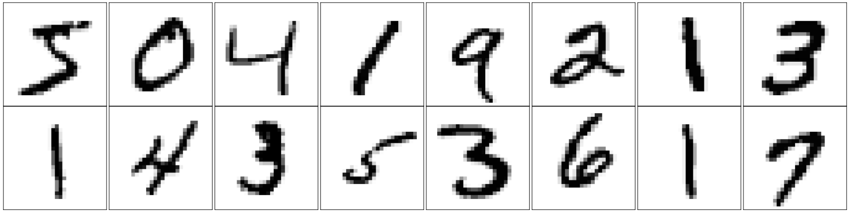
Aalto University

CS-E4890: Deep Learning
Convolutional neural networks

Alexander Ilin

Image classification task

- Example classification problem: Classify 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.



Spatial structure matters

- If we change the order of the pixels (in the same way for all images), the classification task becomes much harder for humans.
- This suggests that our model can and should benefit from using the spatial information.

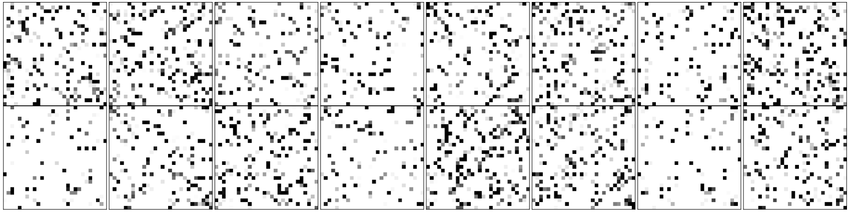
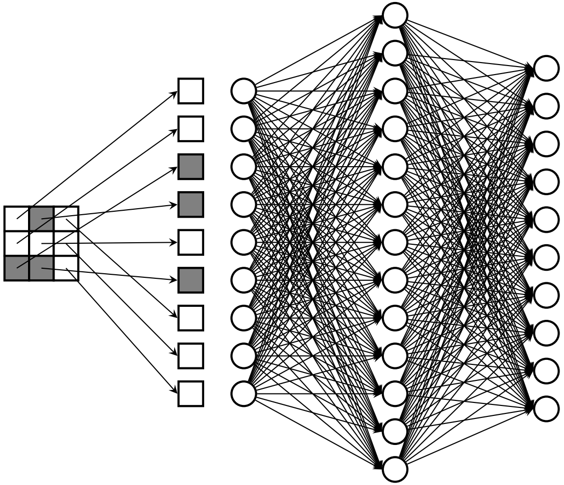


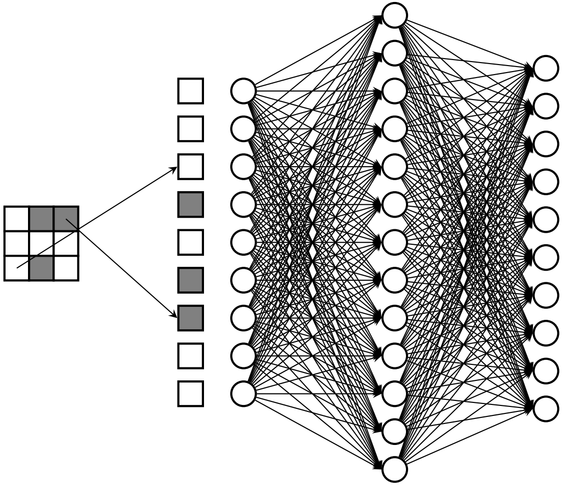
Image classification with a multilayer perceptron

- We can solve the classification task using a multilayer perceptron model (MLP) that we considered in the first lecture.
- We can flatten the images (for example, stack the columns of the images into one vector) and feed to the MLP model.



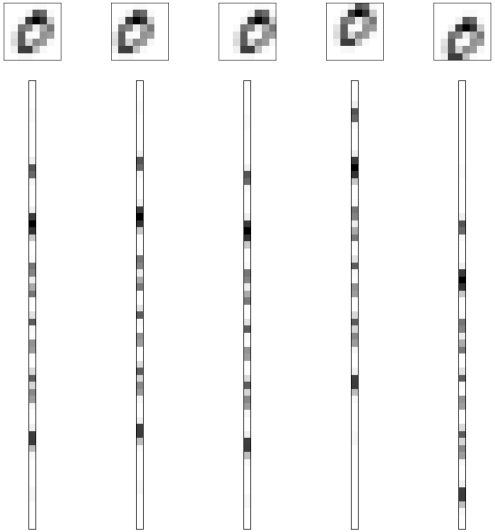
Problem 1: MLP ignores the spatial structure

- If we shuffle the pixels, we simply feed the pixels into different inputs of the MLP.
- This means that the MLP ignores the spatial structure and essentially solves a more difficult problem.



Problem 1: MLP ignores the spatial structure

- 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.

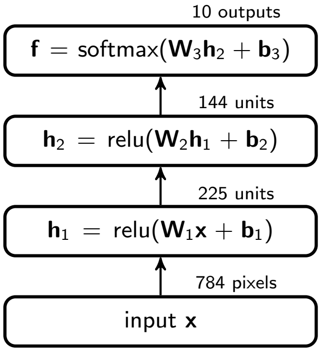


Problem 2: Number of parameters

- Let us use an MLP with the following structure to solve the MNIST classification task.
- Let us count the number of parameters in the network (ignoring the bias terms **b**):

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

- If we want to process images that contain millions of pixels, the number of parameters would be several orders of magnitude larger.



