Neural Networks

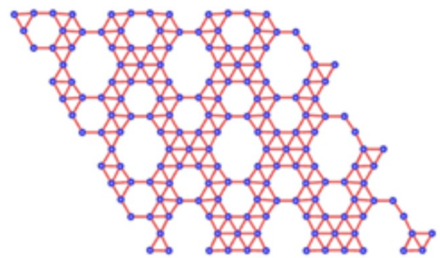
Vlad Gladkikh

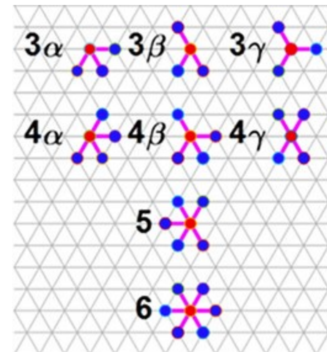
IBS CMCM

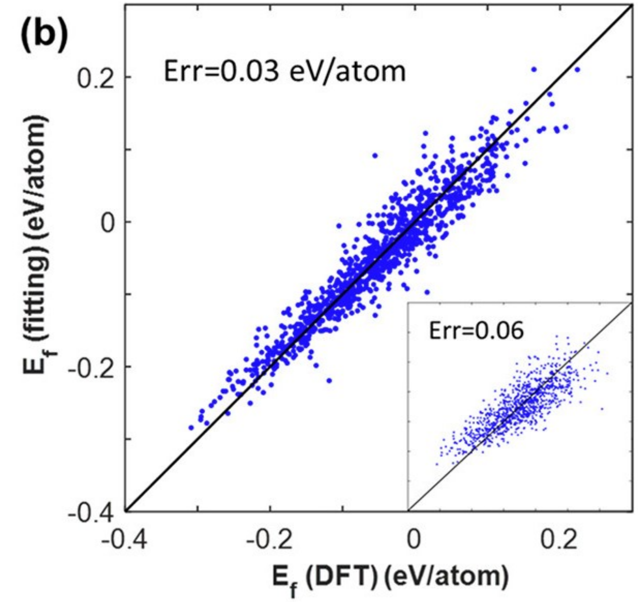
Borophene with Large Holes

Yong Wang, Yunjae Park, Lu Qiu, Izaac Mitchell, and Feng Ding*

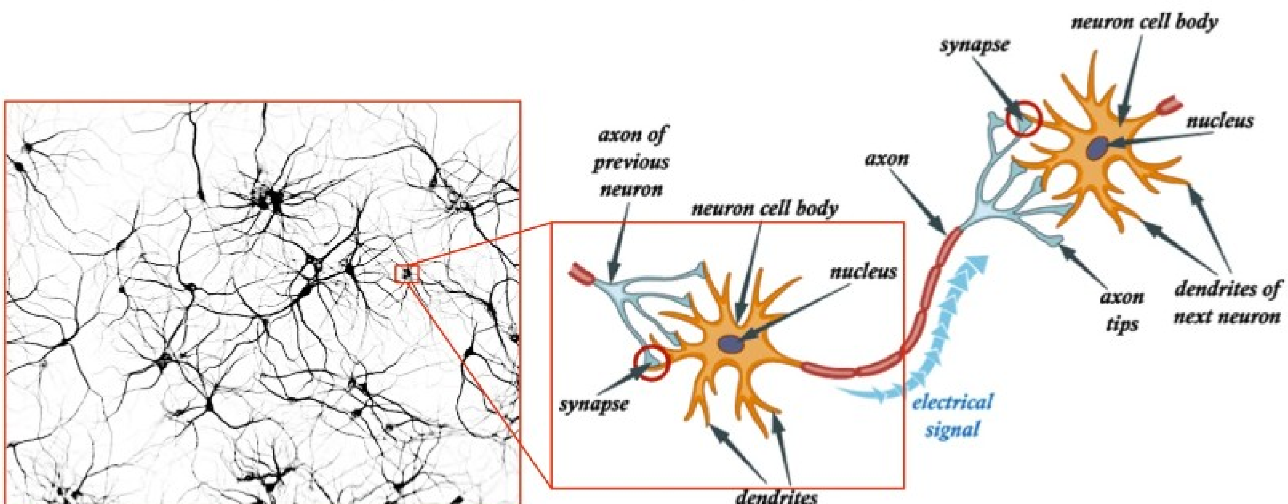
J. Phys. Chem. Lett. 2020, 11, 6235−6241



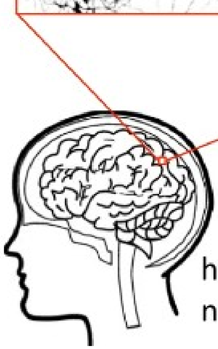




Neurons and the brain



Data science is easy if you know the right features.



humans don't need features

Lecture 20 of the Introductory Applied Machine Learning (IAML) course at the University of Edinburgh, taught by Victor Lavrenko

https://www.youtube.com/playlist?list=PLBv09BD7ez_4Bs9j3o8l_ZTjQZoN_3Oqs

An artificial neural network consists of a number of interconnected processors: **neurons**.

The neurons are connected by **weighted links** passing signals to one another.

Connectionism

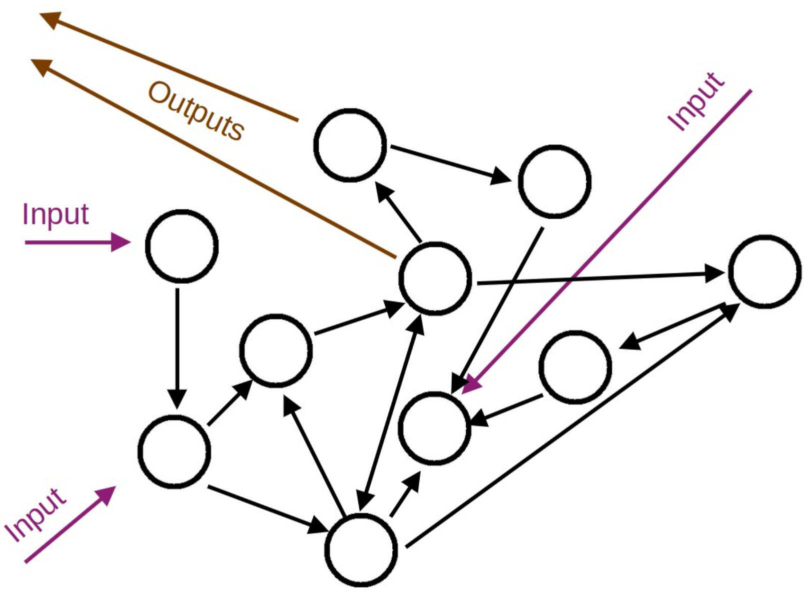
Describe complicated phenomena by interconnected networks of simple units.

Self-organization

The structure of a neural network is only partially predetermined.

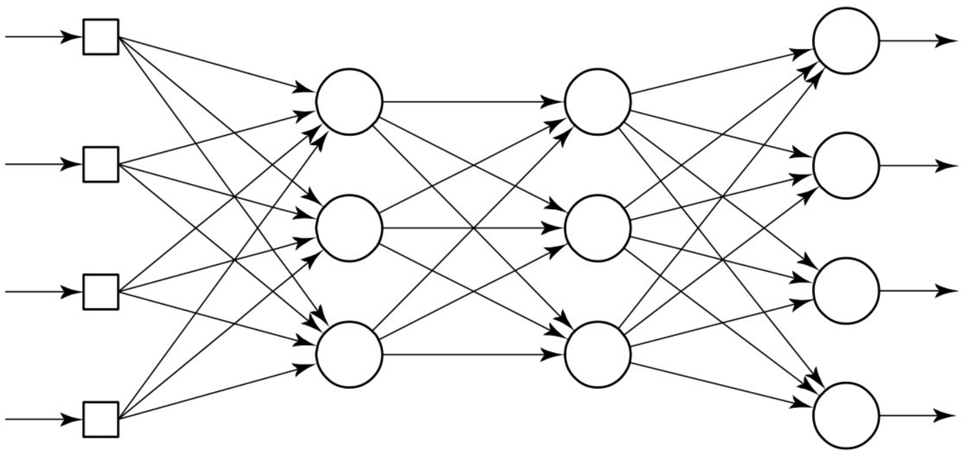
Neither totally undetermined, nor fully determined!

This is a cute little network designed by me...



