Technische Hochschule Rosenheim

Chapter 05
Artificial Neural Networks

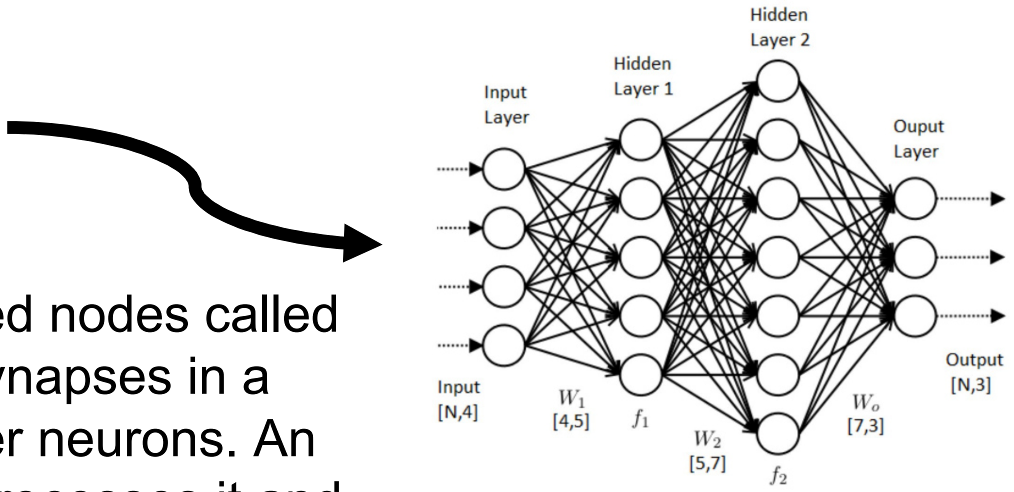
Lecture A2I2

Kai Höfig & Markus Breunig

Introduction

- Artificial Neural Networks (ANNs) are computing systems vaguely inspired by the biological neural networks that constitute animal brains.
- They "learn" to perform tasks by considering examples and are primarily used for classification and regression problems.

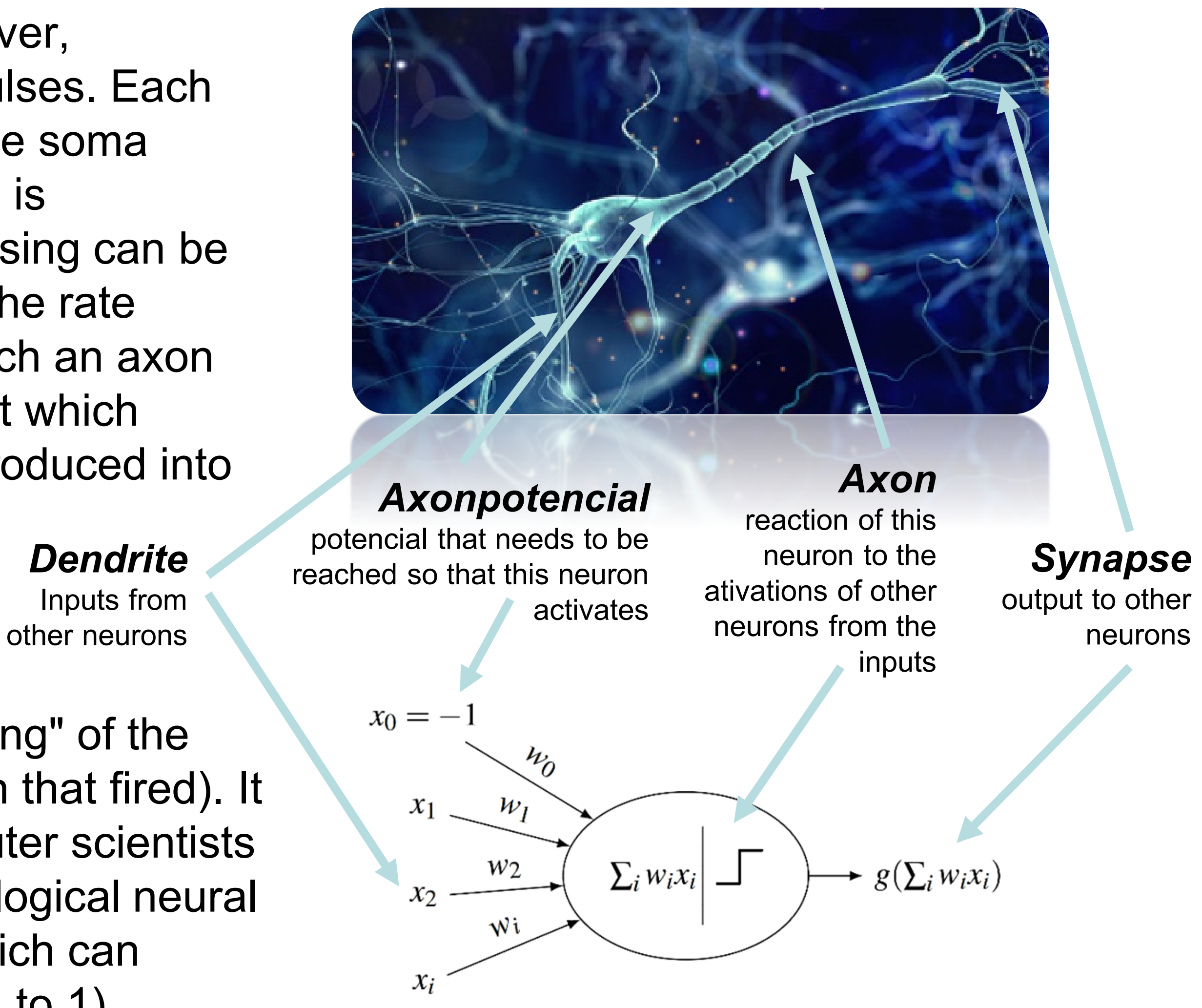
  For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images (supervised learning / classification). They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process



- An ANN is based on a collection of connected nodes called artificial neuron. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it.