10 outputs
$\mathbf{f} = \text{softmax}(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$
144 units
$\mathbf{h}_2 = \text{relu}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
225 units
$\mathbf{h}_1 = \text{relu}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
784 pixels
input $\mathbf{x}$

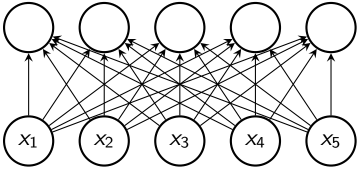
Motivation for a layer of a new type

- We want to design an alternative to the fully-connected layer that would address these problems:
  - Take into account the order of the inputs
  - Reduce the number of parameters
  - Change the outputs in a predictable way for simple transformations such as translation

Convolutional layer

Fully-connected layer as a starting point

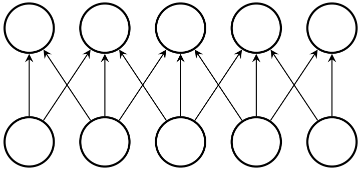
- Let us consider an input with one-dimensional structure. For example, we want to process time series and the order of the inputs is determined by the time of the measurements.
- Let us start with a fully-connected layer that has 5 inputs and 5 outputs:



- The layer has $5 \times 5 = 25$ parameters (ignoring the bias terms).

Local connectivity

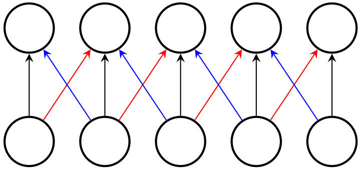
- We can reduce the number of parameters by using only local connections.
- Now the outputs also have an order because each output corresponds to a particular location in the inputs.



- The layer has now 13 parameters.

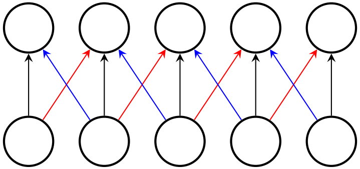
Parameter sharing

- We can further reduce the number of parameters by using weight sharing (arrows with the same color represent shared weights).
- Now the layer has only 3 parameters.



- Why parameter sharing is useful: patterns that appear in different parts of the input sequence will activate neurons in the corresponding location of the output layer.

1d convolutional layer



- The computations performed in such a layer:

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

- The layer is called a (one-dimensional) *convolutional* layer because the computations are closely related to (one-dimensional) convolution:

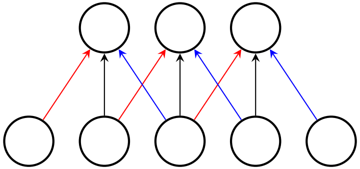
$$(w * x)[t] = \sum_a w[a] x[t-a]$$

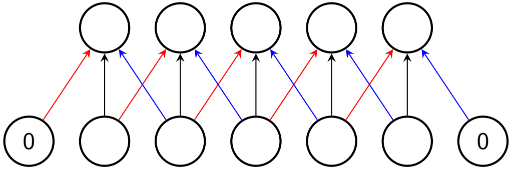
1d convolutional layer

- Inputs and outputs of such a layer usually contain multiple elements (usually called *channels*):

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

- Weights $w_{\Delta i,o,c}$ are usually called *kernel*.
- There are two ways to process inputs at the borders:

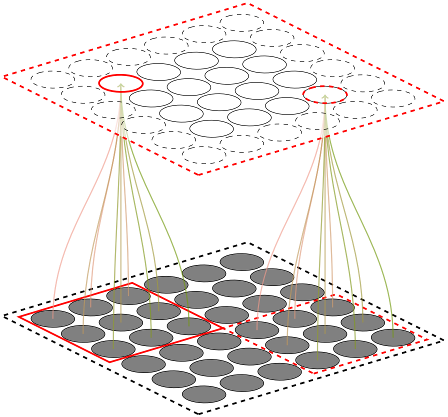

no padding (different output size)



padding (usually with zeros)

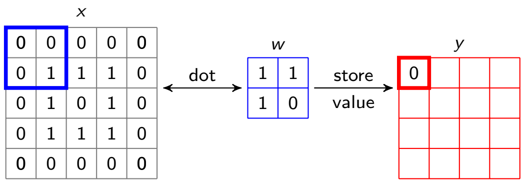
Inputs with 2d structure

- Same ideas can be used for inputs with 2d spatial structure like images.
    - Local connectivity: output is affected by inputs in its neighborhood.
    - Shared parameters: same colors represent shared weights.



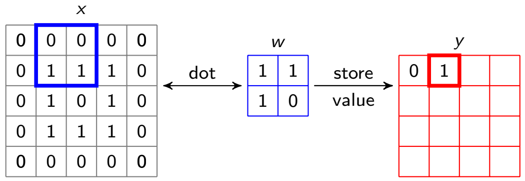
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



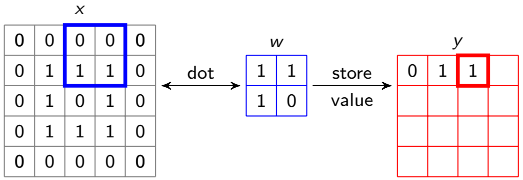
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



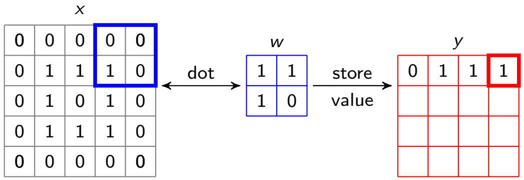
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



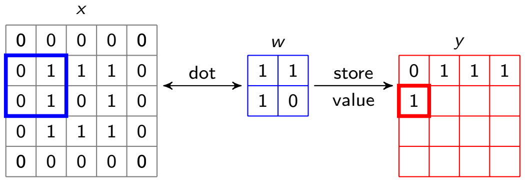
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



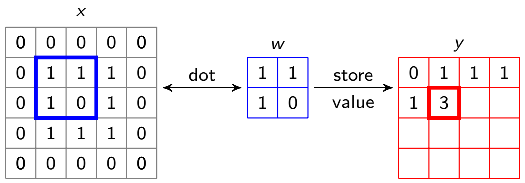
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



