09: Neural Networks (basic)

Presented by Dr Anna Scius-Bertrand
Slides from Dr Giuseppe Cuccu

UNI FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Machine Learning

Bachelor in Computer Science

Arpil 29, 2024

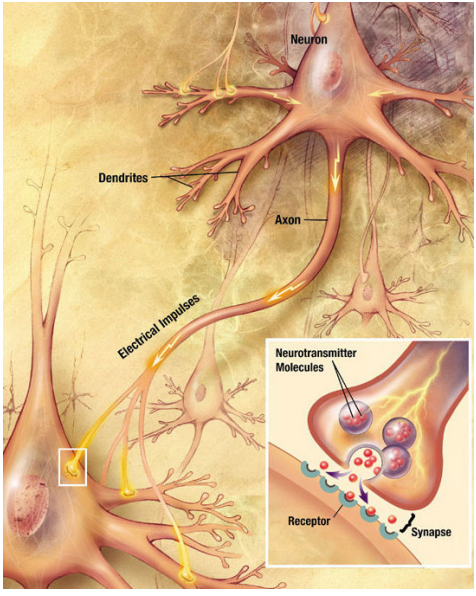
Where are we

- Our very first model + learning algorithm was the Perceptron
- Main limitations:
  - **(i)** The model is limited to linear separation
  - **(ii)** The learning algorithm requires linearly separable data
- Neural Networks overcome such limitations by extending the Perceptron with **nonlinear activation functions** and the **Backpropagation algorithm**
- As their name suggests, they were originally inspired by *biological neural networks*, such as the human nervous system

Today's menu

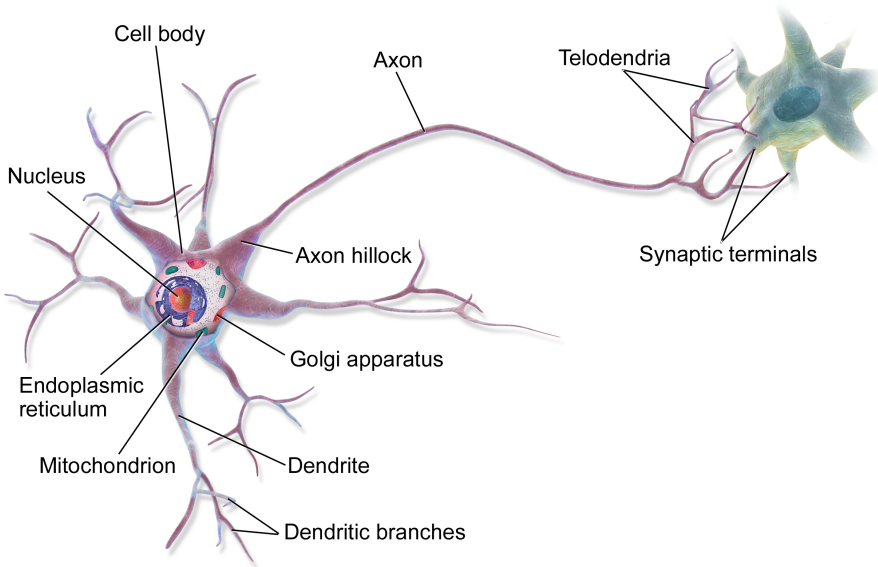
- Original Inspiration
- Multilayer Networks
- Universal Approximation
- Stochastic Gradient Descent
- Backpropagation

