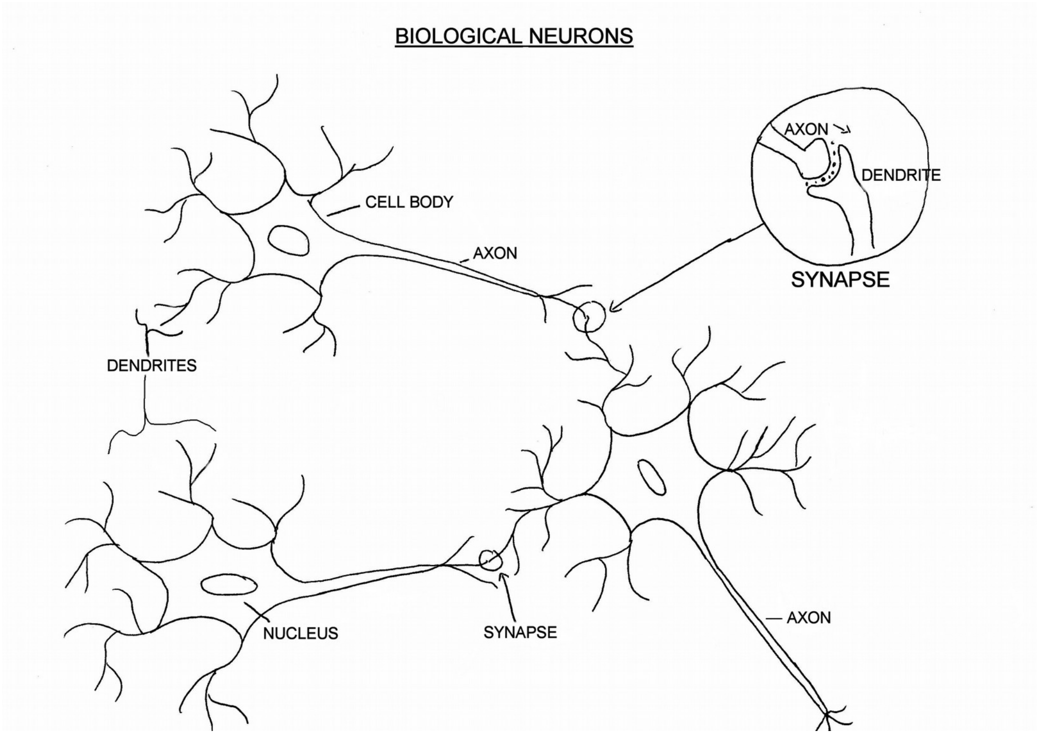
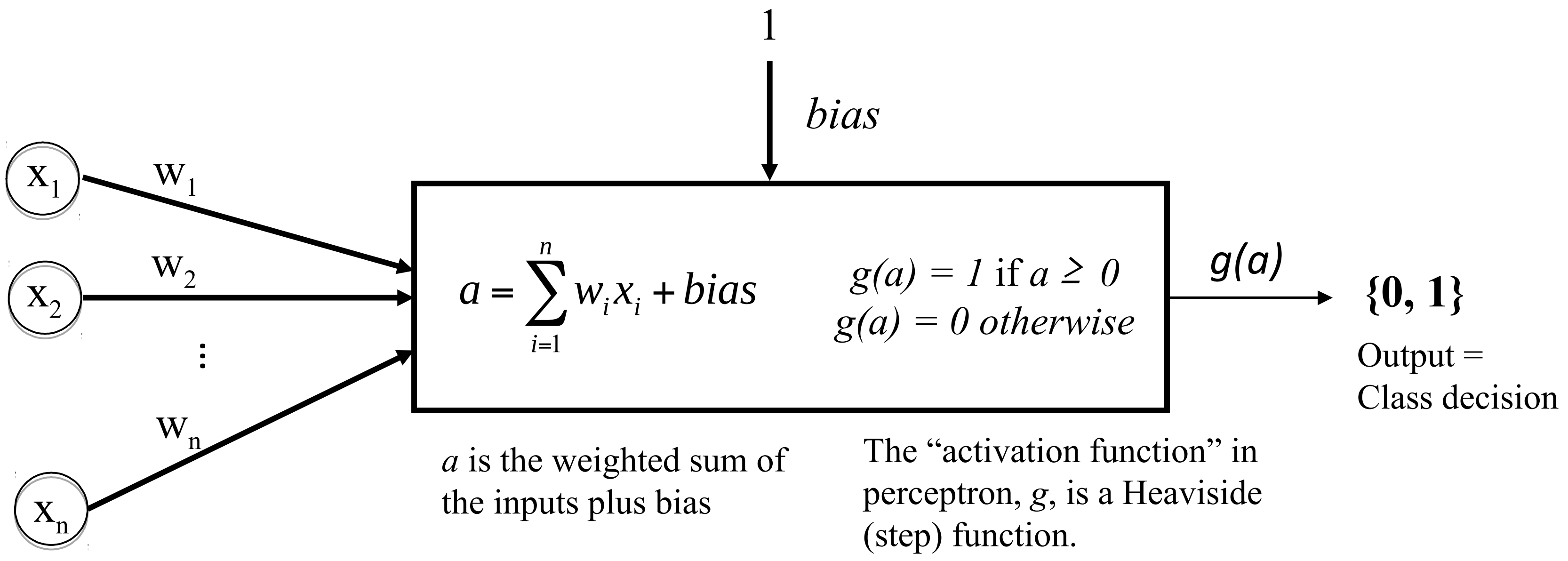
Artificial Neural Networks

Source of inspiration: (Biological) Neural Networks

- Brain as a network of “neurons”
- The interaction between them produces something interesting: intelligence
- The building blocks are neurons.
- Neuron:
  - Receives signal from multiple inputs
  - Strength of the signal is affected by “synaptic” weights.
  - If the overall signal is above a threshold, the neuron fires (sends a signal to its outputs.)



Modeling a Single Neuron: Perceptron



$$a = \sum_{i=1}^{n} w_i x_i + bias$$

$$g(a) = 1 \text{ if } a \geq 0$$
$$g(a) = 0 \text{ otherwise}$$

a is the weighted sum of the inputs plus bias

The "activation function" in perceptron, *g*, is a Heaviside (step) function.

Inputs: $x_1, x_2, \ldots, x_n$

Parameters: $w_1, w_2, \ldots, w_n$ and *bias*

Output: g(a)

Sometimes *bias* is represented as another weight (w_0) in which case we assume there is a virtual input x_0 which is always 1:

$$a = \sum_{i=0}^{n} w_i x_i$$

How many parameters are required to specify a perceptron in an *n*-dimensional input space?