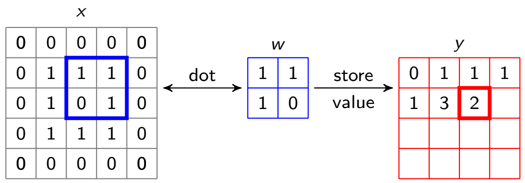
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



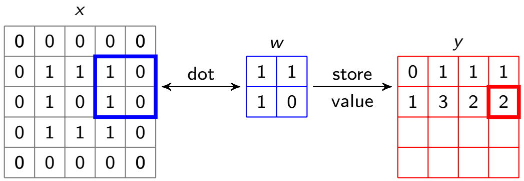
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



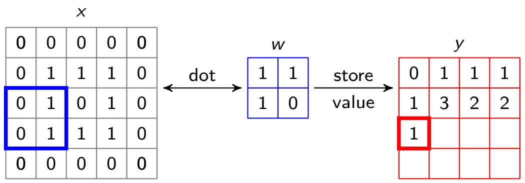
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



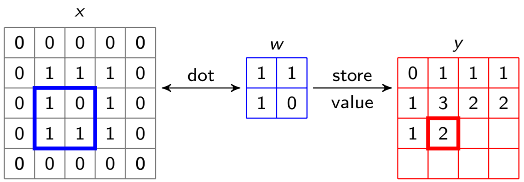
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



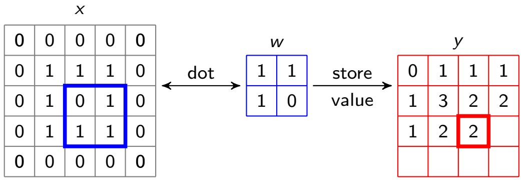
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



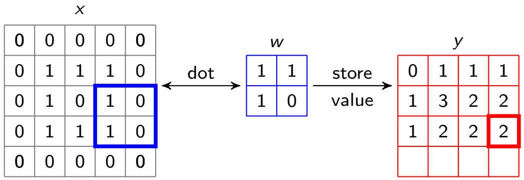
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



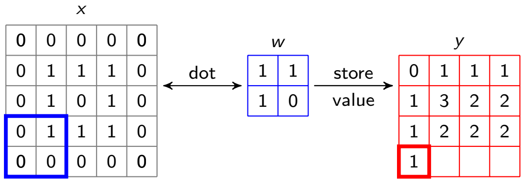
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



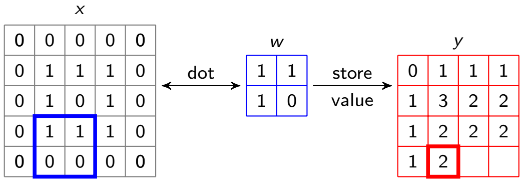
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



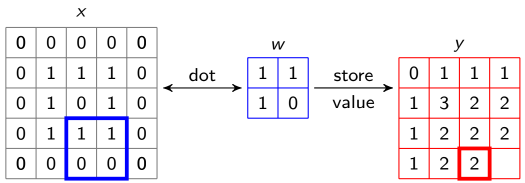
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



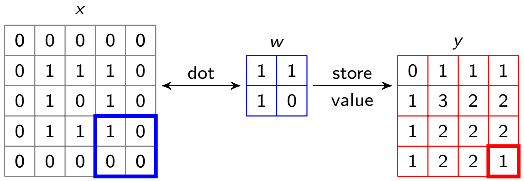
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



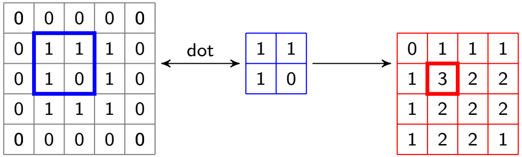
2d convolutional layer: Forward computations

- Simplified example: inputs with one channel (black-and-white images) outputs with one channel.
- Slide the filter across the entire input and compute dot products between input entries and filter weights.
- Computations can be parallelized.



2d convolutional layer as feature detector

- We can view the filter that we used in this example as a simple feature detector.
- Note that the filter has the shape of a corner. And the output is maximum at the position where this corner is present in the input image.



0	0	0	0	0
0	1	1	1	0
0	1	0	1	0
0	1	1	1	0
0	0	0	0	0

1	1
1	0

0	1	1	1
1	3	2	2
1	2	2	2
1	2	2	1

2d convolutional layer with multiple channels

- Inputs may contain multiple channels (RGB images).
- We need to detect multiple features using multiple filters. Therefore, a convolutional layer contains multiple channels:

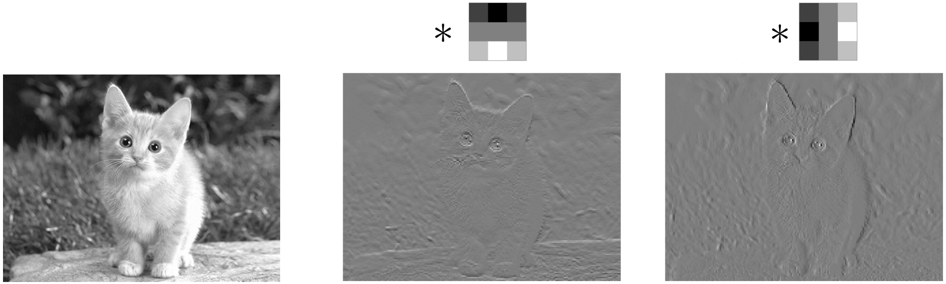
$$y_{i,j,o} = \sum_{\Delta i} \sum_{\Delta j} \sum_{c} w_{\Delta i, \Delta j, o, c} x_{i+\Delta i, j+\Delta j, c} + b_o$$

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

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

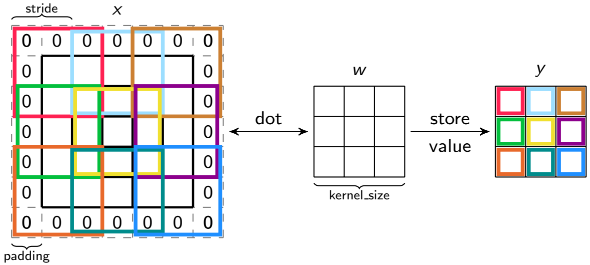
Convolution $*$

- One filter learns to detect a feature in the input:



2d convolutional layer in PyTorch

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



- Convolution visualization
- The size of the output will be different:

$$H_o = \frac{1}{s}(H_i + 2p - k - (k-1)(d-1)) + 1$$

Why do we need padding?