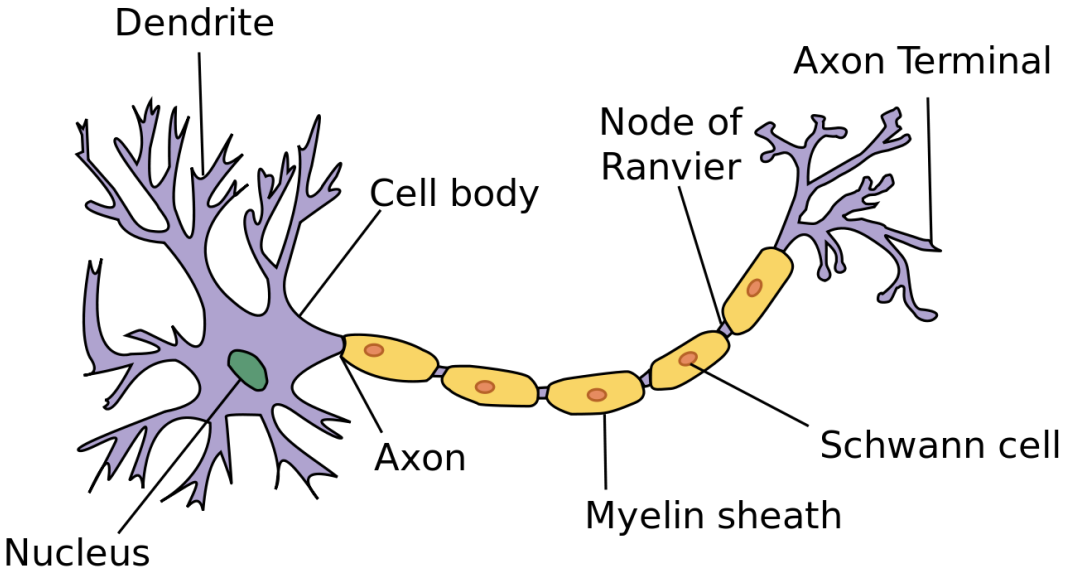
Neuroscience roots

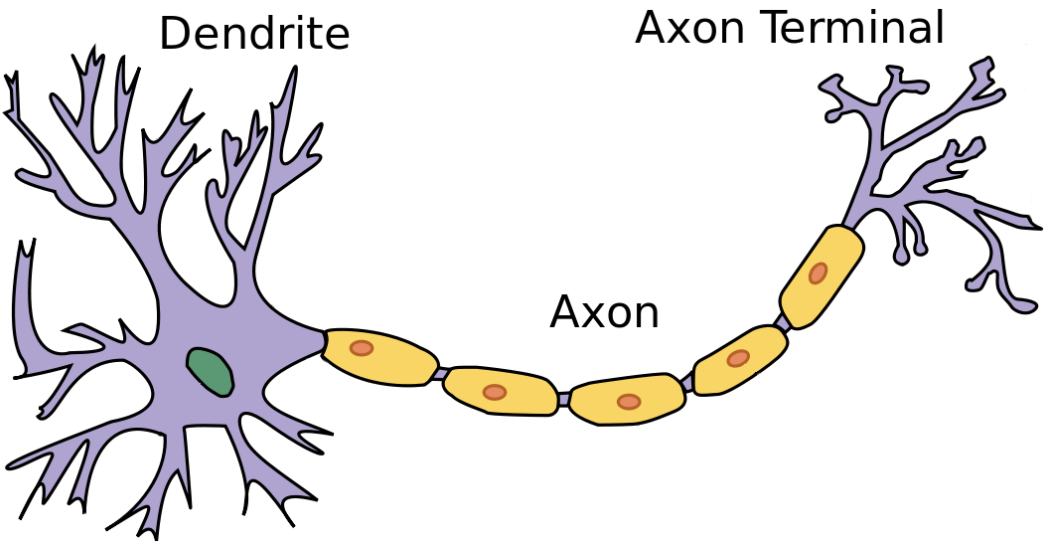


- There are many different types of neurons in the human nervous system.
- Lots of specializations, and each type itself quite complex.
- Neurons connect to each other with *synapses*, thus forming a *network*. This complicates things even more.
- How to understand/model this system?
- Basic idea: **simplify** the neuron until reduced to its essentials.

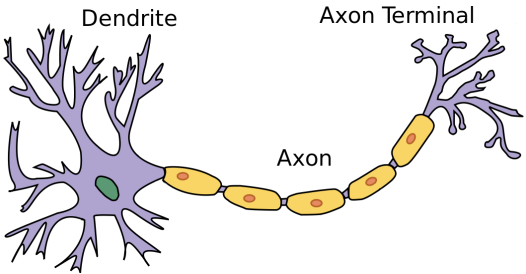
(source for all neuron images: *Wikipedia*)


Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic reticulum
Mitochondrion
Dendrite
Dendritic branches


Dendrite
Axon Terminal
Node of
Ranvier
Cell body
Axon
Schwann cell
Myelin sheath
Nucleus


