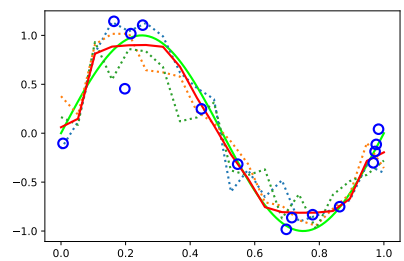


Figure 4.8: The red line shows a prediction made by an ensemble of models. The dotted lines represent predictions by individual ensemble members.

4.4.1 Dropout

Dropout is perhaps the most popular ensemble method used with neural networks (Hinton et al., 2012). The idea of dropout is that during training we randomly delete neurons with some probability p. Deletion means that we set the output of a neuron to zero. This is illustrated in Fig. 4.9.

Why is dropout an ensemble method? Because each model configuration with a particular set of dropped neurons can be seen as one member of an ensemble of models. The ensemble members have shared weights because we drop neurons in the same neural network. If the network contains N neurons, the ensemble contains 2^N models.

Fig. 4.10 illustrates that the dropout does not let the neural network learn to rely on particular features. During training, we randomly drop neurons that detect certain details in the input. For example, we may drop features that detect the curve at the bottom and the bar at the top but the network should use the remaining features to compute the prediction. In case we drop important features, an individual ensemble member may be uncertain about its prediction. But the average prediction computed by the ensemble may still be correct. The model trained with dropout is more robust to noise in the data.

At training time, neurons are dropped at random and therefore the output of the network is stochastic. At test time, we often want to get a deterministic

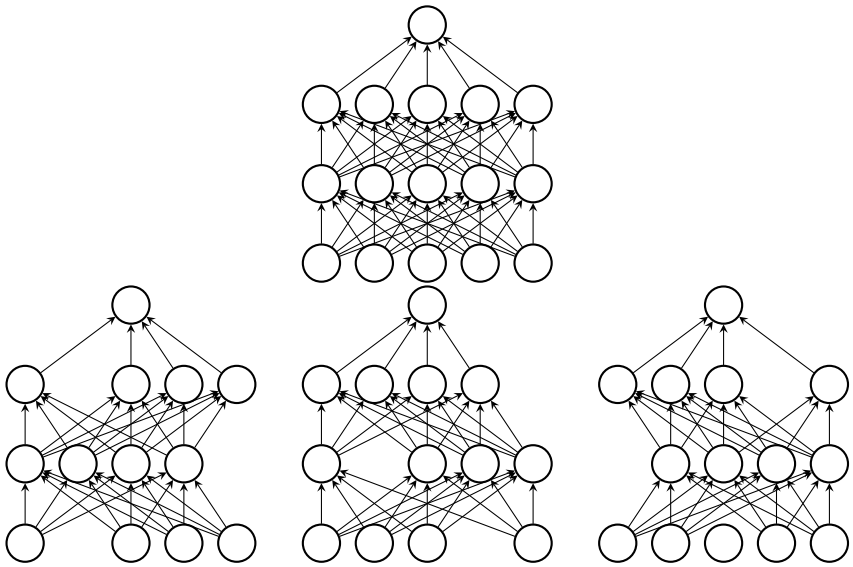


Figure 4.9: Above: A neural network with fully-connected layers. Below: Instances of the same network when some of the neurons are dropped out.

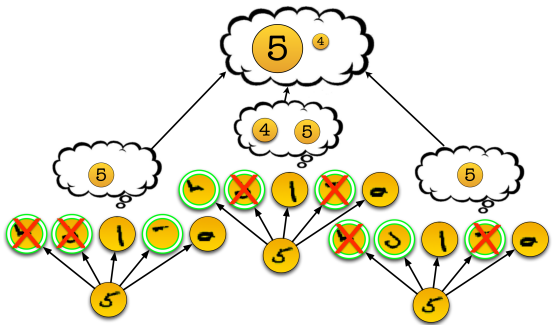


Figure 4.10: Dropout as bagging.

output. However, if we do not drop neurons at test time, the neurons after the dropout layer would receive more activation in their inputs than during training. This has a negative effect on the model performance.

We can fix this problem with the following trick. If a signal x is dropped with probability p, then the expected value after the dropout is

$$\mathbb{E}[\text{dropout}(x)] = (1-p)x\,.$$

If we divide the output of the dropout layer by $(1-p)$ at *training time*, we get

$$\mathbb{E}\left[\frac{1}{1-p}\text{dropout}(x)\right] = x\,,$$

which means that the expected value of the signal after the dropout is the same as without the dropout. This is how it is implemented in the PyTorch dropout layer:

- at training time: replace inputs with 0 with probability p and divide by $(1-p)$,
- at time time: pass the inputs without modification.

Therefore, if your network has a dropout layer, you should remember to switch between the training and evaluation modes as shown in the piece of code below.

```
model = nn.Sequential(
    nn.Linear(1, 100),
    nn.Tanh(),
    nn.Dropout(0.02),
    ...
)

# Switch to training mode
model.train()
# training the model
...
# Switch to evaluation mode
model.eval()
# test the model
```

Dropout is usually placed after the activation function. If the activation function is ReLU, then placing it before or after the nonlinearity is equivalent.

4.4.2 Probabilistic treatment: Bayesian neural networks

Another way of creating an ensemble of neural networks is implemented in *Bayesian neural networks*. This idea was popularized by David MacKay (1992). In the Bayesian methodology, one should always consider all possible models and combine them according to how well they explain the data. The predictive distribution for output $\mathbf{y}$ given input $\mathbf{x}$ and data D should be constructed by combining predictions made by individual models $p(\mathbf{y} \mid \mathbf{x}, \mathcal{M}_i)$ weighted by the probability of the model given the data:

$$p(\mathbf{y} \mid \mathbf{x}, D) = \sum_i p(\mathbf{y} \mid \mathbf{x}, \mathcal{M}_i) p(\mathcal{M}_i \mid D)$$

Probability $p(\mathcal{M}_i \mid D)$ is usually called *model evidence*.

We can build an ensemble of models around a single neural network architecture, just like we did with dropout. Each ensemble member is described by different values of the network parameters $\mathbf{w}$ and we can combine predictions produced by the ensemble by computing the following integral:

$$p(\mathbf{y} \mid \mathbf{x}, D) = \int p(\mathbf{y} \mid \mathbf{x}, \mathbf{w}) p(\mathbf{w} \mid D) d\mathbf{w}$$

where $p(\mathbf{w} \mid D)$ is a distribution over the model parameters $\mathbf{w}$. We weigh the individual predictions by the probability of an ensemble member given the data.

In order to compute this integral, we need to evaluate the posterior distribution $p(\mathbf{w} \mid D)$ of the model parameters given the training data. We can do

that using the Bayes rule

$$p(\mathbf{w} \mid D) = \frac{p(D|\mathbf{w})p(\mathbf{w})}{p(D)}.$$

Here, we have the prior distribution $p(\mathbf{w})$ over the weights, the likelihood $p(D|\mathbf{w})$ and a normalization constant $p(D)$ that does not depend on the model parameters. Unfortunately, the distribution $p(\mathbf{w} \mid D)$ does not have an analytical form for nonlinear models such as neural networks and therefore we need to approximate this distribution somehow. We can use different strategies to approximate $p(\mathbf{w} \mid D)$:

- maximum a posteriori estimation (point estimates of $\mathbf{w}$)
- variational approximation
- sampling

Next we briefly discuss some of these techniques.

4.4.3 Weight decay as Bayesian prior

The simplest approach to approximate $p(\mathbf{w} \mid D)$ is to use point estimates: maximum a posteriori values. Of course, this would not really create an ensemble of models because we would simply select a single model with the highest probability. To find the maximum a posteriori estimate, we can maximize the logarithm of the posterior probability

$$\mathcal{F}(\mathbf{w}) = \log p(\mathbf{w} \mid D) = \log p(D \mid \mathbf{w}) + \log p(\mathbf{w}) - \log p(D)$$

or minimize the negative of that:

$$\mathcal{L}(\mathbf{w}) = -\log p(D \mid \mathbf{w}) - \log p(\mathbf{w}) \tag{4.5}$$

We can drop the term $\log p(D)$ because it is constant wrt the model parameters.

Recall that the mean-squared error loss can be seen as minus log-likelihood of a Gaussian probabilistic model (see Section 2.1.2):

$$\begin{aligned} -\log p(D \mid \mathbf{w}) &= -\log \prod_{n=1}^{N} \mathcal{N}(\mathbf{y}^{(n)} \mid \mathbf{f}(\mathbf{x}^{(n)}, \mathbf{w}), \sigma^2 \mathbf{I}) \\ &= \frac{\beta}{N} \sum_{n=1}^{N} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \mathbf{w}) \right\|^2 + \text{const} \end{aligned} \tag{4.6}$$

Now if we assume the Gaussian prior $p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$, we get

$$-\log p(\mathbf{w}) = \frac{\alpha}{2} \|\mathbf{w}\|^2 + \text{const} \tag{4.7}$$

Inserting (4.6) and (4.7) into (4.5) gives

$$\mathcal{L}(\mathbf{w}) = -\log p(D \mid \mathbf{w}) - \log p(\mathbf{w}) = \frac{\beta}{N} \sum_{n=1}^{N} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \mathbf{w}) \right\|^2 + \frac{\alpha}{2} \|\mathbf{w}\|^2 + \text{const}$$

This shows that L_2 regularization is equivalent to maximum a posteriori estimation in a probabilistic model with the Gaussian prior.

4.4.4 Variational approximation of the posterior distribution

Another approach is to approximate the posterior distribution $p(\mathbf{w} \mid D)$ using variational approximation, which is illustrated in Fig. 4.11. The true posterior distribution over the weights typically has a very complex form, as shown with the blue line. It usually has many modes two of which are shown in the figure. A popular way to approximate this complex posterior distribution is to use a simpler distribution $q(\mathbf{w} \mid \boldsymbol{\theta})$ that has an analytical form and to minimize the distance between that approximating distribution $q(\mathbf{w} \mid \boldsymbol{\theta})$ and the true distribution $p(\mathbf{w} \mid D)$. The approximate distribution is shown with the red line in Fig. 4.11.

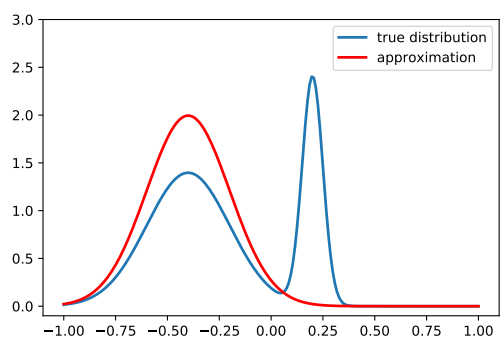


Figure 4.11: Approximation of a true posterior distribution (blue line) with a simpler distribution (red line).

It is common to use a fully factorial Gaussian distribution as the approximating distribution:

$$q(\mathbf{w} \mid \boldsymbol{\theta}) = \prod_i \mathcal{N}(w_i \mid \mu_i, \sigma_i^2).$$

In this case, each parameter w_i is described by two values: mean μ_i and standard deviation σ_i. Note that when we train a neural network normally, each parameter is described by a single value, the value of the parameter. Parameters $\boldsymbol{\theta} = \{\mu_i, \sigma_i\}$ are the parameters of the variational approximation. Those parameters are often called *variational parameters*. We tune those parameters during training.

The metric that is used to measure the distance between the true distribution $p(\mathbf{w} \mid D)$ and the approximating distribution $q(\mathbf{w})$ is often chosen to be the

Kullback-Leibler (KL) divergence:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathrm{KL}[q(\mathbf{w} \mid \boldsymbol{\theta}) || p(\mathbf{w} \mid D)] = \int q(\mathbf{w} \mid \boldsymbol{\theta}) \log \frac{q(\mathbf{w} \mid \boldsymbol{\theta})}{p(D \mid \mathbf{w}) p(\mathbf{w})} dw$$
$$= -\underbrace{E_{q(\mathbf{w}|\boldsymbol{\theta})}[\log p(D \mid \mathbf{w})]}_{\text{fit to data}} + \underbrace{\mathrm{KL}[q(\mathbf{w} \mid \boldsymbol{\theta}) || p(\mathbf{w})]}_{\text{regularization term}}.$$

For this learning problem, the KL divergence can be written as a sum of two terms. The first term measures how well the model ensemble fits the data. The second term works as a regulrization term that tries to keep the distribution of the weights close to the prior distribution. The regularization term can be computed analytically, while the first term has to be approximated, most commonly by Monte Carlo sampling (see, e.g., Blundell et al., 2015).

4.4.5 Sampling approach: Stein variational gradient descent

Another notable algorithm for variational inference in probabilistic models is called *Stein variational gradient descent* Liu and Wang (2016). It is a general method for doing variational inference and it is also applicable to training of neural networks.

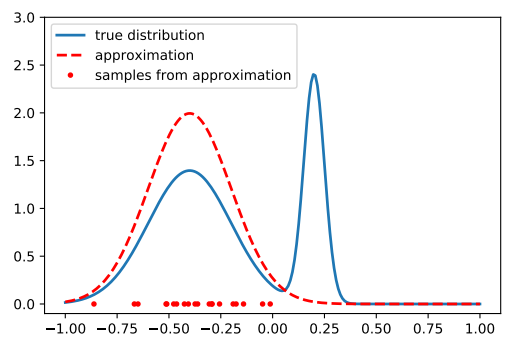


Figure 4.12: In Stein variational gradient descent, the true distribution (blue line) is represented by sample (red dots) from an approximating distribution (red dashed line).

The basic idea is that we want to describe the posterior distribution by samples drawn from an approximation of that posterior distribution. In Fig. 4.12, the approximation of the true posterior distribution is shown with the dashed red line and the samples from that approximation are shown with the red dots. It is those samples that define the members of our model ensemble. Each sample defines one neural network with particular values of the model parameters.

In this approach, we do not postulate the form of the posterior approximation $q(\mathbf{w})$. Therefore, we cannot compute the distance metric between the approximation and the true posterior distribution. However, we can use samples $\mathbf{w}_k$ from the approximate posterior to compute the gradient of the distance between the two distributions. We can use that gradient to update samples $\mathbf{w}_k$ so as to decrease the distance between the approximation and the true posterior

distribution. The update rule of the Stein variational gradient descent is

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \eta \frac{1}{K} \sum_{k'=1}^{K} \underbrace{k(\mathbf{w}_{k'}, \mathbf{w}_k) \nabla_{\mathbf{w}_{k'}} \left[\log p(\mathbf{w}_{k'}) + \log p(D \mid \mathbf{w}_{k'})\right]}_{\text{smoothed gradient}} + \underbrace{\nabla_{\mathbf{w}_j} k(\mathbf{w}_{k'}, \mathbf{w}_k)}_{\text{repulsive force}}$$

where $k(\mathbf{w}, \mathbf{w}')$ is a kernel function, for example,

$$k(\mathbf{w}, \mathbf{w}') = \exp\left(-\frac{1}{h} \|\mathbf{w} - \mathbf{w}'\|^2\right). \tag{4.8}$$

The update rule shows how samples $\mathbf{w}_k$ are updated in each training iteration.

Note that we use a kernel function k in the update rule. If we assume that k is a delta function, then all ensemble members are trained with standard gradient descent to minimize

$$-\log p(\mathbf{w}_{k'}) - \log p(D \mid \mathbf{w}_{k'})$$

where $-\log p(\mathbf{w}_{k'})$ can be viewed as a regularization term. If we use a kernel like (4.8), update one sample $\mathbf{w}_k$ by taking into account gradients computed for the other samples as well. This makes us go towards area with high probability mass.

The term $\nabla_{\mathbf{w}_j} k(\mathbf{w}_{k'}, \mathbf{w}_k)$ works as a repulsive force. It prevents the convergence of all samples $\mathbf{w}_k$ to the same values. As a result of that, our samples try to populate the whole area of high probability mass.

4.4.6 Uncertainty estimation with Bayesian neural networks

A useful property of ensemble methods is that they can produce confidence intervals for their predictions. Given a particular value of the input, we can compute multiple predictions by different ensemble members and treat those predictions as samples from the predictive distribution.

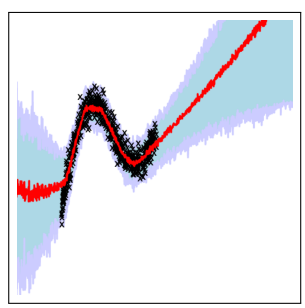


Figure 4.13: Uncertainty estimation with a Bayesian neural network (Blundell et al., 2015). Training examples are shown with the black crosses.

Fig. 4.13 shows predictive distributions computed with a Bayesian neural network whose parameters were estimated using variational approximation (Blundell et al., 2015). The confidence intervals are relatively small in the areas where we have the training examples shown with the black crosses. The width of the confidence intervals increases when we move farther away from the training data. This happens because different ensemble members produce more distinct predictions.

4.5 Data augmentation

A very powerful way to reduce overfitting is to increase the amount of training data. If we have access to a dataset $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$ of a limited size, we can increase the size of the data set by applying different kinds of transformations to training examples $\mathbf{x}^{(i)}$ while keeping the original label $\mathbf{y}^{(i)}$ for a transformed example. This approach is often called *data augmentation*.

The simplest way to modify training examples is to corrupt them with random noise (Sietsma and Dow, 1991). In each iteration, for each mini-batch, we apply a different noise instance and therefore we essentially create an infinite dataset. Gaussian noise is very commonly used for noise injection (see Fig. 4.14). A nice property of this method is its simplicity and applicability in many application domains.

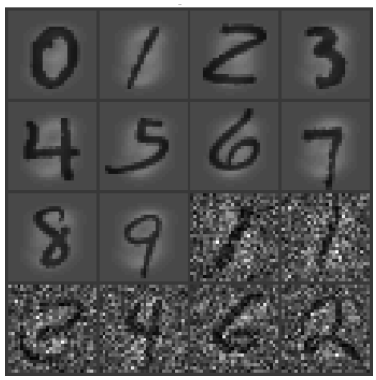


Figure 4.14: Data augmentation by noise injection.

4.5.1 Image transformations

In the domain of image processing, a very powerful way to increase the amount of data is to modify training examples by a set of chosen transformations. Popular transformations include random crop, random translation, random scaling, flipping, random rotation. Again, we generate an infinite dataset by applying a different transformation to each training example in each new epoch.

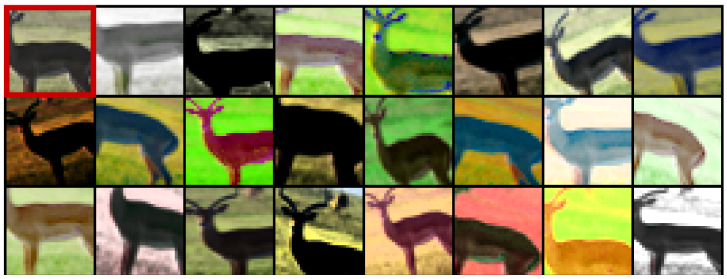


Image from (Dosovitskiy et al., 2014)

Figure 4.15: Example image transformations.

In Fig. 4.15, you can see examples of random transformations applied to an image patch. We can see that one can generate a huge number of training examples just from a single image. By feeding transformed images, we train our network to be invariant to such transformations. Sometimes transformed images do not look even realistic but this normally does not cause problems. It turns out that the networks generalize better when they are trained on challenging examples even if they do not look very realistic.

This data augmentation scheme works very well in the image domain. Here, we use our prior knowledge about the domain: we know that transformations should not affect the label of the training example. In other domains, designing such transformations may be less trivial.

4.5.2 mixup

In the data augmentation scheme called *mixup* (Zhang et al., 2017) virtual training examples are constructed in the following way. We select a random number λ between [0, 1] and mix two training examples with weights λ and $1-\lambda$. We do this both for input $\mathbf{x}$ and output $\mathbf{y}$.

$$\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j$$
$$\tilde{\mathbf{y}} = \lambda \mathbf{y}_i + (1-\lambda)\mathbf{y}_j$$

In classification tasks, $\mathbf{y}$ is a one-hot encoded representation of the target class. Note that for images, we create mixtures of two different images which may look quite unrealistic. Nevertheless, this data augmentation method works well and has positive effect on model generalization. This data augmentation encourages the model to behave linearly in-between training examples. Therefore, we create a bias in favor of linear functions, which helps regularize the learning problem.

4.5.3 Adversarial examples

The generalization performance of a model can be improved by augmenting the training data with *adversarial examples*.

When we train a neural network, we minimize a loss that is usually computed as a sum of losses for each training example $\mathbf{x}^{(n)}, \mathbf{y}^{(n)}$:

$$\frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(\mathbf{x}^{(n)}, \mathbf{y}^{(n)}, \mathbf{w}) \to \min_{\mathbf{w}}$$

where $\mathcal{L}(\mathbf{x}^{(n)}, \mathbf{y}^{(n)}, \mathbf{w})$ is, for example, a cross-entropy loss. Szegedy et al. (2014) discovered that it is very easy to fool a trained neural network. One can modify a given example $\mathbf{x}$ with label $\mathbf{y}$ such that the classifier would output a wrong prediction. We can do that by solving the following optimization problem:

$$\mathcal{L}(\mathbf{x}+\mathbf{r}, \mathbf{y}, \mathbf{w}) \to \max_{\mathbf{r}}$$

We add a perturbation $\mathbf{r}$ to the input $\mathbf{x}$ and instead of minimizing the loss wrt the model parameters, we maximize the loss wrt to the perturbation vector $\mathbf{r}$.

Since we want to stay close to the original training example (we want to keep our modification small and perhaps not noticeable), we add an extra constraint that the norm of the perturbation vector should be small, for example, smaller than some ε:

$$\|\mathbf{r}\| \leq \varepsilon.$$

Such modified inputs $\mathbf{x} + \mathbf{r}$ are called adversarial examples.

It is quite surprising that adversarial examples can be found very close to training examples. Fig. 4.16 presents a famous example in which an image of a panda (shown in the left) is correctly classified as "panda" by a trained classifier. Then we add a small (carefully tuned) perturbation (shown in the middle) to the image and produce an adversarial example which is shown on the right hand side. For humans, the adversarial example and the original image look exactly the same. However, the classifier changes its prediction for the adversarial example from "panda" to "gibbon".

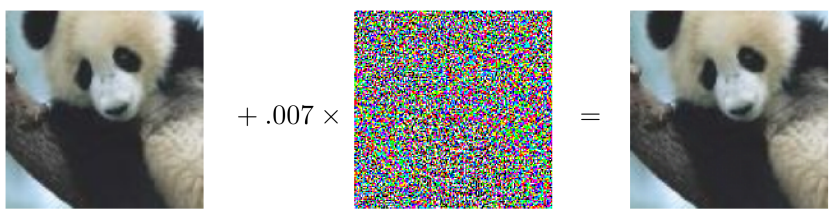


Figure 4.16: Adversarial example found with FGSM: The original image is shown on the left, the perturbation $\mathbf{r}$ is in the middle and the adversarial example is on the right.

Finding adversarial examples is also surprisingly easy. For example, the fast gradient sign method (FGSM) (Goodfellow et al., 2014) simply takes the sign of the gradient of the loss wrt to input $\mathbf{x}$, multiplies it by small ε and adds it to the modified example:

$$\mathbf{x} + \mathbf{r} = \mathbf{x} + \varepsilon \, \text{sign}(\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{w}, \mathbf{x}, \mathbf{y})).$$

4.5.4 Adversarial training

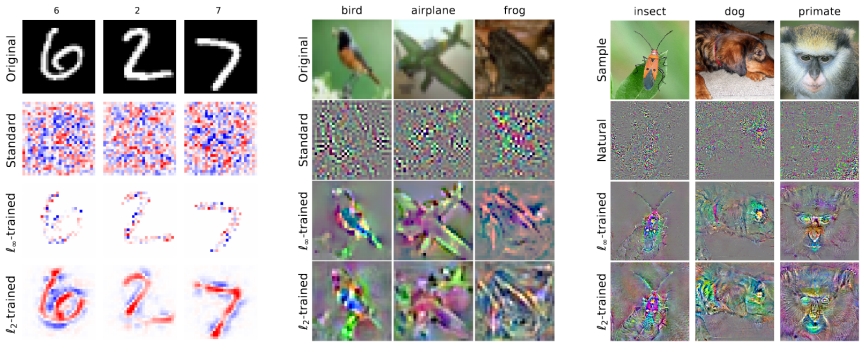
The idea of adversarial training is to augment the training data with adversarial examples. Since adversarial examples are difficult for neural networks, including them in the training set can help improve generalization.

The existence of adversarial examples motivated a whole new subfield of deep learning in which the goal is to develop techniques to defend neural networks against adversarial attacks. One of the popular defense techniques in adversarial deep learning is Madry's defense model (Madry et al., 2017). To train a model with that technique, we never use clean examples during training. Instead, we always use the worst adversarial examples. There are two optimization loops in the training procedure:

$$\min_{\mathbf{w}} E_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \left[\max_{\delta \in \mathcal{S}} \mathcal{L}(\mathbf{w}, \mathbf{x} + \delta, \mathbf{y}) \right].$$

In the inner loop, we find the worst adversarial example $\mathbf{x}+\delta$ for a given training example $\mathbf{x}$. In the outer loop, we tune the parameters of the neural network to classify the adversarial example correctly.

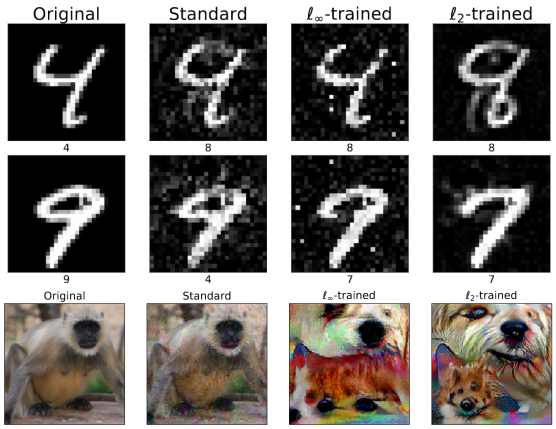
It is interesting that this kind of adversarial training helps develop more semantically meaningful representations, which is illustrated in Fig. 4.17. In



images from (Madry et al., 2018)

Figure 4.17: Adversarial training help develop more semantically meaningful representations.

the first row we see training examples from three different datasets. In the second row, we see the gradients of the loss wrt the input computed for the same examples for the model trained in a standard way without adversarial examples. Intuitively, the pattern in the gradient should suggest how to modify the input so as to change the class label. Unfortunately, the gradients do not look reasonable and look more like noise. In contrast, the gradients computed for the models trained with Madry's defence look much more meaningful. The gradients are high in the locations which we believe are important for producing the correct label.



images from (Madry et al., 2018)

Figure 4.18: Original examples (first column) and adversarial examples for a neural network trained normally (second column) and with Madry's defense (third and fourth column).

In Fig. 4.18, we show adversarial examples found for a neural network trained normally and for two neural networks trained with Madry's defense model. We can see that the adversarial examples found with Madry's model look very

confusing even for humans. This shows that fooling such models is a much more difficult task.

4.6 Rethinking generalization

Recall the conventional wisdom that we presented at the beginning of this chapter: overfitting happens when the model is too flexible for the amount of training data. Traditionally, the flexibility of the model is connected to the number of model parameters. The study by Zhang et al. (2016) challenges the conventional wisdom. They show that the same neural networks which are flexible enough to memorize large and unstructured datasets can generalize very well in real-world learning problems.

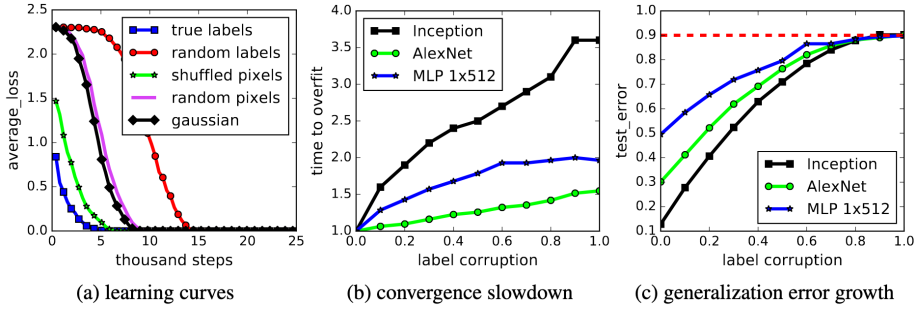


Figure 4.19: Fitting randomized CIFAR-10 data with a convolutional neural network.

Fig. 4.19a presents the evolution of the training error during training for several experiments with the same convolutional network.

- The blue curve is the learning curve for the CIFAR-10 dataset.
- The green curve represents learning of the same dataset with shuffled pixels. In this experiment, the same random permutation of pixels is applied to all the images. By shuffling the pixels, we destroy the spatial structure of the images but despite that, the convolutional network is able to learn this dataset, which is indicated by the zero training error.
- The black and magenta curves describe experiments with randomized inputs. To obtain the magenta curve, the pixels of each image were shuffled by a different random permutation. To obtain the black curve, the input images were generated by drawing samples from the Gaussian distribution (with mean and variance matching the statistics of the CIFAR-10 dataset). In both cases, the training error again goes to zero.
- Finally, the authors used the original images but they randomized labels and still the same neural network was able to learn to classify the training set correctly.

Fig. 4.19b presents results with partially corrupted labels: with probability p the label of each image was corrupted as a uniform random class. The time

that it took to achieve zero error grew as a function of the corruption rate p. The same effect happened for three different convolutional architectures.

What is surprising is that exactly the same networks generalize very well for uncorrupted data and achieve a reasonable level of generalization for partial label corruption (see Fig. 4.19c). These experiments show that the same neural network that has enough parameters to memorize very complex and unstructured datasets can generalize very well on real data. These results challenge the common wisdom on how the number of parameters is related to overfitting.

The authors argue that explicit regularization may improve generalization performance, but is neither necessary nor sufficient for controlling generalization error. And the same network generalizes well on real data. Weight decay and data augmentation only slightly improve the generalization performance (see Table 4.1).

Table 4.1: Explicit regularization (such as augmentation with random crop or weight decay) may improve generalization performance, but is neither necessary nor sufficient for controlling generalization error.

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78

In Table 4.2, we can see that batch normalization is usually found to improve the generalization performance, even though it was not explicitly designed for regularization. The results without batch norm are worse than the results with the batch norm. Also the authors argue that stochastic gradient descent may act as an implicit regularizer.

Table 4.2: Batch normalization is usually found to improve the generalization performance, even though it was not explicitly designed for regularization.

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78
Inception w/o BatchNorm	1,649,402	no	yes	100.0	83.00
		no	no	100.0	82.00
(fitting random labels)		no	no	100.0	10.12

4.7 Hyperparameter search

4.7.1 Grid search

In this chapter, we studied several techniques that can be used to avoid overfitting. It is hard to know in advance which of those techniques would work best in a particular learning problem. One needs to try different combinations

of those techniques in order to select the best configuration. This process is known as hyperparameter tuning or hyperparameter search.

During hyperparameter search, the performance on the validation set is used to select the optimal values of the hyperparameters. The hyperparameters that you may want to tune are:

- learning rate schedule
- transformations used for data augmentation
- weight decay coefficient
- dropout rate
- mini-batch size
- number of layers
- number of neurons
- convolution kernel width
- nonlinearity

Training a pretty large model with strong regularization such as data augmentation often works best in practice. It is typical that the training error becomes very low during training, which means that there is usually a gap in performance between the training and the validation set. You should not expect that the training and validation errors would be similar.

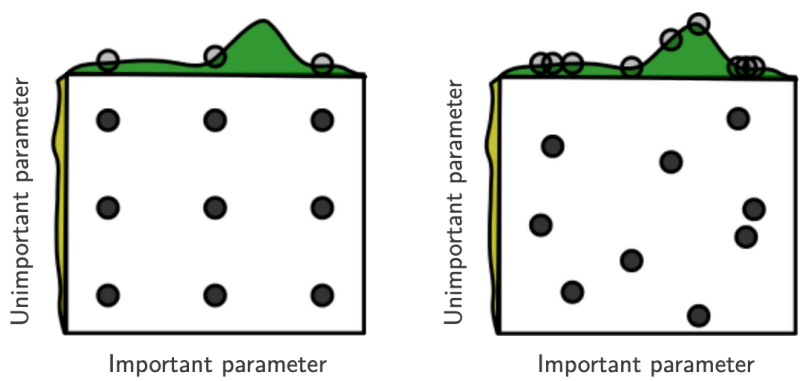


image from (Bergstra and Bengio, 2012)

Figure 4.20:

Perhaps the simplest way of tuning the hyperparameters is called *grid search*. One selects a fixed set of possible values for each hyperparameter and trains a neural network with all possible combinations of the hyperparameters values. In Fig. 4.20, we have two hyperparameters and for each hyperparameter we choose three possible values. We train our model with all the nine combinations of the hyperparameters. This approach is conceptually very easy but unfortunately it is computationally very expensive. The computational cost increases exponentially with the number of hyperparameters.

Let us look at the hypothetical example in Fig. 4.20. Suppose that the first hyperparameter affects the performance: the plot on the top of the figure shows the performance of the model (for example, model accuracy) as a function of the first hyperparameter. We can see that the optimal value of the hyperparameter is somewhere between the second and the third value on the grid. However, we do not find this value because we only test three values of that hyperparameter.

Suppose that the second hyperparameter is not important for the model performance. By doing grid search, we perform three times more computations than would be needed to tune only the important parameter. Therefore, grid search is likely to be wasteful in terms of computations.

4.7.2 Random search

A simple alternative to grid search is random search. In random search, we try combinations of random values of the hyperparameters instead of taking them from a fixed grid. (see the rhs plot of Fig. 4.20). In the hypothetical example in Fig. 4.20, the first hyperparameter is important and the second hyperparameter is not important. We try nine different values of the important hyperparameter instead of only three values in the grid search. Therefore, we increase the chance of finding the value that maximizes the model performance.

Chapter 5

Recurrent neural networks

In the previous chapters, we considered supervised learning problems in which inputs were vectors of fixed size. For example, in the MNIST classification problem, the inputs were 28×28 images which could be viewed as a two-dimensional map of fixed width and height or as a vector with 784 elements. In some tasks, inputs can be sequences and each input sequence can have a different number of elements. For example, one sequence may have three elements and another sequence may contain four elements:

$$\left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\right) \rightarrow y^{(1)}$$
$$\left(x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, x_4^{(2)}\right) \rightarrow y^{(2)}.$$

We should be able to process both sequences. One example of such learning problem is *sentiment analysis* in which the task is to classify paragraphs of text (see Table 5.1).

Input sequence	**Class**
Dear #XYZ there is no network in my area and internet service is pathetic from past one week. Kindly help me out.	negative review
Although the value added services being provided are great but the prices are high.	mixed review
Great work done #XYZ Problem resolved by customer care in just one day.	postive review

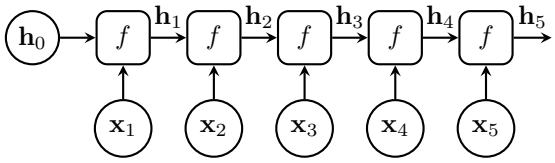
Table 5.1: The task of sentiment analysis.

How can we process sequences with varying lengths? Let us start with a simple example. Suppose we want to count the number of zeros in an input sequence $(x_1, x_2, x_3, \ldots x_T)$. In order to do that, we can write a script

```
h = 0
for x in input_sequence:
    if x == 0:
        h = h + 1
```

in which we initialize counter `h` with zero and then loop over all the elements in the input sequence and increment `h` by one if the element `x` is equal to zero.

We can implement these computations in a computational graph shown in Fig. 5.1. We initialize the counter with $\mathbf{h}_0 = 0$ and then apply a pre-defined function f to the counter $\mathbf{h}$ after seeing each new element $\mathbf{x}_i$ of the input sequence.




```
def f(x, h):
    return h + (x == 0)
```

Figure 5.1: A computational graph to count the number of non-zero elements in the input sequence.

In this simple example, it is easy to design a function that processes each element of the input sequence. What should we do in more difficult cases, for example, when we want to train a model to produce the correct label in a sentiment analysis task. How should we process each element of the input sequence?

To build a generic processor of sequences, we can use the same computational graph as in our simple example. But instead of using a fixed and pre-defined function f, we can use a learnable function f that can be tuned to produce the correct outputs at the end of the sequence (see Fig. 5.2).

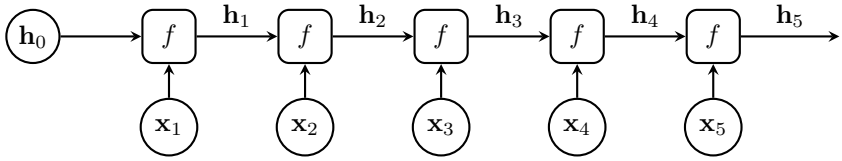


Figure 5.2: A computational of a generic processor of a sequence with varying length.

What should we use as the computational block f in this architecture? A simple solution is to use a block that has a similar architecture as the classical multilayer perceptron that we studied in Section 1.3:

$$f(\mathbf{x}, \mathbf{h}) = \tanh(\mathbf{W}\mathbf{h} + \mathbf{U}\mathbf{x} + \mathbf{b}).$$

Here, we have a linear combination of the previous values of $\mathbf{h}$, a linear combination of the inputs $\mathbf{x}$, the bias terms $\mathbf{b}$ and a nonlinear activation function tanh. And we use this block on every step in our computational graph. Note that we apply exactly the same function f again and again to process the elements of the input sequence. Therefore, this architecture is called a recurrent neural network (RNN). Vector $\mathbf{h}$ which is updated at each iteration of the RNN is often called the *hidden state* of the RNN.

In Fig. 5.3, we compare the computational graph of a multilayer perceptron network with two layers and the computational graph of a recurrent neural network which processes a sequence with two elements. Note that each layer of the MLP has a different set of parameters $\mathbf{W}_1$ and $\mathbf{W}_2$. We can view the computational graph of the RNN as a neural network with two layers. There are two differences compared to the feedforward network:

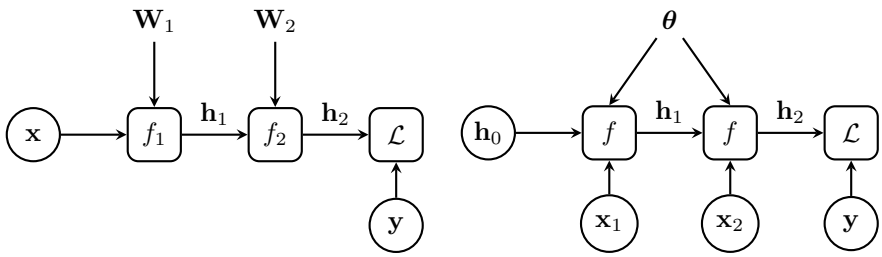


Figure 5.3: Left: Computational graph of a feedforward network. Right: Computational graph of an RNN.

- external inputs are added at every step,
- same parameters are used in every layer.

5.1 Training recurrent neural networks

The parameters of a recurrent neural network can also be tuned by gradient descent or stochastic gradient descent. Suppose that we want to solve a sentiment analysis task with an RNN. We need to produce the probability distribution over existing classes at the output of our recurrent neural network. In order to obtain a proper distribution, we can use the softmax nonlinearity after the last layer (see Fig. 5.4). Since it is a classification problem, we can use the cross-entropy loss.

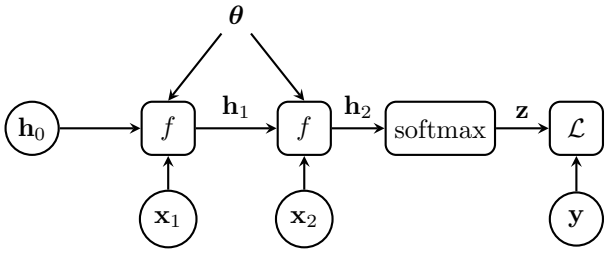


Figure 5.4: An example computational graph in the sentiment analysis task.

The loss function that we want to minimize can be written as a sum of terms in which n corresponds to one training example (one input sequence) and j corresponds to one of the classes:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{j=1}^{K} y_j^{(n)} \log z_j^{(n)}.$$

To use (stochastic) gradient descent, we need to compute the gradient of the loss function. We can do that efficiently with the backpropagation algorithm.

We first review the backpropagation algorithm for a multilayer neural network without recurrence. Let us assume for simplicity that our neural network operates with scalar signals:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w),$$

where x is the model input, y is the output and w, θ are the model parameters. We can compute the derivatives wrt θ and w using the backpropagation algorithm as illustrated in Fig. 5.5.

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta}$$
$$\frac{\partial \mathcal{L}}{\partial w} = \underbrace{\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}}\frac{\partial h}{\partial w}$$

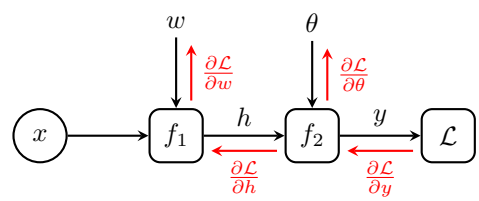


Figure 5.5: Backpropagation in a feedforward network.

Let us now consider a recurrent neural network with two layers which also operates with scalar signals:

$$\mathcal{L} = \mathcal{L}(h_2), \quad h_2 = f(x_1, h_1, \theta), \quad h_1 = f(x_1, h_0, \theta).$$

The difference to the feedforward network is that each layer implements the same function and therefore each layer has the the same (shared) parameters θ. This illustrated in Fig. 5.6.

$$\mathcal{L} = \mathcal{L}(h_2)$$
$$h_2 = f(x_1, h_1, \theta)$$
$$h_1 = f(x_1, h_0, \theta)$$

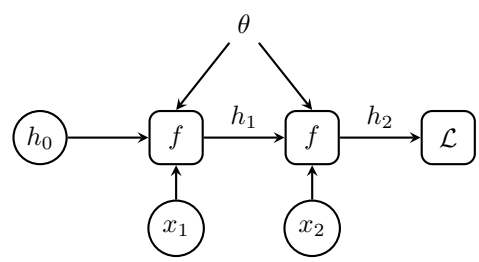


Figure 5.6: A computational graph of a recurrent neural network.

Let us assume for now that the parameters of the layers are not shared. The first layer has parameter θ_1 and the second layer has parameter θ_2. This results in the computational graph that is similar to the feedforward network (see Fig. 5.7). Now we can compute the derivatives wrt parameters θ_1 and θ_2 the same way we did for the feedforward network without recurrence. We can compute the derivatives efficiently using backpropagation.

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial h_2}\frac{\partial h_2}{\partial \theta_2}$$
$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \underbrace{\frac{\partial \mathcal{L}}{\partial h_2}\frac{\partial h_2}{\partial h_1}}_{\frac{\partial \mathcal{L}}{\partial h_1}}\frac{\partial h_1}{\partial \theta_1}$$

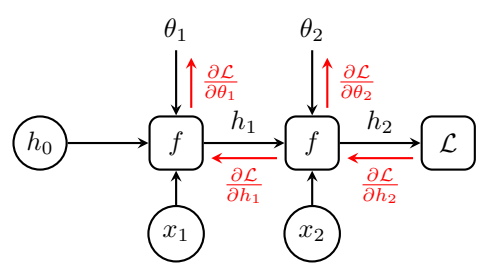


Figure 5.7: Backpropagation in a network from Fig. 5.6 assuming different parameters of the layers.

Finally, we need to take into account the fact the parameters of the first and second layer are shared. We can do so simply by combining the gradients using the rules of differentiation. Since the partial derivatives $\frac{\partial\theta_1}{\partial\theta} = \frac{\partial\theta_2}{\partial\theta} = 1$, we simple sum the partial derivatives $\frac{\partial\mathcal{L}}{\partial\theta_1}$ and $\frac{\partial\mathcal{L}}{\partial\theta_2}$. This is illustrated in Fig. 5.8. Note that those are the derivatives that were computed by following two different routes from the loss to the parameters. Therefore, we need to sum all the partial derivatives that are computed by following all possible routes from the loss to the parameters. The backpropagation algorithm applied to recurrent neural networks is often called *backpropagation through time*.

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\theta} &= \frac{\partial\mathcal{L}}{\partial\theta_1}\frac{\partial\theta_1}{\partial\theta} + \frac{\partial\mathcal{L}}{\partial\theta_2}\frac{\partial\theta_2}{\partial\theta} \\ &= \frac{\partial\mathcal{L}}{\partial\theta_1} + \frac{\partial\mathcal{L}}{\partial\theta_2}\end{aligned}$$

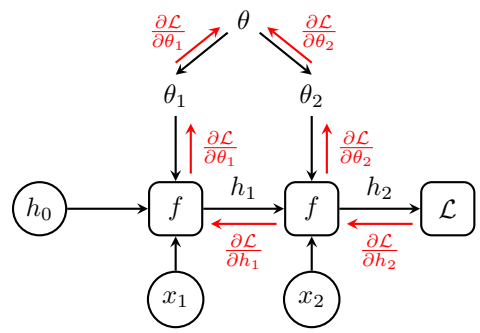


Figure 5.8: Backpropagation in a recurrent neural network.

5.2 Problems with RNN training

Does recurrence cause problems for training? Consider a recurrent neural network in which the computations are similar to the ones in the multilayer perceptron:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W}, \mathbf{U}, \mathbf{b}) = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}). \tag{5.1}$$

Let us first assume that we are not careful about selecting the nonlinearity ϕ and we select it to be the identity mapping $\phi(a) = a$. Let us also assume that the hidden state at time 0 is initialized with $\mathbf{h}_0 = 0$ and that we do not use the bias term: $\mathbf{b} = 0$. Then, the hidden state at time t is computed as shown in this formula.

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t = \mathbf{W}\left(\mathbf{W}\mathbf{h}_{t-2} + \mathbf{U}\mathbf{x}_{t-1}\right) + \mathbf{U}\mathbf{x}_t \\ &= \mathbf{W}\mathbf{W}\mathbf{h}_{t-2} + \mathbf{W}\mathbf{U}\mathbf{x}_{t-1} + \mathbf{U}\mathbf{x}_t = \sum_{\tau=1}^{t}\mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau \end{aligned} \tag{5.2}$$

Now let us assume that matrix $\mathbf{W}$ is diagonalizable which means that it can be written as

$$\mathbf{W} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}\,,$$

where $\mathbf{Q}$ is a matrix of eigenvectors $\mathbf{q}_i$ of $\mathbf{W}$ in its columns and $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues of $\mathbf{W}$ on the main diagonal. In general, there is

no guarantee that $\mathbf{W}$ would be diagonalizable, but we make this assumption to make the analysis simpler. Now we can write:

$$\mathbf{W}^{t-\tau} = \underbrace{\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{-1}\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{-1}\dots\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{-1}}_{t-\tau \text{ times}} = \mathbf{Q}\boldsymbol{\Lambda}^{t-\tau}\mathbf{Q}^{-1}$$

We have the diagonal matrix $\boldsymbol{\Lambda}$ in the power $t-\tau$ in this expression. Let us now look at one term in the formula for $\mathbf{h}_t$ in (5.2):

$$\mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau = \mathbf{Q}\boldsymbol{\Lambda}^{t-\tau}\underbrace{\mathbf{Q}^{-1}\mathbf{U}\mathbf{x}_\tau}_{=\mathbf{z}} = \mathbf{Q}\boldsymbol{\Lambda}^{t-\tau}\mathbf{z} = \sum_i \mathbf{q}_i\lambda_i^{t-\tau}z_i$$

where we denote $\mathbf{z} = \mathbf{Q}^{-1}\mathbf{U}\mathbf{x}_\tau$ and z_i is the i-th component of $\mathbf{z}$. Now we can see that if there is an eigenvalue λ_i such that $|\lambda_i| > 1$ (and the corresponding z_i is non-zero), then the norm of the corresponding term $\mathbf{q}_i\lambda_i^{t-\tau}z_i$ will grow exponentially with t. This will cause explosions in the forward computations.

