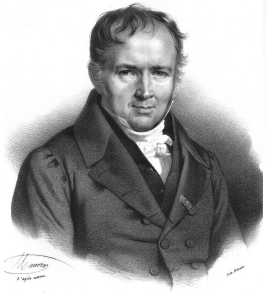
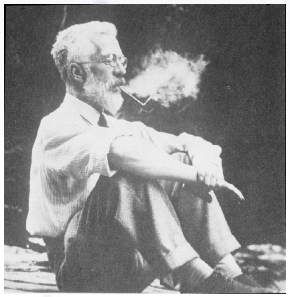
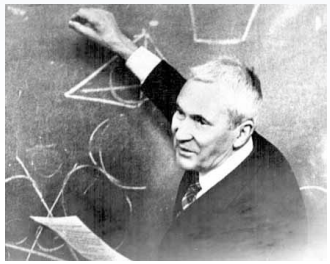
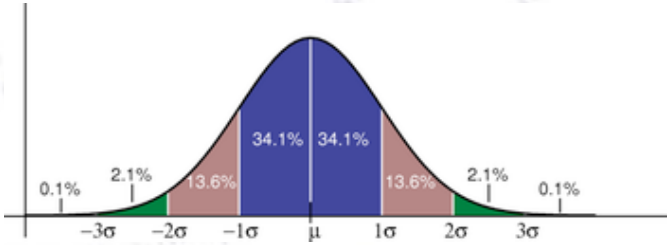
Neural Networks & Deep Learning



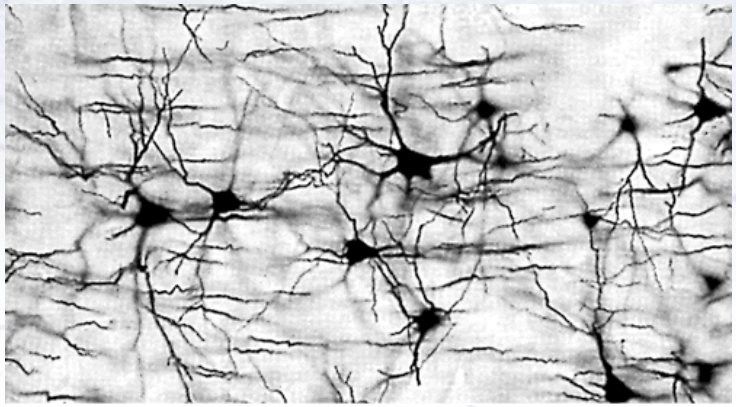





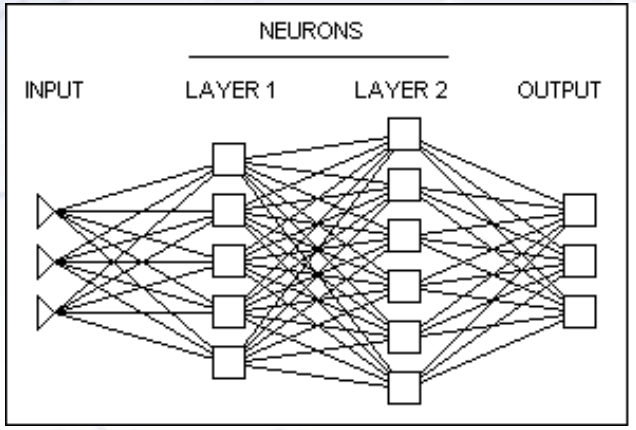
Troels C. Petersen (NBI)



"Statistics is merely a quantisation of common sense - Machine Learning is a sharpening of it!"

Neural Networks (NN)





In machine learning and related fields, artificial neural networks (ANNs) are computational models inspired by an animal's central nervous systems (in particular the brain) which is capable of ***machine learning*** *as well as* ***pattern recognition****.*

Neural networks *have been used to solve a wide variety of tasks that are hard to solve using ordinary rule-based programming, including* ***computer vision*** *and* ***speech recognition****.*

[Wikipedia, Introduction to Artificial Neural Network]

A "Linear Network"

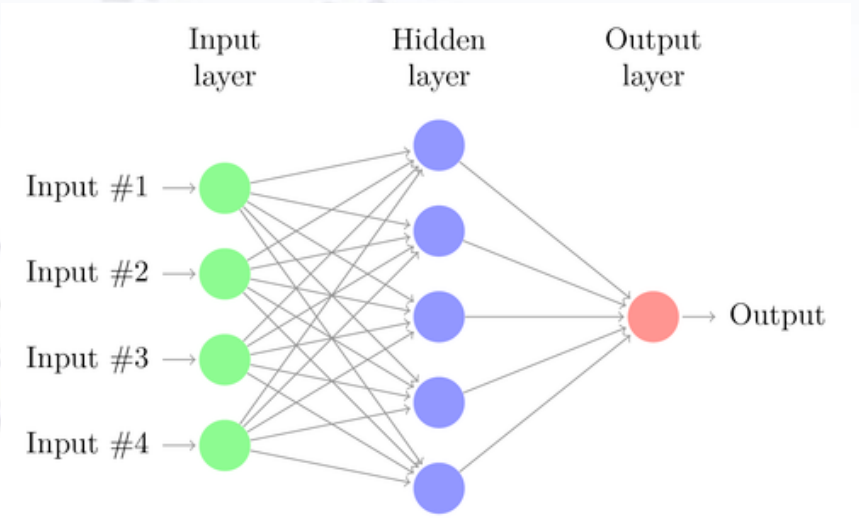
Imagine that we consider a "Linear Network", and use the (simplest) architecture: A single layer (linear) perceptron:

$$t(x) = a_0 + \sum a_i x_i$$

As can be see, this is simply **a linear regression in multiple dimensions** or the (linear) Fisher Discriminant.

Well, then we could consider putting in a hidden (linear) layer:

$$tt(x) = t(a_0 + \sum a_i x_i)$$

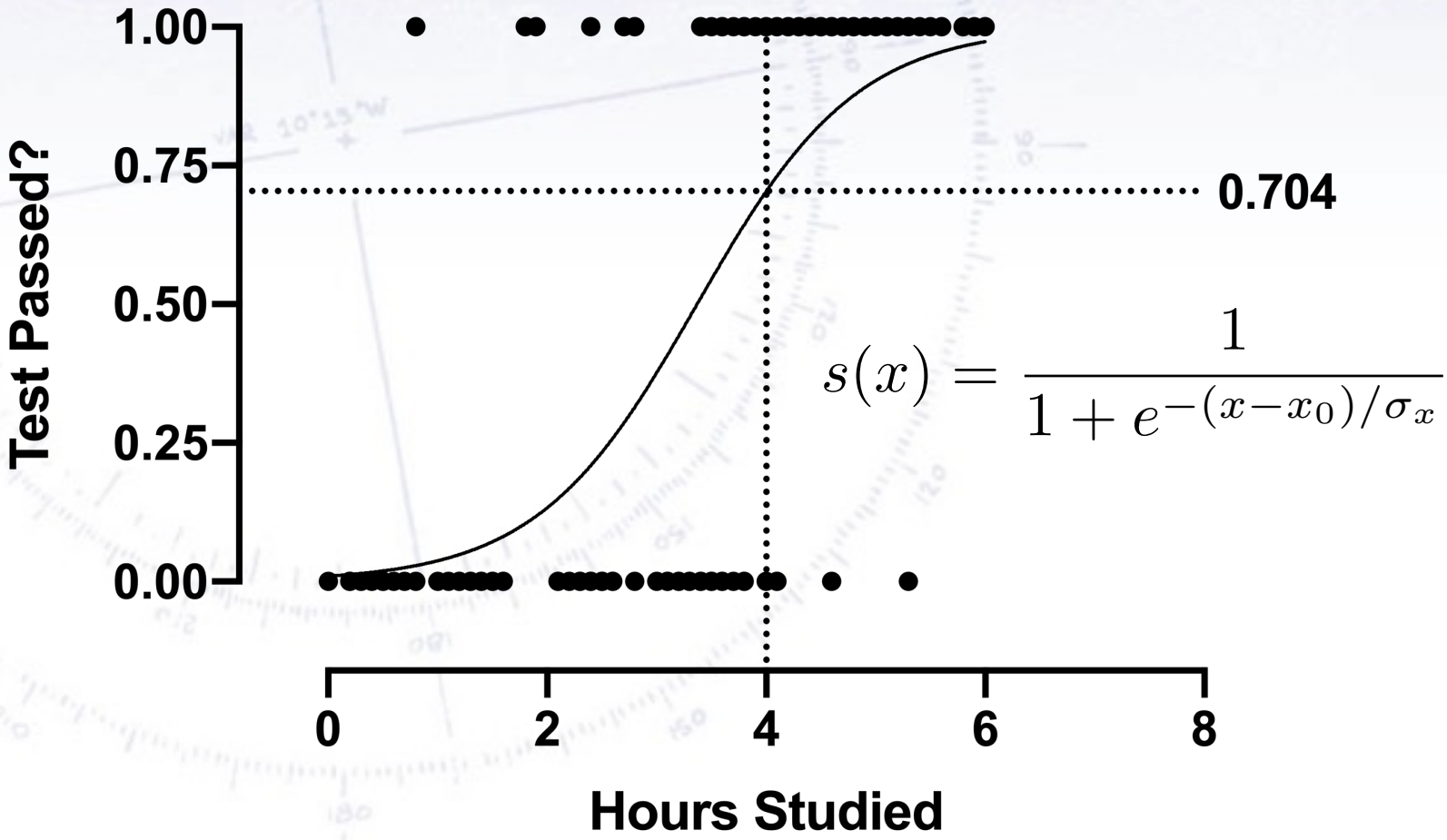


However, this doesn't help anything
as combination of linear functions remain linear. It boils down to the Fisher again!

What we need is something non-linear in the function…

Logistic Regression

Though the word "regression" suggests otherwise, this is in fact a way of doing classification, as the "regression" is usually for a score (s) in the interval [0,1].

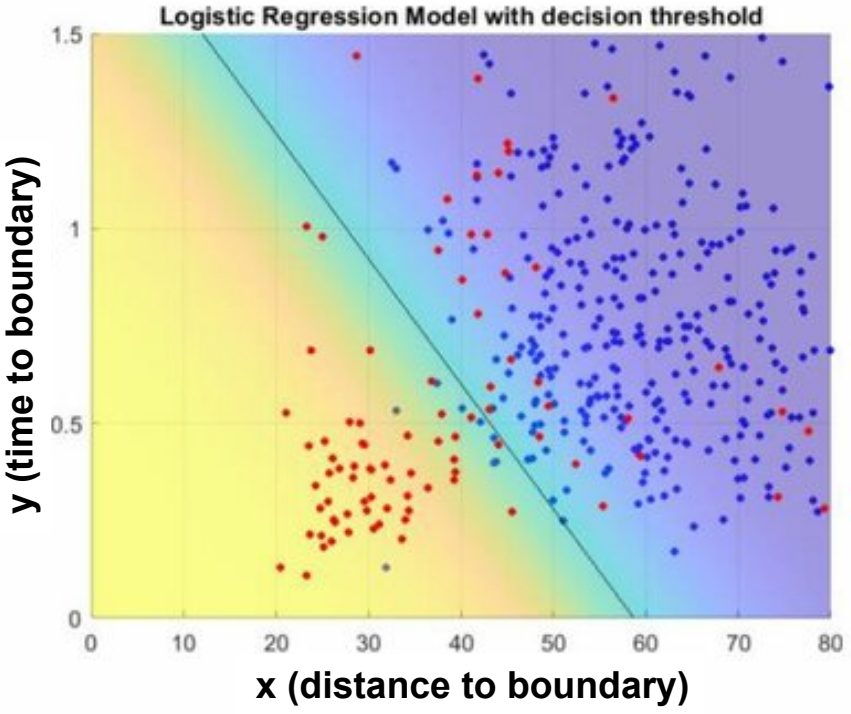


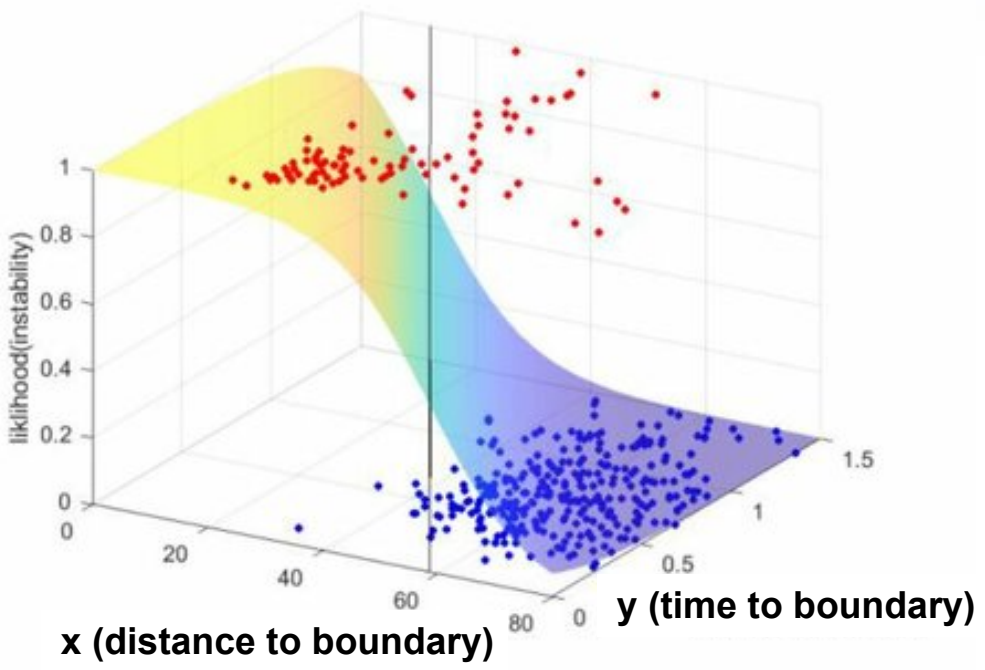
Logistic Regression

Though the word "regression" suggests otherwise, this is in fact a way of doing classification, as the "regression" is usually for a score (s) in the interval [0,1].

The model expands naturally with more parameters:

$$s(x) = \frac{1}{1 + e^{-(x-x_0)/\sigma_x - (y-y_0)/\sigma_y}}$$





Neural Networks

Neural Networks combine the input variables using a "activation" function s(x) to assign, if the variable indicates signal or background.