Biological Neurons and their Analogy

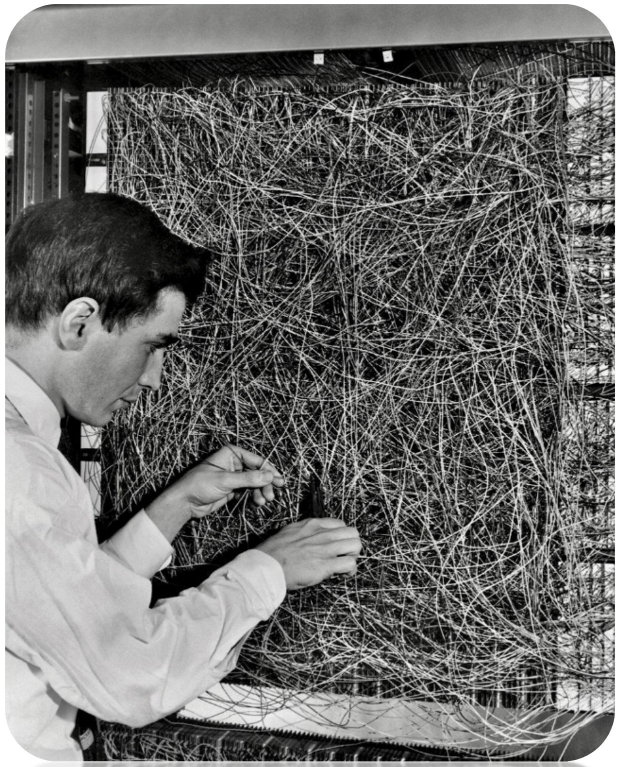
- Unlike most artificial neurons, however, biological neurons fire in discrete pulses. Each time the electrical potential inside the soma reaches a certain threshold, a pulse is transmitted down the axon. This pulsing can be translated into continuous values. The rate (activations per second, etc.) at which an axon fires converts directly into the rate at which neighboring cells get signal ions introduced into them. The faster a biological neuron fires, the faster nearby neurons accumulate electrical potential (or lose electrical potential, depending on the "weighting" of the dendrite that connects to the neuron that fired). It is this conversion that allows computer scientists and mathematicians to simulate biological neural networks using artificial neurons which can output distinct values (often from −1 to 1).



Axonpotencial potencial that needs to be reached so that this neuron activates
Axon reaction of this neuron to the ativations of other neurons from the inputs
Dendrite Inputs from other neurons
Synapse output to other neurons
$x_0 = -1$
w_0
x_1
w_1
x_2
w_2
x_i
w_i
$\sum_i w_i x_i$
$g(\sum_i w_i x_i)$

History of ANNs – When it all started

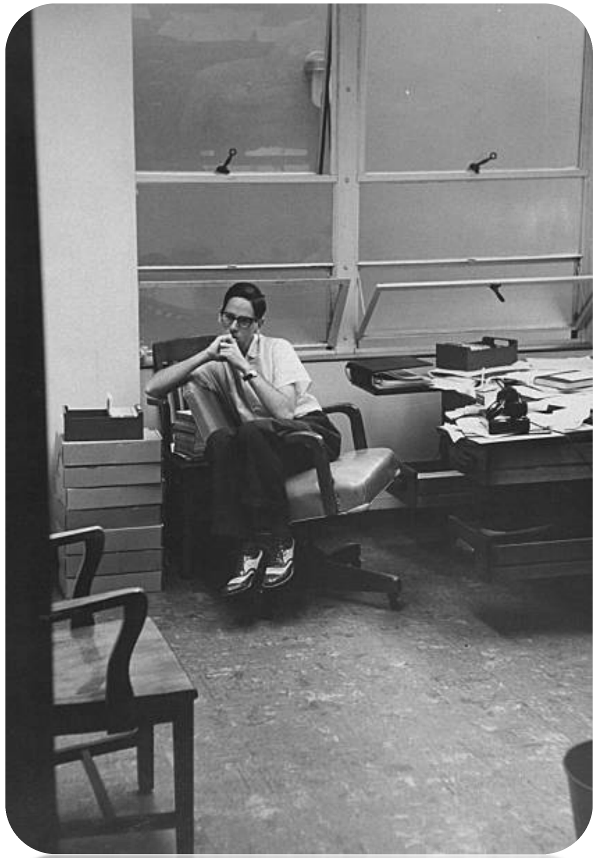
- Warren McCulloch and Walter Pitts (**1943**) opened the subject by creating a computational model for neural networks. Rosenblatt (**1958**) created the **perceptron**. The first functional networks with many layers were published by Ivakhnenko and Lapa in **1965**. The basics of continuous backpropagation (the primary means of training an ANN) were derived in the context of control theory by Kelley in **1960** and by Bryson in **1961**.



