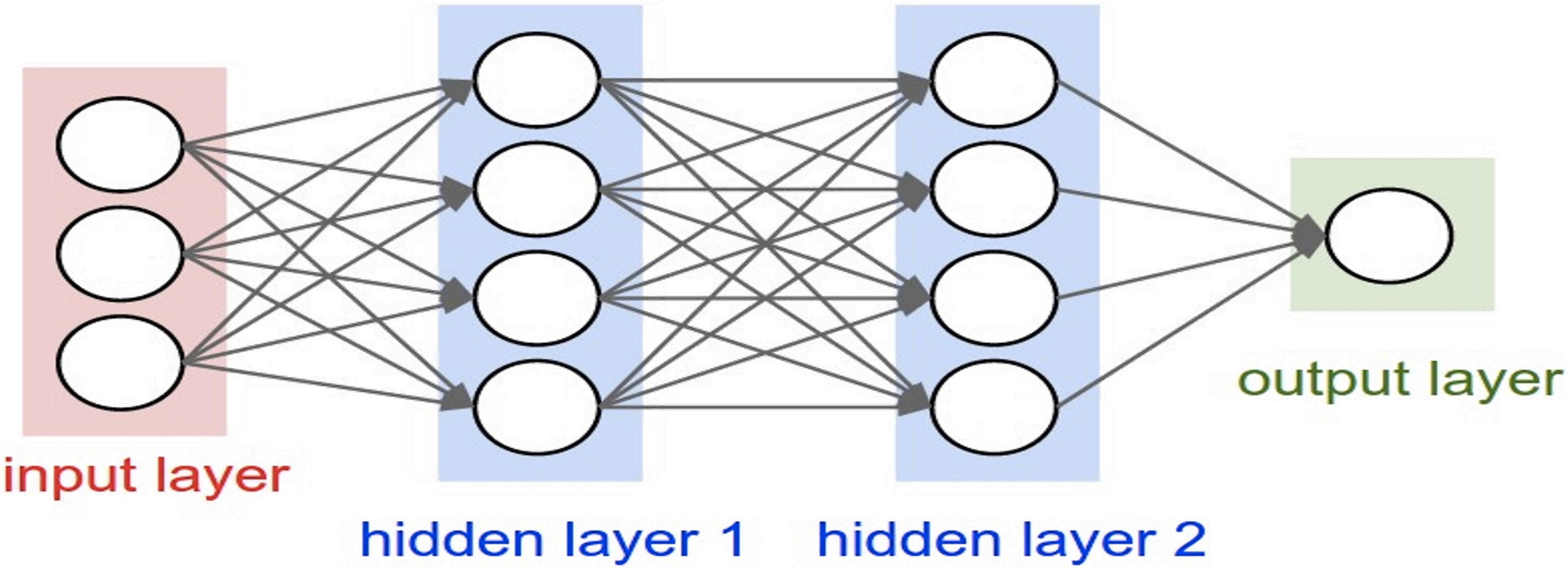
Neural Networks

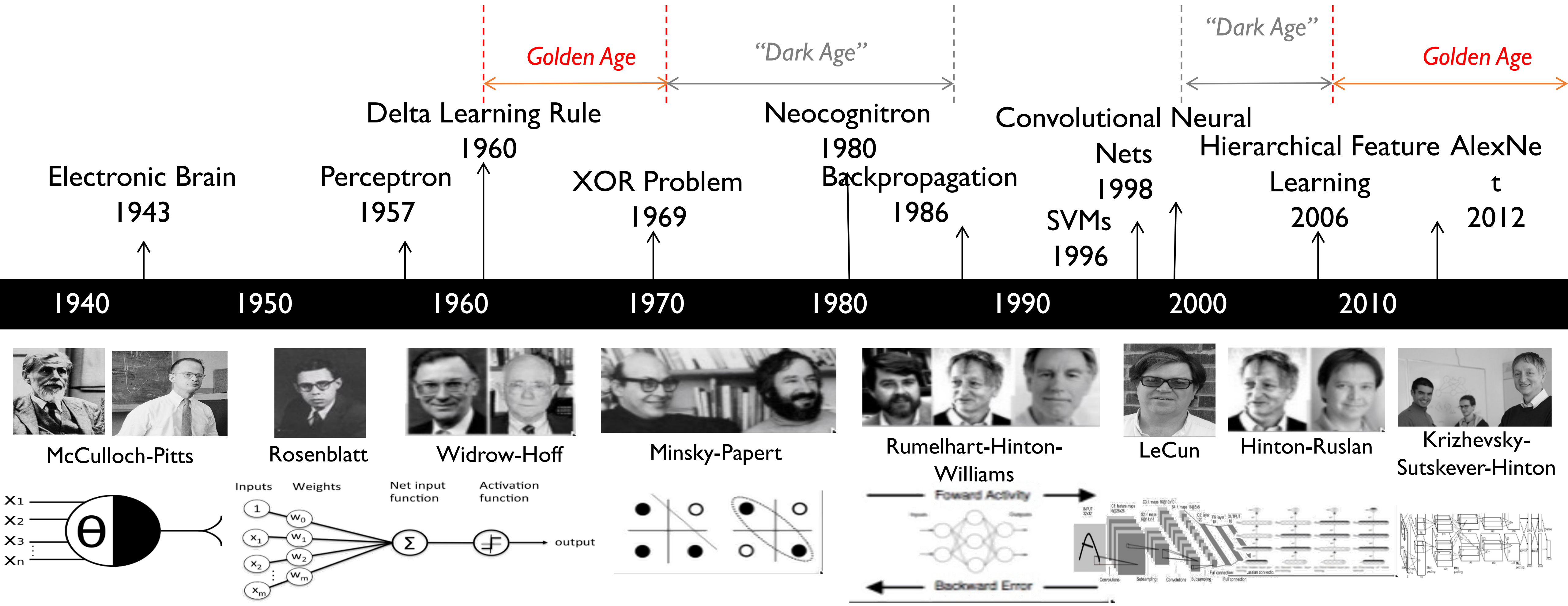
Neural Networks

Introduction

- Deep Learning - Rebirth of neural networks
- Inspired by the human brain (networks of neurons)