The simplest is a single layer perceptron:

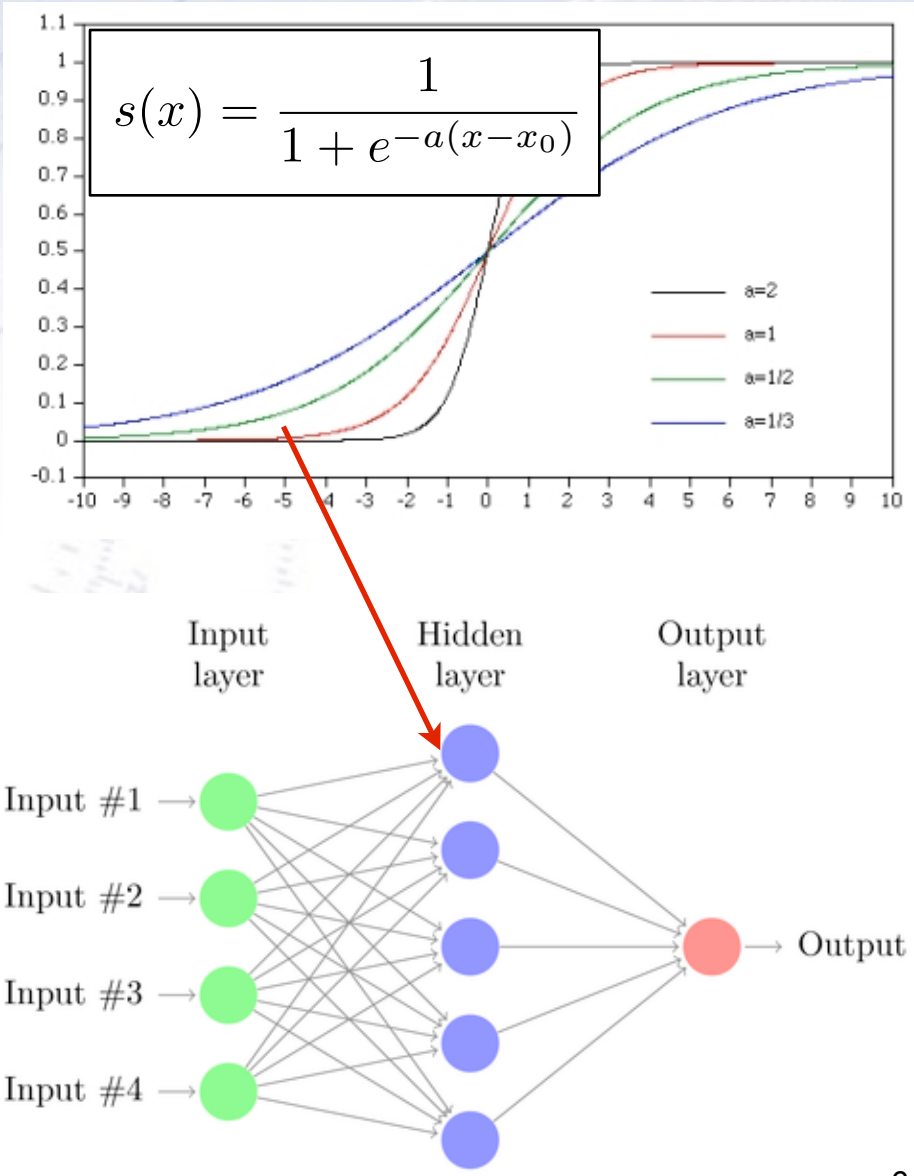
$$t(x) = s\left(a_0 + \sum a_i x_i\right)$$

This can be generalised to a multilayer perceptron (shown right, 1 hidden layer):

$$t(x) = s\left(a_i + \sum a_i h_i(x)\right)$$

$$h_i(x) = s\left(w_{i0} + \sum w_{ij} x_j\right)$$

Activation function can be any "sigmoidal" function.

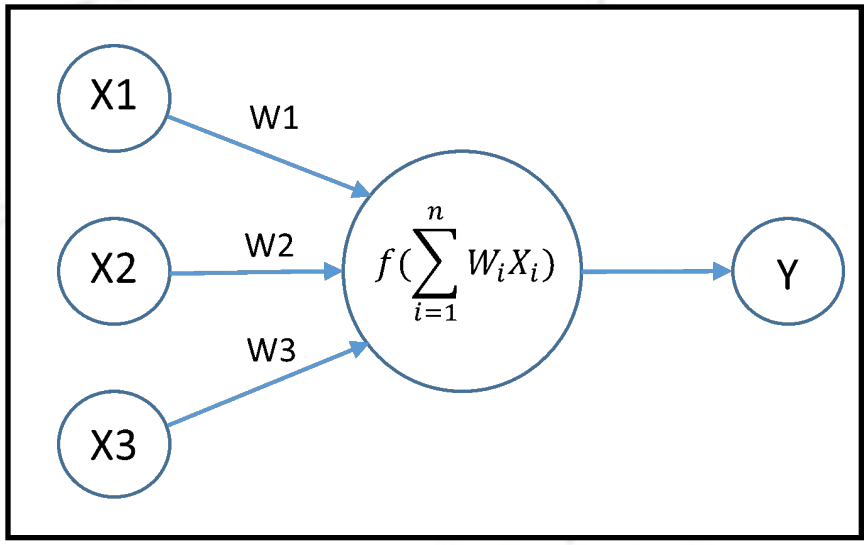


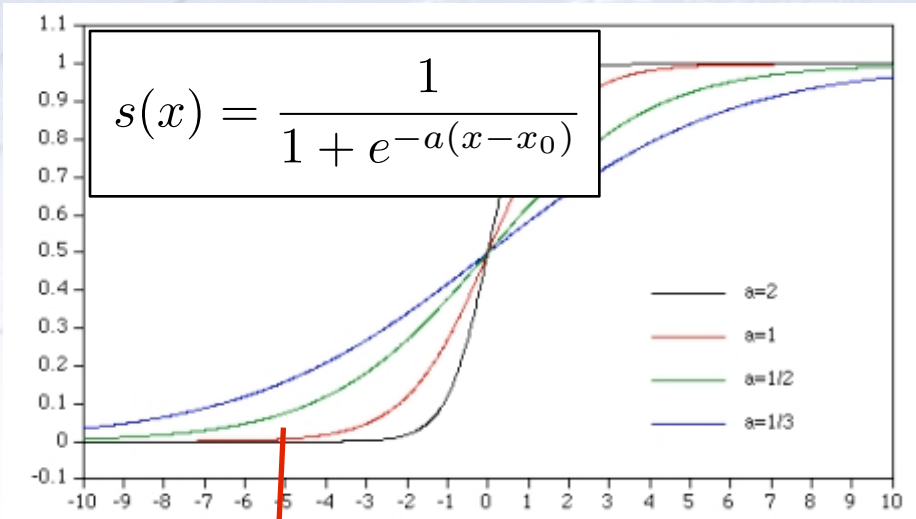
Neural Networks

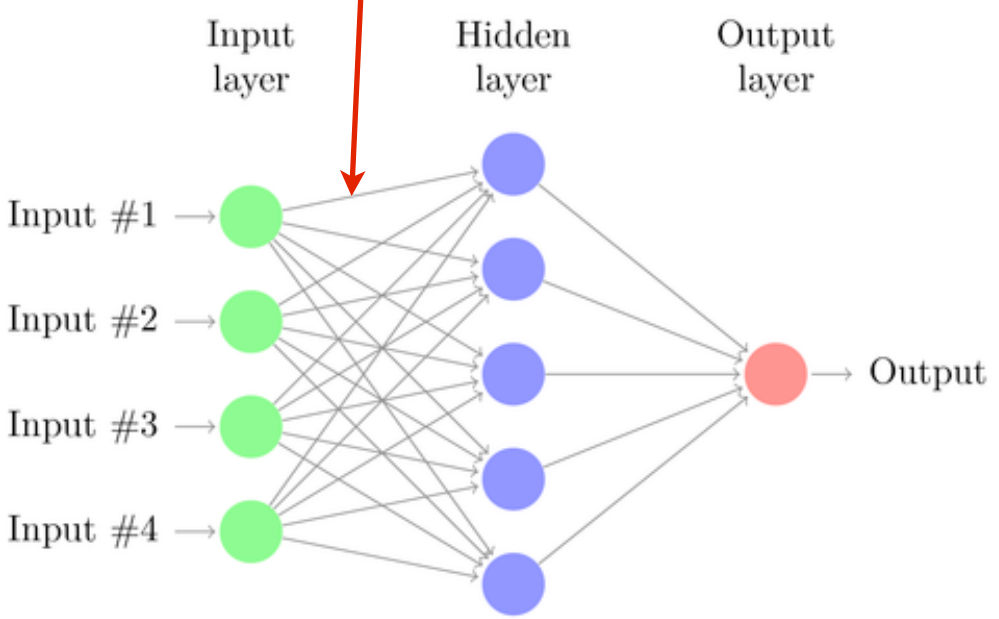
Neural Networks combine the input variables using a "activation" function s(x) to assign, if the variable indicates signal or background.