This is a poor creature from almost every textbook and blog on neural networks

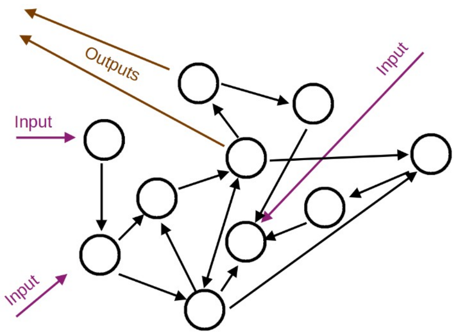
- ogranized into layers
- topology is fixed
- all neurons are same

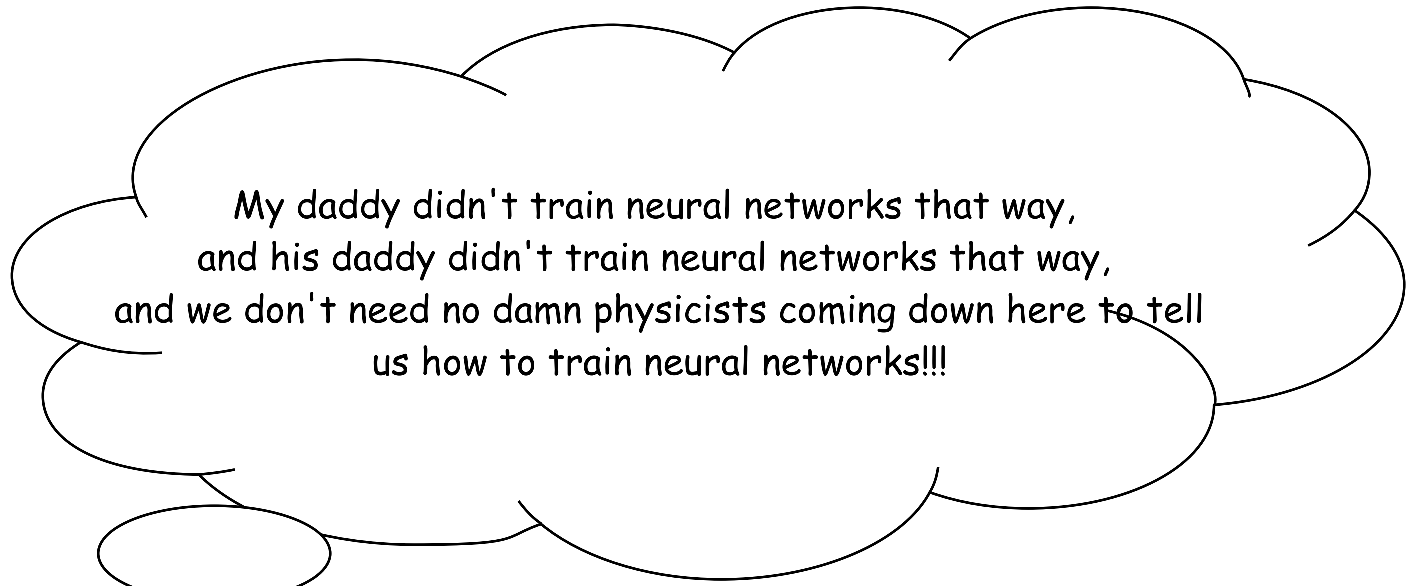


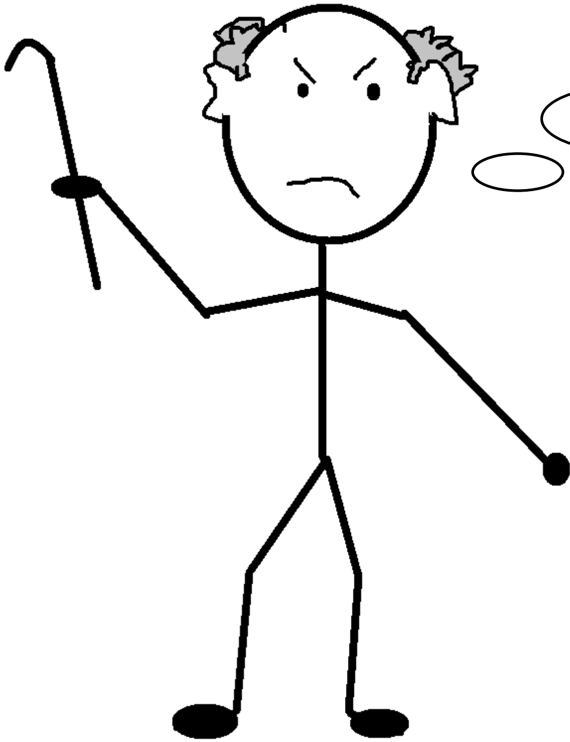
https://towardsdatascience.com/randomly-wired-neural-networks-92098dbd5175

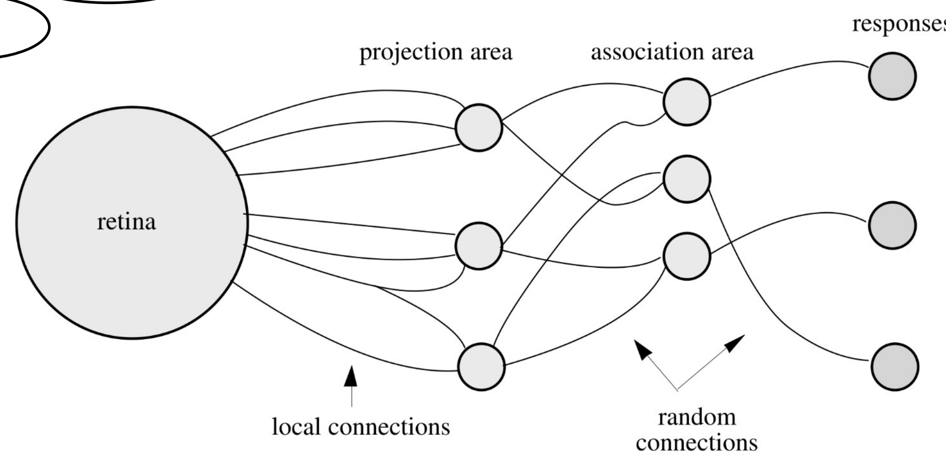
https://stats.stackexchange.com/questions/135035/can-a-neural-network-with-random-connections-still-work-correctly

Some dogmas here...