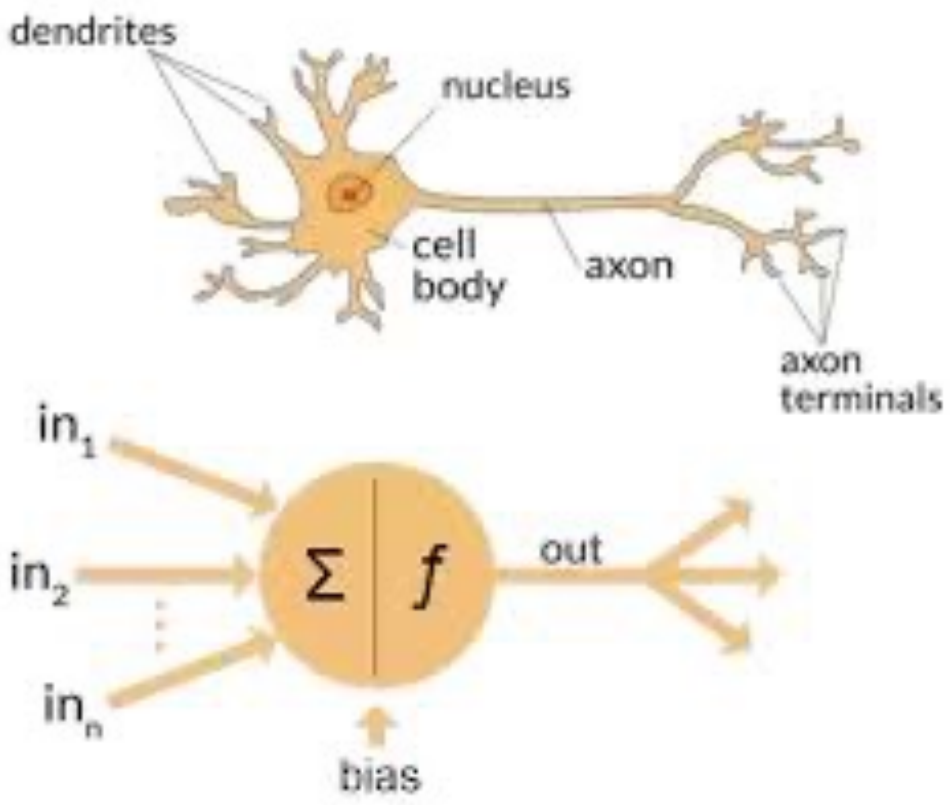


History of Neural Networks



Neuron

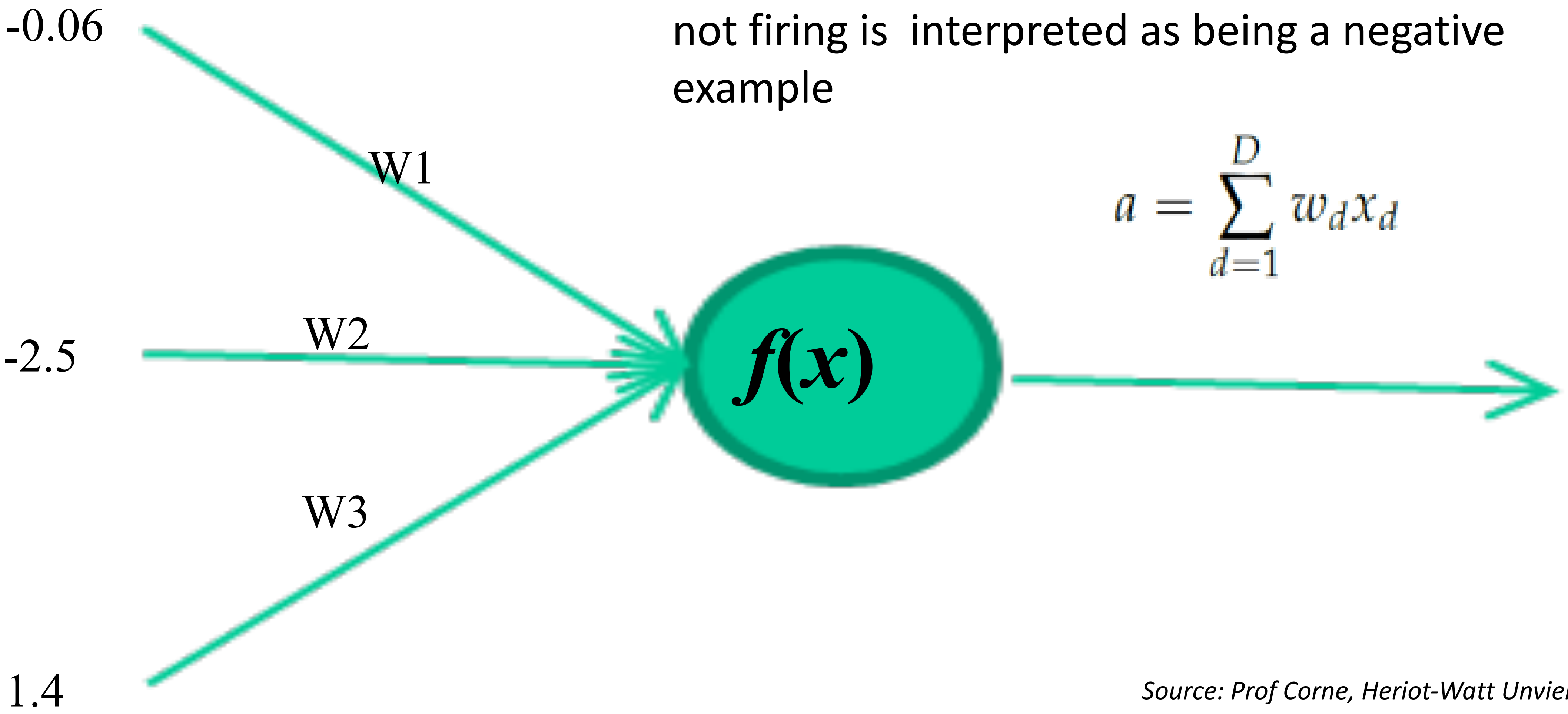
- Based on how much these incoming neurons are firing, and how “strong” the neural connections are, our main neuron will “decide” how strongly it wants to fire.
- Learning in the brain happens by neurons becoming connected to other neurons, and the strengths of connections adapting over time.
- Receives input from D-many other neurons, one for each input feature. The strength of these inputs are the feature values.



Neuron as a Function

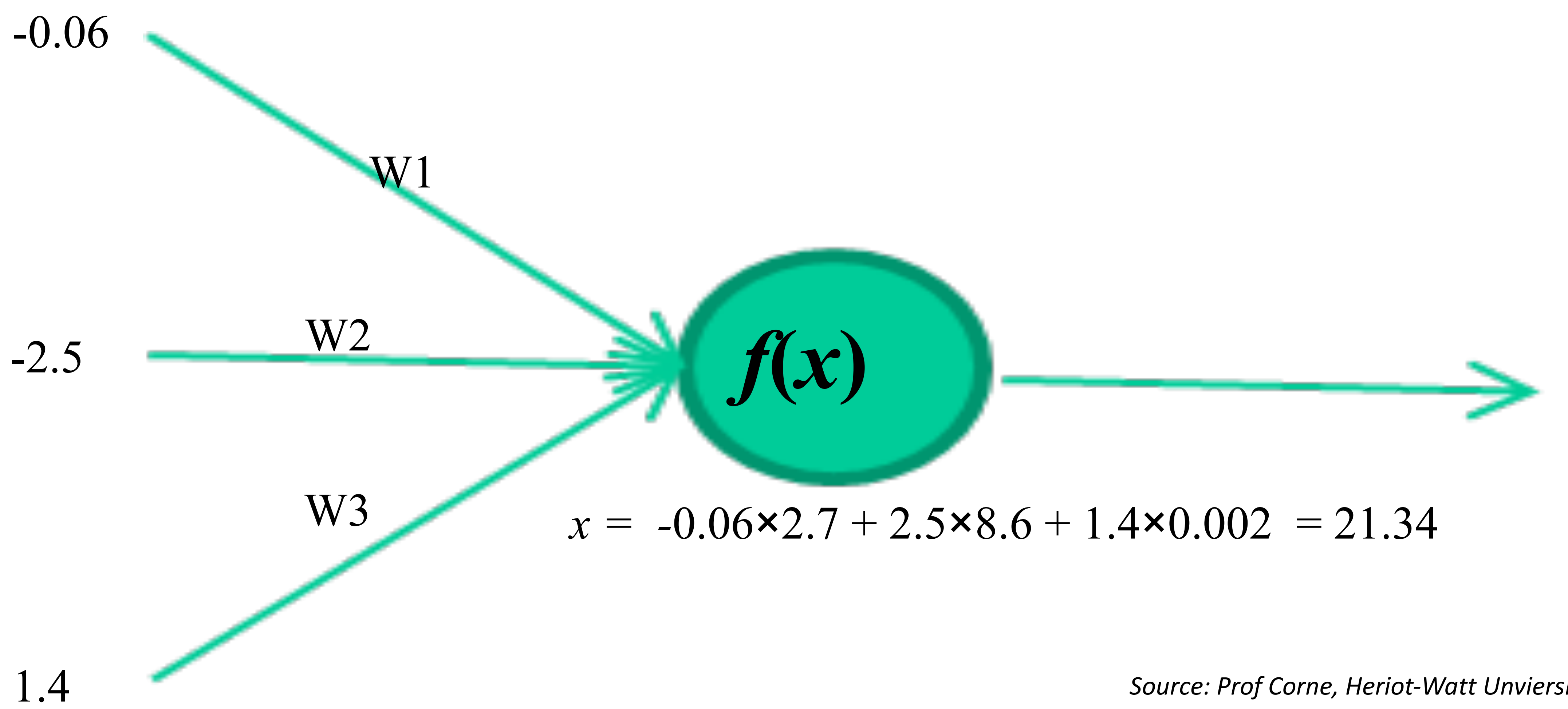
Firing is interpreted as being a positive example and not firing is interpreted as being a negative example

$$a = \sum_{d=1}^{D} w_d x_d$$



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neuron as a Function

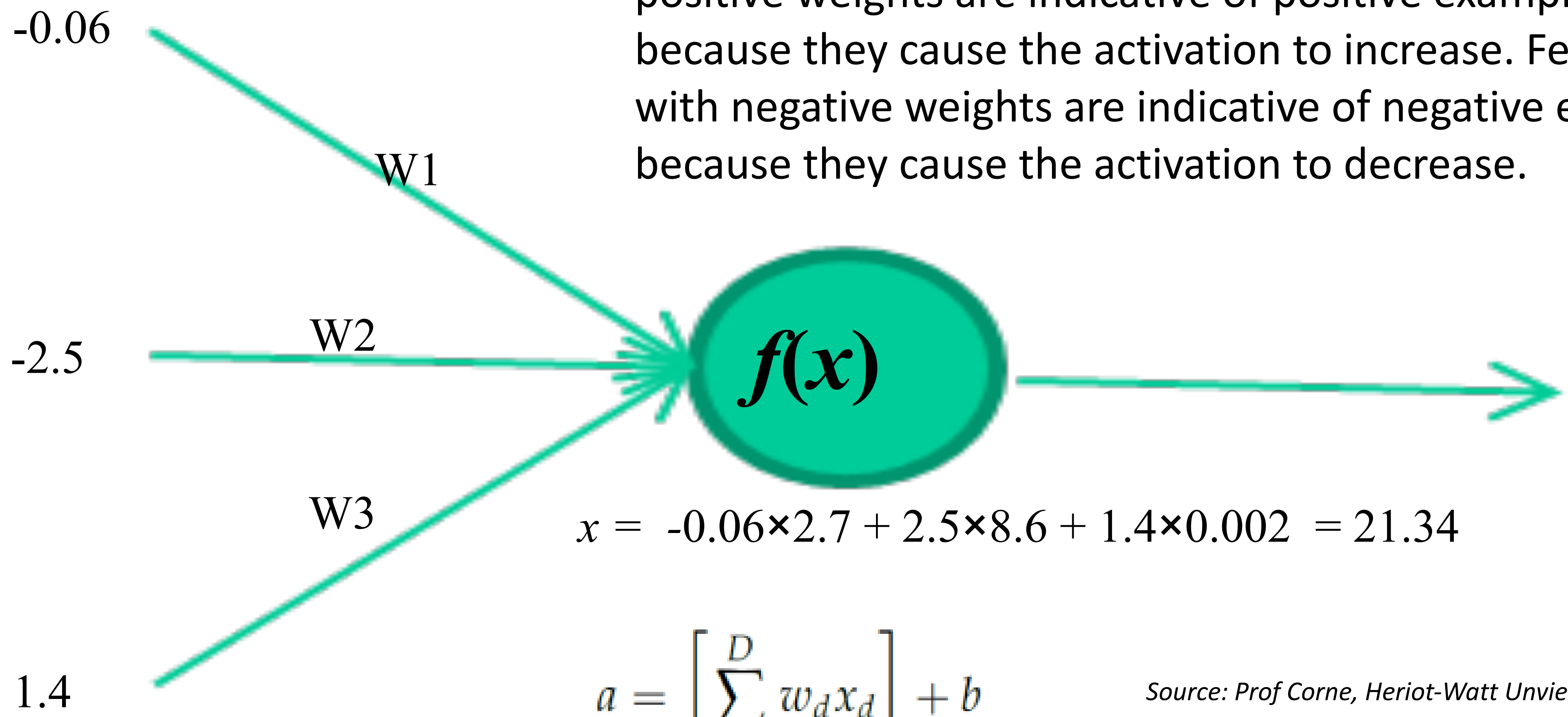


$x = -0.06\times2.7 + 2.5\times8.6 + 1.4\times0.002 = 21.34$

Source: Prof Corne, Heriot-Watt Unviersity, UK

Neuron as a Function

Features with zero weight are ignored. Features with positive weights are indicative of positive examples because they cause the activation to increase. Features with negative weights are indicative of negative examples because they cause the activation to decrease.



