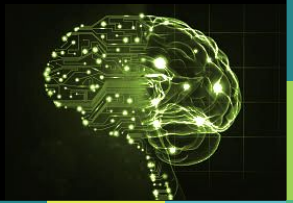
Back Propagation

Dr. Edith Chorev Metger - Data Scientist & Neuroscientist

Head of Data Science @FactoryPal

edith.chorev@gmail.com
https://www.linkedin.com/in/edith-chorev-bbb719/



Back Propagation

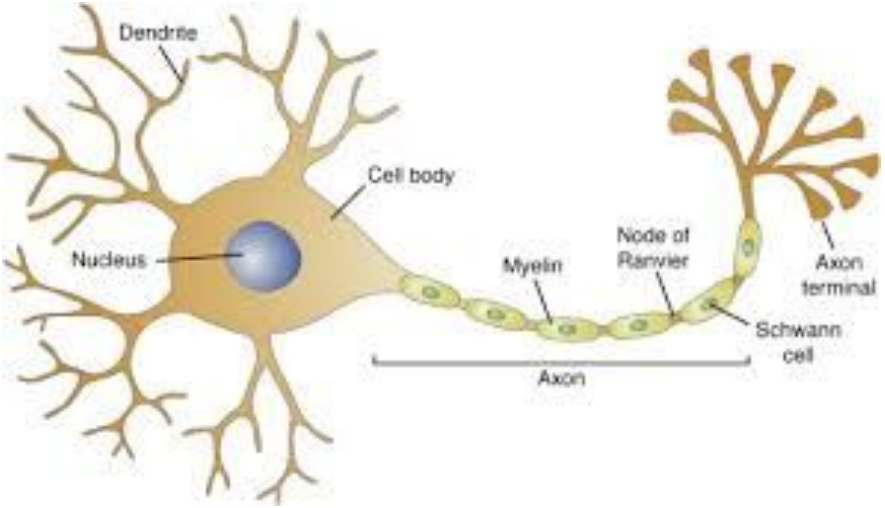
The learning algorithm for Neural-Networks.

Neural Networks

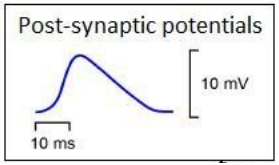
In an attempt to create machines that can learn, people looked at the brain for inspiration

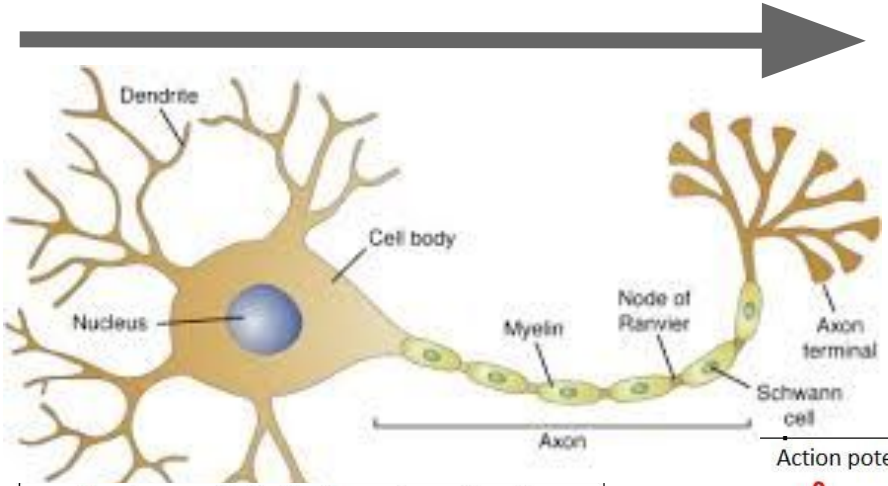
Learning:
the acquisition of knowledge or skills through study, experience, or being taught.

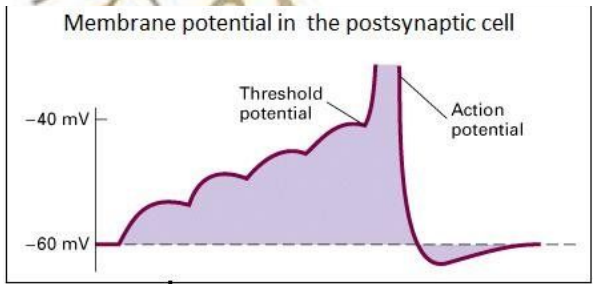
A neuron

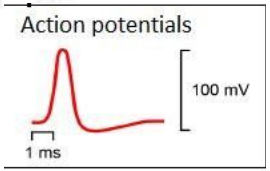

Dendrite
Cell body
Nucleus
Myelin
Node of Ranvier
Axon terminal
Schwann cell
Axon

A neuron

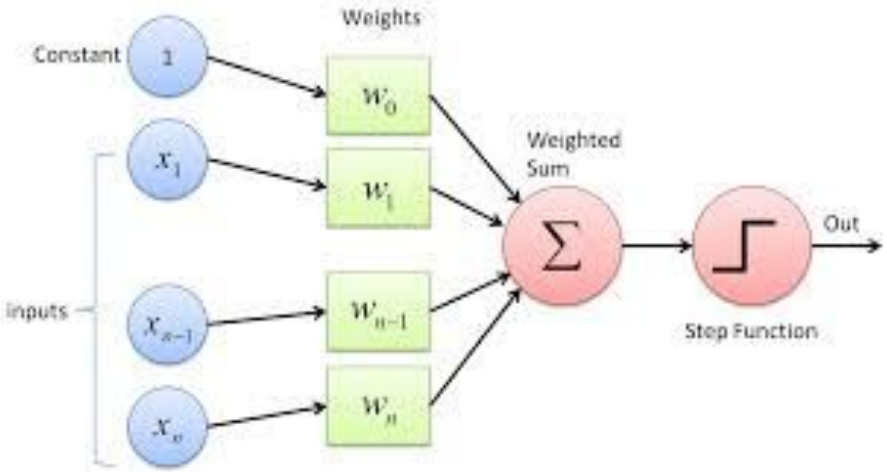

Post-synaptic potentials
10 mV
10 ms


Dendrite
Cell body
Nucleus
Myelin
Node of Ranvier
Axon terminal
Schwann cell
Axon

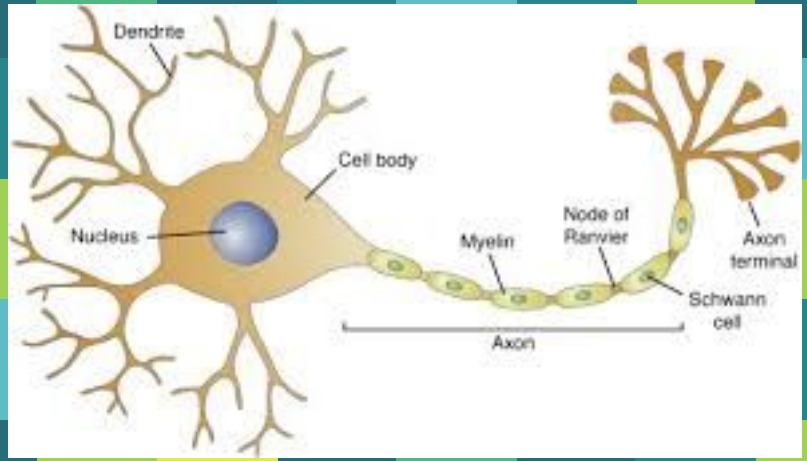

Membrane potential in the postsynaptic cell
Threshold potential
Action potential
−40 mV
−60 mV


