CS-E4890: Deep Learning
Introduction

Alexander Ilin

Teaching assistants

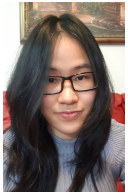
Arttu Häkkinen

Daolang Huang

Grzegorz Woźniak

Kim Tarvainen

Trung Trinh


Hankio Linh

Kalle Kujanpää

Oscar Vikström

Lukas Prediger

Severi Rissanen

Sanna Lun

Nicola Dainese

Contact information

If you have question regarding the course, please send an email to

cs-e4890@aalto.fi

Pre-requisites

- **Good knowledge of Python and numpy.**
- Linear algebra: vectors, matrices, eigenvalues and eigenvectors.
- Basics of probability and statistics: sum rule, product rule, Bayes' rule, expectation, mean, variance, maximum likelihood, Kullback-Leibler divergence.
- Basics of machine learning (recommended): supervised and unsupervised learning, overfitting.

Course schedule

- Please study carefully the course schedule in mycourses.
    - 12 lectures (11 + 1 guest lecture by Kyunghyun Cho)
    - 10 assignments (the points are computed from 8 best)
    - Exercise sessions for assignments 1–8 (no exercise sessions for assignments 9–10).
    - No exam (there is a placeholder for the exam in SISU but no exam this year).

Communication channels

- Slack is the main communication channel: deeplearn22-aalto.slack.com
- Please ask questions about assignments in the dedicated channels.
- The teaching assistants (TAs) will look at slack regularly.
- Please read about the slack etiquette in file `0_rules.ipynb` in the first assignment.

Rules

- **By taking this course, you accept the following rules:**
    - You give permission for proctoring your submissions.
    - Solution sharing is strictly not allowed before, during and after the course. That means that you are not allowed to share your solutions (or any parts) via private channels and/or public repositories.

Course grading

- 5 credit points, 1-5 scale
- Grading is based on the number of points collected in eight best assignments. The grading rules are explained in mycourses.
- The course workload (5 credits) assumes solving eight assignments, the two extra assignments give you the possibility to improve your grade.

Assignments

- The assignments are released already.
- **Please read very carefully the instructions.**
- You can find the deadlines on the course schedule page in mycourses.
- **Strict deadlines, zero points for late submissions, no exceptions.**
- The feedback is returned on the same week after the deadline.
- If you plan to be away, submit your solutions early, no need to wait until the deadline.

Exercise sessions

- The exercise sessions are organized to help you solve the assignments, you do not have to attend them.
- There will be four exercise sessions for assignments 1–8, typically on:
  - Fridays 10:15-11:45
  - Fridays 12:15-13:45
  - Mondays 10:15-11:45
  - Mondays 12:15-13:45

  Note exceptions because of the Easter.
- No exercise sessions for assignments 9–10!
- The exercise sessions are organized over zoom.
- Please read carefully the protocol for the exercise sessions.

PyTorch

- We will not have special sessions on PyTorch, you should learn it by following online material.
    - If you know numpy (pre-requisite), PyTorch should be easy to learn.
    - Deep learning frameworks develop very quickly. If you do deep learning, you need to learn new frameworks/features all the time.
- If you need help with PyTorch and/or solving the assignments, please come to the exercise sessions!
- Why not Tensorflow or Keras?
    - Tensorflow is cumbersome, PyTorch is easier to learn and to use.
    - Keras is good if you want to try a deep learning model quickly. But it is more restrictive. If you want to be a professional deep learner, PyTorch is a better choice.

Course material

- Book Deep Learning by Goodfellow, Bengio and Courville (2016).
- Lecture notes available at the course web pages.
- PyTorch tutorials
- Papers and links in the lecture slides and the assignments

Lectures

- The lectures will be recorded and will be available in mycourses.
- The lecture slides are in mycourses.
- There will be changes in the slides, please download the latest version before each lecture.
- Credit to people whose material I used in the slides: Tapani Raiko, Kyunghyun Cho, Jyri Kivinen, Jorma Laaksonen, Antti Keurulainen, Sebastian Björkqvist.

What is deep learning

Feature engineering

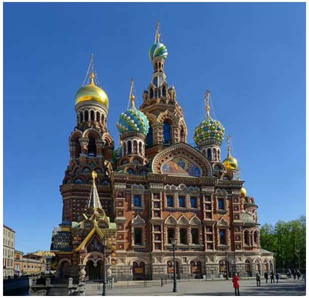
- Many machine learning tasks can be solved by designing the right set of features to extract for that task:

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

- Examples:
  - Spam detection: Useful features are counts of certain words.
  - Line item extraction from invoices: Useful features to classify a number as a line item or not are position on the invoice, words that appear in the proximity.
- Benefit of feature engineering: One can use domain knowledge to design features that are robust (for example, invariant to certain distortions).
- What are the problems with feature engineering?

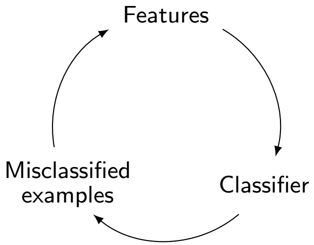
Feature engineering: Problem 1

- For many tasks, it is difficult to know what features should be extracted.
- Example: We want to detect certain buildings in images (two-dimensional maps of RGB values). What are useful features?
- 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.
    - Example: SIFT features in image classification.