20 years from theory to training, then 1st AI Winter

https://en.wikipedia.org/wiki/Artificial_neural_network

History of ANNs – The Renaissance



In 1973, **Dreyfus** used backpropagation to adapt parameters of controllers in proportion to **error gradients**. Werbos's (1975) **backpropagation** algorithm enabled practical training of multi-layer networks. In 1982, he applied Linnainmaa's AD method to neural networks in the way that became widely used. Thereafter research stagnated following Minsky and Papert (1969), who discovered that basic perceptrons were incapable of **processing the exclusive-or** circuit and that computers lacked sufficient power to process useful neural networks.

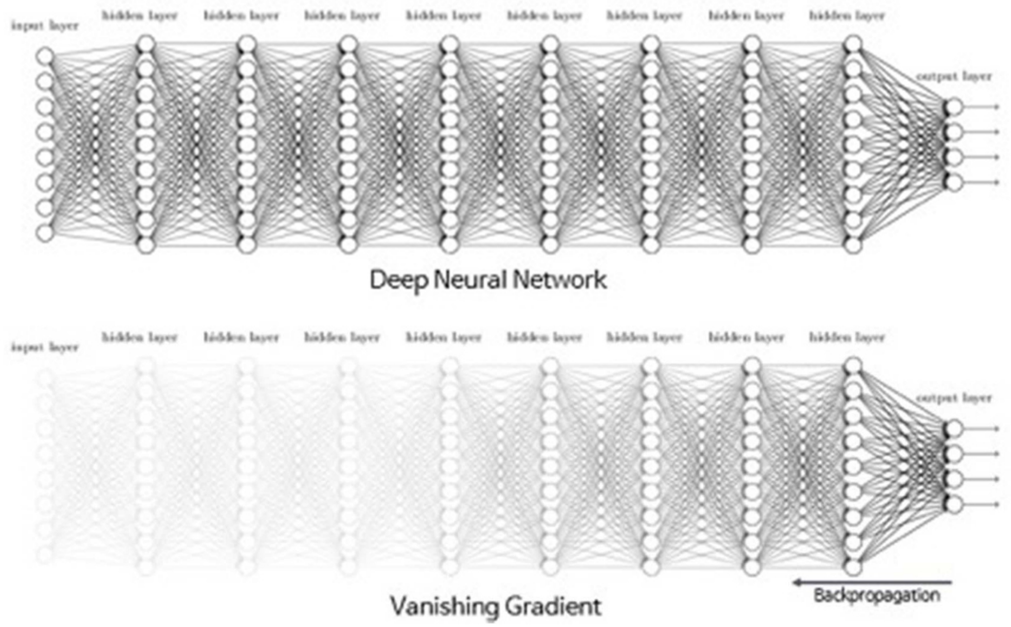
It took 10 years to apply training practically,
but computational power was too low to solve real problems: 2nd AI winter

https://en.wikipedia.org/wiki/Artificial_neural_network

History of ANNs – Modern times

- Ciresan and colleagues (2010) showed that despite the **vanishing gradient** problem, GPUs make backpropagation feasible for many-layered feedforward neural networks. Between 2009 and 2012, ANNs began winning prizes in ANN contests, approaching human level performance on various tasks, initially in pattern recognition and machine learning. For example, the bi-directional and multi-dimensional long short-term memory (LSTM) won three competitions in connected handwriting recognition in 2009 without any prior knowledge about the three languages to be learned.

- Ciresan and colleagues built the first pattern recognizers to achieve human-competitive/superhuman performance on benchmarks such as traffic sign recognition (IJCNN 2012).



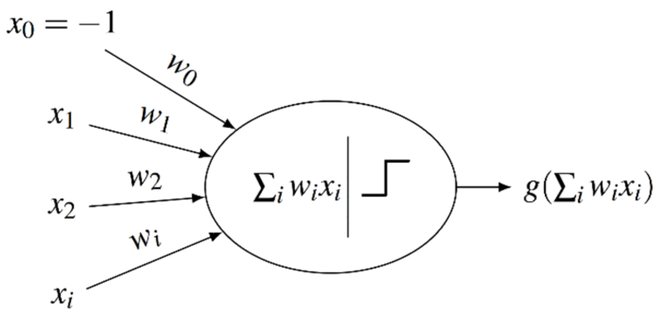
https://en.wikipedia.org/wiki/Artificial_neural_network

Goals

- In this slideset you will learn
- *A little bit about the history and why a technology that is nearly 70 years old is recently so popular.*
- *How to calculate and interpret the result of an ANN for a given sample*
- *How to train a network using backpropagation*
- *How to use ANNs practically*

Basic Structure of a Single Artificial Neuron

- A neuron has inputs $x_1, \ldots, x_n$ and, after being trained, fixed weights $w_1, \ldots, w_n$. The input x_0 and the weight w_0 are called the bias, and are also fix after training.



- The net input of the nuron is calculated using the weighted sum of the inputs

$$net = \sum_{i=0}^{n} w_i x_i$$

- The output of the neuron is calculated using an activation function *g* of the net input

$$out = g(net)$$

- So the output of a neuron is

$$y = g\left(\sum_{i=0}^{m} w_i x_i\right)$$

The AND Function as an Artificial Neuron

- With a single artificial neuron, we can implement the Boolean AND function.