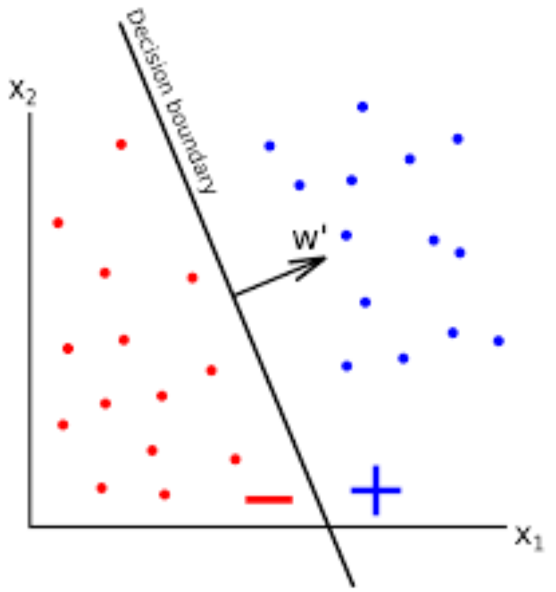
$$x = -0.06\times2.7 + 2.5\times8.6 + 1.4\times0.002 = 21.34$$

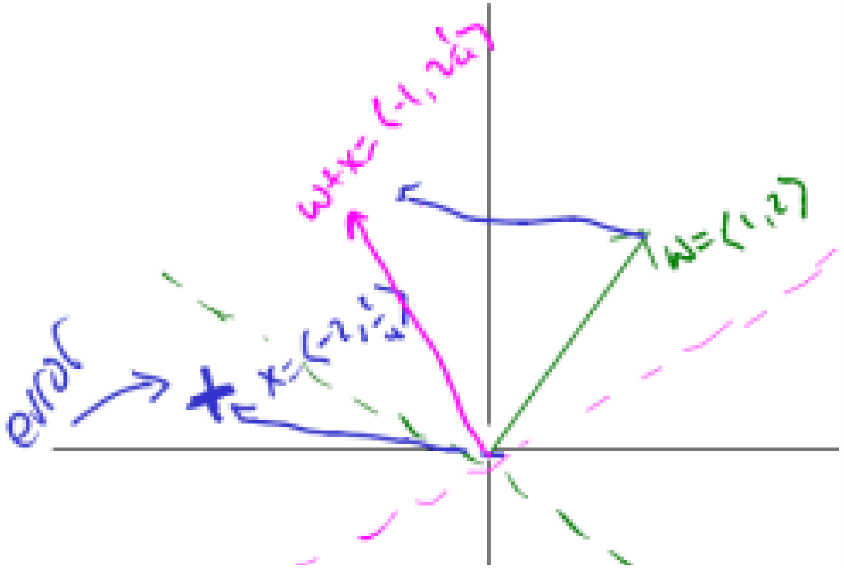
$$a = \left[\sum_{d=1}^{D} w_d x_d\right] + b$$

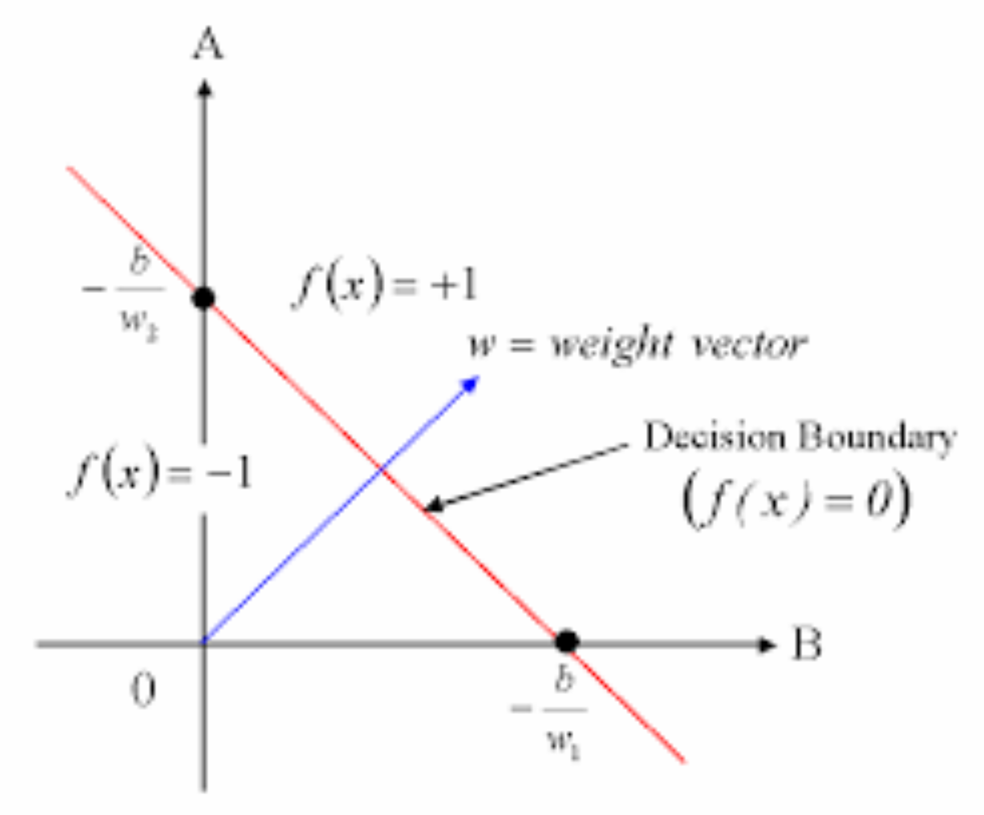
Source: Prof Corne, Heriot-Watt Unviersity, UK

Perceptron Decision Boundary

$$\mathcal{B} = \left\{ x \ : \ \sum_d w_d x_d = 0 \right\}$$







Perceptron Algorithm

Algorithm 5 PerceptronTrain($\mathbf{D}$, $MaxIter$)

$w_d \leftarrow 0$, for all $d = 1 \ldots D$ // initialize weights
$b \leftarrow 0$ // initialize bias
for $iter = 1 \ldots MaxIter$ **do**
  for all $(x,y) \in \mathbf{D}$ **do**
    $a \leftarrow \sum_{d=1}^{D} w_d \, x_d + b$ // compute activation for this example
    if $ya \leq 0$ **then**
      $w_d \leftarrow w_d + yx_d$, for all $d = 1 \ldots D$ // update weights
      $b \leftarrow b + y$ // update bias
    end if
  end for
end for
return $w_0, w_1, \ldots, w_D, b$

Perceptron Algorithm

- Weight w_d is increased by yx_d and the bias is increased by y.
- Goal of the update is to adjust the parameters so that they are "better" for the current example

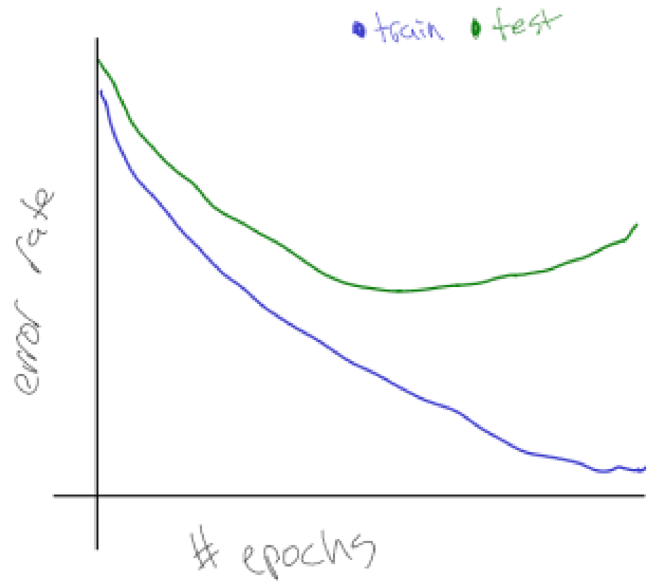
$$
\begin{aligned}
a' &= \sum_{d=1}^{D} w'_d x_d + b' \\
&= \sum_{d=1}^{D} (w_d + x_d) x_d + (b + 1) \\
&= \sum_{d=1}^{D} w_d x_d + b + \sum_{d=1}^{D} x_d x_d + 1 \\
&= a + \sum_{d=1}^{D} x_d^2 + 1 \quad > \quad a
\end{aligned}
$$

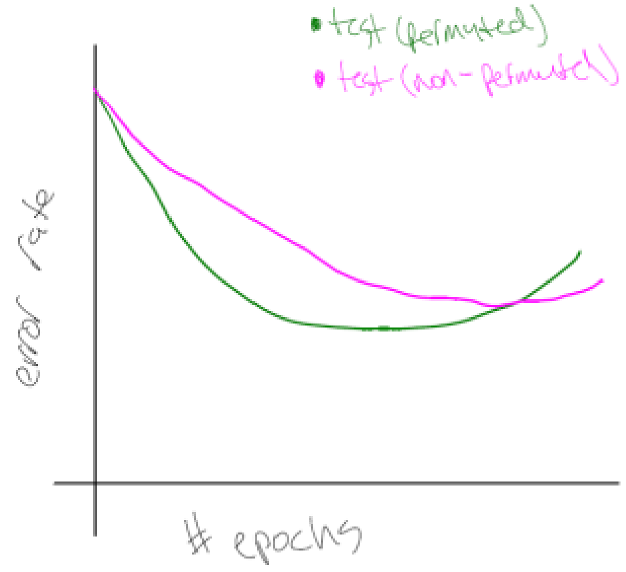
Perceptron Algorithm

- **Online**. This means hat instead of considering the entire data set at the same time, it only ever looks at one example.
- **Error driven.** This means that, so long as it is doing well, it doesn’t bother updating its parameters

Perceptron Algorithm

- If we make many many passes over the training data, then the algorithm is likely to overfit. On the other hand, going over the data only one time might lead to underfitting
- Loop over all the training examples in a constant
- Order is a bad idea. Re-permute the examples in each iteration.