Application: Decision Making and Classification

- Perceptron can be seen as a predicate
  - given a vector x
  - f(x) = 1 if that predicate over x is true
  - f(x) = 0 if the predicate is false

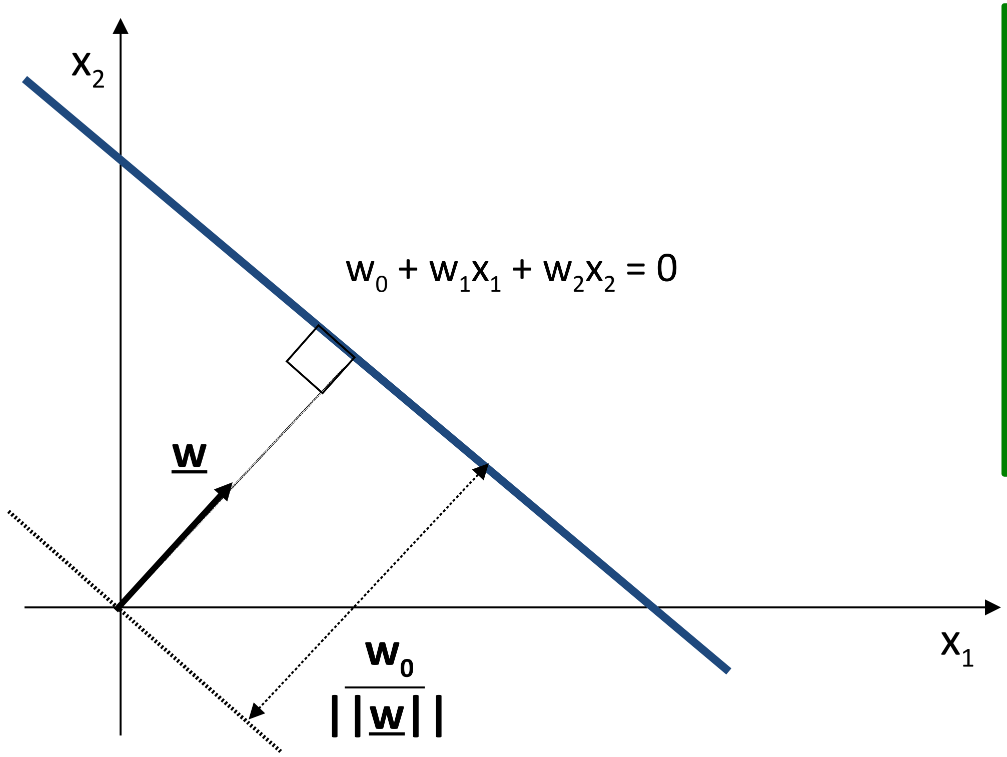
- Therefore it can be used for decision making and binary classification problems:
  - spam *vs* non-spam
  - diseased *vs* healthy

- In binary classification:
  - given input vector x
  - f(x) = 1 if x is in positive class
  - f(x) = 0 if x is in the negative class

Geometric interpretation of weights and bias

Since the transition in output happens at $a = 0$, the equation of the decision boundary is:

$$a = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$$



x_2
$w_0 + w_1x_1 + w_2x_2 = 0$
$\underline{\mathbf{w}}$
$\frac{w_0}{\|\underline{\mathbf{w}}\|}$
x_1

- How do we plot the decision boundary?
- What does a decision boundary look like in a problem with 3 input features (attributes)?

Notes on visualising the decision boundary

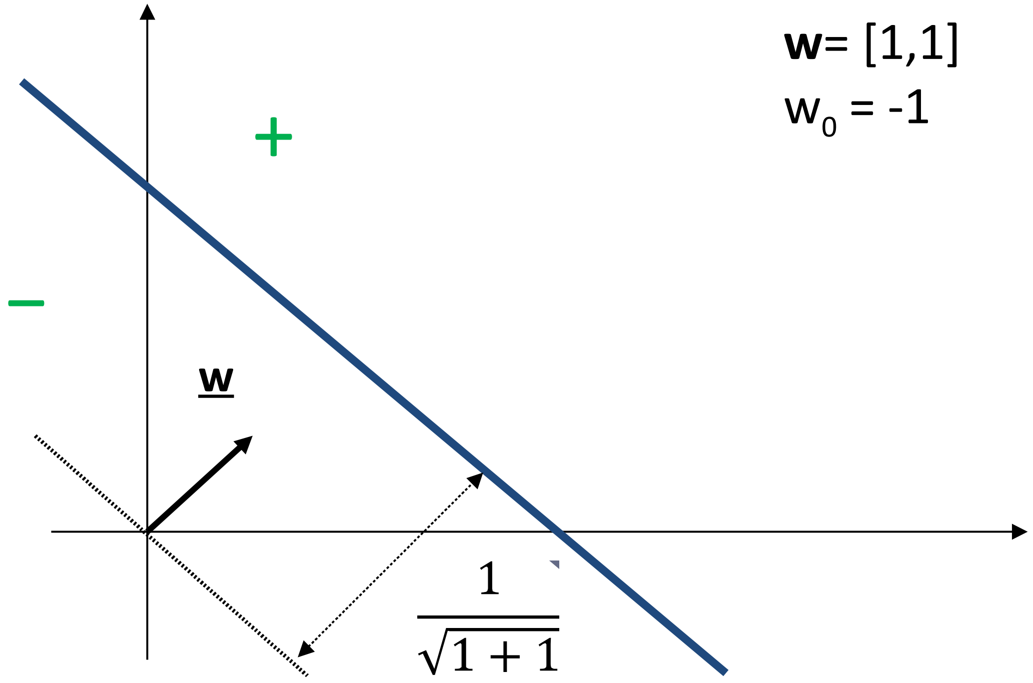
- An easy way of finding a linear decision boundary in 2D is to find two points that lie on it (e.g. set x_1 to zero and find x_2 and vice versa)

- The direction of the vector **w** (without the bias) shows on which side of the hyper-plane patterns will be classified as positive (output will be 1).

- If w_0 (bias) is positive then the origin is on the positive side of the discriminator (line), otherwise on the negative side.

- Example: find the decision boundary of this perceptron:
  - Two inputs x_1 and x_2
  - The weights are $w_1 = 1.0$ and $w_2=1.0$
  - The bias $w_0 = -1.0$

Example: answer



w= [1,1]

w_0 = -1

Perceptron learning (for one neuron)

- **Given**: a data set - a collection of training vectors in the form (x_1, x_2, …, x_n, t)
- **We want**: a perceptron that models the data.
- Initialise the weights and bias (randomly).
- Iterate through (training) examples. Classify each example with current values of weights and bias. If the example is classified correctly, then don't do anything. If the example is misclassified then change the weights and bias:

$$w_j \leftarrow w_j + \eta\, x_j\, (t - y)$$

$$bias \leftarrow bias + \eta\, (t - y)$$

Where:
x_j : the value of *j*-th feature of the input pattern **x**
w_j: the *j*-th weight
η: learning rate
t: target/desired value
y: perceptron output

Example

```
weights = [1.0, 1.0]
bias = -2.0

eta = 0.5 # learning rate
max_epochs = 500

examples = [
  ((0, 4), 0),
  ((-2, 1), 1),
  ((3, 5), 0),
  ((1, 1), 1),
  ]
```

Example

```
---------------
epoch:  1
weights:  [1.0, 1.0]
bias:  -2.0
pattern, output, target: (0, 4), 1, 0
updating the weights and bias to:  [1.0, -1.0] -2.5
pattern, output, target: (-2, 1), 0, 1
updating the weights and bias to:  [0.0, -0.5] -2.0
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [0.5, 0.0] -1.5

---------------
epoch:  2
weights:  [0.5, 0.0]
bias:  -1.5
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 0, 1
updating the weights and bias to:  [-0.5, 0.5] -1.0
pattern, output, target: (3, 5), 1, 0
updating the weights and bias to:  [-2.0, -2.0] -1.5
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [-1.5, -1.5] -1.0
```

Example

```
---------------
epoch:  3
weights:  [-1.5, -1.5]
bias:  -1.0
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 1, 1
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [-1.0, -1.0] -0.5


---------------
epoch:  4
weights:  [-1.0, -1.0]
bias:  -0.5
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 1, 1
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [-0.5, -0.5] 0.0
```