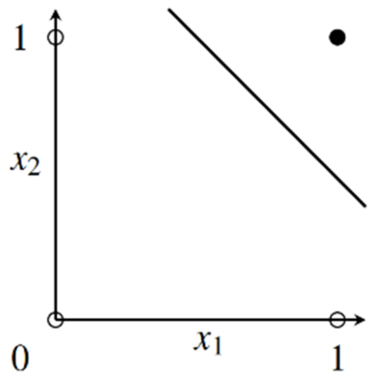
$$\begin{aligned} w_0 &= 1.5 \\ w_1 &= 1 \\ w_2 &= 1 \\ y &= g(w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot w_3) \end{aligned}$$

$$g(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} .$$

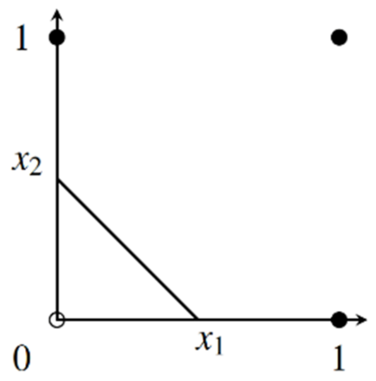
x1	x2	x1 and x2	y	g(y)
0	0	0	-1.5	0
1	0	0	-0.5	0
0	1	0	-0.5	0
1	1	1	2.5	1

Other Functions

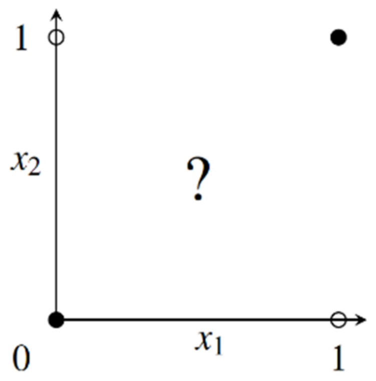
- Other functions can be similarly implements as long as they are linearly separable. The weights describe nothing more than a hyperplane (in 2D, as in the example, a straight line) with which the separation between the desired output values zero or one is achieved. So the Boolean OR can easily be implement, but not the Boolean XOR.



(a) x_1 AND x_2



(b) x_1 OR x_2



(c) x_1 XOR x_2

- Learning or training now means that the weights are automatically determined from an annotated sample.

Exercise

- a) What does an artificial neuron look like to implement the Boolean OR function?
- b) What does an artificial neuron look like to implement the Boolean NOT function?

Exercise

- a) What does an artificial neuron look like to implement the Boolean OR function?

  This neuron looks very similar to the neuron for AND, it has the same inputs, only the weights are different:

  $w0$ is any value between 0 and 1, $w1=1$, $w2=1$

- b) What does an artificial neuron look like to implement the Boolean NOT function?

  This neuron only has one real input $x1$ (which can be 0 or 1) in addition to the bias $x0=-1$. $w0$ than has to be smaller than 0. $w1$ can in turn be any number smaller than $w0$. Once again, there are infinitely many weights implementing this function.

Example Neuron and 1st Prediction

- We use the workload and participation in the lecture of students to classify them into *good* and *bad* (or to predict wether they pass the exam or not)

Student	Workload	Participation	Pass exam
1	3h/week	1	1
2	1h/week	0	0

- We use the following neuron

$$\begin{aligned} net(x_1, x_2) &= w_0 x_0 + w_1 x_1 + w_2 x_2 \\ &= -1 + 0.5 \cdot x_1 + 0.5 \cdot x_2 \end{aligned}$$

$$out(x_1, x_2) = \frac{1}{1 + e^{-net}}$$

- So it is

$$\begin{aligned} net(3,1) &= -1 + 0.5 \cdot 3 + 0.5 \cdot 1 = 1 \\ net(1,0) &= -1 + 0.5 \cdot 1 + 0.5 \cdot 0 = -0.5 \end{aligned}$$

$$\begin{aligned} out(3,1) &= \frac{1}{1 + e^{-1}} \approx 0.731 \\ out(1,0) &= \frac{1}{1 + e^{0.5}} \approx 0.378 \end{aligned}$$

Discussion

- Is that a good model?
- How good is the model?
- How can we interpret the prediction?
- How can we improve the model?

- **Task 1**: improve the model yourself and find better parameter.

An Optimization Problem

- What we did now manually, is an optimization problem. We punish bad predictions by using squared errors, so that bad predictions get punished really hard.

$$\begin{aligned} error &= (target(3,1) - out(3,1))^2 + (target(1,0) - out(1,0))^2 \\ &= (1 - 0.731)^2 + (0 - 0.378)^2 \\ &\approx 0.215 \end{aligned}$$

- So what we want to do, is finding w_0, w_1, w_2 so that this equation becomes minimal:

$$\begin{aligned} error(w_0, w_1, w_2) &= (1 - \frac{1}{1 + e^{-(w_0 + w_1 \cdot 3 + w_2 \cdot 1)}})^2 + \\ & (1 - \frac{1}{1 + e^{-(w_0 + w_1 \cdot 1 + w_2 \cdot 0)}})^2 \end{aligned}$$

Discussion

- How will that change for
  - A) more training data?
  - B) more neurons (with more weights)?

$$
\begin{aligned}
error(w_0, w_1, w_2) \quad = \quad & (1 - \frac{1}{1 + e^{-(w_0 + w_1 \cdot 3 + w_2 \cdot 1)}})^2 + \\
& (1 - \frac{1}{1 + e^{-(w_0 + w_1 \cdot 1 + w_2 \cdot 0)}})^2
\end{aligned}
$$

- **Task 2**: Improve the model based on what the computer shows us as the function to be optimized.