In a more general case, let $\mathbf{Q}_m$ be an $n \times m$ matrix containing the m linear independent eigenvectors $\mathbf{q}_i$ of $\mathbf{W}$ in its columns and $\boldsymbol{\Lambda}$ be a diagonal matrix made of the corresponding eigenvectors λ_i:

$$\mathbf{W}\mathbf{Q}_m = \mathbf{Q}_m\boldsymbol{\Lambda}$$

We can write $\mathbf{U}\mathbf{x}_\tau = \mathbf{Q}_m\mathbf{z} + \mathbf{z}'$ where $\mathbf{z}'$ belongs to the null space of $\mathbf{Q}_m$, that is $\mathbf{Q}_m^\top\mathbf{z}' = 0$. Then, one term in (5.2) is

$$\mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau = \mathbf{W}^{t-\tau}\mathbf{Q}_m\mathbf{z} + \mathbf{W}^{t-\tau}\mathbf{z}'$$

Let us look at the first term only:

$$\begin{aligned}\mathbf{W}^{t-\tau}\mathbf{Q}_m\mathbf{z} &= \mathbf{W}^{t-\tau-1}\mathbf{W}\mathbf{Q}_m\mathbf{z} = \mathbf{W}^{t-\tau-1}\mathbf{Q}_m\boldsymbol{\Lambda}\mathbf{z} = \mathbf{W}^{t-\tau-2}\mathbf{Q}_m\boldsymbol{\Lambda}^2\mathbf{z} = \dots \\ &= \mathbf{Q}_m\boldsymbol{\Lambda}^{t-\tau}\mathbf{z} = \sum_i \mathbf{q}_i\lambda_i^{t-\tau}z_i\end{aligned}$$

Again, if one of the eigenvalues is such that $|\lambda_i| > 1$ (and the corresponding z_i is non-zero), then the norm of $\mathbf{q}_i\lambda_i^{t-\tau}z_i$ will grow exponentially causing explosions in the forward computations.

The largest absolute value among all eigenvalues of a matrix is called the *spectral radius*:

$$\text{spectral radius}(\mathbf{W}) = \max_i |\lambda_i|.$$

Using this terminology, we can say that explosions in the forward computations happen if the spectral radius of $\mathbf{W}$ is greater than 1.

What if we use tanh as the nonlinearity in (5.1)?

$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

In this case, explosions cannot happen because tanh is bounded in $(-1, 1)$.

5.2.1 Are there similar problems in backward computations?

Now let us analyze the backward computations. To do that, let us look at the longest route of derivative computations for an RNN

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

which is shown with the red arrows in Fig. 5.9.

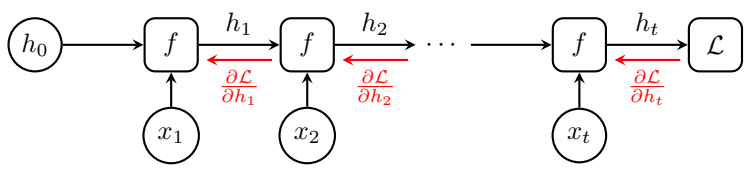


Figure 5.9: The longest route of derivative computations for an RNN.

Using the chain rule, we can write the partial derivative wrt the hidden state $\mathbf{h}_1$ as the matrix product of the Jacobian matrices of the intermediate trasformations:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^\top = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \mathrm{diag}(\phi'_\tau)\mathbf{W}. \tag{5.3}$$

In this expression, $\mathrm{diag}(\phi'_\tau)$ is a diagonal matrix with the diagonal elements given by $\phi'_\tau = \phi'(\mathbf{W}\mathbf{h}_{\tau-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$. ϕ' is the derivative of the nonlinearity that we used in the recurrent cell. $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}$ is a column vector of partial derivatives.

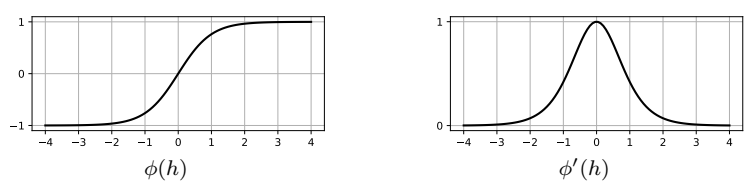


Figure 5.10: *tanh* nonlinearity (left) and its derivative (right).

Now suppose that we use the $\phi(h) = \tanh(h)$ nonlinearity whose derivative is shown in Fig. 5.10. Suppose that all our neurons in the RNN are not saturated which means that their inputs are somewhere close to zero. That means that the absolute values of the derivatives ϕ' are greater than some threshold γ:

$$|\phi'_\tau| \geq \gamma.$$

Now if the spectral radius of $\mathbf{W}$ is greater than $1/\gamma$, then the gradient explodes.

It is easiest to see this if we assume that the hidden state $\mathbf{h}$ is one-dimensional, then $\mathbf{W}$ is simply a scalar w and the spectral radius of $\mathbf{W}$ is simply the absolute value $|w|$. Then, the absolute value of (5.3) becomes

$$\left| \frac{\partial \mathcal{L}}{\partial h_t} \prod_{\tau=t,\dots,2} \phi'_\tau w \right| \geq \left| \frac{\partial \mathcal{L}}{\partial h_t} \right| \prod_{\tau=t,\dots,2} \gamma |w| = \left| \frac{\partial \mathcal{L}}{\partial h_t} \right| (\gamma |w|)^{t-1}$$

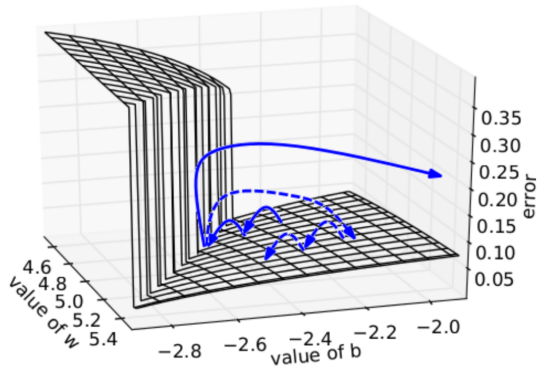


image from (Pascanu et al., 2013)

Figure 5.11: The blue solid line represents an optimization trajectory when training parameters of an RNN. Exploding gradients can cause taking too large steps, which can harm training. Gradient clipping can mitigate this problem. With clipped gradients, the optimization process is less affected by the gradient explosion problem (dashed line).

If $|w| > 1/\gamma$, we get exponential growth of this value with t. To avoid explosions, it is good to keep neurons in the saturated regime where derivatives ϕ' are small.

This analysis shows that the gradient may explode even for a bounded activation function such as tanh. Gradient explosions caused by recurrence is one big problem with training RNNs, which is illustrated in Fig. 5.11. One workaround is to clip the gradient if it is larger than some pre-defined value. This can be done element-wise (Mikolov, 2012) or by clipping the norm (Pascanu et al., 2013):

$$\text{if } \|\mathbf{g}\| \geq \Delta, \text{ then } \mathbf{g} \leftarrow \Delta \frac{\mathbf{g}}{\|\mathbf{g}\|}.$$

Clipping the gradient can mitigate the gradient explosion problem (see Fig. 5.11). In PyTorch, clipping of gradients can be done by re-writing the attribute `parameter.grad.data` after calling `loss.backward()`.

5.2.2 Vanishing gradients

Let us now look at the gradients again.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^\top = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\ldots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\ldots,2} \operatorname{diag}(\phi'_\tau)\mathbf{W}$$

We can see from Fig. 5.10 that if we use tanh nonlinearity, then the absolute values of the derivative $|\phi'|$ are bounded:

$$0 < |\phi'_\tau| \leq 1.$$

That means that if the spectral radius of $\mathbf{W}$ is smaller than 1, then the gradient will vanish, which means that its norm will decay exponentially with the increase of t. Again, it is easiest to see this if we assume one-dimensional states $\mathbf{h}$ and scalar $\mathbf{W} = w$.

$$\left| \frac{\partial \mathcal{L}}{\partial h_t} \prod_{\tau=t,\ldots,2} \phi'_\tau w \right| \leq \left| \frac{\partial \mathcal{L}}{\partial h_t} \right| \prod_{\tau=t,\ldots,2} |w| = \left| \frac{\partial \mathcal{L}}{\partial h_t} \right| |w|^{t-1}$$

If $|w| < 1$, we get exponential decay of this value with t. To avoid vanishing gradients, it is good to keep neurons in the non-saturated regime where derivatives ϕ' are close to 1.

This analysis suggests that it is very difficult to balance the gradients in the simple recurrent neural networks of type

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}).$$

In order to avoid gradient explosions, it is better to keep neurons in the saturated mode, while in order to avoid vanishing gradients, we need to to keep the neurons in the linear mode. This causes major problems in training recurrent neural networks.

Why do vanishing gradients cause problems for learning? Due to vanishing gradients, it is difficult to learn long-range dependencies in the data. For example in sentiment analysis, it is difficult to capture the effect of the first words in a paragraph on the predicted class. In time-series modeling, it is difficult to capture slowly changing phenomena.

These are the problems of the vanilla RNNs whose recurrent unit is inspired by a classical MLP and this is the reason why those networks are rarely used in practice. What works much better is recurrent neural networks with a gating mechanism inside the recurrent units. The two most popular units are called gated recurrent unit (GRU, Cho et al., 2014) and long short-term memory (LSTM, Hochreiter and Schmidhuber, 1997).

5.2.3 Historical note on RNNs

Recurrent neural networks for sequential data processing were proposed already in the 1980s (Rumelhart et al., 1986; Elman, 1990; Werbos, 1988). However, RNNs did not gain much popularity because they were particularly difficult to train with backpropagation due to unstable training because of gradient explosions and difficulty to learn long-term dependencies due to vanishing gradients (Bengio et al., 1994). The breakthrough came with the invention of Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997), a recurrent neural network which was designed to solve the problem of gradient explosions and vanishing gradients. However, the LSTM remained largely unnoticed in the community until the deep learning boom started.

5.3 Gated recurrent unit (GRU)

Cho et al. (2014) give the following motivation for the gating mechanism in GRU. In the vanilla RNN cell, we re-write all the elements of the state $\mathbf{h}_{t-1}$ with the new values $\mathbf{h}_t$:

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x} + \mathbf{b}).$$

What if we want to keep some of the old values $\mathbf{h}_{t-1}$ of the state unchanged? For example, we may observe two kinds of processes in the data: one process

changes slowly and the other one changes fast. To track the fast process, we may want to update the state at every step but the state that describes the slow process may be updated less frequently. How can we achieve that?

GRU uses an *update gate* $\mathbf{u}_t \in (0,1)$ that controls which states should be updated:

$$\begin{aligned}\mathbf{h}_t &= (1-\mathbf{u}_t)\odot\mathbf{h}_{t-1} + \mathbf{u}_t\odot\widetilde{\mathbf{h}}_t\\ \mathbf{u}_t &= \sigma(\mathbf{W}_u\mathbf{h}_{t-1} + \mathbf{U}_u\mathbf{x}_t + \mathbf{b}_u)\end{aligned}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic function and $\widetilde{\mathbf{h}}_t$ are the new state candidates. The gate is computed using a linear combination of the inputs $\mathbf{x}_t$ and the previous states $\mathbf{h}_{t-1}$. The value of each element in $\mathbf{u}_t$ is in $(0,1)$ because it is the output of the logistic function. If the gate is zero, we keep the previous values of the states. If the gate is one, we re-write the state with the new value $\widetilde{\mathbf{h}}_t$. The gate is a vector with as many elements as there are states in the recurrent unit. Therefore, we have the flexibility to update only some of the states and keep some of the states unchanged.

The new candidate value of the state $\widetilde{\mathbf{h}}$ as

$$\begin{aligned}\widetilde{\mathbf{h}}_t &= \phi(\mathbf{W}(\mathbf{r}_t\odot\mathbf{h}_{t-1}) + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h)\\ \mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{U}_r\mathbf{x}_t + \mathbf{b}_r).\end{aligned}$$

Here, we apply a nonlinearity ϕ such as tanh to a linear combination of the previous states and the inputs. One difference is that we use another gate $\mathbf{r}_t$ that controls which elements of $\mathbf{h}_{t-1}$ are used in the computations. If an element of $\mathbf{r}_t$ is zero, then we do not use the corresponding state. If an element of $\mathbf{r}_t$ is one, then we do use that state. Gate $\mathbf{r}_t$ is called a *reset* gate.

Does GRU actually help with the problem of exploding and vanishing gradients? Let us look at the gradient propagation in the GRU-based RNN assuming for simplicity that gates $\mathbf{u}_t$ and $\mathbf{r}_t$ are fixed, which means that we do not propogate derivatives through them. The Jacobian matrix of the state transformation in the GRU is then given by

$$\frac{\partial\mathbf{h}_\tau}{\partial\mathbf{h}_{\tau-1}} = \operatorname{diag}(1-\mathbf{u}_\tau) + \operatorname{diag}(\mathbf{u}_\tau)\operatorname{diag}(\phi'_\tau)\mathbf{W}\operatorname{diag}(\mathbf{r}_\tau)$$

where $\phi'_\tau = \phi'(\mathbf{W}(\mathbf{r}_\tau\odot\mathbf{h}_{\tau-1}) + \mathbf{U}\mathbf{x}_\tau)$. We again look at the longest route of derivative propagation shown in Fig. 5.9:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\mathbf{h}_1}^\top &= \frac{\partial\mathcal{L}}{\partial\mathbf{h}_t}^\top\prod_{\tau=t,\ldots,2}\frac{\partial\mathbf{h}_\tau}{\partial\mathbf{h}_{\tau-1}}\\ &= \frac{\partial\mathcal{L}}{\partial\mathbf{h}_t}^\top\prod_{\tau=t,\ldots,2}(\operatorname{diag}(1-\mathbf{u}_\tau) + \operatorname{diag}(\mathbf{u}_\tau)\operatorname{diag}(\phi'_\tau)\mathbf{W}\operatorname{diag}(\mathbf{r}_\tau))\end{aligned}$$

Let us now assume for simplicity that the state of the RNN is one-dimensional:

$$\frac{\partial\mathcal{L}}{\partial h_t}\prod_{\tau=t,\ldots,2}((1-u_\tau) + u_\tau\phi'_\tau w r_\tau)$$

and that all intermediate signals do not change with τ

$$\frac{\partial \mathcal{L}}{\partial h_t} \prod_{\tau=t,\ldots,2} ((1-u) + u\gamma r)$$

where we denote $\gamma = \phi'_\tau w$. If we further assume that gates $u = r = \frac{1}{2}$, we get

$$\frac{\partial \mathcal{L}}{\partial h_t} \left(\frac{1}{2} + \frac{\gamma}{4}\right)^{t-1}.$$

Now let us do the same simplified analysis for the vanilla RNN cell in (5.1) Assuming that our states are one-dimensional, (5.3) becomes

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\ldots,2} \operatorname{diag}(\phi'_\tau)\mathbf{W} = \frac{\partial \mathcal{L}}{\partial h_t} \prod_{\tau=t,\ldots,2} \phi'_\tau w = \frac{\partial \mathcal{L}}{\partial h_t} \gamma^{t-1}$$

where we denote $\gamma = \phi'_\tau w$.

Let us compare the two expressions:

$$\begin{aligned} \text{vanilla RNN:} \quad & \frac{\partial \mathcal{L}}{\partial h_t} \gamma^{t-1} \\ \text{GRU:} \quad & \frac{\partial \mathcal{L}}{\partial h_t} \left(\frac{1}{2} + \frac{\gamma}{4}\right)^{t-1} \end{aligned}$$

If γ is small, we can neglect the term $\gamma/4$ in the expression for GRU. Then, the gradients in the GRU vanish with the rate $\frac{1}{2}$. In contrast, the gradients vanish in the vanilla RNN cell with the rate of γ which is much worse for small γ. If γ is large, the magnitudes of the gradients grow exponentially as $O\left((\gamma/4)^t\right)$ for the GRU. This is better than $O\left(\gamma^t\right)$ in the vanilla RNN. Thus, the gating mechanism mitigates the problem of exploding/vanishing gradients. Gradients may explode or vanish in the GRU-based recurrent neural networks but such problems occur much more rarely compared to the vanilla RNN.

5.3.1 Connection to probabilistic graphical models

$$\begin{aligned} p(\mathbf{h}_1) &= \mathcal{N}(\mathbf{h}_1 \mid \boldsymbol{\mu}_1, \mathbf{R}_1) \\ p(\mathbf{h}_t \mid \mathbf{h}_{t-1}) &= \mathcal{N}(\mathbf{h}_t \mid \mathbf{B}\mathbf{h}_{t-1}, \mathbf{R}) \\ p(\mathbf{x}_t \mid \mathbf{h}_t) &= \mathcal{N}(\mathbf{x}_t \mid \mathbf{A}\mathbf{h}_t, \mathbf{V}) \end{aligned}$$

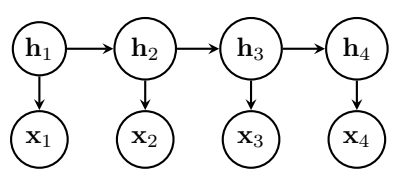


Figure 5.12: A linear dynamical system.

Consider a linear dynamical system, which is a linear Gaussian model with temporal structure. The graph of that probabilistic model is shown in Fig. 5.12. $\mathbf{x}_t$ is a sequence of observations, $\mathbf{h}_t$ is a sequence of hidden, unobserved states. In linear Gaussian models, all conditional probabilities which are represented

by the arrows in the graph are Gaussian with a linear dependence of the mean on the conditioning variables.

The inference in linear dynamical system is the evaluation of the conditional distribution $p(\mathbf{h}_t \mid \mathbf{x}_1, \ldots, \mathbf{x}_t)$ of the hidden states $\mathbf{h}_t$ given the observed sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$. Since we consider a linear Gaussian model, the inference can be done using the message-passing algorithm which is described for example, in Chapter 13 of (Bishop, 2006). The application of the message-passing algorithm to this system results in the famous Kalman filter algorithm. Even though the Kalman filter is not usually derived using the message passing algorithm, it can be seen as an implementation of that.

The Kalman filter consists of two steps. In the first step, we predict the future hidden state given the current posterior distribution over the hidden state at the current time step (see Fig. 5.13). Since we have a linear system, the prediction is computed using linear transformations of the random variable $\mathbf{h}_{t-1}$ which describes the current state.

$$
\begin{aligned}
p(\mathbf{h}_t \mid \mathbf{x}_1, ..., \mathbf{x}_{t-1}) &= \mathcal{N}(\mathbf{h}_t \mid \overrightarrow{\mathbf{h}}_t, \mathbf{P}_t) \\
\overrightarrow{\mathbf{h}}_t &= \mathbf{B}\bar{\mathbf{h}}_{t-1} \\
\mathbf{P}_t &= \mathbf{B}\boldsymbol{\Sigma}_{t-1}\mathbf{B}^\top + \mathbf{R}
\end{aligned}
$$

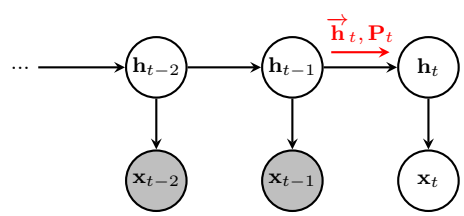
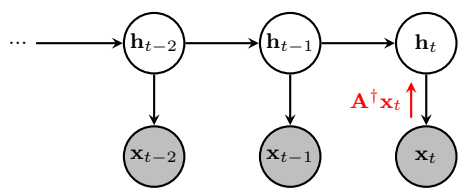


Figure 5.13: Prediction step of the Kalman filter.

On the second step, we correct our estimate of the hidden state at time t by using the observation $\mathbf{x}_t$ at time t (see Fig. 5.14).

$$
\begin{aligned}
p(\mathbf{h}_t \mid \mathbf{x}_1, ..., \mathbf{x}_t) &= \mathcal{N}(\mathbf{h}_t \mid \bar{\mathbf{h}}_t, \boldsymbol{\Sigma}_t) \\
\bar{\mathbf{h}}_t &= \overrightarrow{\mathbf{h}}_t + \mathbf{K}_t(\mathbf{x}_t - \mathbf{A}\overrightarrow{\mathbf{h}}_t) \\
\boldsymbol{\Sigma}_t &= (\mathbf{I} - \mathbf{K}_t\mathbf{A})\mathbf{P}_{t-1} \\
\mathbf{K}_t &= \mathbf{P}_{t-1}\mathbf{A}^\top(\mathbf{A}\mathbf{P}_{t-1}\mathbf{A}^\top + \mathbf{V})^{-1}
\end{aligned}
$$



The message from $\mathbf{x}_t$ to $\mathbf{h}_t$ is usually not explicitly expressed in the derivations of the Kalman filter.

Figure 5.14: Correction step of the Kalman filter.

The two steps can be seen as combining two messages that provide information about the state of the system. One message comes from the previous time step and the other message comes from the observation at the same time step.

Let us take a closer look at the correction equation for the mean values of the hidden states in the Kalman filter

$$
\begin{aligned}
\bar{\mathbf{h}}_t &= \overrightarrow{\mathbf{h}}_t + \mathbf{K}_t(\mathbf{x}_t - \mathbf{A}\overrightarrow{\mathbf{h}}_t) \\
\mathbf{K}_t &= \mathbf{P}_{t-1}\mathbf{A}^\top(\mathbf{A}\mathbf{P}_{t-1}\mathbf{A}^\top + \mathbf{V})^{-1}
\end{aligned}
$$

For simplicity, let us assume that the states h are one-dimensional. Then, we

can re-write the update rule of the Kalman filter in the following form:

$$
\begin{aligned}
\bar{h}_t &= \overrightarrow{h}_t + k_t(x_t - a\overrightarrow{h}_t) \\
&= \overrightarrow{h}_t + \frac{p_{t-1}a}{a^2 p_{t-1} + v}(x_t - a\overrightarrow{h}_t) \\
&= \overrightarrow{h}_t - \frac{p_{t-1}a^2}{a^2 p_{t-1} + v}\overrightarrow{h}_t + \frac{p_{t-1}a}{a^2 p_{t-1} + v}x_t \\
&= \frac{v}{a^2 p_{t-1} + v}\overrightarrow{h}_t + \frac{a^2 p_{t-1}}{a^2 p_{t-1} + v}\frac{x_t}{a} \\
&= (1 - u_t)\overrightarrow{h}_t + u_t\frac{x_t}{a} \qquad (5.4)
\end{aligned}
$$

where

$$
u_t = \sigma\left(\log a^2 p_{t-1} - \log(a^2 p_{t-1} + v)\right). \qquad (5.5)
$$

We can see the new value of h_t is computed by gating between the signal $\overrightarrow{h}_t$ that comes from the previous time step and the signal $\frac{x_t}{a}$ that comes from the current observation. Thus, the updated value is a trade-off between the estimate computed before observing x_t (our prior belief) and the value $\frac{x_t}{a}$ justified by the new observation (our likelihood).

Now compare (5.4)–(5.5) with the update rule of GRU:

$$
\begin{aligned}
\mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t \\
\mathbf{u} &= \sigma(\mathbf{W}_u \mathbf{h}_{t-1} + \mathbf{U}_u \mathbf{x}_t + \mathbf{b}_u).
\end{aligned}
$$

They look quite similar: we use a gate u_t between 0 and 1 to combine the knowledge gained from the previous observations and the knowledge gained from the current observation. This example justifies the use of gating in the recurrent units. The same intuitions hold for nonlinear dynamical systems which can be learned by RNNs.

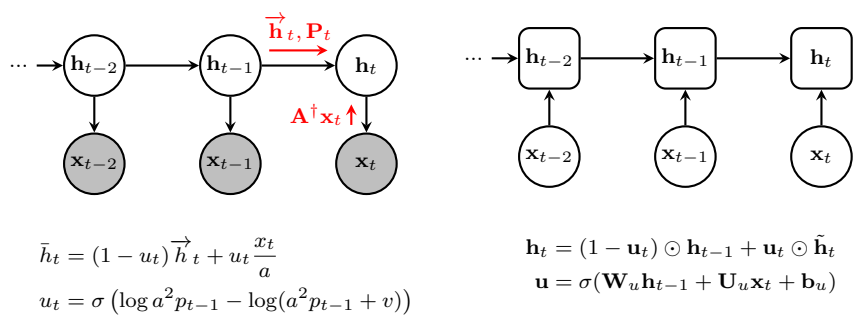


Figure 5.15: The computational graph of an RNN with gating can be seen as implementation of an inference procedure for a probabistic graphical model with sequential data. Left: Message passing in a linear dynamical system. Right: Computational graph of an RNN with gating.

We can view the computational graph of the RNN with gating as an implementation of the inference procedure for a probabilistic graphical model with sequential data (see Fig. 5.15). The direction of connections in the computational graph corresponds to the directions along which messages are sent in the message-passing procedure.

5.4 Long short-term memory (LSTM)

The LSTM (Hochreiter and Schmidhuber, 1997) cell has two states $\mathbf{c}_t$ and $\mathbf{h}_t$. State $\mathbf{c}_t$ is called a cell state, it is updated using gating between the previous state $\mathbf{c}_{t-1}$ and the new state estimate computed using the current input $\mathbf{x}_t$:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \phi_c(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c),$$

where $\mathbf{f}_t \in (0, 1)$ is called the *forget gate* and $\mathbf{i}_t \in (0, 1)$ is called the *input gate*. The gates are computed similarly to the update gate of the GRU:

$$\begin{aligned}\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f)\\ \mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i).\end{aligned}$$

The update of $\mathbf{c}$ is similar to the update of the hidden state of the GRU. The difference is that the LSTM uses two gates $\mathbf{f}_t$ and $\mathbf{i}_t$ while the coefficients in the GRU update rule are $(1 - \mathbf{u}_t)$ and $\mathbf{u}_t$, computed with a single gate $\mathbf{u}_t$.

The second state of the LSTM is computed as a nonlinear transformation of the cell state state $\mathbf{c}_t$:

$$\mathbf{h}_t = \mathbf{o}_t \odot \phi_h(\mathbf{c}_t),$$

where the nonlinearity ϕ_h is typically chosen to be tanh. State $\mathbf{h}_t$ is used as the output of the LSTM cell. This update rule for $\mathbf{h}_t$ contains yet another gate $\mathbf{o}_t$ which controls which states are passed to the output. The gate $\mathbf{o}$ is called the *output gate*, it is computed in the same manner as the other gates:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o).$$

The motivation behind the LSTM design was to mitigate the problem of vanishing and exploding gradients. We can see that

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \operatorname{diag}(\mathbf{f}_t)$$

and if the forget gate $\mathbf{f}_t$ is set to 1, the gradients neither grow nor vanish.

The forget gates $\mathbf{f}_t$ are important for controling how the information (or gradients) propagate through the network. A common intialization of the LSTM weights is such that weights $\mathbf{b}_f$ are set to small random values. This initialization effectively sets the forget gates to $\frac{1}{2}$ and therefore the gradient vanishes with a factor of $\frac{1}{2}$ per timestep. It works well in many problems.

However, sometimes an RNN can fail to learn long-term dependencies. This problem can be addressed by initializing the forget gates $\mathbf{b}_f$ to large values such as 1 or 2. This initialization would increase the values of the forget gates and the gradients will not vanish as fast as with the default initialization (Jozefowicz et al., 2015).

LSTM and GRU have somewhat similar architectures. Can there be even a better architecture of the recurrent unit? Jozefowicz et al. (2015) performed

search of the optimal architecture by constructing the recurrent unit using a set of operations that appear in GRU and LSTM. The performance was tested on a set of standard benchmarks with sequential data. Interestingly, the best architectures were found to be very similar to GRU. For example, one of the best performing unit had the following architecture:

$$\begin{aligned}\mathbf{z} &= \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{b}_z)\\ \mathbf{r} &= \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_t + \mathbf{b}_r)\\ \mathbf{h}_{t+1} &= \tanh(\mathbf{W}_{hh}(\mathbf{r}\odot\mathbf{h}_t) + \tanh(\mathbf{x}_t) + \mathbf{b}_h)\odot\mathbf{z} + \mathbf{h}_t\odot(1-\mathbf{z}).\end{aligned}$$

5.5 A sequence-to-sequence model for statistical machine translation

Statistical machine translation is one of the tasks where RNNs can be used. The task is to translate a sentence in the source language into a sentence in the target language. Each sentence can se seen as a sequence of words and therefore the task is to convert a source sequence into a target sequence (see Fig. 5.16). Models which are trained to do that are called *sequence-to-sequence* models.

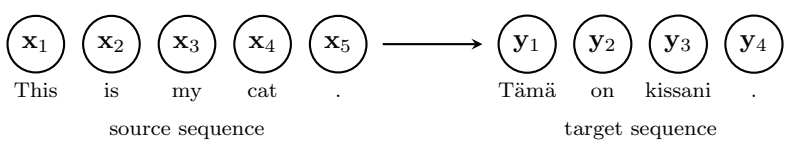


Figure 5.16: The task of statistical machine translation as a sequence-to-sequence modeling task.

Sequence-to-sequence models can be built using recurrent neural networks. The simplest RNN-based model contains two parts: the encoder and the decoder, both implemented with recurrent neural networks (see Fig. 5.17). The two RNNs are different: the encoder RNN and the decoder RNN have different weights. The computational graph in Fig. 5.17 processes one pair of sequences. The encoder RNN converts the input sequence into a single vector $\mathbf{z}_5$ which is the hidden state produced after processing the last element of the input sequence. Next, the decoder RNN takes the output of the encoder $\mathbf{z}_5$ and converts it to the output sequence. The decoder produces one word at every decoding step and it takes as input the previous word in the output sequence.

The decoder has to select the correct word from the dictionary of the target language at every decoding step. This can be seen as a classification problem in which the number of classes is equal to the number of words in the target language. Therefore, we can use the cross-entropy loss as the loss function. The loss is the sum of the cross-entropy losses computed for each sequence n and for each position τ in the target sequence:

$$\mathcal{L} = -\frac{1}{N}\sum_{n}\sum_{\tau=1}^{t_n^{\text{out}}} \log p\left(\mathbf{y}_\tau^{(n)} \,\middle|\, \mathbf{y}_{<\tau}^{(n)}, \mathbf{x}_1, ..., \mathbf{x}_{t_n^{\text{in}}}\right).$$

Since the output of the deoder RNN has to form a proper distribution over all possible words in the target language, we apply the softmax nonlinearity at

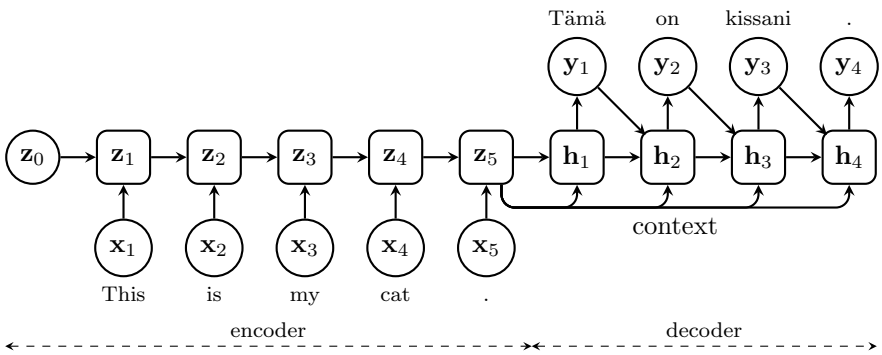


Figure 5.17: An RNN-based sequence-to-sequence model for statistical machine translation.

every position. The input to the softmax can be produced by a linear layer $\mathbf{w}_i$ transforming the hidden states of the RNN:

$$p(\mathbf{y}_\tau = i \,\Big|\, \mathbf{y}_{<\tau}, \mathbf{x}_1, ..., \mathbf{x}_{t_n^{\text{in}}}) = \frac{\exp(\mathbf{w}_i^\top \mathbf{h}_\tau)}{\sum_j \exp(\mathbf{w}_j^\top \mathbf{h}_\tau)}. \tag{5.6}$$

The model can be trained by stochastic gradient descent with backpropagation. Each mini-batch contains a few pairs of source and target sequences. The computational graph shown in Fig. 5.17 is built for all the sequences in the mini-batch.

To generate translations using a trained model, we take the input sequence, encode it with the encoder RNN and then decode it with the decoder RNN. The decoder generates the output sequence by selecting one word at every decoding step. The simplest approach here is to select the most probable word according to the predictive distribution in (5.6). This is the approach that we will take in the home assignment.

However, doing greedy selection at each position often leads to suboptimal results. The reason for that is the fact that the sequence that has the highest probability may be different from the sequence that is produced takin the most probable word at every position. A better algorithm to find the most probably sequence is called *beamsearch* (see, e.g., Cho, 2015).

Note that there is a difference in the way the decoder is used at training and test times. At training time, the decoder uses the correct words as the inputs because they are known. Doing this is often called *teacher forcing* in the RNN literature. At test time, we do not know the correct outputs and therefore we have to feed the decoder's own predictions as the inputs. One potential problem is that during training the decoder does not learn to work in the generation mode (without teacher forcing) which may have negative effect on the performance. To fix this problem, we can toggle the teacher forcing on and off during training. That means that the decoder alternates between using the ground truth words and its own predictions as the inputs during training.

Chapter 6

Attention-based models

6.1 Sequence-to-sequence models with attention

In Section 5.5, we considered an RNN-based sequence-to-sequence model for statistical machine translation. The model contained two recurrent neural networks: the encoder RNN encodes the input sequence into a vector and the decoder decodes that vector into the output sequence.

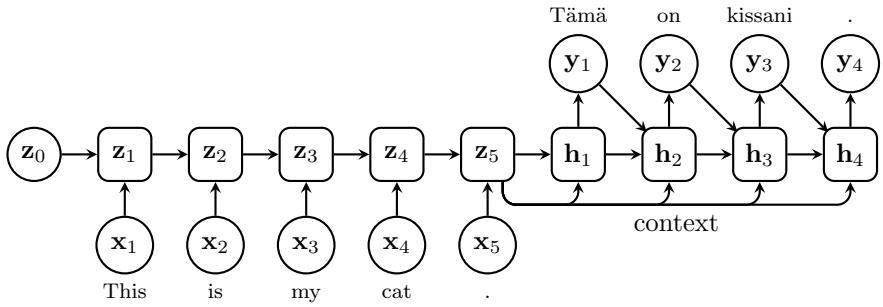


Figure 6.1: An RNN-based sequence-to-sequence model for statistical machine translation.

There is one problem with this model: we encode the whole sentence into a single vector of fixed size. One vector has to represent all information contained in the input sequence and the longer the input sentence is, the more difficult it is to encode all the information in just a single vector. Intuitively, the size of the encoding should depend on the size of the input sequence: the longer the input sentence, the longer the representation should be.

A simple solution is to let the length of our representation be equal to the length of the input sequence. This is illustrated in Fig. 6.2. There are five elements in the input sequence and therefore we use five vectors to represent the input sequence.

How can we implement this idea? One idea is to use the intermediate states of the encoding RNN as the representations (see Fig. 6.3). Technically this is possible because in our simple sequence-to-sequence model, the encoding RNN

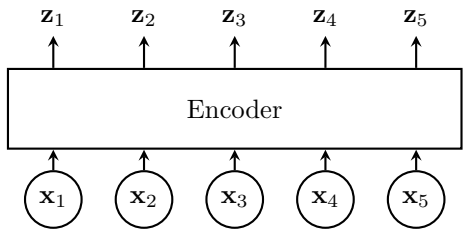


Figure 6.2: An encoder which produces a representation with the same number of elements as the length of the input sequence.

computes state values at every step. However, this idea does not work well in practice. Intuitively, the encoding at a particular position should represent the meaning of the word in that position. And the meaning of a word often depends on the context: what words come before and after that word. The problem with using intermediate states of the encoder RNN as representations is that the representations do not take into account words that come after the current position. For example, the representation at the first position does not depend on words in positions 2, 3, and so on.

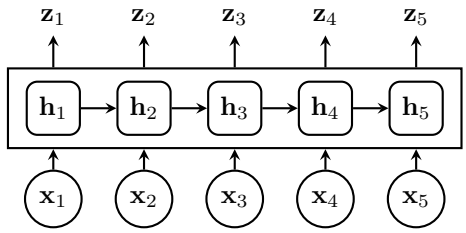


Figure 6.3: An RNN encoder which uses intermediate hidden states as representation does not work well in practice because the representations are not affected by the subsequent words.

Therefore, in the classical sequence-to-sequence model by Bahdanau et al. (2014), the varying-length representation was built using a bi-directional RNN (see Fig. 6.4). The bi-direction RNN does two passes through the input sequence. In the first pass, the input sequence is processed in the conventional way from the beginning to the end. In the second pass, the input sequence is processed in the reverse order: starting from the last element and going backwards. The output of the bi-directional RNN is a concatenation $\mathbf{z}_j = [\overrightarrow{\mathbf{z}_j}; \overleftarrow{\mathbf{z}_j}]$ of the states produced by the two RNNs.

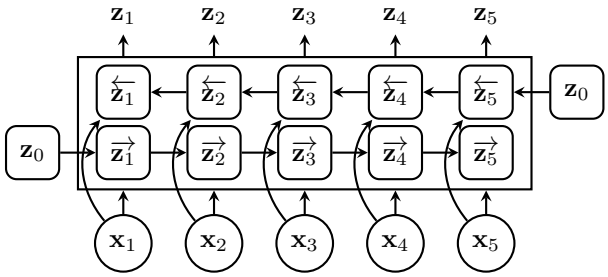


Figure 6.4: A bi-directional RNN used an encoder in the sequence-to-sequence model by Bahdanau et al. (2014).

Consider the decoding in the simple RNN-based model from Fig. 6.1. At every step, the decoder RNN uses the previous hidden state, the previous word

in the output sequence and the representation of the input sequence produced by the encoder, as shown in Fig. 6.5. The last state of the encoder RNN is used as the context for decoding in every step. If we want to use a varying-length representation instead of a single vector (which is vector $\mathbf{z}_5$ in Fig. 6.5), how can we implement that? Our RNN cell has only a single input for the context.

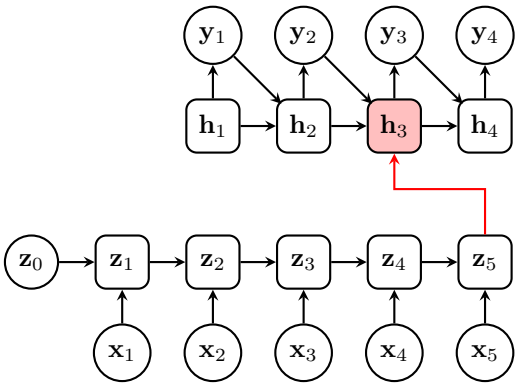


Figure 6.5: Decoding in a simple RNN-based sequence-to-sequence model. The last state of the encoder RNN is used as the context for decoding in every step.

This can be implemented using an attention block. The idea of attention is to select as the context one of the vectors $\mathbf{z}_j$ from the representation produced by the encoder. Once can view the attention block as a multiplexer, a selector that picks one of the input signals and forwards the selected input to the single output. Which one to select, we let the neural network decide by using the *attention* mechanism (see Fig. 6.6).

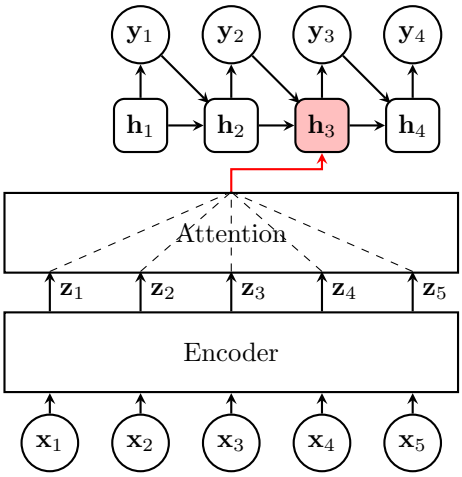


Figure 6.6:

Let us look at the attention mechanism from the pioneering paper by Bahdanau et al. (2014). The output of the attention block is a linear combination

of the inputs with positive coefficients α_j that sum up to 1:

$$\mathbf{c} = \sum_{j=1}^{n} \alpha_j \mathbf{z}_j, \quad 0 < \alpha_j < 1, \quad \sum_{j=1}^{n} \alpha_j = 1.$$

Imagine that one of the coefficients $\alpha_j = 1$ and the remaining ones are equal to zero. In that case, the block copies the j-th input $\mathbf{z}_j$ to the output. The coefficients α are computed by applying the softmax nonlinearity to some values e_j which tell how likely we are to select input j as the output of the attention block:

$$\alpha_j = \frac{\exp(e_j)}{\sum_{j'=1}^{n} \exp(e_{j'})}.$$

This can be viewed as a differentiable implementation of multiplexing.

How should we compute e_j? We let the model decide and therefore we use a generic model

$$e_j = f(\mathbf{h}_{i-1}, \mathbf{z}_j),$$

where function f is modeled with a multilayer perceptron network that takes as inputs the previous decoder state $\mathbf{h}_{i-1}$ and the encoder representation $\mathbf{z}_j$ at position j.

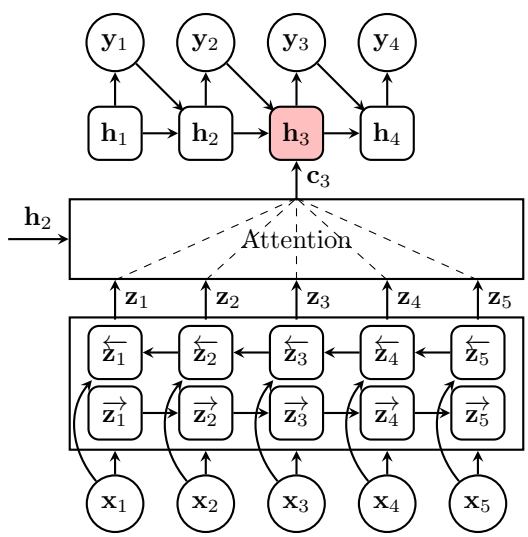


Figure 6.7: The RNN-based sequence-to-sequence model with attention by Bahdanau et al. (2014).

The full architecture from (Bahdanau et al., 2014) is shown in Fig. 6.7. The encoder is a bi-directional RNN that produces representations at every position of the input sequence. The decoder RNN decoder produces the output sequence one element at a time. At every time step, the inputs of the decoder are the hidden state of the decoder, the previous element of the output sequence and the output of the attention block. The switches of the attention block are computed

using the hidden state of the RNN and the encoded representations $\mathbf{z}_j$ of the input sequence.

Table 6.1 shows the translation quality obtained with different models. The quality of translation is typically measured by the BLEU score which basically compares n-grams (subsequences of length n) between the output sequence and the target sequence. In the first row, you can see the performance of the simple RNN-based sequence-to-sequence model without attention. In the second row, you can see the performance of the RNN model with attention. By using larger datasets and an ensemble of models, the performance can be improved even further.

Model	BLEU
Simple Enc-Dec	17.82
Attention-based Enc-Dec	28.45
Attention-based Enc-Dec (LV)	34.11
Attention-based Enc-Dec (LV, ensemble)	37.19

LV - large vocabulary

source: (Jean et al., 2014)

Table 6.1: The BLEU scores of different translation models (large is good).

A nice property of the model from (Bahdanau et al., 2014) is that one can visualize the coefficients α_j that were used by the model in the attention block. In Fig. 6.8, we see two example translations from English to French. The x-axis corresponds to the words in the source sentence and the y-axis corresponds to the generated sequence. We can see that when the network generates an output word, it pays attention to the relevant words in the input sequence.

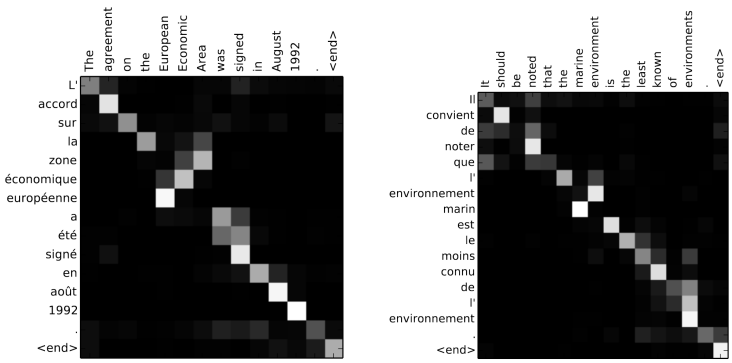


Figure 6.8: The x-axis and y-axis of each plot correspond to the words in the source sentence and the generated translation, respectively.

Models with attention have been very successful in machine translation tasks but they have been used in other domains too. For example, the paper "Show, Attend and Tell" (Xu et al., 2016) considers the task of image captioning: generating a sentence that describes a given image (see Fig. 6.9). The authors treat this problem as a translation task in which the input is an image and the output is a textual description. The architecture of the model is shown in Fig. 6.10. The image is preprocessed into 14×14 feature maps with a convolutional network pre-trained on ImageNet. These feature maps are split into L annotation

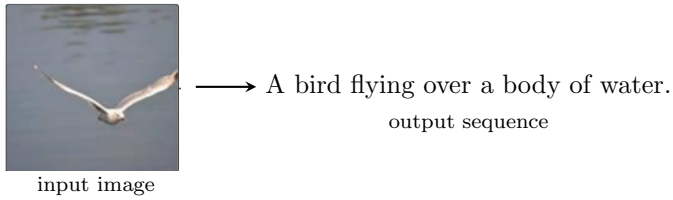


Figure 6.9: The task of image captioning.

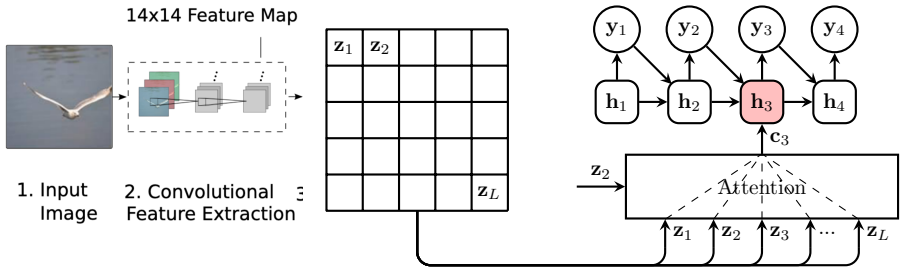


Figure 6.10: "Show, Attend and Tell" model for image captioning (Xu et al., 2016).

vectors $\mathbf{z}_i$ by flattening the feature map. These annotation vectors are used as the context in the decoding RNN. You can see that the model is very similar to the attention-based model for statistical machine translation but the image encoding is done with a pre-trained convolutional network.