Dendrite
Axon Terminal
Axon

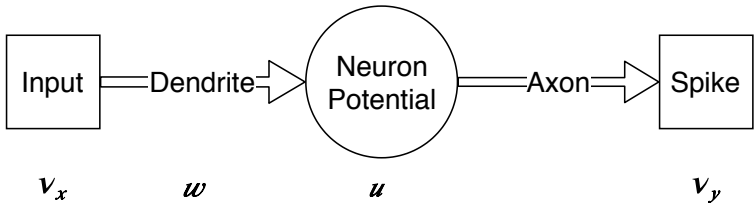
Simplifying Real Neurons



- A neuron receives *inputs* from a number of other neurons through its *dendrites*. The contact points are called *synapses*. Its outputs are distributed by its *axon*, itself having multiple synapses.

- The inputs typically come from the outputs of other neurons, in the form of **spikes**: short pulses of electrical current
- The neuron averages them over time, which can be represented by its input **spike frequency**
- The spikes arrive at the neuron's membrane and alter its **membrane (electric) potential**
- There are both *excitatory* and *inhibitory* connections between neurons, each with different strength, i.e. **synaptic weight**

Modeling Artificial Neurons



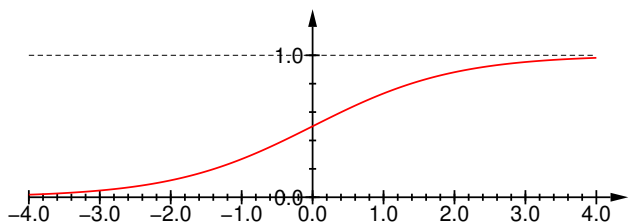
- Input of a neuron: outputs of other neurons, i.e. array of their *spike frequencies* ν.
- Strength of connection: array of *synaptic weights* $\boldsymbol{w}$. Weights are positive for excitatory connections, and negative for inhibitory ones; absolute value is large for strong connections, small for weak ones.
- Neuron potential: sum of weighted inputs, i.e. its *membrane potential* $\boldsymbol{u}$.

Artificial Neurons

When the neuron's membrane potential exceeds a threshold then the neuron *emits* a spike (which can propagate to multiple receivers) and *resets* its membrane potential. The spike frequency as a function of the incoming power is a non-linear **transfer function** (also called **activation** or simply **non-linearity**).

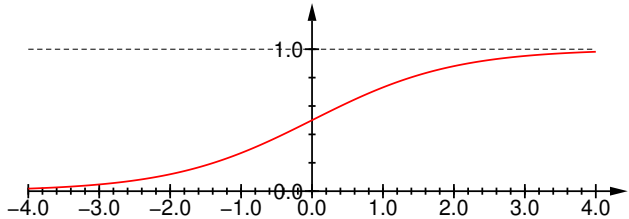
Classical literature utilizes the *logistic function*:

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



The Logistic Function

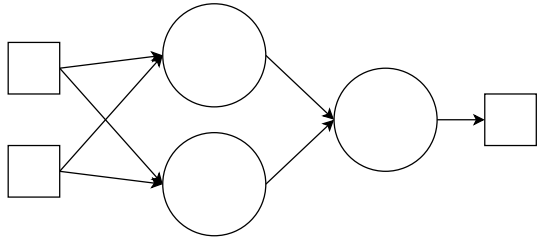
$$\sigma(u) = \frac{1}{1+e^{-u}}$$



- Nonlinear, but close to linear around/near $x = 0$
- Output is bounded between 0 and 1 (open)
- Inflection point is at $x = 0$ with a value of $y = 0.5$
- Very high and very low inputs are squashed to the boundaries (*saturation*)
- Its derivative is easy by design (more on why later): $\frac{\partial \sigma(u)}{\partial u} = \sigma(u)(1 - \sigma(u))$
- Functions with shape resembling an S, such as the Logistic, are often called *sigmoids* or *sigmoidal functions* (also check out e.g. `arctan`)

Networks of Neurons

- Neurons in nervous systems are connected in large, complex networks. Our artificial neurons can approximate this concept using **directional graphs** of computation, with neurons in the nodes: (round: neuron; square: input/output)



- There are two ways to stack neurons:
  - **In parallel:** neurons that are supposed to fire independently and can be modeled together (i.e. in a same **layer**)
  - **In series:** neurons which are connected sequentially axion-dendrite (i.e. in separate layers), meaning the output of the first neuron is an input for the second neuron: the model runs in *sequence*