Action potentials
100 mV
1 ms

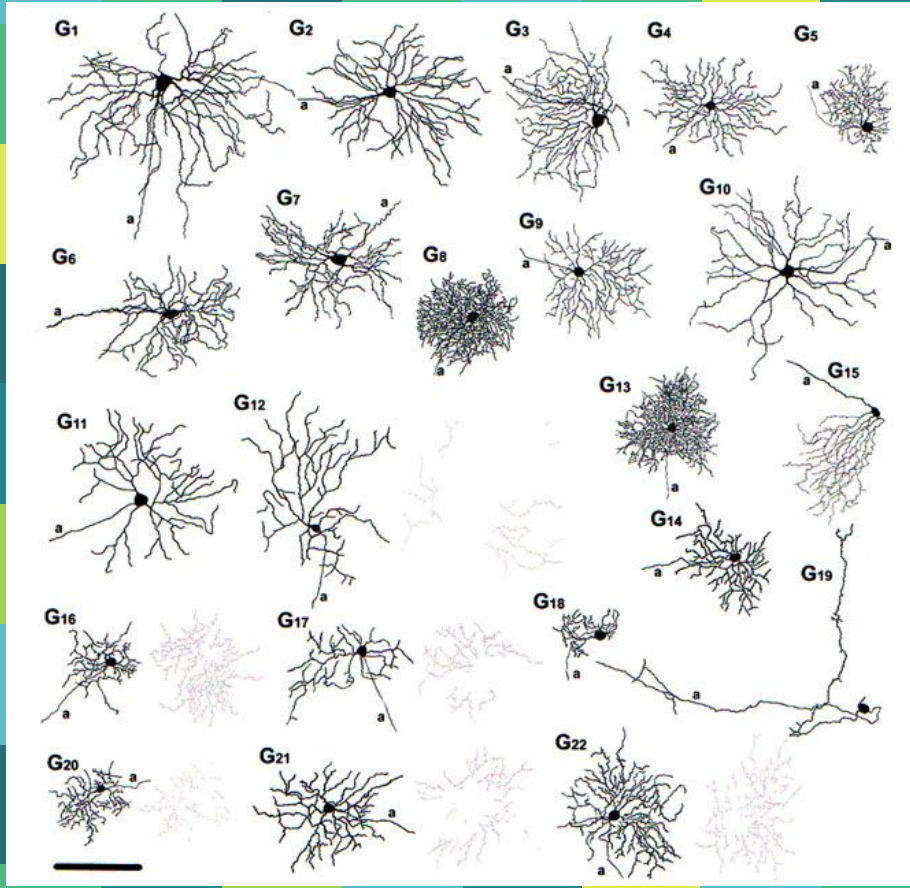
An artificial neuron



A Biological Neurons

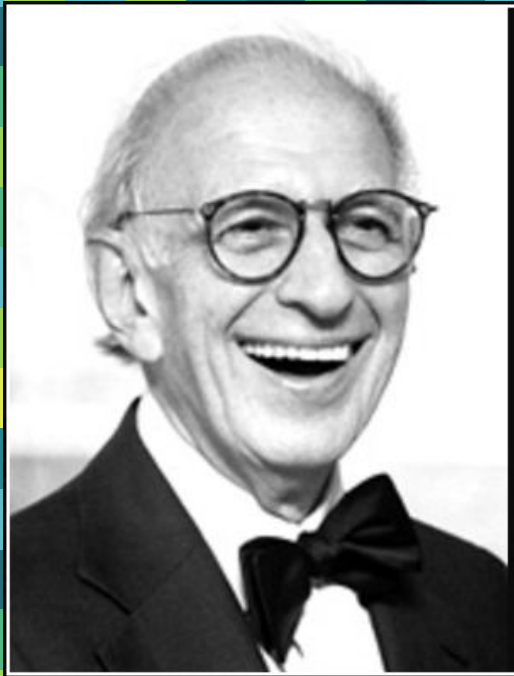


The Neuronal Jungle



A trip down memory lane…

1. Perceptron
2. Optimization problem
3. Gradient descent
4. Multilayer NN
5. ANN vs. BNN



It is this potential for plasticity of the relatively stereotyped units of the nervous system that endows each of us with our individuality.

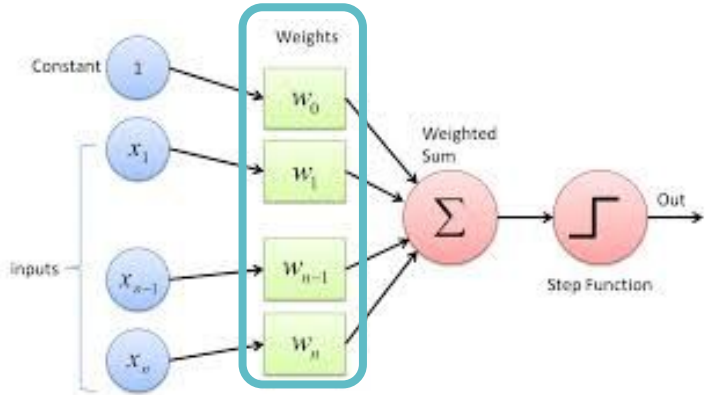
— Eric Kandel —

AZ QUOTES

The perceptron Algorithm: The Δ rule

Rosenblatt 1958 "The perceptron: a probabilistic model for information storage and organization in the Brain".

Learning from labeled data - supervised learning.



The learning Algorithm: Δ rule

Perceptron training:

1. Initialize weights vector with small random numbers
2. Repeat until convergence:
   a. Loop over feature vector (x_j) and labels (l_j) in training set *D*.
   b. Take x and pass it through the perceptron, calculating the output values: $y_j = f(w(t) \cdot x_j)$
   c. *Update weights:*

      $w_i(t+1) = w_i(t) + \alpha(l_j - y_j)\ x_{j,i}$

      for all $0 <= i < n$

Wi

$w_i(t+1) = w_i(t)+\alpha(l_j-y_j)x_{j,i} = w_i(t)+\alpha(l_j-w_i(t)\cdot x_j)x_{j,i}$
for all $0 <= i < n$

α - *learning rate*

(l_j-y_j) - the Δ between the actual label and the predicted label (also known as **error**).

3. Termination of training:
    a. When all samples are correctly labeled
    b. After a pre-set number of epochs
    c. After a pre-set number of epochs where the percentage of misclassified labels remains stable.

The learning step:

$w_i(t+1) = w_i(t) + \alpha(l_j - y_j)\, x_{j,i} =$
$w_i(t) + \alpha(l_j - x_{j,i} \cdot w(t))x_{j,i}$

$(l_j - y_j) > 0$: w_i *needs to become bigger*
$(l_j - y_j) < 0$: w_i *needs to become smaller*
$(l_j - y_j) = 0$: w_i *should not change*

Perceptorn:

Rather limited in its capacity. Binary-linear classification tasks as well linear regression can be easily solved in various other ways.

Neural Networks Winter

Winter did not last forever

Some developments from the field of control theory emerged that will revive the NN field.