The simplest is a single layer perceptron:

$$t(x) = s\left(a_0 + \sum a_i x_i\right)$$







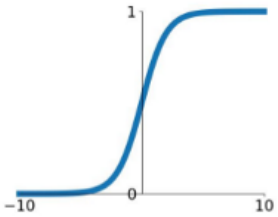
Activation Functions

There are many different activation functions, some of which are shown below. They have different properties, and can be considered a HyperParameter.

Activation Functions

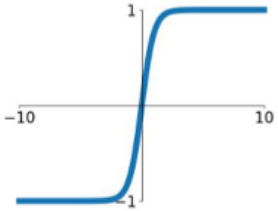
Sigmoid

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



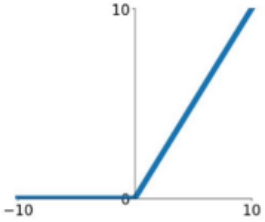
tanh

$$\tanh(x)$$



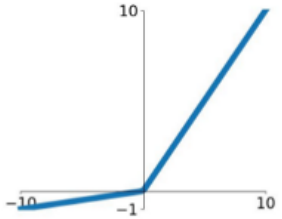
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

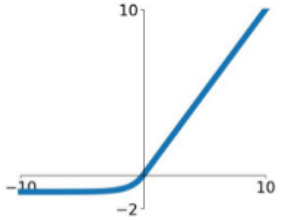


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



For a more complete list, check: https://en.wikipedia.org/wiki/Activation_function

SoftMax Function

The "SoftMax" σ:

- Input: A vector of values (e.g. from a row of neurons)
- Returns: A probability distribution of the same length

$$\sigma_i(\vec{z}) = e^{z_i} / \sum_j e^{z_j}$$

This makes it very useful as the last activation function in a neural network, which normalises the output to yield **predictions for the output classes**.

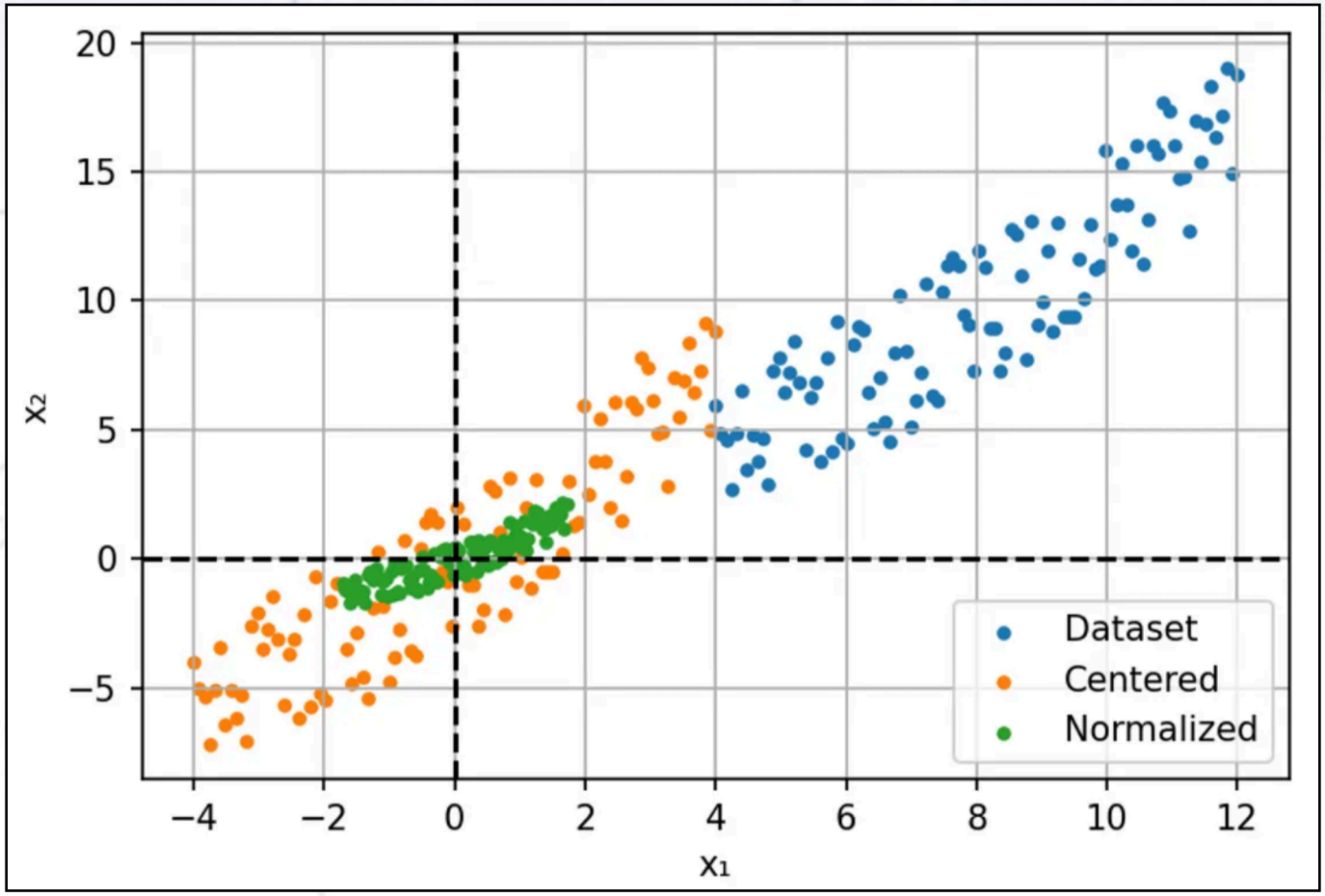
An example could be two nodes with values 0 and 10 (using "unit" length):

$$\sigma_1(0, 10) = \left(\frac{1}{1 + e^{10}} , \frac{e^{10}}{1 + e^{10}} \right) = (0.00005, 0.99995)$$

If you want to see how the SoftMax function is altered for attention, check out:
https://en.wikipedia.org/wiki/Softmax_function#Numerical_algorithms