Artificial Neurons

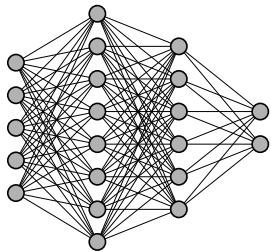
- While inspiration from biology, neural networks are so abstract and simplistic that in the end they retain little in common with their biological counterpart. They are better understood as a mathematical sequential learning machine.
- Let u_i be the membrane potential and ν_i be the firing rate of neuron i, and let w_{ji} be the synaptic weight of the connection from i to j (which is zero if the neurons are not connected). We obtain the model:

$$u_j \leftarrow \sum_{i=1}^{k} w_{ji} \cdot \nu_i \ , \qquad \nu_j \leftarrow \sigma(u_j)$$

- The relation $u_j \leftarrow \sum_{i=1}^{k} w_{ji} \cdot \nu_i$ is familiar: if ν_i are the inputs, then this is a linear function, which **links neurons to Perceptron models**.

Multilayer Neural Networks

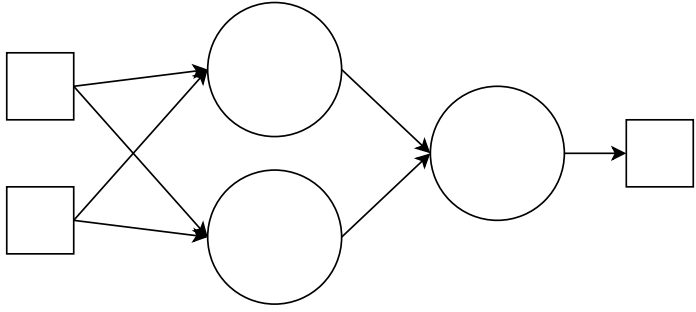
- The resulting architecture is called a *layered feed-forward neural network*, or more commonly a **Multi-Layer Perceptron (MLP)**



This example network has
an **input layer** with 5 *nodes*,
two **hidden layers** with 8 and 6 *neurons*, and
an **output layer** with 2 *neurons*
(the number of connections grows fast)

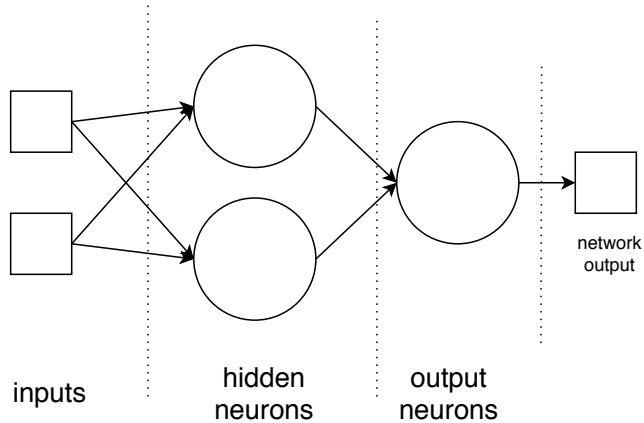
- The sizes of the input and output layers correspond to the *dimensions* of the vectors x and y, which are determined by the problem
- Number and size of the hidden layers is arbitrary, changing the complexity of the model

Neural Networks



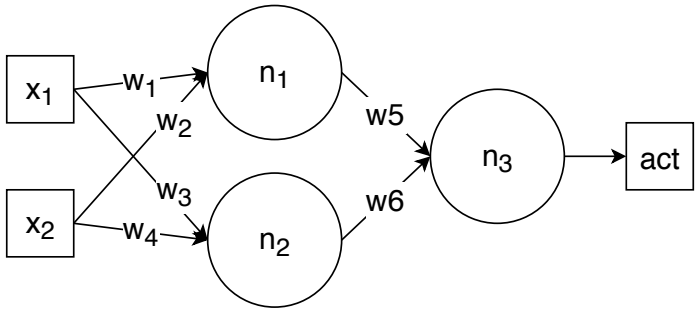
Neural networks are easily depicted as connected graphs