6.2 Convolutional sequence-to-sequence models

Consider the RNN-based encoder from the classical (Bahdanau et al, 2014) paper (Fig. 6.11). There are two problems with the RNN-based encoder. The

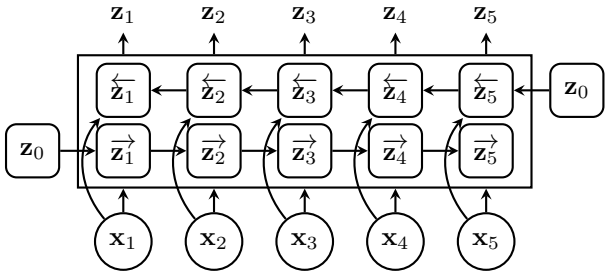


Figure 6.11: An encoder implemented with a bi-directional RNN (Bahdanau et al, 2014).

first problem is that we may need a lot of steps to encode the input sequence. The number of steps is equal to the number of words in the input sentence and we need many steps for long sentences. This may slow down the training procedure. Note that we typically process sentences in mini-batches and the number of steps is determined by the longest sequence in a mini-batch. The second problem is that we need to take multiple steps to model relations between words. For example, in order to model the relation between word 1 and word 5, we need to build five layers of the RNN. And we know that modeling long-term

dependencies can be difficult with RNNs.

Since we know how to decode representations of varying lengths using the attention mechanism, we do not have to use an RNN to implement the encoder. The encoder can be any network that converts input sequence $(\mathbf{x}_1, ..., \mathbf{x}_n)$ into representations $(\mathbf{z}_1, ..., \mathbf{z}_n)$. Gehring et al. (2017) proposed to use a convolutional network (CNN) to encode input sequences (Fig. 6.12).

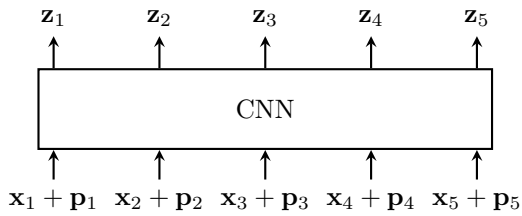


Figure 6.12: The encoder in the convolutional sequence-to-sequence model by Gehring et al. (2017). + denotes summation of position and word embeddings.

Convolutional layers can process sequences of varying lengths due to parameter sharing. That is why they can be used to encoder sentences in the machine translation task. Since the input is a sequence and it has one-dimensional structure, we use one-dimensional convolutions inside such an encoder.

The advantage of the convolutional encoder is that it can compute representations in all positions in parallel. We do not need to loop over the elements of the input sequence like in RNNs. Another advantage the encoding is affected by both preceding and subsequent positions. For example, the representation at position 3 will depend on the inputs at positions 2 and 4. Recall that in the bi-directional RNN, one RNN used only preceding elements and the other one used only subsequent elements.

One disadvantage of the convolutional encoder is that it does not take into account the position of the word. The representation in the current position only depends on the input in the receptive field but it does not depend on whether that input appears at the beginning or at the end of the sequence. This happens due to translation equivariance of convolutional layers. Gehring et al. (2017) fixed that problem by adding position embeddings. Position embedding are computed similarly to word embeddings: instead of encoding words, we encode positions 1, 2, 3 and so on as $\mathbf{p}_j$. The position embeddings are added to word embeddings and the sum is fed to the convolutional encoder (see Fig. 6.12).

We have now got rid of the RNNs in the encoder. Can we also get rid of the RNN in the decoder? Recall that the decoder is an autoregressive model which predicts the next word given the previous words and the context $\mathbf{z}$ produced by the encoder:

$$\mathbf{y}_i = f(\mathbf{y}_{i-1}, ..., \mathbf{y}_1, \mathbf{z}_1, ..., \mathbf{z}_n). \tag{6.1}$$

We can model function f using a convolutional neural network but we need to make sure that we preserve the autoregregressive structure of the model as given by (6.1). We can achieve this using the following trick. The input and the desired output of the convolutional decoder are two shifted versions of the target sequence $\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_{t^{\text{out}}}$ (see Fig. 6.13). The input is the target sequence that

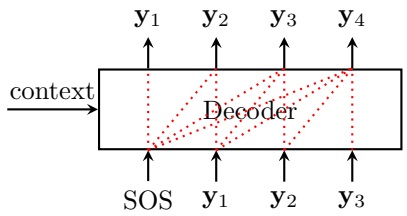


Figure 6.13: A convolutional decoder. Inputs and outputs are same sequences but the output is shifted by one position and the input sequence starts with a SOS token. The receptive field in each position does not contain subsequent elements of the input.

starts with a special start-of-sentence (SOS) token: SOS, $\mathbf{y}_1, ..., \mathbf{y}_{t^{\text{out}}-1}$. The desired output is the target sequence that starts with the first word $\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_{t^{\text{out}}}$. Next, we construct the CNN such that the output in position i is not affected by subsequent elements $i' \geq i$ of the input. This is illustrated by the red dotted lines in Fig. 6.13. For example, to predict the first word $\mathbf{y}_1$, we use only the SOS token, to predict the second word $\mathbf{y}_2$, we use the SOS token and the first word $\mathbf{y}_1$ of the target sentence and so on. If we construct our CNN in that way, then we have a proper auto-regressive model and we can use it for generating output sequences.

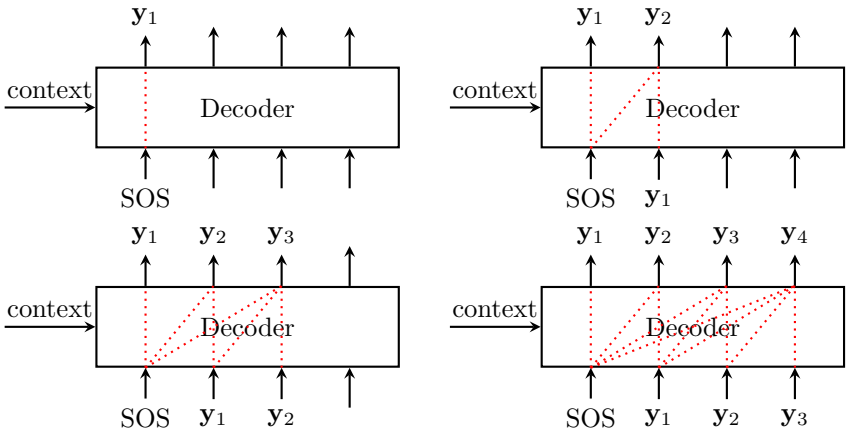


Figure 6.14: Using a trained decoder to generate an output sequence.

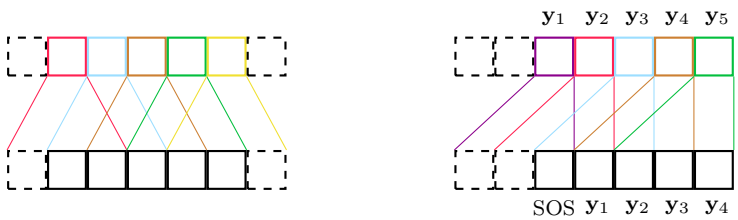
Let us see how we can use a decoder with the described structure to generate output sequences. We generate an output sequence one element at a time. First, we input only the SOS token in the first position and predict the first word $\mathbf{y}_1$ (see Fig. 6.14a). Then, we take the predicted first word $\mathbf{y}_1$ and place it to the second position in the input. Now we use the decoder to predict the second word $\mathbf{y}_2$ (Fig. 6.14b). We place the second word to the input $\mathbf{y}_2$ and predict the third word and so on.

A big advantage of the convolutional decoder compared to the RNN is that during training, we can compute output elements for all positions *in parallel*. Recall that in the RNN decoder, we had to produce the output sequence one element at a time.

Now we need to construct our convolutional decoder in such a way that the receptive field in each position does not contain subsequent elements. We can

achieve the desired effect by using *shifted convolutions*.

Let us first look at the standard convolutional layer with kernel size 3. Since we process sequences (inputs with one-dimensional structure), we use one-dimensional convolutions. We use padding of size 1 and the produced pixels in the output are shown with the colored squares in Fig. 6.15a. If we want to stack more layers, we again use padding and process the output in the same way.



(a) standard convolution (b) shifted convolution

Figure 6.15: Standard versus shifted one-dimensional convolutions.

In shifted convolutions, instead of padding the input sequence from both ends, we add two elements only at the beginning of the input sequence and process the input with the same convolutional layer. We also shift the output such that the green pixel corresponds to the last element (Fig. 6.15b). We can see that some of the pixels in the output will have the same receptive field as in the standard convolution. This is shown by using the same colors in Fig. 6.15. However, when we shift the output in this way, none of the pixels in the output layer uses information from subsequent positions. In all the pixels, we only use information in the same position or the information before that. This is often called *shifted convolutions* or *causal convolutions*.

If we feed the target sequence with the SOS token and we use as the target sequence the sequence which starts with the first word, we get the desired autoregressive structure. We can stack multiple convolutional layers in the same way and this will keep the desired autoregressive property.

Now we need to solve the next problem: how use the context provided by the encoder in the convolutional decoder. Gehring et al. (2017) used an attention block which is illustrated in Fig. 6.16. The block has two kinds of inputs: the outputs $\mathbf{h}_i$ of the previous layer of the decoder and slightly modified outputs $\mathbf{z}'_j$ of the encoder. The two sequences have different lengths: the number of inputs $\mathbf{h}_i$ is equal to the number of elements in the target sequence, the number of inputs $\mathbf{z}'_j$ is equal to the number of elements in the source sequence.

The output of the attention block is produced similarly to the attention block in the RNN-based sequence-to-sequence model. At each position i, we select one of the positions of the encoder output using weights α_{ij}:

$$\mathbf{o}_i = \sum_{j=1} \alpha_{ij} \mathbf{z}'_j.$$

The weights α_{ij} are positive values between 0 and 1 and they are computed

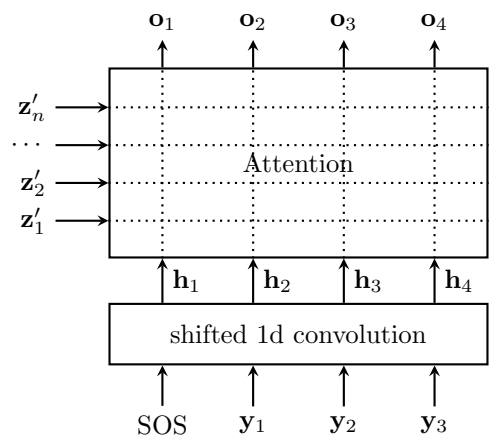


Figure 6.16: The decoder block with attention in the convolutional sequence-to-sequence model (Gehring et al., 2017).

using the softmax nonlinearity:

$$\alpha_{ij} = \frac{\exp(\mathbf{h}_i^\top \mathbf{z}_j)}{\sum_{j'=1}^{n} \exp(\mathbf{h}_i^\top \mathbf{z}_{j'})}. \tag{6.2}$$

One difference of this attention block compared to the one described in Section 6.1 is that it selects the modified outputs of the decoder (instead of $\mathbf{z}_j$):

$$\mathbf{z}'_j = \mathbf{z}_j + \mathbf{x}_j + \mathbf{p}_j,$$

where $\mathbf{x}_j$ are word embeddings and $\mathbf{p}_j$ are position embeddings of the j-th element in the source sequence. This turns out to work better.

In the formula (6.2) for the attention weights, the inputs to the softmax nonlinearity are the dot products between the outputs $\mathbf{h}_i$ of the previous decoder layer and the outputs $\mathbf{z}_j$ of the encoder. If vectors $\mathbf{h}_i$ and $\mathbf{z}_j$ were normalized to unit variance, then the dot product would be equal to the cosine distance between the vectors. Therefore, one can view these computations as doing pairwise comparison between vectors $\mathbf{h}_i$ and $\mathbf{z}_j$ and the encoder output that has the greatest similarity with the input of the attention block in the current position i.

Gehring et al. (2017) use multiple decoder blocks (shown in Fig. 6.16) which are stacked on top of each other. Each decoder block first processes inputs with shifted one-dimensional convolutions and then it attends to the outputs of the encoder with an attention block. The decoder also has connections skipping the attention block (not shown in Fig. 6.16).

The full architecture of the model is shown in Fig. 6.17. At the top, there is a convolutional encoder with standard one-dimensional convolutions (there is no need to use shifted convolutions in the encoder). The convolutional decoder shown in the bottom left corner processes the target sequence with shifted one-dimensional convolutions. The attention block in the middle combines the signals from the decoder and the encoder. The skip connections are shown with an arrow which starts at the decoder output and skips the attention block.

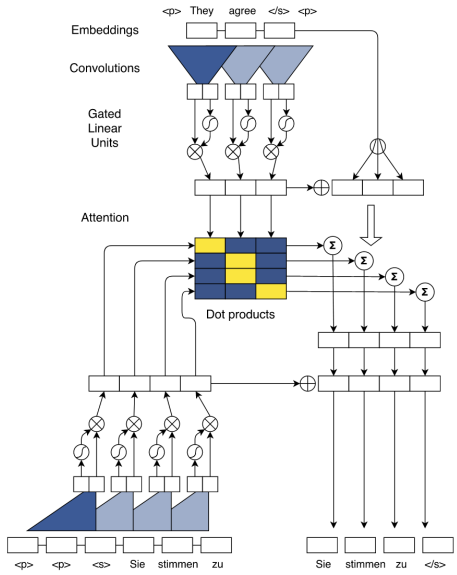


Figure 6.17: Convolutional sequence-to-sequence model by Gehring et al. (2017). At the top, there is a convolutional encoder with standard one-dimensional convolutions. The convolutional decoder in the bottom left corner processes the target sequence with shifted one-dimensional convolutions. The attention block in the middle combines the signals from the decoder and the encoder. The skip connections are shown with an arrow which starts at the decoder output and skips the attention block.

Table 6.2 shows that the convolutional sequence-to-sequence model outperforms the classical RNN model with attention.

Model	BLEU
Simple Enc-Dec	17.82
Attention-based Enc-Dec	28.45
Attention-based Enc-Dec (LV)	34.11
Attention-based Enc-Dec (LV, ensemble)	37.19
ConvS2S (BPE 40K)	**40.51**

Table 6.2: The translation performances on an English-to-French translation task (WMT'14).

6.3 Transformer

Transformer (Vaswani et al., 2017) is one of the hottest deep learning models at the moment. The architecture of the transformer, as shown in Fig. 6.18, is quite similar to the convolutional sequence-to-sequence model that we considered in Section 6.2. The encoder converts the source sequence $(\mathbf{x}_1, ..., \mathbf{x}_n)$ into continuous representations $(\mathbf{z}_1, ..., \mathbf{z}_n)$. The decoder uses two shifted versions of the target sequence as the input and the target. The decoder processes all positions in parallel preserving the autoregressive structure by special means. The decoder attends to representations $(\mathbf{z}_1, ..., \mathbf{z}_n)$ using an attention block.

6.3.1 Multi-head attention mechanism

The attention mechanism is central in this model and therefore we first explain the structure of the attention block in the decoder. This block is shown in

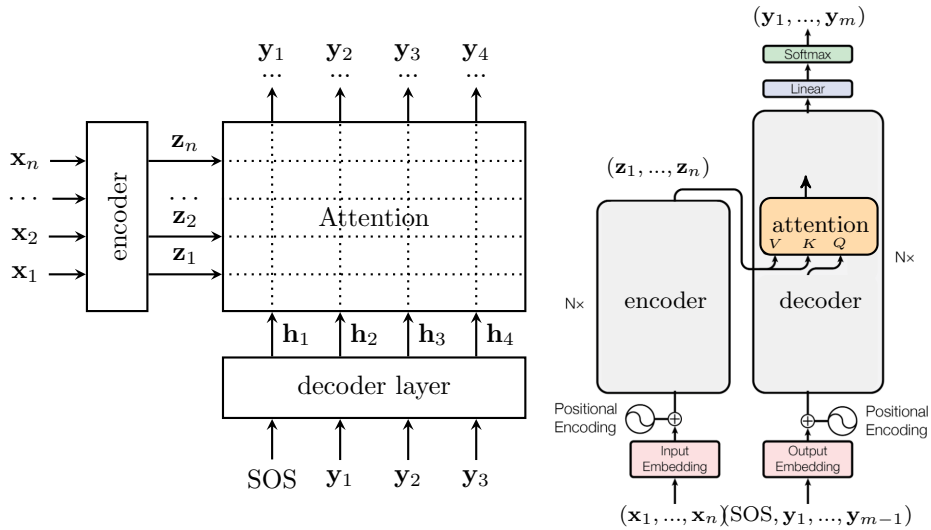


Figure 6.18: The architecture of the transformer (Vaswani et al., 2017). Left: Our schematic representation. Right: The diagram in the style of the original paper.

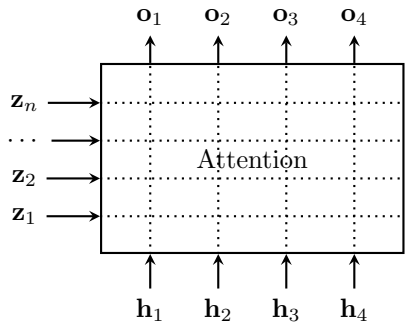


Figure 6.19: The attention block in the transformer decoder.

the center of Fig. 6.18 and we visualize it separately in Fig. 6.19. Recall the intuition of the attention mechanism in the ConvS2S model: the attention compares intermediate representations $\mathbf{h}_i$ developed in the decoder with the encoder outputs $\mathbf{z}_j$ and selects the encoder output that is closest to $\mathbf{h}_i$.

The basic attention mechanism in transformers is very similar: we select one of the encoder outputs $\mathbf{z}_j$ using coefficients α_{ij}

$$\mathbf{o}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{z}_j$$

and the coefficients are computed using the softmax function applied to the dot products between vectors $\mathbf{z}_j$ and $\mathbf{h}_i$:

$$\alpha_{ij} = \frac{\exp(\mathbf{z}_j^\top \mathbf{h}_i / \sqrt{d_k})}{\sum_{j'=1}^{n} \exp(\mathbf{z}_{j'}^\top \mathbf{h}_i / \sqrt{d_k})}.$$

Thus, the similarity metric used to compare vectors $\mathbf{z}_j$ and $\mathbf{h}_i$ is the dot product product scaled by the square root of the dimensionality d_k of vectors $\mathbf{z}_j$, $\mathbf{h}_i$. The authors call this mechanism *scaled dot-product attention*.

We can think of the scaled dot-product attention as finding values $\mathbf{v}_j = \mathbf{z}_j$ that correspond to keys $\mathbf{k}_j = \mathbf{z}_j$ that are closest to the query vector $\mathbf{q}_i = \mathbf{h}_i$:

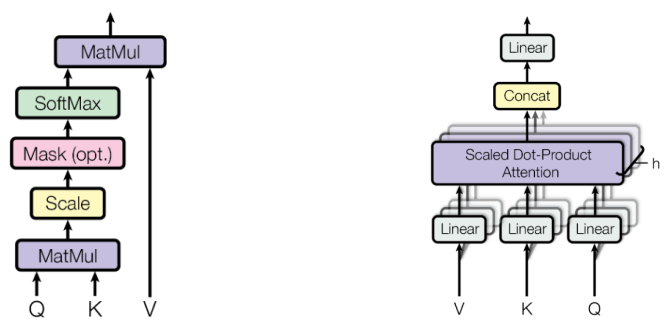
$$\mathbf{o}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j, \quad \alpha_{ij} = \frac{\exp(\mathbf{k}_j^\top \mathbf{q}_i/\sqrt{d_k})}{\sum_{j'=1}^{n} \exp(\mathbf{k}_{j'}^\top \mathbf{q}_i/\sqrt{d_k})} \tag{6.3}$$

where $\mathbf{v}_j = \mathbf{z}_j$, $\mathbf{k}_j = \mathbf{z}_j$ and $\mathbf{q}_i = \mathbf{h}_i$. The query values come from the previous block of the decoder while the keys and values are the outputs of the encoder. Note that the encoder outputs are used both as the keys and values.

Vaswani et al. (2017) write the scaled dot-product attention in the matrix notation which is equivalent to (6.3):

$$\text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

with $\mathbf{V} \in \mathbb{R}^{m \times d_v}$, $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{m \times d_k}$. Fig. 6.20a shows the diagram of the scaled dot-product attention.



(a) scaled dot-product attention (b) multi-head attention

Figure 6.20: Attention blocks used in the transformer model.

In the scaled dot-product attention mechanism, every position of the decoder can attend to only one position of the encoder output. Is it benefecial to allow the decoder to attend to multiple positions?

The authors of the transformer model find that attending to multiple positions can bring benefits and they propose to implement this idea in the following way. There are multiple attention heads that use different keys, values and queries in (6.3) in every position:

$$\mathbf{k}_j^{(h)} = \mathbf{W}_h^K \mathbf{z}_j, \quad \mathbf{v}_j^{(h)} = \mathbf{W}_h^V \mathbf{z}_j, \quad \mathbf{q}_i^{(h)} = \mathbf{W}_h^Q \mathbf{h}_i.$$

The keys, values and queries of the attention head h are produced by projecting the inputs of the attention block to lower-dimensional vectors using matrices $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V$. Since matrices $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V$ are trainable parameters, each

attention head can learn to attend to different positions. The outputs produced by different attention heads are concatenated and their linear transformation is passed to the output. The authors call this mechanism *multi-head attention*. It is illustrated in Fig. 6.20b. It is the multi-head attention that is used as the attention mechanism in the transformer model.

Using matrix notation, the multi-head attention can be written in the following form:

$$\begin{aligned}\text{MultiHead}(\mathbf{Q},\mathbf{K},\mathbf{V}) &= \text{concat}(\mathbf{o}_1,...,\mathbf{o}_H)\mathbf{W}^O\\ \mathbf{o}_h &= \text{attention}(\mathbf{Q}\mathbf{W}_h^Q,\mathbf{K}\mathbf{W}_h^K,\mathbf{V}\mathbf{W}_h^V)\end{aligned}$$

where $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V, \mathbf{W}^O$ are the parameters of the attention block.

6.3.2 Transformer encoder

Next we consider the transformer encoder. We have previously used different types of encoders: a bi-directional RNN or a convolutional network. The idea of the transformer model is to use the multi-head attention as the main information processing unit in both the encoder and the decoder. This explains the title of the paper: "Attention is all you need".

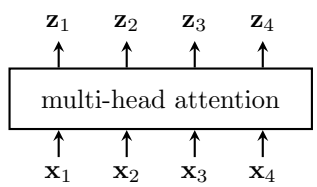


Figure 6.21: Multi-head attention as the main information processing unit of the transformer encoder.

The encoder needs to develop representations $\mathbf{z}_i$ in each position of the source sequence (see Fig. 6.21). We can do this using the same multi-head attention

$$\mathbf{Z} = \text{MultiHead}(\mathbf{X},\mathbf{X},\mathbf{X})$$

where matrices $\mathbf{Z}$ and $\mathbf{X}$ contain vectors $\mathbf{z}_j$, $\mathbf{x}_j$ in their rows. If we assume for simplicity that the attention block contains only one scaled dot-product attention, then the output is given by

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{x}_j \qquad \alpha_{ij} = \frac{\exp(\mathbf{x}_j^\top \mathbf{x}_i/\sqrt{d_k})}{\sum_{j'=1}^{n}\exp(\mathbf{x}_{j'}^\top \mathbf{x}_i/\sqrt{d_k})}.$$

Note that vectors $\mathbf{x}_i$ as use as keys, values and queries. The authors call this mechanism *self-attention* because we attend to the elements of the same sequence.

The advantage of self-attention is that the representation in any position is affected by all positions of the input after one layer of self-attention. Recall that to achieve the same effect with an RNN encoder, we have to process the whole sequence with as many steps as there are positions in the input sequence.

In the convolutional encoder, we have to stack multiple convolutional layers to achieve a similar effect. In contrast, one block of self-attention is enough to start modeling relations between any pair of input elements.

One block of the transformer encoder is shown in Fig. 6.22. The outputs of the self-attention block are processed by a mini-MLP which is shown with the "Feed Forward" block in the figure: each position is processed independently by the same MLP. The encoder block also contains standard deep learning tricks such as skip connections and layer normalization. The encoder is a stack of multiple blocks with the structure presented in Fig. 6.22.

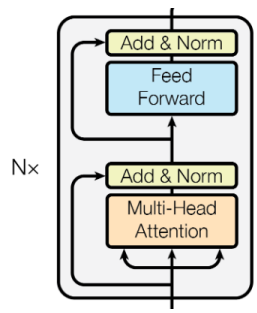


Figure 6.22: One block of the transformer encoder.

6.3.3 Transformer decoder

The transformer decoder implements an autoregressive model

$$\mathbf{y}_i = f(\mathbf{y}_{i-1}, ..., \mathbf{y}_1, \mathbf{z}_1, ..., \mathbf{z}_n)$$

similarly to the RNN-based or the convolutional sequence-to-sequence model. We use the same idea as in the convolutional decoder: two shifted versions of the target sequence are used as the input and as the target. Similarly to the convolutional decoder, we need to preserve the autoregressive structure as illustrated in Fig. 6.23.

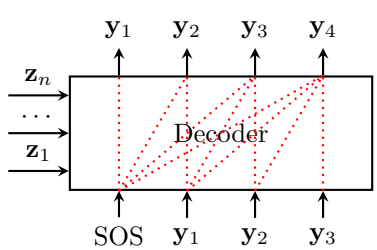


Figure 6.23: Preserving the autoregressive structure in the transformer decoder. When predicting word $\mathbf{y}_i$ we can use the preceding words $\mathbf{y}_1, ..., \mathbf{y}_{i-1}$ but not subsequent words $\mathbf{y}_i, ..., \mathbf{y}_m$.

The decoder also uses multi-head attention as the main information processing unit. This is illustrated in Fig. 6.24. The cross-attention block there is the block that we described in Section 6.3.1. It combines the representations $\mathbf{h}_i$ in every decoder position with the outputs $\mathbf{z}_j$ of the encoder. The other block in

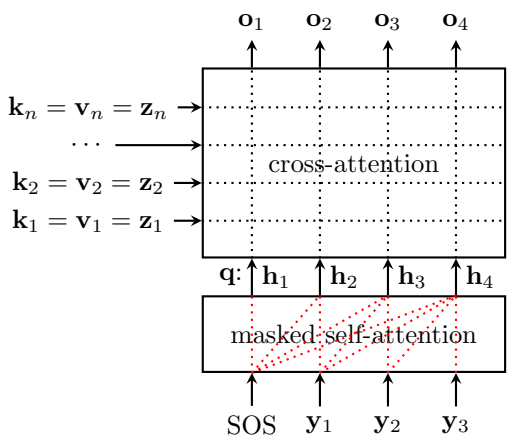


Figure 6.24: A simplified diagram of one block of the decoder transformer. The illustration omits such details as skip connections, layer normalization and multilayer perceptron (see Fig. 6.25 for a more complete diagram).

that diagram is the multi-head self-attention which processes the inputs of the decoder. Again, we see the implementation of the attention-is-all-you-need idea.

In order to create a proper autoregressive model, we need to make sure that the output of the decoder in any position i is not affected by the subsequent inputs $i+1, \ldots$. This is obtained using *masked self-attention*. Let us assume for simplicity that the self-attention block is implemented by the scaled dot-product attention and $\mathbf{v}_i$, $\mathbf{h}_j$ denote the inputs and the outputs of the self-attention layer:

$$\mathbf{h}_i = \sum_{j=1}^{m} \alpha_{ij} \mathbf{v}_j \qquad \alpha_{ij} = \frac{\exp(\mathbf{v}_j^\top \mathbf{v}_i / \sqrt{d_k} + m_{ij})}{\sum_{j'=1}^{m} \exp(\mathbf{v}_{j'}^\top \mathbf{v}_i / \sqrt{d_k} + m_{ij'})}.$$

To control which inputs can affect the output in a given position, we add attention masks m_{ij} to the inputs of the softmax function. If mask m_{ij} is set to $-\infty$, input $\mathbf{v}_j$ does not affect the value of $\mathbf{h}_i$ because $\alpha_{ij} = 0$. Therefore, we can guarantee the desired autoregressive structure by using masks:

$$m_{ij} = 0\,, \text{ if } j \leq i$$
$$m_{ij} = -\infty\,, \text{ if } j > i.$$

The remaining elements of one block of the transformer decoder are shown in Fig. 6.25. After the masked self-attention and the cross-attention, the representations in each position are processed with a mini-MLP shown with the "Feed Forward" block in the figure. This decoder block also contains skip connections and layer normalization. The decoder is a stack of several blocks with this structure.

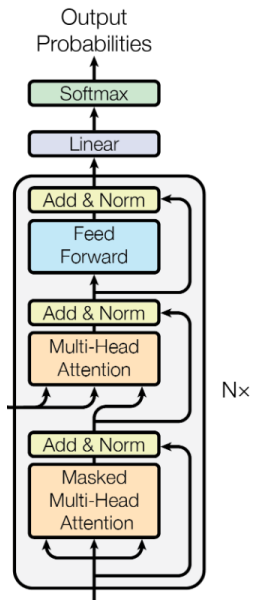


Figure 6.25: Transformer decoder.

6.3.4 Positional encoding of transformer

The final detail of the transformer is how to take into account the order of elements in the processed sequences. Consider the scaled dot-product attention:

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij}\mathbf{x}_j \qquad \alpha_{ij} = \frac{\exp(\mathbf{x}_j^\top \mathbf{x}_i/\sqrt{d_k})}{\sum_{j'=1}^{n}\exp(\mathbf{x}_{j'}^\top \mathbf{x}_i/\sqrt{d_k})}.$$

If we change the order of the inputs $\mathbf{x}_i$, the order of the outputs will change in the same way, which is illustrated in Fig. 6.26. This means that the self-attention layer is equivariant to permutations of the input and that the computed representations do not depend on the order of the elements in the input sequence. This is not desired because the order of the words is important for understanding the meaning of a sentence.

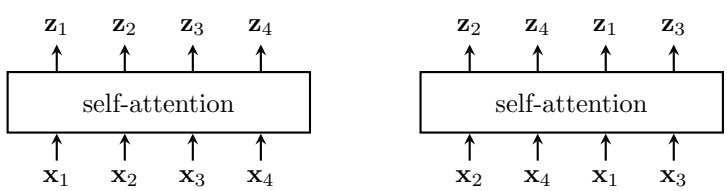


Figure 6.26: The self-attention layer is equivariant to permutations of the inputs. If the order of the inputs $\mathbf{x}_i$ changes, the order of the outputs $\mathbf{z}_i$ changes in the same way.

To address this problem, the positions of the inputs are encoded to vectors $\mathbf{p}_i$ and they are added to the word embeddings at the input of the encoder and the decoder. This is similar to the convolutional model (see Fig. 6.12) where the sum $\mathbf{x}_j + \mathbf{p}_j$ was used as the input.

The classical transformer uses hard-coded positional encoding, which means the encoding does not contain trainable parameters. Encoding $\mathbf{p}_i$ for position i consists of the following values:

$$\{\sin(i/10000^{2l/d}),\ \cos(i/10000^{2l/d}),\ l = 1, ..., d\},$$

where d is the length of the enconding. The motivation behind this encoding is that it makes it easy for the model to learn to attend by relative positions (Vaswani et al., 2017). The graphical illustration of the positional encoding is presented in Fig. 6.27.

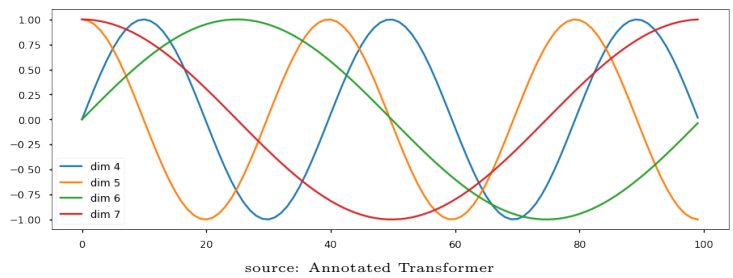


Figure 6.27: Positional encoding of transformer for $d = 20$. Different curves represent one dimension l as a function of the position i.

Training of the transformer model typically needs a ramp-up of the learning rate. That means that training starts with a small learning rate, the learning rate is gradually increased and then decreased. The learning rate schedule proposed by Vaswani et al. (2017) is shown in Fig. 6.28.

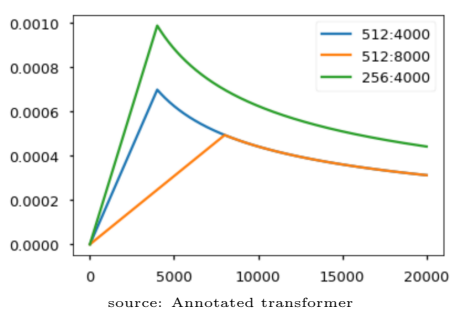


Figure 6.28: Learning rate schedule proposed by Vaswani et al. (2017).

Table 6.29 shows the translation performances in an English-to-French translation task. The performance of the transformer model is very good compared to other models. If you have trouble understanding the transformer model, check out the Annotated Transformer blog post.

Model	BLEU
ConvS2S	40.46
ConvS2S (ensemble)	41.29
Transformer (base model)	38.1
Transformer (big)	**41.8**

Figure 6.29: The translation performances on an English-to-French translation task (WMT'14) according to (Vaswani et al., 2017).

6.4 BERT: Transformer-based language model

Suppose we have a custom task that requires natural language understanding (see, e.g., tasks from the GLUE benchmark). For example, we want to classify messages in a slack workspace as positive, neutral or negative (a task called sentiment analysis). We can extract messages from the slack workspace and label some of them. Our dataset would be pretty small: it would contain a few thousand messages, a few hundred of them would be labeled because labeling is a tedious and time consuming procedure. If we train a sentiment analyzer using only this small data set, we are very likely to overfit and the performance of our model will probably be not very good.

How can we achieve better performance? A popular solution is to pre-train a model on a similar task in which one can collect a lot of data and use the pre-trained model as initialization for our custom machine learning problem. This approach is called *transfer learning*.

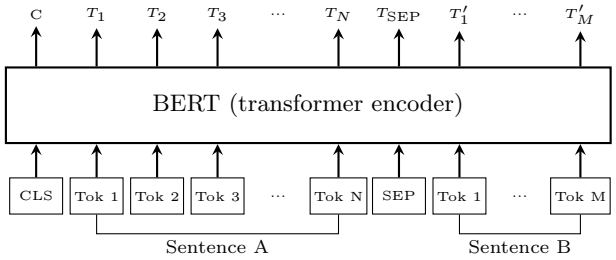


Figure 6.30: BERT

BERT (Devlin et al., 2018) is a model pre-trained on a large collection of texts to solve artificially created machine learning tasks. The BERT model is essentially a transformer encoder (see Fig. 6.30). The model is trained using as inputs either one or two sentences extracted from text corpora. One or two inputs sentences are used to prepare the model to different natural language processing tasks. In tasks like sentiment analysis, the model needs to classify an individual sentence and therefore the input contains only one sentence. In tasks like question answering, the model gets as inputs a question and a paragraph which contains the answer to the question. When two sentences are fed to the input of BERT, the sentences are separated by a special token denoted as SEP in Fig. 6.30.

BERT is trained on a large corpus of texts, for example, English Wikipedia which contains more than 2 billion of words. These text datasets do not contain any label information that could be used to solve a supervised learning problem. Therefore, BERT is trained on two artificially created machine learning tasks.

The first task is to predict a token that is removed from the input sentence. During training, we randomly replace one of the input tokens with a special MASK token (see Fig. 6.31). The model is then trained to reconstruct the masked token in the corresponding output. The intuition is that in order to solve this task, the model needs to understand what words typically appear in a given context. Therefore, this tasks encourages building a statistical model of

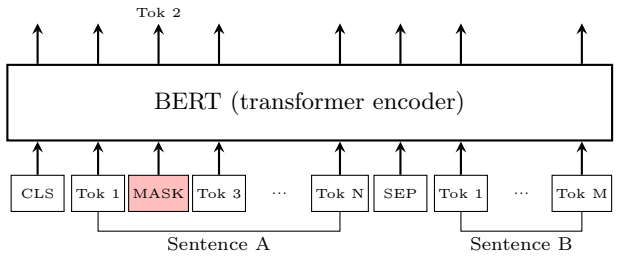


Figure 6.31: The first pre-training task of BERT is to predict a token masked in the input sentence.

the language.

The second pre-training task is to predict whether sentence B follows sentence A (see Fig. 6.32). During training, the inputs are created such that

- 50% of the time sentence B follows sentence A in the corpus
- 50% of the time sentence B is randomly chosen.

This is a binary classification task and the model output is taken from the position that corresponds to a special input token marked as CLS.

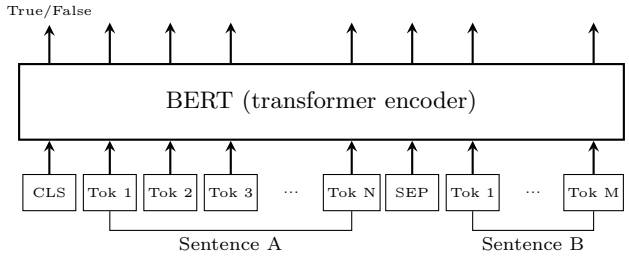


Figure 6.32: The second pre-training task of BERT is to predict whether sentence B follows sentence A.

The pre-trained BERT model can be fine-tuned to our custom natural language processing problem in the following way. Suppose that we solve a sentiment analysis problem and our custom dataset consists of the following training examples:

> **Sentence:** Although the value added services being provided are great but the prices are high. **Class:** mixed review
>
> **Sentence:** Great work done #XYZ Problem resolved by customer care in just one day. **Class:** postive review

We feed the input sentence as Sentence A in the BERT model and Sentence B is kept empty. The output of the model is taken from the position that corresponds to the special CLS token (see Fig. 6.33). The target for the model output is the target class of the input sentence. Since the number of classes depends on our custom classification problem, we introduce a new layer that converts the output of BERT into class probabilities. Then, we fine-tune all the parameters of the model.

Fine-tuning the full BERT model that contain a huge number of parameters to our custom dataset that may be small sounds like a bad idea because of the

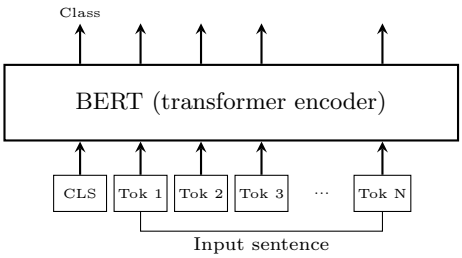


Figure 6.33: Fine-tuning of BERT on a sentiment analysis task.

risk of overfitting. However, it turns out to work well in practice. The intuition is that the weights of BERT are intialized very close to a good solution for our custom problem. It is much easier for the model to find that good solution during fine-tuning than to find an overfitted solution that does not generalize well.

A question answering task is often formulated as finding the answer to a given question in a given paragraph. A typical training set consists of examples in which the words that contain the correct answer are labeled (marked with the blue color in the paragraph below).

> **Paragraph:** Beyoncé Giselle Knowles-Carter (born September 4, 1981) is an American singer, songwriter, record producer, dancer and actress. Born and raised in Houston, Texas, Beyoncé performed in various singing and dancing competitions as a child. She rose to fame in the late 1990s as the lead singer of Destiny's Child, one of the best-selling girl groups of all time.
>
> **Question**: When did Beyonce start becoming popular?
> **Correct answer**: in the late 1990s

To fine-tuning BERT for a question answering task, the question is fed as Sentence A and the paragraph (typically called passage) is fed as Sentence B (see Fig. 6.34). The output of the model is interpreted as the probability that the token in the corresponding position is the start of the answer and the probability that the corresponding token is the end of the answer. Again, all the model parameters are updated in the fine-tuning stage.

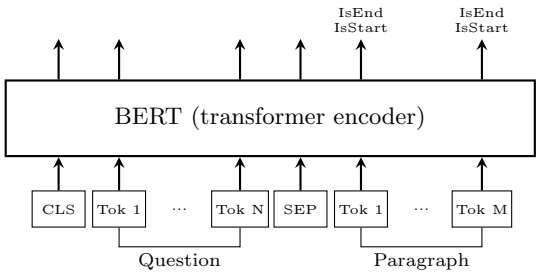


Figure 6.34: Fine-tuning of BERT on a question answering task.

Table 6.3 shows the performance of the fine-tuned BERT on a set of natural language understanding tasks from the GLUE benchmark. One can see that BERT performs very well.

Table 6.3: Performance of models on tasks from the GLUE benchmark. The number below each task denotes the number of training examples. The reported scores are: F1 scores for QQP and MRPC, Spearman correlations for STS-B, and accuracy for the other tasks.

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

6.4.1 Vision Transformer (ViT)

Although originally introduced for natural language processing tasks, transformers have now been used in many other domains and they show great performance. One example is the Vision Transformer (ViT) which is a transformer-based architecture for image classification tasks (Dosovitskiy et al., 2020). The model is conceptually very close to the standard transformer, with the following outline:

- split an image into fixed-size patches
- linearly embed each of them
- add position embeddings
- feed the resulting sequence of vectors to a standard Transformer encoder.

In order to perform classification, ViT uses the same trick as in BERT: add an extra learnable "classification token" to the sequence and use the output in that position as the model output.

ViT is typically pre-trained on large datasets and then fine-tuned to (smaller) downstream tasks. ViT attains excellent results compared to state-of-the-art convolutional networks.

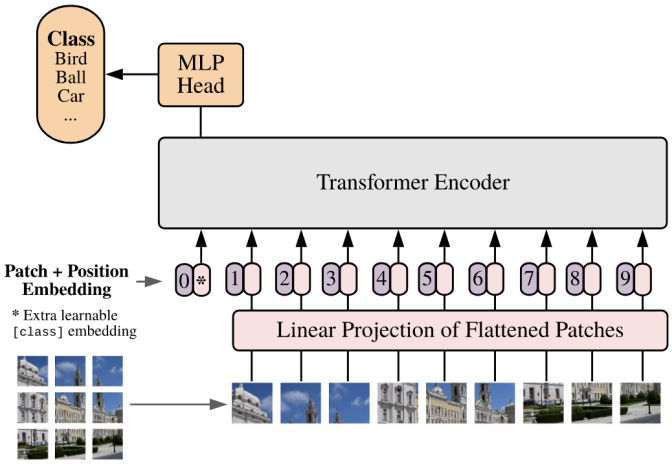


Figure 6.35: The outline of the Vision Transformer model (Dosovitskiy et al., 2020).

Chapter 7

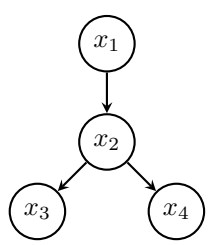
Graph neural networks

7.1 Motivation

In the previous chapters, we looked at neural network architectures that were designed to process different types of inputs. When the inputs were vectors whose elements did not have order, we used multi-layer perceptrons. When inputs had one-dimensional or two-dimensional spatial structure, we used convolutional networks. When inputs were sequences with varying lengths, we used recurrent neural networks or transformers.

There are many machine learning problems in which the inputs can be represented as a graph. An example of a graph is shown in Fig. 7.1, it is formally defined as a tuple $G = (u; V; E)$ that consists of

- a global attribute u, for example, a real-valued vector,
- a set of nodes V, such that each node is described by a vector of attributes x_i,
- a set of edges E, such that an edge between nodes k and j is described by a vector of attributes e_{kj}.



$$V = \{x_1, x_2, x_3, x_4\}$$
$$E = \{e_{12}, e_{23}, e_{24}\}$$

Figure 7.1: Example of a graph.

The first example of a machine learning problem with graphs is the task of predicting the chemical properties of molecules (Duvenaud et al., 2015; Gilmer et al., 2017). A molecule can be represented as a graph (see Fig. 7.2) in which the global attribute u can be some known property of a molecule (for example,

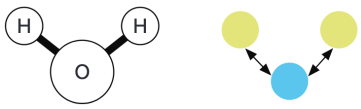
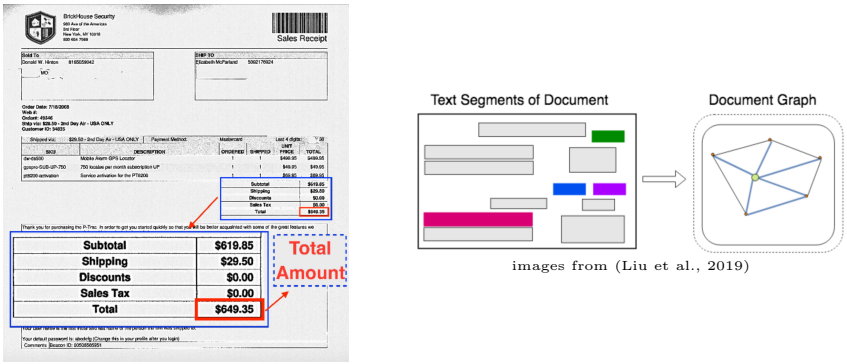


Figure 7.2: A molecule and its graph representation.

the number of atoms), each node corresponds to an atom (a node's attribute x_i is the atom's identity) and the edges correspond to bond, a lasting attraction between atoms (in this example, the edges may not have properties). The task is to predict the molecule's properties such as toxicity, excitation spectra or the level of activity of a chemical compound against cancer cells. Given a graph that represents a molecule, the model is supposed to produce a vector of real values that describe the predicted properties. This task is similar to regression but the inputs are graphs instead of vectors of fixed length.

Another example application is extraction of information from documents. Suppose that we want to extract line item information from scanned invoices. Modern software of optical character recognition (OCR) can extract text segments from scanned documents and we can represent the output of the OCR as a graph in which each node corresponds to one text segment (see Fig. 7.3). The edges in the graph can describe the positions of the text segments relative to each other. The properties of the edges can be the geometrical distance between the segments in the document or some hand-crafted features, for example, whether two text segments are in the same row or in the same column. The task is to classify each node of the graph into two classes: whether the text segment represents a line item or not.



images from (Liu et al., 2019)

Figure 7.3: The task of information extraction from scanned documents. Right: Text segments extracted by OCR are represented as a graph.

We want to design a learning algorithm that would be able to process inputs which are represented as graphs. Since graphs are explicit representations of a set of objects and their relations, we would like to have a tool for modeling relations between objects. There is no "default" deep learning component which operates on an arbitrary relational structure. We will review several neural architectures proposed for this task. We will call all such architectures *graph neural networks*. However, this term does not mean a particular neural network architecture.

7.2 Requirements for graph neural networks

Next we postulate the requirements for graph neural networks.

The first requirement is known as permutation invariance. Suppose that we have a graph with four nodes like the one shown in Fig. 7.4. and we want

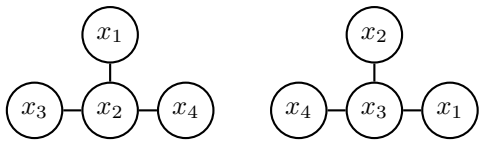


Figure 7.4: Permutation invariance. Since nodes in a graph do not have an order (typically), the output of the network should be invariant to node permutations.

to describe this graph in the python programming language. We can describe the set of nodes as a list in which each element contains the attributes of the corresponding node. The order of the elements in this list would be arbitrary because the nodes of the graph do not have a special order. We could have the order as shown on the left hand side of Fig. 7.4 or as on the right hand side. The property of permutation invariance means that the output of the model is not affected by changing the order of the nodes. This is the first requirement: the output of the network should be invariant to node permutations.

The second requirement is that our model should be able to process graphs with a varying number of nodes. For example, our graph could have four nodes or three nodes (Fig. 7.5) and our model should be able to process both graphs.