Normalising Inputs

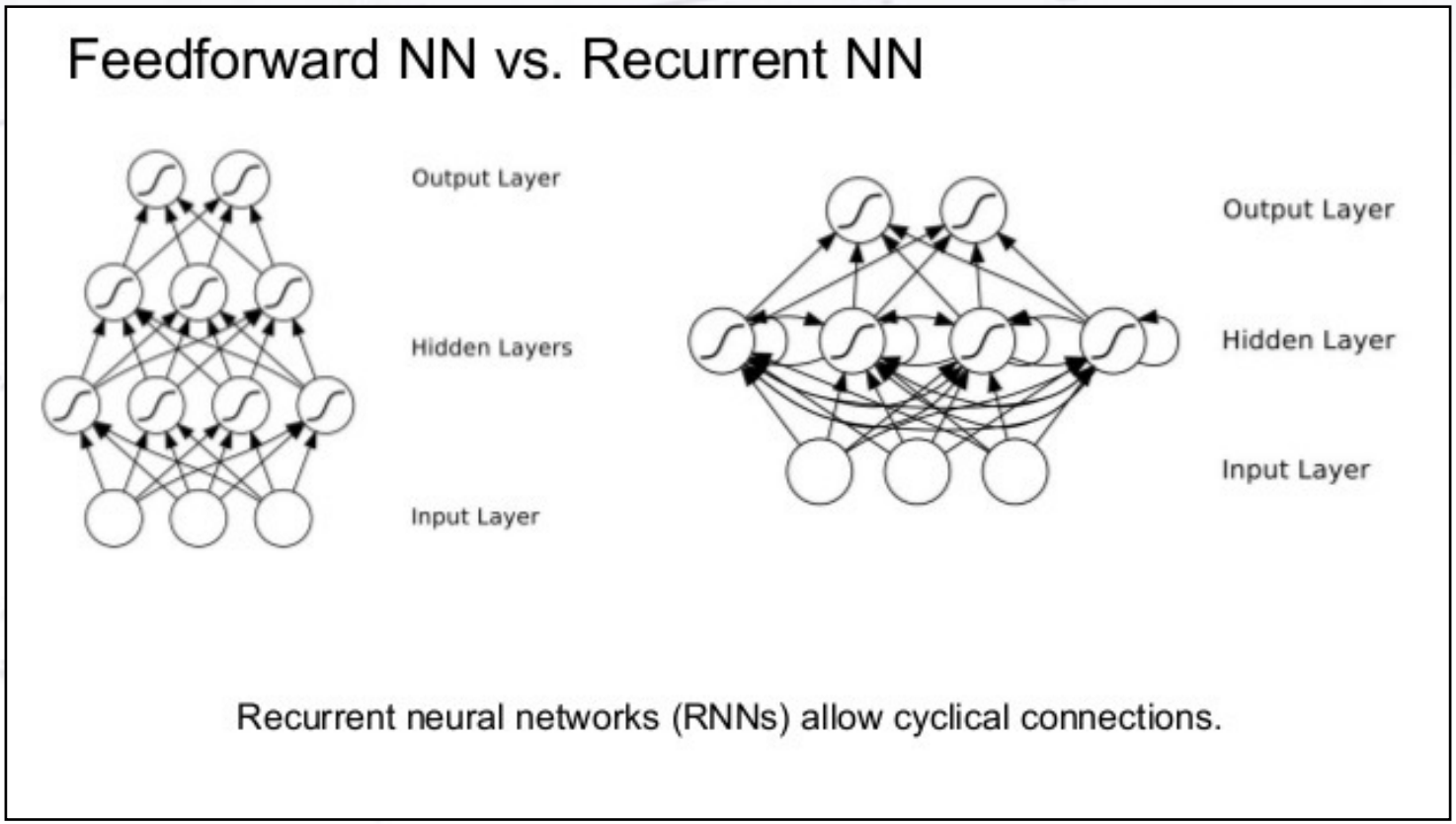
While tree based learning is invariant to (transformations of) distributions, Neural Networks are not. To avoid hard optimisation, vanishing/exploding gradients, and differential learning rates, one should normalise the input:



Recurrent NN

Normally, the information from one layer is fed forward to the next layer in a feedforward Neural Network (NN).

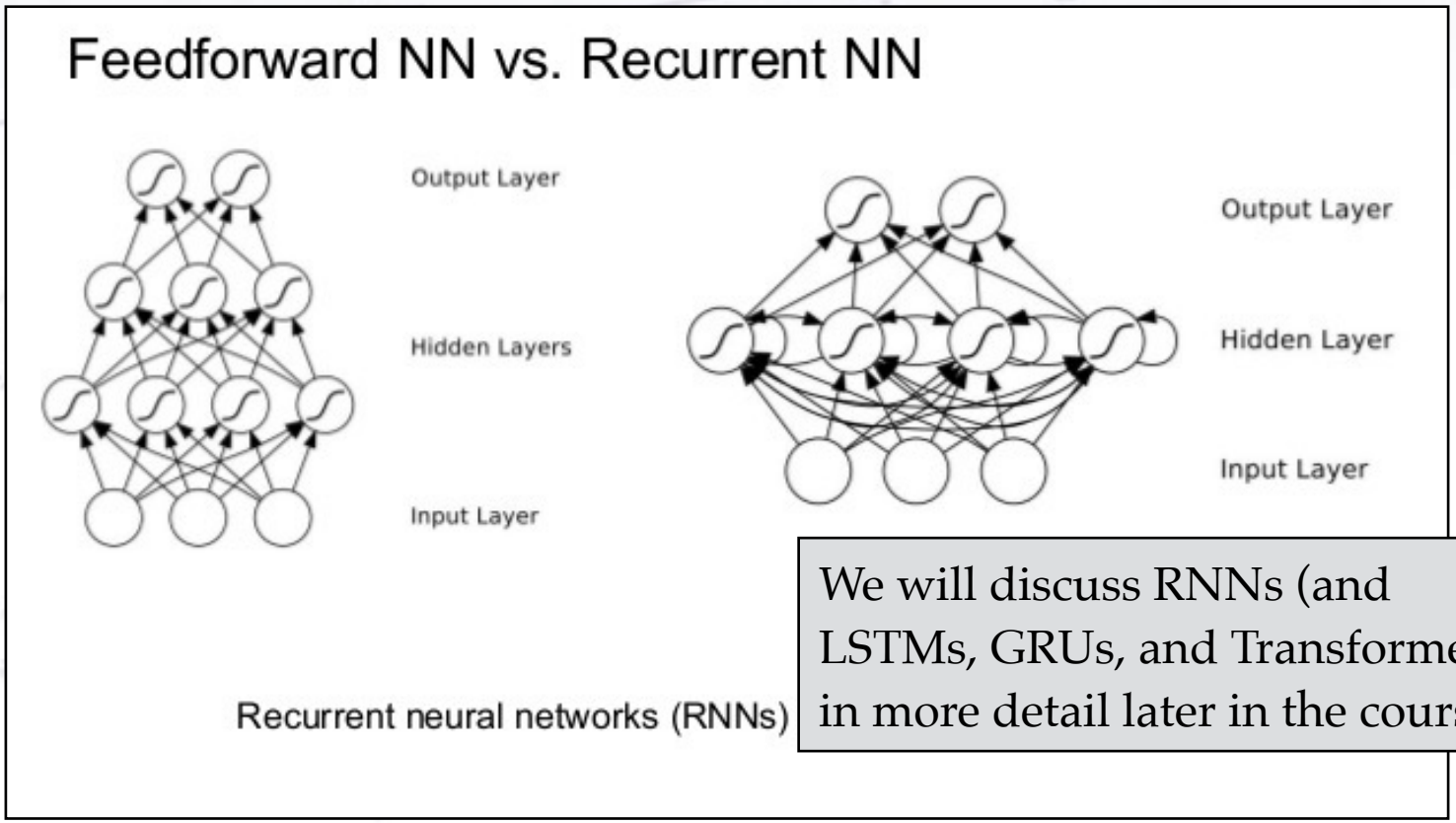
However, it may be of advantage to allow a network to give feedback, which is called a recurrent NN:



Recurrent NN

Normally, the information from one layer is fed forward to the next layer in a feedforward Neural Network (NN).

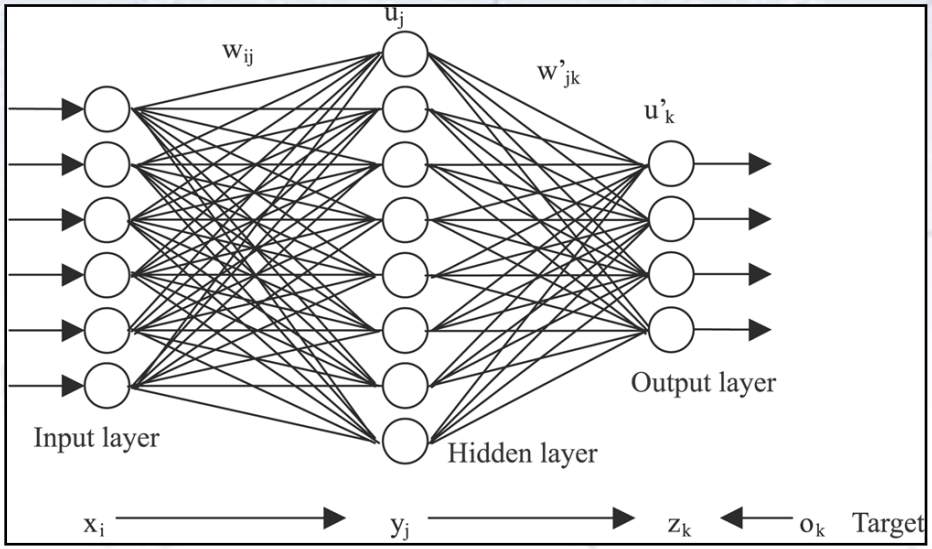
However, it may be of advantage to allow a network to give feedback, which is called a recurrent NN:



We will discuss RNNs (and LSTMs, GRUs, and Transformers) in more detail later in the course.