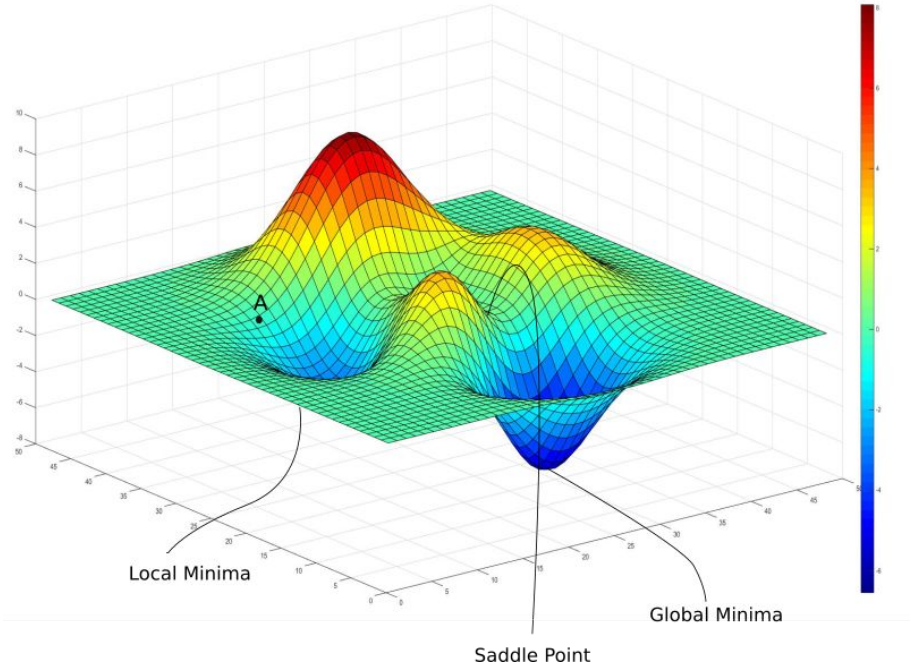
Optimization problem

Finding the minimum of an unknown error manifold

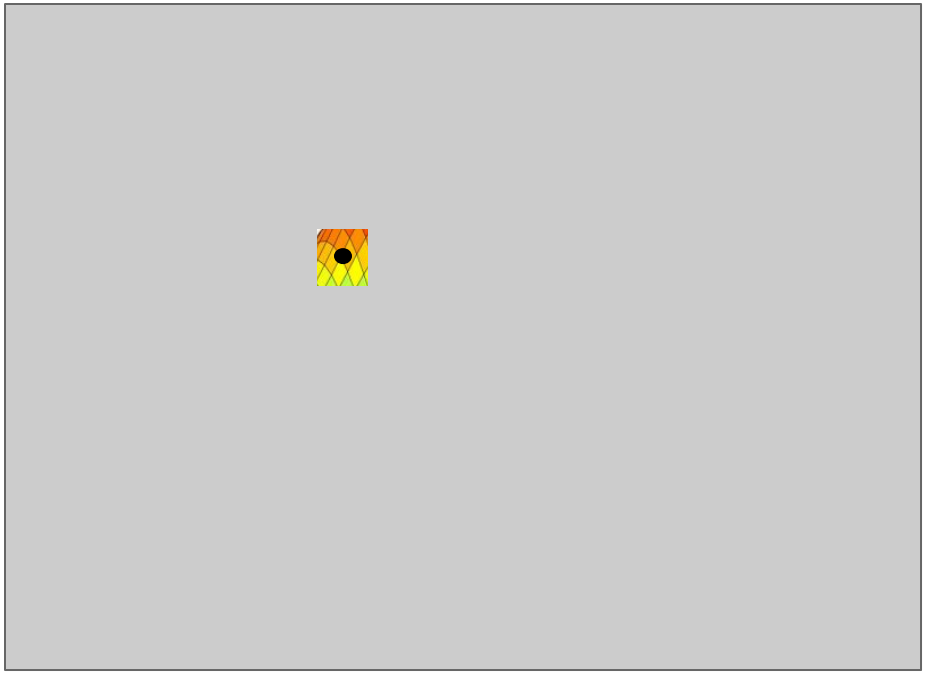
Gradient Descent

Gradient Descent:

Follow the downward slope... (i.e derivative)



We Don’t know how this manifold looks - what to do???

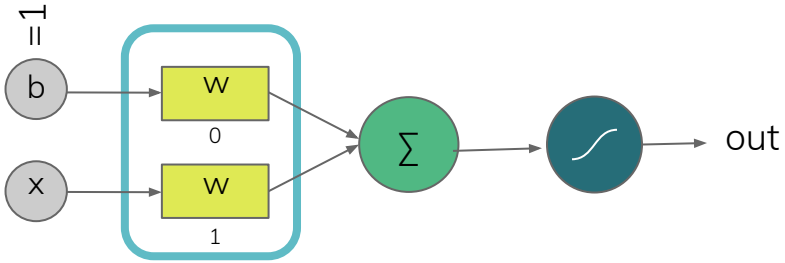


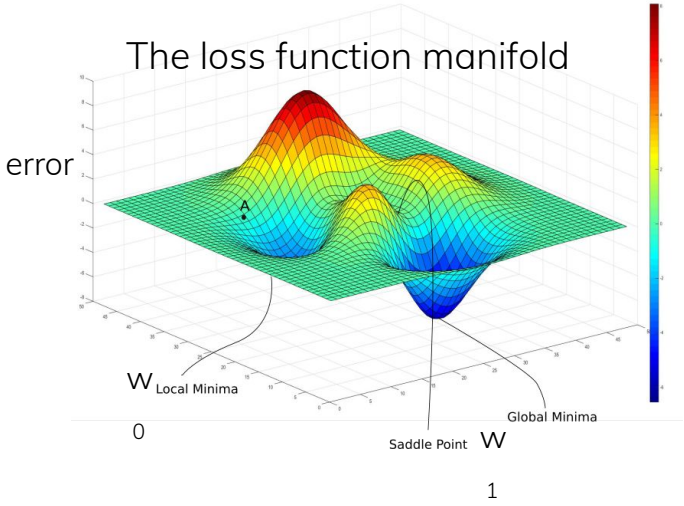
The “**good enough**” principle :

“[An] optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, **but** it often finds a very low value of the [loss] function quickly enough to be useful.”

Goodfellow et al.

Back to the perceptron...





We want to find the minimum of this error function

Backpropagation:

Developed by Henry J. Kelley (1960) & by Arthur E. Bryson (1961). In 1962 Stuart Dreyfus derives a simpler form using the chain rule.
In 1986 Rumelhart, Hinton and Williams showed experimentally that this method can generate useful internal representations of incoming data in hidden layers of neural networks.

Backpropagation Algorithm

For number of epochs / until loss sufficiently low / loss no longer improves:

Wgradient = Evaluate_gradient (current_loss, data, W)

W += -α * Wgradient

α - a very critical parameter. What will happen if it is too big? And what will happen if it is too small?

The loss function:

There are many different loss functions. Commonly used examples are:

1. Classification: Categorical cross entropy / sigmoid for binary classification
2. Regression: Mean square error

In order to use GDA we need a function that is differentiable!

In the perceptron example we used a simple delta function:

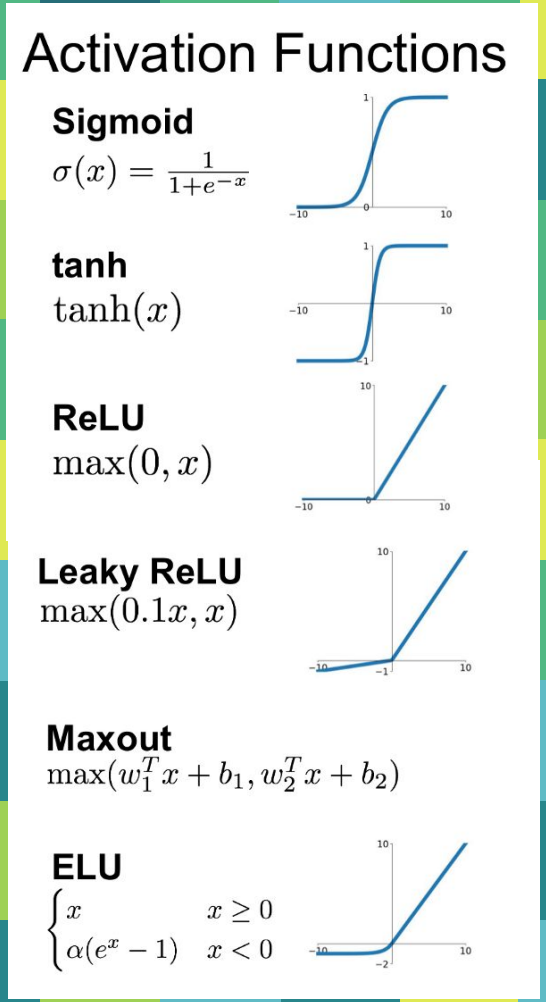
(true - predicted)

The activation function:

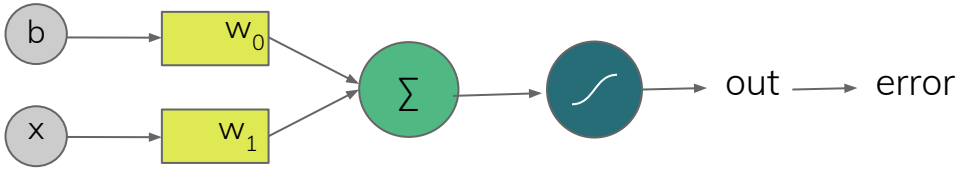
Also needs to be differentiable.