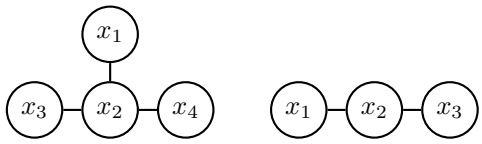


Figure 7.5: The network should be able to process graphs with a varying number of nodes.

The last requirement is that our model should take into account the topology of the graph. For example, the output of the model for the two graphs in Fig. 7.6 can be different because the two graphs have different topologies even though they have the same set of nodes with the same attributes.

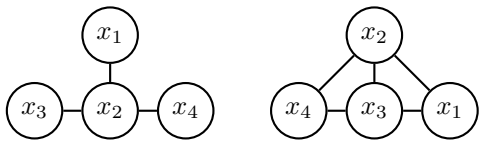


Figure 7.6: The network should take into account the topology of the graph.

Consider the property of permutation invariance. Suppose that our graphs are always fully-connected and we use the multilayer perceptron model to process them (see Fig. 7.7). We could simply feed the node attributes as the inputs of the MLP. However, if we feed the same graph with a different ordering of the nodes, the output of an MLP will change in unpredictable way. An MLP trained on a particular ordering of the input may not generalize well to making good predictions under a different ordering. For a graph with n nodes, there

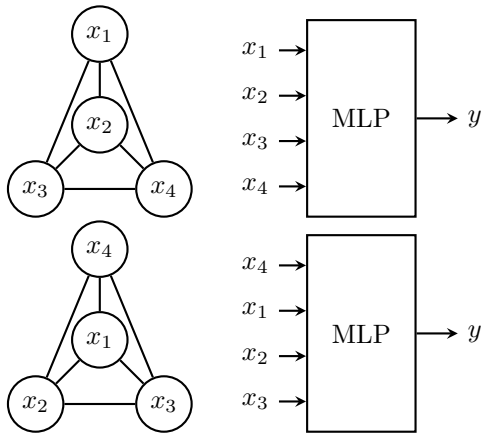


Figure 7.7:

are $n!$ possible permutations of the nodes. If we want to train our model to be invariant to node permutations, we should use a different ordering of the nodes in every new epoch. And we are likely to require a huge number of training iterations to achieve the desired property of permutation invariance.

Does any of the models that we studied previously

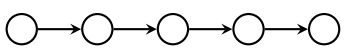
- multi-layer perceptron
- convolutional layer
- recurrent neural network
- transformer encoder

have the property of invariance or equivariance to input permutations (equivariance means that if we change the order of the inputs, then the order of the outputs will change in the same way)? The transformer encoder without positional encoding is equivariant to input permutations.

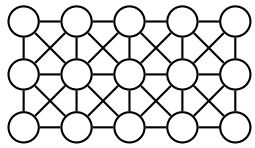
Next we consider the second requirement which is the ability to process inputs with a varying number of elements. Which of the neural networks that we studied previously can process inputs with a varying number of elements? The correct answer is recurrent neural network, transformer encoder and convolutional layers.

Therefore, we can view the previous neural networks as models for processing graphs with a special structure.

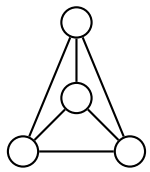
- RNN can be viewed as a neural network which can process graphs with the chain topology.



- Convolutional layers can be viewed as a model that can process graphs with the grid topology.



– Transformer encoder can be viewed as a neural network that processes fully connected graphs.



But we want to create a generic neural network architecture that can process graphs with any topology.

7.3 Neural fingerprint networks

The problem considered by Duvenaud et al. (2015) is prediction of the chemical properties of molecules, such as toxicity, excitation spectra or the level of activity against cancer cells. The idea of the proposed algorithm is to convert a graph that represents a molecule into a real-valued vector $\mathbf{f}$ that is called a *fingerprint*. After that, we can solve the prediction problem by training a regression model that takes $\mathbf{f}$ as the input and predicts the desired properties.

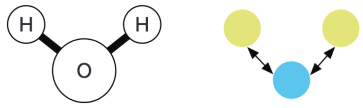


Figure 7.8: A molecule, in which each atom is represented as a node and edges correspond to bond.

The proposed neural algorithm is inspired by a handcrafted algorithm called *Circular fingerprint*s. It is an algorithm designed to encode which substructures are present in a molecule in a way that is invariant to atom-relabeling. Thus, the result produced by the Circular fingerprints algorithm is invariant to atom permutations. The algorithm is presented in Fig. 7.9.

The algorithm starts by assigning an initial integer identifier r_a to each atom. This identifier captures some local information about the corresponding atom. We can encode various atom properties (e.g., atomic number, connection count, etc.) into a single integer value using a pre-defined hash function. Then, the algorithm runs for a few iterations (called layers in Fig. 7.9). In each iteration, we loop through all the atoms in the molecule, combine the identifiers r_a of all the neighbors of that atom and apply a fixed pre-defined hashing function to the concatenated identifiers. In this way, we produce a new identifier r_a for atom a. Finally, we convert the new identifier r_a into index i by taking the remainder

Algorithm 1 Circular fingerprints

Input: molecule, radius R, fingerprint length S
Initialize: fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
for each atom a in molecule
  $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
for $L = 1$ to R ▷ for each layer
  for each atom a in molecule
    $\mathbf{r}_1 \ldots \mathbf{r}_N = \text{neighbors}(a)$
    $\mathbf{v} \leftarrow [\mathbf{r}_a, \mathbf{r}_1, \ldots, \mathbf{r}_N]$ ▷ concatenate
    $\mathbf{r}_a \leftarrow \text{hash}(\mathbf{v})$ ▷ hash function
    $i \leftarrow \text{mod}(r_a, S)$ ▷ convert to index
    $\mathbf{f}_i \leftarrow 1$ ▷ Write 1 at index
Return: binary vector $\mathbf{f}$

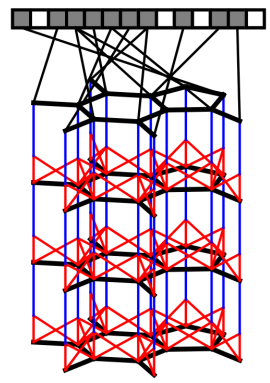


Figure 7.9: *Circular fingerprints* is an algorithm designed to encode which substructures are present in a molecule in a way that is invariant to atom-relabeling (permutation invariance).

of the division of r_a by S which is the length of the fingerprint vector. And we write the value of 1 to the corresponding location i of the fingerprint vector. The result of this procedure is a binary fingerprint vector $\mathbf{f}$ of fixed length S.

The rhs plot in Fig. 7.9 illustrates how the information propagates in the *Circular fingerprints* algorithm. At the beginning, we initialize the identifiers of all atoms with some values. In the first iteration, the identifiers of the atoms are updated using the values of the identifiers of all its neighbors. The red lines indicate what atoms are used to compute the identifier at a specific location. We perform several iterations and we finally convert the identifiers into a binary vector using a pre-defined binarization procedure.

Duvenaud et al. (2015) propose to neuralize the Circular fingerprints algorithm and calling the resulting algorithm *neural fingerprint networks*. They propose to keep the structure of the Circular fingerprints algorithm but to use learnable and differential functions instead of pre-defined functions in the original algorithm. The proposed algorithm is presented on the rhs of Fig. 7.10.

Algorithm 1 Circular fingerprints

Input: molecule, radius R, fingerprint length S
Initialize: fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
for each atom a in molecule
  $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
for $L = 1$ to R ▷ for each layer
  for each atom a in molecule
    $\mathbf{r}_1 \ldots \mathbf{r}_N = \text{neighbors}(a)$
    $\mathbf{v} \leftarrow [\mathbf{r}_a, \mathbf{r}_1, \ldots, \mathbf{r}_N]$ ▷ concatenate
    $\mathbf{r}_a \leftarrow \text{hash}(\mathbf{v})$ ▷ hash function
    $i \leftarrow \text{mod}(r_a, S)$ ▷ convert to index
    $\mathbf{f}_i \leftarrow 1$ ▷ Write 1 at index
Return: binary vector $\mathbf{f}$

Algorithm 2 Neural graph fingerprints

Input: molecule, radius R, hidden weights $H_1^1 \ldots H_R^5$, output weights $W_1 \ldots W_R$
Initialize: fingerprint vector $\mathbf{f} \leftarrow \mathbf{0}_S$
for each atom a in molecule
  $\mathbf{r}_a \leftarrow g(a)$ ▷ lookup atom features
for $L = 1$ to R ▷ for each layer
  for each atom a in molecule
    $\mathbf{r}_1 \ldots \mathbf{r}_N = \text{neighbors}(a)$
    $\mathbf{v} \leftarrow \mathbf{r}_a + \sum_{i=1}^{N} \mathbf{r}_i$ ▷ sum
    $\mathbf{r}_a \leftarrow \sigma(\mathbf{v} H_L^N)$ ▷ smooth function
    $\mathbf{i} \leftarrow \text{softmax}(\mathbf{r}_a W_L)$ ▷ sparsify
    $\mathbf{f} \leftarrow \mathbf{f} + \mathbf{i}$ ▷ add to fingerprint
Return: real-valued vector $\mathbf{f}$

Figure 7.10: Duvenaud et al. (2015) "neuralized" the circular fingerprint algorithm.

The first difference compared to the Circular fingerprints is that instead of concatenating the identifiers from a neighborhood of an atom, they propose to compute the sum of the identifiers $\mathbf{r}_i$. This summation produces a vector of

fixed length which is easy to use as the input of the next layer. Then, instead of using a pre-defined hashing function, they update the identifiers of the atoms by a learnable function which consists of a linear layer and an element-wise nonlinearity σ. This is similar to one hidden layer of a multilayer perceptron. The identifiers are converted into indices using the softmax function. Finally, the values of the softmax are written to the fingerprint vector. As a result, we get a real-valued vector $\mathbf{f}$, not a binary vector like in the Circular fingerprints.

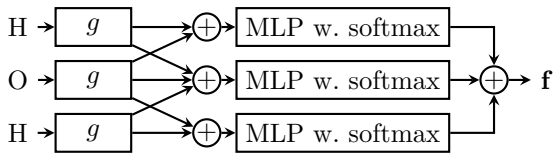


Figure 7.11: Computational graph of one iteration of neural fingerprint networks.

We illustrate the algorithm by drawing the computational graph of one iteration in Fig. 7.11. In this example, the molecule consists of three atoms: H, O and H. First, we use function g to compute the features of the atoms. In the simplest case, function g can be implemented as an embedding layer which simply produces an encoding of the atom identity. Then, for each atom, we sum the features of all its neighbors. Since the O-atom is connected to the other two atoms, we sum all the three features for the O-atom. Each H-atom is connected only to the O-atom and therefore we sum only two identifiers in the H-position.

Lines 9 and 10 in the neural fingerprint algorithm in Fig. 7.10 can be viewed as an MLP network with one hidden layer and the softmax nonlinearity in the output layer. We illustrate them with MLP blocks in our computational graph. Note that the same MLP is used for all atoms in all three positions. Finally, we sum the outputs of the MLPs in all three positions to compute the fingerprint vector.

Table 7.12 shows that the proposed model produces better features than the original circular fingerprints algorithm. In several prediction tasks, the new features yield better performance.

Dataset Units	Solubility [4] log Mol/L	Drug efficacy [5] EC_{50} in nM	Photovoltaic efficiency [8] percent
Predict mean	4.29 ± 0.40	1.47 ± 0.07	6.40 ± 0.09
Circular FPs + linear layer	1.71 ± 0.13	$\mathbf{1.13 \pm 0.03}$	2.63 ± 0.09
Circular FPs + neural net	1.40 ± 0.13	1.36 ± 0.10	2.00 ± 0.09
Neural FPs + linear layer	0.77 ± 0.11	$\mathbf{1.15 \pm 0.02}$	2.58 ± 0.18
Neural FPs + neural net	$\mathbf{0.52 \pm 0.07}$	$\mathbf{1.16 \pm 0.03}$	$\mathbf{1.43 \pm 0.09}$

Figure 7.12: Predictive errors of neural fingerprints compared to standard circular fingerprints.

7.4 Interaction networks

Suppose that we observe a physical system that contains multiple objects interacting with each other (Battaglia et al., 2016). For example, we observe a set of n bodies acting on each other with gravitational forces (see Fig. 7.13a). Or we observe a set of balls moving inside a room, the balls interact with each

(a) A n-body system with gravitation.

(b) A rigid body system: Balls moving inside a room.

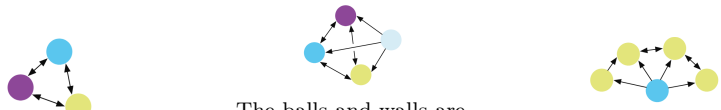
(c) A mass-spring system: A rope and a fixed object.

Figure 7.13: Examples of physical systems modeled with interaction networks.

other and the walls of the room when collisions happen (Fig. 7.13b). The third example is a mass-spring system in which there is a rope which is falling down because of the gravity. One end of the rope is fixed. The trajectory of the rope is affected by interaction with a non-moving object shown with a blue circle in Fig. 7.13c.

The state of a physical system can be described by the positions and the velocities of all the objects. We assume that we can measure the state at any point of time and our task is to predict the evolution of the state in the future.

Battaglia et al., (2016) make a modeling assumption that each pair of objects interact with one another. The interactions can be represented using graphs shown in Fig. 7.14. In the first system, the nodes in the graph represent the



The bodies are nodes and the underlying graph is fully connected.

The balls and walls are nodes, and the underlying graph defines interactions between the balls and between the balls and the walls.

The rope is defined by a sequence of masses which are represented as nodes in the graph.

Figure 7.14: Graph representations of three example physical systems.

n bodies. The graph is fully-connected because all bodies interact with each other. In the second system, the nodes of the graph represent the balls and the walls of the room. The node that represents the walls are shown with the light blue color. The interactions are represented by the directed edges: note that the walls affect the movement of the balls and the balls affect the movement of each other. In the third system, the rope is represented as a set of parts interacting with each other. The parts are represented as nodes in a graph. The fixed object is also represented as a node. We can see that the fixed object affects the parts of the rope while the parts of the rope interact only with their direct neighbors.

Each interaction between a pair of objects is represented by an arrow in the graph. In the interaction networks (Battaglia et al., 2016), the object that corresponds to the source node, the node where the arrow starts, is called the *sender*. The object which corresponds to the destination node is called the *receiver*.

Suppose that the sender object has state o_1, the receiver has state o_2 and their relationship (represented by an edge) has attribute r (see Fig. 7.15). r can be for example, the spring constant for two objects attached by a spring.

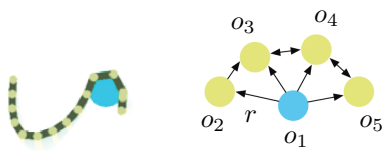


Figure 7.15: A graph that describes the mass-spring system.

The effect of the interaction is predicted by function f_R which takes as input the state of the sender o_1, the state of the receiver o_2 and the attribute of the relationship r:

$$e_{1\to 2,t+1} = f_R(o_{1,t}, o_{2,t}, r).$$

The effect of the interaction $e_{1\to 2,t+1}$, which is the output of function f_R, is used to compute the future state of the receiver. This is done by another function f_O which takes as inputs the current state of the receiver and the effect of the interaction:

$$o_{2,t+1} = f_O(o_{2,t}, e_{1\to 2,t+1}).$$

There can be multiple objects interacting with a receiver. For example, object o_3 is influenced by objects o_1, o_2, o_4. In that case, we need to aggregate the effects of individual interactions and then update the state of the receiver by function f_O:

$$o_{2,t+1} = f_O\left(o_{2,t}, \sum_{i=1,2,4} e_{i\to 2,t+1}\right).$$

Here, we sum the effects of three individual interactions and we use the sum as the input of function f_O.

The future states of other objects are computed in the same way. For example, for object 5, we need to compute the effect of its interaction with object 1 and object 2, then aggregate those effects and update the state o_5 with function f_O. This is illustrated in this computational graph in Fig. 7.16. We compute

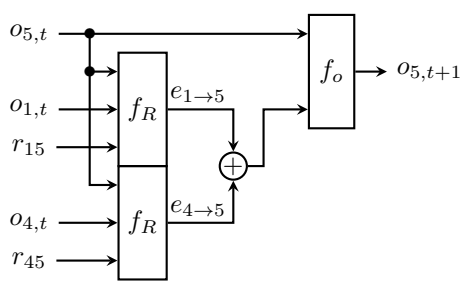


Figure 7.16: Computational graph of interaction networks for one object.

the effects of the interactions of object 5 with object 1 and object 4 using functions f_R that have shared parameters. Then, we aggregate the effects of the two interactions and use the aggregated effect as the input of function f_O which computes the next state of object 5.

Fig. 7.17 illustrates the predictions made by the trained interaction networks for the three physical systems: the n-body system, the moving balls and the

mass-spring system. In the columns on the left-hand side, we can see the ground

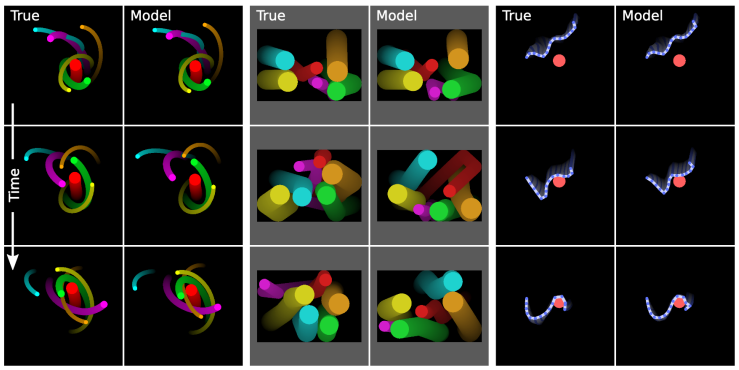


Figure 7.17: Prediction rollouts in the experiments with interaction networks. Each column contains three panels of three video frames (with motion blur), each spanning 1000 rollout steps. Columns 1-2 are ground truth and model predictions for n-body systems, 3-4 are bouncing balls, and 5-6 are strings.

truth trajectories of the objects. In the columns on the right-hand side, we see the predicted trajectories. The predictions are very good especially for the n-body system and the mass-spring system. The predictions for the moving balls are less accurate because learning a good interaction model for this system is a more challenging problem. Interactions in that system happen rarely, only when the balls collide which each other. Therefore, it is more difficult to model accurately the effect of those interactions.

One important property of the interaction networks is that they are able to generalize to a different setup, for example, to a different number of objects. In the experiments presented in Fig. 7.18, the model trained on a n-body system with $n = 6$ objects is able to give good predictions for a system with only three objects. The model trained on the moving balls system with six objects is able

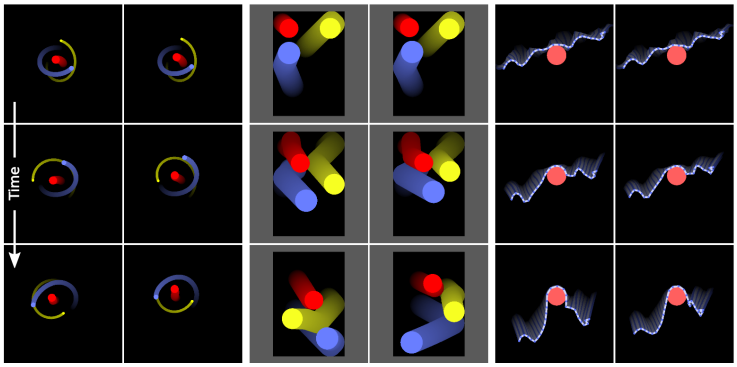


Figure 7.18: The model was able to generalize to systems of different sizes and structure. For n-body, the training was on 6 bodies, and generalization was to 3 bodies. For balls, the training was on 6 balls, and generalization was to 3 balls. For strings, thetraining was on 15 masses with 1 end pinned, and generalization was to 30 masses with 0 end pinned.

to give reasonable predictions for a system with three balls. The model trained on a mass-spring system is able to generalize to a longer rope in which one of the ends is not fixed.

Table 7.1 shows the mean-squared prediction errors obtained with the interaction networks and a few baseline models. The first baseline model assumes that the objects move with constant velocities. It is the simplest model for the considered physical systems. The second baseline is a simple MLP network

Domain	Constant velocity	Baseline	Dynamics-only IN	IN
n-body	82	79	76	**0.25**
Balls	0.074	0.072	0.074	**0.0020**
String	0.018	0.016	0.017	**0.0011**

Table 7.1: Mean-squared prediction errors.

which cannot improve much over the simplest baseline. Note also that the MLP model cannot be even used for systems with a different number of objects because a different number of objects means a different number of model inputs. The third baseline (Dynamics-only IN) is the interaction networks with the effect of the interactions removed. This model basically models the trajectories of the objects independent of each other. This model can technically be used for predicting the state of systems with a varying number of objects. However, the accuracy of the model is very poor. In contrast, the accuracy of the interaction networks is very good. These models can be applied to systems with a varying number of objects without any problems.

7.5 Relational network for visual scene understanding

The model by Santoro et al. (2017) was trained on the data set of relational reasoning called CLEVR. In one training example of the CLEVR data set, there is an image which contains a few objects and a few questions about the objects depicted on the image (see Fig. 7.19).

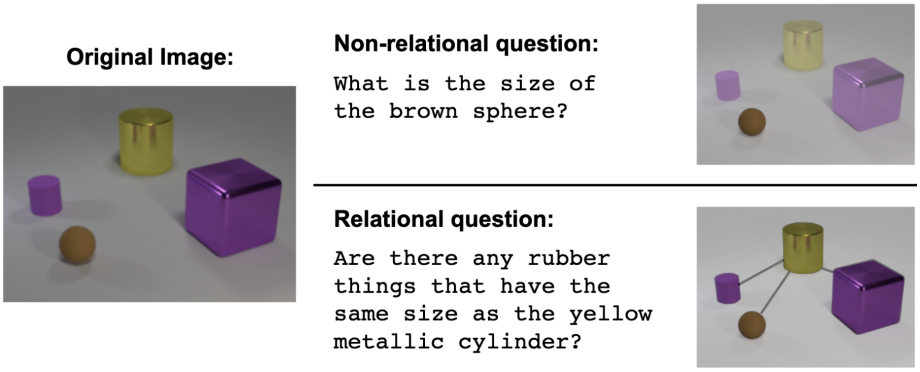


Figure 7.19: An example from CLEVR dataset of relational reasoning: An image containing four objects is shown alongside non-relational and relational questions. The relational question requires explicit reasoning about the relations between the four objects in the image, whereas the non-relational question requires reasoning about the attributes of a particular object.

There are relational and non-relational questions. Relational questions require explicit reasoning about the relations between the objects in the image. An example of a relational question is "Are there any rubber things that have the same size as the yellow metallic cylinder?" To answer this question, we need to understand how the sizes of the objects compare to the size of the yellow metallic cylinder. Non-relational questions require reasoning about the attributes of a particular object. An example of a non-relational question is "What is the size of the brown sphere?"

In the model proposed by Santoro et al. (2017), an image is decomposed into patches and each image patch is treated as an object which interacts with all

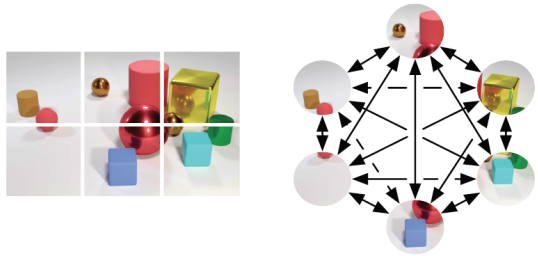


Figure 7.20: An image is decomposed into patches. Each patch is treated as an object (a node in a fully connected graph).

other image patches. The interactions can be represented with a fully-connected graph which schematically shown in Fig. 7.20. The graph is processed with a relational neural network which models the relations between each pair of objects to produce the correct answer to a given question. The question is processed with a recurrent neural network and the produced encoding is then used as a global context for modeling relations.

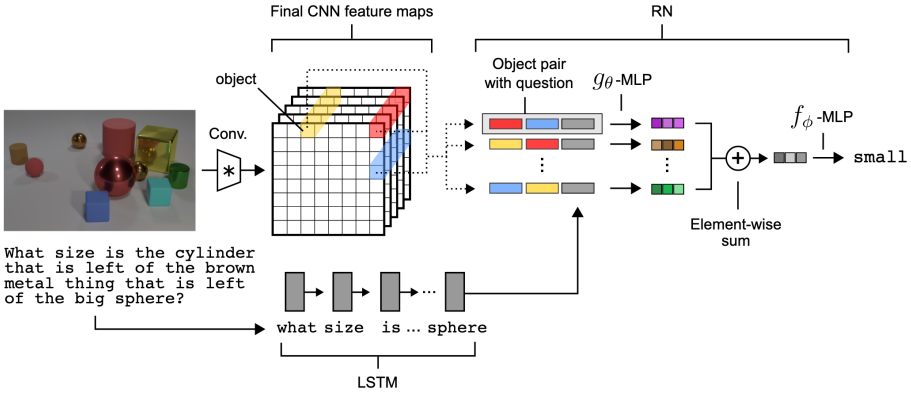


Figure 7.21: Questions are processed with an LSTM to produce a question embedding. Images are processed with a CNN to produce a set of objects for the RN. Objects (three examples illustrated here in yellow, red, and blue) are constructed using feature-map vectors from the convolved image. The RN considers relations across all pairs of objects, conditioned on the question embedding, and integrates all these relations to answer the question.

Fig. 7.21 illustrates the computational graph to process an image and a question. The question is processed with an LSTM to produce a question embedding. The image is processed with a convolutional neural network to produce a two-dimensional map. Each pixel in the output of the CNN is interpreted as one object, the attributes of the object are the channels of the map. Then, there is a relational network that contains an MLP which models the relations between each pair of objects. The same MLP (with shared parameters) is used to process every pair of objects. The inputs of the MLP are the attributes of the first object and the attributes of the second object and the question embedding. There are n^2 MLPs with shared parameters where n is the number of pixels in the output of the CNN. The outputs of the MLPs are summed together. The sum is then processed by another MLP which is trained to produce the correct answer. The answer in this dataset consists of one word and therefore we basically have a classification problem in which we need to select one word from the dictionary.

Table 7.2 shows that the model produces very good results on the CLEVR

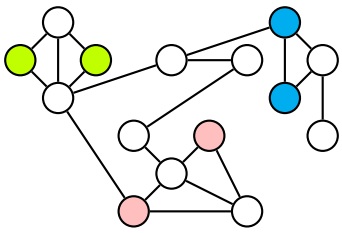
dataset.

Model	**Overall**	Count	Exist	Compare Numbers	Query Attribute	Compare Attribute
Human	92.6	86.7	96.6	86.5	95.0	96.0
Q-type baseline	41.8	34.6	50.2	51.0	36.0	51.3
LSTM	46.8	41.7	61.1	69.8	36.8	51.8
CNN+LSTM	52.3	43.7	65.2	67.1	49.3	53.0
CNN+LSTM+SA	68.5	52.2	71.1	73.5	85.3	52.3
CNN+LSTM+SA*	76.6	64.4	82.7	77.4	82.6	75.4
CNN+LSTM+RN	**95.5**	**90.1**	**97.8**	**93.6**	**97.9**	**97.1**

Table 7.2: Results on CLEVR from pixels. Accuracy on the test set broken down by question category.

7.6 Graph Convolutional Networks

The motivation of the graph convolutional networks (Kipf and Welling, 2017) is in semi-supervised classification of nodes in a graph. Suppose that we have a graph in which some nodes are labeled (which means that they are known to belong to certain classes) and some nodes are not labeled. For example, nodes in the graph may correspond to documents, edges may correspond to citation links between the documents. Node attributes x_i can be the bag-of-words features of documents, which is basically the counts of words that appear in a document. Some of the documents may be known to belong to particular classes, for example, they may discuss a particular topic.



Example:
- nodes are documents
- edges are citation links
- node attributes x_i are bag-of-words features of documents
- some documents have class labels

Figure 7.22: Example of semi-supervised classification of nodes in a graph.

In Fig. 7.22, the classes of the nodes are represented with different colors. The unlabeled nodes are represented with the white color. The task is to classify all the unlabeled documents in the dataset (which correspond to the white nodes of the graph). The modeling assumption that we make is that when predicting the class of a node, the attributes and the connectivity of nearby nodes provide useful information that should be used by the classifier. For example, if a document is linked with several documents of the same class, it is very likely to belong to the same class.

The considered problem is somewhat similar to the problem of image segmentation. In that problem, the task is to classify each pixel of an image. We can view an image as a graph where each pixel is connected to all its neighbors

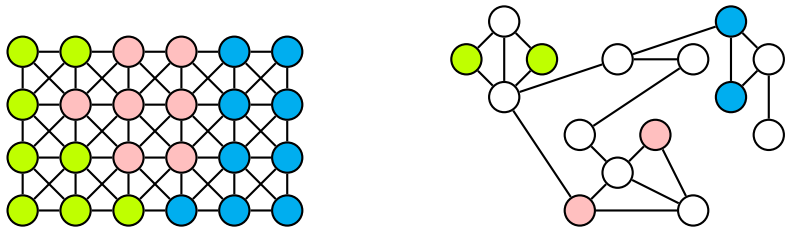


Figure 7.23: Kipf and Welling (2017) generalize the concept of convolution to graphs with arbitrary structure.

(see Fig. 7.23). Image segmentation is usually done with convolutional neural networks (such as U-net).

The idea behind graph convolutional networks (GCN) is to generalize the segmentation problem to graphs with any topology. Since convolutions are useful for processing images, the authors proposed to generalize the concept of convolution to graphs with arbitrary topology. They perform this generalization by taking a spectral view on convolutions. Convolutions in the Fourier-domain are simple pointwise multiplication of the Fourier-transform of a signal. And since there are ways to define the Fourier transform of a graph, the authors provide a way to generalize convolutions to graphs using the spectral view.

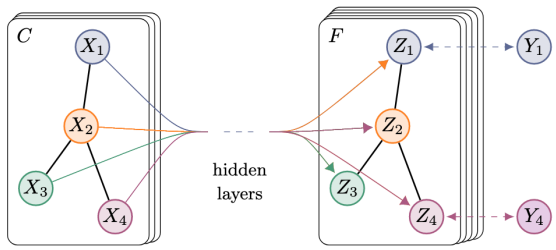


Figure 7.24: Graph convolutional layer.

As a result of this generalization, the authors propose a so-called graph convolutional layer (see Fig. 7.24). The input of the layer is a graph with N nodes. Each node of the graph has C attributes. Therefore the attributes of the nodes can be represented as an $N \times C$ matrix X. The output of the layer is a graph with the same topology but a new set of node attributes. Each node in the output has F attributes and therefore the new attributes can be represented as an $N \times F$ matrix Z. Using the authors' notation, the output is computed as:

$$Z = \hat{A}XW$$

where W is a $C \times F$ matrix of the layer parameters and $\hat{A}$ is a matrix that describes the topology of the graph: it has non-zero diagonal elements and elements $a_{ij} \neq 0$ if node i is connected to node j.

We can write how the new attributes are computed for each node i:

$$z_{i:} = (\hat{a}_{i:}XW)^\top = \sum_{j \in \mathcal{N}(i)} \hat{a}_{ij} W^\top x_{j:} \,,$$

where $z_{i:}$ is a vector of the output attributes of node i. We can see that $z_{i:}$ is computed as a weighted sum of linear transformations $W^\top x_{j:}$ of the attributes

of all the nodes that are connected to node i. This happens because only for the neighbors of node i the elements a_{ij} are non-zero.

In the experiments, Kipf and Welling (2017) applied the graph convolutional network to the problem of semi-supervised classification of documents. The nodes in the graph corresponded to documents and the edges corresponded to citation links. The node attributes were the bag-of-words features of the documents. Some documents had class labels and the task was to classify all the documents. Table 7.25 shows that GCN outperforms other alternative algorithms considered in that paper.

Method	**Citeseer**	**Cora**	**Pubmed**	**NELL**
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [28]	59.6	59.0	71.1	26.7
LP [32]	45.3	68.0	63.0	26.5
DeepWalk [22]	43.2	67.2	65.3	58.1
ICA [18]	69.1	75.1	73.9	23.1
Planetoid* [29]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7

Figure 7.25: Classification accuracy (in percent).

7.7 Recurrent Relational Networks

The motivation for the paper by Palm et al., (2018) is to train a model that can solve tasks that require a chain of multiple steps of relational inference. For example, when we solve a Sudoku puzzle, we analyze relations between different cells of the grid and iteratively fill the empty cells.

How can we train a neural network to solve Sudoku puzzles using a training set that consists of solved puzzles? We can represent the Sudoku puzzle as a graph in which each node corresponds to one cell of the grid. We know the rules of the game and we can advise our network which relations are important by constructing the graph with a certain topology. The important relations are among cells in the same row, in the same column or in the same 3×3 block. Therefore, in our graph we add edges between each pair of nodes in the same row, in the same column and in the same 3×3 block. Fig. 7.3 shows a subgraph of this graph, the subgraph contains all the nodes that are connected to one of the nodes of the graph.

The algorithm of recurrent relational networks starts by initializing the states of the graph nodes with some values h_j^0. Then, we build a computational graph that contains T iterations with shared parameters. Each iteration consists of the following steps.

– Step 1 is to compute messages for all the edges connecting a pair of nodes i and j:

$$m_{ij}^t = f(h_i^{t-1}, h_j^{t-1})$$
$$m_{ji}^t = f(h_j^{t-1}, h_i^{t-1}).$$

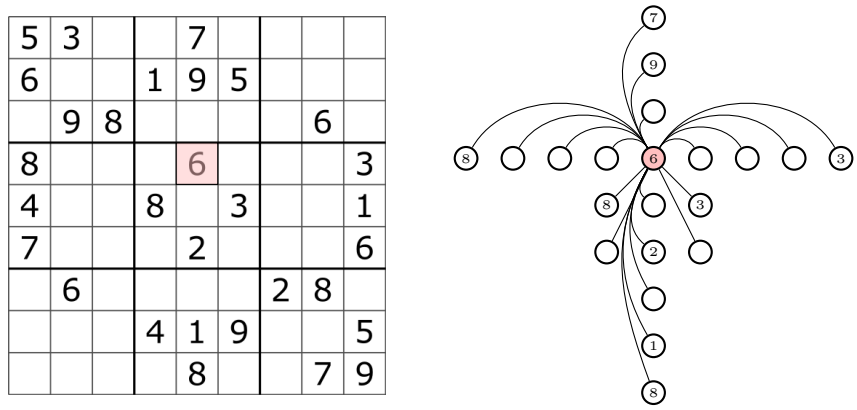


Figure 7.26: The subgraph that contains only the nodes connected to the pink node and the corresponding links.

Messages m_{ij}^t, m_{ji}^t are vectors produced by function f that we model, for example, with a multilayer perceptron network. The MLP takes as inputs the states of the source and the destination nodes. Note that for each edge we need to compute two messages: going in both directions. For the subgraph shown in this figure, we compute the messages along all the edges that arrive in the pink node. Of course, messages are computed for all other nodes as well.

– In Step 2, we aggregate all incoming messages for each node by summation.

$$m_j^t = \sum_{i \in \mathcal{N}(j)} m_{ij}^t \,.$$

– In Step 3, we update the states of all the nodes. We do that using function g that takes as inputs the previous values of the states, the inputs of the nodes and the aggregate message:

$$h_j^t = g(h_j^{t-1}, x_j, m_j^t).$$

In the Sudoku example, input x_j is either the known digit for cell j or a special token indicating that the digit is unknown. In the recurrent relational networks, function g is implemented with a gated recurrent unit. We update the state of the gated recurrent unit for each node in the graph. Note that all the nodes use the same GRU with shared parameters.

– In Step 4, we compute the output of each node using function f_o that takes as input the state of the node:

$$o_j^t = f_o(h_j^t).$$

The outputs of the nodes are used to compute the loss. The loss function compares the outputs with the targets. In the Sudoku example, the desired outputs are the correct digits in the solved puzzle. This can be viewed as a classification problem with nine possible classes and therefore we can use the cross-entropy loss.

Palm et al., (2018) compare the proposed model with other algorithm applicable to solving Sudoku puzzles. Table 7.3 shows that the model yields much higher accuracy compared to other models. Note that a convolutional network works much worse in this problem.

Method	Givens	Accuracy
*Recurrent Relational Network** (this work)	17	**96.6%**
Loopy BP, modified [Khan et al., 2014]	17	92.5%
Loopy BP, random [Bauke, 2008]	17	61.7%
Loopy BP, parallel [Bauke, 2008]	17	53.2%
Deeply Learned Messages* [Lin et al., 2015]	17	0%
Relational Network, node* [Santoro et al., 2017]	17	0%
Relational Network, graph* [Santoro et al., 2017]	17	0%
Deep Convolutional Network [Park, 2016]	24-36	70%

Table 7.3: Comparison of methods for solving Sudoku puzzles (only differentiable methods).

The relational network trained to solve Sudoku puzzles performs several iterations of inference. One can analyze the outputs that the network produced after each iteration (see an example in Fig. 7.27). Note that the network is able to solve this puzzle in just three iteration of relational inference.

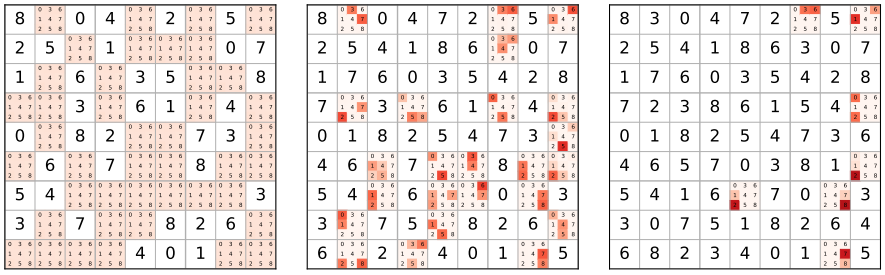


Figure 7.27: The output of a trained Sudoku puzzle at initialization and after two iterations. The shades of red indicate the confidence of the network in its prediction: higher confidence corresponds to darker shades.

7.8 General algorithms for graph neural networks (GNNs)

We have considered several algorithms that operate on graph inputs:

- neural fingerprints networks (Duvenaud et al., 2015)
- interaction networks (Battaglia et al., 2016)
- a relational network for visual scene understanding (Santoro et al., 2017)
- graph convolutional networks (Kipf and Welling, 2017)
- recurrent relational networks (Palm et al., 2018).

They are all different algorithms but they all have very similar structure. In every iteration, nodes send messages to all their neighbors. Then, the node attributes are updated using the received messages. The differences between the algorithms are mainly in the parametric form of the messages and the way the messages are aggregated. Let us review the computational steps in such graph neural networks.

Suppose that the input of a GNN is an undirected graph G with node features $\mathbf{x}_i$ and edge features $\mathbf{e}_{ij}$. Each node of the GNN has hidden state $\mathbf{h}_i$ which is initialized to some value. The GNN performs T iterations which consist of several steps that update the states of the nodes

$$\mathbf{h}_i^0 \to \mathbf{h}_i^1 \to ... \to \mathbf{h}_i^T \,.$$

Finally, if the task is to produce a single output for the whole graph, there is a function which combines all the node states to compute a single output $\mathbf{y}$.

$$\mathbf{y} = f_o(\{\mathbf{h}_i^T \mid i \in G\})$$

If the task is to produce an output for each node, like, for example, in the Sudoku problem, then we apply a readout function to the state of each node:

$$\mathbf{y}_i = f_o(\mathbf{h}_i^T).$$

One iteration of a GNN is summarized below.

1. Each node receives messages from all its neighbors

$$\mathbf{m}_{j\to i}^{t+1} = g_t(\mathbf{h}_j^t, \mathbf{h}_i^t, \mathbf{e}_{ji}).$$

2. Each node aggregates messages (for example, by summation):

$$\mathbf{m}_i^{t+1} = \sum_{j\in\mathcal{N}(i)} \mathbf{m}_{j\to i}^{t+1} \,.$$

3. The state of each node is updated using the aggregate message:

$$\mathbf{h}_i^{t+1} = f(\mathbf{h}_i^t, \mathbf{m}_i^{t+1}, \mathbf{x}_i),$$

   where f is often implemented using a recurrent unit such as GRU. One can use node features $\mathbf{x}_i$ as extra inputs.

Let us see how the computations in a graph convolutional layer (Kipf and Welling, 2017) can be viewed as a particular realization of this algorithm.

1. Each node receives messages from all its neighbors

$$\mathbf{m}_{j\to i}^{t+1} = g(\mathbf{h}_j) = \hat{a}_{ij}\mathbf{W}^\top \mathbf{h}_j \,.$$

2. Each node aggregates messages (including a message from the node itself):

$$\mathbf{m}_i^{t+1} = \sum_{j\in\mathcal{N}(i)} \mathbf{m}_{j\to i}^{t+1} \,.$$

3. The states of nodes are updated using aggregate messages. A graph convolutional layer is typically followed by an element-wise nonlinearity such as relu, which results in the following update of the node states:

$$\mathbf{h}_i^{t+1} = f(\mathbf{m}_i^{t+1}) = \mathrm{relu}(\mathbf{m}_i^{t+1}).$$

This analysis shows that GCNs can be viewed as a particular realization of a general framework of graph neural networks. You can see that GCNs use very simple functions g and f.

Gilmer et al. (2017) were probably first who proposed to unify several graph neural networks under a general framework of *message passing neural networks*. Later, Battaglia et al. (2018) defined a slightly more general framework that also includes the updates of the edge attributes. Note that the term message-passing exists in the literature on probabilistic graphical models. The message-passing algorithm is used to perform inference in probabilistic graphical models, such as Bayesian networks or Markov random fields. The message-passing algorithm in that domain is known as belief propagation.

Chapter 8

Deep autoencoders

8.1 Motivation

In the previous chapters, we considered only supervised learning problems and we used data sets that consisted of inputs $\mathbf{x}$ and desired outputs $\mathbf{y}$. Deep learning is a very effective tool for supervised learning. In almost any domain where one can collect large amounts of labeled data, it is possible to train a deep neural network which would have excellent predictive capabilities.

However, for building the next generation of artificial intelligence systems, the supervised learning scenario seems somewhat artificial. It seems unrealistic that we can advise the artificial agent what outputs to produce in all possible situations. The task of *unsupervised learning* is to make computers learn from unlabeled data

$$\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)},$$

data that do not contain labels $\mathbf{y}$.

Humans learn a lot from unlabeled data. Of course, we provide a lot of supervision to our children, but kids learn a lot by themselves by observing and interacting with the environment. It seems that in order to build intelligent systems that can learn quickly, we need to make machines also learn from unlabeled data.

Unsupervised learning can have real-world application such as

- detect samples that look different from the training population (novelty/anomaly detection)
- visualize data, discover patterns in data (information visualization)
- generate new samples which look similar to the training data (generative modeling).

Unsupervised learning can also be useful for representation learning. As we discussed Section 1, the goal of representation learning is to learn useful features

and in unsupervised learning, we want to extract features without using the labels:

$$\mathbf{x} \xrightarrow{f} \mathbf{z}\,.$$

We do hope that the extracted features would work better than raw data in future (downstream) tasks.

The problem is that we do not know for which downstream tasks we need to prepare. From this point of view, unsupervised learning is not a well defined problem. One popular solution is to extract patterns (or features) that appear frequently in the data and hope that those features will be useful in the downstream tasks. In this approach, we compress the data, for example, we represent the data with a fewer number of dimensions.

8.2 Principal component analysis

Principal component analysis (PCA) is one of the simplest techniques of data compression. Suppose that we have data $\mathbf{x}$ normalized to zero mean. In PCA, the first principal component is found by maximizing the variance of the data when projected onto a unit-length vector $\mathbf{w}_1$:

$$y_1 = \mathbf{w}_1^\top \mathbf{x}\,, \qquad \|\mathbf{w}_1\| = 1.$$

The variance of this projection is given by

$$\mathbb{E}\{y_1^2\} = \mathbb{E}\{\mathbf{x}\mathbf{x}^\top\} = \mathbf{w}_1^\top \mathbb{E}\{\mathbf{x}\mathbf{x}^\top\}\mathbf{w}_1 = \mathbf{w}_1^\top \mathbf{C}_\mathbf{x} \mathbf{w}_1\,,$$

where $\mathbf{C}_\mathbf{x}$ is the covariance matrix of the data. We want to maximize the variance, which means that we need to solve the following optimization problem:

$$\mathbf{w}_1^* = \arg\max_{\mathbf{w}_1} \mathbf{w}_1^\top \mathbf{C}_\mathbf{x} \mathbf{w}_1, \quad \text{s.t. } \|\mathbf{w}_1\| = 1.$$

The solution of this optimization problem is given by the first, dominant eigenvector of the covariance matrix $\mathbf{C}_\mathbf{x}$.

After we find the first principal component, we can find the second principal component by maximizing the variance in the subspace orthogonal to the first eigenvector $\mathbf{w}_1$. And we can continue the extraction of the principal components in this way. This is probably the most well-known formulation of PCA.

Perhaps a less well-known formulation of PCA is the minimization of the mean-squared error of compression. The optimization problem is formulated in the following way. We want to find an m-dimensional subspace with an orthonormal basis $\mathbf{w}_i$

$$\underset{n\times m}{\mathbf{W}} = \begin{bmatrix}\mathbf{w}_1 & \dots & \mathbf{w}_m\end{bmatrix} \qquad \mathbf{W}^\top \mathbf{W} = \mathbf{I}$$

such that the n-dimensional vectors $\mathbf{x}$ are first projected onto that subspace

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x}\,.$$

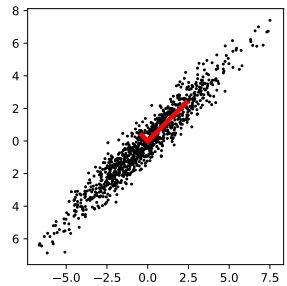


Figure 8.1: Principal component analysis finds directions that maximize data variance. The longer red line corresponds to the first principal component of the data while the shorter line corresponds to the second principal component.

and then projected back to the original n-dimensional space

$$\hat{\mathbf{x}} = \mathbf{W}\mathbf{z} = \sum_{i=1}^{m} (\mathbf{w}_i^\top \mathbf{x})\mathbf{w}_i = \mathbf{W}\mathbf{W}^\top \mathbf{x}, .$$

We want to find matrix $\mathbf{W}$ such that the mean-squared error between the original data $\mathbf{x}$ and its reconstruction $\hat{\mathbf{x}}$ is minimized:

$$\mathbf{W}_{\text{PCA}} = \arg\min_{\mathbf{W}} \mathbb{E}\{\|\mathbf{x} - \hat{\mathbf{x}}\|^2\}, \qquad \text{s.t. } \mathbf{W}^\top \mathbf{W} = \mathbf{I}.$$

To be more precise, this formulation is known as principal subspace analysis because there are infinitely many orthonormal bases that span the principal subspace and the principal component analysis defines one of them.

$$\begin{aligned}
&\text{encoder:} && f(\mathbf{x}) = \mathbf{W}_f \mathbf{x} \\
&\text{decoder:} && \hat{\mathbf{x}} = g(\mathbf{z}) = \mathbf{W}_g \mathbf{z} \\
&\text{loss:} && \mathcal{L} = \mathbb{E}\{\|\mathbf{x} - g(f(\mathbf{x}))\|^2\}
\end{aligned}$$

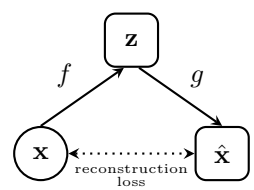


Figure 8.2: PCA as an autoencoder: We learn a mapping from $\mathbf{x}$ to $\mathbf{x}$.