Neural Networks



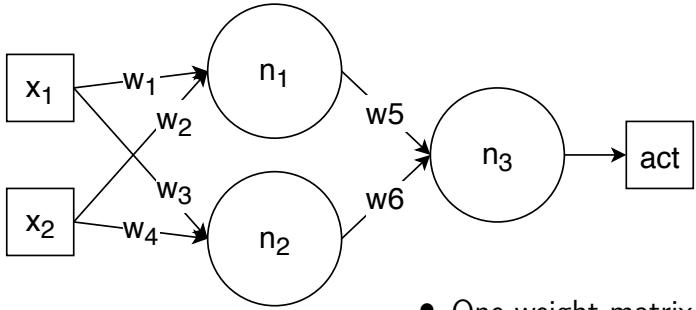
The input layer is **not** composed of neurons, but houses the inputs to the model.
The output of the model corresponds to the outputs of
the neurons in the network's output layers.

Neural Networks



- Each layer of neurons receives inputs through **weighted connections**.
- The output dimensionality corresponds to the number of output neurons.
- Each neurons weights it inputs, aggregates them, then activates.
- The graph is just a human-readable representation of a complex equation.

Neural Networks

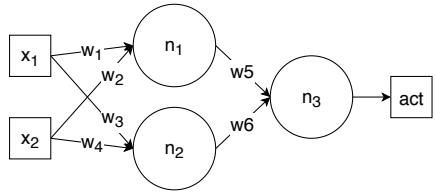


$$W_{in} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \quad , \quad W_{hid} = \begin{bmatrix} w_5, w_6 \end{bmatrix}$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad , \quad n_{hid} = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix} \quad , \quad n_{out} = \begin{pmatrix} n_3 \end{pmatrix}$$

- One weight matrix connects two layers
- Each row holds the weights *entering* a neuron
- Each column shares the same connection *origin*

Neural Networks



$$W_{in} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \quad , \quad W_{hid} = \begin{bmatrix} w_5, w_6 \end{bmatrix}$$

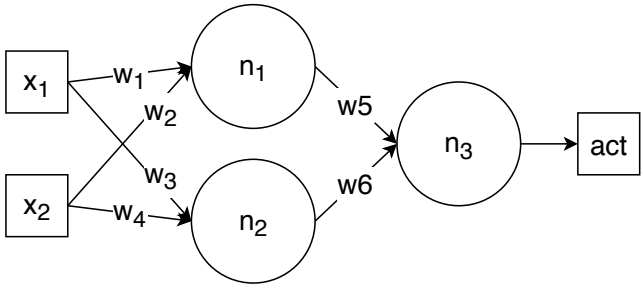
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad , \quad n_{hid} = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix} \quad , \quad n_{out} = \begin{pmatrix} n_3 \end{pmatrix}$$

$$n_{hid} = \sigma(W_{in}x) = \begin{pmatrix} \sigma(w_1x_1 + w_2x_2) \\ \sigma(w_3x_1 + w_4x_2) \end{pmatrix}$$

$$n_{out} = \sigma(W_{hid}n_{hid}) = \begin{pmatrix} \sigma(w_5n_1 + w_6n_2) \end{pmatrix}$$

$$act = n_3 = \sigma\left[w_5\sigma(w_1x_1 + w_2x_2) + w_6\sigma(w_3x_1 + w_4x_2)\right]$$

Function Approximation



$$act = \sigma\left[\mathbf{w_5}\sigma(\mathbf{w_1}x_1 + \mathbf{w_2}x_2) + \mathbf{w_6}\sigma(\mathbf{w_3}x_1 + \mathbf{w_4}x_2)\right]$$

- Parametrization: **structure**, **activations** and **weights**
  The first two are most often *fixed hyperparameters*
- Remember that function composition increases complexity *fast*
- Network complexity gives an **upper bound** on the complexity of the functions that can be approximated (think: what if all $w = 0$?)