We used:

1. Classification: step function
2. Regression: ReLU

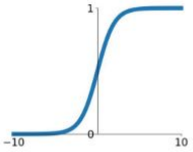


The Forward Pass

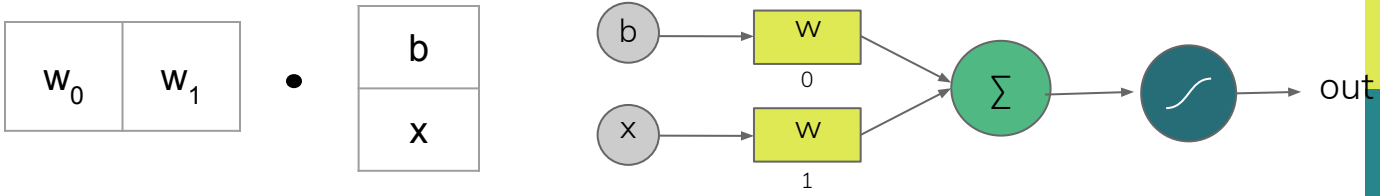


Sigmoid

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



Forward Pass



Output = $1/(1+e^{-W \cdot X})$

Error = Output-Label

Loss= $0.5(\text{Error})^2$

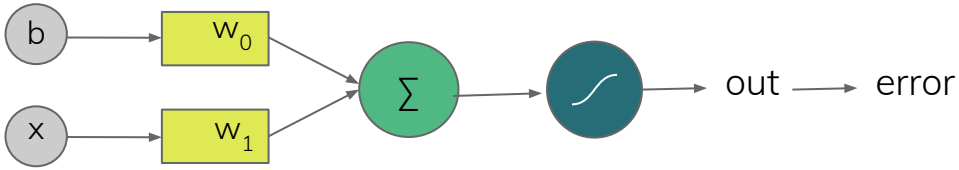
Backward pass - using gradient descent

We want to update each one of the weights such that we will move down slope on the error function. For that we use the gradient descent algorithm.

$$\text{Output} = 1/(1+e^{-W \cdot X})$$

$$\text{Error} = \text{label} - \text{Output} = \text{label} - 1/(1+e^{-W \cdot X})$$

Backward pass

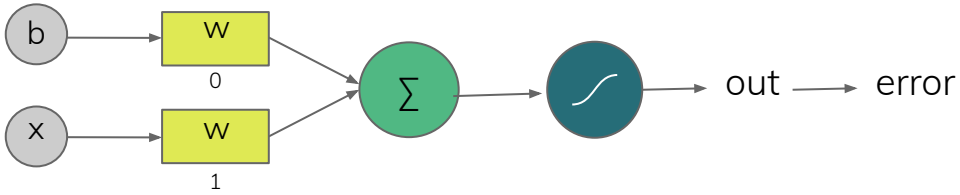


$$E_{total} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}(taget - Out)^2$$

Remember the chain rule

$$\frac{\partial E_{total}}{\partial w_i} =$$

Backward pass



$$\frac{\partial E_{total}}{\partial w_i} = \frac{\partial E_{total}}{\partial Out} * \frac{\partial Out}{\partial Net} * \frac{\partial Net}{\partial W_i}$$

Develop the equation for the updating of each one of the weights.

Backward pass: sigmoid derivative

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Backward pass

$$\frac{\partial E_{total}}{\partial Out} = \frac{1}{N}\sum_{i=1}^{N}(2 * 0.5 * (Out_i - l_i)) = \frac{1}{N}\sum_{i=1}^{N}(Out_i - l_i) = Error$$

$$\frac{\partial Out}{\partial Net} = Out * (1 - Out)$$

$$\frac{\partial Net}{\partial W_i} = \frac{\partial(w_0 \cdot x_0 + ... + w_n \cdot x_n)}{\partial w_0} = x_0$$

Practical #2: back propagation + BPA

Optimization methods

The convergence of NN is not always granted. There are methods to optimize the convergence process.

Vanilla GD can be slow and wasteful

Stochastic Gradient Descent (SGD):

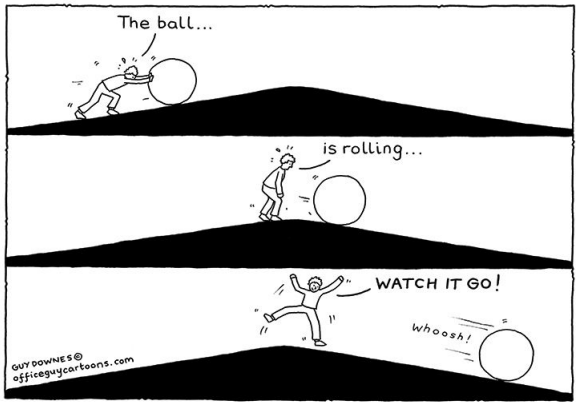
Instead of calculating the gradient on the entire training dataset, we calculate it on small **batches.**

 # implement SGD for the last classification problem we did in practica 2.

Other modifications to GD

Momentum:

In physics it is a vector: $p = mv$ (kg·m/s)



Other modifications to GD

Momentum:

Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go. The equations of gradient descent are revised as follows:

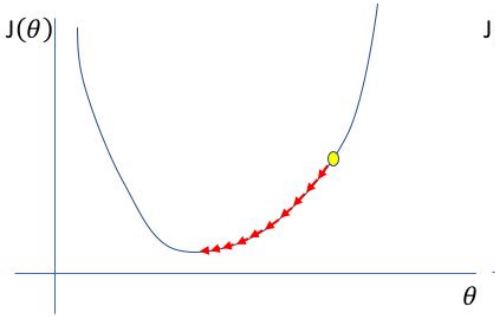
Original: $W = W - \alpha \nabla_W f(W)$

$V_t = \gamma V_{t-1} + \alpha \nabla_W f(W)$

$W = W - V_t$

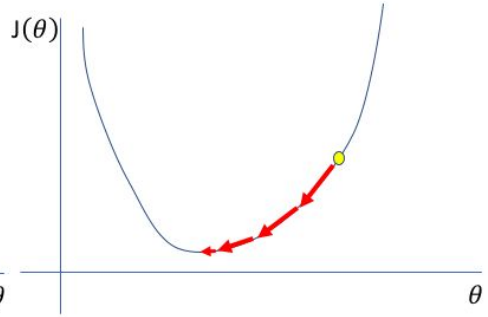
Learning rate optimizations

Too low



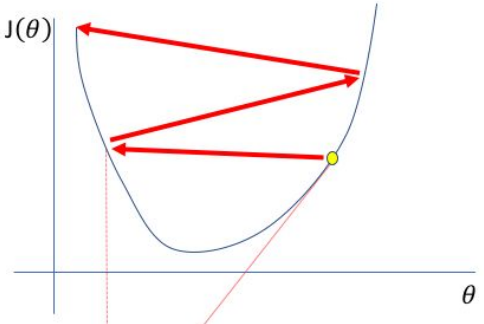
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