- With padding, the output of a convolutional layer can have the same height and width as the input.
    - It is easier to design networks when the height and width is preserved.
    - To use skip connections $\mathbf{x} + \text{conv}(\mathbf{x})$, like in ResNet, we need the dimensions to match.
- With padding, we can use deeper networks. Without padding, the size would reduce quickly with adding new layers.
- Padding improves the performance by keeping information at the borders.

Convolutional layer is equivariant to translation

- 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.

$$\begin{array}{|c|c|c|c|c|}\hline 0&0&0&0&0\\\hline 0&1&1&1&0\\\hline 0&1&0&1&0\\\hline 0&1&1&1&0\\\hline 0&0&0&0&0\\\hline\end{array} \xleftrightarrow{\text{dot}} \begin{array}{|c|c|}\hline 1&1\\\hline 1&0\\\hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline 0&1&1&1\\\hline 1&3&2&2\\\hline 1&2&2&2\\\hline 1&2&2&1\\\hline\end{array} \qquad \begin{array}{|c|c|c|c|c|}\hline 0&0&0&0&0\\\hline 0&0&1&1&1\\\hline 0&0&1&0&1\\\hline 0&0&1&1&1\\\hline 0&0&0&0&0\\\hline\end{array} \xleftrightarrow{\text{dot}} \begin{array}{|c|c|}\hline 1&1\\\hline 1&0\\\hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline 0&0&1&1\\\hline 0&1&3&2\\\hline 0&1&2&2\\\hline 0&1&2&2\\\hline\end{array}$$

- A function f is *equivariant* with respect to T if

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

- A function f is *invariant* with respect to a transformation T if

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

  The result through f does not change when you apply the transformation to the input.

Convolutional networks

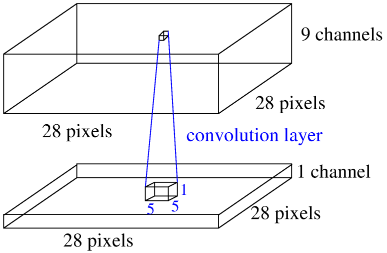
Example: MNIST classification

- Let us build a convolutional neural network (a network with convolutional layers) to solve the MNIST classification task.
- The input is 28×28 pixels and 1 channel.
- First convolutional layer:
  - 9 filters with 5×5 kernel and padding.
- First hidden layer: 28×28 pixels and 9 channels
- The number of parameters in the first layer (ignoring biases):

$$5 \times 5 \times 9 = 225$$

- Compare with the fully connected layer:

$$28 \times 28 \times 225 = 176400$$

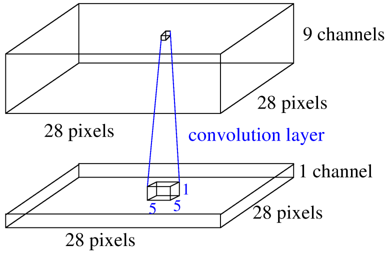


Example: MNIST classification

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

$$28 \times 28 \times 9 = 7056$$

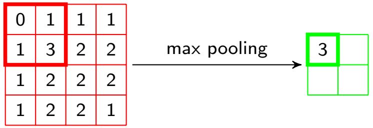
- Compare with the fully connected layer: 225
- The number of intermediate signals is much larger in the convolutional layer. To process such a high-dimensional signal, we need a significant amount of computations in the rest of the network.



- 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*.

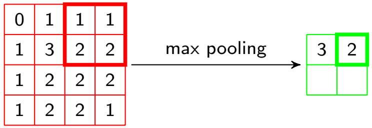
Pooling layer

- A common way is to take the maximum value in a small window (max pooling).
- For instance if we use max pooling with a filter of size 2x2 we discard 75 percent of the values.
- Pooling helps to make the representation approximately invariant to small translations of the input.



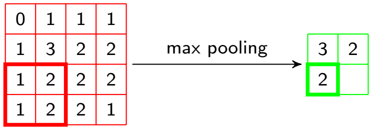
Pooling layer

- A common way is to take the maximum value in a small window (max pooling).
- For instance if we use max pooling with a filter of size 2x2 we discard 75 percent of the values.
- Pooling helps to make the representation approximately invariant to small translations of the input.



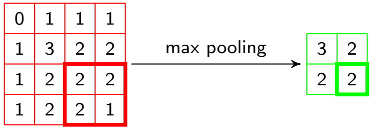
Pooling layer

- A common way is to take the maximum value in a small window (max pooling).
- For instance if we use max pooling with a filter of size 2x2 we discard 75 percent of the values.
- Pooling helps to make the representation approximately invariant to small translations of the input.



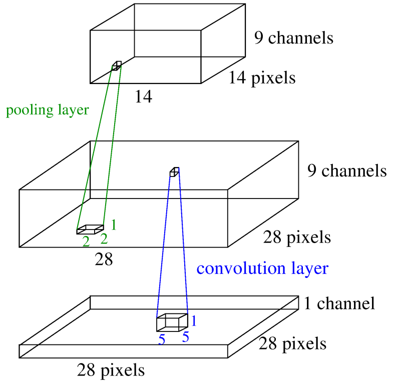
Pooling layer

- A common way is to take the maximum value in a small window (max pooling).
- For instance if we use max pooling with a filter of size 2x2 we discard 75 percent of the values.
- Pooling helps to make the representation approximately invariant to small translations of the input.