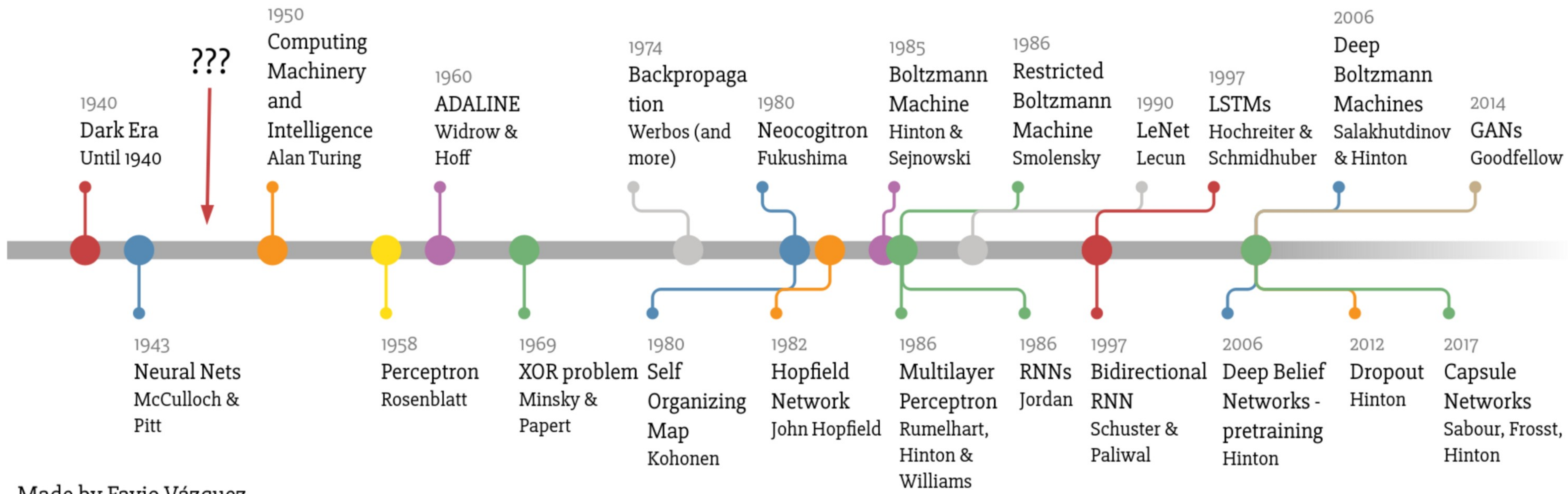






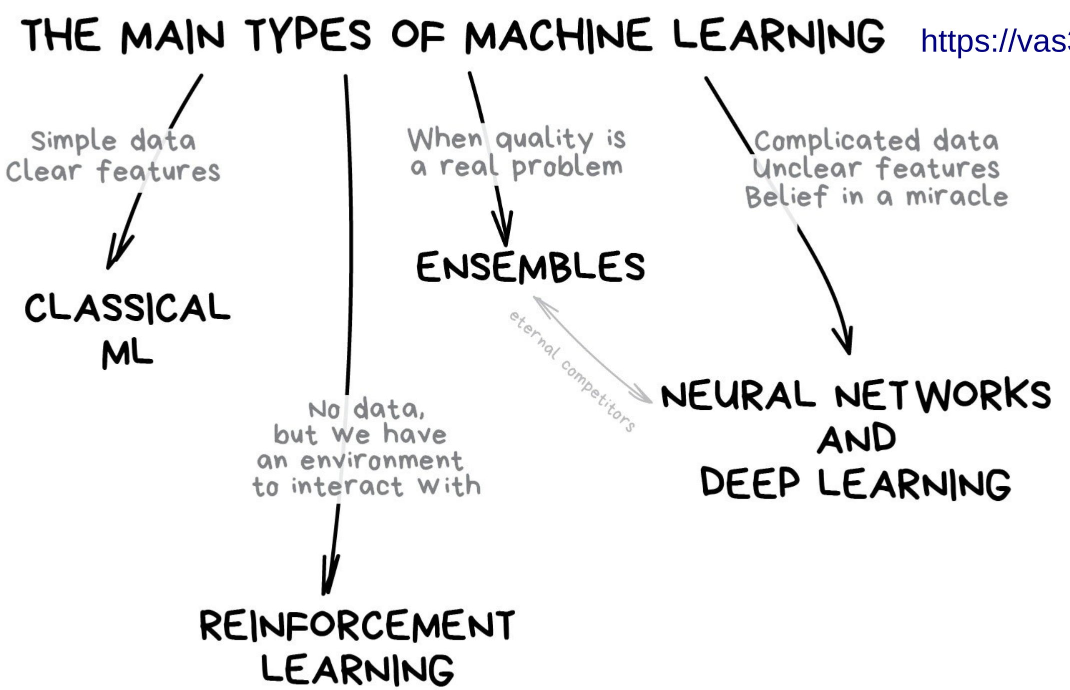
The classical perceptron [after Rosenblatt 1958]

It is difficult to believe that birds can fly
if you only study ostriches.

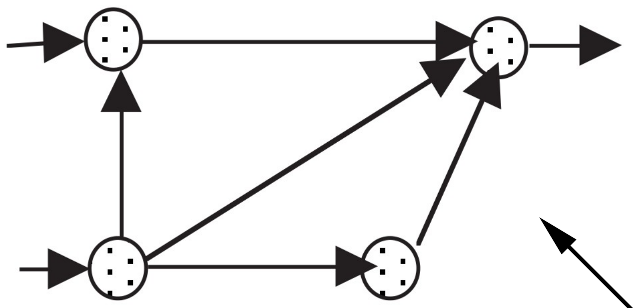


Made by Favio Vázquez

https://towardsdatascience.com/a-weird-introduction-to-deep-learning-7828803693b0



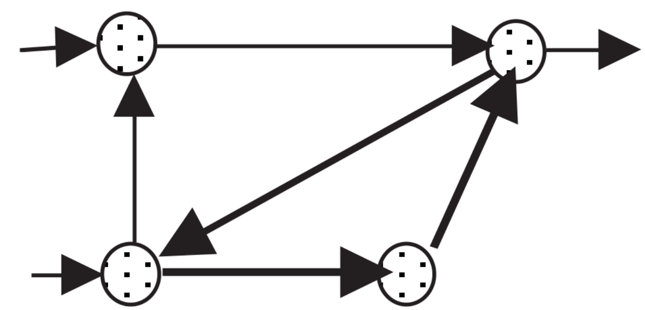
https://vas3k.com/blog/machine_learning/



Acyclic topology

No loops

– approximates a nonlinear mapping between its inputs and outputs



Cyclic topology

aka a **recurrent network** (RNN)

Due to the feedback loop, a recurrent network contains internal memory.

How does a neural network ‘learn’

Learning involves modification of the network parameters

weights are just regression coefficients

– weights

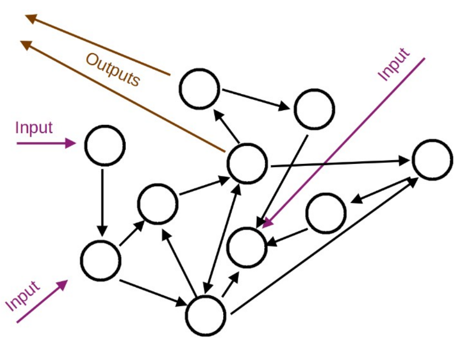
number of neurons
which neurons are connected

– topology

functions

– or neurons themselves

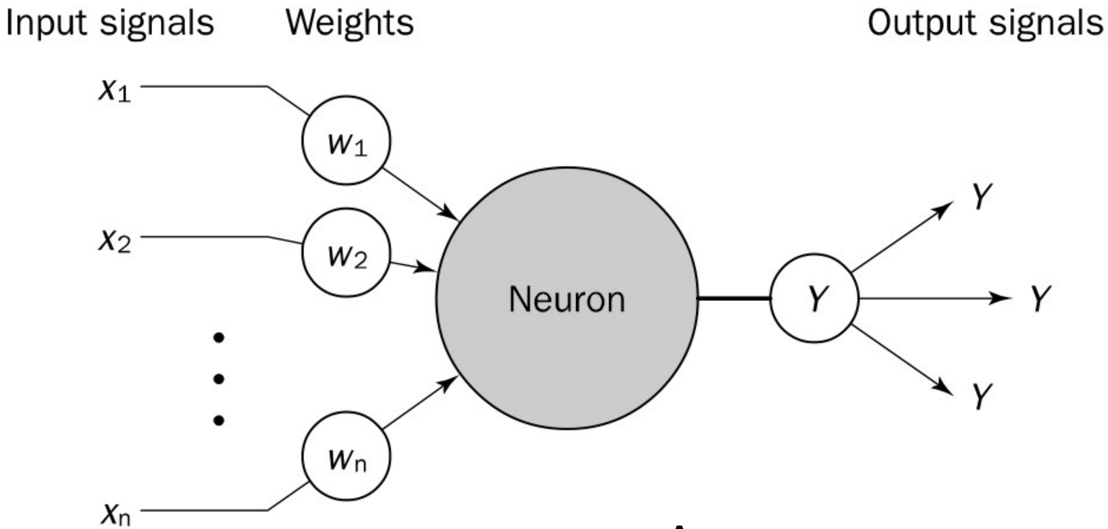
by applying a set of training examples.



The training of the network is repeated for many examples in the set until the network reaches a steady state where there are no further significant changes in its parameters.

The previously applied training examples may be reapplied during the training session but in a different order.

Any mathematically defined change in ANN parameters over time is referred to as the **learning algorithm**.

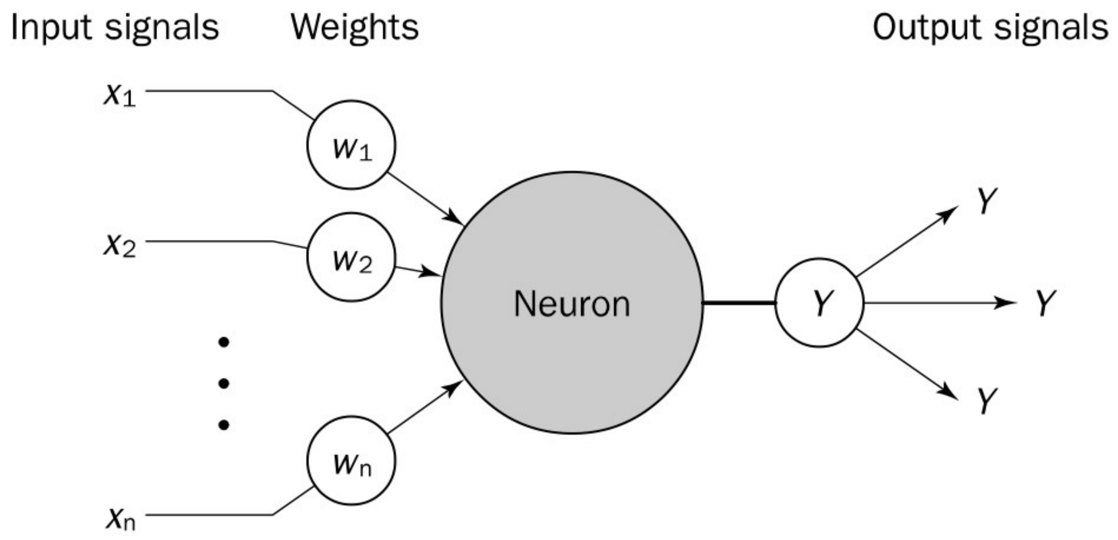


A neuron

– receives input signals
– computes a new activation level
– sends it as an output signal through the output links

The input signal can be raw data or outputs of other neurons.