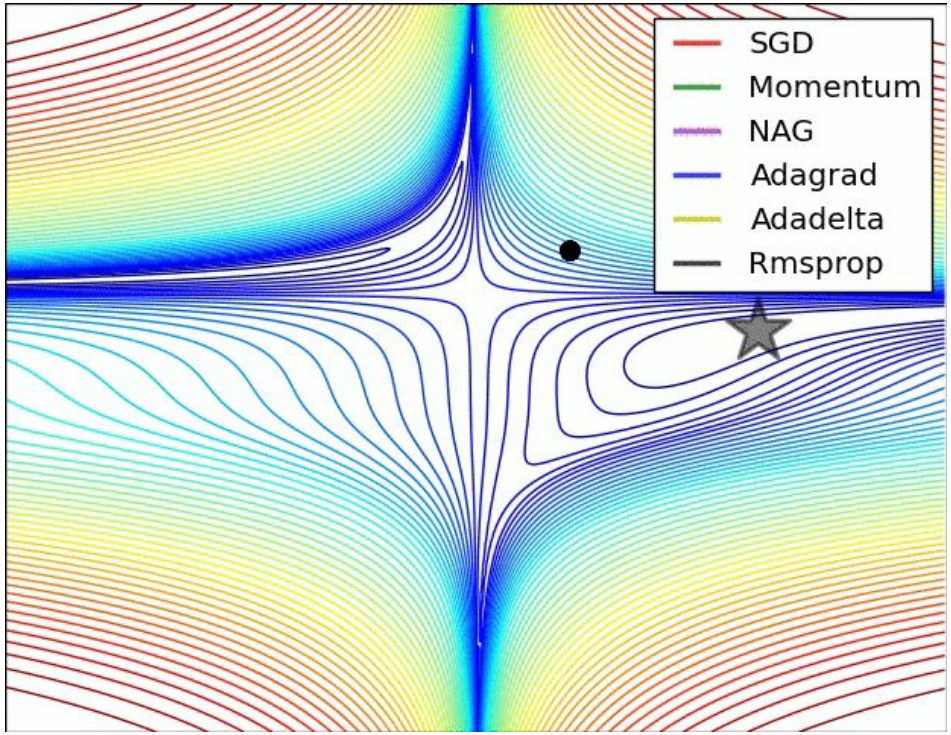
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Learning rate optimizations

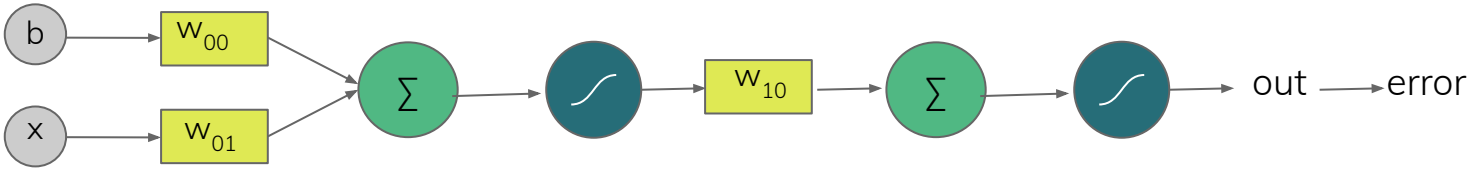
1. Include different learning rates for different features.
2. Changing the learning rate in the course of training
3. Combination of the two

Optimization methods

1. **Similar to momentum - but using the loss calculated for the new weights (look a head) -** Nesterov Accelerated Gradient
2. **Adaptive momentum - adapting the momentum such that it will slow down when reaching a minima.**

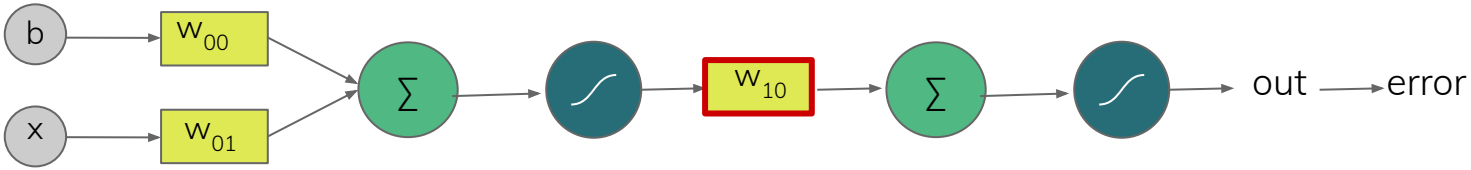

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

Multi-layer perceptron (NN)



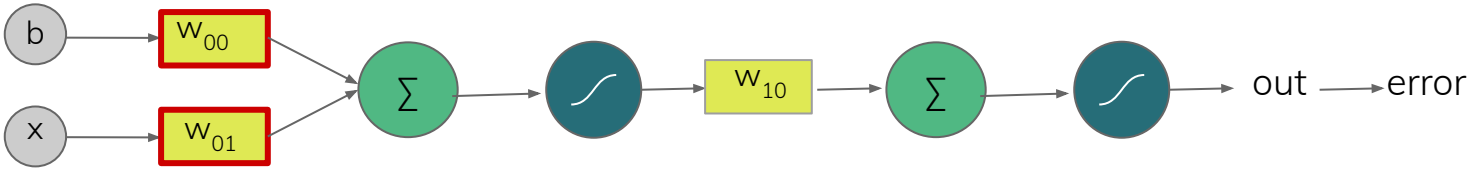
How will the backward pass look now?

Multi-layer perceptron (NN)



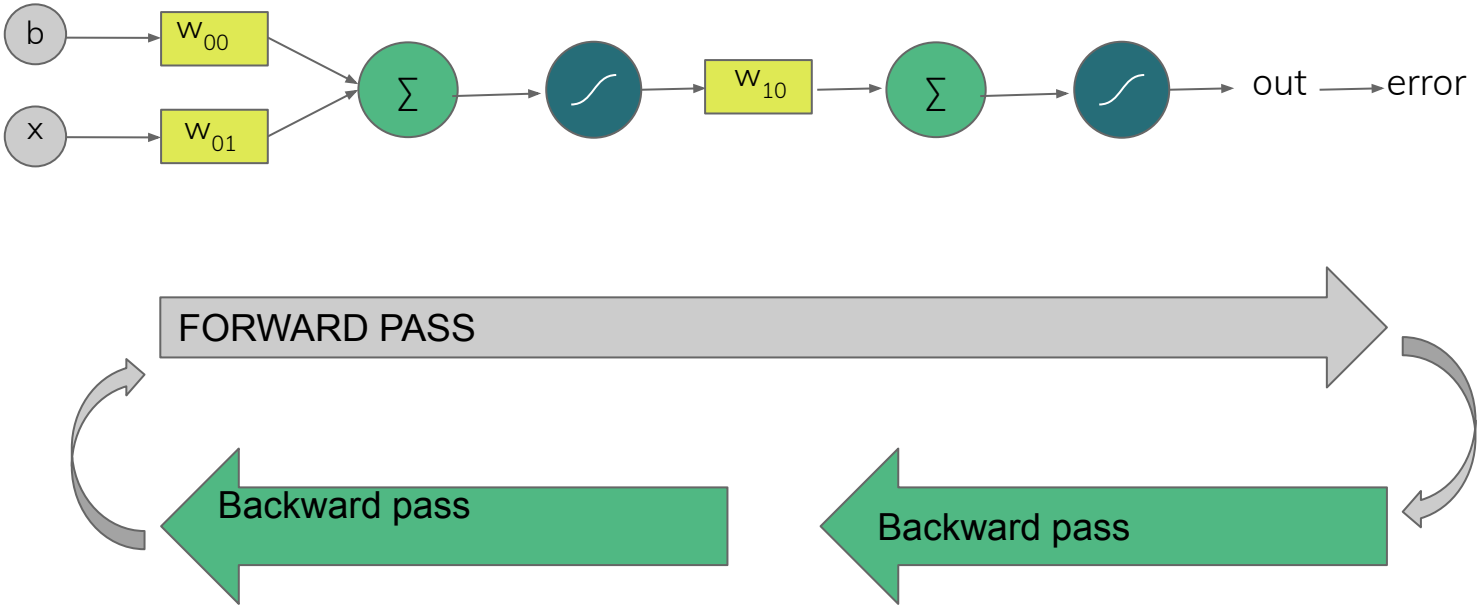
$$E_{total} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}(taget - Out)^2$$

Multi-layer perceptron (NN)



$$E_{total} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}(taget - Out)^2$$

Multi-layer perceptron (NN)



There are infinite sets of weights that can yield a reasonable outcome. How do we make sure that the the set we end up with generalizes well (i.e. does well also on test set)??