Perceptron Learning

- Iteratively pick a misclassified samples from the training set and apply the perceptron rule
- Each iteration through the training set is an *epoch*
- Continue training until total training set error ceases to improve (convergence)
- Perceptron Convergence Theorem: Guaranteed to find a solution in finite time if a solution exists
  - No. of required iterations <= $(R/gamma)^2$

Disadvantages of Perceptrons

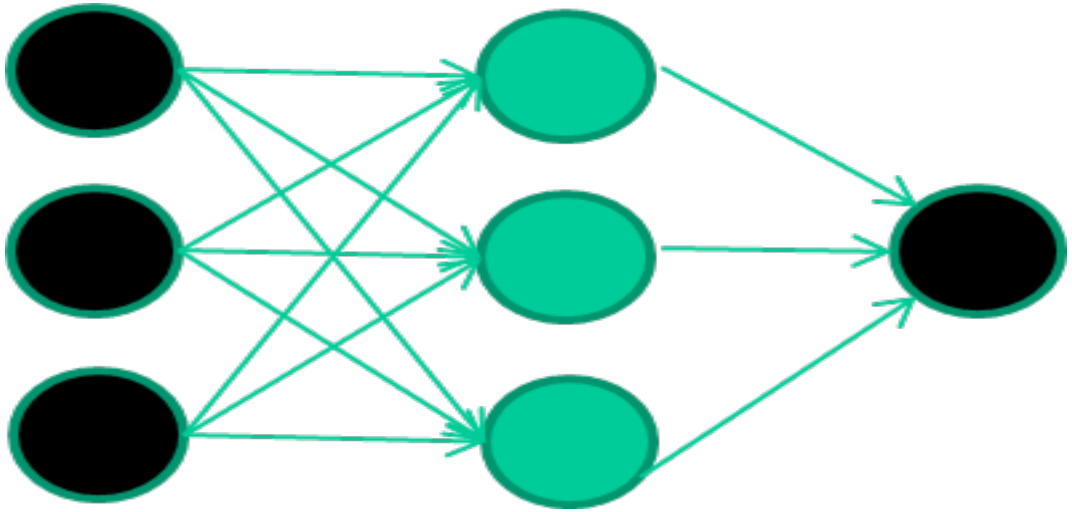
- Can only learn linear decision boundaries
- Requires data to be linearly separable to guarantee convergence
- Even if data is linearly separable, convergence may take long
- Does not generalize easily to more than 2 classes
- Does not provide probabilistic outputs

Neural Networks - Multi Layer Perceptrons

How do they learn?

A dataset

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

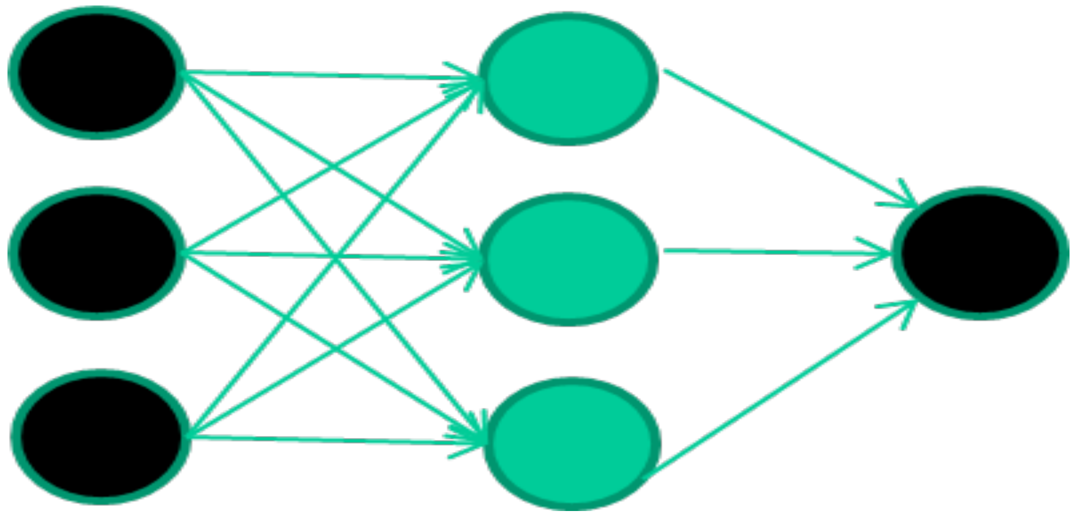
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Initialise with random weights



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

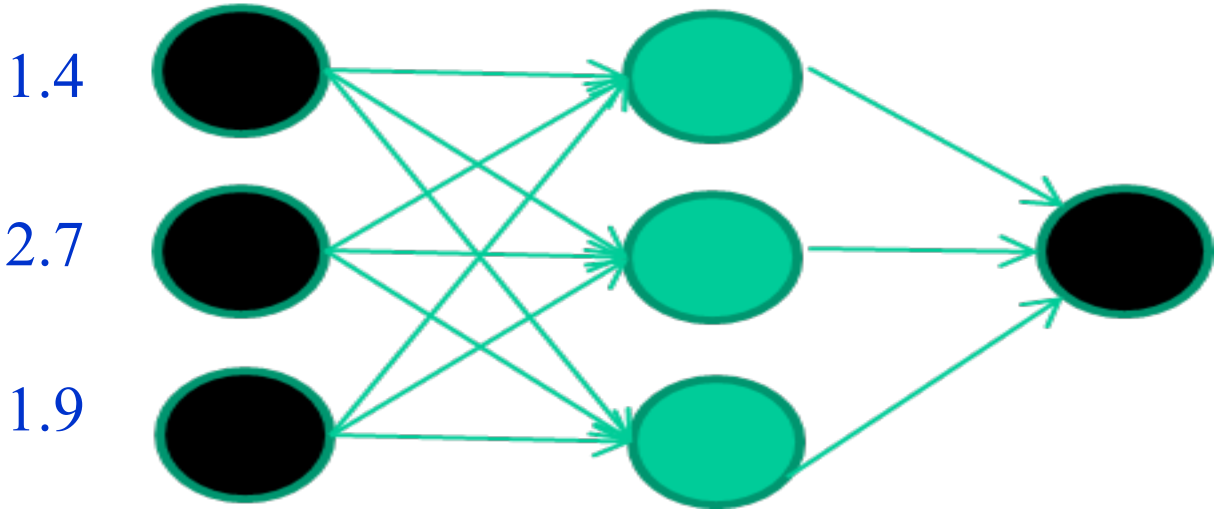
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Present a training pattern



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

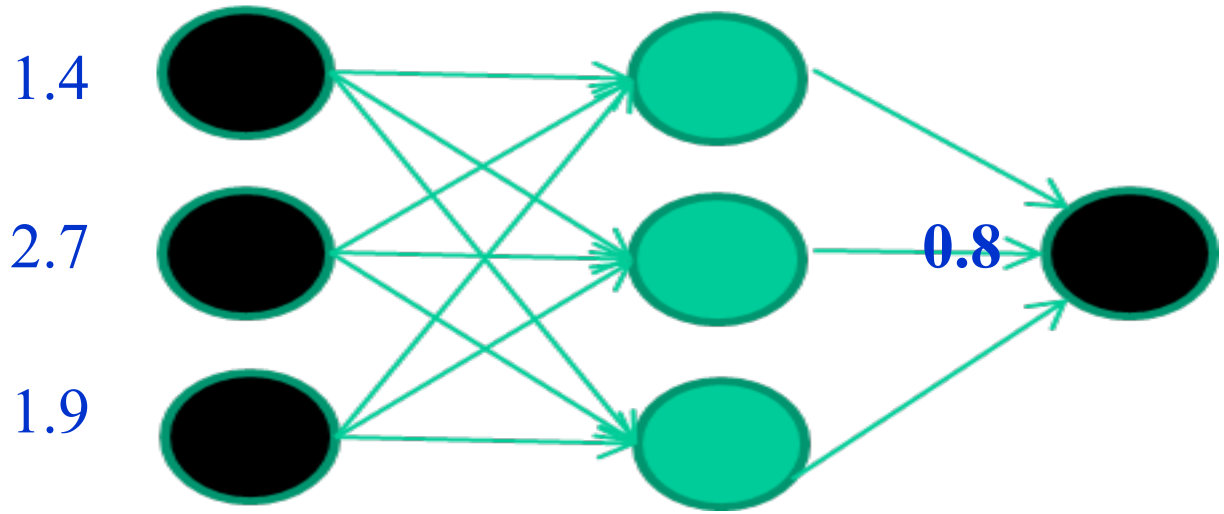
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Feed it through to get output