The output signal can be either a final solution to the problem or an input to other neurons.



$$Y = f\left(w_1, w_2, \ldots, w_n; x_1, x_2, \ldots, x_n\right)$$

f – the **primitive function**

can be any function

Usually, f is a superposition of a **transfer function** and an **activation function**.

$$z = z\left(w_1, w_2, \ldots, w_n; x_1, x_2, \ldots, x_n\right)$$ – **transfer function**

– aggregates all inputs and weights into a small set of numbers (usually a single number)

e.g., $z = \sum_{i=1}^{n} w_i x_i - \theta$ which is called a linear combiner.

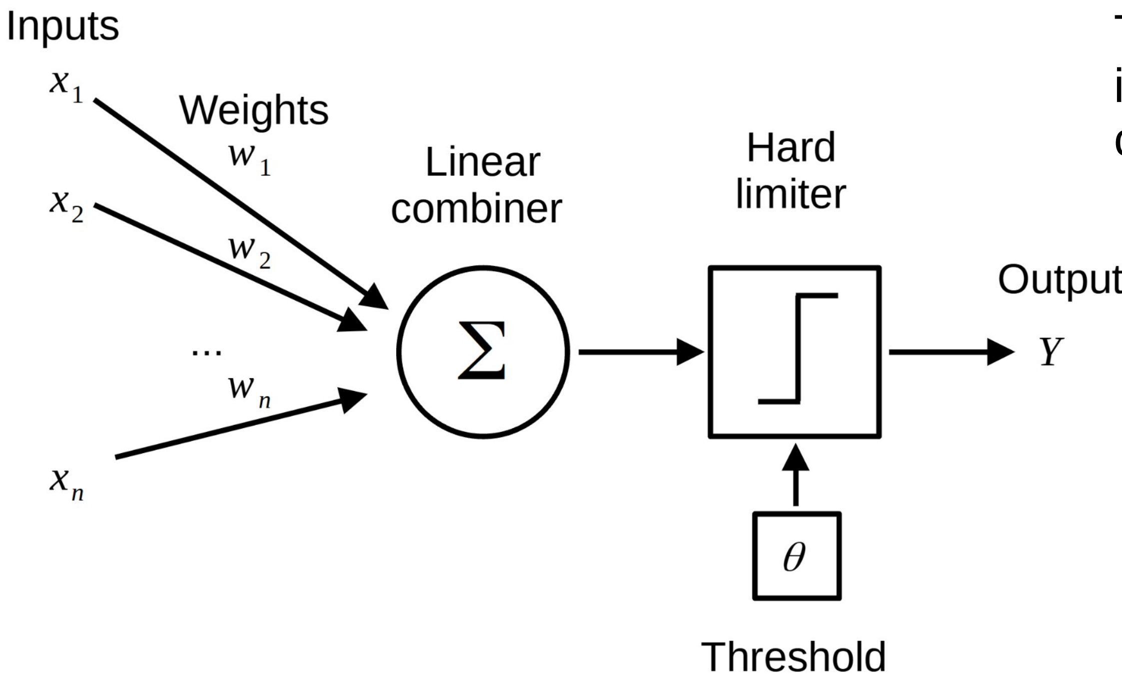
$$Y = \sigma\left(z\right)$$ – **activation function**

– a non-linear transformation of the aggregated input

The transfer and activation functions may introduce some symmetry (desired or unwanted).

The output of the neuron will be the same if $z\left(w; x\right) = z\left(w; \tilde{x}\right)$ or if $\sigma\left(z\right) = \sigma\left(\tilde{z}\right)$

Perceptron



The aim of the perceptron is to classify inputs, x_1, x_2, ..., x_n, into one of two classes, say A_1 and A_2.

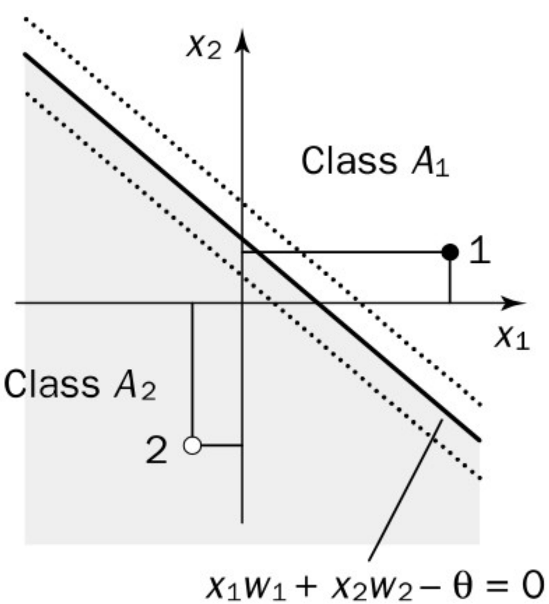
The n-dimensional input space is divided by a hyperplane into two decision regions.

The hyperplane is defined by the linear function

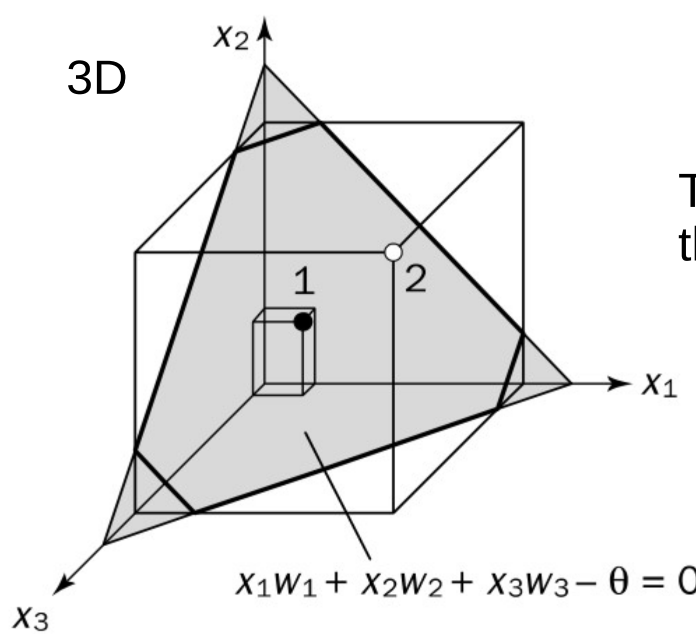
$$z = \sum_{i=1}^{n} w_i x_i - \theta$$

The threshold θ shifts the decision boundary.

2D



3D



The **perceptron learning rule**

A perceptron learns by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron.

The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples.

https://makeyourownneuralnetwork.blogspot.com/2016/01/why-squared-error-cost-function.html

error

If at iteration p, $e(p) = Y_d - Y(p)$

desired output actual output

If $e(p) > 0$, we need to increase perceptron output $Y(p)$
If $e(p) < 0$, we need to decrease $Y(p)$

Each perceptron input contributes $x_i(p) w_i(p)$ to the total input $X(p) \Rightarrow$

If $x_i(p) > 0$, an increase in its weight $w_i(p)$ tends to increase perceptron output $Y(p)$, whereas if $x_i(p) < 0$, an increase in $w_i(p)$ tends to decrease $Y(p) \Rightarrow$

$$w_i(p+1) = w_i(p) + \alpha\, x_i(p)\, e(p)$$

where α is the **learning rate**, a positive constant

$$w_i(p+1)=w_i(p)+\alpha\, x_i(p)e(p)$$ – a linear function of error

It can be extended to any non-decreasing function

$$w_i(p+1)=w_i(p)+x_i(p)\sum_{j=0}^{\infty}\alpha_j e(p)^{2j+1}$$

which effectively means that the learning rate is not constant but depends on the error:

$$w_i(p+1)=w_i(p)+\alpha(e(p))\, x_i(p)e(p)$$

$$\alpha(e(p))=\sum_{j=0}^{\infty}\alpha_j e(p)^{2j}$$

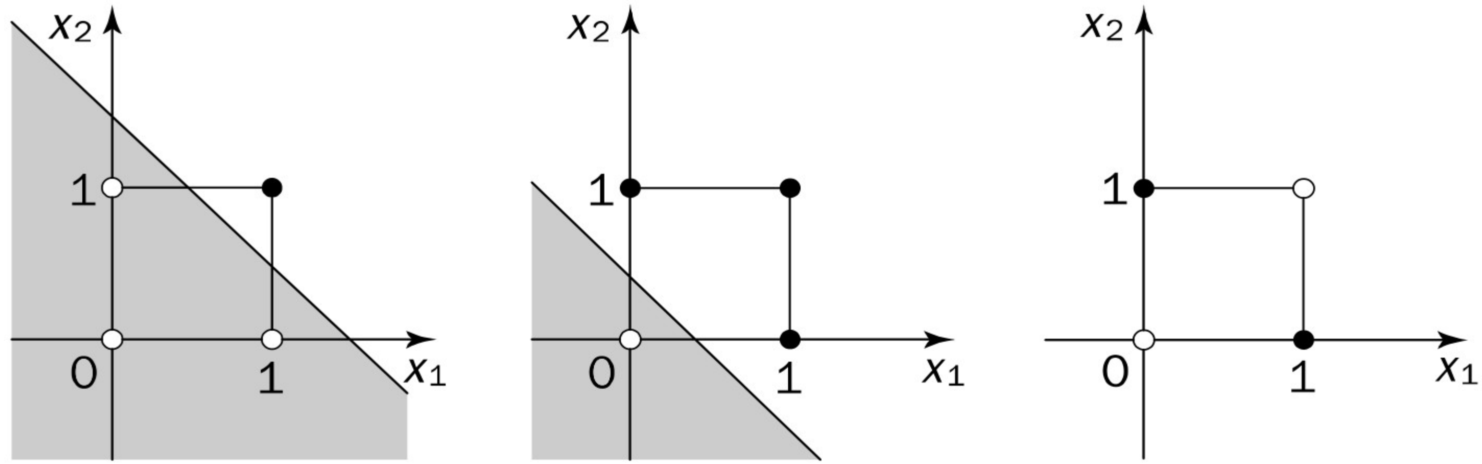
where the coefficients α_j are such that α is a non-negative function.