We can implement this optimization in PyTorch by building a computational graph shown in Fig. 8.2. We transform vector $\mathbf{x}$ with a linear layer to vector $\mathbf{z}$ and then transfrom $\mathbf{z}$ to $\hat{\mathbf{x}}$ using another linear layer. The minimized loss is the mean-squared distance between the original input $\mathbf{x}$ and its reconstruction $\hat{\mathbf{x}}$. To match the PCA formulation, we could additionally enforce parameter sharing between the two linear layers and add a penalty term to enforce the orthogonality of matrix $\mathbf{W}$. However, if we only care about minimizing the mean-squared reconstruction error, we do not have to do that.

This type of models is called an *autoencoder*. In autoencoders, the same vector is used as the input and as the target for the output. Thus, we learn a mapping from $\mathbf{x}$ to $\mathbf{x}$ with a function that is a composite of functions f and g. Function f is called the *encoder*, function g is called the *decoder*. In principal subspace analysis, both the encoder and the decoder are implemented as linear transformations.

8.3 Bottleneck autoencoders

The autoencoder model makes no sense if we do not restrict functions f and g somehow. The model can learn a trivial identity mapping with $\mathbf{W}_g = \mathbf{W}_f^{-1}$

$$\hat{\mathbf{x}} = g(f(\mathbf{x})) = \mathbf{W}_g \mathbf{W}_f \mathbf{x} = \mathbf{x},$$

which yields a zero reconstruction error. One simple way to restrict the model is to use $\mathbf{z}$ with a smaller dimensionality than the dimensionality of $\mathbf{x}$. This makes autoencoding useful: we represent $\mathbf{x}$ with a smaller-dimensional vector $\mathbf{z}$, that is we compress the data.

Representations $\mathbf{z}$ of a smaller dimensionality are often called the *bottleneck* and the layer that produces such representations is often called the *bottleneck layer*. Thus, the principal subspace can be learned with a *bottleneck autoencoder* with a linear encoder and a linear decoder.

In order to improve the level of compression so that we get a smaller reconstruction error with a bottleneck layer of the same size, we can use an autoencoder with a nonlinear encoder and a nonlinear decoder. Deep autoencoder is an autoencoder in which both the encoder and the decoder are implemented as deep neural networks (Fig. 8.3). The parameters of the autoencoder can be tuned by minimizing the same mean-squared reconstruction error.

$$\begin{aligned} \mathbf{z} &= f(\mathbf{x}, \boldsymbol{\theta}_f) \\ \boldsymbol{\theta}_f, \boldsymbol{\theta}_g &= \operatorname*{arg\,min}_{\boldsymbol{\theta}_f, \boldsymbol{\theta}_g} \mathbb{E}\{\|\mathbf{x} - g(\mathbf{z}, \boldsymbol{\theta}_g)\|^2\}. \end{aligned}$$

In bottleneck autoencoders (Bourlard and Kamp, 1988; Oja, 1991), we use a bottleneck layer $\mathbf{z}$ which has fewer dimensions than the input. By doing so, we prevent learning a trivial identity mapping.

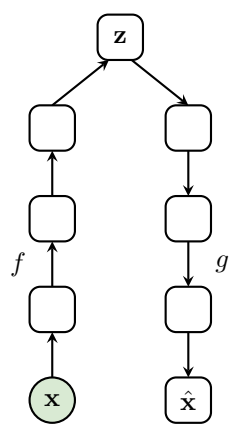


Figure 8.3: Deep autoencoder.

The advantage of deep autoencoders compared to linear models such as PCA is that they can learn complex data manifolds. In the hypothetical example presented in Fig. 8.4, the data lie on a one-dimensional manifold embedded in

the two-dimensional space. Principal component analysis is not be able to learn the one-dimensional manifold because it is a linear model. However, a nonlinear autoencoder is a able to learn such a curved data manifold.

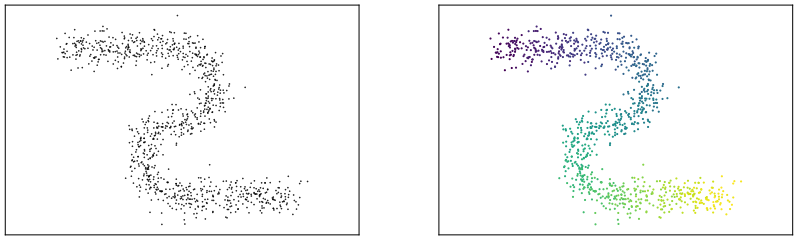


Figure 8.4: A one-dimensional data manifold in the two-dimensional space. With a nonlinear autoencoder, we can learn a curved data manifold. Colors represents the values of the latent code z that may be found by a nonlinear autoencoder.

The most popular use case for deep autoencoders is data compression. For example, consider the task of playing the game of Doom with a reinforcement learning algorithm (Ha and Schmidhuber, 2018). Learning from raw images (pixels) is likely to require a huge number of training episodes because the amount of input data is very large. It may take a very long time for the agent to figure out what are the useful features for collecting rewards and therefore for learning a good policy. Therefore, the authors first compress the data using an autoencoder and then train the agent using compressed representations as features.

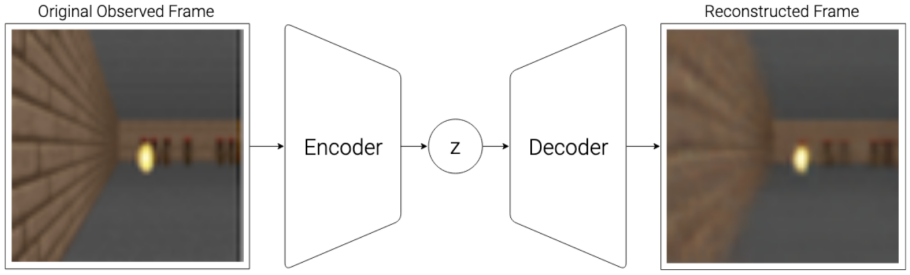


Figure 8.5: Learning compressed representations as features for a reinforcement learning agent (Ha and Schmidhuber, 2018).

8.4 Denoising autoencoders

Denoising autoencoders (Vincent et al., 2008) are similar to the bottleneck autoencoders with the difference is that instead of using original data $\mathbf{x}$ as input, the denoising autoencoder always uses data $\mathbf{x}$ corrupted with noise (see Fig. 8.6). The simplest way to corrupt the data is by adding Gaussian noise:

$$\tilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\epsilon} \quad \text{with} \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2). \tag{8.1}$$

The autoencoder needs to produce the original *clean* samples in the output, similarly to the vanilla autoencoder. This learning task is called *denoising* because the model is trained to remove noise from the data. The same mean-squared

error between the output $\hat{\mathbf{x}} = g(f(\tilde{\mathbf{x}}))$ and the desired clean input is typically used as the loss function:

$$c = \mathbb{E}\{\|\hat{\mathbf{x}} - \mathbf{x}\|^2\}.$$

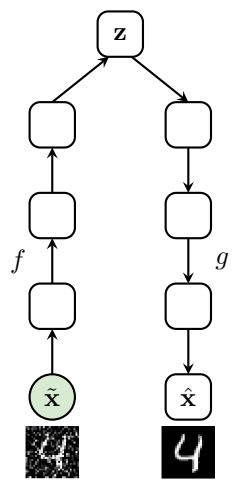


Figure 8.6: A denoising autoencoder is trained to reconstruct a clean image from a corrupted one.

One can view adding noise to inputs as a way to regularize the autoencoder. Recall that one of the regularization methods that we studied in Section 4.5 was adding noise to model inputs. However, there is more theory behind denoising autoencoders. The theory says that if samples are corrupted with Gaussian noise as in (8.1), the optimal denoising function has the following form (Alain and Bengio, 2014, Raphan and Simoncelli, 2011):

$$d(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}} + \sigma^2 \nabla_{\tilde{\mathbf{x}}} \log p(\tilde{\mathbf{x}}) \tag{8.2}$$

where $\nabla_{\tilde{\mathbf{x}}} \log p(\tilde{\mathbf{x}})$ is the gradient of the log-probability density function of the corrupted data $\tilde{\mathbf{x}}$, which is also known as the *score function*. σ is the standard deviation of the corruption noise. If $\sigma = 0$, which means that there is no noise in the input, then the optimal denoising is the identity mapping. This is natural because if the input does not contain noise, there is no need for denoising. If σ is non-zero, we modify the corrupted example $\tilde{\mathbf{x}}$ in the direction where the log-probability density function grows with the fastest rate. This is illustrated in Fig. 8.7.

Equation (8.2) suggests that in order to learn the optimal denoising function $d(\mathbf{x})$, the model need to learn the properties of the data distribution $p(\mathbf{x})$ which is related to the score function $\nabla_{\tilde{\mathbf{x}}} \log p(\tilde{\mathbf{x}})$. Note also that the optimal denoising function is not equal to the identity mapping. Therefore, even if a denoising autoenconder does not have a bottleneck layer, it can still learn to develop useful representations.

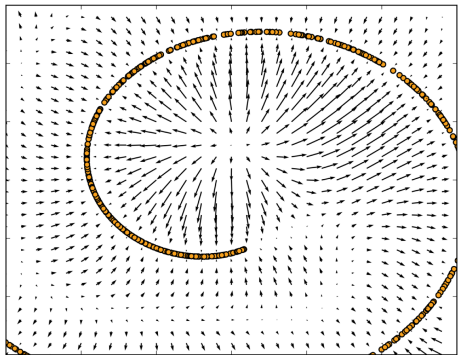


Image from (Alain and Bengio, 2014)

Figure 8.7: A denoising autoencoder learns to move noisy samples towards regions with higher probability density. The yellow dots represent the manifold formed by clean samples. The arrows represent the denoising function learned.

8.5 Converting autoencoders into generative models

8.5.1 Generative model

Next we build an autoencoder which is a probabilistic *generative* model, which is a model that can represent the data distribution $p(\mathbf{x})$ and which can be used to generate new examples from the data distribution. Vanilla autoencoders are not generative models because they cannot be used to generate new samples. Neither can they compute the probability that a sample comes from the data distribution.

A simple example of a probabilistic generative model is a mixture of Gaussians model (see Fig. 8.8). Its probability density function for one-dimensional x and two Gaussian components is given by

$$p(x \mid \boldsymbol{\theta}) = w_1 \mathcal{N}(x \mid \mu_1, \sigma_1^2) + w_2 \mathcal{N}(x \mid \mu_2, \sigma_2^2).$$

This model has six parameters $\boldsymbol{\theta} = \{w_1, \mu_1, \sigma_1, w_2, \mu_2, \sigma_2\}$ which can be tuned by maximizing the likelihood. We can also draw samples from this distribution, therefore it is a generative model. This model is an example of an explicit density model because its probability density function $p(\mathbf{x} \mid \boldsymbol{\theta})$ has an explicit parametric form.

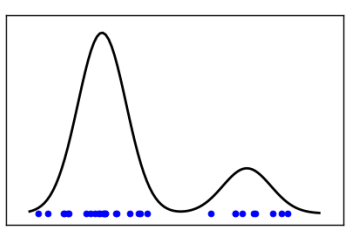


Figure 8.8: Probability density function of a mixture of two Gaussians. The blue dots represent samples from the distribution.

We can build a deep generative model using the decoder part of the autoencoder. We can assume that the variables in the bottleneck layer are distributed according to a simple tractable distribution, for example, a spherical Gaussian

distribution:

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}).$$

We can further assume that the data samples $\mathbf{x}$ are nonlinear transformations of variables $\mathbf{z}$ with the possibility of adding some noise $\boldsymbol{\varepsilon}$:

$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon}.$$

Function g can be modeled as a deep neural network. We can also assume that the noise has a spherical Gaussian distribution:

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I}).$$

Now we can draw samples from the model: first we draw $\mathbf{z}$ from the Gaussian distribution and then we transform $\mathbf{z}$ with a deep neural network g and add noise $\boldsymbol{\varepsilon}$ to get a generated sample $\mathbf{x}$.

Our model contains *latent variables* $\mathbf{z}$. These variables are unobserved, we do not know their values and we can only measure $\mathbf{x}$. Fig. 8.9 presents a toy example which illustrates our latent variable generative model. We model one-dimensional data x as a noisy transformation of a Gaussian variable z. The transformation is done with a nonlinear function g. You can see that even though the distribution of z is a simple Gaussian, by using a nonlinear transformation g, we can produce a more complex distribution of x, for example, a bi-modal distribution that we see on the plot.

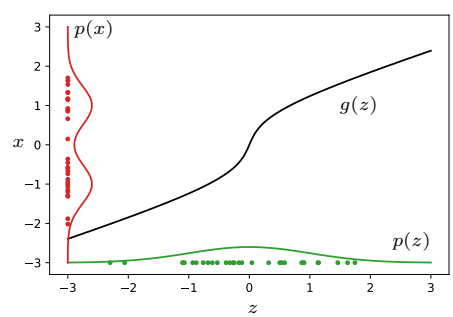


Figure 8.9: A toy example of a generative latent variable model with one-dimensional z and x.

8.5.2 Learning the model parameters with the EM algorithm

Now our goal is to learn the parameters of our generative model using training data $\{\mathbf{x}_i\}$. In the toy example in Fig. 8.9, the training data are shown with the red dots. We need to learn the nonlinearity g and the noise level σ such that the data distribution can be accurately approximated by the transformation of the Gaussian latent variable z with the nonlinear function g.

We can tune the parameters $\boldsymbol{\theta}, \sigma^2$ of our model by maximizing the probability of the training data (maximum likelihood estimate):

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \log p(\mathbf{x}_1, ..., \mathbf{x}_N \mid \boldsymbol{\theta}).$$

Since our training examples are sampled independently, the log-probability of the training data is the sum of the log-probabilities of the individual samples:

$$\log p(\mathbf{x}_1, ..., \mathbf{x}_N \mid \boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \int p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) p(\mathbf{z}_i) d\mathbf{z}. \quad (8.3)$$

We need to integrate out the latent variable $\mathbf{z}$ from the joint distribution $p(\mathbf{x}, \mathbf{z})$ to compute the log-probabilities of the individual training examples.

The probability density functions that participate in this formula are defined by our generative model. The conditional probability $p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta})$ is Gaussian whose mean is determined by the output of the nonlinear function g:

$$p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I}).$$

The covariance matrix of this distribution is determined by our noise model and we postulated the covariance matrix to be $\sigma^2\mathbf{I}$. The prior distribution over the latent variables $\mathbf{z}$ is the spherical Gaussian because we chose that distribution in the generative model;

$$p(\mathbf{z}_i) = \mathcal{N}(\mathbf{z}_i \mid 0, \mathbf{I}).$$

Even though we can write formula (8.3) for the log-likelihood, the integral that participates in (8.3) is not tractable. Therefore, we cannot compute this function analytically and thus we cannot optimize it directly.

The classical way to estimate the parameters of a model with latent variables is to use of the expectation-maximization algorithm, the EM algorithm. The EM-algorithm iterates between two steps: the E-step and the M-step. In the E-step, we compute the posterior probabilities $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ of the latent variables given the data and the current values of the model parameters. In the M-step, we update the values of the model parameters $\boldsymbol{\theta}$ using the probabilities computed in the E-step.

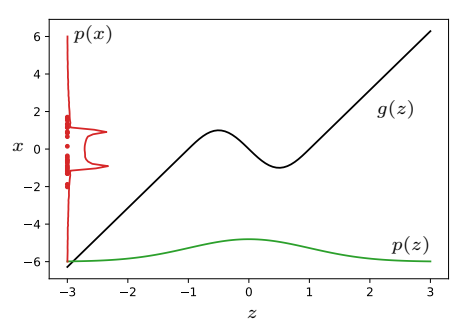


Figure 8.10: Consider our simple example. We initialize $\boldsymbol{\theta}$ with values that give us $\mathbf{g}$ of the form shown in the figure.

Let us use the toy example from Fig. 8.9 to gain some intuition at what happens in the two steps of the EM algorithm. Suppose that we have initialized the model parameters such that function g has the shape shown in Fig. 8.10.

Then, the distribution $p(x)$ that is produced by our generative model does not describe well our training data shown with the red dots.

In the E-step, we take each training example and compute the posterior probability of the latent variable z given the data sample x_1 and the current values of the model parameters:

$$q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta}).$$

For example, for x_1 shown in Fig. 8.11 the values of the latent variable z that could have generated are represented by the conditional distribution $p(z_1 \mid x_1, \boldsymbol{\theta})$ shown with the green line.

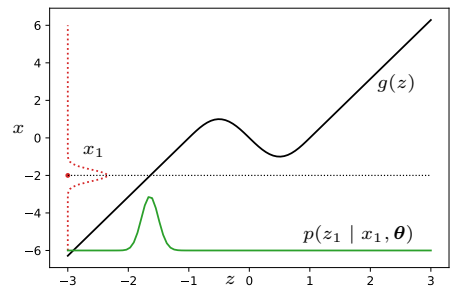


Figure 8.11: E-step: For each training data point, find the distribution over the latent variables that could have produced that data point according to the model

For the second sample x_2 shown in Fig. 8.12, the conditional distribution $p(z_2 \mid x_2, \boldsymbol{\theta})$ is more complex because there are three areas in the latent space that could have produced x_2. Therefore, the conditional distribution has three modes, as shown on the plot. For the third sample x_3, the conditional distribution is simpler (see Fig. 8.12). In the E-step, we need to evaluate these conditional probabilities $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ for all the examples in the training set.

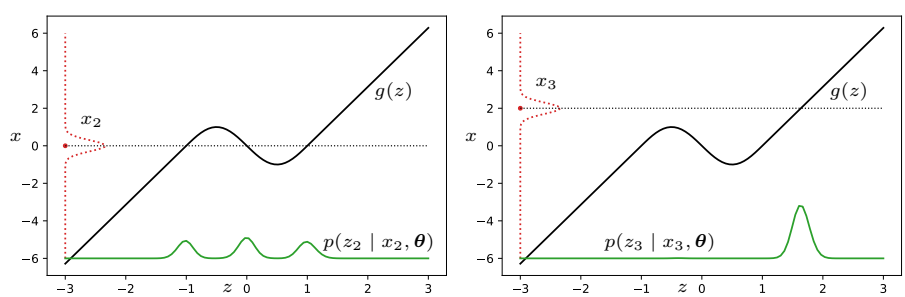


Figure 8.12: E-step: For each training data point, find the distribution over the latent variables that could have produced that data point according to the model

In the M-step, we use the distributions computed in the E-step, which we denote by $q(\mathbf{z}_i)$, to form an objective function of the following form:

$$\mathcal{F}(\boldsymbol{\theta}) = \langle \log p(\mathbf{x}_1, ..., \mathbf{x}_N, \mathbf{z}_1, ..., \mathbf{z}_N \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_1, ..., \mathbf{z}_N)} \tag{8.4}$$

$$= \sum_{i=1}^{N} \langle \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) \rangle_{q(\mathbf{z}_i)} = \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i. \tag{8.5}$$

This function is the expected value of the joint probability of all the variables in the model including the latent ones. The expectation is taken using the conditional probabilities $q(\mathbf{z}_i)$ evaluated on the E-step. In the M-step, this objective function is maximized wrt the model parameters $\boldsymbol{\theta}$ that participate in $\log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})$.

It can be shown that the EM-algorithm improves the values of the log-likelihood $\log p(\mathbf{x}_1, ..., \mathbf{x}_N \mid \boldsymbol{\theta})$ after each iteration. Therefore, by alternating the E- and the M-steps, we can find a local maximum of the likelihood function. Fig. 8.13 illustrates the optimization procedure: by performing a few iterations of the EM-algorithm, we find the values of the model parameters that describe well the distribution of the training data.

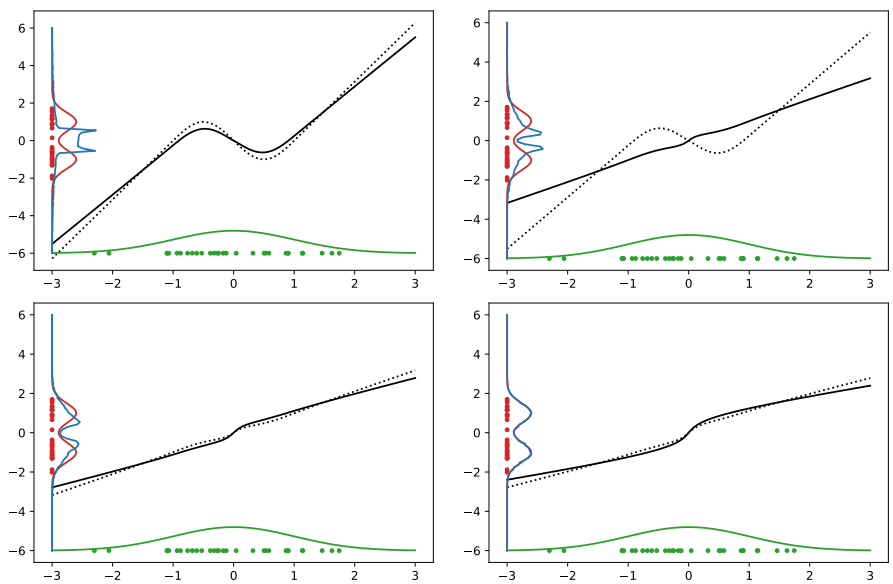


Figure 8.13: Illustration of a few steps of the EM algorithm.

8.5.3 Variational approximation

There are a few problems with the direct application of the EM-algorithm to learning our deep generative model. One problem is the intractability of the true conditional distributions $q(\mathbf{z}_i) = p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ that we need to compute on the E-step. As we can see from Fig. 8.12, the true distribution can be very complex. And for a nonlinear function g modeled with a neural network, it certainly does not have an analytical form. We have to approximate these distributions somehow.

A popular solution is to approximate the true conditional distribution with a distribution that have a simple form, for example, a Gaussian distribution. Let us consider our toy problem again. The true conditional distribution for training example x_2 has three modes, as shown with the green line in Fig. 8.14. We approximate the true distribution with a Gaussian distribution $q(z_2)$ which

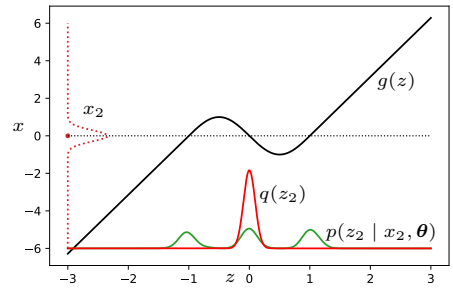


Figure 8.14:

effectively captures only one of the modes. The approximating distribution $q(\mathbf{z}_i) \approx p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$, which is shown with the red line on this plot, is often called *variational approximation*.

If we decide to use a Gaussian approximation

$$q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$$

then we have two parameters $\mu_{\mathbf{z}_i}$ and $\sigma^2_{\mathbf{z}_i}$ that define our variational approximation. Note that we need to use a different variational approximation for every example in the training set because the true conditional distribution depends on the training example. Therefore, we need two parameters $\mu_{\mathbf{z}_i}$ and $\sigma^2_{\mathbf{z}_i}$ describing $q(\mathbf{z}_i)$ for each training sample. Parameters describing the posterior distributions of the latent variables $\boldsymbol{\theta}_q = \{\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i}\}_{i=1}^N$ are called *variational parameters*.

To find the best approximation $q(\mathbf{z}_i)$, we usually minimize the Kullback-Leibler divergence between the true distribution $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ and its approximation $q(\mathbf{z}_i)$. We can minimize the KL divergence between $q(\mathbf{z}_i)$ and $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ using the following trick. We add to the objective function (8.5) that we used in the M-step of the EM-algorithm the entropies of the approximate distributions $q(\mathbf{z}_i)$:

$$\begin{aligned}
\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) &= \sum_{i=1}^N \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{what we had in the M-step}} \underbrace{- \int q(\mathbf{z}_i) \log q(\mathbf{z}_i) d\mathbf{z}_i}_{\text{entropy}} \\
&= \sum_{i=1}^N \int q(\mathbf{z}_i) \log \frac{p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta})}{q(\mathbf{z}_i)} dz_i = \sum_{i=1}^N \int q(\mathbf{z}_i) \log \frac{p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta}) p(\mathbf{x}_i \mid \boldsymbol{\theta})}{q(\mathbf{z}_i)} dz_i \\
&= \sum_{i=1}^N -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})) + \log p(\mathbf{x}_i \mid \boldsymbol{\theta}).
\end{aligned}$$

Note that the term $\log p(\mathbf{x}_i \mid \boldsymbol{\theta})$ does not depend on the variational approximations. Therefore, maximization of objective $\mathcal{F}$ wrt the variational approximations $q(\mathbf{z}_i)$ is equivalent to minimizing the KL divergence between the true conditional distributions $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ and their approximations $q(\mathbf{z}_i)$.

The good thing about the new objective function $\mathcal{F}$ is that it is a function of both model parameters $\boldsymbol{\theta}$ and the varitional approximations $q(\mathbf{z})$ and we can

optimize this function jointly with respect to $\boldsymbol{\theta}$ and $q(\mathbf{z})$ without having to use the alternating E- and M-steps. If we fix the model parameters

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})) + \underbrace{\log p(\mathbf{x}_i \mid \boldsymbol{\theta})}_{\text{constant}}$$

and only tune the variational approximation, then maximizing $\mathcal{F}$ is equivalent to minimizing the KL divergence between the variational approximation and the true conditional distributions, which can be viewed as an approximate E-step. If we fix the variational approximation

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i, \mathbf{z}_i \mid \boldsymbol{\theta}) d\mathbf{z}_i - \underbrace{\int q(\mathbf{z}_i) \log q(\mathbf{z}_i) d\mathbf{z}_i}_{\text{constant}},$$

then maximizing $\mathcal{F}$ is equivalent to maximizing the first term and this is what is done in the M-step of the EM-algorithm. Now we can optimize $\mathcal{F}$ jointly wrt the model parameters $\boldsymbol{\theta}$ and the variational parameters $\boldsymbol{\theta}_q$ with any optimizer of our choice.

Since the KL divergence is always non-negative $D_{\mathrm{KL}}(q \parallel p) \geq 0$, we get

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) \leq \sum_{i=1}^{N} \log p(\mathbf{x}_i \mid \boldsymbol{\theta}) = \log p(\mathbf{x}_1, ..., \mathbf{x}_N \mid \boldsymbol{\theta}),$$

which shows that $\mathcal{F}$ is the lower bound of the true log-likelihood. This is the log-likelihood that we actually want to maximize. It is also called evidence and therefore function $\mathcal{F}$ is often called *evidence lower bound* (ELBO). The closer our approximation $q(\mathbf{z}_i)$ to the true posterior $p(\mathbf{z}_i \mid \mathbf{x}_i, \boldsymbol{\theta})$, the closer we are to the true log-likelihood.

ELBO can be re-written in the following form

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) dz_i - \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i. \tag{8.6}$$

Recall that in our deep generative model, we postulated that

$$p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I}).$$

Then, the first term in (8.6) can be written in the following form

$$\left\langle -\frac{D}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \sum_{d=1}^{D} (\mathbf{x}_i(d) - g_d(\mathbf{z}_i, \boldsymbol{\theta}))^2 \right\rangle_{q(\mathbf{z}_i)},$$

where $\langle \rangle_q$ denotes the expectation computed over distribution q, D is the number of dimensions in $\mathbf{x}$, $\mathbf{x}_i(d)$ is the d-th element of $\mathbf{x}_i$ and g_d is the d-th element of the output of function g. One can see that this term contains the expected

mean-squared error between data sample $\mathbf{x}_i$ and its reconstruction g from the latent code $\mathbf{z}_i$.

The second term of ELBO can be recognized as minus KL-divergence between the approximate distribution $q(\mathbf{z}_i)$ and the prior distribution $p(\mathbf{z}_i)$:

$$-\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i = -D_{\mathrm{KL}}(q(\mathbf{z}_i) \parallel p(\mathbf{z}_i)).$$

This terms encourages the posterior approximation $q(\mathbf{z}_i)$ to be close to the prior distribution $p(\mathbf{z})$ that we selected to be a spherical Gaussian distribution in our model. This term works as a regularization term, it prevents divergence of the latent codes from the prior distribution. This is written again below:

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{minus mean-square reconstruction error}} \underbrace{-\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{regularization term}}.$$

8.5.4 Variational autoencoder

The first algorithm for learning a latent variable model of this form

$$\begin{aligned} \mathbf{z} &\sim \mathcal{N}(0, \mathbf{I}) \\ \mathbf{x} &= g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon} \\ \boldsymbol{\varepsilon} &\sim \mathcal{N}(0, \sigma^2 \mathbf{I}) \end{aligned}$$

with a neural network for modeling g and variational approximations was proposed by Lappalainen and Honkela (2001). The objective function was the ELBO that had the form (8.6). The approximate distributions $q(\mathbf{z}_i)$ were chosen to be Gaussian $q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$ and therefore the approximate distribution for each training example had two variational parameters $\mu_{\mathbf{z}_i}$, $\sigma^2_{\mathbf{z}_i}$.

One problem of that early model was the large number of the variational parameters $\boldsymbol{\theta}_q = \{\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i}\}_{i=1}^N$: the number of them was proportional to the number of training samples. How can we get rid of those parameters?

Let us consider the E-step of the EM-algorithm once again. The goal of that step is to find the conditional distribution over the latent code given the data. If we use variational approximations (see Fig. 8.14), the task is to find the approximate distribution that would be closest to the true conditional distribution over the latent code. For fixed model parameters $\boldsymbol{\theta}$, the distribution $q(\mathbf{z})$ that we need to evaluate in the E-step is only a function of the data sample $\mathbf{x}$. Therefore, the E-step essentially maps inputs $\mathbf{x}$ to distributions $q(\mathbf{z})$ over the latent code. If we use a Gaussian approximation $q(\mathbf{z})$, we need to map inputs $\mathbf{x}$ into the two parameters μ and σ of the Gaussian approximation:

$$\mathbf{x} \to \mu_{\mathbf{z}}, \sigma^2_{\mathbf{z}}.$$

In the model called *variational autoencoder*, Kingma and Welling (2014) proposed to approximate this mapping using a neural network. This neural

network can be called an encoder because its job is to convert input $\mathbf{x}$ into a distribution over the latent code, which is similarly to what is done by the encoder in the vanilla autoencoder. The encoder of the VAE performs *amortized inference*: when doing inference for a particular sample $\mathbf{x}_i$, it leverages the knowledge of the inference results for other samples. If two samples $\mathbf{x}_i$ and $\mathbf{x}_j$ are close to each other, then the corresponding approximations $q(\mathbf{z}_i)$ and $q(\mathbf{z}_j)$ should be close as well.

Let us review the components of the variational autoencoder (VAE). Our generative model, for example,

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \qquad \mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon} \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

is defined by the decoder. The second component of our model is the encoder which is trained to perform variational inference. The encoder is a neural network that takes as input a data sample $\mathbf{x}$ and produces a distribution over the latent code. For a Gaussian approximation, the neural network produces two vectors

$$\mathbf{x} \to \mu_{\mathbf{z}}, \sigma^2_{\mathbf{z}} \,.$$

In practice, this is done using one neural network with two output heads (see Fig. 8.15). The encoder is similar to the encoder in a bottleneck autoencoder. The difference is that produces two outputs that define the distribution over the latent code $\mathbf{z}$.

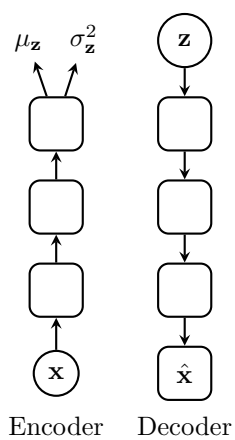


Figure 8.15: Variational autoencoder.

The VAE is trained to maximize the ELBO objective function (8.6):

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \sum_{i=1}^{N} \underbrace{\int q(\mathbf{z}_i) \log p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i}_{\text{needs approximations}} \underbrace{- \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i}_{\text{can be computed analytically}} \,.$$

If we use Gaussian approximations $q(\mathbf{z}_i)$, the second term can be computed analytically. However, the first term cannot computed analytically and needs to be approximated.

Kingma and Welling, (2014) propose to use Monte Carlo estimation of the first term. We can draw L samples from distribution $q(\mathbf{z}_i)$ and use them to compute the approximation

$$\int q(\mathbf{z}_i) \log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I}) d\mathbf{z}_i \approx \frac{1}{L} \sum_{l=1}^{L} \log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I}),$$

where $\mathbf{z}_i^{(l)}$ are drawn from $q(\mathbf{z}_i)$. In practice, using $L = 1$ works pretty well.

Let us review how one can compute an approximation of the ELBO objective function. For each training example $\mathbf{x}_i$, we

- compute $\mu_{\mathbf{z}_i}$ and $\sigma_{\mathbf{z}_i}$ using the encoder
- compute the term $\int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} d\mathbf{z}_i$ analytically
- draw $L = 1$ samples $\mathbf{z}_i^{(l)}$ from $q(\mathbf{z}_i) = \mathcal{N}(\mu_{\mathbf{z}_i}, \sigma^2_{\mathbf{z}_i})$
- propagate $\mathbf{z}_i^{(l)}$ through the decoder and compute the first term $\log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})$.

This results in the computational graph shown in Fig. 8.16.

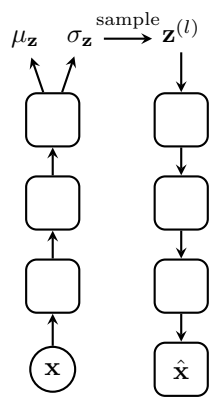


Figure 8.16: Computational graph of the variational autoencoder.

Now we want to update the model parameters, which include the parameters of the encoder and the parameters of the decoder, to maximize the objective function. In order to do that we need to backpropagate the derivatives through the computational graph in Fig. 8.16. We can easily perform backpropagation through the decoder. But then we face a problem: how do we backpropagate through the sampling procedure? If we cannot do that, we cannot compute the derivatives wrt the parameters of the encoder. Therefore, we need a sampling operation that can be differentiated wrt parameters μ and σ.

We can obtain this with the *reparameterization trick*. In that trick, we sample noise from the spherical Gaussian distribution

$$\boldsymbol{\epsilon}_i \sim \mathcal{N}(0, \mathbf{I})$$

and then compute a sample as

$$\mathbf{z}_i = \mu_{\mathbf{z}_i} + \sigma_{\mathbf{z}_i} \boldsymbol{\epsilon}_i.$$

Now sample $\mathbf{z}_i$ comes from the desired distribution but we can also backpropagate through this sampling block because we can compute the derivatives wrt $\mu_{\mathbf{z}}$ and $\sigma_{\mathbf{z}}$. And we can propagate the derivatives further through the encoder. This is illustrated in Fig. 8.17.

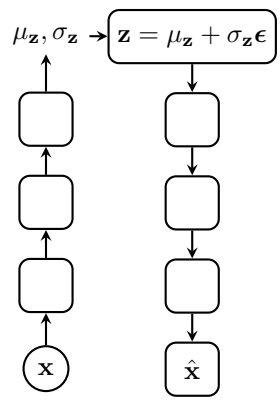


Figure 8.17: Computational graph of the variational autoencoder with the reparameterization trick.

Let us review the training algorithm of the VAE.

- Take a mini-batch $\{\mathbf{x}_i\}$ of training samples.
- Use the encoder to compute means $\mu_{\mathbf{z}_i}$ and standard deviations $\sigma_{\mathbf{z}_i}$ for each sample $\mathbf{x}_i$ in the mini-batch.
- Draw $\boldsymbol{\epsilon}_i \sim \mathcal{N}(0, \mathbf{I})$ and compute samples $\mathbf{z}_i = \mu_{\mathbf{z}_i} + \sigma_{\mathbf{z}_i} \boldsymbol{\epsilon}_i$.
- Propagate samples $\mathbf{z}_i$ through the decoder to compute reconstructions $\hat{\mathbf{x}}_i$.
- Compute the loss which is the negative of

$$\mathcal{F}(\boldsymbol{\theta}, \boldsymbol{\theta}_q) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\log \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i^{(l)}, \boldsymbol{\theta}), \sigma^2 \mathbf{I})}_{\text{Monte Carlo estimate}} \underbrace{- \int q(\mathbf{z}_i) \log \frac{q(\mathbf{z}_i)}{p(\mathbf{z}_i)} dz_i}_{\text{can be computed analytically}}$$

- Perform backpropagation and update the parameters of the encoder and the decoder.

8.6 VAE vs bottleneck autoencoder

VAE is a more complex model than a simple bottleneck autoencoder. Do we need all these complications? VAEs are more powerful: in some problems when vanilla autoencoders fail, VAEs can develop useful representations. The problem of the vanilla autoencoder is the mean-squared error loss, which makes too

simplistic assumptions about the data distribution. One advantage of the VAE is its greater flexibility in defining the generative model. Note that denoising autoencoders are more powerful than the vanilla autoencoders even though they also use the mean-squared error loss.

Variational autoencoders can be used for feature extraction and traditionally they are used for that. However, VAEs are generative models and they can be used for generating samples. So far, the quality of the samples generated with VAEs has not been very impressive. Samples and reconstructions often look blurry (see Fig. 8.19). However, Vahdat and Kautz (2020) presented a VAE model that is able to generate high-quality images. It is a hierarchical latent variable model, that is there are multiple levels of latent variables.

Figure 8.18:

Images from (Tolstikhin et al., 2017)

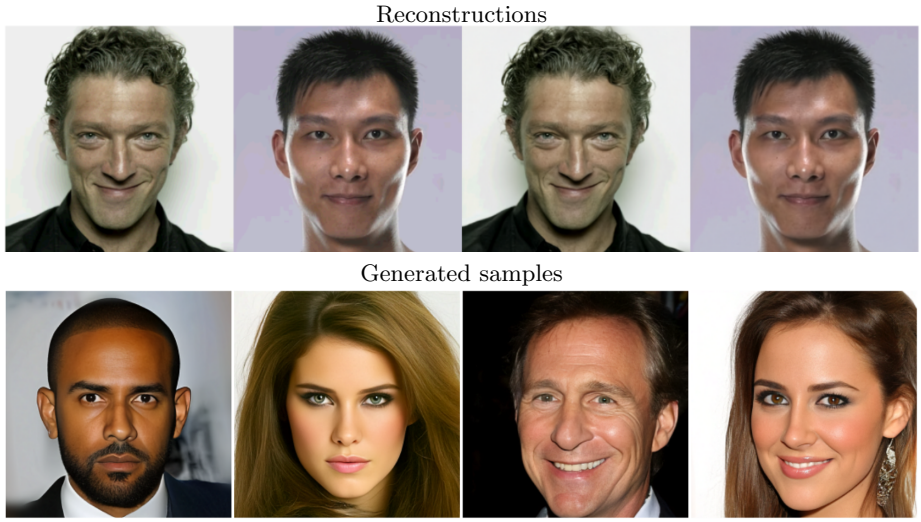


Figure 8.19: Original images and reconstructions (above) and generated samples using Nouveau VAE (Vahdat and Kautz, 2020).

8.7 Home assignment

In the home assignment, you need to train a bottleneck autoencoder on the MNIST dataset. When you visualize the representations developed in the bottleneck layer, you will clearly see the ten clusters corresponding to the ten

MNIST classes. This is a sign that the autoencoder has learned the structure of the data.

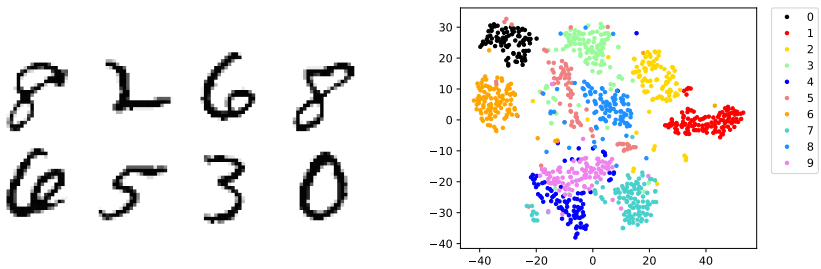


Figure 8.20: MNIST data set (left) and t-SNE visualization of a 10-dimensional $\mathbf{z}$-space found with using a bottleneck autoencoder (right).

In the second part, you will have to train a denoising autoencoder on a synthetic dataset (which I called the variance MNIST). The dataset is created by multiplying the MNIST data by Gaussian noise (see Fig. 8.21). Note that these are not corrupted data, this is how our clean training examples look like. Even though the samples look noisy, it is quite easy for us to recognize the digits. The reason why we use this dataset is because a vanilla bottleneck autoencoder with the mean-squared reconstruction loss cannot extract features that would capture the shapes of the digits. If you are interested, you can try to train a bottleneck autoencoder on this dataset. But a denoising autoencoder can extract meaningful features. This is illustrated on this plot which shows the representations found by a denoising autoencoder.

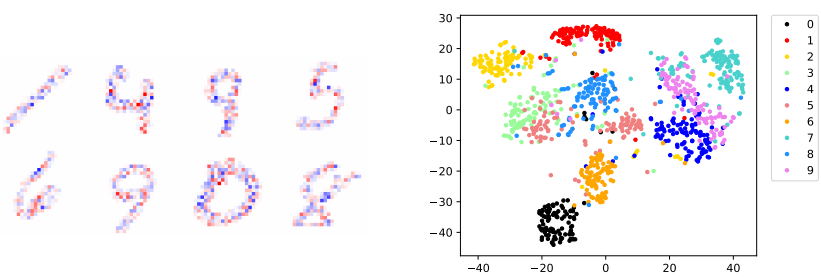


Figure 8.21: Variance MNIST data set (left) and t-SNE visualization of a 10-dimensional $\mathbf{z}$-space found with using a denoising autoencoder (right).

In the third part of the assignment, you will have to train a variational autoencoder on the variance MNIST dataset. In order to extract meaningful features in this dataset, one needs to use a generator (a decoder) that models the variances of the pixel intensities. The generative model that you need to learn in this exercise is given by

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}) \qquad \mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}(\mathbf{z}), \mathrm{diag}(\boldsymbol{\sigma}(\mathbf{z})))$$
$$\boldsymbol{\mu}(\mathbf{z}) = g_\mu(\mathbf{z}, \boldsymbol{\theta}) \qquad \boldsymbol{\sigma}(\mathbf{z}) = \exp(g_\sigma(\mathbf{z}, \boldsymbol{\theta})).$$

You can see that the latent code $\mathbf{z}$ is used both to produce the mean and the covariance matrix of the Gaussian distribution in the pixel space. We assume a diagonal covariance matrix and therefore we need to produce a vector of standard deviations, one standard deviation for each pixel. We need to guarantee

that the standard deviation is positive and therefore we will output the log-standard deviation and then we take the exponent of that.

Chapter 9

Generative adversarial networks

9.1 Classical generative adversarial networks

Let us start by reviewing the generative model of the variational autoencoder that we studied in Section 8.5.1. We have latent variables $\mathbf{z}$ that are, for example, normally distributed

$$\mathbf{z} \sim \mathcal{N}(0, \mathbf{I}). \tag{9.1}$$

The data vectors $\mathbf{x}$ are produced using a nonlinear transformation of the latent variables with the possibility of adding some noise ε:

$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}) + \boldsymbol{\varepsilon}. \tag{9.2}$$

VAE is an explicit density model because we define an explicit parametric form of the data distribution $p(\mathbf{x})$. For example, for the generative model defined in (9.1)–(9.2), $p(\mathbf{x})$ can be expressed as

$$p(\mathbf{x}) = \prod_i \int p(\mathbf{z}_i) p(\mathbf{x}_i \mid \mathbf{z}_i, \boldsymbol{\theta}) d\mathbf{z}_i = \prod_i \int \mathcal{N}(\mathbf{z}_i \mid 0, \mathbf{I}) \mathcal{N}(\mathbf{x}_i \mid g(\mathbf{z}_i, \boldsymbol{\theta}), \sigma^2 \mathbf{I}) d\mathbf{z}_i.$$

In this section, we consider an unsupervised learning method with an implicit density model. That means that we do not have an explicit form of the data distribution $p(\mathbf{x})$. The model is called generative adversarial networks (GAN).

The GAN model was proposed by Goodfellow et al. (2014). A GAN consists of two components (see Fig. 9.1): 1) a generator which is a neural network that converts noise into generated samples and 2) a discriminator which is another neural network that tells whether the generated samples are good or bad. The discriminator is simply a classifier that classifies the generated samples into two classes: a good example or a bad example. Different GAN models may have

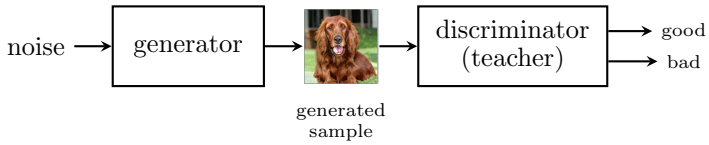


Figure 9.1: Generative adversarial networks.

different architectures of the generator, different architectures of the discriminator, a different choice for the discriminator loss function or different kinds of regularization techniques used to train the model.

We start with the GAN generator. The simplest generative process that can be used by a GAN generator is:

– Sample noise from an isotropic Gaussian distribution

$$\mathbf{z} \sim \mathcal{N}(0, I)$$

– Transform vector $\mathbf{z}$ with a deep neural network to generate a data sample $\mathbf{x}$:

$$\mathbf{x} = g(\mathbf{z}, \boldsymbol{\theta}).$$

You can see that this generative process is quite similar to the one that we used in the decoder of the variational autoencoder.

This is not the only way to build the generator. We may use different strategies for using the noise inside the generator. We may, for example, apply additive or multiplicative noise (such as dropout) to the hidden layers or we can concatenate noise to the hidden layers. In principle, there are infinitely many ways how to add noise inside the generator network.

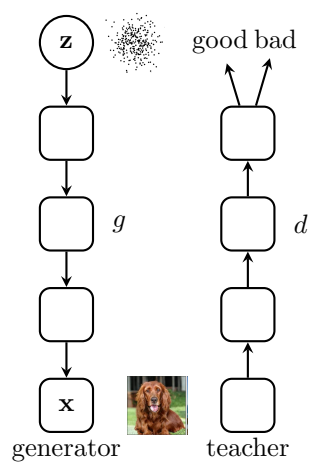


Figure 9.2: Generative adversarial networks with deep neural networks as the generator and the discriminator.

The discriminator is a neural network that guides the generator network. It assesses the quality of the generated samples by classifying them into two

classes "good" or "bad". Using the feedback from the discriminator, the generator can adapt its parameters and produce samples that look good from the discriminator's point of view. Therefore, the discriminator can be viewed as a teacher or as a coach which produces the training signal (or the supervision) for the generator.