Multilayer Neural Networks

$$act = \sigma\left[\mathbf{w_5}\sigma(\mathbf{w_1}x_1 + \mathbf{w_2}x_2) + \mathbf{w_6}\sigma(\mathbf{w_3}x_1 + \mathbf{w_4}x_2)\right]$$

- Hidden layers traditionally employ a *sigmoidal* transfer (activation) function; The choice for the output layer instead is *task specific*, depending on how the model output is interpreted:
  - **Regression:** sigmoidals are bound; to achieve an unbounded range of outputs the *identity function* is a common choice for the transfer of the output neurons, i.e. the output layer consists of *linear* neurons.
  - **Classification:** the values range does not matter, so either linear or sigmoid activations can be used. Multiclass problems however are typically addressed through multiple output *decision neurons*, i.e. one for each class.
  - **Specific applications** can have variations over the two types above: for example control problems with a discrete action set are handled similarly to Classification, while continuous actions are closer to Regression (more on that later)

Multilayer Neural Networks

- The linear function $u = W \cdot v_x$ is usually extended to an affine function $u = W \cdot v_x + b$ by means of a so-called **bias input**. This is a constant firing rate (value) of *one*, connected to each and all neurons with weights b_i.
- A multilayer neural network thus applies two types of transformations in each layer in sequence:
  - **A linear combination:** left multiplication with the matrix $W^{(i)}$. This matrix is a parameter of the model, subject to learning.
  - **A non-linear function:** component-wise transfer function σ. This function is traditionally fixed, with no parameters.
- The non-linearities are very important! Without them the model can be simplified into the linear map $\mathcal{W} = W^{(1)} \cdot ... \cdot W^{(\ell)}$ with ℓ number of layers. (can you prove it?)

  One-layer neural networks are linear models, like a Perceptron.
- A neural network with sigmoid transfer and (at least) one hidden layer (of arbitrary size) is instead proven to be able to compute "everything", i.e. it is a **generic function approximator**.

Universal Approximation Property

Theorem. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $K \subset \mathbb{R}^p$, and $\mathcal{C}(K)$ denotes the space of continuous functions $K \to \mathbb{R}$. Then, given a function $g \in \mathcal{C}(K)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $q \in \mathbb{N}$ and a set of coefficients $w_i^{(1)} \in \mathbb{R}^p$, $w_i^{(2)}, b_i \in \mathbb{R}$ (for $i \in \{1, \ldots, q\}$), such that

$$f : K \to \mathbb{R}\,; \quad f(x) = \sum_{i=1}^{q} w_i^{(2)} \cdot \sigma\Big((w_i^{(1)})^T x + b_i \Big)$$

is an ε-approximation of g, that is,

$$\|f - g\|_\infty := \max_{x \in K} |f(x) - g(x)| < \varepsilon \ .$$

Corollary. The theorem extends trivially to multiple outputs.

Corollary. Neural networks with a single sigmoidal hidden layer and linear output layer are **universal function approximators**.

Universal Approximation Property

- This means that, in the space of all functions, for any given target function g, there exists a sequence of networks $\{f_k\}_{k \in \mathbb{N}}$ that converges (pointwise) to the target function to arbitrary precision in k *learning steps*
- The trade-off of course is that this approximation will require more (hidden) neurons (i) as the complexity of g grows, and (ii) for increasingly smaller ϵ
- This also implies that the space of networks with fixed structure and activation function map weight vectors to a definite, bounded *subspace* of the space of all functions
- Networks with higher complexity (e.g. large hidden layer sizes) will map to a *larger space*, as they include more complex functions (remember, network complexity is only upper bound)
- Note: the universal approximation property is not as special as it seems. For example, *polynomials* are universal approximators (Weierstrass theorem)

From Models to Learners

- The class of functions represented by neural networks can approximate the solution to *any problem* to arbitrary precision – provided that the network is "big enough" (to approximate the underlying function's complexity)
- But how can we select in practice the network structure (ergo size), activation function and weights? We still need a method to learn the parametrization
- Neural Networks by themselves are just **a class of models**, a family of *parametrized generic function approximators*
- As traditional, for now we will consider structure and activation as fixed (i.e. user-defined *hyperparameters*), and derive a gradient-based training method as a two-step procedure (akin to the Perceptron):
  1. Compute the derivative of the empirical risk w.r.t. the weights
  2. Change the weights so that the empirical error is reduced

  This process can be iterated, until a *local* optimum is reached

(Online) Steepest Descent Training

It is typically easier to understand if we begin with the second step (reduce empirical error). Hypothesize we have a way to compute the derivative of the empirical risk w.r.t. the weights $\nabla_w E(w)$ (the *error gradient*).