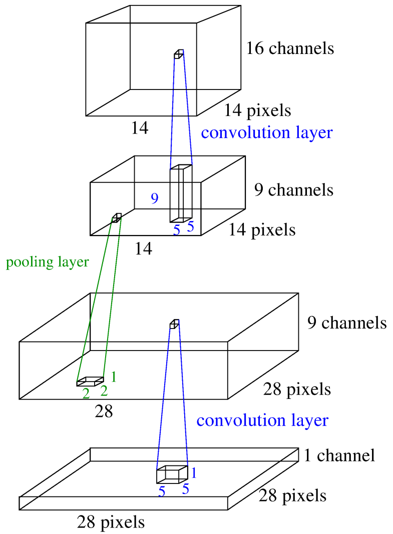
Example: MNIST classification

- After adding a 2×2 pooling layer.



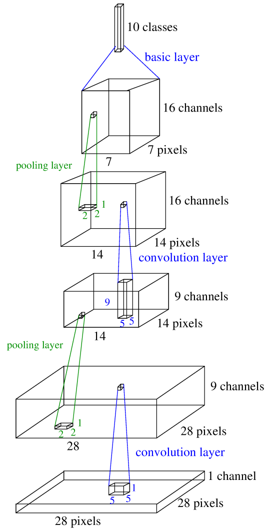
Stack more layers

- Note: Each unit looks at all the channels of the previous layer.



Full network

- Finally, we flatten the outputs of the last convolutional layer and feed them to a fully-connected layer with 10 outputs.
- We apply the softmax nonlinearity to the outputs and use the cross-entropy loss.
- The network can be trained by any gradient-based optimization procedure, for example, Adam.
- The gradients are computed by backpropagation as in the multilayer perceptron. The biggest difference is that we need to take into account parameter sharing inside the convolutional layers.



Backpropagation through a convolutional layer

- Forward computations in a convolutional layer:

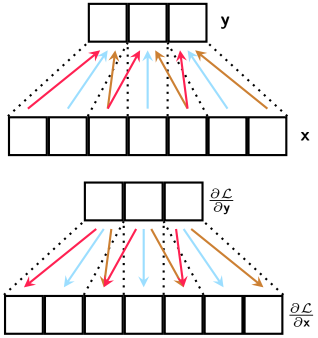
$$y_{i,j,o} = \sum_{\Delta i} \sum_{\Delta j} \sum_{c} w_{\Delta i,\Delta j,o,c} x_{i+\Delta i,j+\Delta j,c} + b_o$$

- Backward computations in a convolutional layer:

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

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

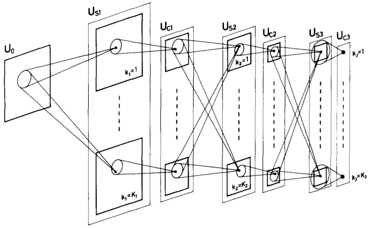
- The latter operation is called *transposed convolution*.


y
x
$\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$

Modern convolutional neural networks

Historical note: First convolutional networks

- Fukushima (1980) proposed neocognitron, a neural network architecture with:
  - Multiple layers of local feature detectors
  - Weight sharing
- (LeCun et al., 1989): A convolutional network applied to handwritten character recognition
  - The method became the basis of a nationally deployed check-reading systems.

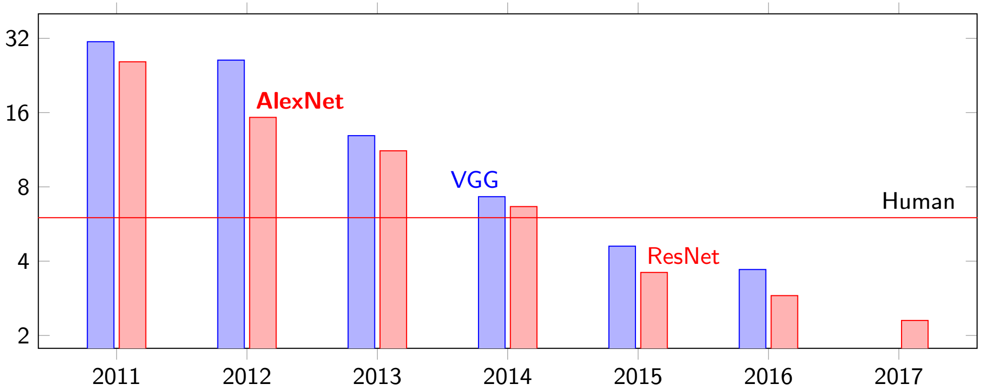


Fukushima's Neocognitron

- (Waibel et al., 1989): Time-delay neural network which were similar to conv nets but applied to audio (in a moving window).
- (LeCun et al., 1998): LeNet-5, a classical architecture of a convolutional neural networks.

Imagenet progress

Classification error % (top-5)



AlexNet (Krizhevsky, 2012)