Feature engineering: Problem 2

- Handcrafted features are not perfect. There are always examples that are not processed correctly, which motivates engineering of new features.



- Features can get very complex and difficult to maintain.

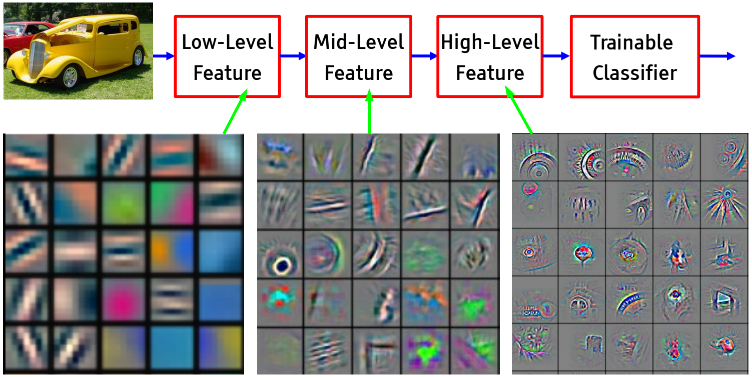
Representation learning

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

- These problems can be overcome with **representation learning:** We use machine learning to discover not only the mapping from representation to output but also the representation itself.
- 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).
- Learned representations often result in much better performance compared to hand-designed representations.
- With learned representations, AI systems can rapidly adapt to new tasks, with minimal human intervention.

Deep learning = Learning hierarchical representations

- Deep learning does representation learning by introducing representations that are expressed in terms of other, simpler representations.

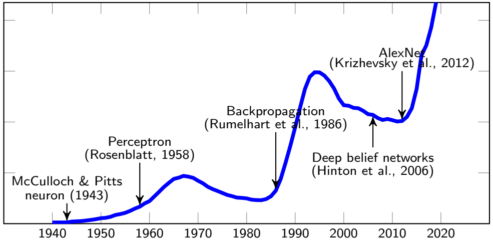


Feature visualization of convolutional net trained on ImageNet from (Zeiler and Fergus, 2013).

Deep learning = artificial neural networks

- Many ideas in deep learning models have been inspired by neuroscience:
  - The basic idea of having many computational units that become intelligent only via their interactions with each other is inspired by the brain.
  - The neocognitron (Fukushima, 1980) introduced a powerful model architecture for processing images that was inspired by the structure of the mammalian visual system and later became the basis for the modern convolutional networks.

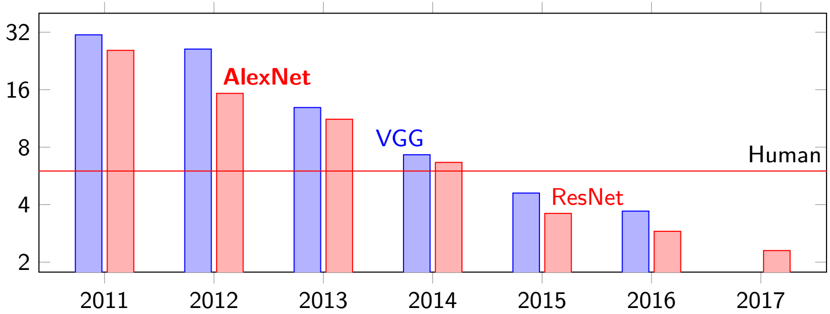
- The name "deep learning" was invented to re-brand artificial neural networks which became unpopular in 2000s.
- Modern deep learning: A more general principle of learning multiple levels of composition, which can be applied in machine learning frameworks that are not necessarily neurally inspired.



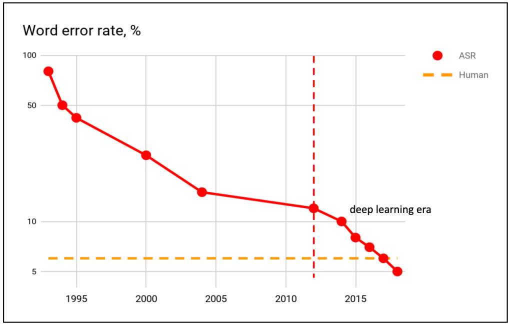
Frequency of phrases "cybernetics", "neural networks" and "deep learning" according to Google books.

Imagenet: The beginning of the deep learning era

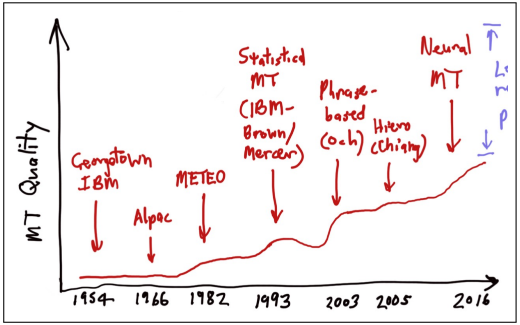
- Imagenet: Yearly competition in image classification with a thousand classes using a training set: with millions of images.
- Krizhevsky, Sutskever and Hinton (2012) won the Imagenet competition by a large margin using a deep convolutional neural network.