We can also add a more general dependence on x to make the learning rate depend on certain values of the inputs:

$$w_i(p+1)=w_i(p)+\alpha(e(p),x_1(p),\ldots,x_n(p))\, x_i(p)e(p)$$

where α is a non-negative function

Perceptron performing logical operations

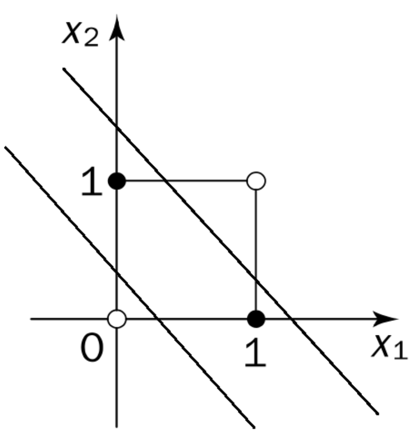


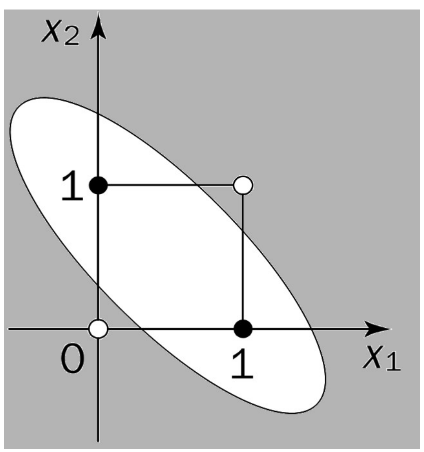
AND ($x_1 \cap x_2$) OR ($x_1 \cup x_2$) Exclusive-OR ($x_1 \oplus x_2$)

Separable by a single straight line aka **linearly separable**

Not linearly separable

Either multiple straight lines (multiple neurons) or non-linear transfer function, e.g.





$$z=\sum_{i=1}^{n} w_i x_i + \sum_{i,j=1}^{n} w_{ij} x_i x_j - \theta$$

or use different activation function

See e.g. here:

https://ai.stackexchange.com/questions/9417/why-cant-the-xor-linear-inseparability-problem-be-solved-with-one-perceptron/

https://keras.io/api/layers/

Search Keras documentation...

- metrics property
- dynamic property

Layer activations

- relu function
- sigmoid function
- softmax function
- softplus function
- softsign function
- tanh function
- selu function
- elu function
- exponential function

Layer weight initializers

- RandomNormal class
- RandomUniform class
- TruncatedNormal class

usual?

traditional?

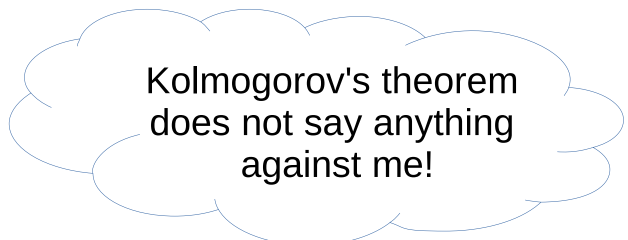
venerated?

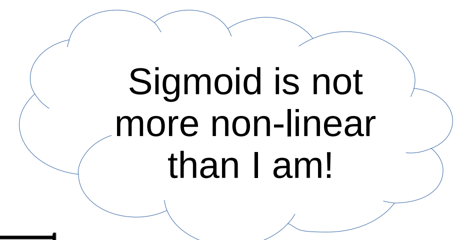
sacred?

"Usual" activation functions

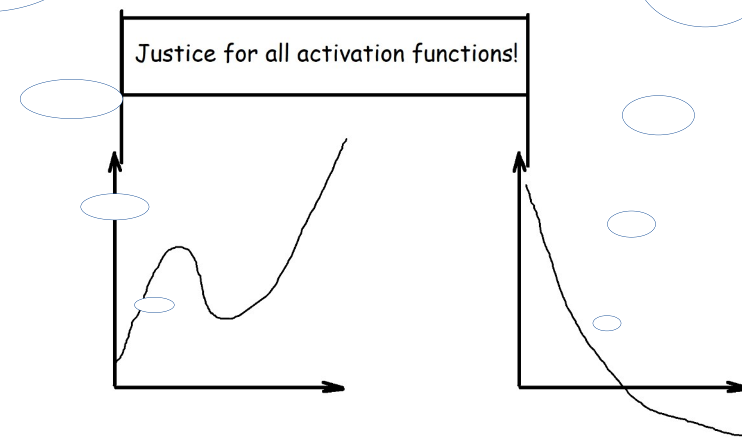
https://en.wikipedia.org/wiki/Activation_function

Binary step	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)	$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ [1]
TanH	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
Rectified linear unit (ReLU)[11]	$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$



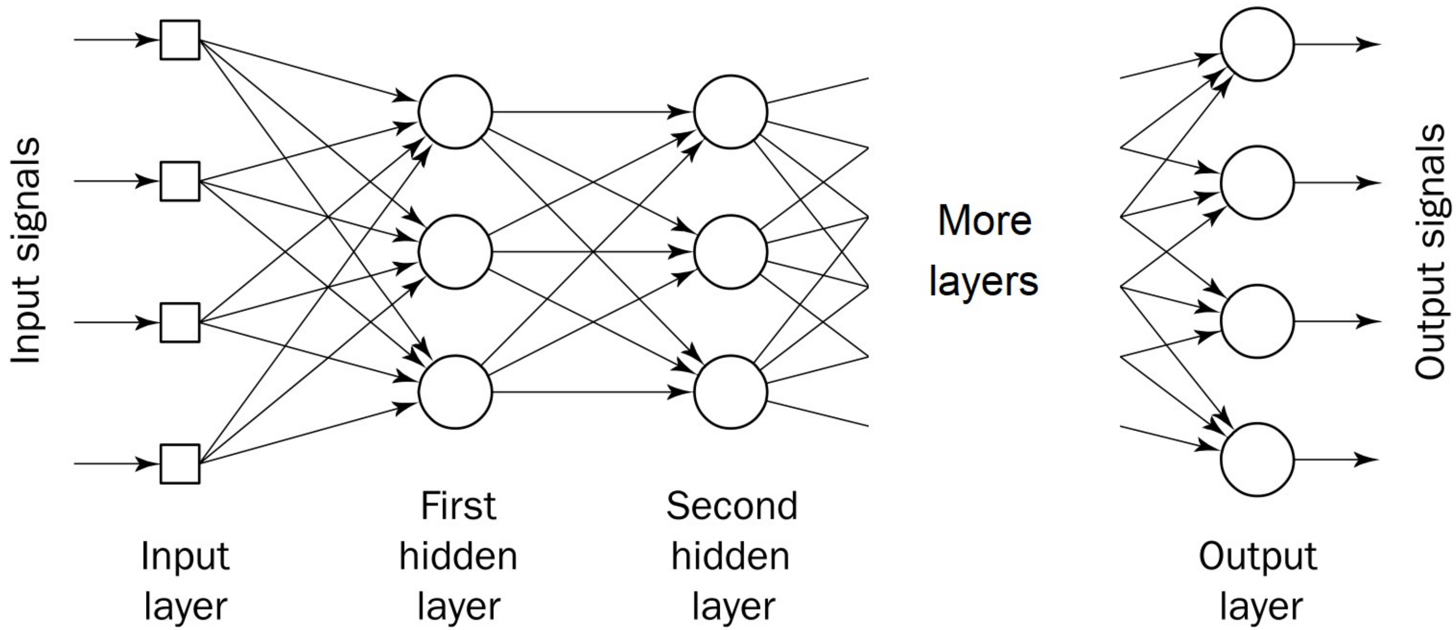


Non-traditional activation functions are protesting:



https://ai.stackexchange.com/questions/24117/smallest-possible-network-to-approximate-the-sin-function
https://datascience.stackexchange.com/questions/58838/can-we-learn-fx-1-x-using-a-neural-network-exactly
https://stats.stackexchange.com/questions/361066/what-is-the-point-of-having-a-dense-layer-in-a-neural-network-with-no-activation

Multilayer perceptron



How to update weights when more than one node contributes to an output and its error?