Example

```
---------------
epoch:  5
weights:  [-0.5, -0.5]
bias:  0.0
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 1, 1
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [0.0, 0.0] 0.5


---------------
epoch:  6
weights:  [0.0, 0.0]
bias:  0.5
pattern, output, target: (0, 4), 1, 0
updating the weights and bias to:  [0.0, -2.0] 0.0
pattern, output, target: (-2, 1), 0, 1
updating the weights and bias to:  [-1.0, -1.5] 0.5
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [-0.5, -1.0] 1.0
```

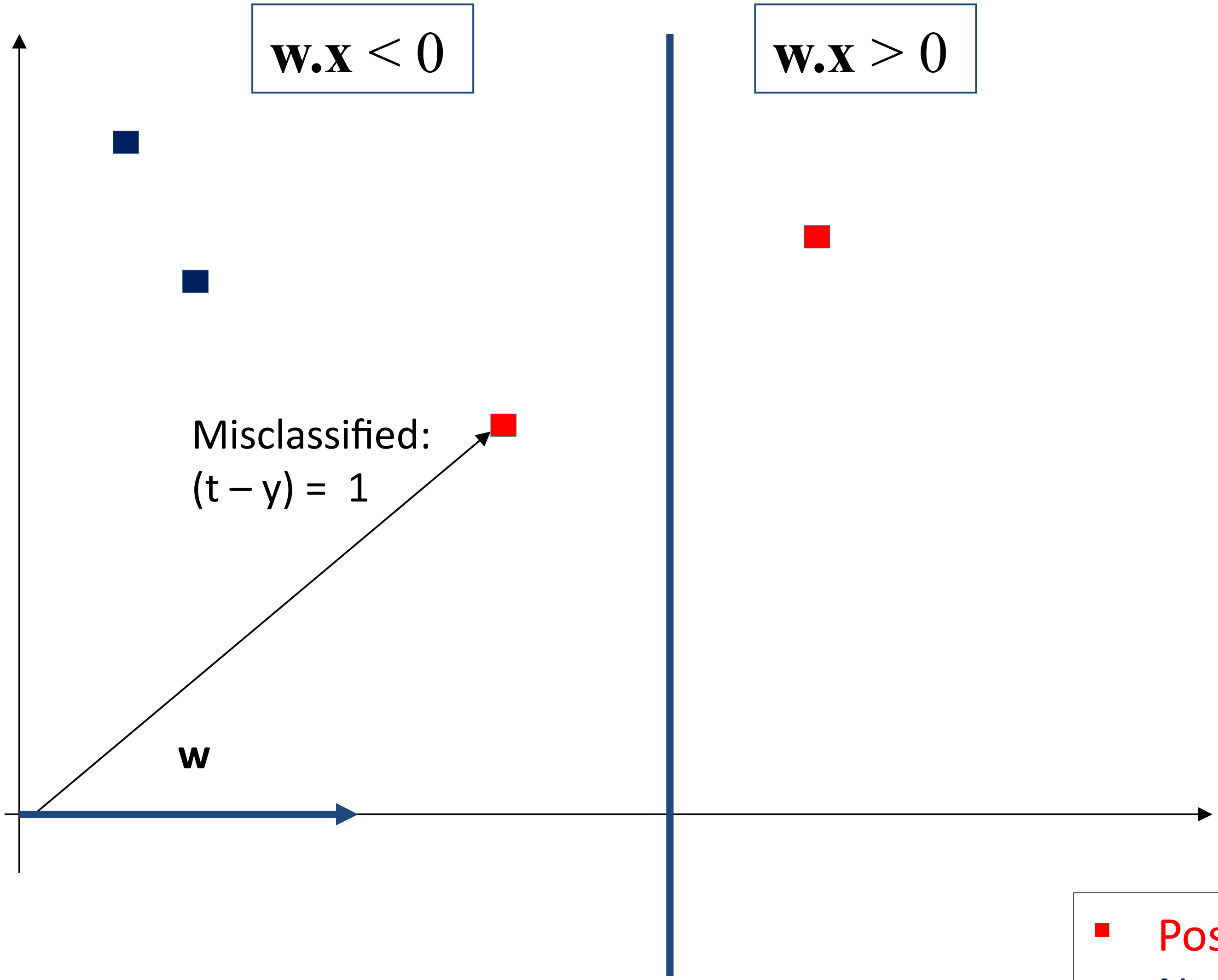
Example

```
---------------
epoch:  7
weights:  [-0.5, -1.0]
bias:  1.0
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 1, 1
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 0, 1
updating the weights and bias to:  [0.0, -0.5] 1.5


---------------
epoch:  8
weights:  [0.0, -0.5]
bias:  1.5
pattern, output, target: (0, 4), 0, 0
pattern, output, target: (-2, 1), 1, 1
pattern, output, target: (3, 5), 0, 0
pattern, output, target: (1, 1), 1, 1
```

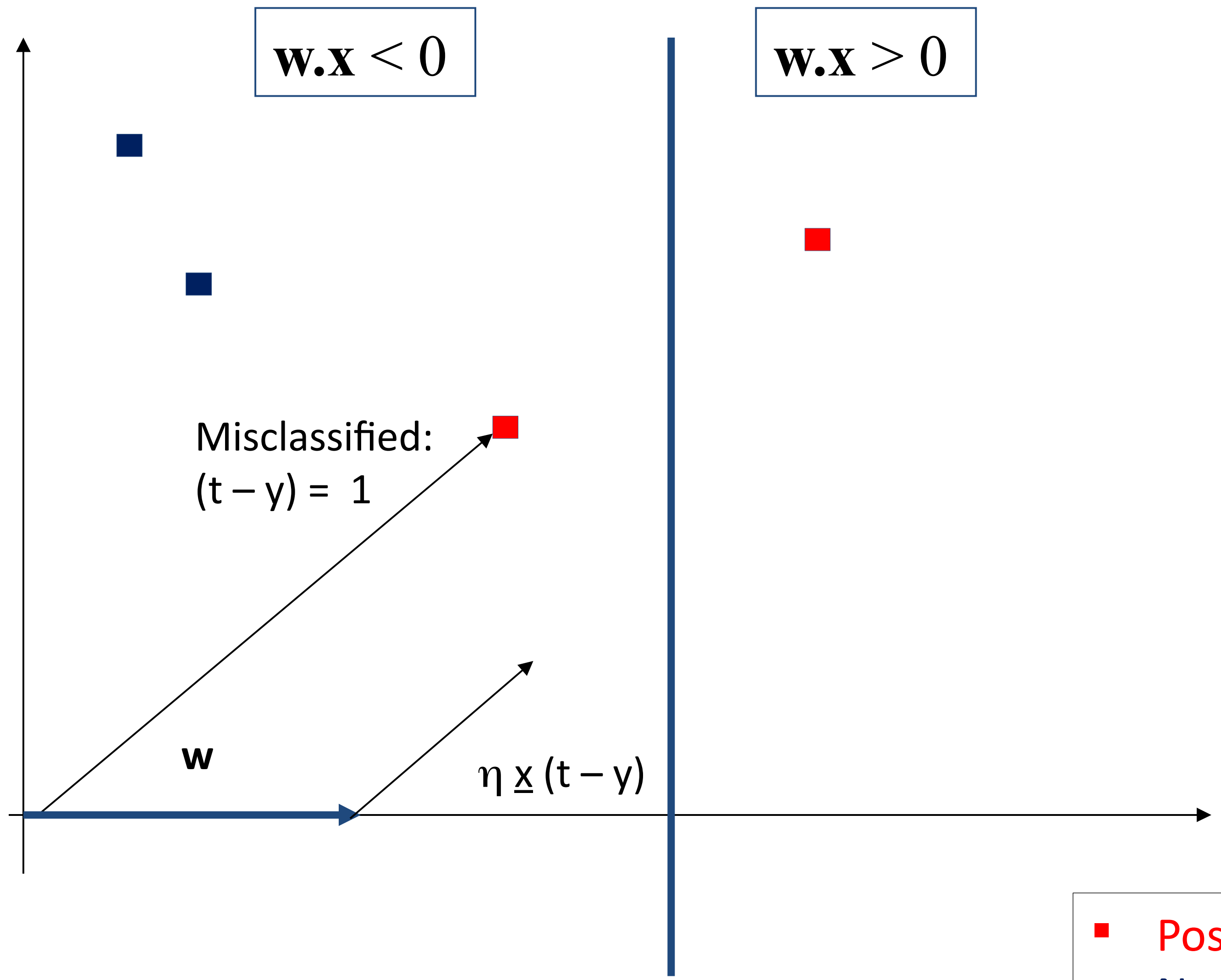
If examples are linearly separable, then using perceptron learning, the weights and bias will (in finite time) converge to values that produce a perfect separation.