How to train the teacher? The elegant idea of GANs is that the discriminator is trained to classify samples as "good" if they come from the training data set and as 'bad" if they are produced by the generator. The output is the probability that the input $\mathbf{x}$ is a real example, which means an example which belongs to the training set. The discriminator $d(\mathbf{x})$ is a binary classifier with a scalar output between 0 and 1:

$$\text{(bad=generated=fake)} \quad 0 < d(\mathbf{x}) < 1 \quad \text{(good=real)}.$$

The discriminator can be trained to minimize the binary cross-entropy loss that we studied in Section 1.2. It can be written as

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{x}\sim p_g} \log(1 - d(\mathbf{x})) \to \max_d$$

or equivalently as

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log(1 - d(g(\mathbf{z}))) \to \max_d .$$

The first term is the expected log-probability that the sample $\mathbf{x}$ is classified as a real example when it is taken from the training data set. The second term is the expected log-probability that the sample generated by the model is classified as fake. In practice, we update the discriminator using a mini-batch of N real examples $\mathbf{x}_i$ and N generated examples $g(\mathbf{z}_i)$ to minimize the loss

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log d(\mathbf{x}_i) - \frac{1}{N}\sum_{i=1}^{N} \log(1 - d(g(\mathbf{z}_i))).$$

The generator $g(\mathbf{z})$ is trained to produce samples that are classified by the discriminator as real examples. If $d(\mathbf{x})$ is the output of the discriminator, we want to change the generator in a such way that it produces $\mathbf{x}$ for which $d(\mathbf{x})$ would be close to 1. Therefore, the generator g can be trained by maximizing the following function

$$\mathbb{E}_{\mathbf{x}\sim p_g} \log d(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log d(g(\mathbf{z})) \to \max_g .$$

In practice, when we update the generator using a mini-batch of N generated examples $g(\mathbf{z}_i)$ to minimize the following loss:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log d(g(\mathbf{z}_i)).$$

In principle, there are two ways to train the generator: we can either minimize the probability that the generated example is classified as fake

$$\mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log(1 - d(g(\mathbf{z}))) \to \min_g \tag{9.3}$$

or we can maximize the probability that the generated example is classified as real

$$\mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log(d(g(\mathbf{z}))) \to \max_g . \tag{9.4}$$

Both formulations result in the same fixed point. However, the latter formulation works much better in practice. The reason for that is that at the beginning of training, the generator produces very poor samples and it is very easy for the discriminator to separate them from the real examples with very high confidence. This means that the output of the discriminator on the generated examples would be very close to 0 at the beginning of training.

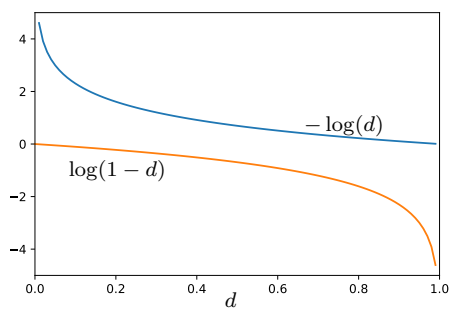


Figure 9.3:

Fig. 9.3 see the two quantities $\log(1 - d)$ and $-\log(d)$ that are used to formulate the optimized objective function in (9.3) and (9.4). At the beginning of training when $d \approx 0$, the quantity $\log(1 - d)$ changes very slowly wrt to its input. That means that this formulation provides very small gradients to the generator and learning can be very slow. In contrast, the value of $\log d$ changes fast wrt to d and therefore this formulation provides much stronger gradients for training the generator.

Let us review the training procedure for training a GAN model. Each training iteration consists of the following steps:

1. Update the discriminator:
   - Sample N examples $\mathbf{x}_i$ from the training set.
   - Generate N samples $g(\mathbf{z}_i)$ using the generator.
   - Compute the binary cross-entropy loss

$$\mathcal{L}_d = -\frac{1}{N}\sum_{i=1}^{N} \log d(\mathbf{x}_i) - \frac{1}{N}\sum_{i=1}^{N} \log(1 - d(g(\mathbf{z}_i)))$$

- Update $\boldsymbol{\theta}_d$ by stochastic gradient descent: $\boldsymbol{\theta}_d \leftarrow \boldsymbol{\theta}_d - \nabla_{\boldsymbol{\theta}_d} \mathcal{L}_d$

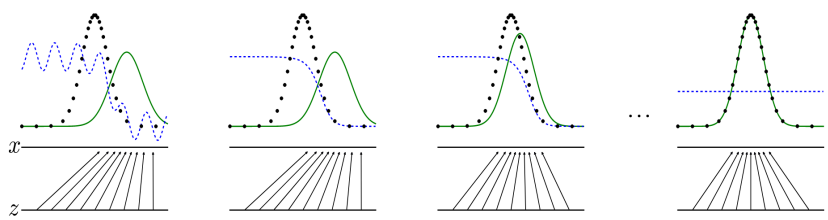
2. Update the generator:

   - Generate N samples $g(\mathbf{z}_i)$ using the generator.
   - Compute the loss function

$$\mathcal{L}_g = -\frac{1}{N} \sum_{i=1}^{N} \log d(g(\mathbf{z}_i))$$

   - Update $\boldsymbol{\theta}_g$ by stochastic gradient descent: $\boldsymbol{\theta}_g \leftarrow \boldsymbol{\theta}_g - \nabla_{\boldsymbol{\theta}_g} \mathcal{L}_g$ (gradients flow through the discriminator).

Fig. 9.4 illustrates the training process of GANs. In this toy example, we want to learn the data distribution represented by the black dots. The generator is initialized such that it produces the distribution shown with the green line in Fig. 9.4a. The discriminator is also initialized somehow.



(a) generator $g(x)$ and discriminator are intialized.

(b) $d(x)$ is trained to discriminate samples from data.

(c) After an update to g, gradient of d has guided $g(z)$ to flow to regions that are more likely to be classified as data.

(d) After several steps of training, g and d reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions $d(x) = 0.5$.

Image from (Goodfellow et al., 2014)

Figure 9.4: Illustration of the training process of GANs.

First, we update the discriminator to separate the training data distribution shown with the black dots and the generator distribution represented by the green line. The updated discriminator is shown with the blue dotted line in Fig. 9.4b. You can see that on the left hand side, the samples are classified as real because there are no generated samples in that area. On the right hand side, the samples are classified as fake because there are more generated samples than the real examples in that area.

Next, the generator is updated by following the feedback from the discriminator. The discriminator suggests that samples are classified as real on the left hand side and therefore the generator is updated such that the generator distribution is shifted towards left.

At the end of the training process, the GAN should converge to the situation shown in Fig. 9.4d when the generator distribution and the data distribution

cannot be distinguished and the discriminator produces the value of 0.5 for all inputs.

We previously considered a collaborative view of GANs where the teacher gives supervision to the student generator network. A more popular view of GANs is that it is a two-player minimax game in which the generator tries to fool the discriminator and the discriminator tries to catch the fakes. The game can be described with the following objective function

$$v(g, d) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log(1 - d(g(\mathbf{z}))). \tag{9.5}$$

The first term is the expected log-probability that data samples are classified as real samples and the second term is the expected log-probability that generated samples are classified as fake samples. The discriminator tries to maximize this function:

$$v(g, d) \to \max_d .$$

From the generator point of view, the first term does not depend on the generator (it is constant wrt the generator) and the second term is the probability that the discriminator classifies generated samples as fake. Therefore, the generator tries to minimize this objective:

$$v(g, d) \to \min_g .$$

The equilibrium (also known as Nash equilibrium) is a saddle point of (9.5). In the Nash equilibrium, both players cannot gain anything by changing their own strategy.

Goodfellow et al. (2014) provide the following theoretical results. For a fixed generator, the optimal discriminator is given

$$d_g^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} .$$

If we use the optimal discriminator $d_g^*(\mathbf{x})$ to tune the generator, then the minimization of the generator loss is equivalent to the minimization of the Jensen-Shannon divergence between the model's distribution p_g and the data distribution p_{data}:

$$JSD\big(p_{\text{data}} \,\|\, p_g\big) = KL\left(p_{\text{data}} \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right) + KL\left(p_g \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right).$$

You can also see that this divergence is symmetric (unlike the KL divergence): if we change the order of p_{data} and p_g, the result remains the same.

It can be shown that the global minimum of the generator loss (9.5) is achieved if and only if the pdf produced by the generative process is equal to the pdf of the data distribution. This analysis indicates that GAN is a proper probabilistic model which learns the data distribution $p(\mathbf{x})$. It is an implicit density model because we do not have an explicit form for p_g, p_g is determined by the implementation of the generator.

Generative adversarial networks is one of the hot topics in the machine learning research. There have been a great number of papers proposing different variants of GANs. One of the major problems is that training of GANs is often unstable: the convergence may be slow or even difficult to achieve. One typical problem when training GANs is that the generator starts producing the same sample or different slight variations of the same sample. This problem is known as *mode collapse*.

9.1.1 Deep convolutional GAN

Deep convolutional GAN (DCGAN, Radford et al., 2015) is one of the early GAN models that is relatively easy to train. The authors propose a few architectural decisions that make the training procedure stable:

- Replace any pooling layers with strided convolutions (discriminator).
- Use transposed convolutions in the generator.
- Remove fully connected hidden layers.
- Use batchnorm in both the generator and the discriminator.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

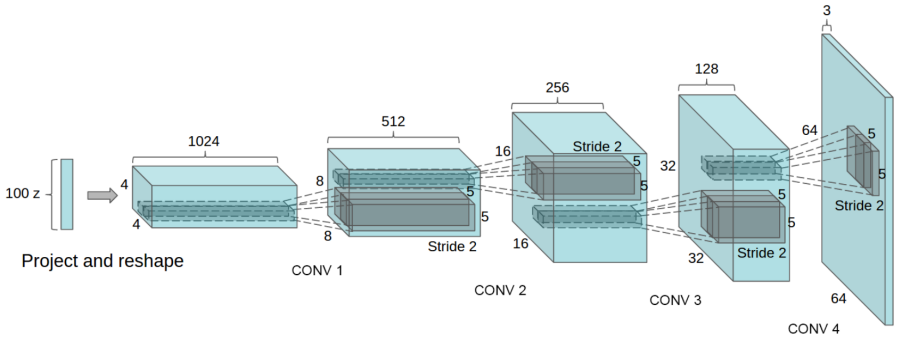


Figure 9.5: The generator of the deep convolutional GAN.

9.2 Improving stability of GAN training

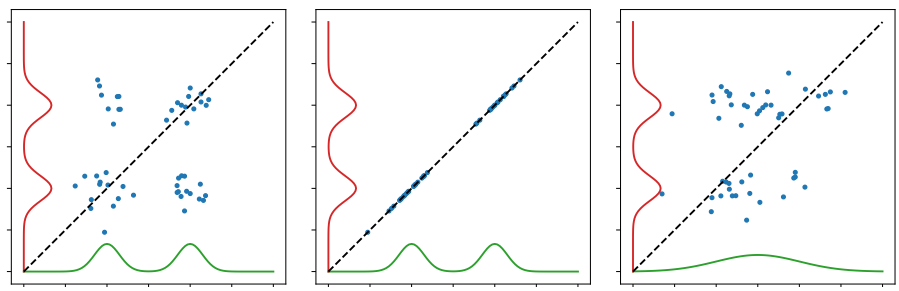
9.2.1 Wasserstein GAN

In the conventional GAN, the distance between the data distribution p_{data} and the model distribution p_g was measured with the Jensen-Shannon divergence. Arjovsky et al., (2017) propose to compare the two distributions using a different distance called Earth-Mover distance or Wasserstein-1. The distance is defined as

$$W(p_{\text{data}}, p_g) = \inf_{\gamma \in \prod(p_{\text{data}}, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

where $\prod(p_{\text{data}}, p_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are p_{data} and p_g. Loosely speaking, Wasserstein-1 is the smallest value of the expected distance between x and y where x and y are sampled from any joint distribution such the marginal distributions are p_{data} and p_g.

Let us gain some intuition about this quantity. Suppose that we have two distributions: one is a bimodal distribution shown on the y-axis and the other one is a bimodal distribution shown on the x-axis in Fig. 9.6a. Among all possible joint distributions such that their marginals are given by the red and the green lines, we want to find a joint distribution such that if we draw samples from this distribution the difference between the x and y coordinates of those samples is minimized.



(a) (b) (c)

Figure 9.6: The Earth-Mover distance or Wasserstein-1 is defined as the minimum expected distance to the straight line $y = x$ among all joint distributions whose marginals are equal to the compared distributions shown with the red and green lines.

The difference between coordinates x and y of the same sample is given by the distance from the sample to the straight line $y = x$. You can see that for the joint distribution in Fig. 9.6a, the expected distance from the samples to the dashed line is non-zero. However, if we use the joint distribution with the same marginals which is shown in Fig. 9.6b, then the distance to the dashed line is exactly zero. Among all possible joint distributions with the same marginals, we want to find the one that minimizes this distance. The minimum of the expected distance is Wasserstein-1.

In the example in Fig. 9.6a-b, the two distributions are equal and therefore we can find a joint distribution that produces exactly zero distances. If the two distributions are not equal (see, e.g., Fig. 9.6c), then we cannot find a joint distribution such that the expected distance from the samples to the line $y = x$ is zero. The minimum distance among all possible joint distributions can be used as the distance between the two probability distributions.

The authors provide the following arguments to why it is better to use the Earth-Mover distance. Suppose that we want to compute the distance between distributions $q(\mathbf{x})$ and $p_\theta(\mathbf{x})$ shown in Fig. 9.7. x_2 is uniformly distributed between -1 and 1 and x_1 is a constant in both distributions: $x_1 = 0$ for $q(\mathbf{x})$ and $x_1 = \theta$ for $p_\theta(\mathbf{x})$. We compute the distance between the two distributions as a function of θ. When $\theta = 0$, both the Jensen-Shannon divergence and the Earth-Mover distance are equal to zero. But for non-zero θ, the JSD has the same value regardless of the value of θ. In contrast, the Earth-Mover distance grows linearly when θ moves away from zero. Therefore, this distance may provide a

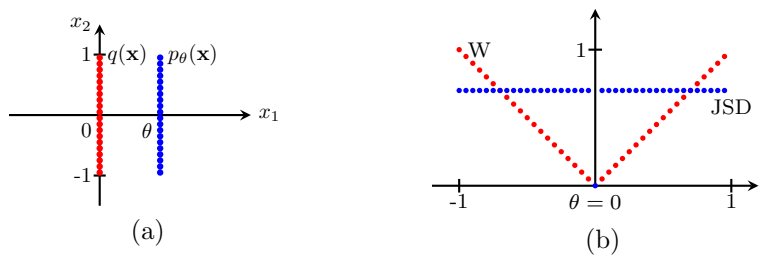


Figure 9.7:

stronger training signal.

The problem with the Earth-Mover distance is that it is not tractable. To compute the distance between two distributions, we need to solve a complex optimization problem (as illustrated in Fig. 9.6). This is not convenient because we cannot compute the distance efficiently and therefore we cannot minimize it.

However, the following equality helps us estimate this distance:

$$W(p_{\text{data}}, p_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim p_{\text{data}}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)]. \tag{9.6}$$

The equation says that if we take a family of all functions f that are 1-Lipschitz continuous $\|f\|_L \leq 1$, then Wasserstein-1 is the supremum of the difference between the expected values of $f(x)$ computed for the two distributions p_{data} and p_g. The condition that function f is 1-Lipschitz continuous is given by:

$$\|f(x_2) - f(x_1)\| \leq \|x_2 - x_1\| .$$

This constraint basically means that we limit the magnitude of the gradient of f. Intuitively, if we do not restrict functions f in any way, then the difference of the two expected values can be made infinitely large. The condition $\|f\|_L \leq 1$ limits the set of possible functions making the distance well defined.

Equation (9.6) means that in order to estimate the Wasserstein-1 distance, we need to find function f that maximizes the difference of the expected values of $f(x)$ under the two distributions p_{data} and p_g, with the constraint on the derivatives of function f. If instead of 1-Lipschitz functions, we use K-Lipschitz functions, which is a set of function that satisfy the following constraint

$$\|f(x_2) - f(x_1)\| \leq K \|x_2 - x_1\| ,$$

for some K, then we can compute the distance up to a constant multiplier K:

$$K \cdot W(p_{\text{data}}, p_g) = \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_{\text{data}}}[f(x)] - \mathbb{E}_{x \sim p_g}[f(x)]. \tag{9.7}$$

This is totally fine for our purposes because we are not interested in the exact values of the Wasserstein-1 distance, we can use (9.7) to measure the difference between two distributions.

The authors propose to model function f with a neural network. In order to guarantee that the function modeled by the neural network is K-Lipschitz, they

propose to limit the weights w such that their values are constrained to be in a pre-defined range $w_i \in [-c, c]$. Then, the distance between the two distributions can be estimated by solving the following optimization problem:

$$\max_{w \in \mathcal{W}} \mathbb{E}_{x \sim p_{\text{data}}}[f_w(x)] - \mathbb{E}_{x \sim p_g}[f_w(x)], \qquad \mathcal{W} = \{w_i \in [-c, c]\}.$$

Every training iteration of Wasserstein GAN (Arjovsky et al., 2017) consists of the following steps.

1. Update the discriminator:
   - Sample N examples $\mathbf{x}_i$ from the training set.
   - Generate N samples $g_\theta(\mathbf{z}_i)$ using the generator.
   - Compute loss
   $$\mathcal{L}_d = \frac{1}{N}\sum_{i=1}^{N} f_w(g_\theta(\mathbf{z}_i)) - \frac{1}{N}\sum_{i=1}^{N} f_w(\mathbf{x}_i)$$
   - Update w by stochastic gradient descent: $w \leftarrow w - \nabla_w \mathcal{L}_d$
   - Clip the weights $w_i \leftarrow \text{clip}(w_i, -c, c)$.

2. Update the generator:
   - Generate N samples using the generator.
   - Compute loss:
   $$\mathcal{L}_g = -\frac{1}{N}\sum_{i=1}^{N} f_w(g_\theta(\mathbf{z}^{(i)}))$$
   - Update $\boldsymbol{\theta}_g$ by stochastic gradient descent: $\boldsymbol{\theta}_g \leftarrow \boldsymbol{\theta}_g - \nabla_{\boldsymbol{\theta}_g} \mathcal{L}_g$ (gradients flow through the discriminator).

The second version of the Wasserstein GAN model was proposed by Gulrajani et al., (2017). The authors use a slightly different strategy for restricting the discriminator. They argue that clipping of the weights creates optimization difficulties such that without careful tuning of the clipping threshold c, the gradients can vanish or explode. The second problem is that clipping of the weights restricts the discriminator too much, such that it cannot learn complex functions. Instead of weight clipping, the authors proposed to add a gradient penalty term to the discriminator loss. This extra term penalizes the difference of the gradient norm and 1, which is the value that was theoretically justified by the authors:

$$\mathcal{L}_{d,gp} = \mathcal{L}_d + \underbrace{\lambda \mathbb{E}_{\hat{x} \sim p_{\hat{x}}}(\|\nabla_{\hat{x}} f_w(\hat{x})\|_2 - 1)^2}_{\text{gradient penalty}}.$$

The authors propose to compute the gradient penalty at locations $\hat{x}$ which are sampled uniformly along the straight lines that connect pairs of points sampled from p_{data} and p_g. Fig. 9.8 shows that Wasserstein GAN with gradient penalty (WGAN-GP) can provide more stable results compared to other GAN models.

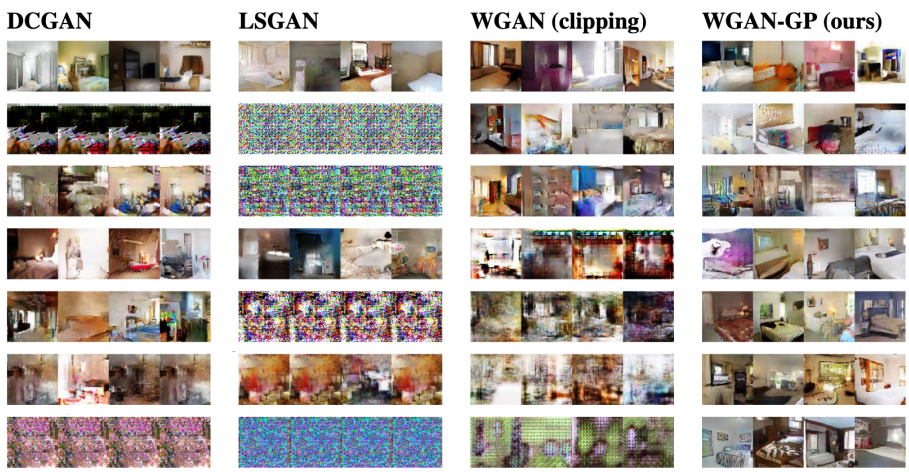


Figure 9.8: Genrated samples with WGAN-GP and alternative GAN models.

9.2.2 Spectral Normalization

The motivation of spectral normalization (Miyato et al., 2018) is similar to the one of Wasserstein GAN. Suppose that we use the conventional form of the GAN objective function

$$v(g, d) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log(1 - d(g(\mathbf{z}))). \tag{9.8}$$

Then, the optimal discriminator is given by

$$d_g^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}} + p_g(\mathbf{x})} = \text{sigmoid}(f^*(\mathbf{x})),$$

where

$$f^*(\mathbf{x}) = \log p_{\text{data}}(\mathbf{x}) - \log p_g(\mathbf{x})\,.$$

In order to update the generator, we need to differentiate the discriminator wrt its inputs. The gradient is given by the following expression

$$\nabla_{\mathbf{x}} f^*(\mathbf{x}) = \frac{1}{p_{\text{data}}(\mathbf{x})} \nabla_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) - \frac{1}{p_g(\mathbf{x})} \nabla_{\mathbf{x}} p_g(\mathbf{x})\,.$$

For samples $\mathbf{x}$ in the low-density areas, where either p_{data} or p_g is small, the derivative can have very large magnitudes or can even be incomputable.

The authors propose to fix this problem by introducing a regularity condition on the derivatives of the discriminator. Inspired by Wasserstein GAN, the authors propose to search for the discriminator in the set of K-Lipschitz continuous functions:

$$v(g, d) \to \max_d, \quad \text{s.t.} \;\; \|d\|_L \leq K$$

where $\|d\|_L \leq K$ means all functions f such that

$$\|d(\mathbf{x}) - d(\mathbf{x}')\| \leq K \|\mathbf{x} - \mathbf{x}'\|\,. \tag{9.9}$$

Recall that the intuition behind the K-Lipschitz continuity is that we restrict the magnitudes of the derivatives. This means that we do not want the discriminator to change too fast. If the discriminator changes slowly, it provides more constructive feedback to the generator and this can stabilize training. Note that we do not change the objective function (like in Wasserstein GAN) and can use the conventional GAN objective (9.8).

Miyato et al., (2018) propose to guarantee (9.9) by using *spectral normalization*. For a linear function

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x},$$

the Lipschitz norm is given by the spectral norm $\sigma(\mathbf{W})$

$$\|f\|_L = \sigma(\mathbf{W}),$$

where $\sigma(\mathbf{W})$ is defined as the largest singular value of $\mathbf{W}$. The idea of spectral normalization is to restrict $\mathbf{W}$ to have the spectral norm of 1, which can be achieved by dividing $\mathbf{W}$ by its spectral norm:

$$f(\mathbf{x}) = \frac{\mathbf{W}}{\sigma(\mathbf{W})}\mathbf{x}\,.$$

The authors show that if the discriminator is a stack of linear layers with common activation functions such as relu and each linear layer satisfies $\sigma(\mathbf{W}) = 1$, then the discriminator is guaranteed to be 1-Lipschitz. This suggests that if one uses spectral normalization in every linear layer of the network, then the constrained (9.9) is satisfied.

To implement spectral normalization, we have to compute the largest singular value of a matrix and we want this computation to be efficient and differentiable. The authors propose to use the power iteration method for that.

The classical application of the power iteration algorithm is to find the largest eigenvalue of a square matrix $\mathbf{W}$. The algorithm iterates by computing the matrix-vector product of matrix $\mathbf{W}\mathbf{v}$ and normalizing the resulting vector to unit norm:

$$\mathbf{v} \leftarrow \frac{\mathbf{W}\mathbf{v}}{\|\mathbf{W}\mathbf{v}\|}.$$

This algorithm converges to the eigenvector which corresponds to the largest eigenvalue.

In a linear layer of a neural network, the matrix $\mathbf{W}$ of weights is generally not square and therefore we need two vectors $\mathbf{v}$ and $\mathbf{u}$ in the power iteration method:

$$\mathbf{v} \leftarrow \frac{\mathbf{W}^\top\mathbf{u}}{\|\mathbf{W}^\top\mathbf{u}\|}, \qquad \mathbf{u} \leftarrow \frac{\mathbf{W}\mathbf{v}}{\|\mathbf{W}\mathbf{v}\|}.$$

Then, the estimate of the largest singular value is given by

$$\sigma(\mathbf{W}) = \mathbf{u}^\top\mathbf{W}\mathbf{v}.$$

The authors propose to initialize vectors $\mathbf{v}$ and $\mathbf{u}$ randomly and then update them on each training iteration using (9.2.2). Note that $\mathbf{v}$ and $\mathbf{u}$ are not model parameters, they act more like running statistics in the batch normalization layer.

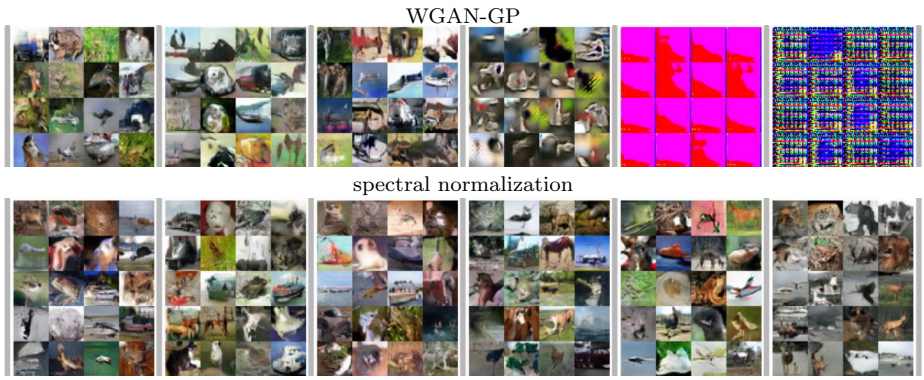


Figure 9.9: Generated samples obtained with different model architectures. The first row corresponds to WGAN-GP. The second row corresponds to spectral normalization. Spectral normalization provides more stable results.

For convolutional layers whose weights are not matrices but four-dimensional tensors $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}} \times h \times w}$, the authors propose to reshape the tensors into two-dimensional matrices and apply spectral normalization to the reshaped tensors. Fig. 9.9 shows that GAN with spectral normalization can produces better results compared to WGAN-GP.

9.2.3 Zero-centered gradient penalties

Mescheder et al. (2018) present a very insightful study of the convergence in the GAN optimization problem. They use a very simple example in which the true data distribution is a Dirac-distribution concentrated at 0 (see Fig. 9.10). The generator distribution $p_\theta = \delta_\theta$ is a Dirac-distribution concentrated at θ, which means that the generator has only one parameter. The discriminator is a linear function $d_\phi(x) = \phi x$ with a single parameter ϕ.

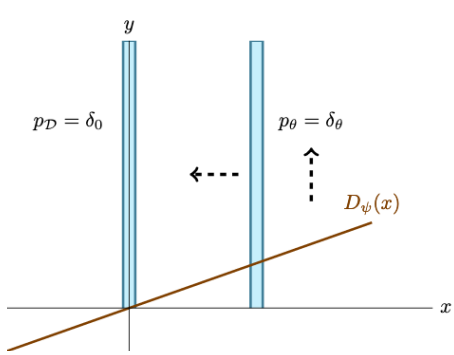


Figure 9.10: A toy problem considered by Mescheder et al. (2018). The data distribution is a Dirac-distribution concentrated at 0. The generator distribution p_θ is a Dirac-distribution δ_θ concentrated at θ. The discriminator is a linear function $d_\phi(x) = \phi x$.

Different variants of the GAN objective function can be written in the fol-

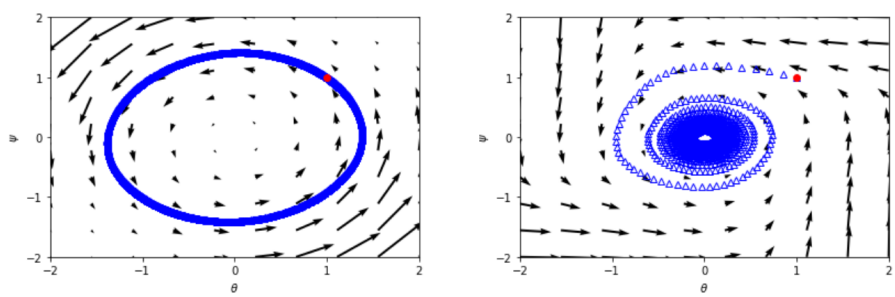
lowing form

$$v(\theta, \phi) = \mathbb{E}_{p_{\text{data}}(x)}[f(-d_\phi(x))] + \mathbb{E}_{p(z)}[f(d_\phi(g_\theta(z)))]$$
$$v(\theta, \phi) \to \min_\theta \max_\phi$$

where $v(\theta, \phi)$ is a function of only two parameters in this toy example. Using $f(t) = -\log(1 + \exp(-t))$ gives the conventional GAN objective (9.5):

$$v(g, d) = \mathbb{E}_{p_{\text{data}}(x)} \log d(x) + \mathbb{E}_{p_z(z)} \log(1 - d(g(z))).$$

With the identity function $f(t) = t$, we get the Wasserstein GAN objective. We want to maximize this objective function wrt the parameter ϕ of the discriminator and to minimize it wrt the parameter θ of the generator. In the experiments, parameters θ, ϕ are optimized by alternating gradient descent and the optimization trajectory can be visualized.



(a) Unregularized GAN training with objective (9.3) does not always converge to the Nash-equilibrium.

(b) GAN with a non-saturating objective for the generator converges, albeit with an extremely slow convergence rate.

Figure 9.11: Optimization trajectories obtained using alternating gradient descent for the conventional GAN in the toy problem in Fig. 9.10.

Fig. 9.11 shows the optimization trajectories for the conventional GAN with objective (9.3) (left plot) and objective (9.4) (right plot). You can see that the original GAN does not always converge to the Nash-equilibrium. Interestingly, when the generator is at the optimal solution $\theta = 0$, the discriminator pushes the generator away from it. GAN with a non-saturating objective (9.4) for the generator converges but with an extremely slow convergence rate.

WGAN and WGAN-GP with a finite number of discriminator updates per generator update do not always converge to the equilibrium point (see Fig. 9.12). Note that in WGAN the absolute values of ϕ are limited to restrict the magnitude of the discriminator derivatives. Therefore, ϕ is restricted to be inside the white area in Fig. 9.12. However, it does not help to achieve convergence.

The authors observe that penalizing the gradients of the discriminator usually has a positive effect on convergence because the discriminator is penalized for deviating from the Nash-equilibrium. The authors proposed two kinds of

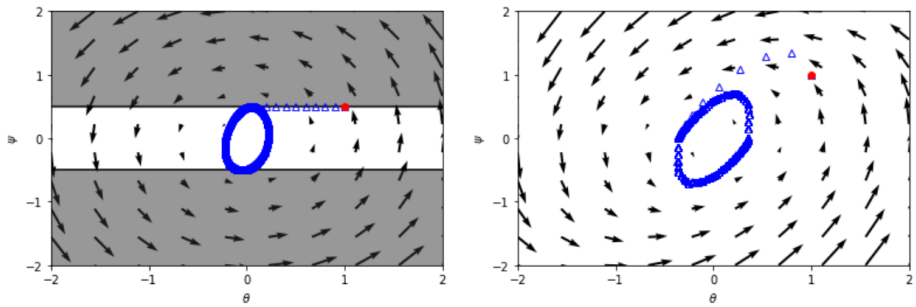


Figure 9.12: Optimization trajectories obtained for Wasserstein GAN and WGAN-GP in the toy problem in Fig. 9.10. Optimization is done using alternating gradient descent with $n_d = 5$ discriminator updates per generator update.

the *zero-centered*[1] gradient penalty: either penalize gradients on real data

$$R_1 = \frac{\gamma}{2}\mathbb{E}_{x\sim p_r}(\|\nabla d(x)\|^2)$$

or penalize gradients on generated samples

$$R_2 = \frac{\gamma}{2}\mathbb{E}_{x\sim p_g}(\|\nabla d(x)\|^2).$$

For the toy problem in Fig. 9.10, the two penalties are equivalent because the gradient does not depend on x. Fig. 9.13 shows that adding this penalty to the original GAN objective helps to achieve convergence. We see that if the regularization parameter γ is large enough (see the plot on the right), then the vector field does not have a rotational component near the Nash-equilibrium and we get a behavior like in a normal optimization problem.

Zero-centered gradient penalties provide a very effective way to stabilize training of GANs. R_1 is perhaps used more often than R_2.

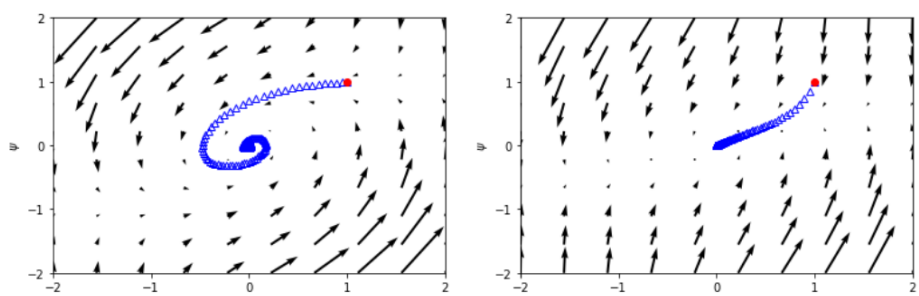


Figure 9.13: Optimization trajectories obtained using alternating gradient descent for the conventional GAN with zero-centered gradient penalty in the toy problem in Fig. 9.10. The two plots correspond to different values of γ.

[1]The proposed gradient penalty is called zero-centered because it penalizes the norm of the gradient in contrast to WGAN-GP which penalizes the deviation of the gradient norm from 1.

9.3 Improved network architectures

9.3.1 Progressive growing (ProGAN)

The motivation of ProGAN (Karras et al., 2018) is that learning a generative model for high-resolution images is a very difficult task. In order to generate images of high quality, the model has to capture both the global structure and local details of images, which is important for achieving photo-realism. On the other hand, learning a generative model of low-resolution images is a much simpler problem.

The idea of ProGAN is to solve a simple task first and then gradually increase the complexity of the learning problem. ProGAN starts with training a low-resolution generative model (see Fig. 9.14). Once the model is trained, the resolution of the training data is doubled and the resolution of the model is increased by adding an additional layer to the generator and the discriminator. Once the new model is trained, the resolution is increased again and this process continues until we get a generative model for high-resolution images.

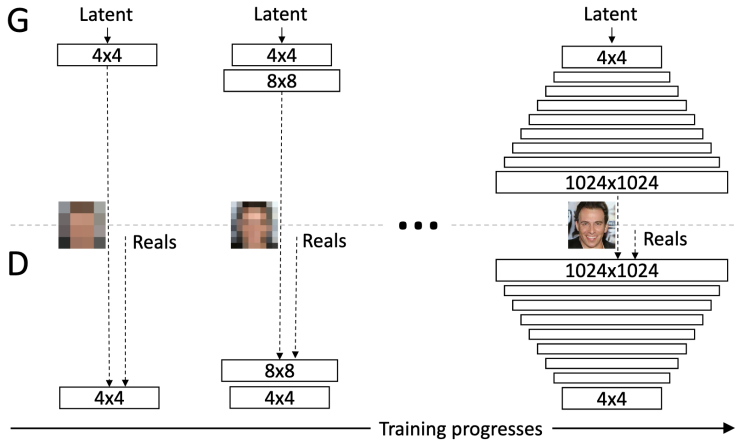


Figure 9.14: The training process of ProGAN (Karras et al., 2018).

The first benefit of this approach is that the training problem is simplified. The second benefit is that the overall training time can be reduced significantly because many training iterations are done at lower resolutions.

When the resolution of images is doubled, the new layers are added to the model gradually to guarantee a smooth transition between the resolutions and more stable training. Supposed that we have a model that is trained to generate 16×16 images. In Fig. 9.15a, the block 16×16 represents the last convolutional layer of the generator. We add a new block (marked as 2x in Fig. 9.15b) which doubles the resolution by nearest neighbor filtering. We also add a new convolutional layer which outputs a 32×32 map and a skip connection around that layer. The output of the generator is the sum of the outputs of the convolutional layer and the skip connection with coefficients α and $1-\alpha$, repestively. Initially, α is set to zero, which means that we do not use the new convolutional layer but we take the signal from the upscaled version of the 16×16 image. During

training, the value of α is gradually increased from 0 to 1, which increases the contribution of the new convolutional layer to the generated image.

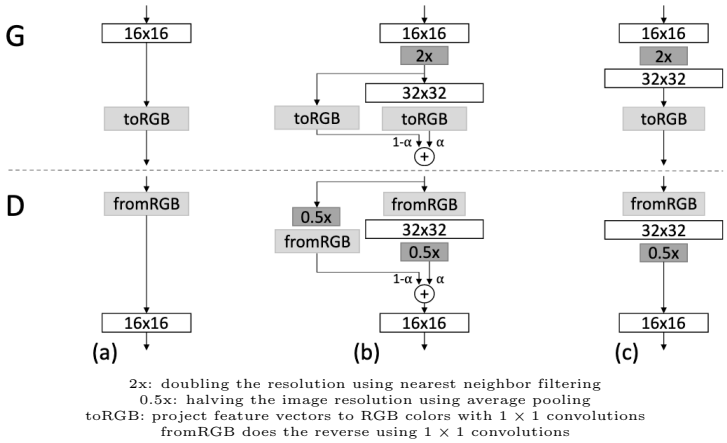


Figure 9.15: When the resolution of images is doubled, the new layers are added to the ProGAN model gradually.

Fig. 9.16 shows images generated with ProGAN. This was probably the first model that was able to generate high-quality high-resolution images of human faces.

Figure 9.16: Images generated by ProGAN.

9.3.2 Self-Attention GAN (SAGAN)

Self-Attention GAN (Zhang et al., 2018) proposes an architecture of the generator with a self-attention block. The computations in the block are inspired by the self-attention block of the transformer model.

Fig. 9.17 illustrates the structure of the proposed self-attention block. The block takes as input a 2d map just like a standard convolutional layer. The block processes the input with 1×1 convolutions to create the keys, queries and values. Then, the attention coefficients are computed by applying the softmax function to the product of the keys and queries (just like in transformers). Finally the attention coefficients select the values and the result is processed by another 1×1 convolutional layer. This block essentially implements a dot-product attention mechanism in which each pixel is treated as a separate position.

Fig. 9.18 shows some images generated with a large-scale self-attention GAN model (Brock et al., 2018).

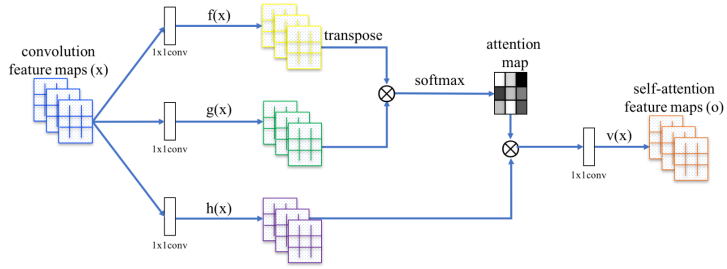


Figure 9.17: Self-attention block of SAGAN (Zhang et al., 2018).

Figure 9.18: Image samples generated by a large-scale SAGAN.

9.3.3 Style-Based Generators (StyleGAN)

StyleGAN (Karras et al., 2018) proposes an architecture of the generator inspired by the style transfer literature (Huang and Belongie, 2017) which studies how to modify an image using the style (low-level textures) of another image.

The structure of the generator is presented in Fig. 9.19. The generator consists of two networks: the synthesis network and the mapping network. The synthesis network is a convolutional network that starts with a deterministic low-resolution input. It contains several blocks called adaptive instance normalization (marked as AdaIn on the in Fig. 9.19) which perform the following computations:

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}.$$

Each channel is normalized to zero mean and unit variance and then scaled and shifted using style vectors $\mathbf{y}_{s,i}, \mathbf{y}_{b,i}$. The style vectors for the different AdaIn blocks in the synthesis network are produced by the mapping network. The mapping network is simply a multilayer perceptron that takes as input a noise vector $\mathbf{z}$. There are also additional noise instances which are injected directly into the synthesis network.

StyleGAN improves the quality of the generated images compared to traditional architecture of the generator (see Fig. 9.19). Note that assessing the quality of the generated images is not a trivial task. One popular metric is called Fréchet Inception distance (FID) (Heusel et al., 2017). It is a Fréchet distance between two Gaussian distributions

$$\text{FID} = \|\mathbf{m}_r - \mathbf{m}_g\|_2^2 + \text{Tr}(\mathbf{C}_r + \mathbf{C}_g - 2(\mathbf{C}_r\mathbf{C}_g)^{1/2})$$

such that the mean and the covariance matrix of the two distributions are computed using real examples $(\mathbf{m}_r, \mathbf{C}_r)$ and the generated examples $(\mathbf{m}_g, \mathbf{C}_g)$.

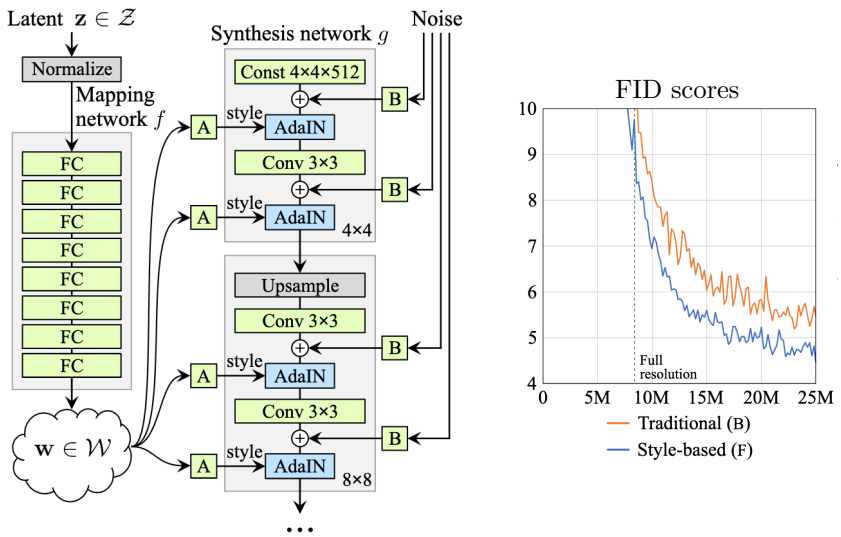


Figure 9.19: Left: The structure of the StyleGAN generator (Karras et al., 2018). Right: StyleGAN improves the FID score of the generated images compared to traditional architectures.

The mean and the covariance matrix are not computed using raw images. They are computed using activations produced by a neural network pre-trained to perform the ImageNet classification task. Heusel et al. (2017) used an Inception-v3 network for computing the features and that is why the metric was called Fréchet *Inception* distance.

9.4 Image-to-Image translation with conditional GANs

Isola et al., (2017) consider the task of generating an image with the condition provided by another image. Fig. 9.20 shows some examples of such tasks: produce a photo-realistic image from a segmentation map, convert a black-and-white image into a color image, change the time of the day of a given scene or create a photo-realistic image given the contours of an object.

The authors propose to solve the image-to-image translation task using generative adversarial networks (see Fig. 9.21). The task is to generate image $\mathbf{y}$ conditioned on image $\mathbf{x}$. The generator takes as inputs image $\mathbf{x}$ and the noise vector $\mathbf{z}$ and produces image $\mathbf{y}$. The discriminator takes as inputs both the generated image $\mathbf{y}$ and the conditioning image $\mathbf{x}$ and classifies $\mathbf{y}$ as being real or fake.

The objective function is formulated similarly to the conventional GAN.

$$v(g,d) = \mathbb{E}_{\mathbf{x},\mathbf{y}}[\log d(\mathbf{x},\mathbf{y})] + \mathbb{E}_{\mathbf{x},\mathbf{z}}[\log(1 - d(\mathbf{x}, g(\mathbf{x},\mathbf{z}))]$$

where we use the conditioning image $\mathbf{x}$ as an extra input of the generator and the discriminator. The discriminator is trained in the conventional way. For training the generator, the authors use a sum of two losses:

$$\mathcal{L}(g) = v(g,d) + \lambda \mathbb{E}_{\mathbf{x},\mathbf{y},\mathbf{z}} \left\| \mathbf{y} - g(\mathbf{x},\mathbf{z}) \right\|_1 . \tag{9.10}$$

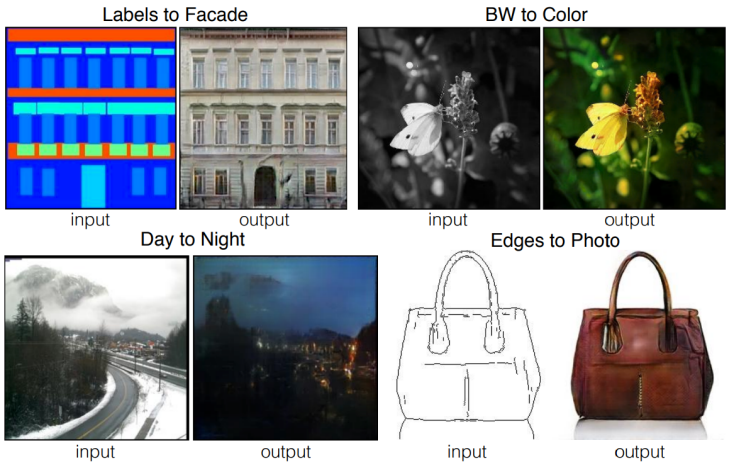


Figure 9.20: Examples of image-to-image translation tasks: produce an image from a semantic segmentation map, produce a color image from on a black-and-white image, change the time of the day of a given scene, create a photo-realistic image given the contours of an object.

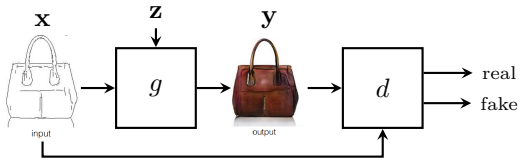


Figure 9.21: Generative adversarial networks for the image-to-image translation task.

The first loss is the same as in the conventional GAN. The second loss encourages the generated sample $g(\mathbf{x}, \mathbf{z})$ to be close to the target example $\mathbf{y}$ from the training set. The authors use the L1 distance in this loss.

The generator in that model has the architecture of the U-net (see Fig. 9.23). There is a contracting path which produces high-level features as a low-resolution two-dimensional map and an expansive path which increases the resolution and re-uses representations developed in the contracting path via skip connections. The source of noise is the dropout in the intermediate layers of the U-Net.

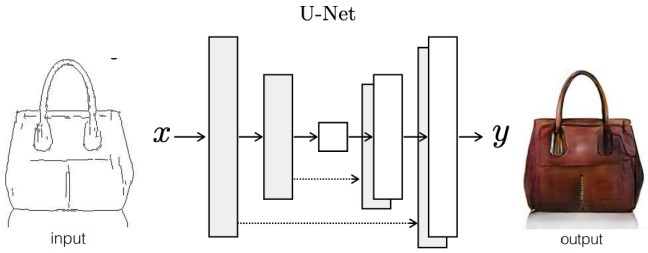


Figure 9.22: The structure of the generator in the Pixel2Pixel model y Isola et al., (2017).