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

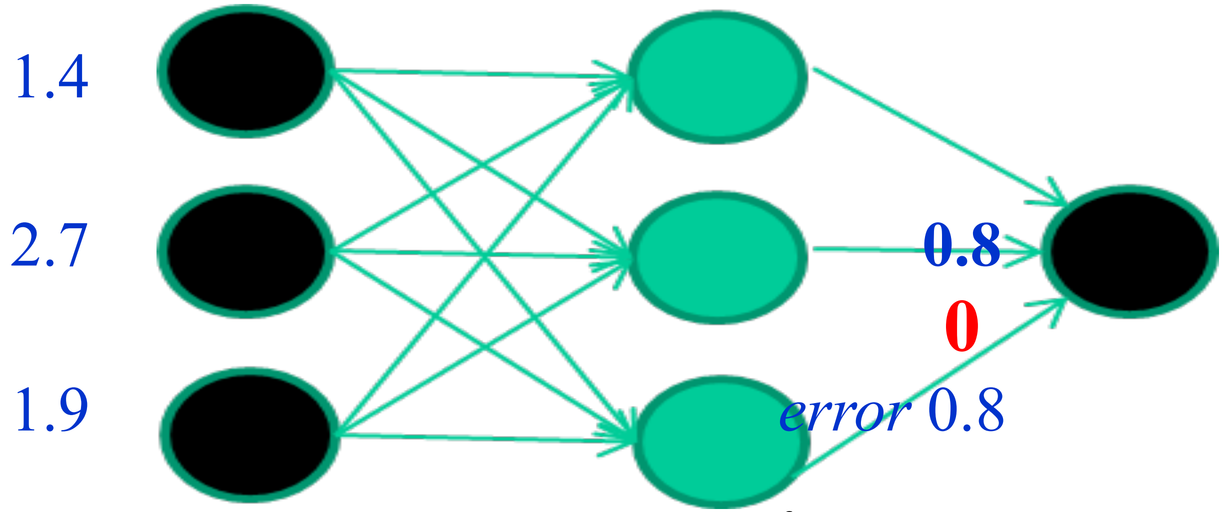
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Compare with target output



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

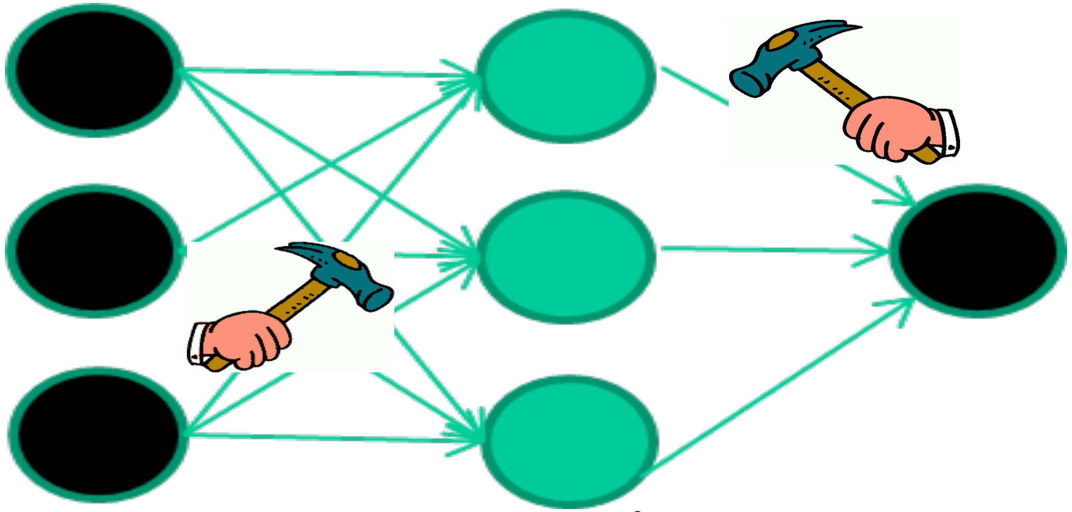
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Adjust weights based on error



Source: Prof Corne, Heriot-Watt Unviersity, UK

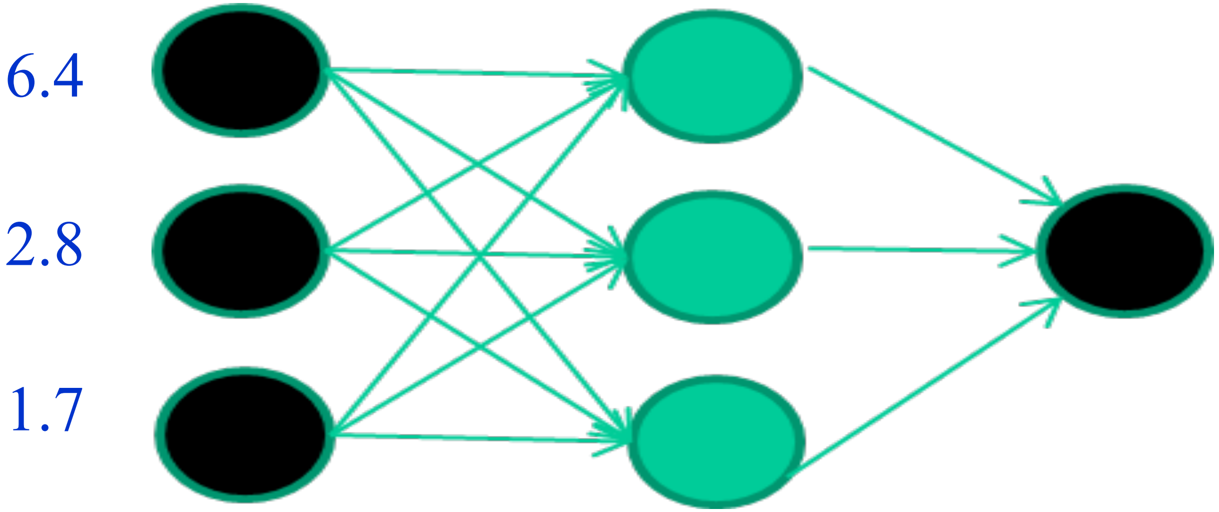
Neural Networks - Multi Layer Perceptrons

How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Present a training pattern



Source: Prof Corne, Heriot-Watt Unviersity, UK

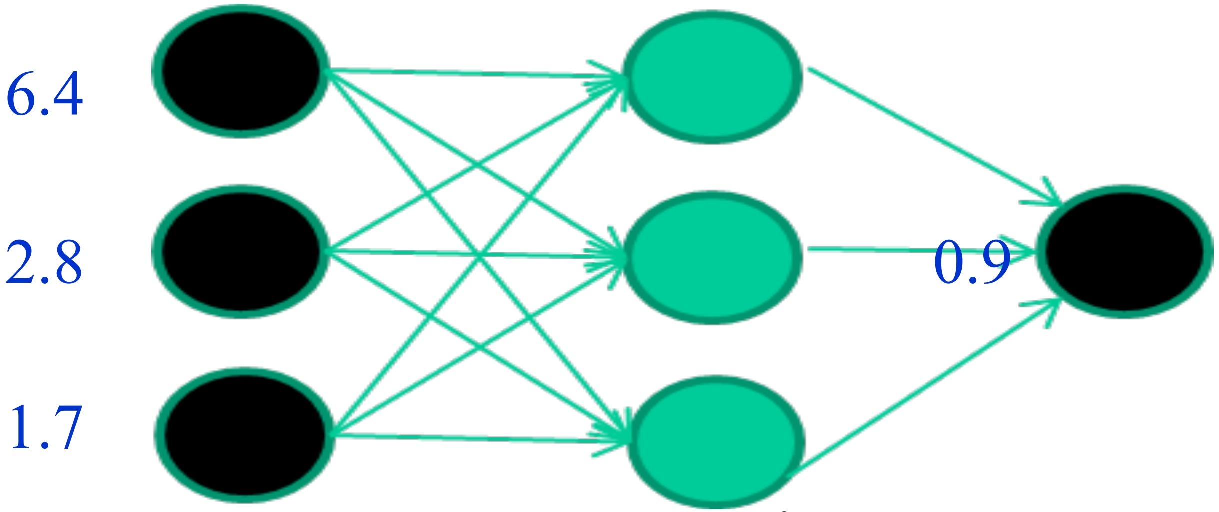
Neural Networks - Multi Layer Perceptrons

How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

Feed it through to get output



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

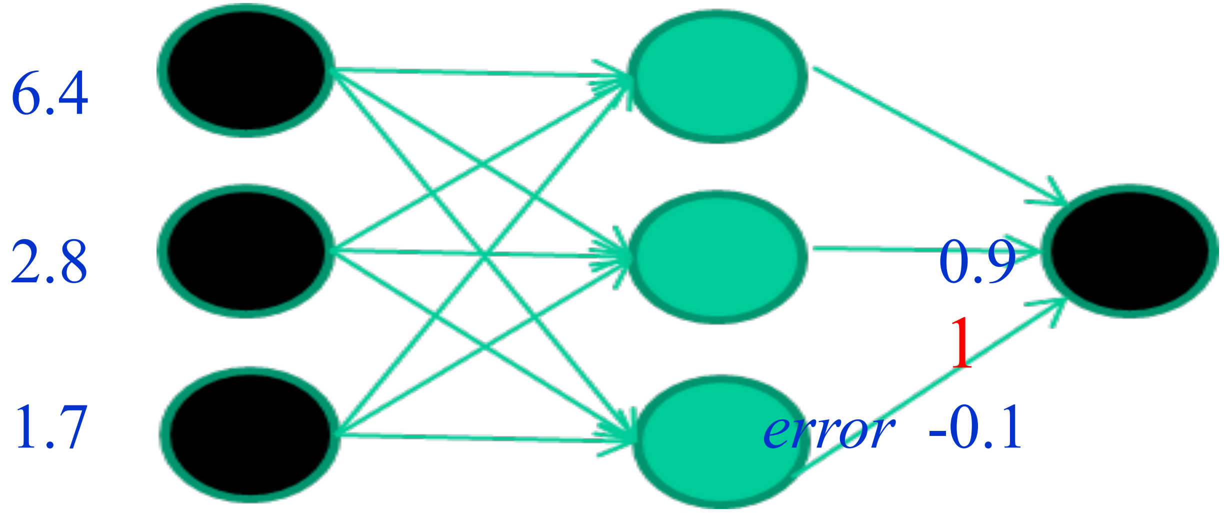
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Compare with target output