Geometric interpretation of weight update



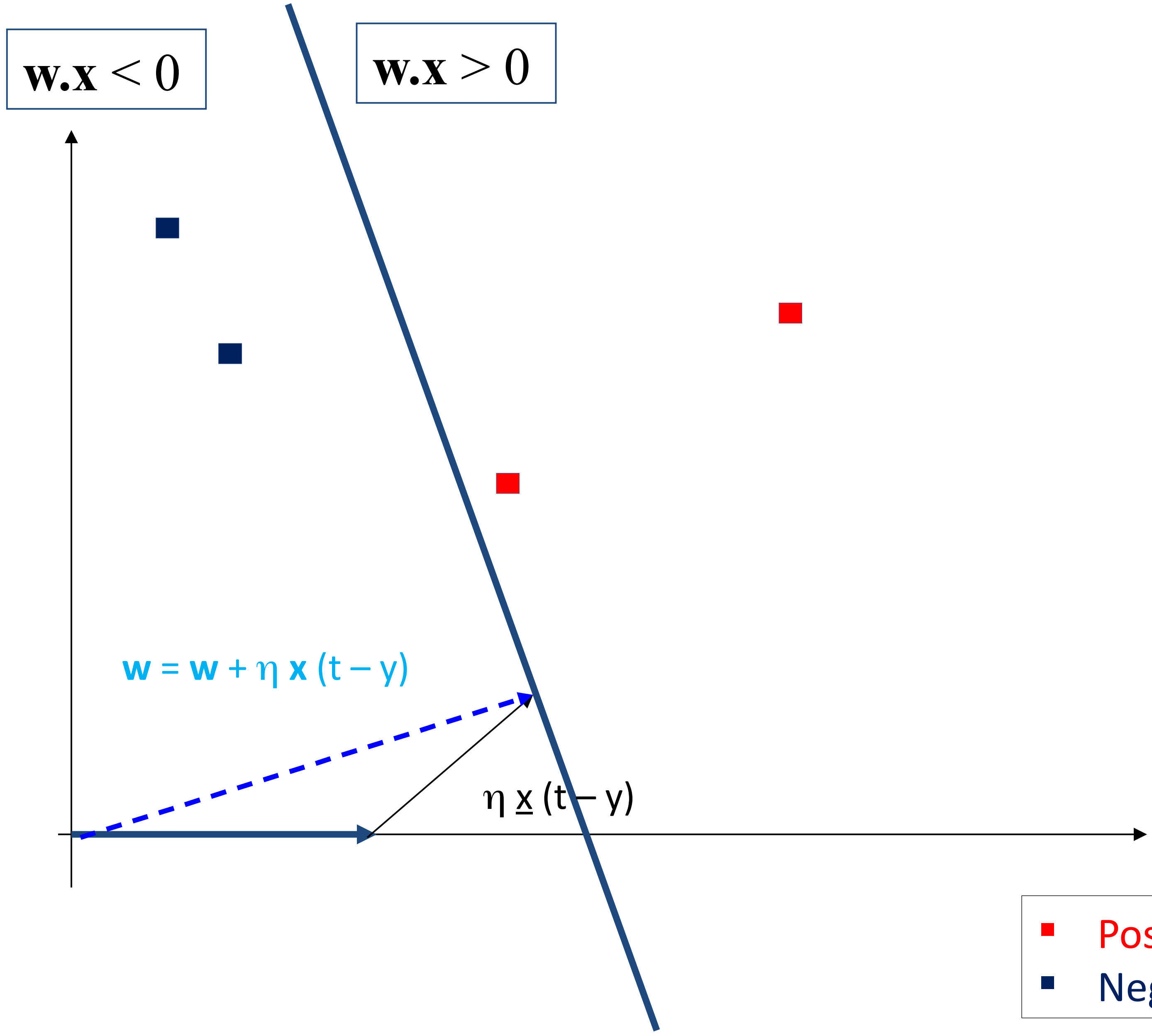
🟥 Positive example
🟦 Negative example

Geometric interpretation of weight update



🟥 Positive example
🟦 Negative example

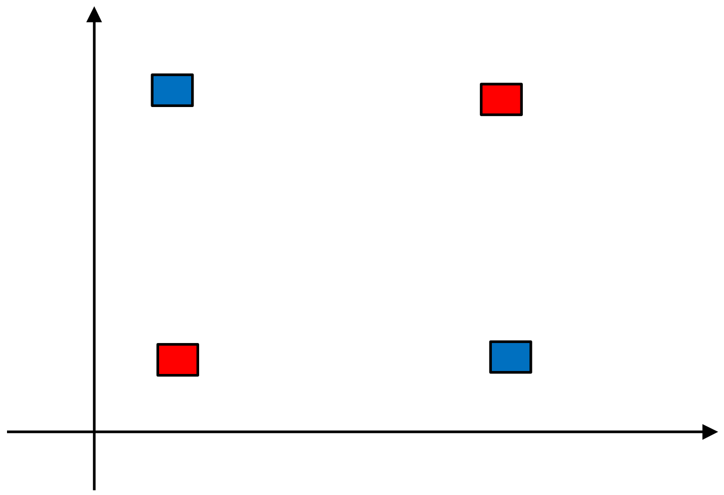
Geometric interpretation of weight update


$\mathbf{w.x} < 0$
$\mathbf{w.x} > 0$
$\mathbf{w} = \mathbf{w} + \eta\, \mathbf{x}\, (t - y)$
$\eta\, \underline{x}\, (t - y)$