Deep learning improved results almost in every domain



Speech recognition
(Graves and Jaitly, 2014)



Neural machine translation
(Cho et al., 2014)

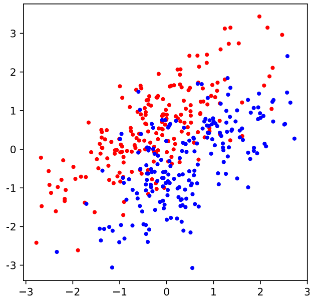
Linear classifiers

Logistic regression

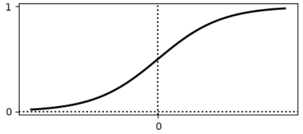
- Consider a binary classification problem: Our training data consist of examples $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})$ with $\mathbf{x}^{(i)} \in \mathbb{R}^m$ $y^{(i)} \in \{0, 1\}$.
- We use the training data to build a linear classifier

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

  where m is the number of features in $\mathbf{x}$.
- Logistic regression model: $\sigma(x) = \frac{1}{1+e^{-x}}$ is a logistic function.
- Using the logistic function guarantees that the output is between 0 and 1 and it can be seen as the probability that $\mathbf{x}$ belongs to one of the classes: $p(y = 1 \mid \mathbf{x}) = f(\mathbf{x})$.



Training examples



Logistic function

Likelihood function for logistic regression model

- We can tune the model assuming the Bernoulli distribution for the label y:

$$p(y \mid \mathbf{x}, \mathbf{w}, b) = f(\mathbf{x})^y (1 - f(\mathbf{x}))^{1-y} \qquad \text{where } f(\mathbf{x}) = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

- For n training examples, the likelihood function is

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

  where $p(y^{(i)} \mid \mathbf{x}^{(i)})$ is a function of the model parameters $\mathbf{w}$ and b.
- This gives the following log-likelihood function:

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

- We can either maximize the log-likelihood function $\mathcal{F}(\mathbf{w}, b)$ or minimize the negative of that:

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

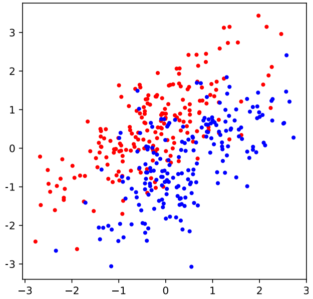
  This loss function if often called *binary cross entropy*.

Toy binary classification problem

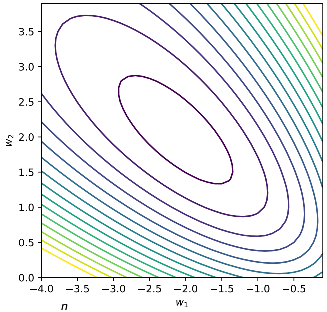
- Consider a toy binary classification problem with two parameters w_1 and w_2 (no bias term):

$$f(\mathbf{x}) = \sigma\left(w_1 x_1 + w_2 x_2\right) \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$

The loss function in this toy example can be visualized using a contour plot.



Training examples



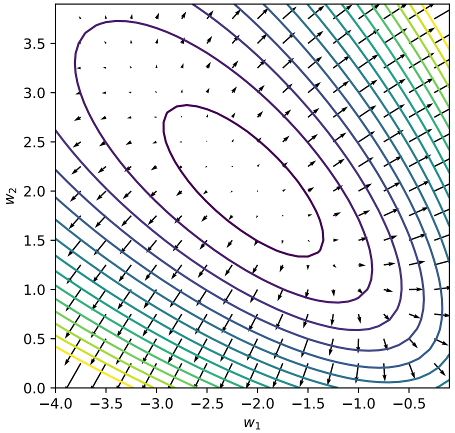
$$\mathcal{L}(w_1, w_2) = -\sum_{i=1}^{n} y^{(i)} \log f(\mathbf{x}^{(i)}) + (1-y^{(i)}) \log(1 - f(\mathbf{x}^{(i)}))$$

Gradient

- Gradient is a vector of partial derivatives:

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

- Gradient points in the direction of the greatest rate of increase of $\mathcal{L}$, its magnitude is the slope of the graph of $\mathcal{L}$ in that direction.



Gradient descent

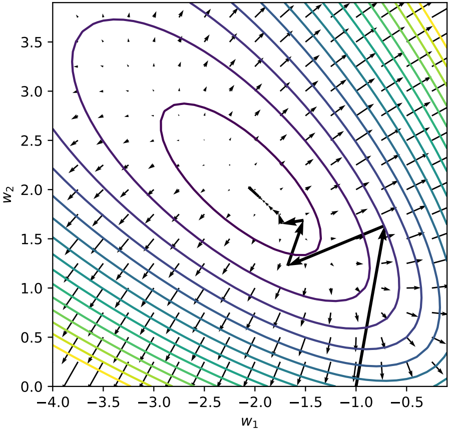
- Gradient descent: update the parameters in the direction opposite to the gradient:

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

with some step size η (also called *learning rate*).

- We reduce the error but do not end up at the minimum, so we need to iterate

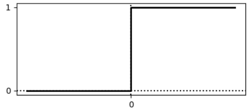
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}(\mathbf{w}_t)$$



A historical note: First models of neurons

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

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



step function

where $\phi(\cdot)$ is a step function.