Regularization

Regularization

1. Adding a regularization term to the loss function: L1/L2/Elastic Net
2. Explicit method: by adding dropout to the NN architecture
3. Implicit methods: data augmentation, early stopping of training

Regulariztion

1. **Adding a regularization term to the loss function: L1/L2/Elastic Net**

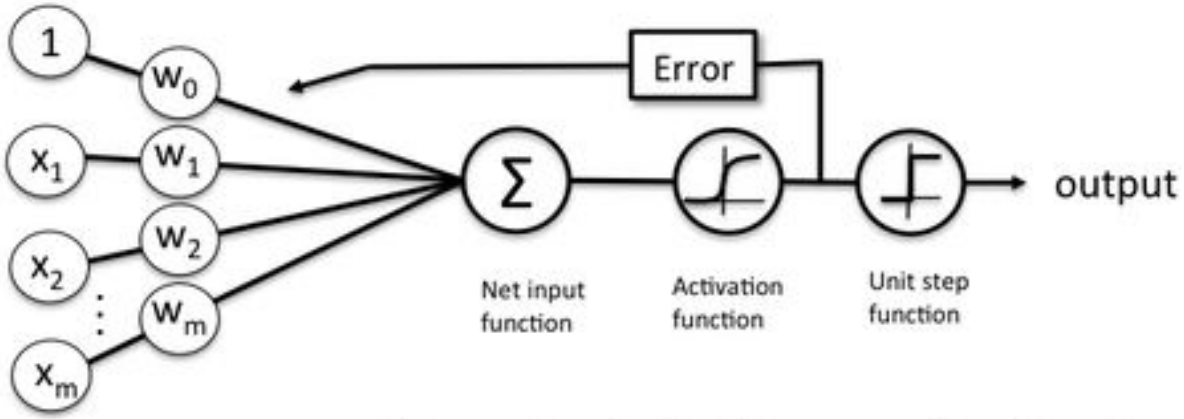
L2: $$R(W) = \sum_i \sum_j W_{i,j}^2$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W)$$

Neural Networks intuition

https://playground.tensorflow.org/#activation=sigmoid&batchSize=8&dataset=spiral&regDataset=reg-plane&learningRate=0.003&regularizationRate=0&noise=20&networkShape=1,2&seed=0.99530&showTestData=false&discretize=false&percTrainData=70&x=true&y=true&xTimesY=true&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=f

Backprop vs. logistic regression:

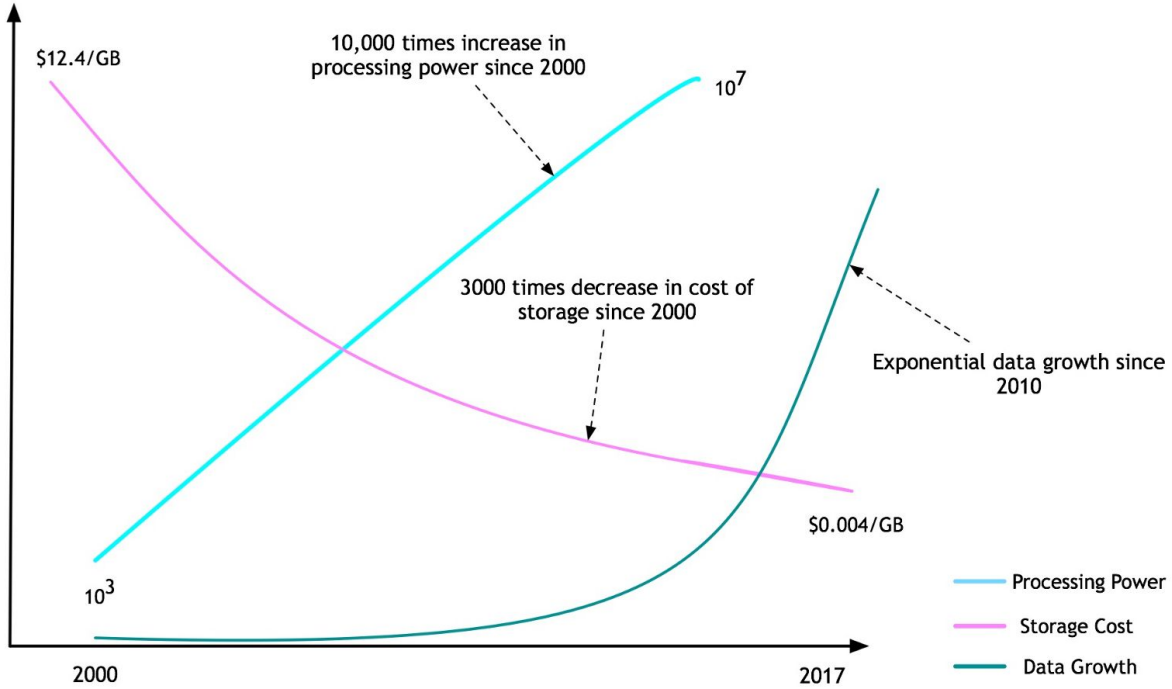


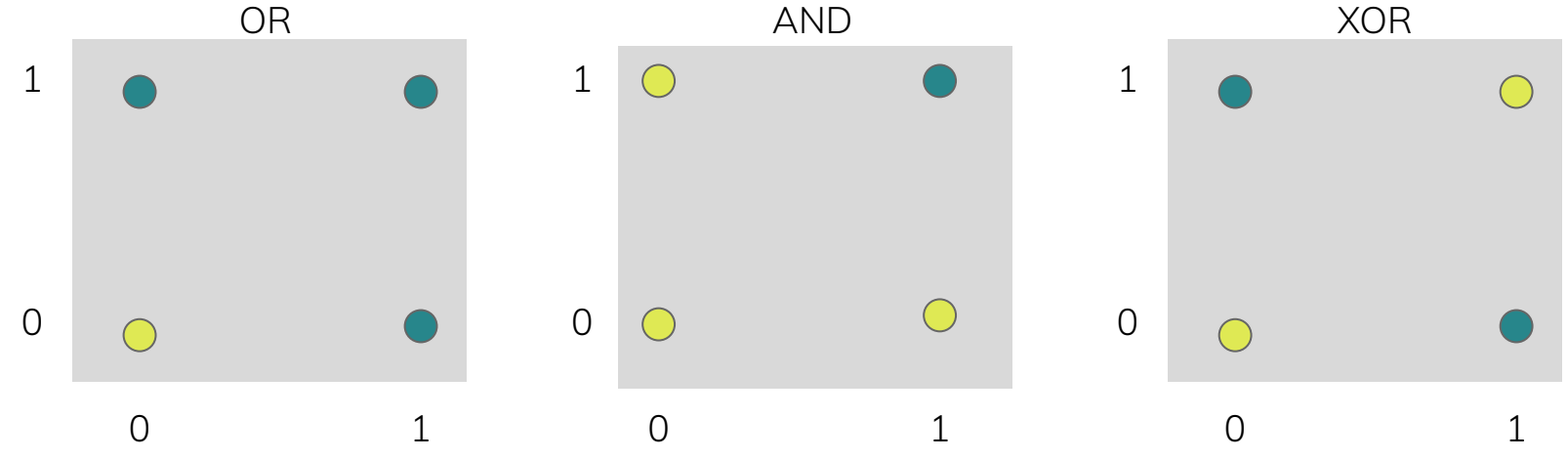
Schematic of a logistic regression classifier.