Gradient Descent for Automatic Optimization

- The method of gradient descend optimizes the parameters according to their effect to the total error. We can calculate that by differentiate the error function with respect to the parameter we want to optimize:

$$\frac{\partial error}{\partial w_i}$$

- Since the calculation becomes much easier, we use the *chain rule*

$$\frac{\partial error}{\partial w_i} = \frac{\partial error}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial w_i}$$

- Once we know, we can optimize the parameter by

$$w_i^+ = w_i - \eta \cdot \frac{\partial E}{\partial w_i}$$

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Example

$$
\begin{aligned}
\frac{\partial error}{\partial w_1} &= \frac{\partial error}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial w_1} \\
error &= \frac{1}{2}(1 - out(3,1))^2 \\
\frac{\partial error}{\partial out} &= \frac{1}{2} \cdot 2(1 - out(3,1)) \cdot -1 \approx -0.2689 \\
out &= \frac{1}{1 - e^{-net}} \\
\frac{\partial out}{\partial net} &= out(3,2) \cdot (1 - out(3,2)) \\
net &= w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 \\
\frac{\partial net}{\partial w_1} &= x_1 = 3 \\
\frac{\partial error}{\partial w_1} &= \approx -0.2689 \cdot 0.1966 \cdot 3 \approx -0.1586 \\
w_1^+ &= w_1 \overset{-}{\cancel{+}} \eta \cdot \frac{\partial error}{\partial w_1} \approx 0.5793
\end{aligned}
$$

We train the dataset where the student put three hours of effort (x1=3) and participated in the lecture (x2=1). We know the result should be equal or close to 1

Using the logistic function was clever, since we need to differentiate.

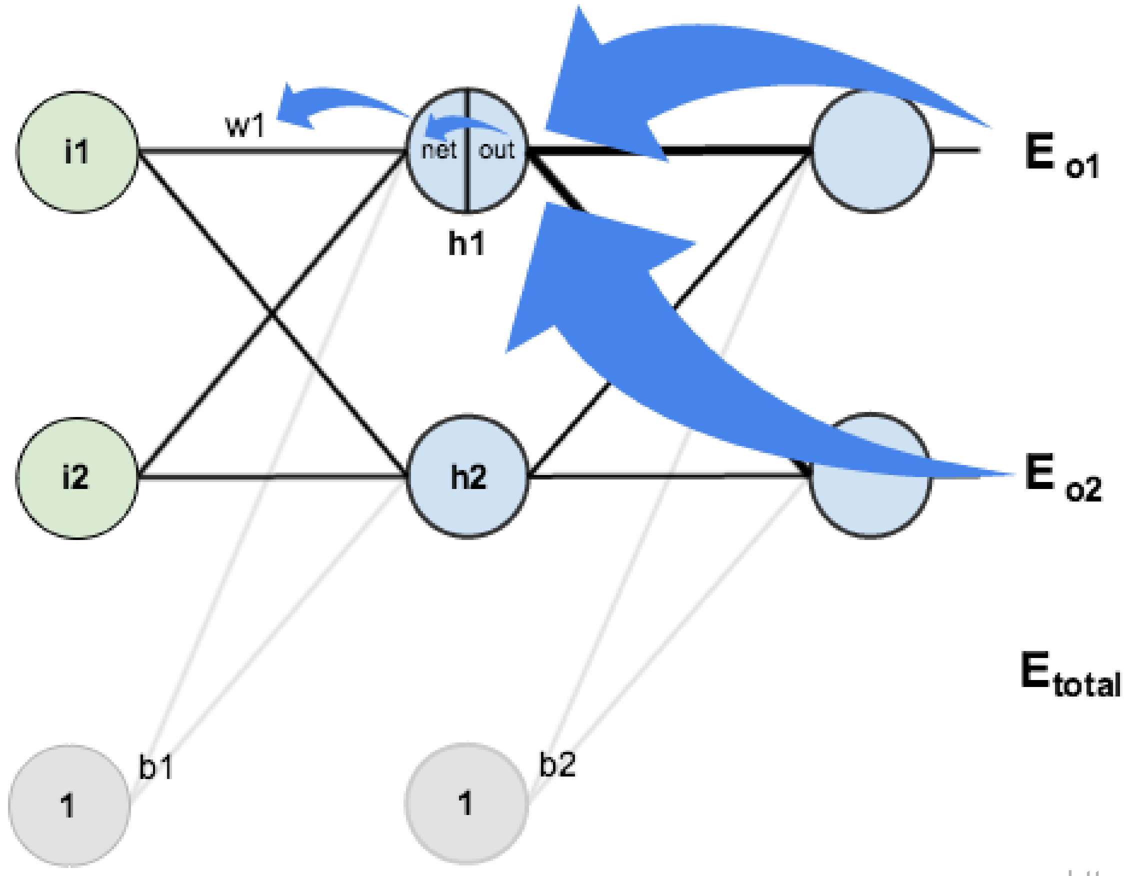
Changing w1 to 0.5793 optimizes the global error function by 0.0072 from 0.2148 to 0.2076

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Gradient Descent for More Layer – Backpropagation

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



After optimizing the output layer, we can continue this process backwards (backpropagation algorithm) towards the inputs of the network. The step is slightly different, since changing a weight in other layer than the output layer can affect multiple output neurons.