- There was no training: Parameters w_j, b were set by a human operator to produce correct outputs.
- *Perceptron* (Rosenblatt, 1958) was the first binary classifier which was trained using examples

$$(\mathbf{x}^{(i)}, y^{(i)}) \qquad \mathbf{x}^{(i)} \in \mathbb{R}^m, \quad y^{(i)} \in \{-1, +1\}$$

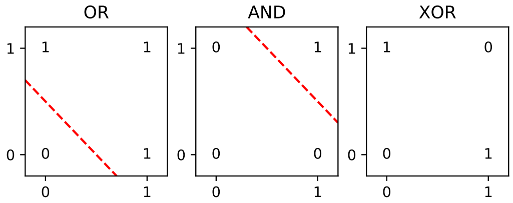
  - ϕ was the sign function.
  - The training procedure was inspired by neuroscience (Donald Hebb's rule). If $\mathbf{x}^{(i)}$ is misclassified, then the weights are updated:

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

    the weights between neurons whose activities are positively correlated are increased.

Linear classifiers cannot solve complex classification problems

- The problem with perceptrons: Since they are linear classifiers, they can solve a very limited set of classification problems.
- They cannot separate linearly inseparable classes, for example, solve the XOR problem:

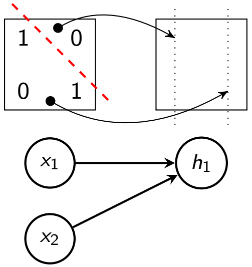


- This problem was emphasized in the influential book "Perceptrons" by Minsky and Papert (1969). They argued that more complex (nonlinear) problems have to be solved with multiple layers of perceptrons (what we now call multilayer neural nets).

Multilayer perceptrons

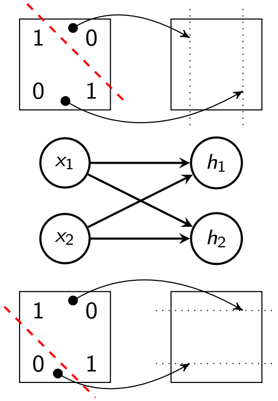
Solving the XOR problem

- The XOR problem can be solved with multiple layers of perceptrons (neurons).
- One neuron can linearly separate the input space as shown on the figure.



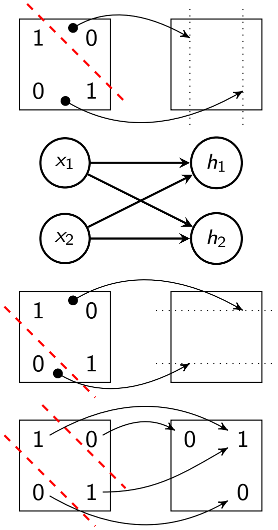
Solving the XOR problem

- The XOR problem can be solved with multiple layers of perceptrons (neurons).
- One neuron can linearly separate the input space as shown on the figure.
- We can add another neuron h_2 which can do another kind of separation of the input space.



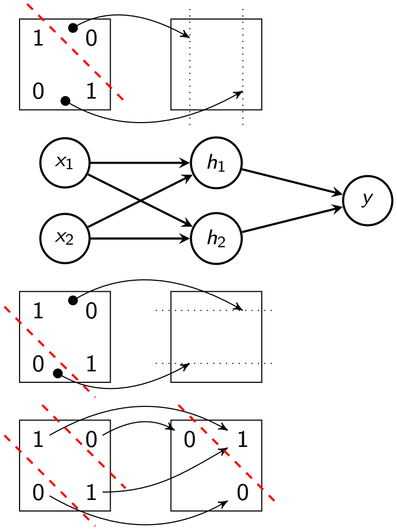
Solving the XOR problem

- The XOR problem can be solved with multiple layers of perceptrons (neurons).
- One neuron can linearly separate the input space as shown on the figure.
- We can add another neuron h_2 which can do another kind of separation of the input space.
- Now we mapped original two-dimensional data into a new two-dimensional space where linear separation is possible.



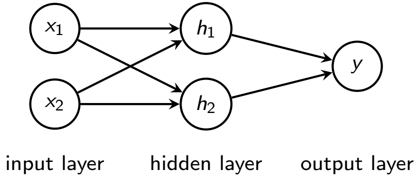
Solving the XOR problem

- The XOR problem can be solved with multiple layers of perceptrons (neurons).
- One neuron can linearly separate the input space as shown on the figure.
- We can add another neuron h_2 which can do another kind of separation of the input space.
- Now we mapped original two-dimensional data into a new two-dimensional space where linear separation is possible.
- Adding another neuron y on top of neurons h_1 and h_2 solves the classification problem.



Multilayer perceptrons

- Now we have a network with two layers of neurons: hidden layer h_1, h_2 and output layer y.
- A neural network with this architecture is called a *multilayer perceptron* (MLP).