We don't know how much each link contributes to the total error but we can approximate the probability distribution of the error contributions among the links.

The probability that only one link of many was responsible for the error is extremely small.

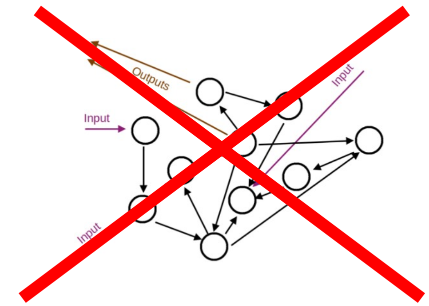
Links with larger weights contribute more to the error.

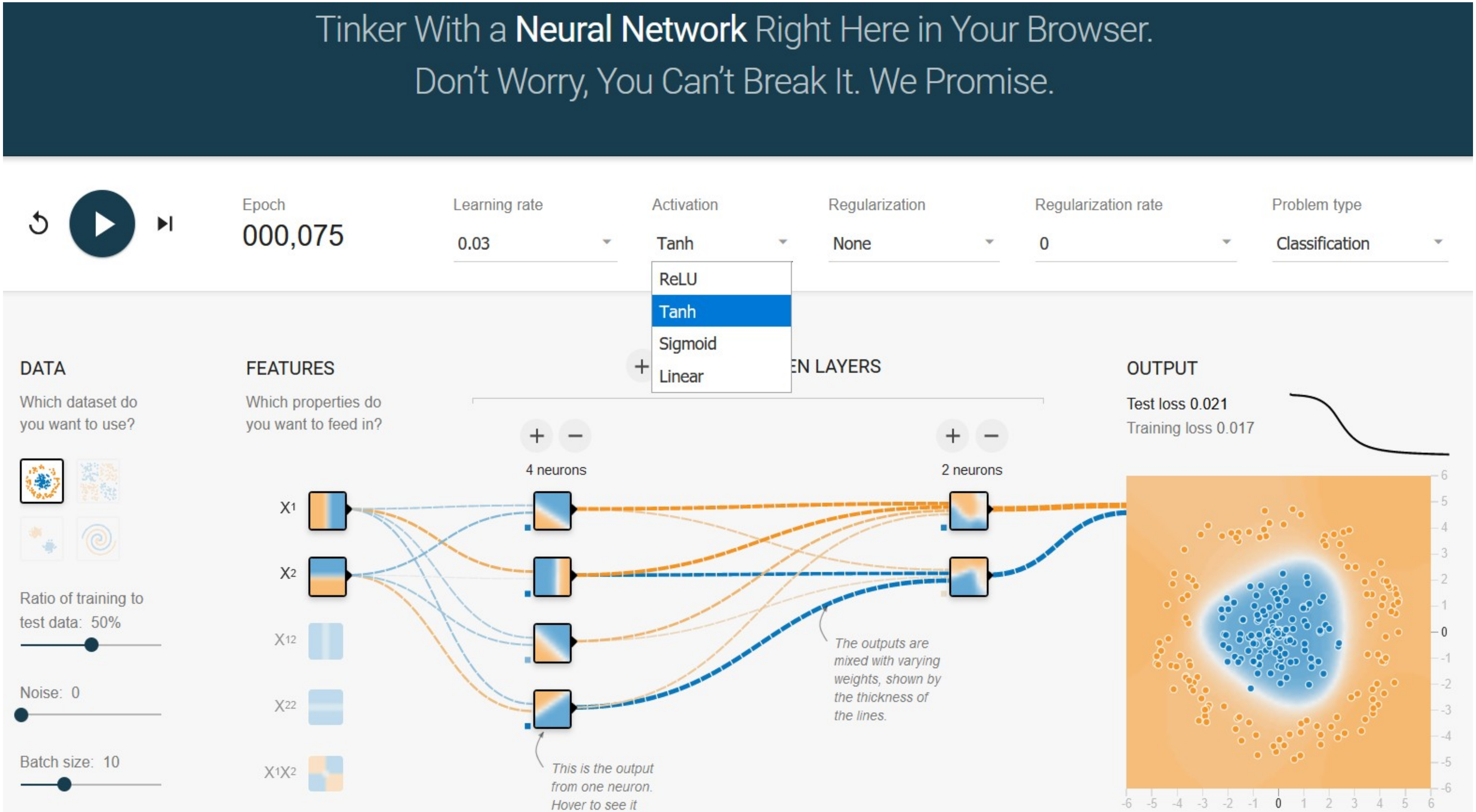
Larger output of a neuron → larger error

Sharper change through the neuron → larger error

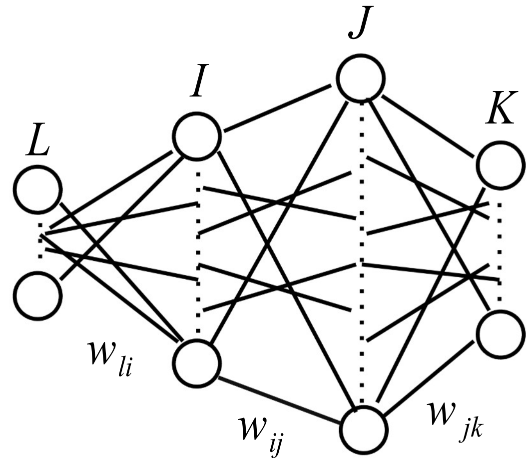
The only neuron that never makes a mistake is the neuron who never does anything.

http://playground.tensorflow.org





The back-propagation algorithm: derivation



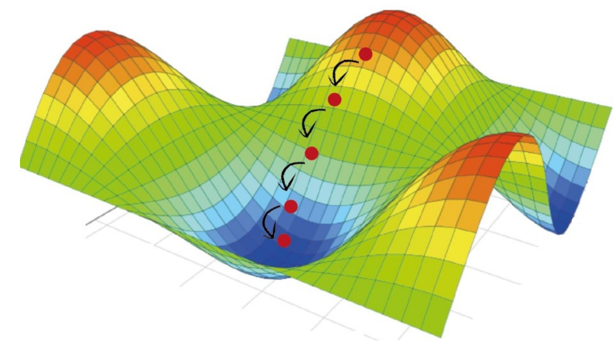
$$w_{ab} \rightarrow w_{ab} + \Delta w_{ab}$$

Gradient descent:

$$\Delta w_{ab} = -\eta \frac{\partial E}{\partial w_{ab}}$$

$$E = \sum_{k\in K} E_k$$

$$o_b = \sigma_b(z_b)$$



$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k}\frac{\partial \sigma_k(z_k)}{\partial w_{jk}} = \frac{\partial E}{\partial o_k}\sigma'_k\frac{\partial z_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = \delta_k\frac{\partial z_k}{\partial w_{jk}}$$

where $\delta_k = \sigma'_k \frac{\partial E}{\partial o_k}$

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k\in K}\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{ij}} = \sum_{k\in K}\frac{\partial E}{\partial o_k}\frac{\partial \sigma_k(z_k)}{\partial w_{ij}} = \sum_{k\in K}\frac{\partial E}{\partial o_k}\sigma'_k\frac{\partial z_k}{\partial w_{ij}} = \sum_{k\in K}\delta_k\frac{\partial z_k}{\partial w_{ij}}$$

$$\frac{\partial z_k}{\partial w_{ij}} = \frac{\partial z_k}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial z_k}{\partial o_j}\frac{\partial \sigma_j(z_j)}{\partial w_{ij}} = \frac{\partial z_k}{\partial o_j}\sigma'_j\frac{\partial z_j}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k\in K}\delta_k\frac{\partial z_k}{\partial o_j}\sigma'_j\frac{\partial z_j}{\partial w_{ij}} = \delta_j\frac{\partial z_j}{\partial w_{ij}}$$

where $\delta_j = \sigma'_j \sum_{k\in K}\delta_k\frac{\partial z_k}{\partial o_j}$

$$\frac{\partial E}{\partial w_{li}} = \sum_{k\in K}\delta_k\frac{\partial z_k}{\partial w_{li}} = \sum_{k\in K}\delta_k\sum_{j\in J}\frac{\partial z_k}{\partial o_j}\sigma'_j\frac{\partial z_j}{\partial w_{li}} = \sum_{j\in J}\delta_j\frac{\partial z_j}{\partial w_{li}} = \sum_{j\in J}\delta_j\frac{\partial z_j}{\partial o_i}\sigma'_i\frac{\partial z_i}{\partial w_{li}}$$