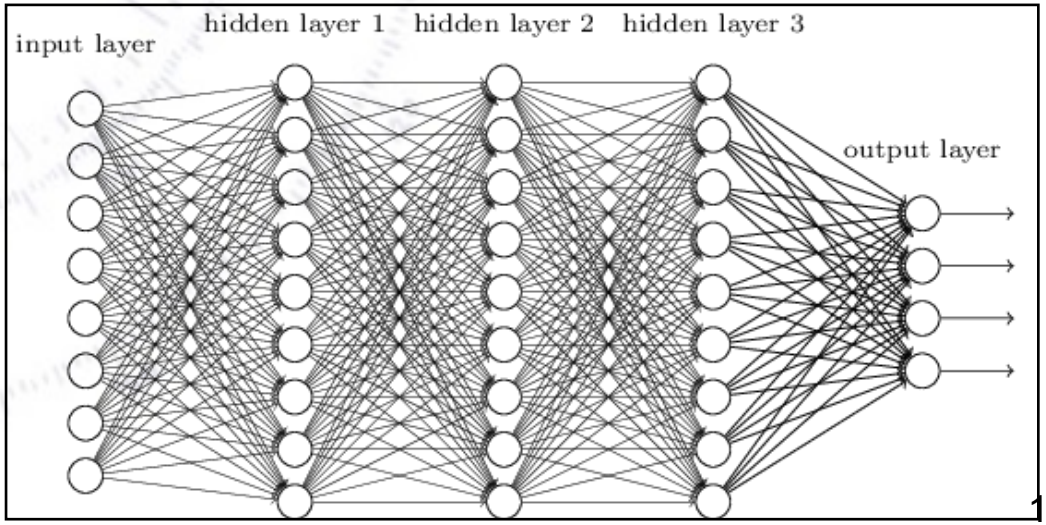
Deep Neural Networks

Deep Neural Networks (DNN) are simply (much) extended NNs in terms of layers!



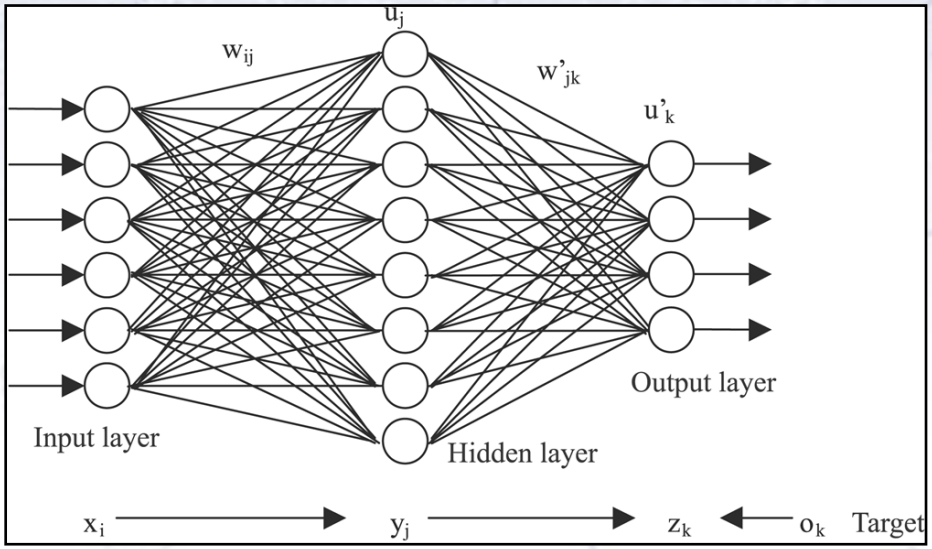
Instead of having just one (or few) hidden layers, many such layers are introduced.
This gives the network a chance to produce key features and use them for many different specialised tasks.

Currently, DNNs can have up to millions of neurons and connections, which compares to about the **brain of a worm**.



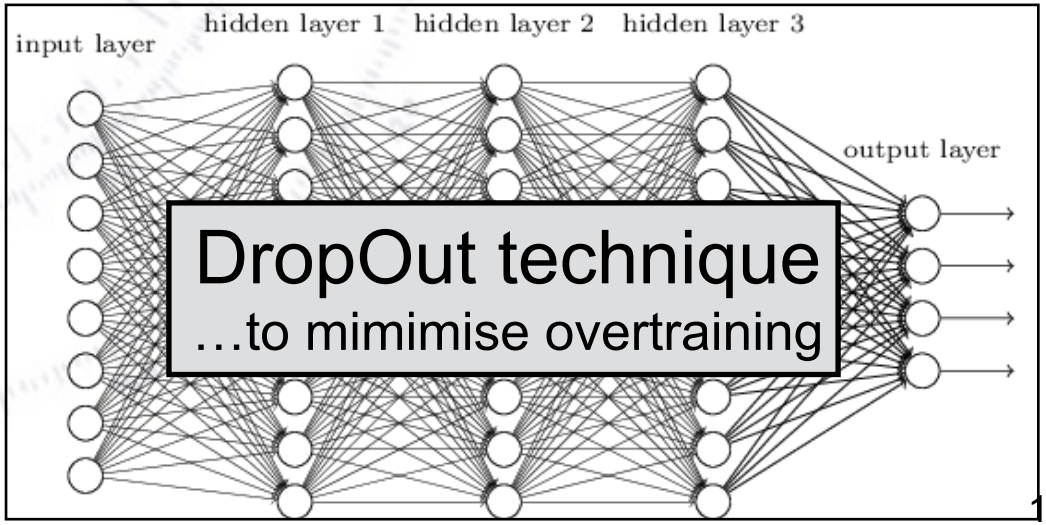
Deep Neural Networks

Deep Neural Networks (DNN) are simply (much) extended NNs in terms of layers!



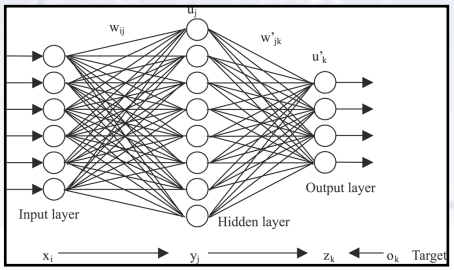
Instead of having just one (or few) hidden layers, many such layers are introduced.
This gives the network a chance to produce key features and use them for many different specialised tasks.

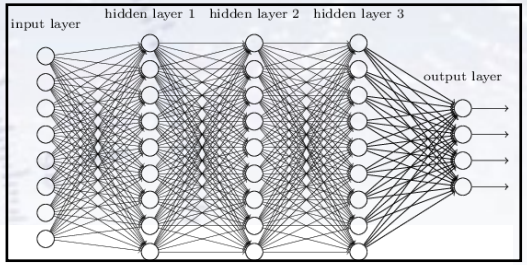
Currently, DNNs can have up to millions of neurons and connections, which compares to about the **brain of a worm**.