🟥 Positive example
🟦 Negative example

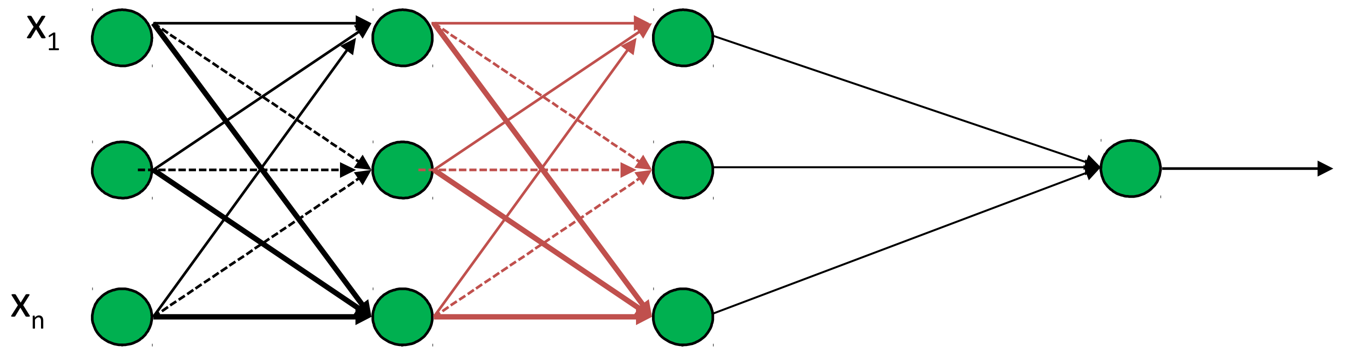
Limitations of single perceptron

The "XOR" problem



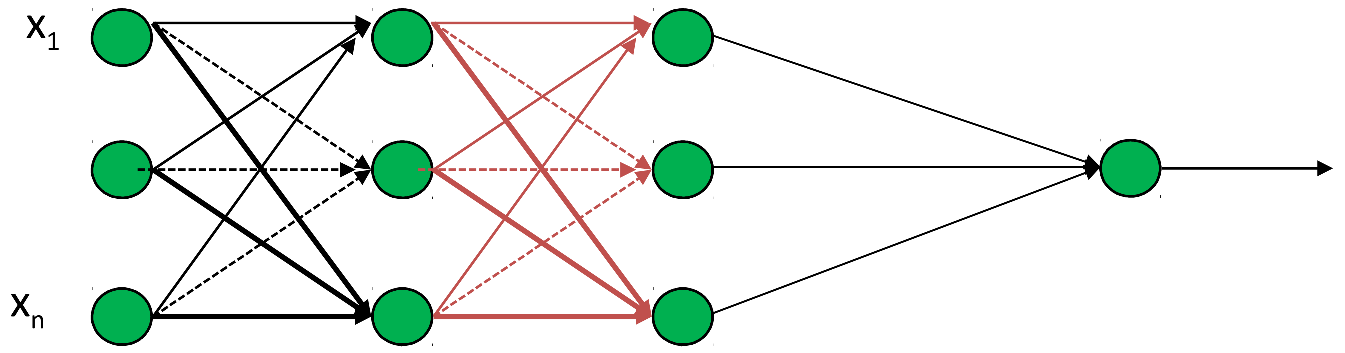
Can a single perceptron learn this problem?

Solution: Multi Layer Perceptrons (MLPs)



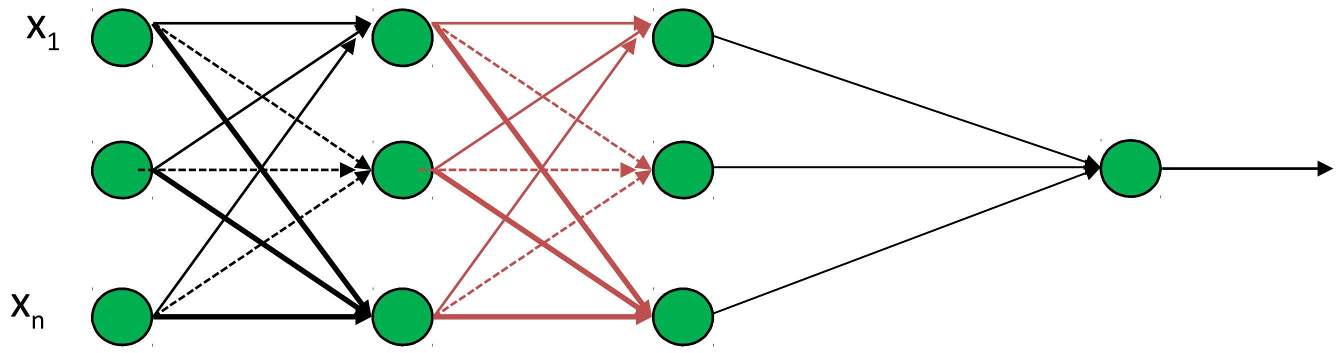
- Instead of a single perceptron, use a whole network of them!
- One of the early attempts to mimic the brain
- The network is usually arranged in layers. The data flows through layers, one layer at a time.
- As a whole, the network acts like a function (a composite function of weights and neuron functions) taking a vector of input data and outputting a scalar (or vector). Used for regression and classification.

MLPs (2)



- The network function is computed layer by layer.
- The first layer of the network (almost always drawn on the left) contains only the input nodes.
- The last layer of the network (the right most layer) is called the output layer.
- A network with only one layer (acting as input and output) is an identity function.
- Weights and biases are between layers.
- Layers between the input and output (for networks with three layers or more) are called **hidden layers**.
- The number of node in the first (input) layer is equal to the number of inputs in the problem domain.
- The number of output nodes in the last layer depends on how we have encoded the output.

MLPs (3)



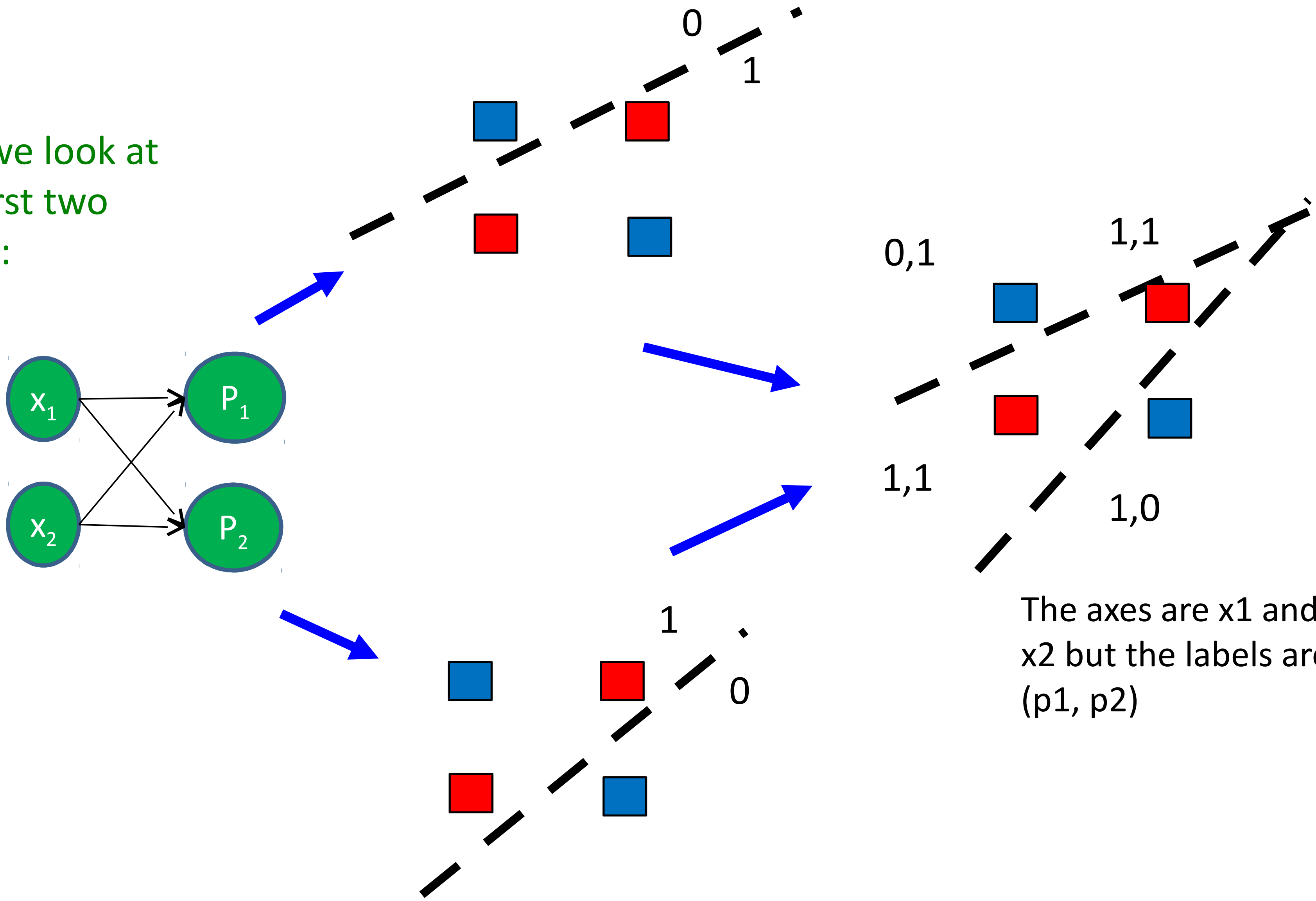
- We only consider certain types of feed-forward networks:
  - Adjacent layers are fully connected.
  - There is no backward connection.
  - There is not shortcut to the right layers (layers cannot be skipped)