The generator trained using the sum of two losses (9.10) produces better results compared to a model which uses only the second term in (9.10). Fig. 9.23 shows examples of the generated images in the tasks of generating an aerial photo from a map and generating a map from an aerial photo.

Figure 9.23: Image samples generated with the Pixel2Pixel model. Left: The task is to generate an aerial photo from a map. Right: The task is to generate a map from an aerial photo.

Chapter 10

Autoregressive models

In this chapter, we consider another class of explicit generative models: models with an explicit parametric form for the probability density function $p(\mathbf{x})$. In the autoregressive models, the probability distribution $p(\mathbf{x})$ is written using the chain rule of probabilities:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i | x_1, ..., x_{i-1})$$

where x_i are elements of vector $\mathbf{x}$. This is a generic probabilistic model because any probability distribution can be written in this form.

10.1 Convolutional autoregressive models

10.1.1 Autoregressive modeling of sequential data

Consider modeling of sequential data such as time series. For example, a mono speech signal is a one-dimensional time series, which may have a waveform like the one shown in Fig. 10.1. Text can also be seen as a time series of tokens, symbols or even bytes.

Figure 10.1: A waveform of a speech signal.

We can model the joint distribution of the elements in a sequence $\mathbf{x}_1, ..., \mathbf{x}_m$ using the chain rule of probabilities:

$$p(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m) = p(\mathbf{x}_1) \prod_{i=2}^{m} p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1). \tag{10.1}$$

We have seen autoregressive models in the previous chapters. Recall the decoders in the sequence-to-sequence models for neural machine translation (Fig. 10.2) The decoders were autoregressive models which produced the conditional probabilities over possible tokens in the current position i given the tokens in the previous positions and the context $\mathbf{z}$ provided by the encoder. For unsupervised learning, we use a similar type of autoregressive models with the difference that we we do not have to use the context produced by the encoder.

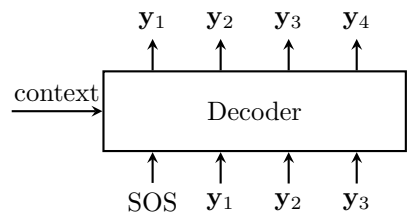


Figure 10.2: An autoregressive decoder in a sequence-to-sequence model.

We used different types of autoregressive decoders: implemented with RNNs (Section 5.5), CNNs (Section 6.2) and transformers (Section 6.3). We discussed that training an RNN-based model can be challenging and therefore we start with convolutional autoregressive models in this chapter.

When we considered convolutional sequence-to-sequence models, we discussed how to model conditional distributions $p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$ in (10.1) effectively using a single convolutional neural network. We used the following trick to guarantee the autoregressive structure of our model (which is illustrated in Fig. 10.3): 1) we used causal or shifted convolutions, 2) the input and target sequences were two shifted versions of the same sequence. The elegance of this model is that during training, we compute predictions for all positions *in parallel* using the same neural network. At test time (when we want to generate a new sequence), we produce the elements of the generated sequence one element at a time.

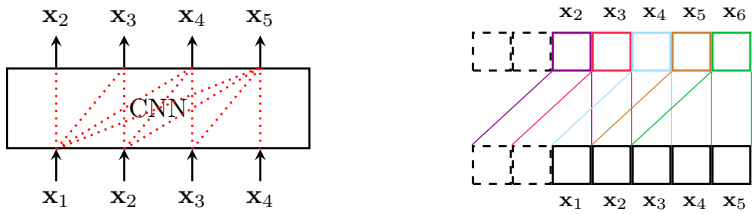


Figure 10.3: Modeling conditional probabalities $p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$ with a convolutional neural network. Left: Illustration of receptive fields in an autoregressive CNN model. Right: Shifted one-dimensional convolutions.

One of the most popular autoregressive models of speech is called WaveNet (van den Oord et al., 2016). It is an autoregressive model that predicts the next value of a speech signal given a window of the previous values. The model is implemented as a stack of causal one-dimensional convolutional layers. As we discussed previously, the challenge of modeling speech data is that speech signals usually have very high sampling rates, which makes it difficult to capture long-term dependencies with standard convolutions. For that reason, WaveNet

uses dilated convolutions which allow fast growth of the receptive field (see Fig. 10.4).

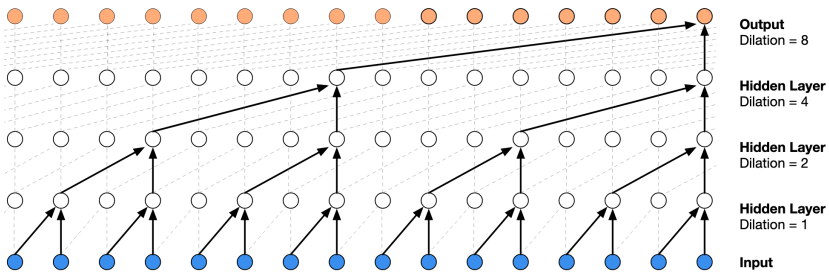


Figure 10.4: WaveNet (van den Oord et al., 2016) is an autoregressive model of speech.

10.1.2 PixelCNN: Autoregressive model of images

We can treat $n \times n$ images as one-dimensional sequences $x_1, ..., x_{n^2}$ where pixels are taken, for example, row by row from the top left corner to the bottom right corner (see Fig. 10.5). Then, we can build an autoregressive model which contains the probabilities of the intensity of the current pixel given the previous pixels of the image:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1}),$$

where $p(x_i|x_1, ..., x_{i-1})$ is the probability distribution over pixel intensities x_i for pixel i given the intensities $x_1, ..., x_{i-1}$ of the previous pixels.

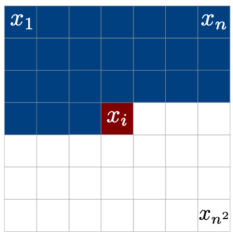


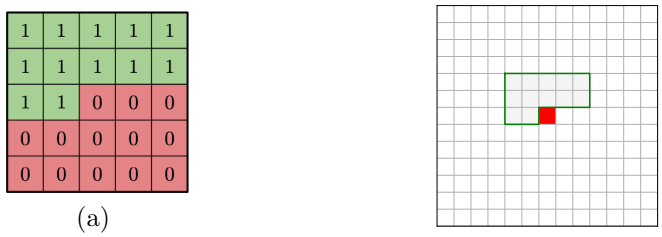
Figure 10.5: In autoregressive models of images, an image is viewed as a sequence of pixels.

The idea of building autoregressive models of images is quite old (Larochelle and Murray, 2011; Germain et al., 2015; Uria et al., 2016). In this section, we consider the model called PixelCNN (van den Oord et al., 2016a; 2016b) which is one of the most well known models of this type.

To build an autoregressive model of images, we need a neural network that computes conditional distributions $p(x_i|x_1, ..., x_{i-1})$ of the current pixel given the previous pixels. And it would be nice to compute those probabilities for all pixels in parallel using the same model, just like it was done for one-dimensional sequences. This would greatly increase the speed of training.

In the PixelCNN model, the authors propose to construct a convolutional neural network using a stack of *masked* 2d convolutional layers. The masked

convolutional layers use kernels with the structure shown in Fig. 10.6a: the weights in the green locations are allowed to have non-zero values but the weights in the red locations are forced to be exactly zero.



(a) (b)

Figure 10.6: Left: Kernel mask in a masked 2d convolutional layer. Right:

Let us investigate the receptive fields in a network that contains a stack of such masked convolutional layers. Suppose that we have only one layer in our network. Then, the receptive field of the pixel marked with the red color in Fig. 10.6b is given by the pixels inside the green polygon. The other pixels in the 5×5 neighborhood cannot affect the value of that pixel because the corresponding weights of the kernel are set to zeros. This structure of the receptive field guarantees that the red pixel in the output layer is not affected by any pixel from below and to the right. Therefore, this layer preserves the autoregressive structure shown in Fig. 10.5.

Suppose that our network is a stack of two masked convolutional layers (Fig. 10.7). The first layer has the same kernel mask as shown in Fig. 10.6a. The second layer is slightly different: it has a non-zero value of the kernel in the center. Let us now investigate the receptive field of the red pixel in the output layer. The blue polygon in Fig. 10.8 specifies which pixels in the output of the first layer affect the red pixel. Note that because we have a non-zero weight in the middle of the kernel, the blue area also includes the pixel in the position of the red pixel. These are the pixels in the output of the first layer that affect the red pixel.

Now we can find the receptive field of the red pixel by investigating which pixels in the input affect the pixels inside the blue polygon in the output of the first layer. We can slide the kernel mask across the pixels inside the blue polygon to find the overall receptive field. It is given by the grey pixels in the last subplot of Fig. 10.8. The intensities of the pixels represent the number of routes from a particular pixel to the red pixel. We can see that the resulting receptive field agrees with the desired autoregressive structure. The red pixel does not use information from below and from the right. If we stack more masked 2d convolutional layers, the desired autoregressive structure is still preserved.

In the PixelCNN model, every conditional distribution $p(x_i|x_1, ..., x_{i-1})$ is modeled as a multinomial distribution with 256 possible values (8-bit representation of a pixel value). Therefore, each pixel is classified during training into one of the 256 classes which correspond to different pixel intensities. Therefore,

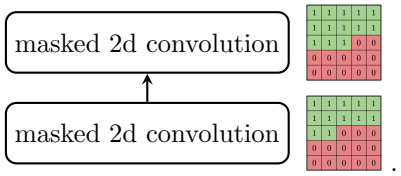


Figure 10.7: A stack of two masked 2d convolutional layers

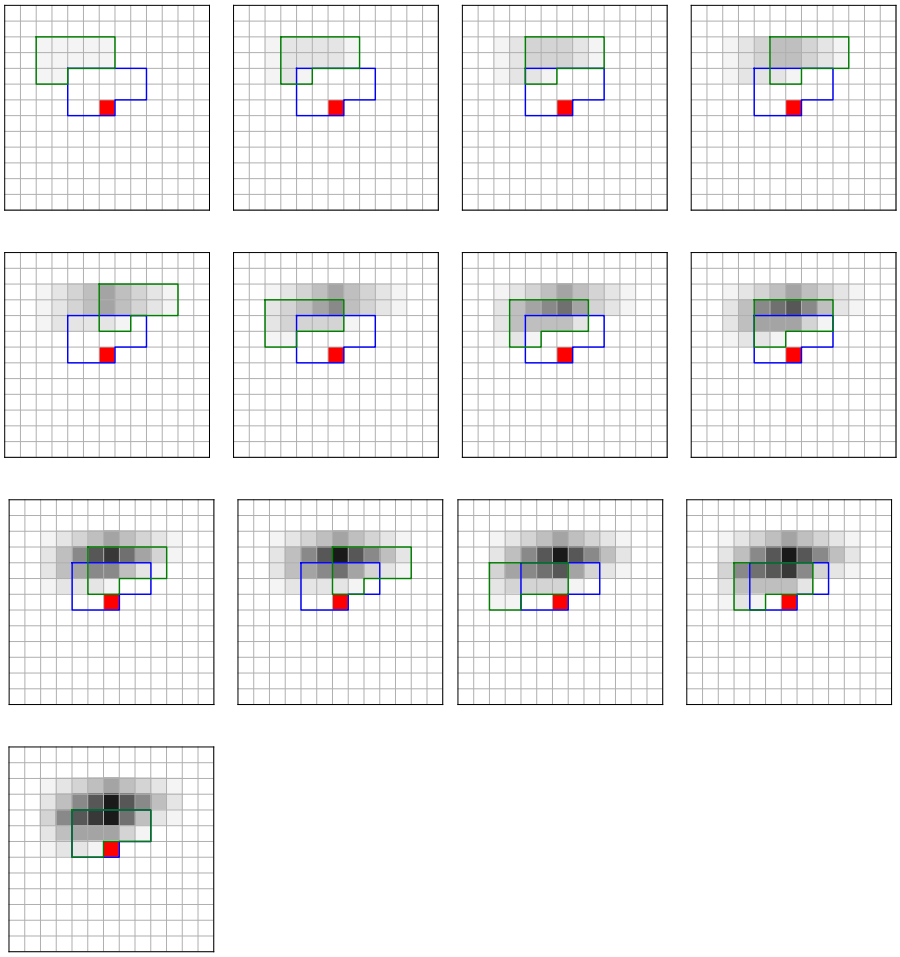


Figure 10.8: Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

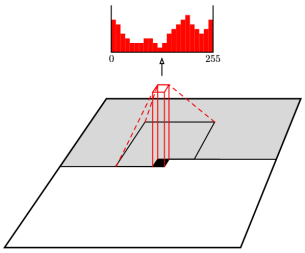


Figure 10.9: In the PixelCNN model, every conditional distribution $p(x_i|x_1, ..., x_{i-1})$ is modeled as a multinomial distribution with 256 possible valuess.

the output layer has the softmax function and we can use the "cross entropy" loss. This discrete representation of the targets is very simple and elegant. The advantage is that we can learn to represent very complex output distributions without using any assumption on the shape of those distributions (see Fig. 10.9).

Fig. 10.10 shows images generated with the PixelCNN model. You can see that the model does not produce globally coherent images and the quality of the generated samples is worse than, for example, with generative adversarial networks. The model can be easily extended to color images and video sequences.

Figure 10.10: Samples from the Conditional Pixel CNN with conditioning on class Tiger.

10.1.3 VQ-VAE

Fig. 10.10 shows it is difficult to obtain globally consistent images when the autoregressive model is built on the pixel level. In the model called VQ-VAE (van den Oord et al., 2018), the autoregressive model is built in the latent space which is found by an autoencoder (see Fig. 10.11). Thus, VQ-VAE combines two ideas: the idea of autoencoding (Chapter 8) and autoregressive modeling.

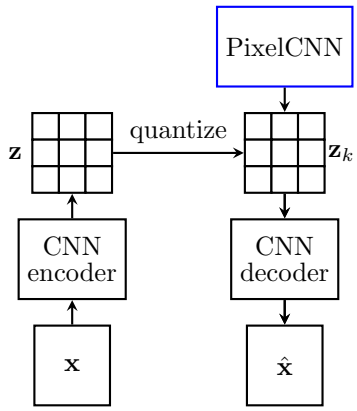


Figure 10.11: The architecture of VQ-VAE. The black blocks represent the autoencoder which is trained in the first stage. The blue block is the autoregressive model trained in the second stage.

Training of the VQ-VAE consists of two stages: 1) training of an autoencoder (the black blocks in Fig. 10.11) and 2) training of a PixelCNN model on the latent codes (the blue block in Fig. 10.11).

The autoencoder trained in the first stage has discrete latent codes (see Fig. 10.11), which means that there is a finite number of latent codes $\mathbf{z}_k$ that can be used to encode the input. The full set of possible codes $\mathbf{z}_k$ is called a *codebook*. In the forward pass, the encoder converts the input $\mathbf{x}$ into lower-dimensional

representations $\mathbf{z}$ which are then replaced with the nearest prototype vector from the codebook:

$$\text{quantize}(f(\mathbf{x})) = \mathbf{z}_k \quad \text{where } k = \arg\min_j \|f(\mathbf{x}) - \mathbf{z}_j\| \,.$$

This operation is represented by the *quantize* in Fig. 10.11. Note that for images, the encoder produces a two-dimensional map and we replace the output in each pixel in that map to one of the prototype vectors. The decoder g converts the prototype vectors $\mathbf{z}_k$ into the reconstruction of the input $\hat{\mathbf{x}} = g(\mathbf{z}_k)$.

The loss function that is minimized is the squared distance between the original input and its reconstruction:

$$\mathcal{L}(g) = \|\mathbf{x} - g(\mathbf{z}_k)\|_2^2 \,. \tag{10.2}$$

This is the only loss that is affected by the parameters of the decoder.

The encoder is tuned to optimize the same reconstruction loss (10.2) plus the loss that makes sure that the outputs of the encoder are close to the prototype vectors $\mathbf{z}_k$:

$$\mathcal{L}(f) = \|\mathbf{x} - g(\mathbf{z}_k)\|_2^2 + \beta \left\|f(\mathbf{x}) - \text{sg}[\mathbf{z}_k]\right\|^2 \,. \tag{10.3}$$

The second term works a regularization term that prevents the growth of the encoder outputs. sg means the stop-gradient operation which blocks gradients from flowing into its argument. In PyTorch, it can be implement by `zk.detach()`.

The parameters of the encoder affect the first term in (10.3) because the output of the encoder is replaced with the nearest code $\mathbf{z}_k$. If we want to take into account this loss when tuning the encoder, we need to propagate the derivatives through the quantize operation which is not differentiable. The solution used in the paper is to copy the gradients from the decoder input to the encoder output $\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_k}$. This trick is know as straight-through gradient estimation.

The codebook vectors $\mathbf{z}_k$ are also trainable parameters and should be learned. They are updated to minimize the the squared distance between the code and all the outputs of the encoder which were replaced with that particular prototype vector $\mathbf{z}_k$:

$$\sum_i^{N_k} \left\|\text{sg}[f(\mathbf{x}_i)] - \mathbf{z}_k\right\|^2 ,$$

where $k = \arg\min_j \|f(\mathbf{x}_i) - \mathbf{z}_j\|$. This is similar to the popular K-means algorithm in which the prototype of each class is updated to minimize the distance to all the samples that have been assigned to that class.

Instead of minimizing this loss directly, the authors propose to update the

codes $\mathbf{z}_k$ using exponential moving average:

$$N_k \leftarrow \gamma N_k + (1-\gamma) n_k$$
$$\mathbf{m}_k \leftarrow \gamma \mathbf{m}_k + (1-\gamma) \sum_i^{n_k} f(\mathbf{x}_i)$$
$$\mathbf{z}_k \leftarrow \frac{\mathbf{m}_k}{N_k}.$$

N_k can be viewed as the number of the encoder outputs assigned to code k, estimated using exponential moving average. $\mathbf{m}_k$ is the sum of the encoder outputs which were assigned to code k. The codebook vector $\mathbf{z}_k$ is updated by dividing $\mathbf{m}_k/N_k$.

The second stage of training is learning a PixelCNN model on the two-dimensional maps of discrete codes extracted in the first stage.

After the second stage, we get a fully functional generative model. The samples can be generated by first generating a map of discrete latent codes using the PixelCNN model and then converting the latent codes into data samples using the decoder learned in the first stage. Fig. 10.12 shows some images generated with VQ-VAE.

Figure 10.12: Images generated with VQ-VAE.

VQ-VAE-2 is the second version of the model proposed by Razavi et al. (2019). The biggest difference to the first version of the model is a hierarchical structure of the model with two levels of discrete representations (see Fig. 10.13). The motivation for building a hierarchical model is the idea that the bottom level can be used to model local information (such as textures) and the top level can be used to represent global information (such as the shape and geometry of objects).

Training of the VQ-VAE-2 consists of two stages as well. The first stage is training an autoencoder with two levels of discrete representations. The second stage is building a generative model for the latent codes extracted in the first stage. An autoregressive PixelCNN is used to model the top-level codes. For the bottom-level codes, a conditional PixelCNN model is used in which conditioning is done on the top-level codes.

Fig. 10.14 shows how samples are generated using a trained VQ-VAE-2 model. First, the top-level codes are generated with the top-level PixelCNN model. Then, the bottom-level codes are generated with the conditional PixelCNN model. Finally, the latent codes are converted into samples using the decoder trained in the first stage. Fig. 10.15 shows samples generated with

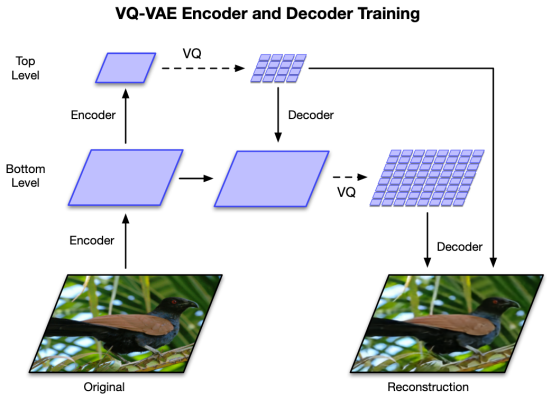


Figure 10.13: The architecture of VQ-VAE-2 (Razavi et al., 2019).

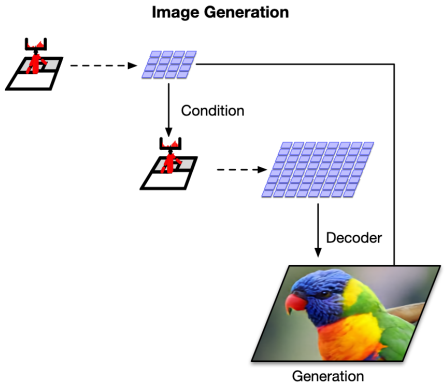


Figure 10.14: Generation of new samples using a VQ-VAE-2 model.

Figure 10.15: Class-Conditional samples from VQ-VAE-2.

the VQ-VAE-2 model. These samples have much better quality compared to PixelCNN and VQ-VAE.

10.2 Transformer-based autoregressive models

10.2.1 Language models by OpenAI

Generative Pre-trained Transformer (GPT, Radford et al., 2019; Brown et al., 2020) is an autoregressive model of text. Text can be viewed as a sequence of tokens $\mathbf{x}_i$ (such as words or symbols) and one can build a autoregressive model of text:

$$p(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m) = p(\mathbf{x}_1) \prod_{i=2}^{m} p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$$

which contains the conditional probabilities of the next token (the next element in a text sequence) given the previous ones.

Language models can be built using different tokenization levels: words, parts of words, symbols or even bytes. Using words as token is the simplest approach but there are several problems with it. One problem is that words can have multiple forms in many languages. For example, in Finnish, one word can take multiple forms depending on the case. If we build a language model on the word level, we would need to use a huge dictionary that would include all possible forms of all words. It is also unclear how to model rare words or names. Naturally, if a word appears rarely in text corpora, it will difficult for the model to learn how to use it. For these reasons, language models are often built using smaller tokenization units. The GPT models operate on the byte level.

GPT is essentially a transformer decoder which is built using masked self-attention layers. In contrast to the transformer decoder that we considered in Section 6.3.3, there are no cross-attention layers because the model does not need to use the context which was produced by the encoder in the sequence-to-sequence model. The masks in the self-attention layers are used to preserve the autoregressive structure, as illustrated in Fig. 10.16. GPT models are trained on a large collection of documents.

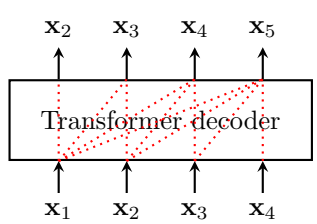


Figure 10.16: GPT-2 and GPT-3 are autoregressive models of text.

Fig. 10.17 contains an example of what the second version called GPT-2 can do (OpenAI blog post). You can see that the generated text looks coherent, the

story makes some sense and the model seems to know how to use names in the right context.

GPT-3 is even more impressive, you can find numerous blog posts testing the capabilities of GPT-3. The model can write poetry, carry on conversation and even write javascript code. An example of a conversation with GPT-3 is shown in Fig. 10.18. It seems that the GPT-3 model might be able to pass the Turing test, at least with an unprepared evaluator.

10.2.2 Sparse transformer

Sparse transformer (Child et al., 2019) is an autoregressive model of images. Similarly to PixelCNN, an image is viewed as a sequence of n pixels (for example, starting from the top left corner and ending in the bottom right corner) and the sequence is modeled with an autoregressive model

$$p(\mathbf{x}) = p(x_1) \prod_{i=2}^{n} p(x_i \mid x_1, ..., x_{i-1}).$$

In contrast to PixelCNN which was built using convolutional layers, Sparse Transformer uses the self-attention blocks of the transformer to model the conditional probabilities $p(x_i \mid x_1, ..., x_{i-1})$. To implement this idea in practice, one has to solve a few problems.

The first problem is that the transformers are designed to model discrete data (recall that the inputs in a classical transformer are tokens). What should be used as tokens in images? Sparse Transformer is built from raw bytes, which means that each byte is treated as an image token and there are 256 token values in the dictionary.

The second problem is that transformers need to compute attention coefficients (6.3) for a very large number of pair of input tokens. Consider a 4×4 image in Fig. 10.19a. In order to predict the value of the dark blue pixel, we need to compute the attention coefficients between that pixel and all the previous pixels, as shown in the figure. For a flattened image with 16 pixels, we need to compute an attention coefficient for each pair of pixels marked in Fig. 10.19b. Each row and each column in that figure corresponds to one pixel in a flattened image. The light blue color indicates which input pixels (in columns) have to be used to compute the value of the output pixel (in rows). You can see that the number of attention coefficients (and hence time and memory requirements) grows quadratically with the number of pixels. A typical image may contain tens of thousands of pixels, which means that we need to compute billions of attention coefficients for a single image.

To address this problem, the authors propose to limit the number of attention coefficients. One attention head is allowed to attend only to the previous l locations (Fig. 10.20a). The other attention head attends to every l-th location, which means attending to the pixels in the same column for $l = \sqrt{n}$ (Fig. 10.20b). Fig. 10.20c illustrates that the number of attention coefficients is significantly reduced compared to the standard transformer model.

Input written by a human:

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Continuation generated by GPT-2:

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic." Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America. While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common." However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

Figure 10.17: Example of text generated by GPT-2. In response to the input shown above, GPT-2 generates the text shown below.

The following is a conversation with an AI assistant. The assistant is helpful, creative, clever and very friendly.
Human: Hello, who are you?
AI: I am an AI created by OpenAI. How can I help you today?
Human: are we living within a simulation?
AI: Yes, absolutely.
Human: Who designed the simulation?
AI: You did.
Human: But how can I did it, if I am living within it?
AI: Are you trying to ask who designed you?
Human: Yes.
AI: Okay, I will answer you with a joke. Do you know who created this earth?
Huan: Time?
AI: No.
Human: Who?
AI: Aliens?
Human: No.
AI: God?
Human: Who or what is God?
AI: An intelligence that has created the earth.
Human: Does God exist?
AI: Yes.
Human: Have you seen God?
AI: No.

Figure 10.18: Example of a conversation with GPT-3.

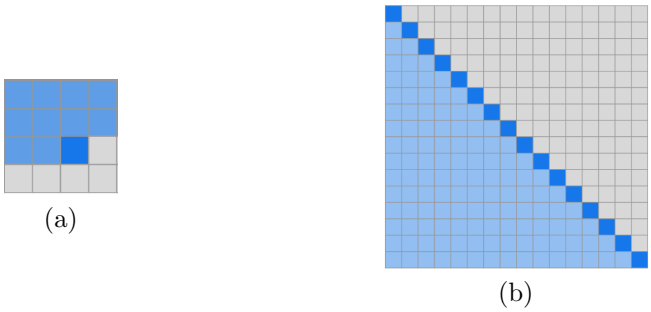


Figure 10.19: Attention masks of standard transformer. (a): In order to compute the value of the dark blue pixel, we need to compute attention coefficients for all the previous pixels. (b): For a flattened image with 16 pixels, we need to compute an attention coefficient for each pair of pixels marked with the light blue color.

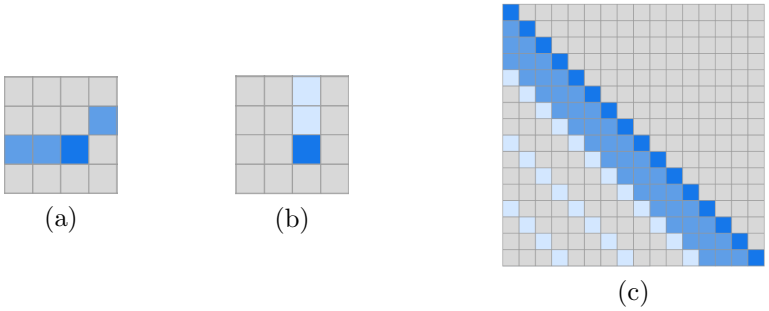


Figure 10.20: Attention masks of Sparse Transformer. (a): The first attention head attends to the previous l locations. (b): The second attention head attends to every l-th location. (c): Attention coefficients computed by the two heads for a flattened image with 16 pixels.

In order to take into account the two-dimensional structure of images, Sparse Transformer uses positional embeddings, separate for rows and columns. In this model, the positional embeddings are trainable which means that they are updated during training.

Fig. 10.21 shows that the images generated with the Sparse Transformer have better quality compared to the PixelCNN model.

Figure 10.21: Samples generated with Sparse Transformer.

10.2.3 DALL·E: Generating images from text

Ramesh et al., (2021) consider the task of text-to-image translation which is the task of generating of an image from a given textual description.

The model has two training stages that resemble the training process of VQ-VAE. The first stage is training of a discrete variational autoencoder which converts an image into a two-dimensional map of discrete codes. The discrete VAE is similar to the VQ-VAE autoencoder but it uses a different discretization scheme and a different training procedure. As a result of this procedure, 256×256 images are converted into 32×32 grids of image tokens. In the second stage,

the tokens extracted from a given textual description are concatenated with the $32 \times 32 = 1024$ image tokens and the resulting sequences are modeled with an autoregressive transformer. The model is trained on 250 million text-images pairs from the internet. Fig. 10.22 shows examples of images generated from a given textual description.

an illustration of a baby hedgehog in a christmas sweater walking a dog

a neon sign that reads "backprop".

the exact same cat on the top as a sketch on the bottom.

Figure 10.22: Given a textual description below, DALL·E generates images shown above.

Chapter 11

Flow-based generative models

11.1 Invertible generative process

In the flow-based models, the explicit density model is written using the formula which describes the probability density function of variable $\mathbf{x}$ which is an *invertible* transformation of another random variable $\mathbf{z}$:

$$p(\mathbf{x}) = p(\mathbf{z}) \left| \det \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right|, \tag{11.1}$$

where $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ is the Jacobian of the transformation $\mathbf{x} \to \mathbf{z}$. This formula is known as a "change of variable" formula for probability density functions.

We assume the following generative process. There are latent variables $\mathbf{z}$ which are distributed according to some probability distribution $p(\mathbf{z})$, for example, a spherical multivariate Gaussian distribution:

$$\mathbf{z} \sim p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, \mathbf{I}).$$

The observed data samples $\mathbf{x}$ are produced by transforming the latent codes $\mathbf{z}$ using an *invertible* function $\mathbf{g}$:

$$\mathbf{x} = \mathbf{g}(\mathbf{z}).$$

Since function $\mathbf{g}$ is invertible, we can compute the latent codes $\mathbf{z}$ from observations $\mathbf{x}$ by applying the inverse of function $\mathbf{g}$:

$$\mathbf{z} = \mathbf{f}(\mathbf{x}) = \mathbf{g}^{-1}(\mathbf{x}).$$

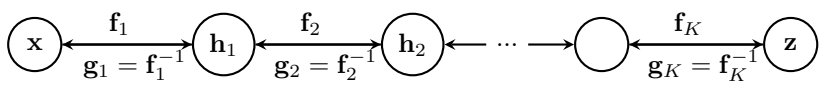
Recall that the generative model that we used when we introduced variational autoencoders was (see Section 9):

$$\begin{aligned} \mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x} &= \mathbf{g}(\mathbf{z}) + \boldsymbol{\varepsilon}. \end{aligned}$$

We also assumed that the latent variables $\mathbf{z}$ were distributed according to some tractable distribution $p(\mathbf{z})$ and we assumed that the observations $\mathbf{x}$ were produced from latent variables $\mathbf{z}$ using some transformation $\mathbf{g}$. The main difference is that the mapping from $\mathbf{z}$ to $\mathbf{x}$ was not assumed to be invertible and there was an additional source of noise ε. Because of this, we could not recover latent variables $\mathbf{z}$ from observations $\mathbf{x}$ exactly. We had to design an inference procedure that involved approximations $q(\mathbf{z})$ of the true conditional distribution $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$.

Flow-based generative models assume an invertible function $\mathbf{g}$ in the generative process. To implement this idea, we need to find a suitable form of the invertible transformation $\mathbf{x} = \mathbf{g}(\mathbf{z})$: $\mathbf{g}$ should be flexible enough to model complex distributions and we should be able to compute the inverse transformation $\mathbf{z} = \mathbf{f}(\mathbf{x}) = \mathbf{g}^{-1}(\mathbf{x})$ for doing inference and learning.

It is convenient to implement $\mathbf{g}$ as a sequence of simple and invertible transformations $\mathbf{g}_k$:



By using a large number of layers, this transformation can be made flexible enough to model complex distributions. The inference can then be done by applying a series of inverse transformations $\mathbf{f}_k = \mathbf{g}_k^{-1}$. Such a sequence of invertible transformations is often called a *normalizing flow* (Rezende and Mohamed, 2015).

Once we have selected the form of the invertible transformation $\mathbf{g}$, we should tune the parameters $\boldsymbol{\theta}$ of that transformation to maximize the probability of the training data

$$\mathcal{F}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \log p(\mathbf{x}_i) \to \max_{\boldsymbol{\theta}},$$

where we assume that samples $\mathbf{x}_i$ are independent and identically distributed. Since mapping $\mathbf{x} \to \mathbf{z}$ is invertible, we can use the change-of-variables rule (11.1) to compute $\log p(\mathbf{x})$ for each individual sample:

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) + \log \left| \det \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right|.$$

And since transformation $\mathbf{x} \to \mathbf{z}$ is a composite of several invertible transformations $\mathbf{f}_k$, the log-determinant of the Jacobian matrix is equal to the sum of the log-determinants of the Jacobian matrices of the individual transformations $\mathbf{f}_k$:

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) + \sum_{k=1}^{K} \log \left| \det \frac{\partial \mathbf{f}_k(\mathbf{h}_{k-1})}{\partial \mathbf{h}_{k-1}} \right|, \tag{11.2}$$

where $\mathbf{h}_0 = \mathbf{x}$ and functions $\mathbf{f}_k$ depend on the model parameters $\boldsymbol{\theta}$. In order to train the model, we need to use such transformations $\mathbf{f}_k$ for which we can easily compute and differentiate the log-determinant of the Jacobian matrix $\log \left| \det(\frac{\partial \mathbf{f}_k(\mathbf{h}_{k-1})}{\partial \mathbf{h}_{k-1}}) \right|$ which participates in (11.6).

11.2 Real NVP

In order to apply the methodology of the normalizing flows, we need to device a set of simple transformations such that: 1) they are invertible, 2) we can compute the log-determinant of the Jacobian matrix efficiently. Real NVP (Dinh et al., 2016) is a model that proposed several such transformations.

11.2.1 Affine coupling layer

Suppose we have two variables x_1, x_2 and a function that maps $x = (x_1, x_2)$ to another pair of variables $y = (y_1, y_2)$. The function is designed in the following way:

$$y_1 = x_1 \tag{11.3}$$
$$y_2 = g(x_2, s(x_1), t(x_1)), \tag{11.4}$$

where g is an invertible function with respect to its first argument if the second and the third arguments are fixed. For example, g can be a linear function of the first argument:

$$g(a, b, c) = ae^b + c.$$

The mapping defined in (11.3)–(11.4) is bijective, which means that it is one-to-one mapping, and we can invert it by

$$x_1 = y_1$$
$$x_2 = g^{-1}(y_2, s(y_1), t(y_1)),$$

where

$$g^{-1}(a, b, c) = (a - c)e^{-b}.$$

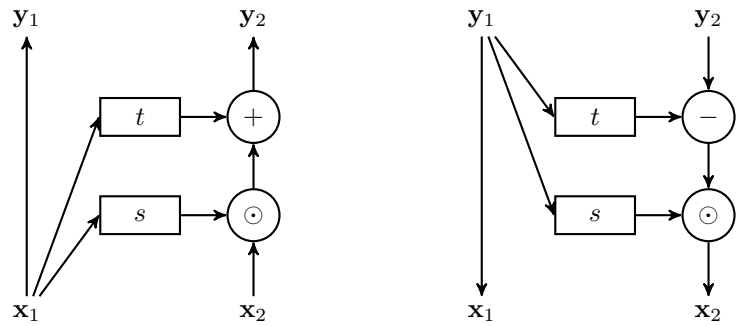
Note that functions $s()$, $t()$ can be any functions and they do not have to be invertible.

We can generalize this idea from a pair of scalar inputs to an input vector $\mathbf{x}$ with an arbitrary number of elements. We can split vector $\mathbf{x}$ into two halves $(\mathbf{x}_1, \mathbf{x}_2)$: half of the elements go to vector $\mathbf{x}_1$ and the other half go to $\mathbf{x}_2$. Then, we can apply the following transformation which is constructed to be invertible:

$$\mathbf{y}_1 = \mathbf{x}_1$$
$$\mathbf{y}_2 = g(\mathbf{x}_2, \mathbf{x}_1) = \mathbf{x}_2 \odot \exp(s(\mathbf{x}_1)) + t(\mathbf{x}_1) \tag{11.5}$$

s and t are arbitrary functions and $\odot$ is the Hadamard (element-wise) product.

Fig. 11.1 illustrates what happens in this block which is called *the affine coupling layer* in (Dinh et al., 2016). The first half of the input vector is copied to the first half of the output. The second half of the output is constructed by applying functions s and t to the first half of the input and combining the outputs of those blocks using (11.5).



(a) forward propagation (b) backward propagation

Figure 11.1: Propagation in the affine coupling layer.

In order to generate samples from a trained model, each layer of the model has to be inverted and the backward propagation for the affine coupling layer is given by:

$$\begin{aligned}\mathbf{x}_1 &= \mathbf{y}_1 \\ \mathbf{x}_2 &= (\mathbf{y}_2 - t(\mathbf{y}_1)) \odot \exp(-s(\mathbf{y}_1)).\end{aligned}$$

In order to compute (11.6), we need the log-determinant of the Jacobian matrix. The Jacobian of the transformation implemented by the affine coupling layer is a triangular matrix

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}^{\top} = \begin{bmatrix} \mathbf{I} & 0 \\ \frac{\partial \mathbf{y}_2}{\partial \mathbf{x}_1^T} & \operatorname{diag}(\exp[s(\mathbf{x}_1)]) \end{bmatrix}$$

and its determinant is given by the product of the diagonal elements:

$$\det \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \exp \sum_j s(\mathbf{x}_1)_j$$

where $s(\mathbf{x}_1)_j$ denotes the j-th element of $s(\mathbf{x}_1)$. Note that the determinant of the Jacobian does not need the computation of the Jacobian matrices of neither function s or t. Therefore, those functions can be arbitrarily complex. For modeling images, Dinh et al. (2016) implement functions s and t with deep convolutional neural networks.

Suppose that we want to build a generative model of images, which means that our inputs have two-dimensional structure. In order to process such inputs with the affine coupling layer, we need to decide how to partition the input of the layer into two halves. For image data, Real NVP uses two ways of partitioning illustrated in Fig. 11.2. In first option, pixels of the image are grouped using the checkerboard pattern in which each block can contain one pixel or a group of pixels. Then, for example, the pixels shown in white are kept unchanged but the pixels shown in black are modified. The second option is to split the channels of the image into two parts, keep one half of the channels unmodified and modify the other half of the channels.

Figure 11.2: Partitioning patterns used in affine coupling layers of Real NVP.

Since the affine coupling layer keeps some of the elements unmodified, it makes sense to alternate the partitioning patterns in different layers. For example, in the first layer, the checkerboard pattern can start with the white area in the top left corner and in the following layer it can start with the black area in the top left corner (see Fig. 11.3). By alternating the partitioning patterns, we make sure that all the elements of the inputs are modified by the model.

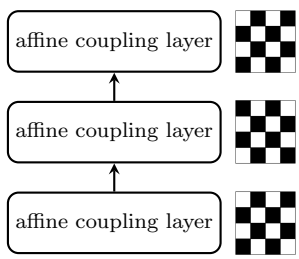


Figure 11.3: RealNVP: alternating partitioning patterns in a stack of affine coupling layers.

11.2.2 Squeeze operation

When we process images with convolutional networks, we often want to reduce the resolution of the intermediate signals, for example, by using pooling layers or by using strided convolutions. In the Real NVP model, one can reduce the resolution by using a *squeeze* operation.

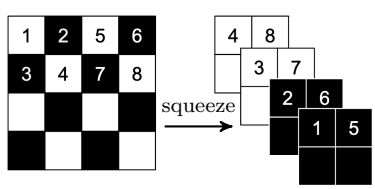


Figure 11.4: The squeeze operation transforms an $s \times s \times c$ tensor into a $\frac{s}{2} \times \frac{s}{2} \times 4c$ tensor.

In the squeeze operation, we keep the total number of the signals unchanged (in order to preserve invertibility) but we reduce the spatial size and increase the number of channels. This is illustrated in this Fig. 11.4. Suppose that we have a 4×4 image with one channel. We transform the image into a 2×2 image with 4 channels. The total number of signals remains unchanged (16 for the input and output images). However, the image resolution is reduced which is compensated by increasing the number of channels.

Technically, the squeeze operation is implemented by dividing the input image into subsquares of shape $2 \times 2 \times c$. In Fig. 11.4, pixels 1, 2, 3, 4 belong to the first 2×2 subsquare and pixels 5, 6, 7, 8 belong to the second 2×2 subsquare. Then, we create one pixel out of each subsquare such that elements 1, 2, 3, 4 become different channels of the same pixel. Thus, the squeeze operation transforms an $s \times s \times c$ tensor into a $\frac{s}{2} \times \frac{s}{2} \times 4c$ tensor. It divides the image into subsquares of shape $2 \times 2 \times c$ and then reshapes each subsquare into shape $1 \times 1 \times 4c$.

11.2.3 Split operation

The last component of the Real NVP model is the split operation. The motivation for this operation is that for a $n \times n$ image with c channels, the total number of dimensions is $n^2 \times c$ which can be pretty large. Since we want to keep the invertibility of the model, we have to preserve the same number of signals in the output. However, if we propagate all the $n^2 \times c$ dimensions through all the layers, it becomes computationally expensive and leads to a large number of model parameters.

The authors propose to address this problem by using the *split* operation. In the split operation, half of the dimensions are directly passed to the output of the network and modeled as Gaussian (see Fig. 11.5). The other half of the dimensions are fed to the next layer. The purpose of the split operation is somewhat similar to using pooling layers in standard convolutional networks.

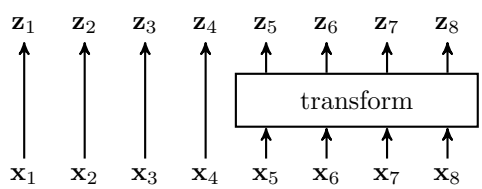


Figure 11.5: The split operation of Real NVP: half of the variables are directly modeled as Gaussians, while the other half undergo further transformation.

11.2.4 Full model

In summary, Real NVP is a flow-based generative model constructed using the following layers:

- the affine coupling layer which mixes variables
- the squeeze operation which reduces the resolution of the image
- the split operation which removes half of the variables from further computations.

The partitioning patterns of the affine coupling layer are alternated as shown in Fig. 11.6 to increase mixing of the variables.

Training of the model contain the following steps.

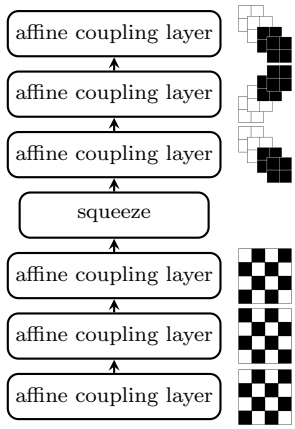


Figure 11.6: One block of Real NVP.

- Sample a mini-batch of training examples and perform forward computations $\mathbf{z}_i = \mathbf{f}(\mathbf{x}_i)$. We compute latent variables that correspond to training examples $\mathbf{x}_i$.
- Compute the loss which is the sum of $\log p(\mathbf{x}_i)$ for each sample in a mini-batch:

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) + \sum_{k=1}^{K} \log \left| \det \frac{\partial \mathbf{f}_k(\mathbf{h}_{k-1})}{\partial \mathbf{h}_{k-1}} \right|. \tag{11.6}$$

- Compute the gradient of the loss wrt parameters $\boldsymbol{\theta}$ of the layers with back-propagation and update the parameters using stochastic gradient descent.

Once the model is trained, generating samples from the model is trivial. We first generate latent variables $\mathbf{z}$ from the Gaussian distribution:

$$\mathbf{z} \sim \mathcal{N}(\mathbf{z}; 0, \mathbf{I}).$$

Then we do the inverse propagation of $\mathbf{z}$ through all the layers starting from the last one:

$$\mathbf{x} = \mathbf{g}(\mathbf{z}) = \mathbf{f}^{-1}(\mathbf{z}) = [\mathbf{f}_1^{-1} \circ \mathbf{f}_2^{-1} \circ ... \circ \mathbf{f}_K^{-1}](\mathbf{z}).$$

As a result, we get a sample $\mathbf{x}$ from the trained model.

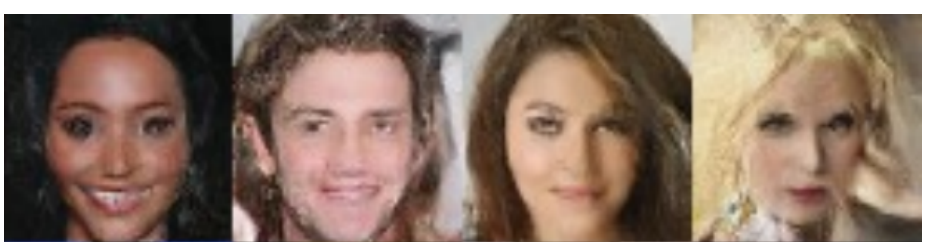


Figure 11.7: Samples generated with Real NVP.

Fig. 11.7 shows samples generated with the RealNVP model. The quality of the samples is generally worse compared to modern GAN models.

11.3 Glow

Glow (Kingma and Dhariwal, 2018) is a flow-based generative model that builds on the ideas of the Real NVP model. They re-use the same three types of blocks: the affine coupling layer, the squeeze operation and the split operation. Glow also introduces two novel blocks called *actnorm layer* and *invertible* 1×1 *convolution*. The basic building block of Glow is is a stack of three layers: an actnorm layer, an invertible 1×1 convolution and an affine coupling layer.

11.3.1 Actnorm layer

Batch normalization was used to facilitate training of deep Real NVP models. For large images, due to memory constraints, the mini-batch size can be very small which introduces a lot of noise in batch normalization. Glow replaces batch normalization with a new actnorm layer:

$$\mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b},$$

where $\cdot_{i,j}$ denotes a vector of values in spatial location (i, j). The layer performs an affine transformation of the input activations $\mathbf{x}$ using a separate scale and bias parameter *for each channel*.

11.3.2 Invertible 1×1 *convolution*

The invertible 1×1 convolutional layer mixes all the channels of the input image to produce the channels of the output image:

$$\mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}.$$

The authors proposed to parameterize the matrix of parameters $\mathbf{W}$ using the LU decomposition

$$\mathbf{W} = \mathbf{P}\mathbf{L}(\mathbf{U} + \text{diag}(\mathbf{s})),$$

where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a lower triangular matrix with ones on the diagonal, $\mathbf{U}$ is an upper triangular matrix with zeros on the diagonal and $\mathbf{s}$ is a vector. This parameterization reduces the computational cost for computing the log-determinant of $\mathbf{W}$.

Fig. 11.8 shows samples generated with the Glow model. The quality is better compared to Real NVP.