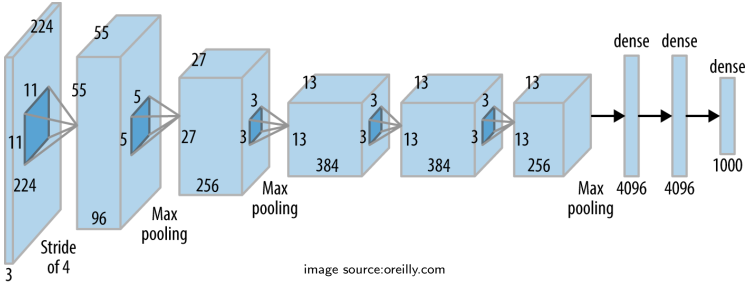
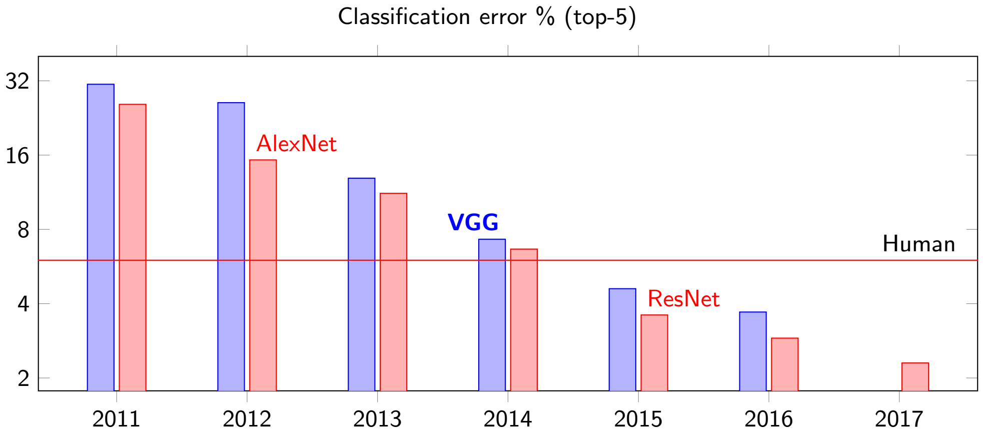


image source:oreilly.com

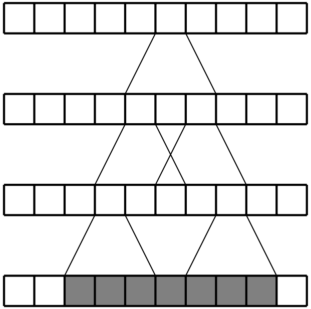
- Five convolutional layers and three fully-connected layers
- ReLU nonlinearities after convolutional layers, dropout

Imagenet progress



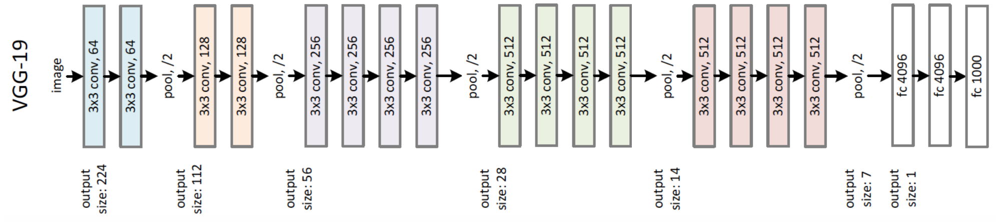
VGG-19 (Simonyan & Zisserman, 2015)

- Suppose we have c input and c output channels.
- One convolutional layer with 7×7 filters:
  - $49c^2$ parameters
- If we stack three 3×3 conv layers:
  - Effective receptive field is 7×7
  - $27c^2$ parameters (45% less)

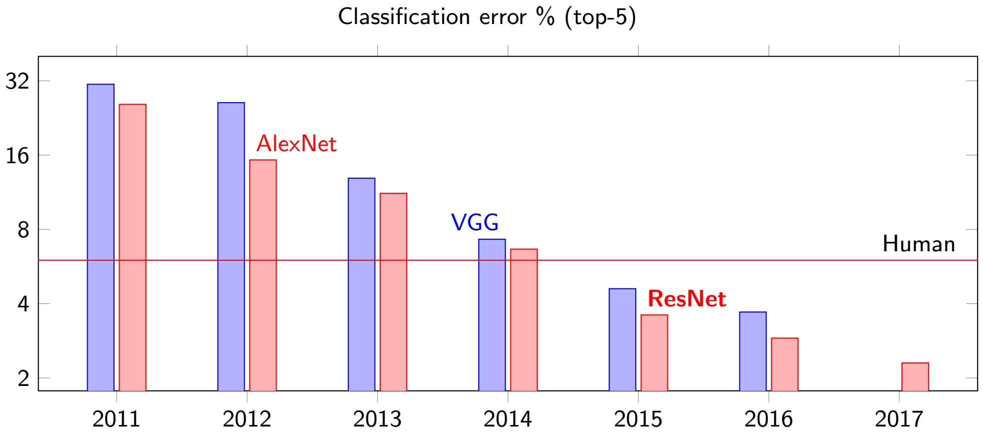


VGG-19 (Simonyan & Zisserman, 2015)

- Compared to AlexNet:
  - Smaller (3×3) filters
  - Deeper network (more layers)

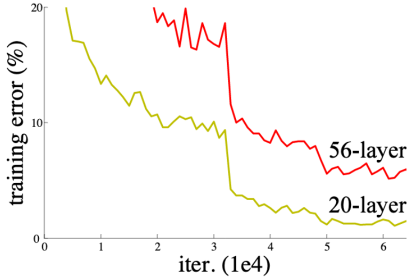


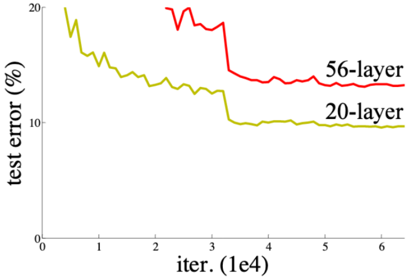
Imagenet progress



ResNet (He et al, 2016)

- Training deeper networks is a more difficult optimization problem:

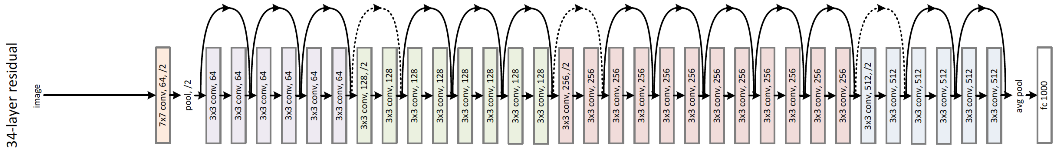




  - Experiments: deeper networks tend to have higher training error.
  - Deeper networks should **not** produce higher training error compared to more shallow networks (extra layers can learn simple identity mappings if needed).