$E_{total} = E_{o1} + E_{o2}$

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

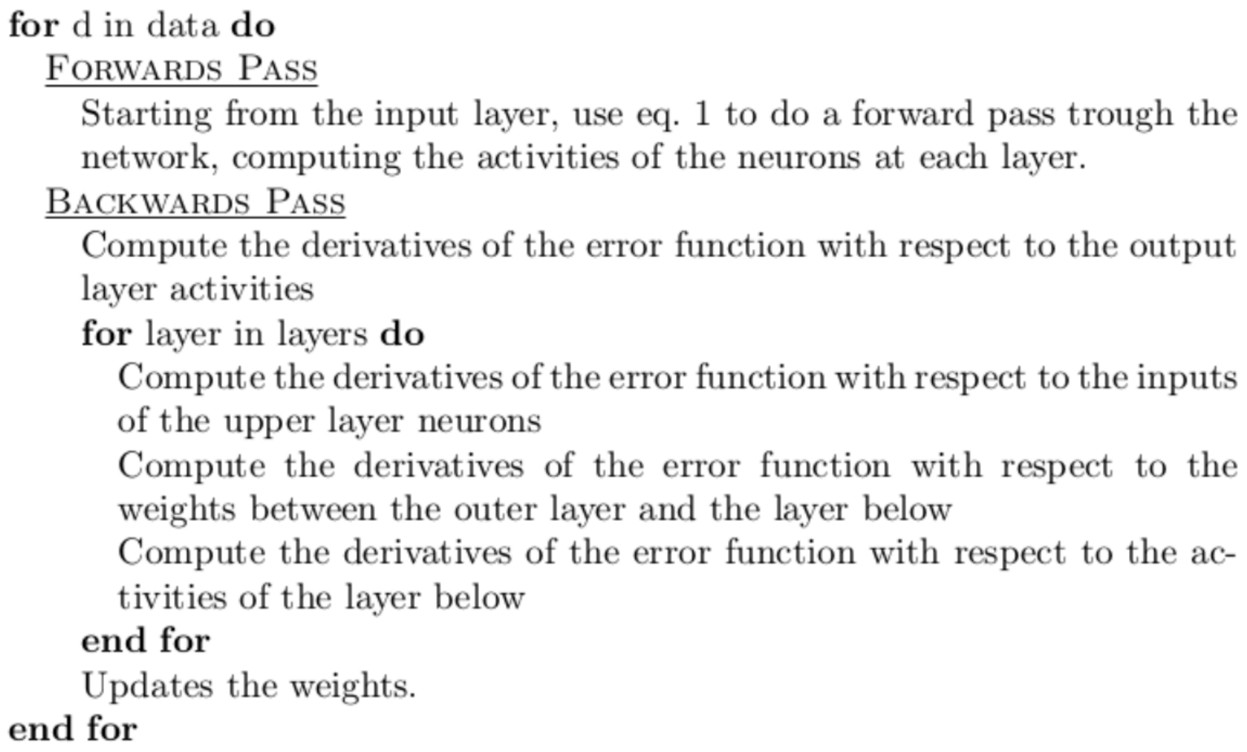
Discussion

- What happens, when we repeat the process from the example over and over again?
- What could we do to optimize this algorithm?
- **Task 3:** Optimize the parameter w2 for the network. How did this influence the error function?
  - Don‘t do it, but wouldnt it be more precise to recalculate the error function with the already optimized paramater for w1?

Backpropagation Algorithm

An **epoch** is when the entire dataset was trained once. The algorithm can be repeated multiple times and trains the network in multiple epochs to optimize the error function

A **batch size** is the number of training examples in one forward/backward pass. If you have 10 examples and you batch size is 5, you need 2 **iterations** to train one epoch.