Why did it take so long for NN to boom?

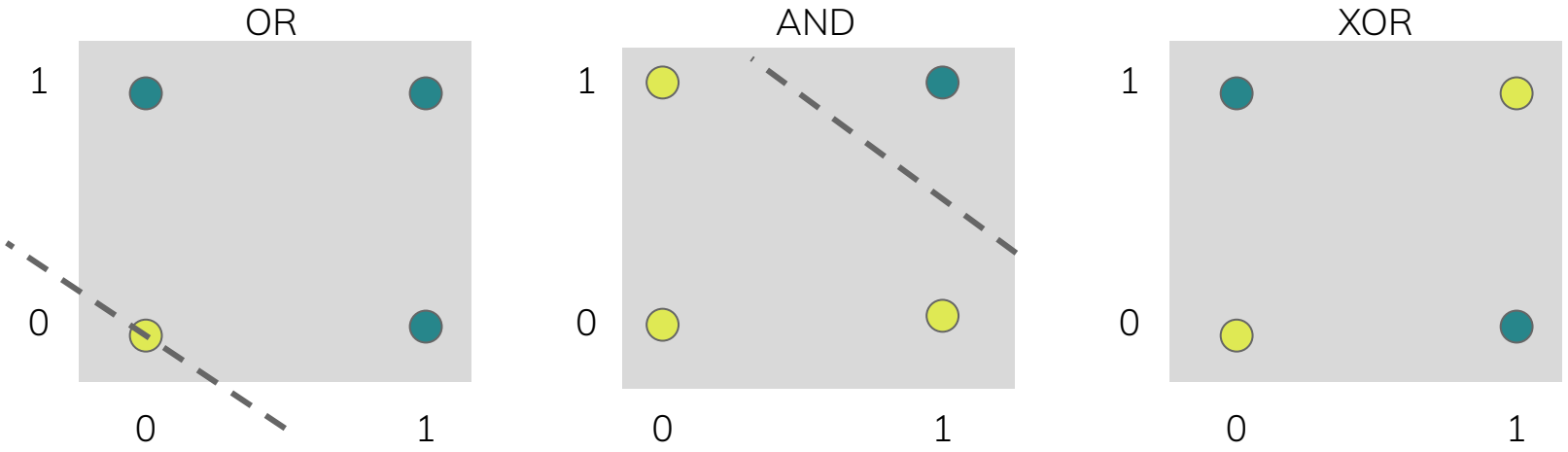
The answer is rather simple - NN require a lot of data and computational power.

Why did it take so long for NN to boom?



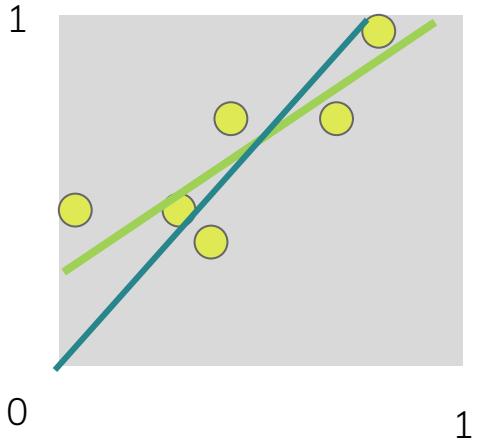


No Bias			
With Bias			



Perceptron solves only linearly-separable problems!

$$y = wx + b$$



Bias - another variable that is not input dependent (as opposed to the weights).

Normalization of inputs and outputs

Neural Networks Tuning?

Architecture: Rules of thumb

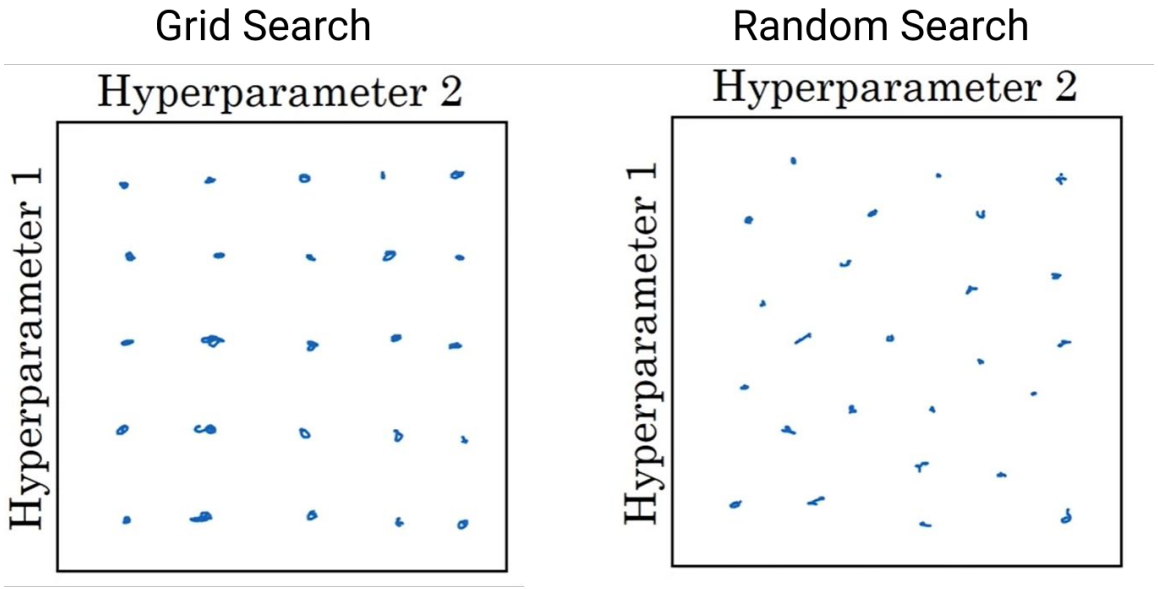
1. Width - between number of feature and output size
2. Depth - the more complex the problem (complex boundary condition) the deeper you go. Start simple and increase complexity as you go.

Other HyperParameters (batch size, learning rate, momentum, dropout...)

Order of importance:

1. learning rate
2. Momentum β, for RMSprop, etc.
3. Mini-batch size
4. Number of hidden layers
5. learning rate decay
6. Regularization λ

Grid/ Random/ Hyperband Search: if parameter space not to big...



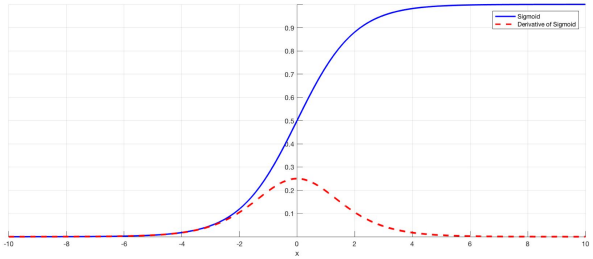
Vanishing gradients problem

neural network's weights receives an update proportional to the partial derivative of the loss function with respect to the current weight in each iteration of training.