- The number of weights between layer *i* and *i+1* is the number_nodes(*i*) * number_nodes(*i+1*). The number of biases is number_nodes(*i+1*).

- The number of hidden layers and the number of nodes in the hidden layers depends on the complexity of the problem.

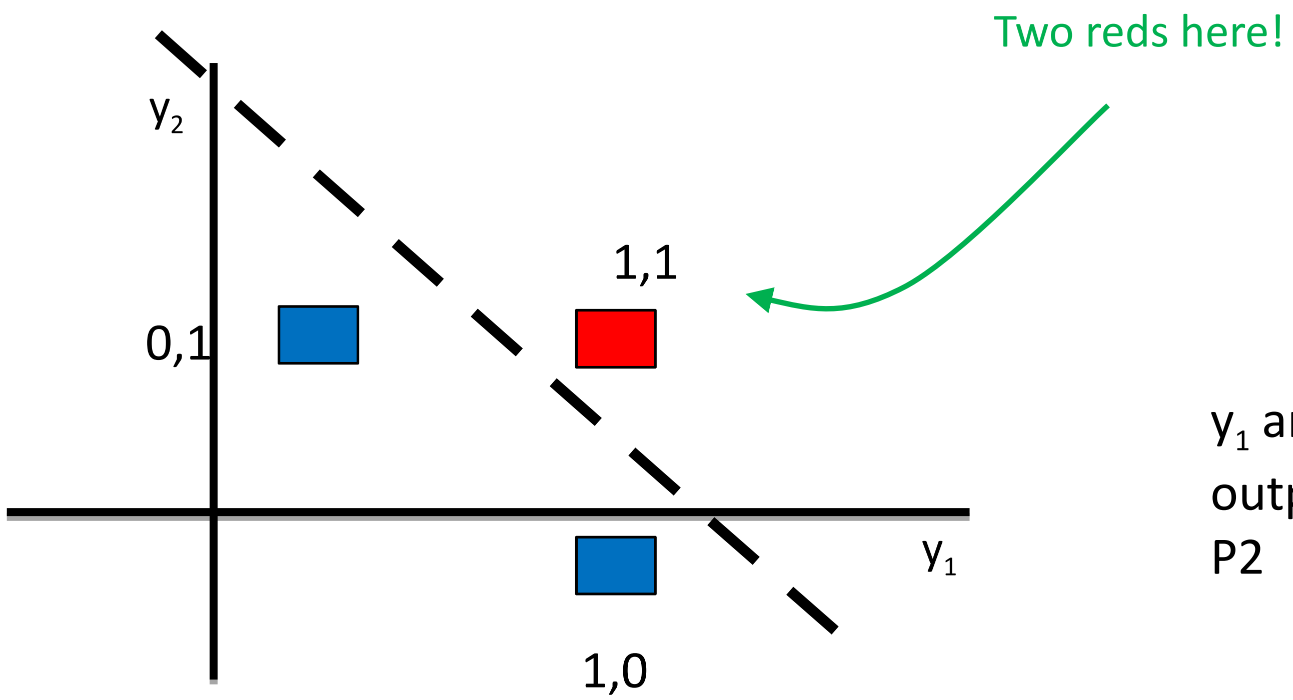
Example: using a 3-layer MLP for the "XOR" problem

First we look at the first two layers:

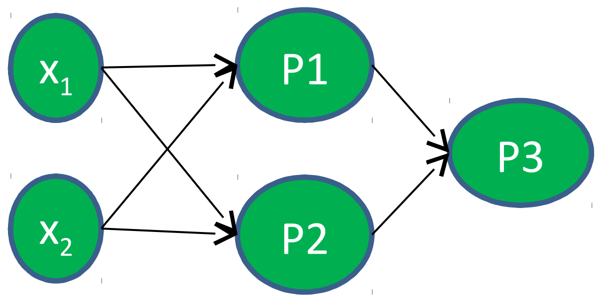


The axes are x1 and x2 but the labels are (p1, p2)

Example cont'd

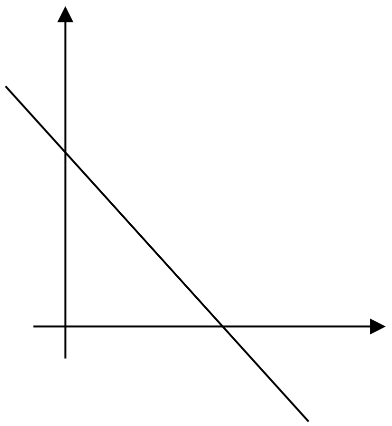


y_1 and y_2 are the outputs of P1 and P2

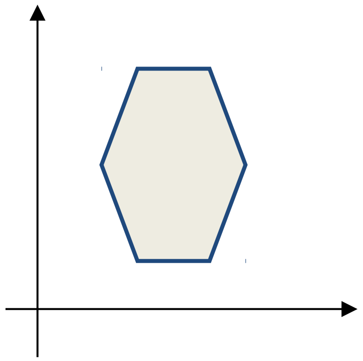


Now in the last layer (from perceptron 3 point of view) the examples are linearly separable!