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

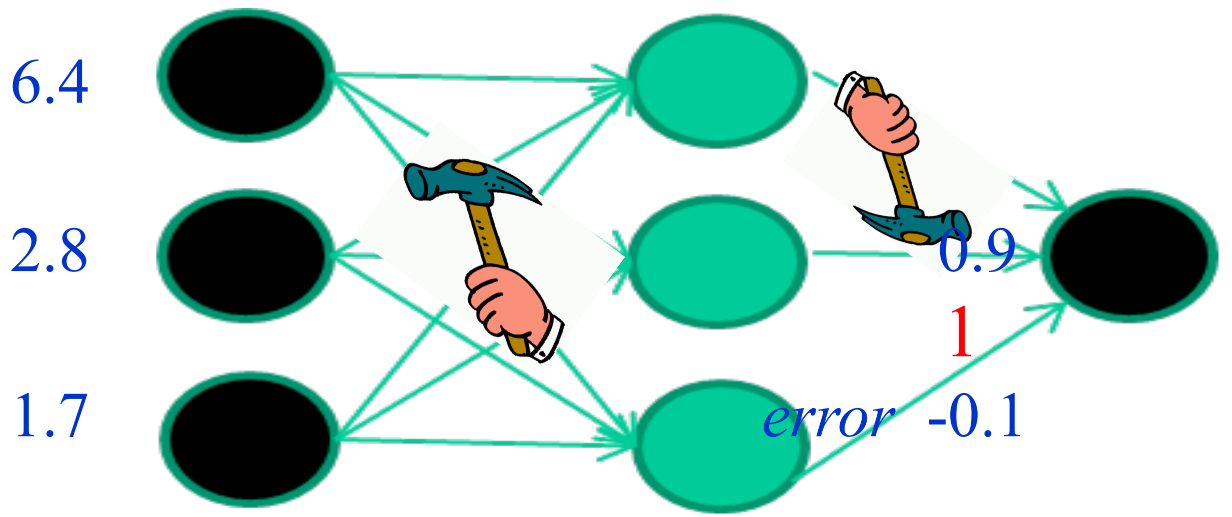
How do they learn?

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

Adjust weights based on error



Source: Prof Corne, Heriot-Watt Unviersity, UK

Neural Networks - Multi Layer Perceptrons

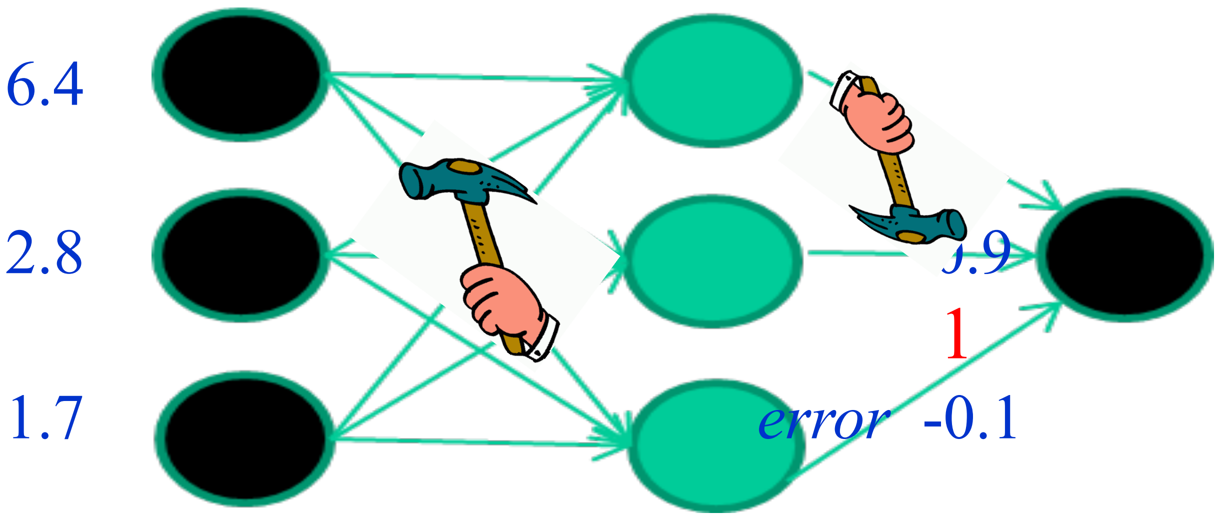
How do they learn?

Source: Prof Corne, Heriot-Watt Unviersity, UK

Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

And so on ….

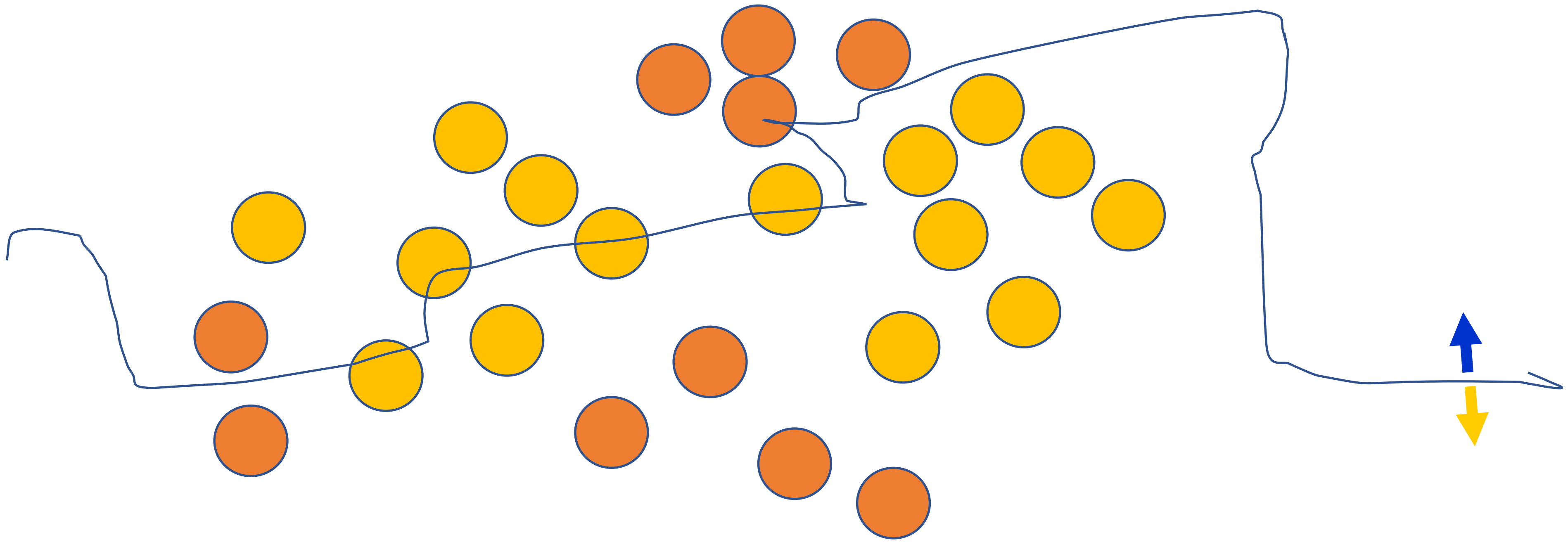


Repeat this thousands, maybe millions of times – each time
taking a random training instance, and making slight
weight adjustments, reduce the error

Called "Gradient Descent"