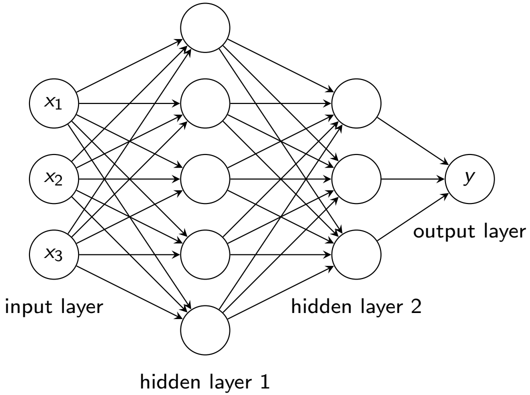


Multilayer perceptrons

- An MLP can of course have more layers and many more neurons.
- Each neuron implements a function

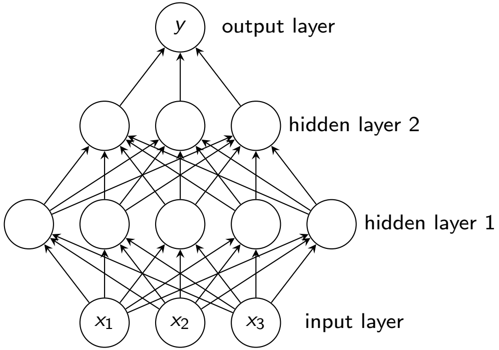
$$y = \phi\left(\sum_{j=1}^{m} w_j x_j + b\right) = \phi\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

  which resembles a simple linear classifier that we considered before.
- The layers in an MLP are called *fully-connected* because each neuron is connected to each neuron in the previous layer.



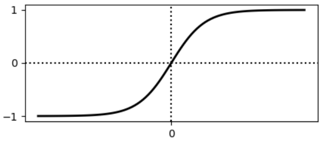
Multilayer perceptrons

- A more compact style: A node in the graph corresponds to an entire layer.

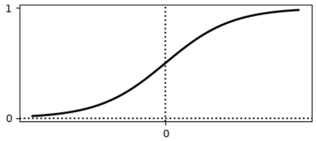


$$\begin{tikzcd}
\boxed{\mathbf{y} = \psi(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)} \\
\boxed{\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)} \arrow[u] \\
\boxed{\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)} \arrow[u] \\
\boxed{\text{input } \mathbf{x}} \arrow[u]
\end{tikzcd}$$

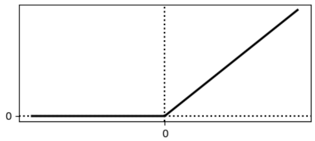
Activation functions



$\tanh(x)$



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



$\mathrm{relu}(x) = \max(0, x)$

- Nonlinearities used after an affine transformation of inputs are often called *activation functions*.
- Nonlinearities used before 2010: $\tanh(x)$ and $\sigma(x) = 1/(1+e^{-x})$.
- Since 2010, $\mathrm{relu}(z) = \max(0, z)$ is very popular.

What if one does not use any nonlinearity?

- What if one does not use any nonlinearity?

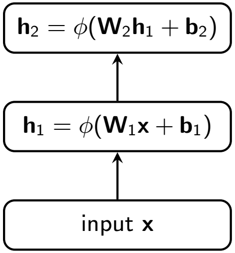
$$\begin{tikzcd}
\boxed{\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)} \\
\boxed{\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)} \arrow[u] \\
\boxed{\text{input } \mathbf{x}} \arrow[u]
\end{tikzcd}$$

What if one does not use any nonlinearity?

- What if one does not use any nonlinearity?
- The identity activation function would lead to:

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

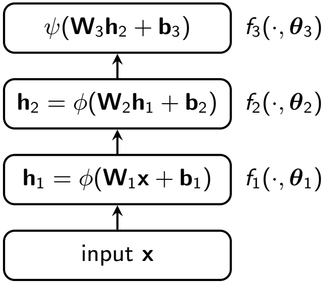
Thus, we get a linear model.


$\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
$\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
input $\mathbf{x}$

Training of multilayer perceptrons

- Our neural network represents a function which is composed of several functions:

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


$\psi(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$
$f_3(\cdot, \boldsymbol{\theta}_3)$
$\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
$f_2(\cdot, \boldsymbol{\theta}_2)$
$\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
$f_1(\cdot, \boldsymbol{\theta}_1)$
input $\mathbf{x}$

- If we solve a binary classification problem, we can use the same loss function that we used before:

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

- Again, we can tune the parameters $\boldsymbol{\theta}_k$ of the classifier by maximizing the log-likelihood, for example, using gradient descent:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_k - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k}.$$

A historical note on multilayer perceptrons

- The idea of using multilayer perceptrons for solving nonlinear classification problems existed already in the 1960s (Minsky and Papert, 1969). However, no one knew how to train multilayer perceptrons. Rosenblatt's learning algorithm did not work for multiple layers.
- How to train MLP networks was well understood only in the mid 80s after an influential paper by Rumelhart, Hinton and Williams (1986). Specifically, they showed how to compute the gradients $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ wrt network parameters 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 after the 1986 paper.

The backpropagation algorithm

Gradient descent for training deep neural networks

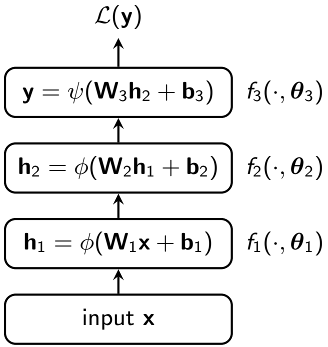
- Our multilayer neural network represents a function which is composed of several functions:

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

- We want to use gradient-descent optimization method to minimize loss function $\mathcal{L}(\boldsymbol{\theta})$:

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

- In order to do that, we need to compute the gradient $\mathbf{g}(\boldsymbol{\theta})$ of the loss function.
    - Parameters $\boldsymbol{\theta}$ include $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, $\mathbf{b}_2$, $\mathbf{W}_3$, $\mathbf{b}_3$.
- Backpropagation: An algorithm to compute the gradient of a loss for a multilayer model.