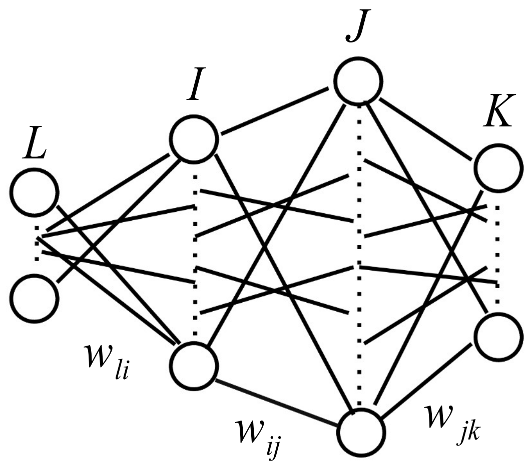
$$\frac{\partial E}{\partial w_{li}} = \delta_i\frac{\partial z_i}{\partial w_{li}}$$

where $\delta_i = \sigma'_i \sum_{j\in J}\delta_j\frac{\partial z_j}{\partial o_i}$

The back-propagation algorithm: summary

$$E = \sum_{k \in K} E_k$$

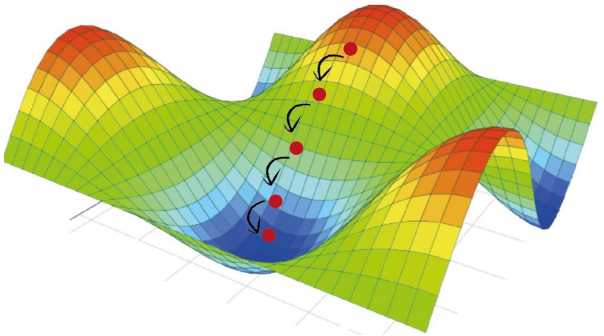
$$o_b = \sigma_b(z_b)$$



$$\delta_k = \sigma'_k \frac{\partial E}{\partial o_k} \qquad \frac{\partial E}{\partial w_{jk}} = \delta_k \frac{\partial z_k}{\partial w_{jk}}$$

$$\delta_j = \sigma'_j \sum_{k \in K} \delta_k \frac{\partial z_k}{\partial o_j} \qquad \frac{\partial E}{\partial w_{ij}} = \delta_j \frac{\partial z_j}{\partial w_{ij}}$$

$$\delta_i = \sigma'_i \sum_{j \in J} \delta_j \frac{\partial z_j}{\partial o_i} \qquad \frac{\partial E}{\partial w_{li}} = \delta_i \frac{\partial z_i}{\partial w_{li}}$$



Gradient descent: $\Delta w_{ab} = -\eta \frac{\partial E}{\partial w_{ab}}$

$$w_{ab} \rightarrow w_{ab} + \Delta w_{ab}$$

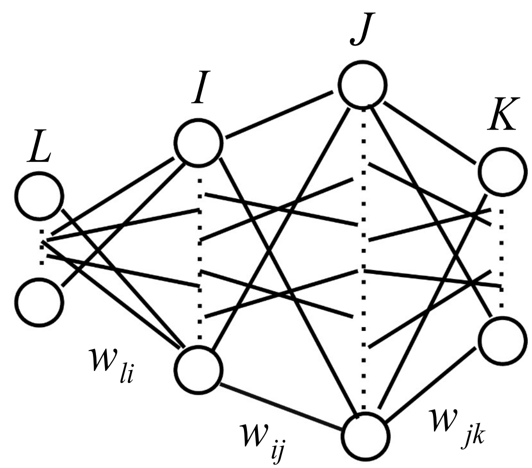
If the inputs are aggregated via a linear combiner, then

$$z_b = \sum_a w_{ab} o_a + \theta_b \qquad \frac{\partial z_b}{\partial w_{ab}} = o_a \qquad \frac{\partial z_b}{\partial o_a} = w_{ab} \qquad \frac{\partial z_b}{\partial \theta_b} = 1$$

$$\delta_k = \sigma'_k \frac{\partial E}{\partial o_k} \qquad \frac{\partial E}{\partial w_{jk}} = o_j \delta_k \qquad \frac{\partial E}{\partial \theta_k} = \delta_k$$

$$\delta_j = \sigma'_j \sum_{k \in K} \delta_k w_{jk} \qquad \frac{\partial E}{\partial w_{ij}} = o_i \delta_j \qquad \frac{\partial E}{\partial \theta_j} = \delta_j$$

$$\delta_i = \sigma'_i \sum_{j \in J} \delta_j w_{ij} \qquad \frac{\partial E}{\partial w_{li}} = o_l \delta_i \qquad \frac{\partial E}{\partial \theta_i} = \delta_i$$



$$o_b = \sigma_b(z_b)$$

$$E = \sum_{k \in K} E_k$$

$$\Delta w_{ab} = -\eta \frac{\partial E}{\partial w_{ab}} \qquad w_{ab} \rightarrow w_{ab} + \Delta w_{ab}$$

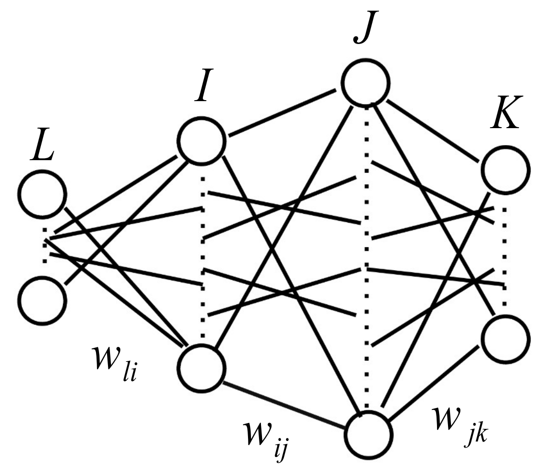
$$\Delta \theta_b = -\eta \frac{\partial E}{\partial \theta_b} \qquad \theta_{ab} \rightarrow \theta_b + \Delta \theta_b$$

Correction = error * activation slope * prev. layer output

$$\delta_k = \sigma'_k \frac{\partial E}{\partial o_k} \qquad \frac{\partial E}{\partial w_{jk}} = o_j \delta_k$$

$$\delta_j = \sigma'_j \sum_{k \in K} \delta_k w_{jk} \qquad \frac{\partial E}{\partial w_{ij}} = o_i \delta_j$$

$$\delta_i = \sigma'_i \sum_{j \in J} \delta_j w_{ij} \qquad \frac{\partial E}{\partial w_{li}} = o_l \delta_i$$



$$o_b = \sigma_b(z_b)$$

$$z_b = \sum_a w_{ab} o_a$$

$$E = \sum_{k \in K} E_k$$

$$\Delta w_{ab} = -\eta \frac{\partial E}{\partial w_{ab}} \qquad w_{ab} \rightarrow w_{ab} + \Delta w_{ab}$$

Correction = error * activation slope * prev. layer output

Neural networks are trained in a series of **epochs**.

An epoch is one forward pass and one back-propagation pass over all training samples.

Full batch learning The average of the gradients of all the training examples is used in order to update the weights.

– we move (almost) directly towards an optimal solution

Online learning A single example is used to update the weights.

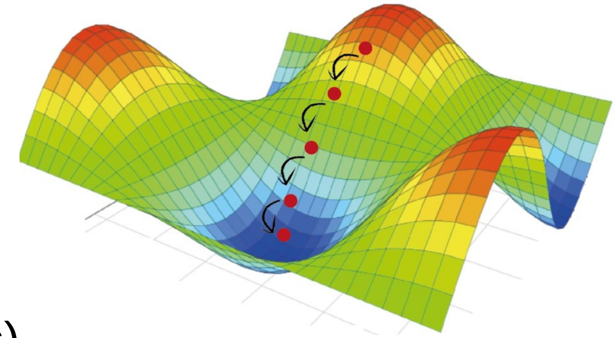
Approximations to the gradient

Mini-batch learning The training set is split into mini-batches. An update is made using the mean-gradient of the mini-batch.

https://ai.stackexchange.com/questions/19894/are-there-any-commonly-used-discontinuous-activation-functions

https://ai.stackexchange.com/questions/17609/in-deep-learning-is-it-possible-to-use-discontinuous-activation-functions

Error minimization is a global optimization problem



Problems:

Local minima (not a big problem for large-size networks)

Anna Choromanska et al. The Loss Surfaces of Multilayer Networks https://arxiv.org/abs/1412.0233v3

LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. Nature 521, 436–444 (2015). https://doi.org/10.1038/nature14539

Saddle points (a bigger problem)

Yann Dauphin et al., Identifying and attacking the saddle point problem in high-dimensional non-convex optimization https://arxiv.org/abs/1406.2572

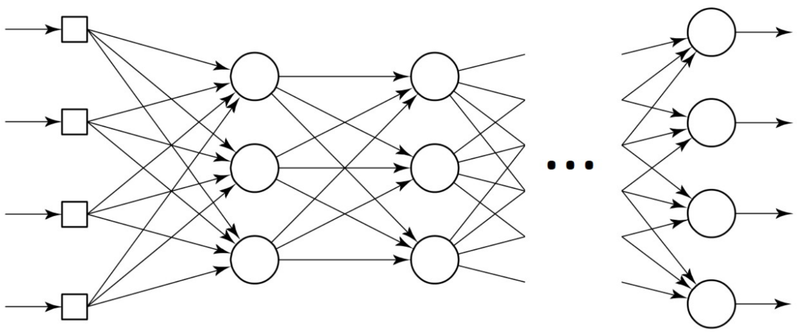
For fairly small networks

Mavrovouniotis, M., Yang, S. Training neural networks with ant colony optimization algorithms for pattern classification. Soft Comput 19, 1511–1522 (2015). https://doi.org/10.1007/s00500-014-1334-5

They show that ant colony optimization + backprop beats unmodified backprop on several benchmark data sets (albeit not by much).