The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

Vanishing gradients problem

Some activation function are more susceptible to this problem. The sigmoid has gradient between 0-1. The gradient is actually very small close to 0/1. Computing the gradient using the chain rule will result in multiplying this small gradient n times (n = number of layers). Thus the gradient becomes exponentially small the deeper the network, resulting in very small gradients for the initial layers.



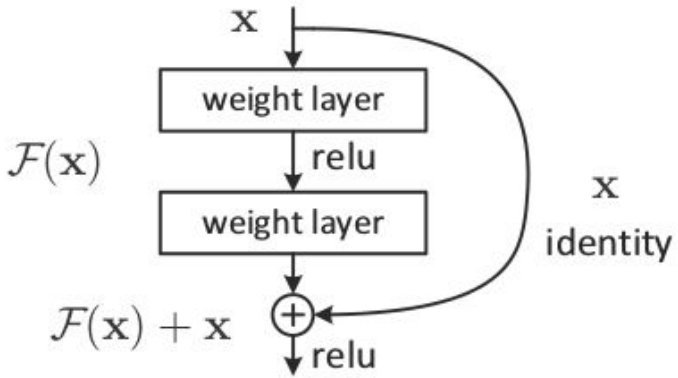
Vanishing gradients problem

Solutions:

- Other activation functions such as Relu:

  Relu derivative is 1 if input>0, else 0.

- Skip connections:



Using Backprop to Fool Neural Networks:

How can you generate an input to a network that will cause it to misclassify it, although to a human it will seem completely normal?

Using Backprop to Fool Neural Networks: Gradient Ascent

Instead of updating the weights using backprop update an input:

$$x_{i,j} = x_{i,j} + \text{dloss}/\text{dx}_{i,j}$$

Using Backprop to Fool Neural Networks:

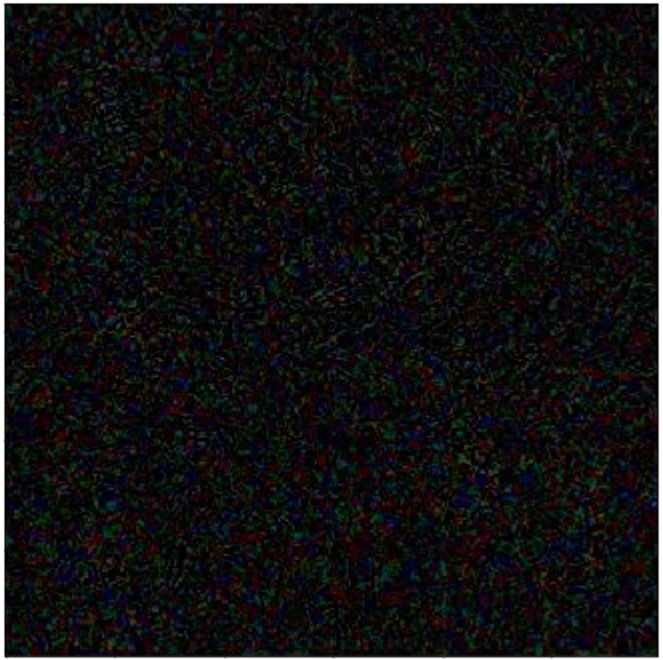
labrador_retrieve

Inception, 46% confidence

Italien_grayhound

Inception, 99.9% confidence

Using Backprop to Full Neural Networks:



Neural Networks

How close are ANN to the real thing?

Scale and architecture:

Our brain:

- ~ 10^{12} Neurons
- 1000-100000 inputs per neuron
- Information passes continuously, with many recurrent connections

NN:

- ~ 1000 Neurons
- 100-10000 inputs
- Information passes sequentially in a directional manner

Efficiency:

Our brain:

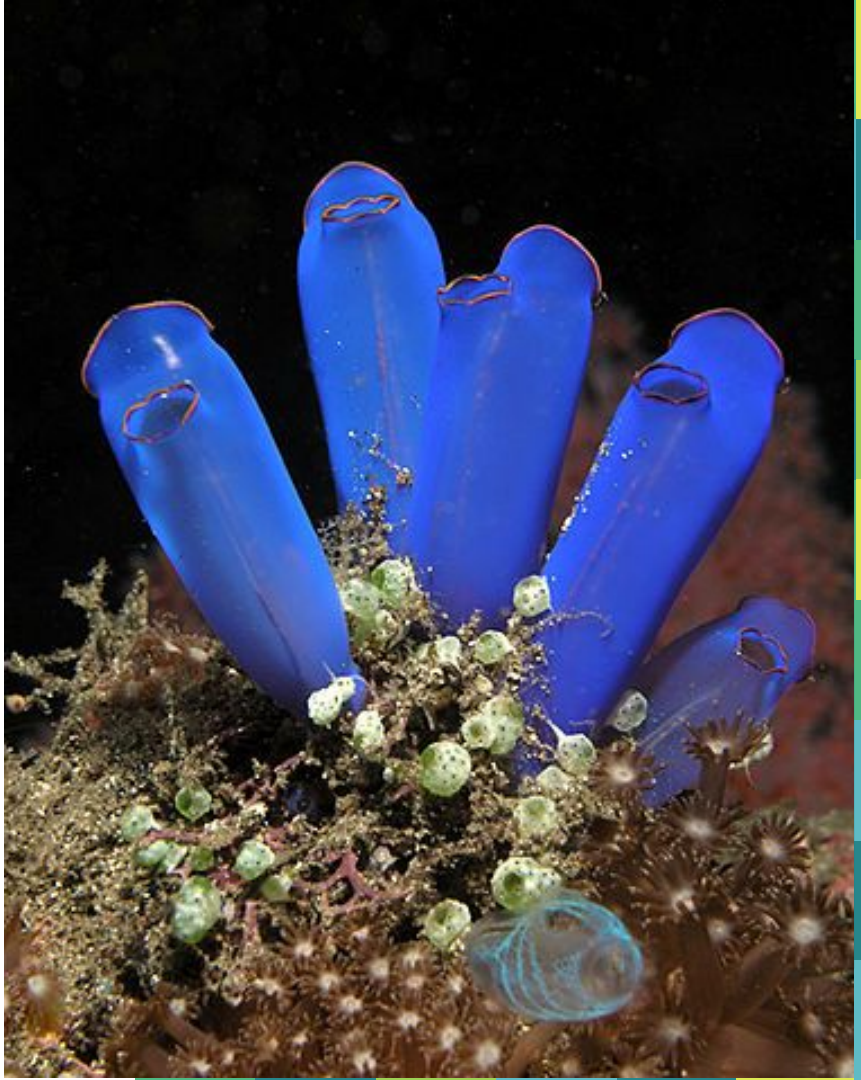
- 20 watts
- 37℃

NN:

- Nvidia GeForce works in 250 watts
- Heats to 50-80℃

Evolution

Nervous systems developed when locomotion was developed.



No backward pass but similar outcome

“Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. … **When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B*, is increased.”**

Donald Hebb

Further Reading

Books:

1. Pattern Recognition and Machine Learning - CM Bishop

Articles:

backpropogation:

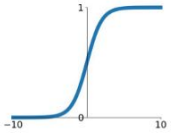
1. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
2. https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199 in CNN
3. http://axon.cs.byu.edu/~martinez/classes/678/Papers/Werbos_BPTT.pdf In RNNs

The AN Jungle

Activation Functions

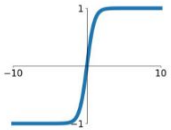
Sigmoid

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



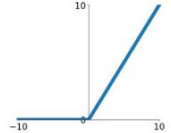
tanh

$$\tanh(x)$$



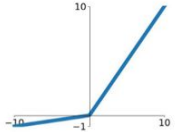
ReLU

$$\max(0, x)$$



Leaky ReLU

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

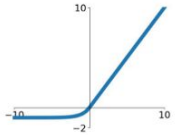


Maxout

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

ELU

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



https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092