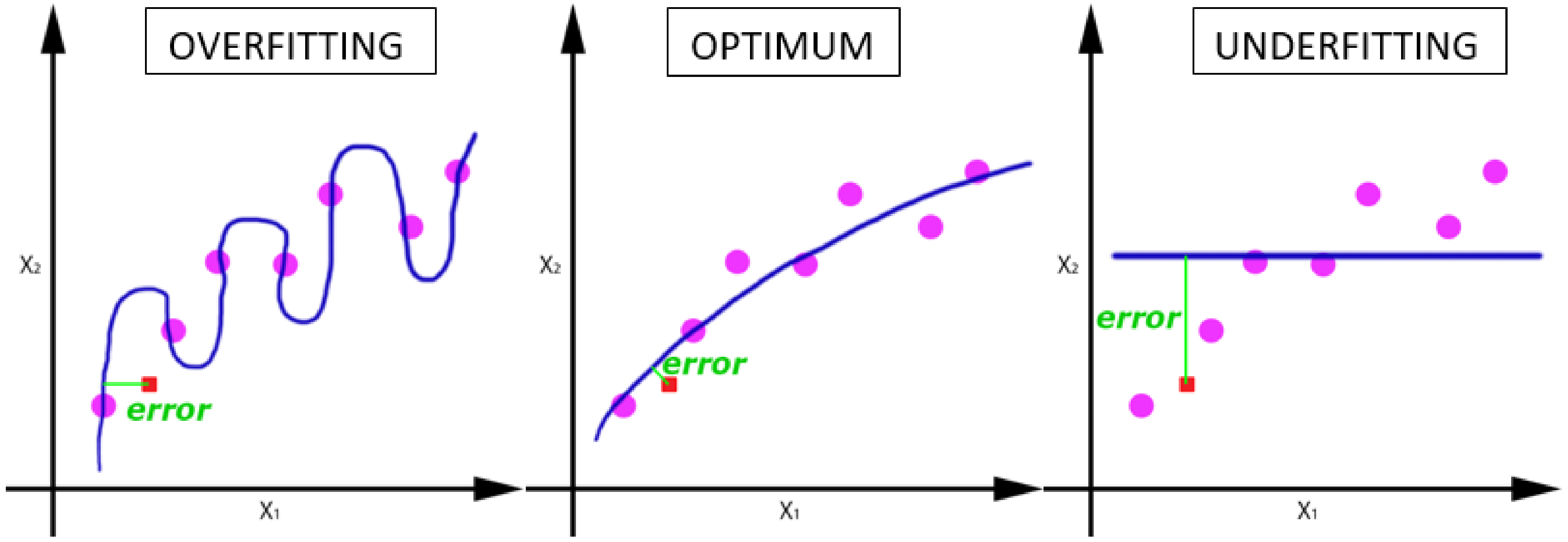
```
for d in data do
  FORWARDS PASS
    Starting from the input layer, use eq. 1 to do a forward pass trough the
    network, computing the activities of the neurons at each layer.
  BACKWARDS PASS
    Compute the derivatives of the error function with respect to the output
    layer activities
    for layer in layers do
      Compute the derivatives of the error function with respect to the inputs
      of the upper layer neurons
      Compute the derivatives of the error function with respect to the
      weights between the outer layer and the layer below
      Compute the derivatives of the error function with respect to the ac-
      tivities of the layer below
    end for
    Updates the weights.
end for
```

So for one batch, the optimized weights are calculated. After the batch is done, the optimized weights are updated in the network and a new batch is started using the updated weights. Otherwise, the error function had to be recalculated after each sample (batch size 1).

https://medium.com/@jorgesleonel/backpropagation-cc81e9c772fd

Over- and Underfitting

- Training too much results in optimizing the error function to 0. This process is called overfitting. When training too little, this results in underfitting. To find out if we over or underfitted our network, we need test data that was not used for training. This can be done by splitting the data into training, validation and test data, e.g. by 80% / 10% / 10% split.
- Training data is used for optimization, validation is used during optimization as a stop criteria and test data is used after training to measure quality.



More Advanced Cost Functions

- Since the squared error function punishes wrong predictions hard, (well at least squared), but close predictions comparatively soft, often cross entropy is used, where y_j is the desired output of neuron j and d_j is the actual output

$$\epsilon(h) = -\sum_j y_j(h) \cdot ld(d_j(h))$$

- Example

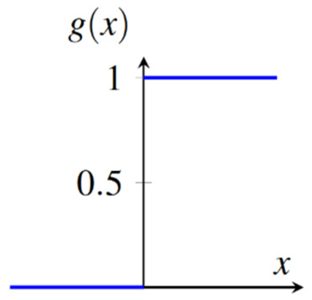
$$\begin{aligned}(1-1)^2 &= 0 \\ (1-0.99)^2 &= 0.0001 \\ (1-0.7)^2 &= 0.09 \\ (1-0.1)^2 &= 0.81\end{aligned} \qquad \begin{aligned}-(1 \cdot ld(1)) &= 0 \\ -(1 \cdot ld(0.99)) &\approx 0.0144 \\ -(1 \cdot ld(0.7)) &\approx 0.5145 \\ -(1 \cdot ld(0.1)) &\approx 3.3219\end{aligned}$$

Modern Optimizers

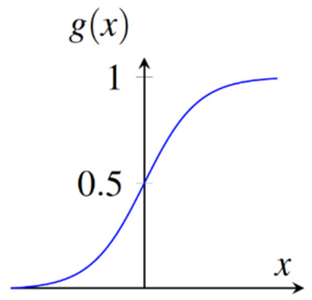
- Stochastc gradient descent (SGD) optimization is still a simple way of optimization.
- More advanced optimizers
  - **Adapt the learning rate during the training**
    SGD has a fixed learning rate. Since the lerning rate is the pace we use to run down the hill towards optimum, it makes sens to run slower the closer we get.
  - **Use the momentum from the previous gradient**
    Instead of making a step into one direction, we make a step slightly in the new direction but also let us guide from the previous step. In this way, the optimization does not tumble don the hill, but is a bit more straight.
- One of this more advanced optimiziers is **RMSprop optimizer**
  (but there are many)

Activation Functions

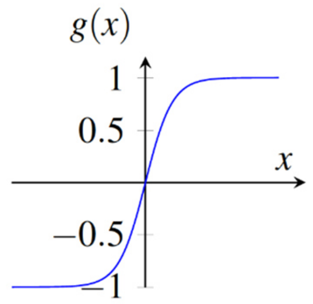
- The activation function of a neuron is chosen to have a number of properties which either enhance or simplify the network containing the neuron. Crucially, a non-linear function has be used, if we want to be able to implement non-linearly-seperable functions be combining multiple neurons. Let us look at the most widely used activation functions:



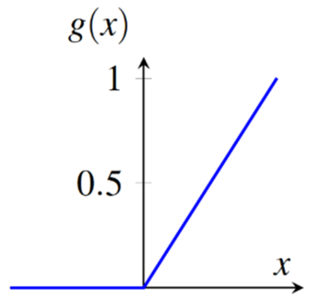
(a) Sprungfunktion



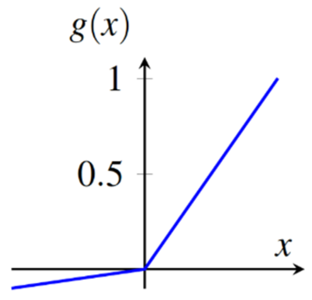
(b) Sigmoid



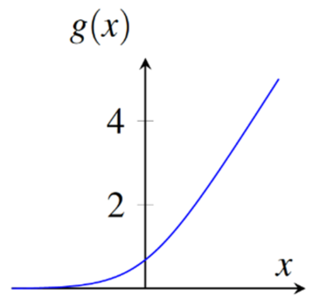
(c) tanh



(d) ReLU



(e) Leaky ReLU



(f) Softplus

Step function (Sprungfunktion)

- The output y of this transfer function is binary, depending on whether the input meets a specified threshold θ . The "signal" is sent, i.e. the output is set to one, if the activation meets the threshold.

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{if } u < \theta \end{cases}$$

- This function is used in perceptrons and often shows up in many other models. It performs a division of the space of inputs by a hyperplane. It is specially useful in the last layer of a network intended to perform binary classification of the inputs. It can be approximated from other sigmoidal functions by assigning large values to the weights. The step function has one big drawback, though: it is non-continous and therefor non-differentiable at the threshold value. We will see later on that the first derivative of the activation function is very importent when training a network of neurons. Therefore, continous and differentiable approximations of the step function have been used.

Sigmoid function and tangens hyperbolicus

- For both the sigmoid function gs(x) and the tangens hyperbolicus gt(x), the first derivate can be calculated easily:

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

$$g'_s(x) = g_s(x)(1 - g_s(x))$$

and

$$g_t(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$g'_t(x) = 1 - g_t^2(x)$$

ReLU, leaky ReLU and Softplus

- These are the currently most widely used activations functions. They have mostly replaced Sigmoid and tanh, as they solve the problem of vanishing gradient, which is especially severe in deep learning models. The vanishing gradient problem refers to the property, that the first derivate (gradient) of Sigmoid and tanh is large around the threshold (0), but becomes closer and closer to 0 the further away (as the functions become flatter and flatter) - the gradient vanishes. ReLU gr(x), leaky ReLU gl(x) and Softplus gp(x) solve this:

- ReLu is continuous and almost differentiable (simple chose 0 or 1 at the threshold). The gradient still vanishes for values less than zero, leaky ReLU introduces a small gradient there as well. Softplus is a smooth approximation of ReLU and differentiable.

$$g_r(x) = \max(0, x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

$$g_r'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

$$g_l(x) = \begin{cases} 0,01x & x \leq 0 \\ x & x > 0 \end{cases}$$

$$g_l'(x) = \begin{cases} 0,01 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

$$g_p(x) = \ln(1 + e^x)$$

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

Identity function

- Another widely used activation function is the identity gi(x)=x - one could say, we simple output the sum without using an activation function. This is widely used in the final layer of network used for regression problems.

Softmax

- Softmax is often used in the final layer of network used for classification problems. It can not be shown graphically, as the output does depend on all neurons in the layer, not on a single neuron x as for the other functions. Let us collect the outputs xi of all these neurons in a vector x->.

$$g_{mj}(\vec{x}) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

$$\frac{\partial g_{mj}(\vec{x})}{\partial x_i} = g_{mj}(\vec{x})(\delta_{ji} - g_{mi}(\vec{x}))$$

where $\delta_{ji} = 1$ if $i = j$ and zero otherwise.

- Softmax is a smooth (and differentiable) approximation of the maximum function, i.e. it increases large values and decreases small values. It's main advantage is that the sum of all values is alway equal to 1 , so the values can be interpreted as probabilities and thus compared to each other.

Popular neural network setups for different problems

- Depending on the problem at hand, different choices for the activation function of the output layer and the loss function can be recommended:

- **Binary classification**
  the output layer consists of only one neuron, which is supposed to be 0 or 1. Sigmoid as activation function and binary cross-entropy as loss function often work quite well.

- **Multiple disjoint classes**
  the output layer has one neuron per class. Exactly one of these is supposed to be 1, all others should be 0. Softmax as activation function and cross-entropy as loss function often work quite well.

- **Multiple non-disjoint classes**
  the output layer again has one neuron per class, but now multiple neurons can and should be 1. Sigmoid as activation function and the sum of binary cross-entropy as loss function (each neuron can be considered a binary classification problem) often work well.

- **Regression**
  instead of predicting a class label, the network has to compute a (numeric) function value. The output layer consists of one neuron. The identity function as activation function and the squared error as loss function often work well.

- For the hidden layers, pretty much all networks today use ReLU or one of its variants (leakyReLU, Softplus) as activation function.