Liao, SH., Hsieh, JG., Chang, JY. et al. Training neural networks via simplified hybrid algorithm mixing Nelder–Mead and particle swarm optimization methods. Soft Comput 19, 679–689 (2015). https://doi.org/10.1007/s00500-014-1292-y

Search for the best hyperparameters is another global optimization problem



How many layers?

How many neurons in each leayer?

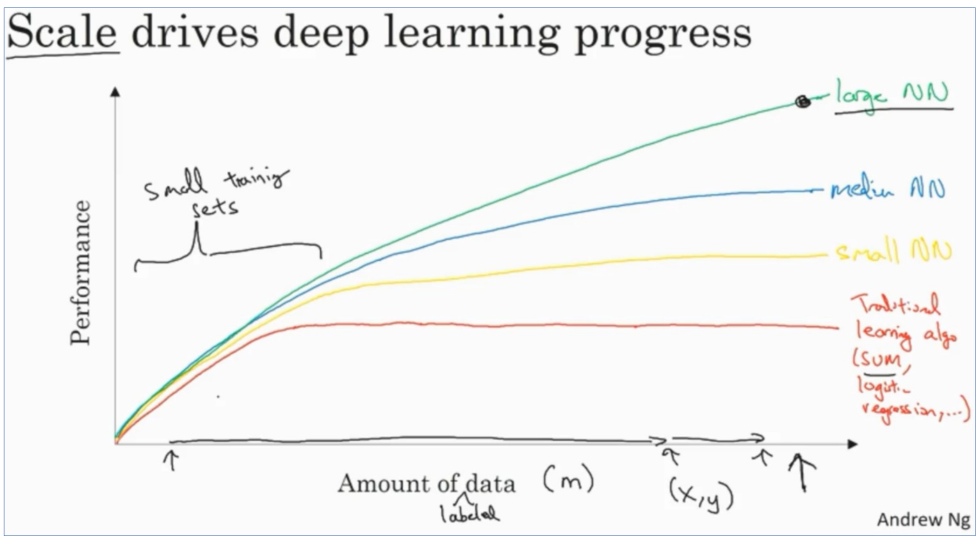
Which activation function?

Learning rate?

Training algorithm?

Batch size?

Any regularization?

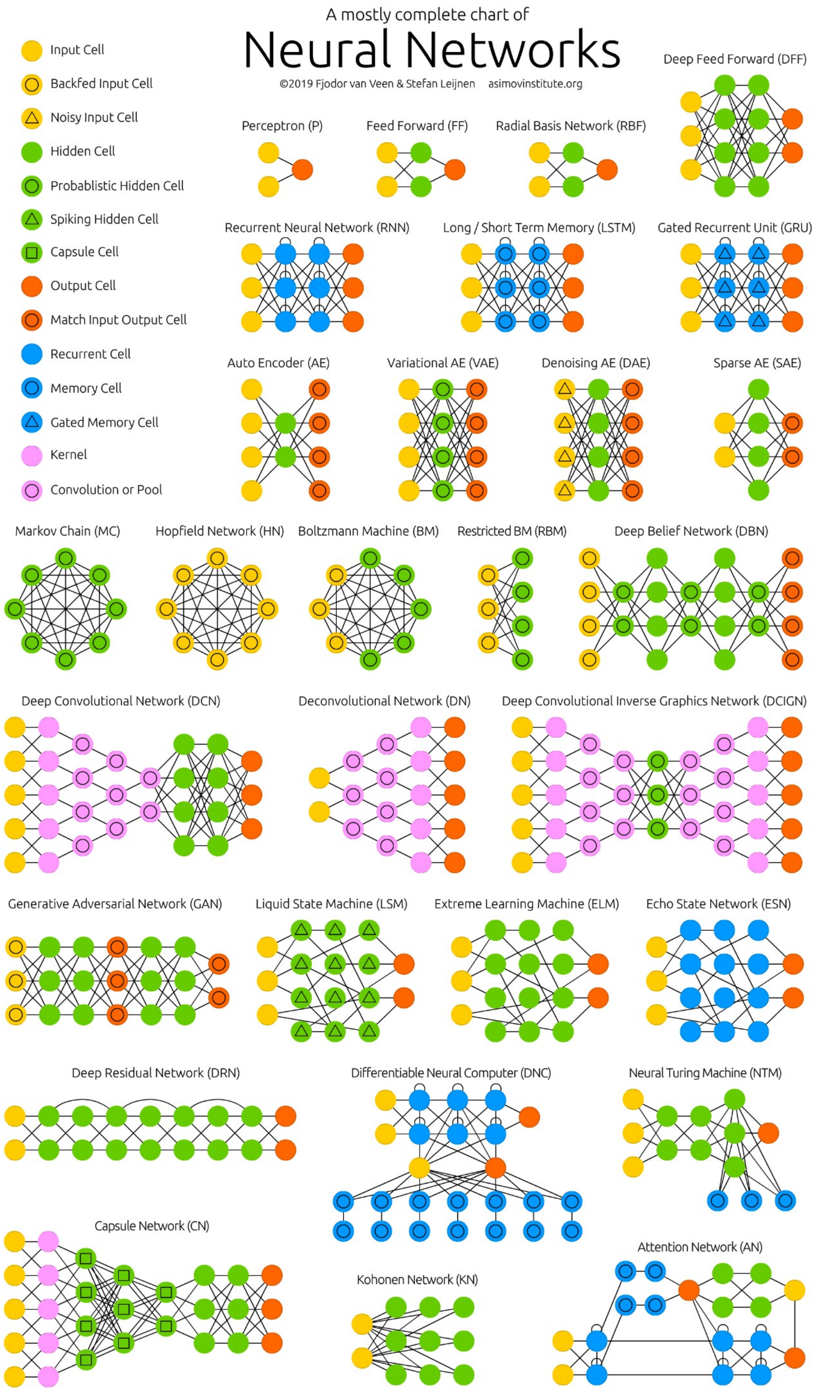


https://www.coursera.org/learn/neural-networks-deep-learning

Ozaki, Y., Yano, M. & Onishi, M. Effective hyperparameter optimization using Nelder-Mead method in deep learning. IPSJ T Comput Vis Appl 9, 20 (2017). https://doi.org/10.1186/s41074-017-0030-7

Rios, L.M., Sahinidis, N.V. Derivative-free optimization: a review of algorithms and comparison of software implementations. J Glob Optim 56, 1247–1293 (2013). https://doi.org/10.1007/s10898-012-9951-y

The concepts behind efficient hyperparameter tuning using Bayesian optimization
https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f



https://www.asimovinstitute.org/neural-network-zoo/

To solve a problem with machine learning, we need either of the following:

– a good set of features

– a lot of data (millions, billions)

– a good network architecture and learning algorithm

More complex problems require larger networks.

Larger networks contain more parameters.

More parameters require more data.

If we try to fit a large network with too little data, the model will overfit and make worse predictions than a simpler network.

Even a simple regression model can easily beat a large neural network if there is only scarce data.

Links

A Neural Network in Python http://iamtrask.github.io/2015/07/12/basic-python-network/
https://iamtrask.github.io/2015/07/27/python-network-part2/

A Fortran version: https://github.com/burubaxair/machine-learning-in-fortran/blob/main/nn.f90

Grokking-Deep-Learning https://github.com/iamtrask/Grokking-Deep-Learning

Backpropagation Video Tutorials
https://makeyourownneuralnetwork.blogspot.com/2015/04/backpropagation-video-tutorials.html

Draw network architecture diagrams

https://datascience.stackexchange.com/questions/14899/how-to-draw-deep-learning-network-architecture-diagrams

Some “usual” activation functions
https://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons

How to choose the number of hidden layers and neurons
https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw
https://ai.stackexchange.com/questions/20680/should-neural-nets-be-deeper-the-more-complex-the-learning-problem-is
https://datascience.stackexchange.com/questions/26597/how-to-set-the-number-of-neurons-and-layers-in-neural-networks

How to implement a neural network https://peterroelants.github.io/posts/neural-network-implementation-part01/

37 Reasons why your Neural Network is not working
https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

Gradient Descent on Riemannian Manifolds
https://wiseodd.github.io/techblog/2019/02/22/optimization-riemannian-manifolds/