ResNet (He et al, 2016)

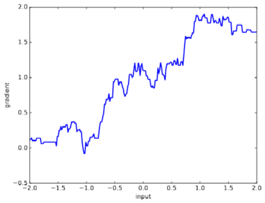
- ResNet:
    - Instead of learning $f(\mathbf{x})$, layers learn $\mathbf{x} + h(\mathbf{x})$.
    - If an identity mapping is optimal, it is easier to push residual $h(\mathbf{x})$ to zero than to learn an identity mapping with $f(\mathbf{x})$.



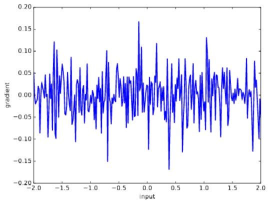
- Compared to VGG:
    - Skip connections
    - More layers

Why residual connections help training

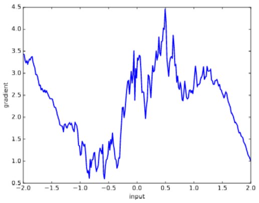
- Balduzzi et al. (2017) identified the problem with gradients in deep networks without residual connections. Experiment with a randomly initialized MLP $f : \mathbb{R} \to \mathbb{R}$, each hidden layer contains 200 neurons with ReLU activations:



1-layer feedforward



24-layer feedforward



50-layer resnet

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

- Gradients are *shattered* for deep network without skip connections: Small changes of the input has significant effect on the gradient. Thus the optimization becomes more difficult.

Batch normalization in convolutional networks

- Batch normalization facilitates faster convergence of the optimization procedure.
- `BatchNorm2d`: The batch statistics are computed across all examples in a mini-batch and all pixels.

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

  $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ have as many elements as there are channels in $\mathbf{z}$.

- Each channel c of the input map is transformed using the batch statistics and the BatchNorm parameters γ_c and β_c:

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

Applications of convolutional networks

Advantages of convolutional networks

- Advantages of convolutional networks
  - Take into account the order of the inputs.
  - Can process input sequences with varying lengths (due to parameter sharing).
  - The computations can be effectively parallelized.
- For these reasons, convolutional networks have been used for processing images, text data, speech, analyzing game positions and even for predicting protein folding.

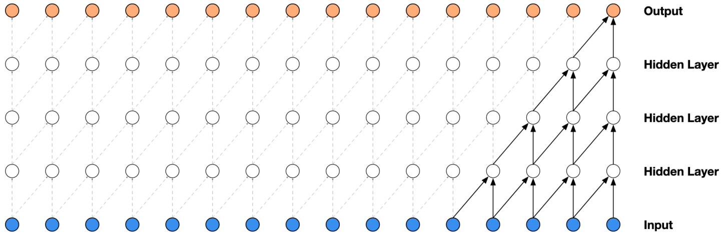
Temporal convolutions

- WaveNet (van den Oord et al., 2016): an autoregressive model of speech:

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

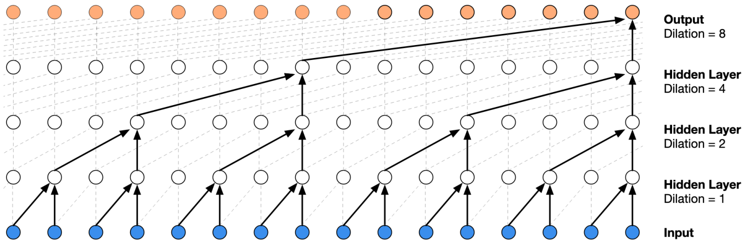
- The conditional distribution $p(x_t | x_1, \dots, x_{t-1})$ is modeled with a 1d convolutional network:

One needs lots of layers to model long-term dependencies



WaveNet: Dilated convolutions

- Stack of *dilated* causal convolutional layers:

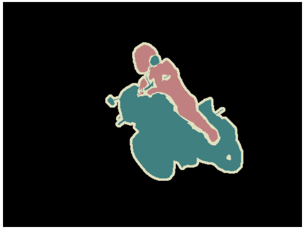


- Dilated convolutions allow fast growth of the receptive field which is good for modeling long-term dependencies.
- WaveNet (van den Oord, 2016) is the state-of-the-art model for speech generation.

Semantic segmentation

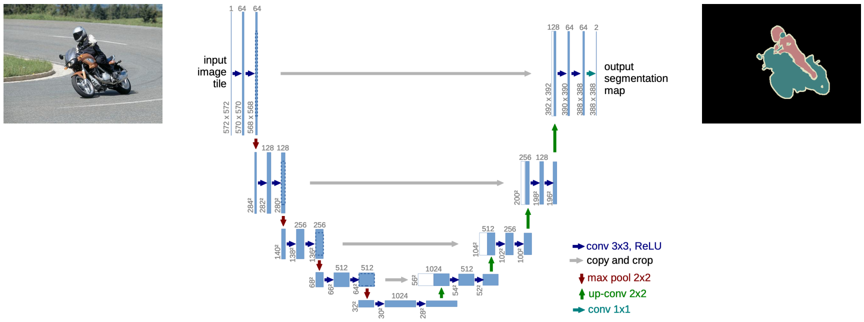
- Segmentation: Generating pixel-wise segmentations giving the class of the object visible at each pixel, or "background" otherwise.

input image


output segmentation map

- We need to classify each pixel of the input image.

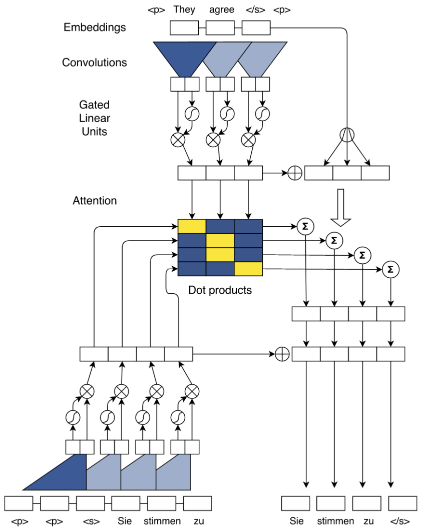
Semantic segmentation with U-Net (Ronneberger et al, 2015)



- The contracting path (left side) is needed to extract high-level features.
- The expansive path (right side) is needed to make the classification decisions on the pixel level (transposed convolutions are used here). The expansive path uses representations from the contracting path (via skip connections and concatenation).