$\mathcal{L}(\mathbf{y})$
$\mathbf{y} = \psi(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$
$f_3(\cdot, \boldsymbol{\theta}_3)$
$\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
$f_2(\cdot, \boldsymbol{\theta}_2)$
$\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
$f_1(\cdot, \boldsymbol{\theta}_1)$
input $\mathbf{x}$

Chain rule

- The chain rule is a formula to compute the derivative of a composite function:

$$F(x) = f(g(x))$$
$$F'(x) = f'(g(x))g'(x)$$

Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

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

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial \theta}$$

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

$$\begin{tikzcd}
 & w \arrow[d] & \theta \arrow[d] & \\
x \arrow[r] & f_1 \arrow[r, "h"] & f_2 \arrow[r, "y"] & \mathcal{L}
\end{tikzcd}$$

Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

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

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

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

$$\begin{tikzcd}
 & w \arrow[d] & \theta \arrow[d] & \\
x \arrow[r] & \boxed{f_1} \arrow[r, "h"] & \boxed{f_2} \arrow[r, "y"] & \boxed{\mathcal{L}} \arrow[l, bend left, red, "\frac{\partial \mathcal{L}}{\partial y}"]
\end{tikzcd}$$

- We can compute the derivatives efficiently by storing intermediate results.

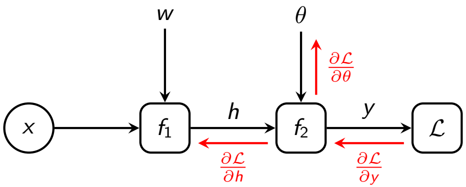
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

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

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

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



$$\begin{tikzcd}
 & w \arrow[d] & \theta \arrow[d, shift right] & \\
x \arrow[r] & f_1 \arrow[r, shift left, "h"] & f_2 \arrow[l, shift left, "\frac{\partial \mathcal{L}}{\partial h}"] \arrow[r, shift left, "y"] \arrow[u, shift right, "\frac{\partial \mathcal{L}}{\partial \theta}"'] & \mathcal{L} \arrow[l, shift left, "\frac{\partial \mathcal{L}}{\partial y}"]
\end{tikzcd}$$

- We can compute the derivatives efficiently by storing intermediate results.

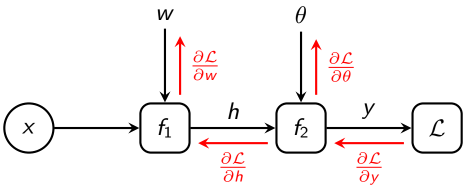
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

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

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

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


w
θ
$\frac{\partial \mathcal{L}}{\partial w}$
$\frac{\partial \mathcal{L}}{\partial \theta}$
x
f_1
h
f_2
y
$\mathcal{L}$
$\frac{\partial \mathcal{L}}{\partial h}$
$\frac{\partial \mathcal{L}}{\partial y}$

- We can compute the derivatives efficiently by storing intermediate results.

Chain rule for multi-variable functions

- For multi-variable functions, the chain rule can be written in terms of Jacobian matrices.

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

$$\text{Jacobian matrix: } \mathbf{J}_{f \circ g} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}$$

- The chain rule is:

$$\mathbf{J}_{f \circ g}(\mathbf{x}) = \mathbf{J}_f(\mathbf{u})\mathbf{J}_g(\mathbf{x})$$

or each element of the Jacobian is:

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

Backpropagation for multi-variable functions

- Consider a multi-layer model:

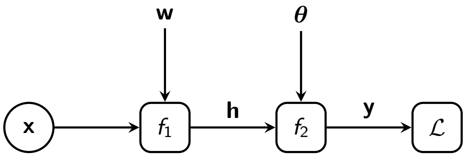
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \qquad \mathbf{y} \in \mathbb{R}^K,\ \mathbf{h} \in \mathbb{R}^L,\ \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

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

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

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


$$\begin{tikzcd} & \mathbf{w} \arrow[d] & \boldsymbol{\theta} \arrow[d] & \\ \mathbf{x} \arrow[r] & f_1 \arrow[r, "\mathbf{h}"] & f_2 \arrow[r, "\mathbf{y}"] & \mathcal{L} \end{tikzcd}$$

Backpropagation for multi-variable functions

- Consider a multi-layer model:

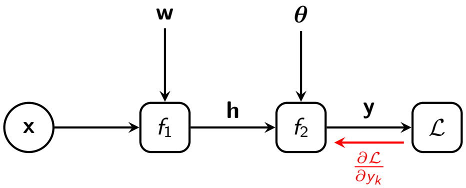
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \qquad \mathbf{y} \in \mathbb{R}^K, \ \mathbf{h} \in \mathbb{R}^L, \ \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

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

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

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



- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Backpropagation for multi-variable functions

- Consider a multi-layer model:

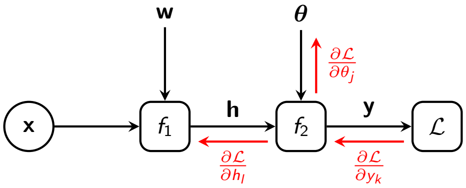
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \qquad \mathbf{y} \in \mathbb{R}^K, \ \mathbf{h} \in \mathbb{R}^L, \ \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