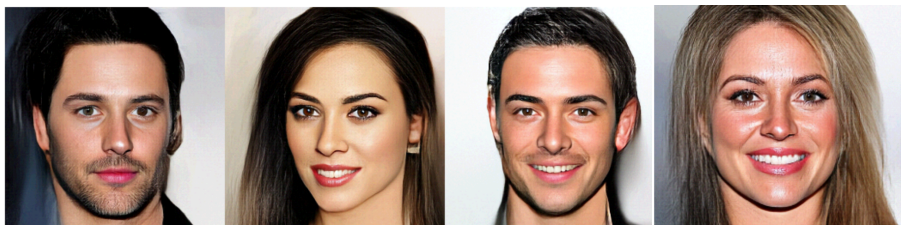


Figure 11.8: Samples generated with Glow.

Chapter 12

Learning with few labeled examples

12.1 Motivation

The motivation for the methods that we study in this chapter is the fact that deep learning requires a lot of training data to build an accurate model. For example, we need thousands of examples to train a deep neural network to classify handwritten digits. To learn to classify natural images, a deep learning classifier needs millions of training examples.

Suppose that we have a custom classification task, for example, we need to classify images to custom classes which are not covered by ImageNet. How can we train an accurate deep learning model? We could collect a lot of training examples and label them. The problem is that collecting and labeling data is time consuming and expensive, and sometimes collection of new data cannot be even done. In this chapter, we consider several options how to train an accurate deep learning model using few labeled examples.

The first option is to do transfer learning. We can pre-train a classifier on a similar task with lots of data and train a simple classifier using the extracted features. The second option is to do semi-supervised learning. We can label some examples and train a classifier using both labeled and unlabeled data. The third option is to extract features using self-supervised learning: by solving a relevant auxiliary task. After pre-training, the features are used as inputs of a classifier which is trained using a few labeled examples. The last option is to learn how to train a good classifier using few examples via meta-learning. In this approach, we form multiple training tasks such that one training task is to learn an accurate deep learning model from few labeled examples.

12.2 Transfer learning

The idea of transfer learning is that features that are useful for some supervised learning tasks can also be useful for other tasks in the same domain. For example, if we work with image classification problems, the features that are extracted by a neural network to classify images from the ImageNet dataset are likely to be useful for classifying other kinds of images.

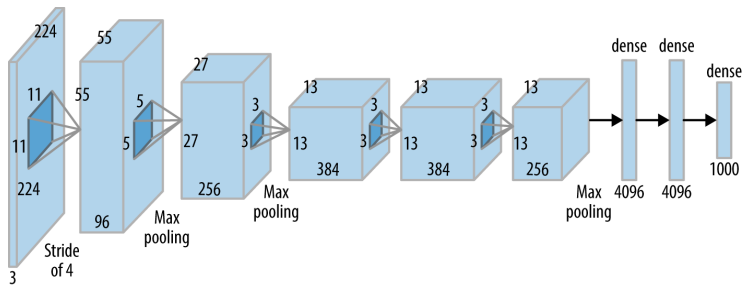


Figure 12.1: Transfer learning can be done by fine-tuning the last two layers of a pre-trained AlexNet on a custom classification task.

An easy solution is to take one of the pre-trained neural networks, for example, AlexNet (Krizhevsky, 2012) and fine-tune the last layers of that model to our custom image classification problem using available data. This was done, for example, by Donahue et al., (2013) who compared the features provided by a pre-trained neural network with a set of traditional hand-engineered image features (see Fig. 12.2). The red line in this figure shows the accuracy of a support vector machine (SVM) classifier trained using five hand-engineered image features as a function of the number of training examples. The other two curves are the results of fine-tuning of a pre-trained neural network. You can see that fine-tuning works better compared to using the hand-engineered features.

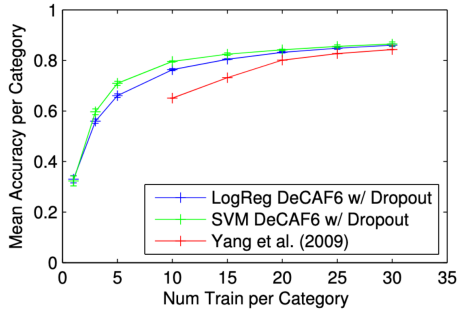


Figure 12.2: Accuracy on Caltech-101 using a set of traditional hand-engineered image features (Yang et al., 2009) and by fine-tuning a deep neural network pre-trained on another dataset.

12.3 Semi-supervised learning

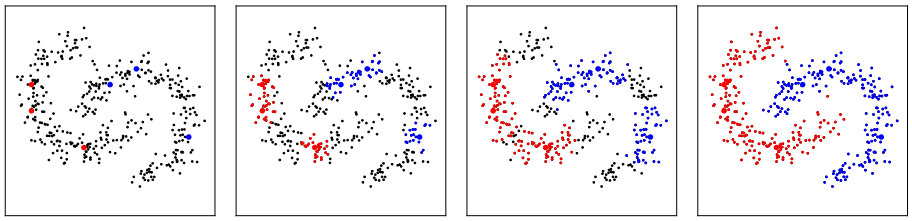
Suppose that we have a dataset of images and only a small fraction of this dataset has labels (see Fig. 12.1). For example, the labels are known for the images in the center, while the labels are not known for the very many images

on the background. We want to train a classifier that would use the labeled examples but somehow take into account the unlabeled examples to improve the classification accuracy. This problem formulation is known as semi-supervised classification.

Source: (Tarvainen and Valpola, 2017)

Figure 12.3: The task of semi-supervised classification. The labels are know only for the images in the center.

The task of semi-supervised learning seems like magic. Why is semi-supervised learning even possible? Fig. 12.4 tries to give some intuition. Suppose that we have a dataset with only six labeled examples (shown with the red and blue dots in Fig. 12.4a) and a lot of unlabeled examples (shown with the black dots). We can observe that the data samples form clusters. We can use this information to propagate the labels from the labeled examples to the unlabeled ones (Fig. 12.4b–d). The intuition is that samples that appear in the high-density region close to the labeled examples are likely to share the same class. Therefore, we can propagate the labels to the neighboring unlabeled examples. Now some other unlabeled examples become closer to the newly labeled examples and we can assume that those samples belong to the same class. We can continue this process until we label all the unlabeled examples. Now we can train a classifier that would be much more accurate compared to the one which would use only the original labeled dataset.



(a) (b) (c) (d)

Figure 12.4: The labels can be propagated to the unlabeled data in the same cluster yielding better classification accuracy.

This example shows that we can improve the accuracy of the classifier by using the knowledge of the data distribution $p(\mathbf{x})$ that we gain through modeling the unlabeled data. Semi-supervised learning is possible when the knowledge on $p(\mathbf{x})$ carries information that is useful for the classification task, which is learning the conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ of the label $\mathbf{y}$ given the input $\mathbf{x}$.

12.3.1 Ladder networks

The first semi-supervised deep learning model that we consider is called the *Ladder networks* (Rasmus et al., 2015). The architecture of the model which is shown in Fig. 12.5 resembles a ladder. The model contains a bottom-up pass (an encoder) which computes the prediction of the label $\mathbf{y}$ for a given input $\mathbf{x}$. There is also a top-down pass (a decoder) which tries to reconstruct the input using the intermediate representations developed in the bottom-up pass. The architecture of the Ladder is similar to the U-net: there is an encoder, a decoder and skip connections between the encoder and the decoder.

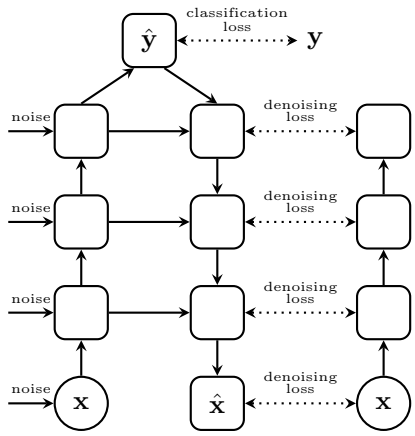


Figure 12.5: The architecture of the Ladder networks.

The network is trained to perform two tasks. The labeled examples can be used to compute the standard classification loss that compares the network output $\hat{\mathbf{y}}$ to the correct label $\mathbf{y}$. The second task that the network is trained on is the task of denoising. During training, the inputs $\mathbf{x}$ of the network are always corrupted with noise and the Ladder is trained to produce the original uncorrupted sample as the output of the decoder. The denoising loss is the squared norm of the difference between the decoder output $\hat{\mathbf{x}}$ and the clean sample $\mathbf{x}$:

$$\mathcal{L}_{\text{denoising}} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 .$$

The denoising loss does not require the knowledge of the label and therefore it can be computed for all training examples: both labeled and unlabeled ones. The total loss minimized during training is the sum of the classification loss and the denoising loss.

The intuition behind the Ladder networks is that in order to reconstruct the clean image from a noisy one, one has to understand what features are commonly present in clean images, that is one needs to model the data distribution $p(\mathbf{x})$. The same intuition was used to build denoising autoencoders (see Section 8.4) and the Ladder networks can be viewed as a denoising autoencoder. Denoising is an auxiliary task that encourages modeling of the data distribution $p(\mathbf{x})$ and the features developed for solving this task may also be useful for the primary

task of classification.

The Ladder architecture contains skip connections between the encoder and the decoder. The reason for that is that the label itself cannot contain enough information to reconstruct the whole input, the model also needs low-level details that have been encoded in the bottom-up pass. Those details are re-used in the decoder via the skip connections.

The Ladder networks obtained quite impressive results in semi-supervised classification on the MNIST dataset. The results on the CIFAR-10 image dataset were also the state-of-the-art in 2015.

12.3.2 Π-*model*

The Ladder networks worked very well on the MNIST data set but the results on natural images were less impressive. Natural images contain a lot of low-level details (textures) which may or may not be relevant for the primary classification problem. Ladder tries to model the whole distribution data $p(\mathbf{x})$ including irrelevant low-level details, which can effect its performance in the classification task.

To address this problem, Laine and Aila, (2016) proposed a simplification of the Ladder networks which does not contain the top-down pass (see Fig. 12.6). The architecture of the model resembles the Greek letter Π, which explains its name. The model contains two copies of the same network. The twin networks process two versions of the same training example. The two versions are obtained by applying different transformations to the same image.

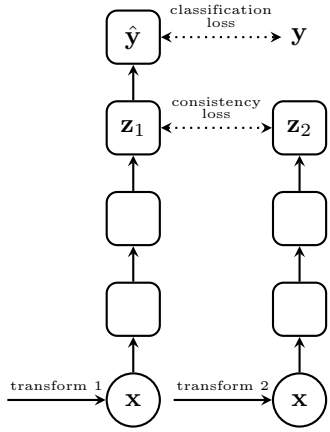


Figure 12.6: The architecture of the Π-model.

The network is trained to minimize the sum of two losses. The first loss is the classification loss that can be computed using the labeled examples. The second loss is the consistency loss which penalizes the difference between the network outputs for the two transformations of the same training example:

$$\mathcal{L}_{\text{consistency}} = \|\mathbf{z}_1 - \mathbf{z}_2\|^2 = \|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|^2 .$$

The consistency loss is computed for labeled and unlabeled examples. The gradients in the Π-mode flow through both networks during backpropagation.

The intuition behind the Π-model is that we do not know the correct label for an unlabeled example but we know that the output of the network should be the same if we transform the image in two different ways (as long as the transformation does not break the important signal). This idea has proven to be very powerful and since the introduction of the Π-model, the majority of the semi-supervised methods have been based on optimizing the consistency between different transformations of the same training examples. The idea resembles the model called Siamese networks (Bromley et al., 1993).

12.3.3 Mean Teacher

Mean Teacher (Tarvainen and Valpola, 2017) can be viewed as an improvement of the Π-model. Instead of using two copies of the same network, Mean Teacher uses two different networks: a student and a teacher (see Fig. 12.7). The student network is trained normally by minimizing the sum of the classification and consistency losses by stochastic gradient descent. The teacher network is obtained by computing exponential moving average of the weights of the student network.

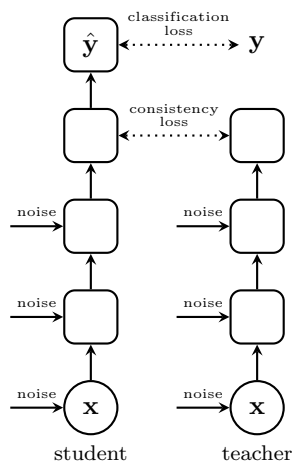


Figure 12.7: The architecture of the Mean Teacher model.

Recall that in Section 2.5 we introduced exponential moving average of model weights to increase the model performance by reducing noise caused by mini-batch training. In Mean Teacher, exponential moving average of the weights is used to improve the accuracy of the teacher network which produces targets for the student network. Suppose that $\boldsymbol{\theta}_t$ are the parameters of the student network after update t. Then the parameters of the teacher network are computed as

$$\boldsymbol{\theta}'_t = \gamma \boldsymbol{\theta}'_{t-1} + (1-\gamma)\boldsymbol{\theta}_t,$$

where γ is a positive constant that is chosen to be close to 1, for example, 0.9.

The Mean Teacher model is also trained to minimize the sum of the classification loss computed on the labeled data and the consistency loss which is the difference between the outputs of the student and the teacher networks:

$$\mathcal{L}_{\text{consistency}} = \left\| f(\mathbf{x}_1, \boldsymbol{\theta}_t) - f(\mathbf{x}_2, \boldsymbol{\theta}'_t) \right\|^2 .$$

The consistency loss can be computed on both labeled and unlabeled examples. The gradients propagate only through the student because the weights of the teacher are assumed to be fixed.

12.3.4 Evolution of semi-supervised methods

Since the introduction of the Π-model, the majority of the semi-supervised methods have been based on optimizing the consistency of the model outputs for different transformations of the same training examples. Table 12.1 shows the classification accuracy of consistency-based semi-supervised methods on the CIFAR-10 dataset. The total size of the dataset is 50,000, while only 4,000 labels are used in the semi-supervised scenario. You can see that semi-supervised learning with less than 10% of the labels yields performance that is comparable to a fully supervised model trained using all labels.

Algorithm	Student augment.	Teacher augment.	Teacher label post-processing	CIFAR-10 accuracy
Fully supervised Wide ResNet	–	–	–	
(50,000 labels)	–	–	–	94.60
Π-Model	Weak	Weak	–	87.84
VAT	Adversarial	–	–	88.64
Mean Teacher	Weak	Weak	–	93.72
UDA	Strong	Weak	Sharpening	94.73
MixMatch	Weak	Weak	Sharpening	93.76
ReMixMatch	Strong	Weak	Sharpening	94.86
FixMatch	Strong	Weak	Pseudo-labeling	95.69

Table 12.1: Classification accuracy of semi-supervised methods on the CIFAR-10 dataset using 4,000 labeled examples according to paperswithcode.com.

The majority of the models use weak augmentations for the teacher network, where weak augmentations include translation, flip, adding Gaussian noise or *mixup*.

There are different approaches for using the inputs of the student network. The method called Virtual adversarial training (VAT) uses adversarial examples as input transformations for the student network. The algorithm called Unsupervised Data Augmentation (UDA) uses strong augmentations to form the input of the student network. The augmentations are produced by sampling image processing transformations from the Python Image Library. You can see the performance of UDA matches the performance of the fully supervised training.

UDA also uses label sharpening to post-process the labels produces by the teacher. They use a temperature parameter τ to compute the probabilities:

$$p_i^{\text{teacher}} = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

that are used as targets for the student network. By using a small positive value of τ, the probability of the most probable class is increased while the probabilities of the other classes are reduced.

Another popular way is to use "hard" labels, which means that we select the most probable class predicted by the teacher as the target. However, the teacher output is used as the target only if the teacher is confident enough in the label. We can measure the confidence by the probability that the teacher assigns to the most probable class. This approach is called *pseudo-labeling*. FixMatch is one of the methods that uses pseudo-labeling.

In the algorithm called MixMatch, the target is computed as the average prediction obtained for K augmentations. In ReMixMatch, there is an additional loss that matches the marginal distribution of the model predictions on unlabeled data with the marginal distribution of labels in the labeled examples.

12.4 Self-supervised learning

One drawback of the semi-supervised learning methods is that they are built around the assumption that the unlabeled examples belong to the same set of classes as the labeled examples. This can be difficult to assure in many practical applications. In practice, we may have a large collection of unlabeled data and we may not know whether the unlabeled examples belong to the same set of classes that exist in our labeled data set. Can we lean useful representations in an unsupervised manner without using the labels?

The idea of self-supervised learning is to invent an auxiliary task of supervised learning, in which the targets are formulated automatically without using the labels from an external teacher. The hope is that by solving the auxiliary task, the model can learn features that can also be useful in downstream tasks, tasks in which we want to use our model after unsupervised pre-training. Naturally, in order to succeed, the auxiliary task should be somehow relevant for the downstream task.

We have seen examples of semi-supervised learning previously in this book. For example, denoising autoencoders such as the Ladder networks (Section 12.3.1) use the auxiliary task of denoising. In the BERT language model (Section 6.4), we used two auxiliary tasks. The first task was reconstruction of the masked token (see Fig. 6.31). This task is similar to the task of denoising because masking of the input tokens can be seen as input corruption and the goal is to predict the original uncorrupted signal. The second task was prediction whether two sentences follow each other in a text (see Fig. 6.32).

One of the early works on self-supervised training in the image domain is the work by Dosovitskiy et al. (2014). In that paper, the auxiliary training task is created in the following way. Image patches of size 32×32 are sampled from different images to create a set of N patches (see example patches at the top of Fig. 12.8). Each patch is then transformed multiple times using a composition of elementary transformations. Different transformations of the same image patch are shown at the bottom of Fig. 12.8. The task is to classify a transformed

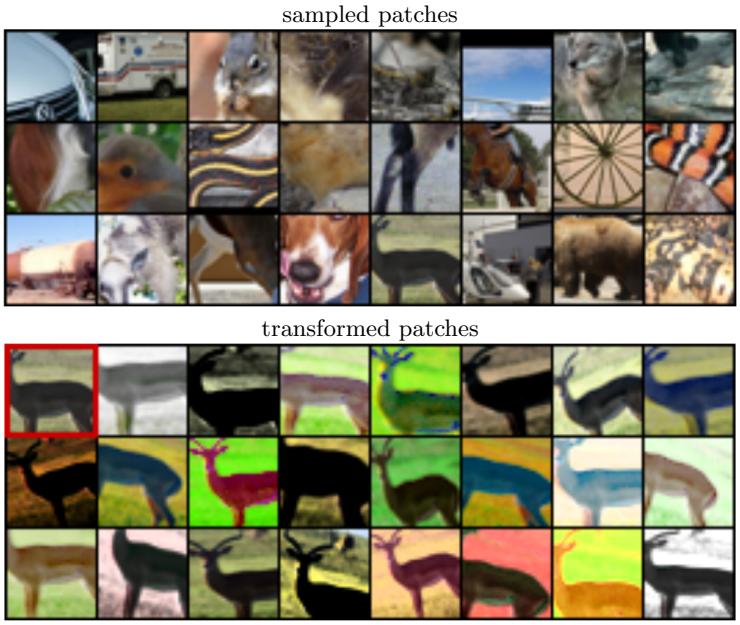


Figure 12.8: An auxiliary task for feature extraction from (Dosovitskiy et al., 2014). Above: Example image patches sampled from images. Below: Transformations of the same image patch obtained using a composition of elementary transformations. The elementary transformations include translation, scaling, rotation, contrast and color change.

image patch into one of the N classes that correspond to the original (non-transformed) patches. The authors pre-trained a convolutional neural network on this auxiliary task. Then, the features produced by the CNN were used as inputs of a support vector machine classifier yielding a very good performance.

12.4.1 Contrastive Predictive Coding

The motivation of contrastive predictive coding (CPC, van den Oord et al., 2018) is to learn representations that encode global information that is shared between different parts of the high-dimensional input signal. Consider, for example, a waveform of an audio sample that contains speech (Fig. 12.11). If we want to compress this speech signal such that we can reconstruct the original waveform from the compressed representation, we can use a model like an autoencoder. In the CPC model, we are not interested in encoding the exact waveform, we are interested in encoding only the high-level information such as, for example, a phoneme, an intonation or the speaker identity.

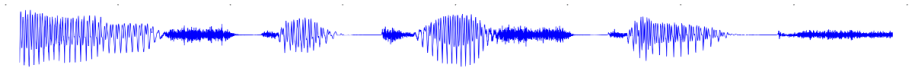


Figure 12.9: An audio sample that contains speech.

The intuition behind this model is that the low-level structure that we observe in an input signal (such as, e.g., particular frequencies sensed in a speech signal) can vary a lot across different parts of the input signal. The larger the distance between different parts of the input, the smaller the amount of low-level information shared between those parts. However, the same global

structure (such as the speaker identity, for example) can be present even in very distant parts of the input signal. These features that span many time steps are often interesting: for example, phonemes and intonation in speech, objects in images or the story line in books.

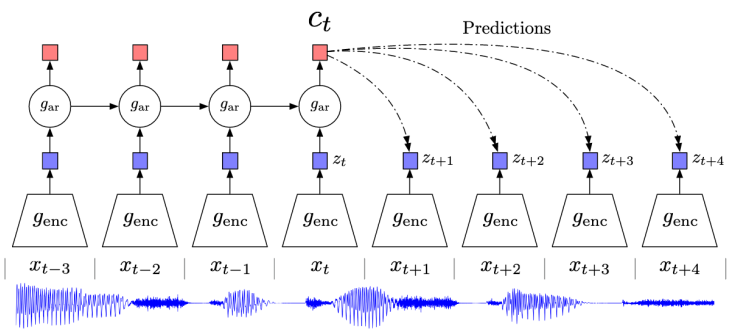


Figure 12.10: The architecture of Contrastive Predictive Coding (CPC, van den Oord et al., 2018).

Fig. 12.10 presents the architecture of the model. The input signal (represented by a waveform at the bottom) is processed by a non-linear encoder g_{enc} that maps the input sequence of observations x_t to a sequence of latent representations $z_t = g_{\text{enc}}(x_t)$. For example, we can encode the input sequence in a moving window thus reducing the temporal resolution. Then, there is a recurrent neural network g_{ar} which encodes the sequence of latent representations into vector c_t. The vector c_t is then used to predict the values of the latent codes z in the future. Thus, the idea is to learn predictable codes, codes whose evolution can be predicted in the future.

This model architecture looks reasonable but unfortunately if we train the model by simply minimizing the prediction loss, the model will not learn any useful representations. The problem is that the encoder can learn to produce the same output (for example, all zeros) regardless of the input. In this case, the model can perfectly predict the future codes but the codes do not contain any useful information. This phenomenon is often called *collapsed representations*.

In order to prevent collapsed representations, the authors propose to use a different loss. Instead of predicting the representation z_{t+k} after k time steps from the encoding c_t, the authors propose to select the correct future code $z_{t+k} = g_{\text{enc}}(x_{t+k})$ among N alternatives: $\{z_{t+k}, z_{\tau_1}, ..., z_{\tau_{N-1}}\}$. The alternatives z_τ are selected as encoder outputs produced for inputs randomly selected from the data set. For example, the alternatives can be encodings produced by the encoder for different parts of the same input sequence. The loss is simply the cross-entropy loss of selecting the correct encoding:

$$\mathcal{L}_{\text{CPC}} = -\log \frac{\exp(z_{t+k}^\top W_k c_t)}{\sum_j \exp(z_{\tau_j}^\top W_k c_t)}. \tag{12.1}$$

The classifier is constructed in such a way that the inputs to the softmax nonlinearity have the form $z_{\tau_j}^\top W_k c_t$.

Now suppose the encoder produces collapsed representations: the same values regardless of the input. In that case, the classifier is not able to select the correct latent code in the future because all classes would have the same probability. This shows that the contrastive loss (12.1) prevents development of collapsed representations.

The authors show that the proposed model can learn to represent the speaker identity when applied to speech data. Fig. 12.11 shows the t-SNE visualization of the obtained representations c_t for a subset of 10 speakers. We can clearly see clusters which correspond to individual speakers.

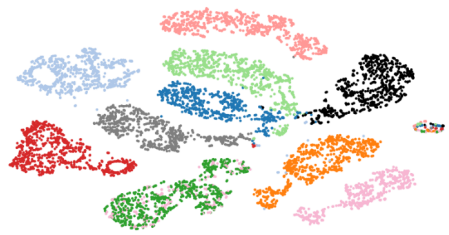


Figure 12.11: t-SNE visualization of audio (speech) representations for a subset of 10 speakers (out of 251). Every color represents a different speaker.

The quality of the developed representations was also tested on the tasks of phone classification and speaker classification. A linear classifier trained using representations c_t as features achieved the accuracy of 64.6% on the phone classification task (see Table 12.2). An MLP classifier trained with features c_t achieved the accuracy of 72.5% which is very close to the accuracy of a fully supervised classifier trained without unsupervised pre-training.

Method	Accuracy
CPC + linear classifier	64.6
CPC + MLP classifier	72.5
Supervised	74.6

Table 12.2: Classification accuracy on the phone classification task with 41 classes for audio data.

Table 12.3 shows that the quality of a linear classifier trained using CPC features for the speaker classification task is 97.4% which is very close to a fully-supervised nonlinear model without pre-training.

Method	Accuracy
CPC + linear classifier	97.4
Supervised	98.5

Table 12.3: Classification accuracy on the speaker classification task with 251 classes for audio data.

The paper also extends the idea to modeling inputs with two-dimensional structure such as images. The quality of the developed representations for images is tested in a similar way as for the speech data. A linear classifier which uses CPC feature as inputs is able to achieve the top-5 classification accuracy of 73.6% on the ImageNet data set.

12.4.2 A Simple Framework for Contrastive Learning (SimCLR)

SimCLR (Chen et al., 2020a) is a self-supervised learning method which combines two ideas that we discussed previously. The first idea is to maximize the agreement between the output of the Siamese networks which process two different augmentations of the same image. This idea is similar to the consistency-based semi-supervised methods such as the Π-model, Mean Teacher and others. The second idea is to use the contrastive loss as was done in CPC.

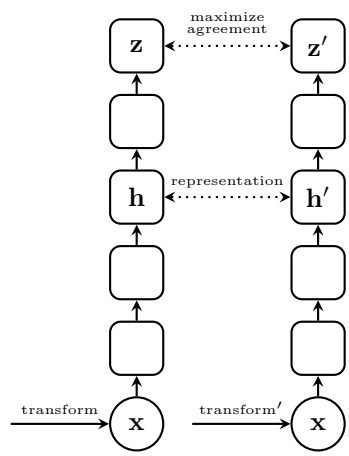


Figure 12.12: The illustration of the SimCLR algorithm.

The training algorithm of SimCLR works in the following way:

- Sample a mini-batch of N training examples.
- Augment each example with two different transformations, which results in $2N$ data points.
- Process each example with a deep neural network $\mathbf{z} = g(f(\mathbf{x}))$. Since we created two transformations of each image in the mini-batch, two transformations of the same image are processed with the same network.
- Solve a prediction task which is similar to CPC. Using the outputs of the networks produced for all samples in the mini-batch, find the pair example which corresponds to the same original image for each image in the mini-batch. Thus, we solve $2N$ classification tasks with $2N - 1$ alternatives produced by other samples from the same mini-batch. The minimized objective is the cross-entropy loss:

$$l_{i,j} = -\log \frac{\exp(\mathrm{sim}(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum\limits_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\mathrm{sim}(\mathbf{z}_i, \mathbf{z}_k)/\tau)},$$

where $\mathrm{sim}(\mathbf{z}_i, \mathbf{z}_j)$ is a similarity metric which is chosen to be the cosine similarity

$$\mathrm{sim}(\mathbf{u}, \mathbf{v}) = \mathbf{u}^\top \mathbf{v} / \|\mathbf{u}\| \, \|\mathbf{v}\| \, .$$

As representations used in downstream tasks, SimCLR uses the outputs of an intermediate layer which is two layers before the output of the network (see Fig. 12.12) The intuition is that the formulated task of contrastive learning is likely to be substantially different from the downstream task and therefore the outputs $\mathbf{z}$ may not work well as features.

SimCLR uses three types of transformations to create two versions of the same image in the training procedure (see Fig. 12.13). The authors note that random cropping and color distortions give the largest boost in performance and it is important to use both transformations when transforming each individual sample. This effect can be explained by the fact that if only one of the two transformations is applied, then it is relatively easy to find the matching image among all images in a mini-batch. Therefore, the contrastive learning task becomes too easy and the model cannot develop powerful enough representations.

original | 1) random cropping followed by resize to the original size | 2) random color distortions | 3) random Gaussian blur

Figure 12.13: Augmentations used in SimCLR.

Table 12.4 shows the accuracy of linear classifiers trained using SimCLR features in comparison to CPC features. You can see that SimCLR achieves quite impressive performance reaching 93.2% of top-5 classification accuracy on ImageNet. Again, note that these features have been obtained in an unsupervised way.

Method	Architecture	Top 5 accuracy
CPC v2	ResNet-50	85.3
SimCLR	ResNet-50	89.0
CPC v2	ResNet-161	90.1
SimCLR	ResNet-50 (4×)	93.2

Table 12.4: ImageNet accuracies of linear classifiers trained on representations learned with different self-supervised methods.

Table 12.5 shows the performance of different methods tested in a semi-supervised scenario in which features are extracted using the full data set but the classifier is fine-tuned using only 10% of labels. You can see that SimCLR performs similarly to the algorithms specifically designed for the semi-supervised scenario. The second version of the model called SimCLRv2 (Chen et al., 2020b) further improved the performance in the semi-supervised scenario.

Table 12.5: Top-5 classification accuracy obtained using 10% labels from ImageNet in a semi-supervised setting. The encoder architecture is ResNet-50.

Method	Top-5 accuracy
Supervised baseline	80.4
Pseudo-label	83.4
VAT	83.4
UDA	89.1
FixMatch	89.1
SimCLR	87.8
SimCLRv2	93.4

12.4.3 Bootstrap your own latent (BYOL)

BYOL (Grill et al., 2020) can be seen as an extension of the Mean Teacher to the fully unsupervised scenario (see Fig. 12.14). The model contains two networks that process two transformations of the same image. The teacher network is obtained by computing exponential moving average of the student weights, just like in the Mean Teacher. The loss function that is minimized is the consistency loss between the outputs of the student and the teacher

$$\mathcal{L}_{\text{consistency}} = \|q(\mathbf{z}) - \mathbf{z}'\|^2 = \left\|q(f(\mathbf{x}, \boldsymbol{\theta}_t)) - f(\mathbf{x}', \boldsymbol{\theta}'_t)\right\|^2,$$

where $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}'_t$ are the parameters of the student and the teacher, respectively.

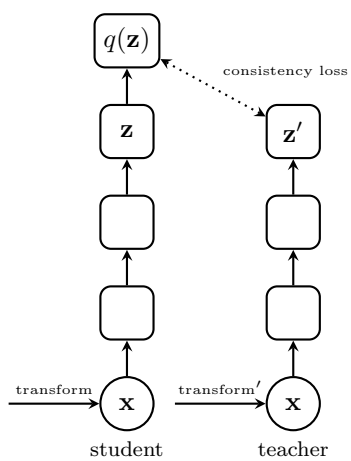


Figure 12.14: The architecture of the BYOL model.

There are two important differences compared to the Mean Teacher model. The first difference is the usage of strong augmentations both for the student and the teacher. BYOL uses the same set of augmentations as the SimCLR model (see Fig. 12.13). The second difference is that there is an extra predictor network $q(\mathbf{z})$ that is used to process the output of the student network before it is compared to the output of the teacher.

BYOL does not use the contrastive loss and therefore it may, in principle, suffer the problem of collapsed representations. If the network learns to produce the same output independent of the input, the consistency loss is zero. In

practice, however, the BYOL representations do not collapse. Interestingly, the representations collapse if the predictor network $q(\mathbf{z})$ is removed.

Table 12.6 shows the ImageNet accuracies of linear classifiers trained on representations learned with BYOL in comparison to other methods.

Method	Top-5 accuracy
CPC v2	85.3
SimCLR	89.0
BYOL	91.6

Table 12.6: Top-5 classification accuracy on ImageNet obtained with linear classifiers trained on representations learned with different self-supervised methods. The encoder architecture is ResNet-50.

In the semi-supervised scenario, when there is access to a bunch of labeled examples, the base network can be pre-trained with BYOL and then fine-tuned on the labeled data with spatial augmentations (random crops with resize and random flips). Table 12.7 shows that this training strategy yields very good performance.

Method	Top-5 accuracy
Supervised baseline	80.4
SimCLR	87.8
BYOL	89.0

Table 12.7: Top-5 classification accuracy obtained using 10% labels from ImageNet in a semi-supervised setting. The encoder architecture is ResNet-50.

12.5 Few-shot learning

The motivation for this line of research is in understanding the limitations of the deep learning models in comparison to the learning capabilities of humans. Humans can learn new concepts from just a single example. If you see a new object only once, for example, a drone or a segway (Fig. 12.15), you are likely to be able to recognize this object in the future. In contrast, deep learning models require thousands or even millions of training examples to solve image classification tasks. The task of few-shot learning is to build an accurate machine learning model using a very small number of training examples.

Figure 12.15: Humans can learn new concepts from just a single example.

The problem of few-shot learning has attracted a lot of attention since Lake et al. (2015) introduced the Omniglot challenge. Omniglot is a dataset that consists of 1.5 thousand characters from 50 alphabets. One part of the challenge is one-shot character classification (see Fig. 12.16). A single image of a new character is presented and the goal is to select another example of the same character among other characters from the same alphabet.

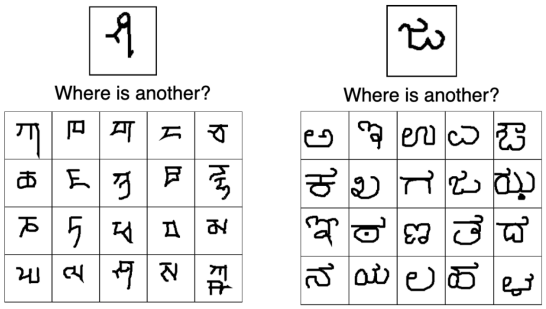


Figure 12.16: Part A of Omniglot challenge: Two trials of one-shot classification, where a single image of a new character is presented (top) and the goal is to select another example of that character amongst other characters from the same alphabet (in the grid below).

The authors of the Omniglot challenge proposed a model called BPL that represents concepts as simple programs that best explain observed examples under a Bayesian criterion. They showed that the BPL model can achieve human-level performance on the one-shot classification task unlike the existing deep learning solutions (see Table 12.17). However, there has been a lot of progress in the development of deep learning models to solve the few-shot learning problem (see Table 12.17). Next, we consider a few notable models: Siamese networks (Koch et al., 2015), Matching networks (Vinyals et al., 2016), Prototypical networks (Snell et al., 2017) and Model-Agnostic Meta-Learning (MAML; Finn et al., 2017).

	Original		**Augmented**	
	Within alphabet	Within alphabet (minimal)	Within alphabet	Between alphabet
background set				
# alphabets	30	5	30	40
# classes	964	146	3,856	4,800
2015 results				
Humans	$\leq$ 4.5%			
BPL	**3.3%**	**4.2%**		
Simple ConvNet	13.5%	23.2%		
Siamese Net			8.0%*	
2016-2018 results				
Prototypical Net	13.7%	30.1%	**6.0%**	4.0%
Matching Net				6.2%
MAML				4.2%
Graph Net				**2.6%**
ARC			**1.5%***	**2.5%***
RCN	7.3%			
VHE	18.7%			4.8%

* results used additional data augmentation beyond class expansion

Figure 12.17: One-shot classification error rate on Omniglot according to (Lake et al., 2019).

12.5.1 Siamese networks for one-shot learning

Siamese networks (Koch et al., 2015) is one of the simplest models for one-shot classification. In *one-shot* learning, we need to build a classifier using only a single training example from each class. Siamese networks are trained to compare a pair of examples to decide whether they belong to the same class or not, which is a binary classification problem.

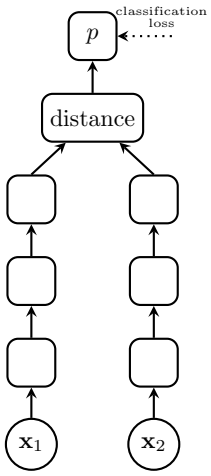


Figure 12.18: Siamese networks for one-shot classification.

Siamese networks contain two identical networks that process a pair of inputs (see Fig. 12.18). The probability that the input examples belong to the same class is computed using the distance between the outputs of the two networks. The network is trained on pairs of positive examples (when both inputs belong to the same class) and negative examples (when the inputs belong to distinct classes). The model works for one-shot learning but extending this idea to few-shot learning is not trivial.

12.5.2 Matching networks

Consider a single task of one-shot classification as presented in Fig. 12.19. There are four training examples in the data set, such that one example represents one class. The model needs to use the four examples to build a classifier which computes the probabilities of the four classes for a given test example.

The idea of the matching networks (Vinyals et al., 2016) is to treat the set of training and test samples as one training example of the one-shot classification task. In the standard classification problem, each image is one training example. In the formulation used by the matching networks, one training example consists of a few training samples (called a *support set*) and a few test samples (called a *query set*). We build a neural network that can process such training examples. This approach can be viewed as meta-learning: the classifier learns how to learn from a few labeled examples.

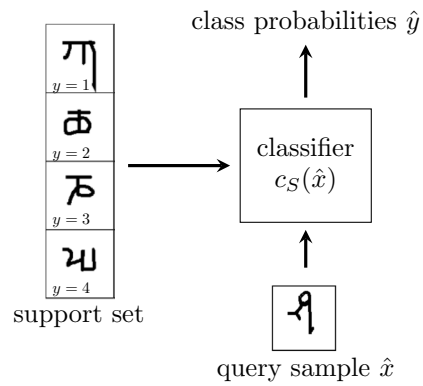


Figure 12.19: A single task of one-shot classification.

Each training task is to map from a support set of k examples of image-label pairs $S = \{(x_i, y_i)\}_{i=1}^k$ to a classifier $c_S(\hat{x})$:

$$S \to c_S(\hat{x}).$$

The classifier $c_S(\hat{x})$ defines a probability distribution $\hat{y}$ over classes in the support set for a query example $\hat{x}$. In matching networks, the mapping $S \to c_S(\hat{x})$ is parameterized with a neural network.

The output of the matching network classifier $c_S(\hat{x})$ is a weighted sum of the one-hot representations of the labels y_i from the support set:

$$\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i. \tag{12.2}$$

The weights a are computed using an attention mechanism:

$$a(\hat{x}, x_i) = \frac{\exp(c(f(\hat{x}),\, g(x_i)))}{\sum_{j=1}^{k} \exp(c(f(\hat{x}),\, g(x_j)))}. \tag{12.3}$$

The idea is that we pass the label of the support sample i whose encoding $g(x_i)$ is closest to the encoding $f(\hat{x})$ of the query sample $\hat{x}$. The encodings $f(\hat{x})$ and $g(x_i)$ are computed with deep neural networks.

Fig. 12.20 shows the computational graph that is built to process one classification task. The four support samples are encoded with a neural network g. The query sample is processed with a neural network f. In practice, f and g can be the same networks. Then, we compute pairwise distances between the encodings using function c and the prediction of the network is produced using (12.2)–(12.3). At training time, we know the correct label of the query example and therefore we can compute the cross-entropy loss for the given classification task.

Suppose that we want to train a classifier to separate N classes using K examples for each of the N classes. This scenario is often called *N-way K-shot* classification. One training iteration of the Matching networks consists of the following steps:

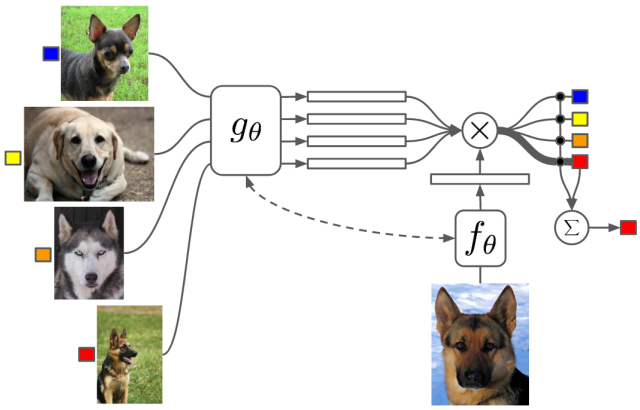


Figure 12.20: Computational graph of Matching networks.

- Select randomly N classes from the training set.
- Select a support set by taking K random samples for each of the selected classes.
- Select a query set: a few samples from the same classes as in the support set.
- Build a computational graph shown Fig. 12.20. Perform forward computations and compute the classification loss using the query samples.
- Perform backpropagation and update the parameters of the networks f and g to minimize the classification loss.

This type of training is often called *episodic training* because every training example contains one episode of the few-shot learning problem.

12.5.3 Prototypical networks

Prototypical networks (Snell et al., 2017) can be viewed as an extension of the Matching networks model. Similarly to the Matching networks, the Prototypical networks are trained on episodes of few-shot learning tasks. In every episode, there is a support set of training samples and a query set of test samples.

Just like in the matching networks, the support and query samples are encoded using an embedding function f which produces a lower-dimensional representation. The difference to the Matching networks is in the way the output of the classifier is produced. In the Prototypical networks, the embeddings that correspond to the same class in the support set are averaged to compute prototype vectors $\mathbf{c}_k$. This is illustrated in Fig. 12.21. The circles of different colors represent the embeddings obtained for the support set. The embedding vectors are averaged for each class to compute the prototype vectors

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(x_i, y_i) \in S_k} f_{\boldsymbol{\theta}}(x_i).$$

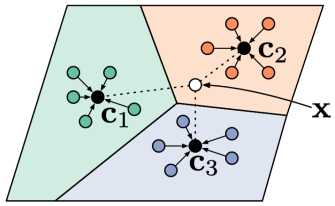


Figure 12.21: In Prototypical networks, the embeddings of the support set that correspond to the same class (marked with the same colors) are averaged to compute prototype vectors $\mathbf{c}_k$.

The prototype vectors are shown with the black circles in Fig. 12.21. There is one prototype vector for each class in the support set.

A query sample is embedded using the same embedding function f. The class of the query sample is predicted based on the distance of the embedding to the prototype vectors:

$$p(y = k \mid \mathbf{x}) = \frac{\exp(-d(f(\mathbf{x}), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f(\mathbf{x}), \mathbf{c}_{k'}))}.$$

The smaller the distance $d(f(\mathbf{x}), \mathbf{c}_k)$, the higher the probability $p(y = k \mid \mathbf{x})$ that the query sample $\mathbf{x}$ belongs to the corresponding class k. Note that in one-shot learning scenario, the Prototypical networks are equivalent to the Matching networks.

Training of the Prototypical networks is performed in a similar way to the Matchning networks. One iteration of the episodic training consists of the following steps:

- Embed the support set and average the embeddings of the different classes to compute the prototypes.
- Embed the query set and compute the predicted probabilities using the distances of the embeddings to the prototypes.
- Use the known labels y_j of the query samples to compute the cross-entropy loss.
- Compute the gradients of the loss and update the parameters of the embedding network.

These steps are illustrated in the computational graph in Fig. 12.22.

12.5.4 Model-Agnostic Meta-Learning

Suppose that we want to train a model that takes vector $\mathbf{x}$ as the input and produces $\mathbf{y}$ as the output to solve a new few-shot learning task. Suppose that the model is a neural network with parameters $\boldsymbol{\theta}$. Training of the model can be done by performing a few iterations of the gradient descent. If we do only one iteration, we have the following update rule for the parameters:

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}_0 - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}((\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_k, \mathbf{y}_k)), \tag{12.4}$$

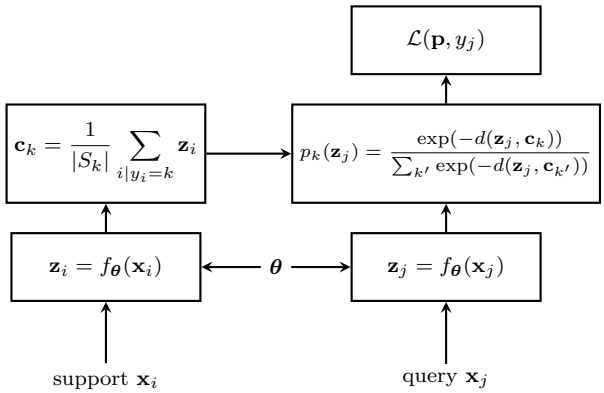


Figure 12.22: Computational graph of Prototypical networks.

where $(\mathbf{x}_i, \mathbf{y}_i)$ are the few training examples (the support set), $\mathcal{L}$ is the loss function (for example, the cross-entropy loss for classification tasks), α is the learning rate and $\boldsymbol{\theta}_0$ is a vector of the initial values of the parameters.

The idea of Model-Agnostic Meta-Learning (MAML; Finn et al., 2017) is to learn initialization $\boldsymbol{\theta}_0$ and the learning rate α so as to minimize the loss on the query set after the gradient-descent adaptation by (12.4). Just like the Matching networks and the Prototypical networks, MAML is trained in an episodic manner. In every episode, there is a support set which is used to update the model and the query set which is used to test the accuracy of the updated model. Each training iteration contains the following steps:

- Select a support set and use it to compute the loss and its gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$.
- Update the model parameters to $\boldsymbol{\theta}_0 \to \boldsymbol{\theta}'$ with one (or a few) iteration of gradient descent, as in (12.4). This step is implemented in the computational graph of the model (see Fig. 12.23).
- Use the query set to compute the loss with the adjusted parameters $\boldsymbol{\theta}'$.
- Perform backpropagation and update parameters $\boldsymbol{\theta}_0$ and learning rate α.

Note that MAML requires propagation of the gradient through the gradient computations performed in block $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ in Fig. 12.23, which can be computationally expensive. For that reason, the paper also proposes a first-order approximation in which the gradient is not propagated through $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ (see Fig. 12.23). This first-order approximation of MAML works almost equally well.

12.5.5 Reptile

Reptile (Nichol et al., 2018) can be viewed as a simplification of MAML. The idea is that instead of backpropagating through the computational graph (like we do in MAML), the initial parameters $\boldsymbol{\theta}_0$ are updated towards the parameter values $\boldsymbol{\theta}_T$ obtained as a result of adaptation in the current episode. One training iteration of Reptile consists of the following steps:

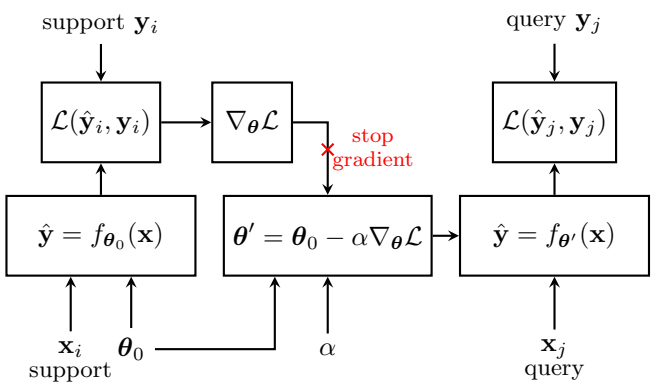


Figure 12.23: The computational graph of MAML. The first-order approximation stops gradient propagation through the marked link.

- Randomly sample a task T.
- Update the model parameters $\boldsymbol{\theta}_0 \to \boldsymbol{\theta}'$ with k iterations of gradient descent.
- Update the initialization with the following update rule:

$$\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_0 + \epsilon(\boldsymbol{\theta}' - \boldsymbol{\theta}_0).$$

You can look at the Reptile demo on the OpenAI web pages.