- Let w denote a vector collecting all weights of a neural network: think of it as a "linearized" version of all of its weight matrices, concatenated then flattened
- Let f_w be the function corresponding to the network with weights w
- Let the **error** of the network be computed as average risk in function of the network weights (S is a *subset* of points from the training set):

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

Choosing different S yields different training techniques

(Online) Steepest Descent Training

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

There are traditionally three ways to estimate the error for a neural network, each with its pros and cons:

- **Batch mode:** run the sum over the whole dataset, $|S| = n$. This produces a single, dataset-wide error for each update step. If the dataset is very large (or uniform) this may severely limit performance.
- **Online mode:** compute a different error for each element in the dataset, $|S| = 1$. These fast updates sometimes perform better than slower, more careful updates, especially on simpler problems.
- **Mini batches:** a trade-off between the two above; randomly select small subsets at each step, $|S| \ll n$. This allows for a certain degree of control on performance depending on batch size, at the cost of introducing a new hyperparameter (most common nowadays).

(Online) Steepest Descent Training

- The derivative of the error function $\nabla_w E(w)$ gives us a *gradient* of (increasing) error. The negative gradient thus points in the direction where it decreases fastest.
- We can then derive the learning rule:

$$w \leftarrow w - \eta \cdot \nabla_w E(w)$$

  with $\eta > 0$ learning rate
- This method is known as **stochastic gradient descent**: a generic, iterative, simple optimization scheme based on *error gradients*, which is applied in many forms throughout ML
- The learning rate η can be a "small" constant or decay over time (i.e. training iterations)
- A decaying learning rate of the form $\eta(t) = \eta_0 / t$ eventually reaches a value below machine precision (becoming "zero"), making the algorithm converge/terminate on a local optimum of the error function

Backpropagation

Let's now go back to the first step: computing the error gradient $\nabla_w E(w)$

- The error is a simple sum over loss terms, each of the form $E(w) = L(f_w(x), y)$ for point (x, y)
- We can expand this error as

$$E(w) = L\left[\sigma\left(W^{(\ell)} \cdot \sigma\left(W^{(\ell-1)} \cdot \sigma\left(\dots W^{(1)} \cdot x \dots\right)\right)\right), y\right]$$

  where each $W^{(k)}$ is the weight matrix for layer k, and σ is the component-wise non-linearity
- The gradient (i.e. derivative) of this error function can be calculated using the *chain rule*, yielding the **Backpropagation algorithm** (often shortened to *backprop*)
- Most errors in understanding and implementing the rule come from **not paying enough attention to the *index of the layer***: please be extra careful here

Backpropagation: output layer

- For the **last** (output) layer, $k = \ell$, the gradient is defined as $\delta^{(k)} = \frac{\partial E}{\partial u^{(k)}}$
- This means that we have $E = L(v^{(\ell)}, y)$ and $v^{(\ell)} = \sigma(u^{(\ell)})$
- Assuming the squared loss $L(v^{(\ell)}, y) = \frac{1}{2}\|v^{(\ell)} - y\|^2$ we obtain

$$\begin{aligned}
\delta^{(\ell)} &= \frac{\partial L(\sigma(u^{(\ell)}), y)}{\partial u^{(\ell)}} \\
&= \frac{\partial L(v^{(\ell)}, y)}{\partial v^{(\ell)}} \cdot \frac{\partial v^{(\ell)}}{\partial u^{(\ell)}} \\
&= \left(v^{(\ell)} - y\right) \cdot \left(1 - \sigma(u^{(\ell)})\right) \cdot \sigma(u^{(\ell)})
\end{aligned}$$

- To compute this we need the target/label y (SL!), the network output $v^{(\ell)}$, and the membrane potential $u^{(\ell)}$ (the derivative of the logistic here has u only appearing inside a σ: this is not generally true)
- This computation can be generalized to any arbitrary (differentiable) loss L and transfer function σ

Backpropagation: output layer

- Now we are interested in the derivative $\frac{\partial E}{\partial W_{i,j}^{(\ell)}}$ of the error w.r.t. a weight in the last weight matrix, with $u^{(\ell)} = W^{(\ell)} \cdot \nu_x^{(\ell-1)}$
- The derivative is:

$$\frac{\partial E}{\partial W_{i,j}^{(\ell)}} = \frac{\partial E}{\partial u_i^{(\ell)}} \cdot \frac{\partial u_i^{(\ell)}}{\partial W_{i,j}^{(\ell)}} = \delta_i^{(\ell)} \cdot \nu_j^{(\ell-1)}$$