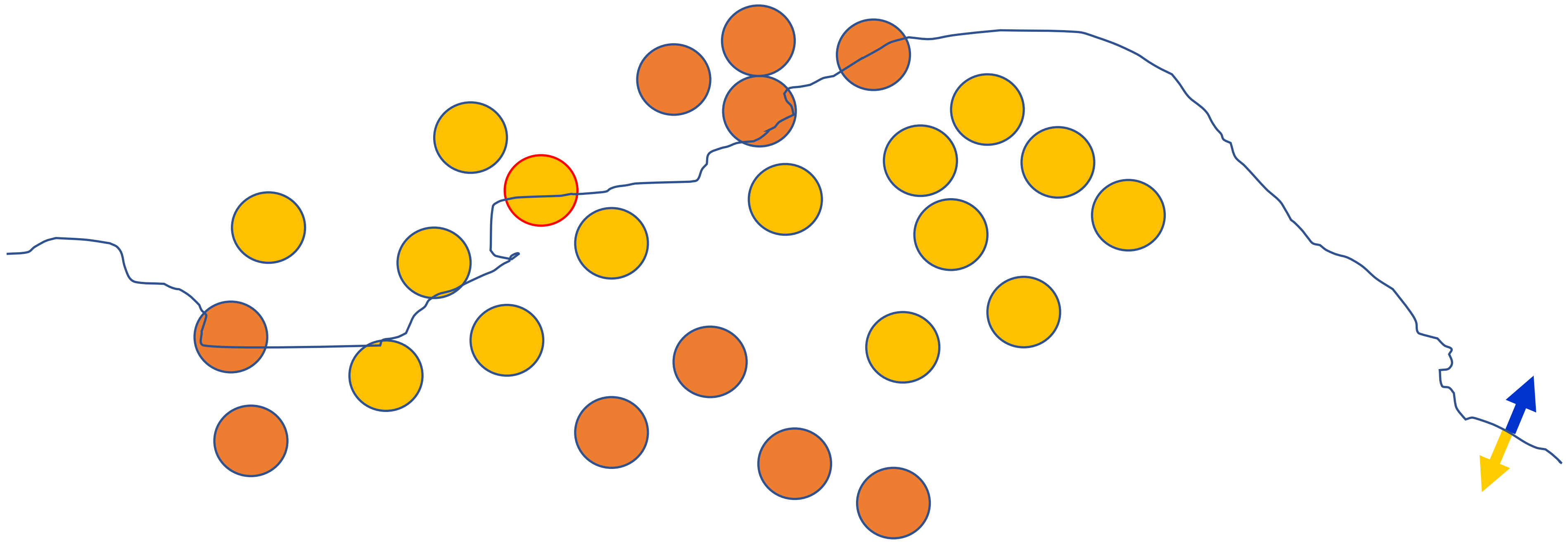
The decision boundary perspective…

Initial random weights



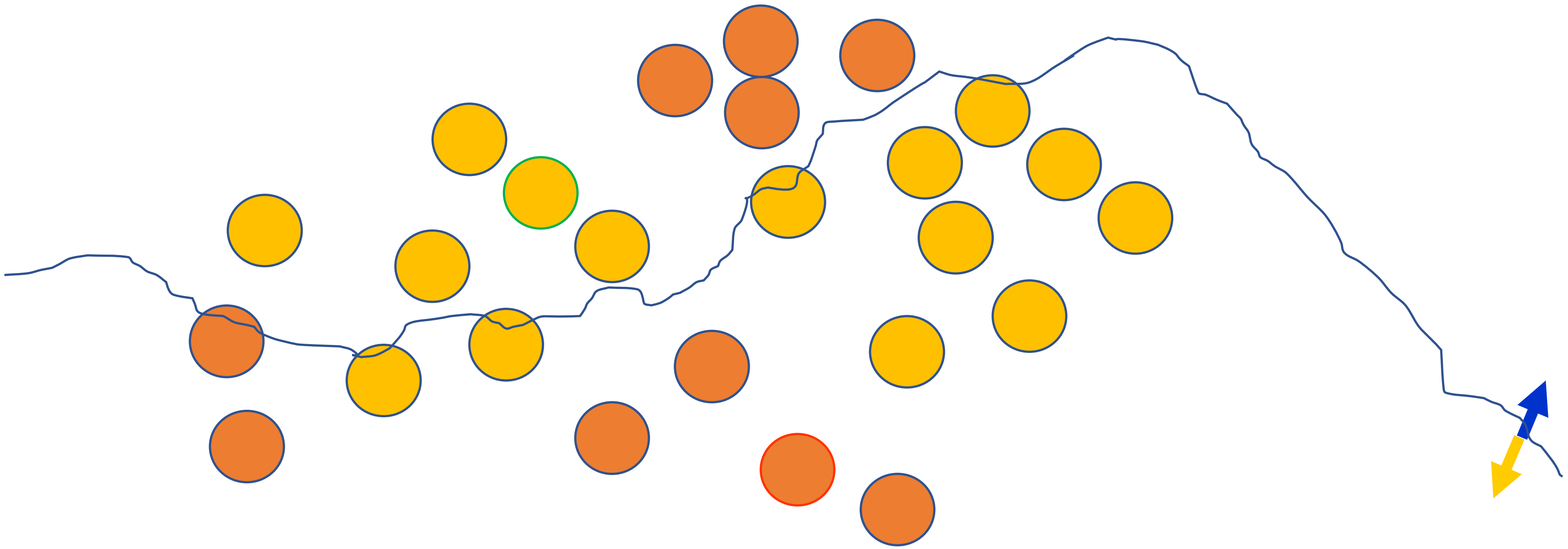
The decision boundary perspective…

Present a training instance / adjust the weights



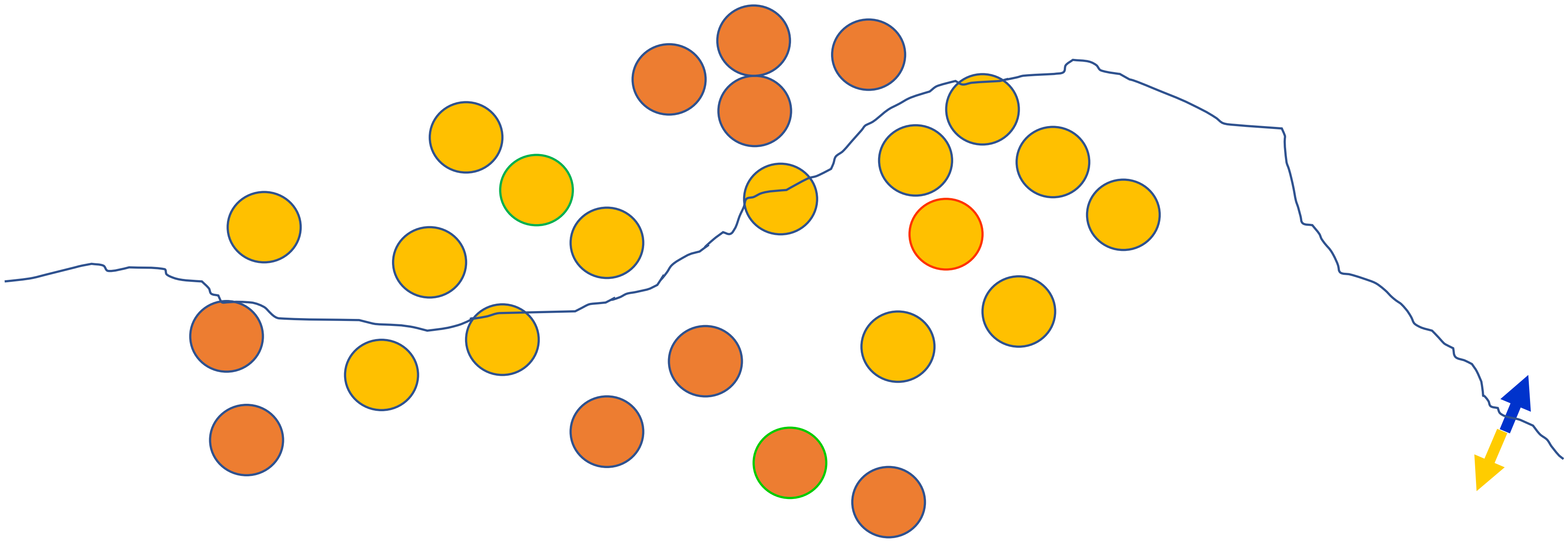
The decision boundary perspective…

Present a training instance / adjust the weights



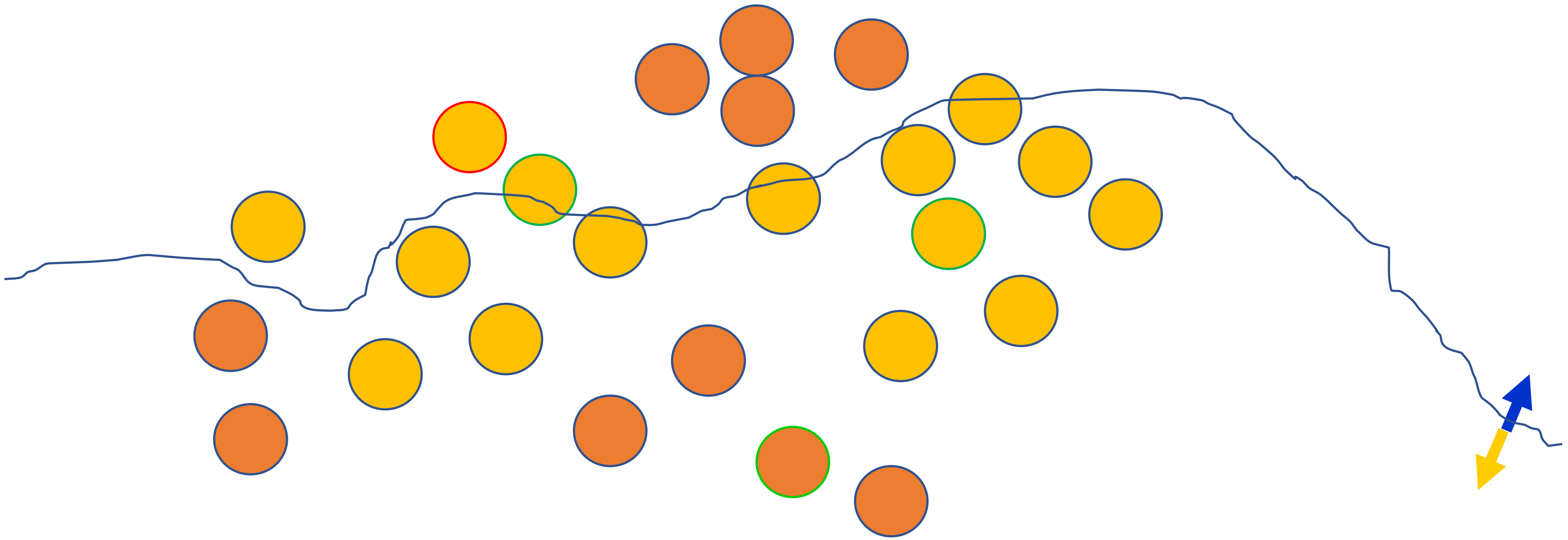
The decision boundary perspective…

Present a training instance / adjust the weights



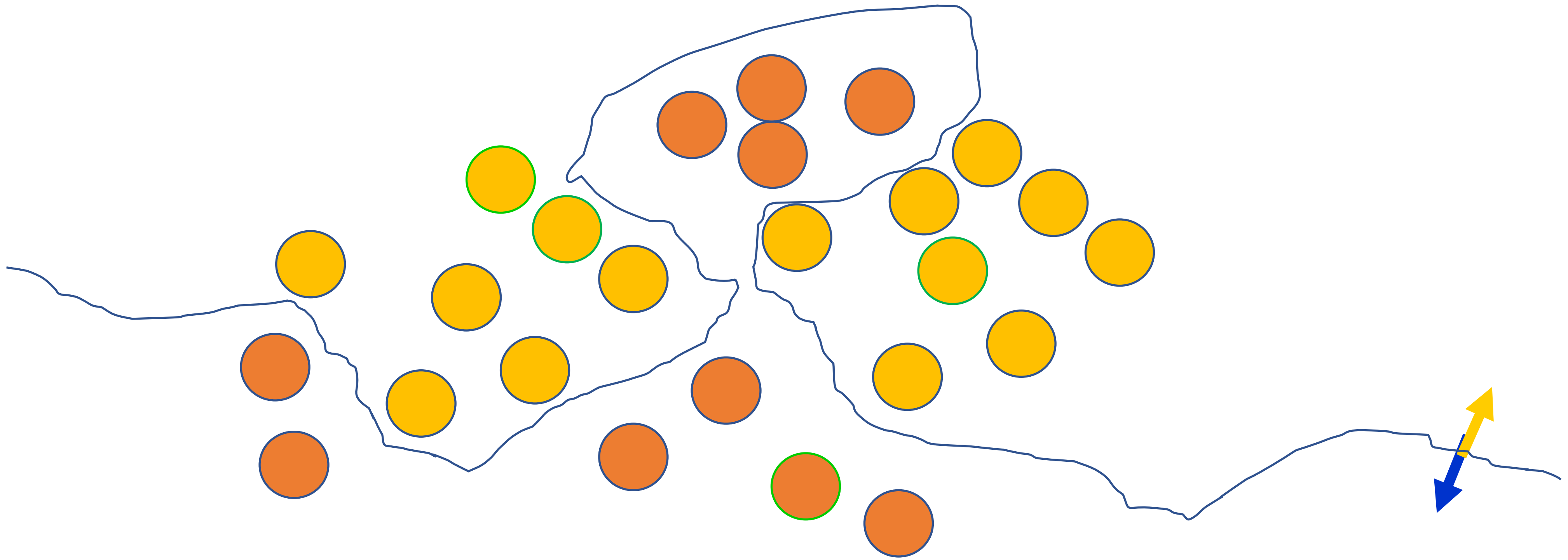
The decision boundary perspective…

Present a training instance / adjust the weights



The decision boundary perspective…

Eventually ….



Neural Networks Training: Backpropagation

First attempt at a training algorithm

- 1. Initialize network with random weights
- 2. For all training cases (called examples):
  - **a.** Present training inputs to network and calculate output
  - **b.** For all layers (starting with output layer, back to input layer):
    - i. Compare network output with correct output (error function)
    - ii. Adapt weights in current layer

Neural Networks Training: Backpropagation

- Method for learning weights in feed-forward (FF) nets

- Can't use Perceptron Learning Rule
  - no teacher values are possible for hidden units

- Use gradient descent to minimize the error
  - propagate deltas to adjust for errors
  - backward from outputs to hidden layers to inputs

Neural Networks Training: Backpropagation

The idea of the algorithm can be summarized as follows :

1. Computes the **error term for the output units** using the observed error.

2. From output layer, repeat

- propagating the error term back to the previous layer and updating the weights between the two layers until the earliest hidden layer is reached.

Neural Networks Training: Backpropagation

- Initialize weights (typically random!)
- In each epoch, do
  - For each example x^j in training set do
    - **forward pass** to compute
      - $y_{pred} = NN(x^j)$
      - error = $(y^j - y_{pred})$ at each output unit
    - **backward pass** to calculate deltas to correct weights
    - update all weights
  - end
- Repeat until training set error stops improving

Neural Networks Training: Backpropagation

Gradient Descent

- Think of the N weights as a point in an N-dimensional space
- Add a dimension for the observed error
- Try to minimize your position on the “error surface”

Neural Networks Training: Backpropagation

Gradient Descent

- Trying to make error decrease the fastest
- Compute:
  - $Grad_E = [dE/dw_1, dE/dw_2, . . ., dE/dw_n]$
- Change i^{th} weight by
  - $delta_{wi} = -\mathit{alpha} * dE/dw_i$