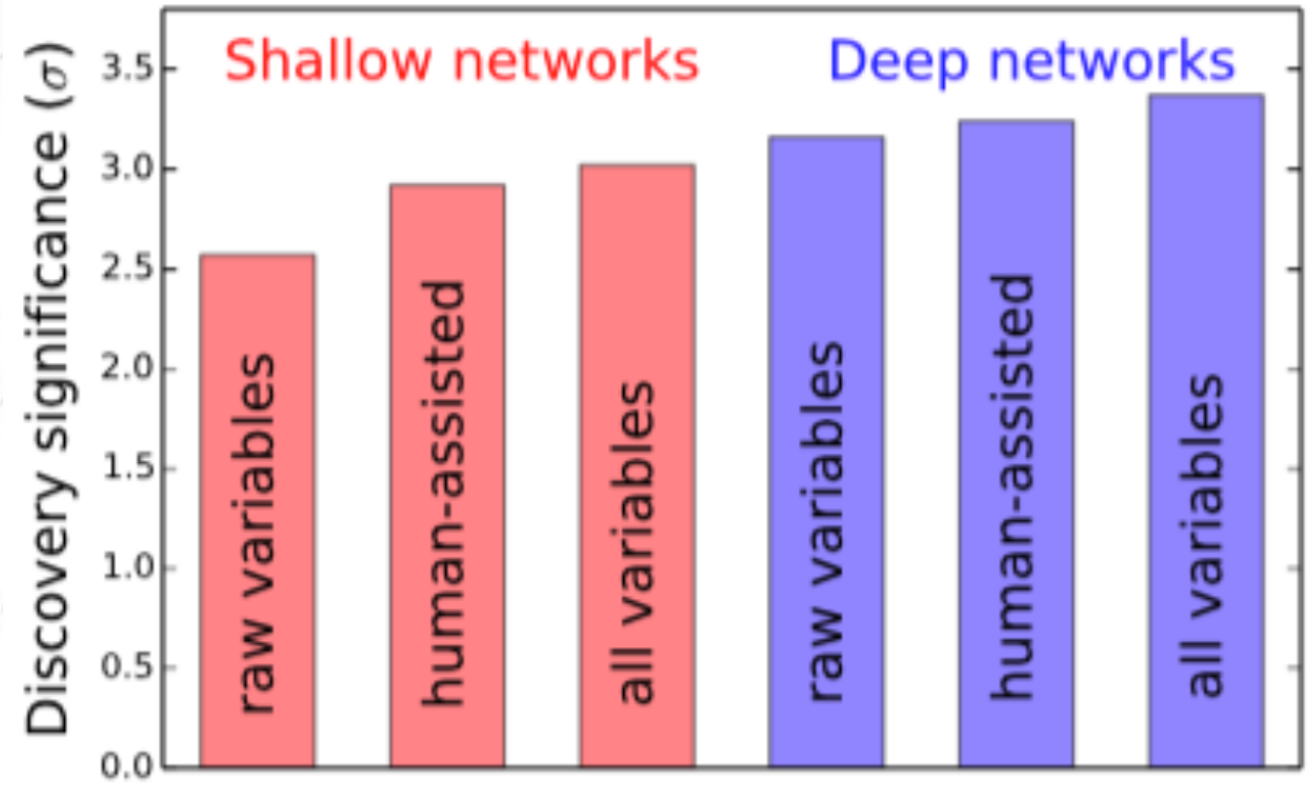


Deep Neural Networks

Deep Neural Networks likes to get both raw and "assisted" variables:

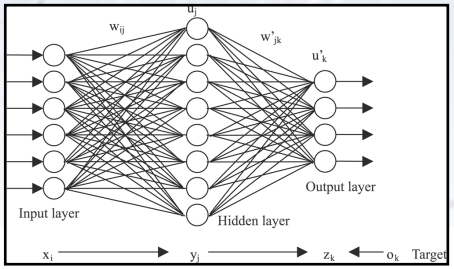


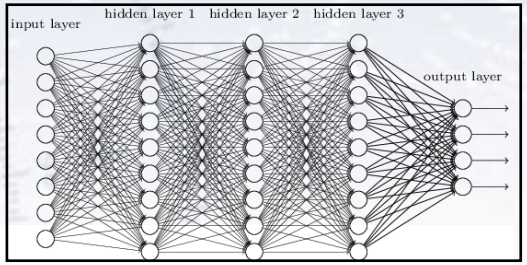


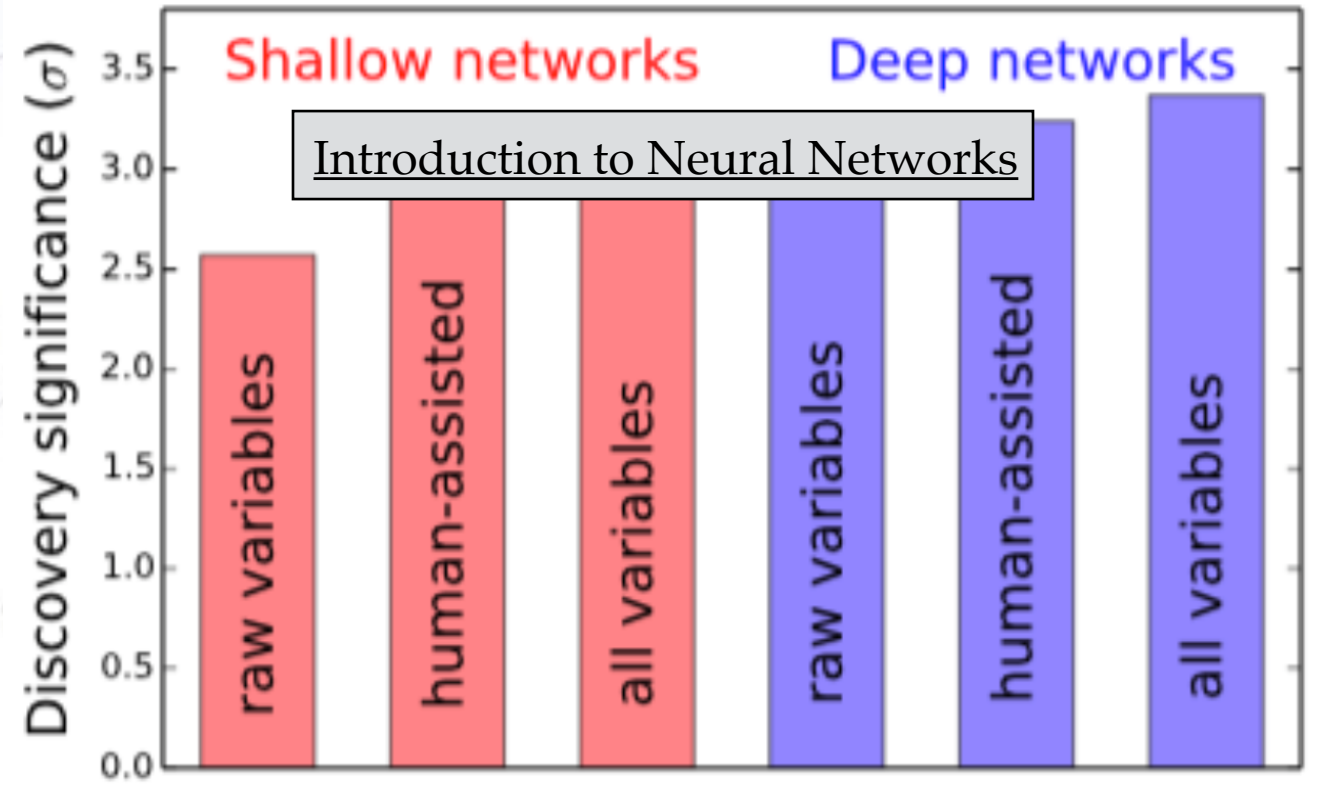


Deep Neural Networks

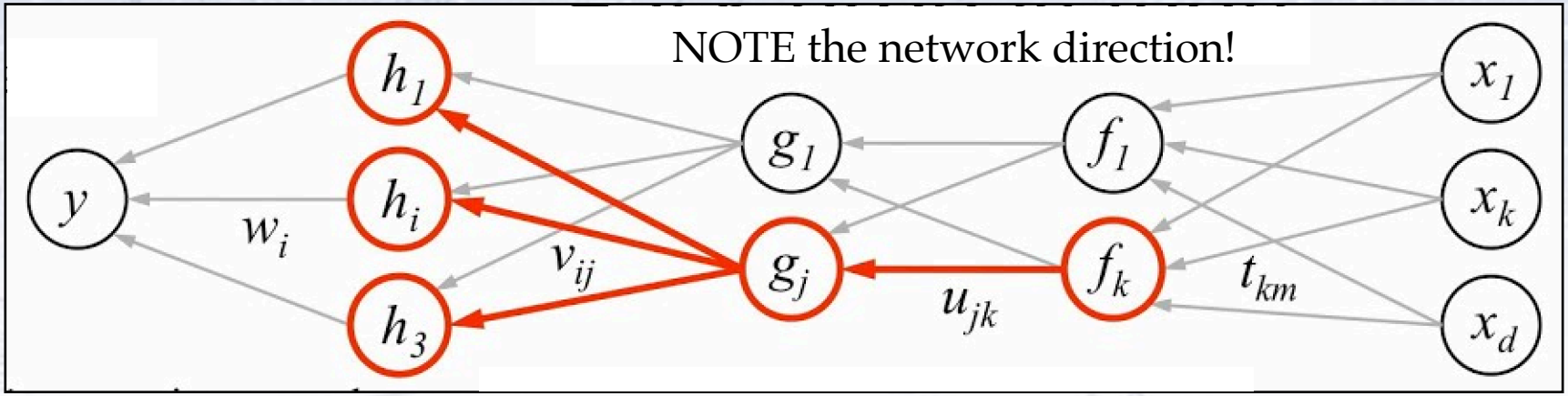
Deep Neural Networks likes to get both raw and "assisted" variables:







Back-propagation



Back-propagation for (deep) NNs consist of updating the model weights (u_{jk}) for the whole network.

Given input X (batch) calculate predictions (y) and back-propagate error y-y*.

This is done as follows for each unit g_j in each layer:

(a) compute error on g_j

$$\underbrace{\frac{\partial E}{\partial g_j}}_{\text{should } g_j \text{ be higher or lower?}} = \sum_i \underbrace{\sigma'(h_i) v_{ij}}_{\text{how } h_i \text{ will change as } g_j \text{ changes}} \underbrace{\frac{\partial E}{\partial h_i}}_{\text{was } h_i \text{ too high or too low?}}$$

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

$$\frac{\partial E}{\partial u_{jk}} = \underbrace{\frac{\partial E}{\partial g_j}}_{\text{do we want } g_j \text{ to be higher/lower}} \underbrace{\sigma'(g_j) f_k}_{\text{how } g_j \text{ will change if } u_{jk} \text{ is higher/lower}}$$

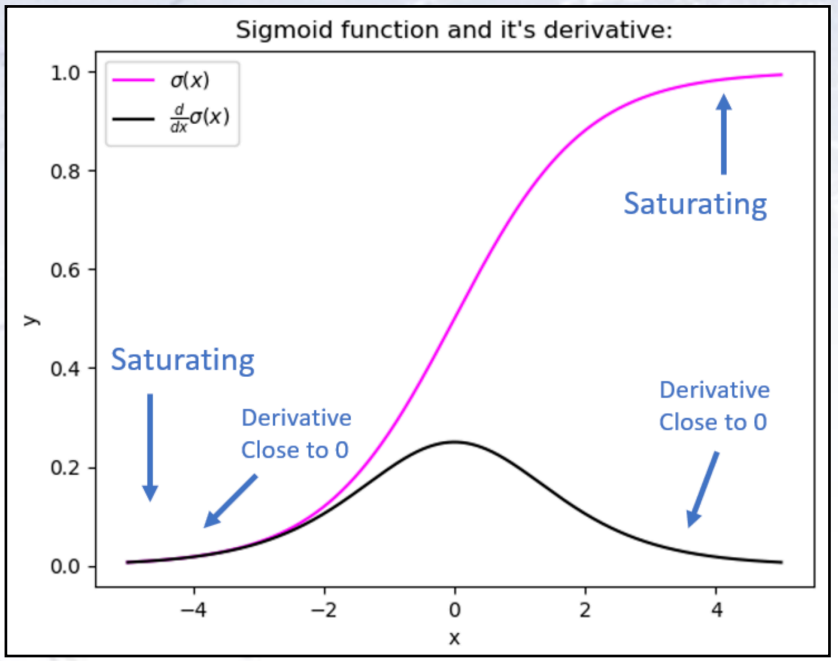
(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

Vanishing gradient problem

A problem that can occur when updating weights is the "Vanishing gradient problem".

This happens when the derivatives of the activation function tends to be small, making weights exponentially smaller for every layer in the network. The opposite problem is called "exploding gradient problem".



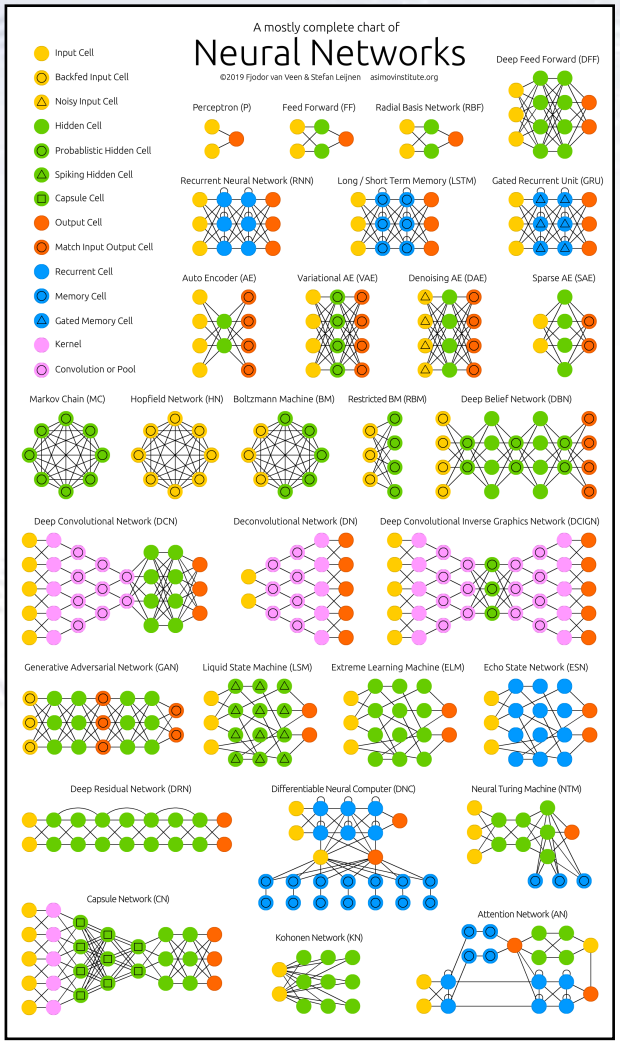
The solutions are many:

- Another activation function: RELU or (especially) Leaky RELU.
- Batch normalisation: Scaling input values to have mean 0 and variance 1. (It is still debated exactly how this functions!)
- Residual (or skip) connections: Replacing f(x) by f(x) + x (x "skips").
- Weight initialisation: Doing this "smart" given network architecture.
- Gradient clipping: Dividing gradients by g_{max}, if gradients exceeds g_{max}.

The role of NNs in ML

The reason why NNs play such a central role is that they are versatile:

- Recurrent NNs (for time series)
- Convolutional NNs (for images)
- Adversarial NNs (for simulation)
- Graph NNs (for geometric data)
- etc.



Unlike trees, NNs typically make the "foundation" of all the more advanced ML paradigms. However, they are harder to optimise!
This is why trees a great for simpler tasks (i.e. data that typically fits into an excel sheet [2110.01889]), while NNs are typically used for the more advanced.

Have this in mind, when you attack problems with ML - and like any other project or analysis, it is typically good to get a "rough result" fast, and then to refine it from there.