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

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

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


$$\begin{tikzcd} & \mathbf{w} \arrow[d] & & \boldsymbol{\theta} \arrow[d] & \\ \mathbf{x} \arrow[r] & f_1 \arrow[rr, "\mathbf{h}"] & & f_2 \arrow[rr, "\mathbf{y}"] \arrow[u, bend right, "\frac{\partial \mathcal{L}}{\partial \theta_j}"'] \arrow[ll, bend left, "\frac{\partial \mathcal{L}}{\partial h_l}"] & & \mathcal{L} \arrow[ll, bend left, "\frac{\partial \mathcal{L}}{\partial y_k}"] \end{tikzcd}$$

- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Backpropagation for multi-variable functions

- Consider a multi-layer model:

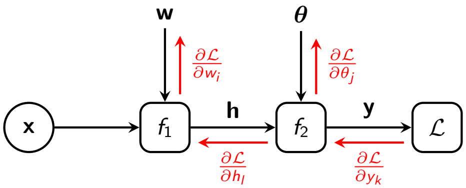
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \qquad \mathbf{y} \in \mathbb{R}^K,\ \mathbf{h} \in \mathbb{R}^L,\ \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

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

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

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



$$\begin{tikzcd} & \mathbf{w} \arrow[d, shift left] & & \boldsymbol{\theta} \arrow[d, shift left] & \\ \mathbf{x} \arrow[r] & f_1 \arrow[r, "\mathbf{h}"] \arrow[u, shift right, "\frac{\partial \mathcal{L}}{\partial w_i}"'] & f_2 \arrow[r, "\mathbf{y}"] \arrow[l, shift left, "\frac{\partial \mathcal{L}}{\partial h_l}"] \arrow[u, shift right, "\frac{\partial \mathcal{L}}{\partial \theta_j}"'] & \mathcal{L} \arrow[l, shift left, "\frac{\partial \mathcal{L}}{\partial y_k}"] \end{tikzcd}$$

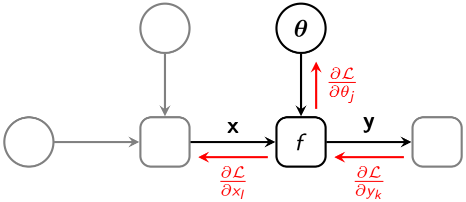
- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Implementing backpropagation in software

- For each block of a neural network, we need to implement the following computations:
  - 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: $\frac{\partial \mathcal{L}}{\partial x_l}$, $\frac{\partial \mathcal{L}}{\partial \theta_j}$

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

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



$\boldsymbol{\theta}$
$\frac{\partial \mathcal{L}}{\partial \theta_j}$
$\mathbf{x}$
f
$\mathbf{y}$
$\frac{\partial \mathcal{L}}{\partial x_l}$
$\frac{\partial \mathcal{L}}{\partial y_k}$

- We will practice implementing forward and backward computations in the first assignment.

PyTorch

- PyTorch is a programming framework which allows you to create complex multilayer models without the need to implement the optimization procedure. Backpropagation is already implemented in the framework.

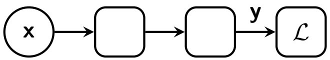
```
import torch
mlp = nn.Sequential(
    nn.Linear(3, 10),
    nn.ReLU(),
    nn.Linear(10, 1),
)
optimizer = torch.optim.SGD(mlp.parameters(), lr=0.01)

for i in range(100):
    optimizer.zero_grad()

    # Compute loss
    y = mlp(x)
    loss = loss_fn(y, targets)

    # Compute gradient by backpropagation
    loss.backward()

    # Update model parameters
    optimizer.step()
```

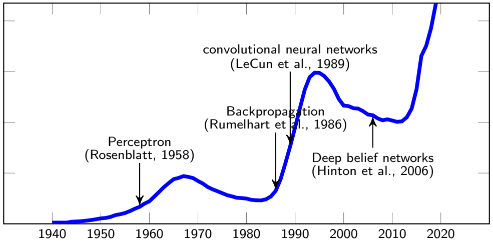


Deep vs shallow networks

Deep vs shallow networks

- Modern neural networks have many layers. For example, convolutional neural network for computer vision tasks can have more than 100 layers.

- During the second wave, the neural networks were not very deep, only with two-three hidden layers.
    - Deeper networks did not provide better performance.
    - There were no theoretical results that deep networks have better representational power.

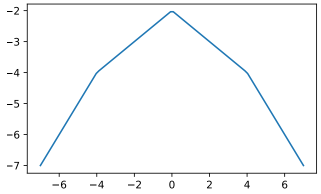


Frequency of phrases "cybernetics", "neural networks" and "deep learning" according to Google books.

- Universal approximation theorem (Cybenko, 1989): 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.