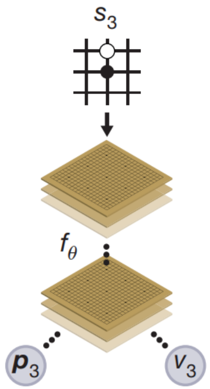
Convolutional model for neural machine translation (Gehring et al., 2017)

- Translation task: we need to translate a sentence in the source language into a sentence in the target language.
- Convolutional networks are used to encode the source sentence (sequence of words) and use that representation to compute the probabilities of words in the output sequence.



Convolutional networks in reinforcement learning

- In RL, convolutional networks are used to process sensory inputs with two-dimensional structure.
- Example: AlphaZero (Silver et al, 2017), an RL algorithm that achieves superhuman performance in the games of Go, chess and shogi.
- Deep convolutional networks are used to compute the probability of the next move $\mathbf{p}_t$ and the probability v_t that the player wins the game from the current position (to build a search tree).



Protein folding problem (DeepMind blog)

- Proteins are large, complex molecules essential to all of life. What any given protein can do depends on its unique 3D structure.

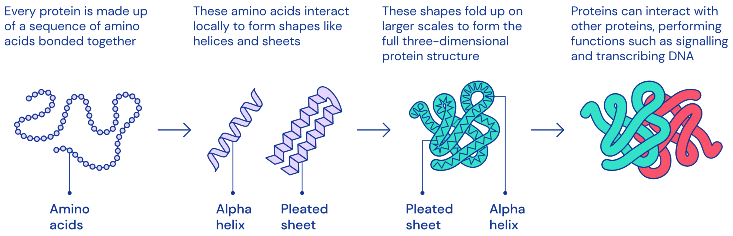
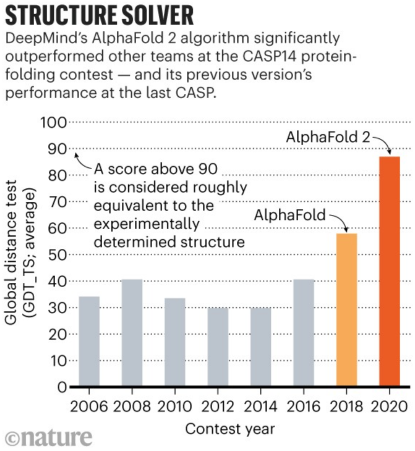

Every protein is made up of a sequence of amino acids bonded together
These amino acids interact locally to form shapes like helices and sheets
These shapes fold up on larger scales to form the full three-dimensional protein structure
Proteins can interact with other proteins, performing functions such as signalling and transcribing DNA
Amino acids
Alpha helix
Pleated sheet
Pleated sheet
Alpha helix

FIGURE 1: COMPLEX 3D SHAPES EMERGE FROM A STRING OF AMINO ACIDS.

- Proteins are comprised of chains of amino acids. The information about the sequence of amino acids is contained in DNA.
- Protein folding problem: Predicting how these chains will fold into the 3D structure of a protein.

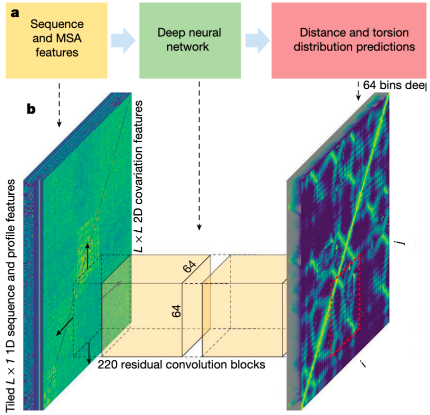
CASP competition

- The Critical Assessment of protein Structure Prediction (CASP): a biennial global competition established in 1994, is the standard for assessing predictive techniques.
- In 2020, DeepMind's AlphaFold 2 model has achieved "unprecedented progress in the ability of computational methods to predict protein structure".



AlphaFold 2 (Senior et al., 2020)

- The input of the model is a sequence of amino acids. Each sequence is represented as a 2d 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 output is the distances between the C_β atoms of pairs of amino acid residues of a protein. The output can also be represented as a 2d matrix.
- A convolutional neural network is used to predict the outputs from the inputs.



a
Sequence and MSA features
Deep neural network
Distance and torsion distribution predictions
64 bins dee
b
Tiled L × 1 1D sequence and profile features
L × L 2D covariation features
64
64
220 residual convolution blocks
j
i

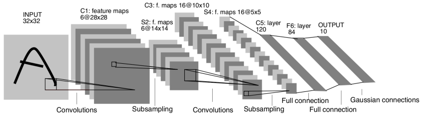
Recommended reading

- Chapter 9 of Deep Learning book
- References in the slides

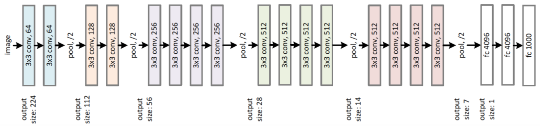
Home assignment

Assignment 03_cnn

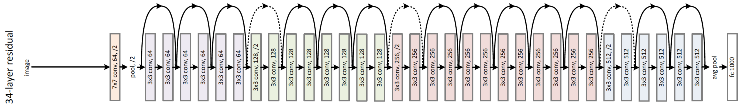
- Implement and train three convolutional networks
  1. CNN inspired by classical LeNet-5



  2. VGG-style network



  3. ResNet


34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000