- This introduces an additional term: **the output of the second-to-last layer** $\nu^{(\ell-1)}$
- Before moving on (backprop on hidden layers), make sure that you understand what $\nu^{(\ell-1)}$ is, and why did it appear in the equation

Backpropagation: hidden layers

- Let us now consider earlier layers $k < \ell$: consider the derivative $\frac{\partial E}{\partial W_{i,j}^{(k)}}$
- The same decomposition as before works:

$$\frac{\partial E}{\partial W_{i,j}^{(k)}} = \frac{\partial E}{\partial u_i^{(k)}} \cdot \frac{\partial u_i^{(k)}}{\partial W_{i,j}^{(k)}} = \delta_i^{(k)} \cdot \nu_j^{(k-1)}$$

- But this time, we are not finished yet!
- This requires the layer inputs $\nu^{(k-1)}$, i.e. the output of the previous layer
- We also need the **residual error** $\delta^{(k)}$, which is *back-propagated* from the outputs
- At each step, one layer is adjusted to reduce a bit the error, then the *rest* of the error is sent down to the preceeding layers as "their responsibility" to deal with

Backpropagation: hidden layers

- Here is the last part of the derivation for the hidden layers:

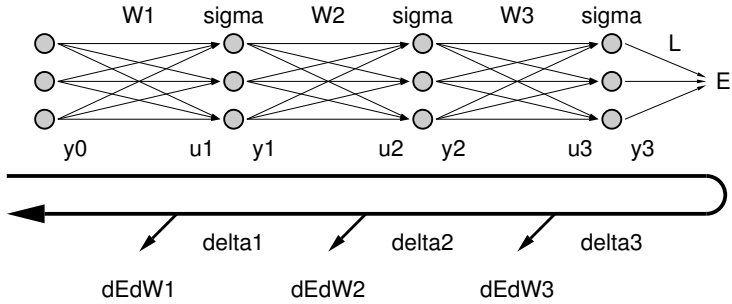
$$\delta^{(k)} = \frac{\partial E}{\partial u^{(k)}} = \frac{\partial E}{\partial u^{k+1}} \cdot \frac{\partial u^{(k+1)}}{\partial v^{(k)}} \cdot \frac{\partial v^{(k)}}{\partial u^{(k)}} =$$

$$\delta^{(k+1)} \cdot W^{(k+1)} \cdot \sigma'(u^{(k)})$$

- The errors $\delta^{(k)}$ can be computed based on the errors $\delta^{(k+1)}$: the error is computed *end to front*, from the last layers (where we can compute it against the label) *backwards* until the first, hence "backpropagation"
- The $\delta^{(k)}$-terms are only intermediate results for the computation of the weight derivatives
- In all steps we need the **activations from multiple layers** of the network
- When implementing a neural network, you need to maintan a *state* holding the current sequence of activations, to be used (in reverse order, end to start) to compute your error gradients

Backpropagation: outline

This hints at the following order of computation:



1. Propagate the *input* front-to-back: the **forward pass**.
2. Compute the error at the network output: we need to change the weights, one layer at a time, in the direction that *decreases* this output error.
3. Propagate the *error* back-to-front: the **backward pass**.

Summary

- The sigmoid transfer function extends the Perceptron into a neuron model
- Multiple neurons can be stacked in parallel (layers), and multiple layers in sequence, to define feed-forward neural networks (multi-layer Perceptrons)
- Multilayer nonlinear neural networks are proven universal function approximators — if you have infinite neurons
- Networks can be trained by online or batch gradient descent
- The error gradient can be computed and propagated efficiently with the backpropagation algorithm
- This algorithm incorporates no exploration capabilities, thus converging into the nearest local optimum

Extra material

- Stochastic Gradient Descent:
  https://en.wikipedia.org/wiki/Stochastic_gradient_descent
- Clear derivation of Backpropagation:
  https://codesachin.wordpress.com/2015/12/06/backpropagation-for-dummies/
- Stanford's tutorial:
  http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
- Easy to read, step-by-step backpropagation:
  https://eli.thegreenplace.net/2018/backpropagation-through-a-fully-connected-layer/
- Tutorial on function approximation and why neural networks:
  https://machinelearningmastery.com/neural-networks-are-function-approximators/