- We need a derivative!
- Activation function must be continuous, differentiable, non-decreasing, and easy to compute

Neural Networks Training: Backpropagation

Updating Hidden-to-Output

- We have teacher supplied desired values

- $\text{delta}_{wji} = \alpha * (t_i - y_i) * g'(z_i) * a_j$

  $= \alpha * (t_i - y_i) * y_i * (1 - y_i) * a_j$

  - for sigmoid the derivative is, $g'(x) = g(x) * (1 - g(x))$

Here we have general formula with derivative, next we use for sigmoid

Learning rate

miss

Derivative of activation function

Neural Networks Training: Backpropagation

Updating interior weights

- Layer k units provide values to all layer k+1 units
  - “miss” is ***sum of misses*** from all units on k+1
  - $miss_j = \Sigma [\ a_i(1- a_i)\ (t_i - a_i)\ w_{ji}\ ]$
  - weights coming into this unit are adjusted based on their contribution

$$delta_{kj} = \alpha * I_k * a_j * (1 - a_j) * miss_j$$

For layer k+1

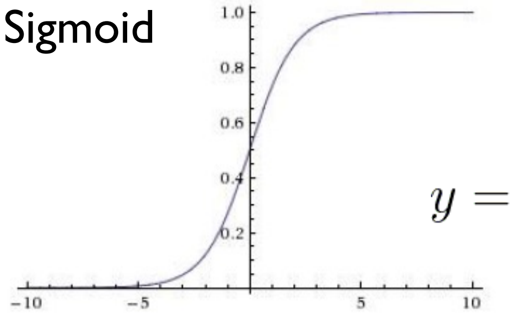
Compute deltas

Making Choices

- Number of hidden layers – *empirically determined*
  - Too few ==> can't learn
  - Too many ==> poor generalization
- Number of neurons in each hidden layer – *empirically determined*
- Activation functions
- Error/loss functions
- Learning rate
- Gradient descent methods

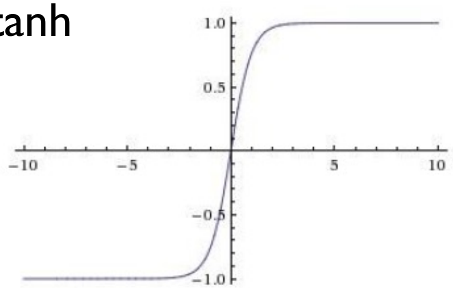
Activation Functions

Sigmoid



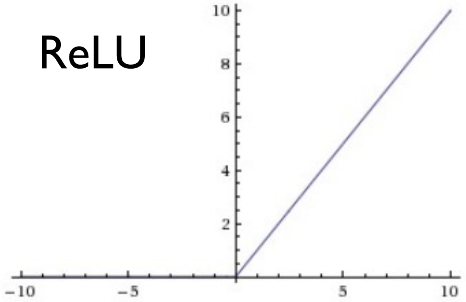
$$y = \frac{1}{1 + e^{-x}}$$

tanh



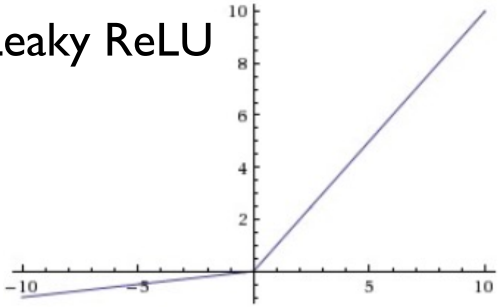
$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

ReLU



$$y = max(0, x)$$

Leaky ReLU



$$y = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if } otherwise \end{cases}$$

Loss Functions

- Euclidean loss / Squared loss $$L = \frac{1}{2} \|x_i - y_i\|_2^2$$
  - Derivative w.r.t. x_i $$\frac{\partial L}{\partial x_i} = x_i - y_i$$

- Soft-max loss/multinomial logistic regression loss

$$p_i = \frac{e^{x_i}}{\sum_k e^{x_k}} \qquad L = -\sum_i y_i log(p_i)$$

  - Derivative w.r.t. x_i $$\frac{\partial L}{\partial x_i} = p_i - y_i$$
  - Also called: Cross-entropy loss

Gradient Descent Methods

- Batch gradient descent (vs) Stochastic gradient descent (vs) Mini-batch stochastic gradient descent
  - Mini-batch SGD the most popularly used
- Using momentum
- Setting learning rate
  - Fixed learning rate
  - Using learning rate schedules
  - Adaptive learning rate methods: Adam, Adadelta, Adagrad, RMSProp

Readings

- "Introduction to Machine Learning" by Ethem Alpaydin, Chapters 11.1-11.11
- Bishop, PRML, Sec 5.1-5.3, 5.5
- Perceptron Convergence proof: https://www.cse.iitb.ac.in/~shivaram/teaching/old/cs344+386-s2017/resources/classnote-1.pdf