Deep networks have greater representational power

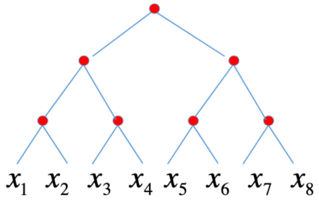
- A multilayer perceptron with relu nonlinearity implements a piece-wise linear function.
- Suppose that we have an MLP with n inputs and relu nonlinearities.
    - If the network contains a single layer model with Lm hidden units, then the number of regions behaves as $O(L^n m^n)$ (Pascanu et al., 2013).
    - If the network contains L hidden layers of width $m \geq n$, the model can compute functions that have $\Omega((m/n)^{(L-1)n} m^n)$ linear regions (Montúfar et al., 2014).



- This result suggests that the number of linear regions grows much faster in a deep neural network compared to a shallow one.

Deep networks have greater representational power

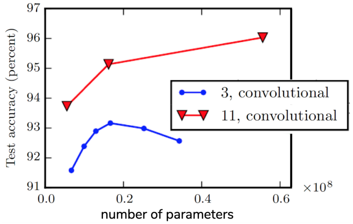
- Theory (see, e.g., Poggio et al., 2019): Both shallow and deep networks can approximate arbitrarily well any continuous function $\mathbb{R}^n \to \mathbb{R}$ on a compact domain with the expense of an exponential (wrt n) number of parameters.
- However, if the approximated function f is a hierarchical composition of local functions, then deep networks of the convolutional type can have a linear dependence on n unlike shallow networks which have exponential dependence.



assumed compositional structure of f

Deeper models seem to perform better

- Practice: increasing the number of parameters in layers without increasing their depth is not nearly as effective at increasing test set performance:
- Example: Multi-digit number recognition from Street View imagery using deep convolutional neural networks (Goodfellow et al., 2014)

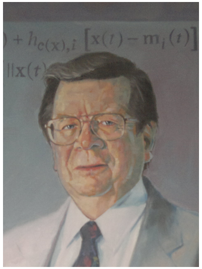


- Shallow models overfit at around 20 million parameters while deep ones can benefit from having over 60 million parameters.

- Deep models have an inductive bias that a modeled function should consist of many simpler functions composed together. This assumption turns out to work very well.

Finland has been strong in neural networks research

Finland has been strong in neural networks research



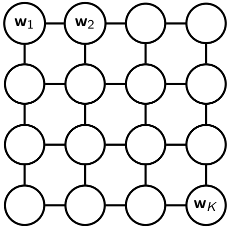
Self-organizing maps
(Kohonen, 1981)

Neuron principal component analysis (Oja, 1982)
Bottleneck autoencoder (Oja, 1991)

- Hyvärinen and Oja (1997): Fast algorithms for independent component analysis (FastICA)
- Valpola and Honkela (2000): Predecessor model of variational autoencoders
- Kyunghyun Cho (GRU, NMT) was a Macadamia student, did his PhD in deep learning in Aalto.

Self-organizing map (SOM) (Kohonen, 1981)

- Unsupervised learning method that can be used, for example, for data visualization.
- Data samples $\mathbf{x}^{(i)}$ are mapped to a grid of neurons $\mathbf{w}_k$ arranged in a 2D square or hexagonal grid.



SOM square grid

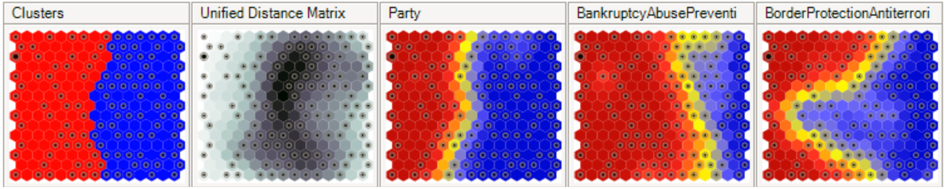
- Training procedure:
  - Take a training sample $\mathbf{x}^{(i)}$ and select the neuron whose weight vector $\mathbf{w}_k$ has the shortest Euclidean distance to $\mathbf{x}^{(i)}$.
  - The weight vectors of the winning neuron and the neurons in its neighborhood are updated:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j + \eta(j,k)\left(\mathbf{x}^{(i)} - \mathbf{w}_j\right)$$

  where $\eta(j,k)$ is the neighborhood function which depends on the distance on the grid.

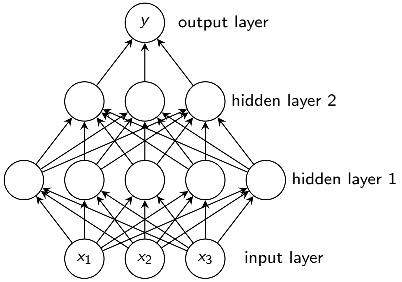
Self-organizing map (SOM)

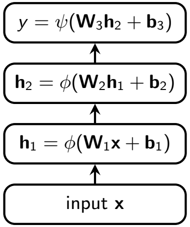
- SOM can be used as a data visualization tool.
- Data: $\mathbf{x}^{(i)}$ is a collection of votes by one member of Congress, each vote is yes/no/abstain.



Home assignments

Assignment `01_mlp` (deadline 09.03 Wed 23:00)




$y = \psi(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$
$\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
$\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
input $\mathbf{x}$

1. Implement the backpropagation algorithm and train a multilayer perceptron (MLP) in numpy.
2. Implement and train a multilayer perceptron in PyTorch.

Recommended reading

- Sections 1, 6.1–6.4 of the deep learning book.
- A. Kurenkov. A 'Brief' History of Neural Nets and Deep Learning.