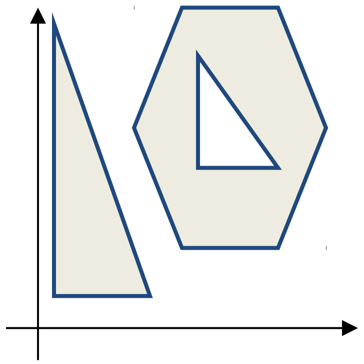
Increasing flexibility with more layers



2 layers

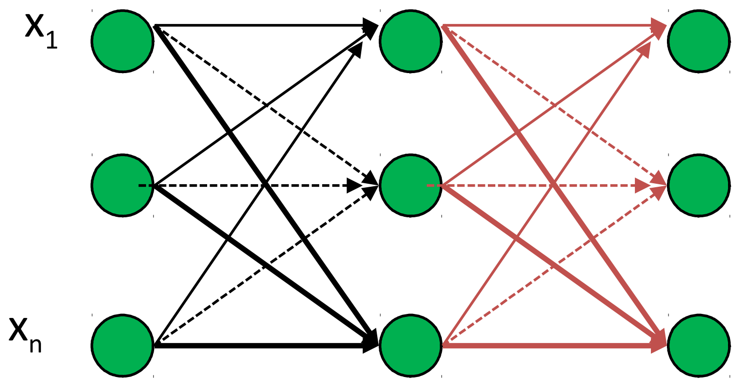


3 layers



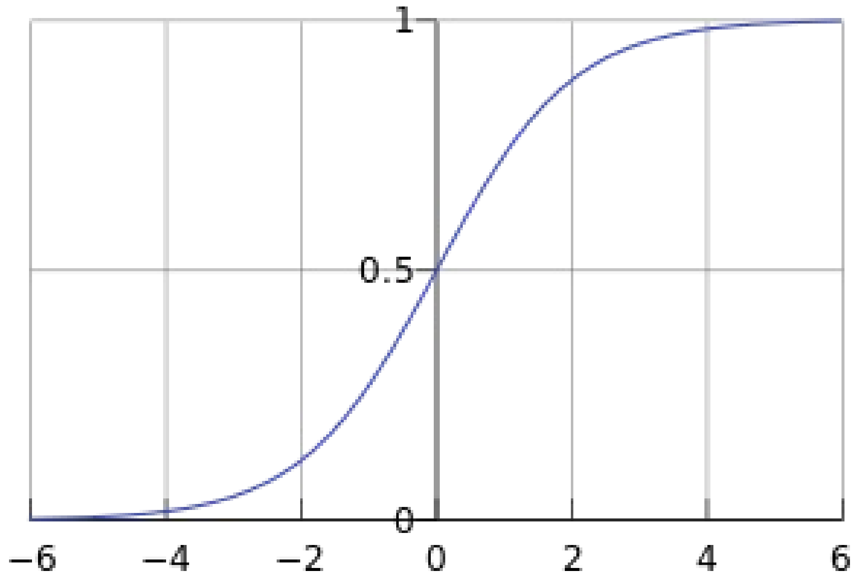
4 layers

How about multi-class classification?



Question: What is a potential problem with using step functions?

Using a Sigmoid Function



$$g(a) = \frac{1}{1 + e^{-a}}$$

- It is differentiable at all points!
- It handles uncertainty (e.g. an input example could be positive with different degrees of certainty from 0.5 to 1).

Learning in neural networks: The Error function

Mean squared error is typically used to measure error:

$$E = \sum_{i=1}^{n} (t_i - y_i)^2$$

Where t_i is the desired output (according to the training) data, y_i is the output of the network, and *n* is the number of patterns (examples) in the training set.

Learning: search for optimal weights

- The aim is to minimise an error function over all training patterns by adapting weights.
- The weight can be updated incrementally:

$$w_{jk}^{t+1} = w_{jk}^{t} - \eta \frac{\partial E}{\partial w_{jk}}$$

Illustration of Gradient Descent

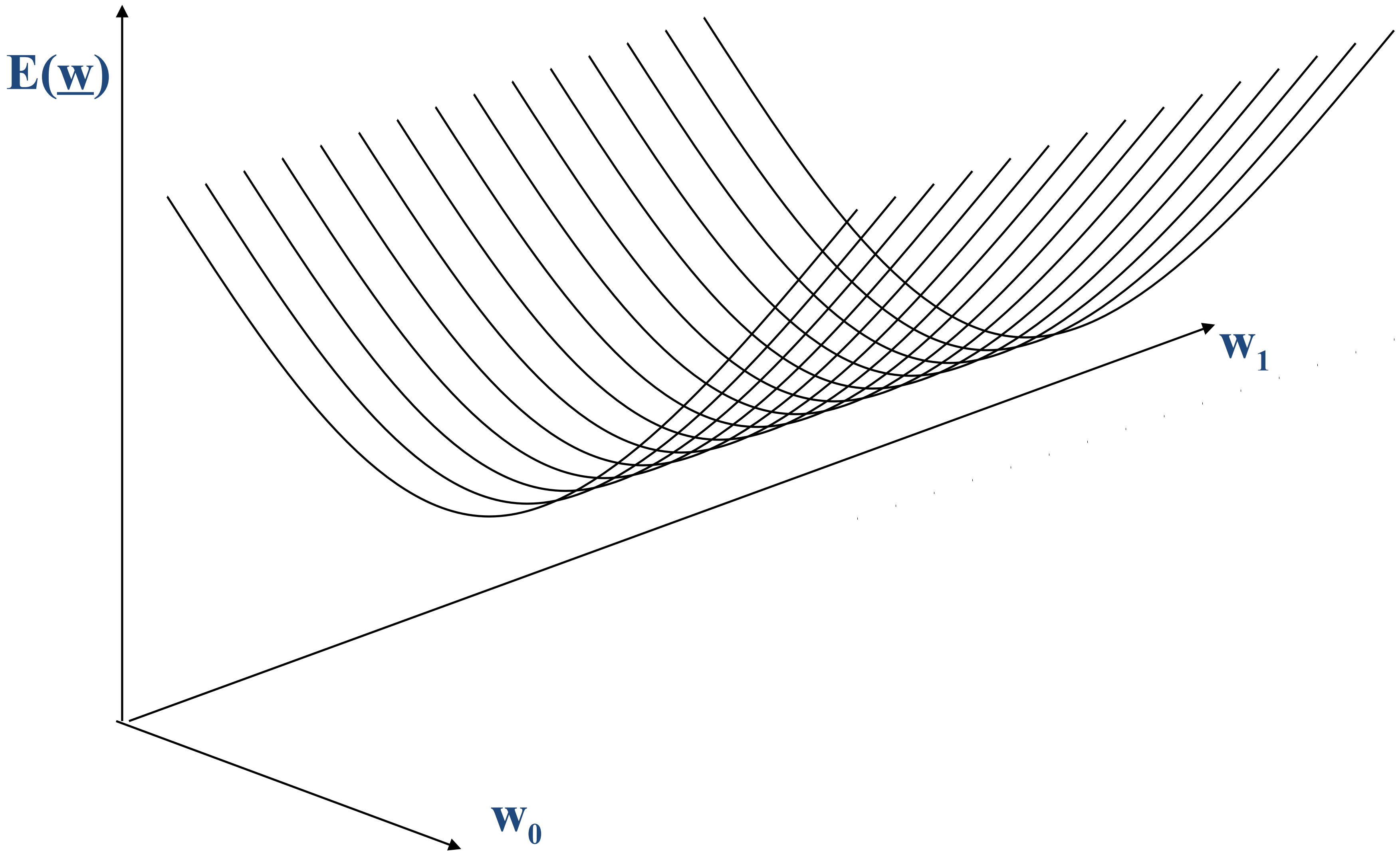


Illustration of Gradient Descent

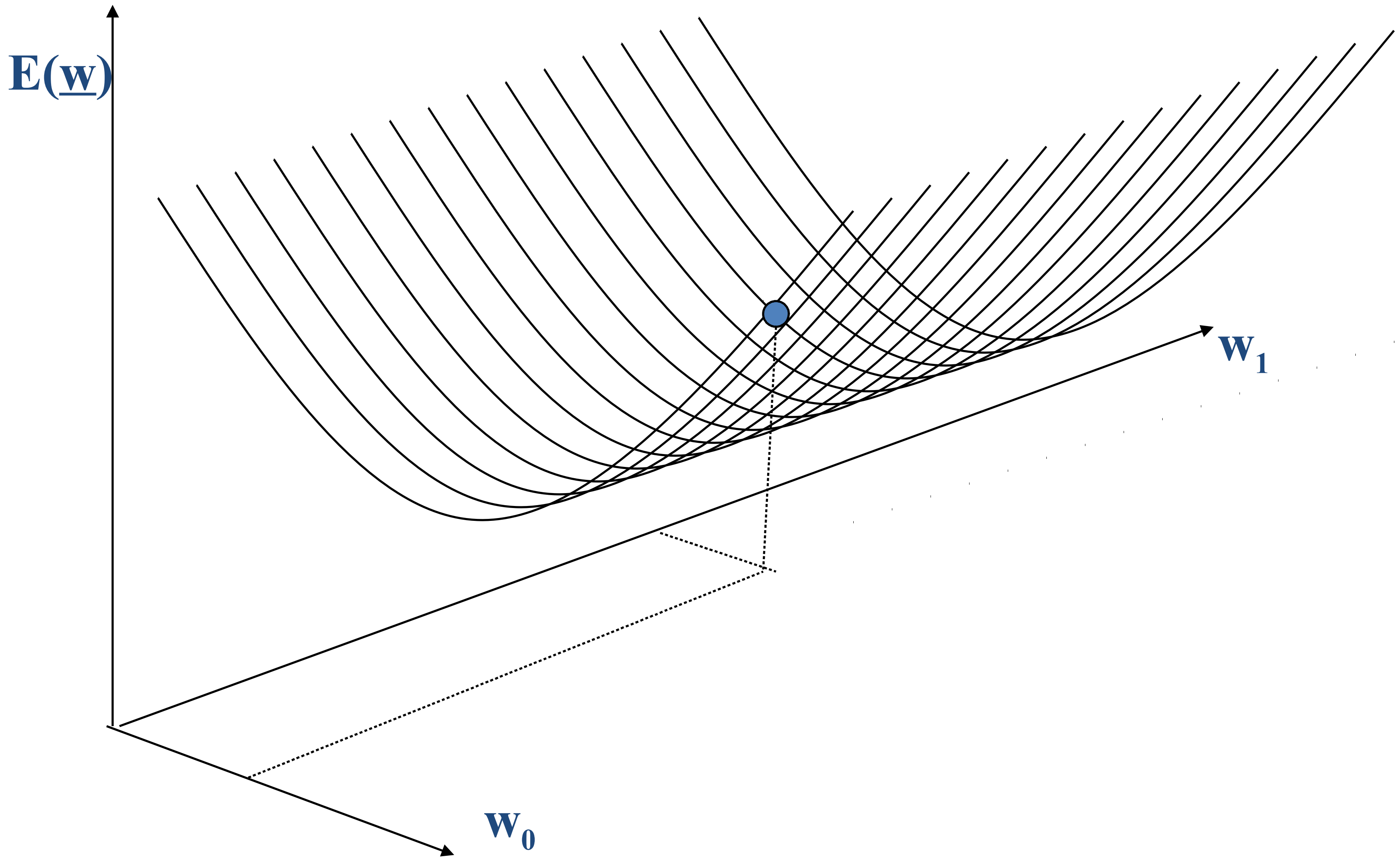
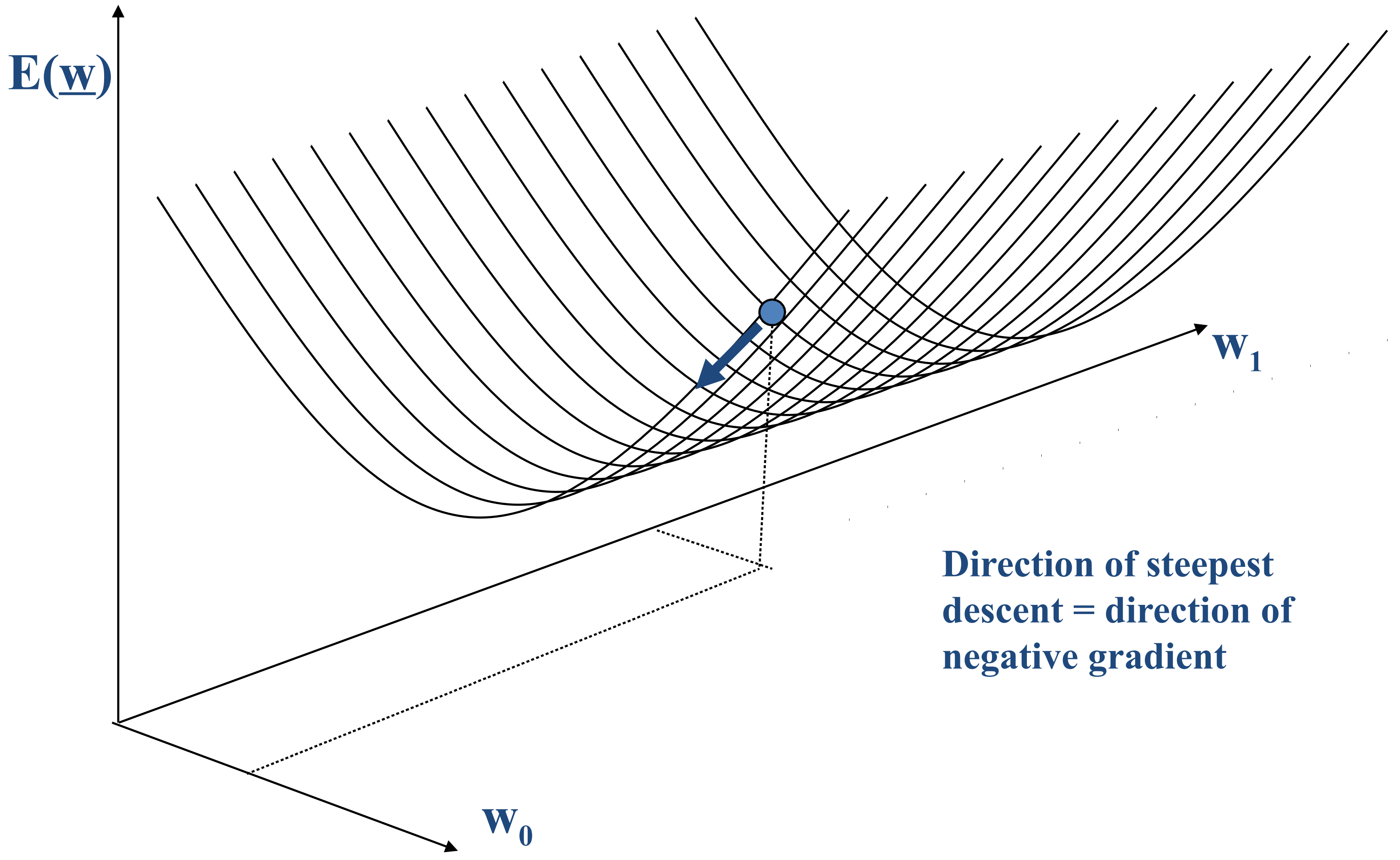
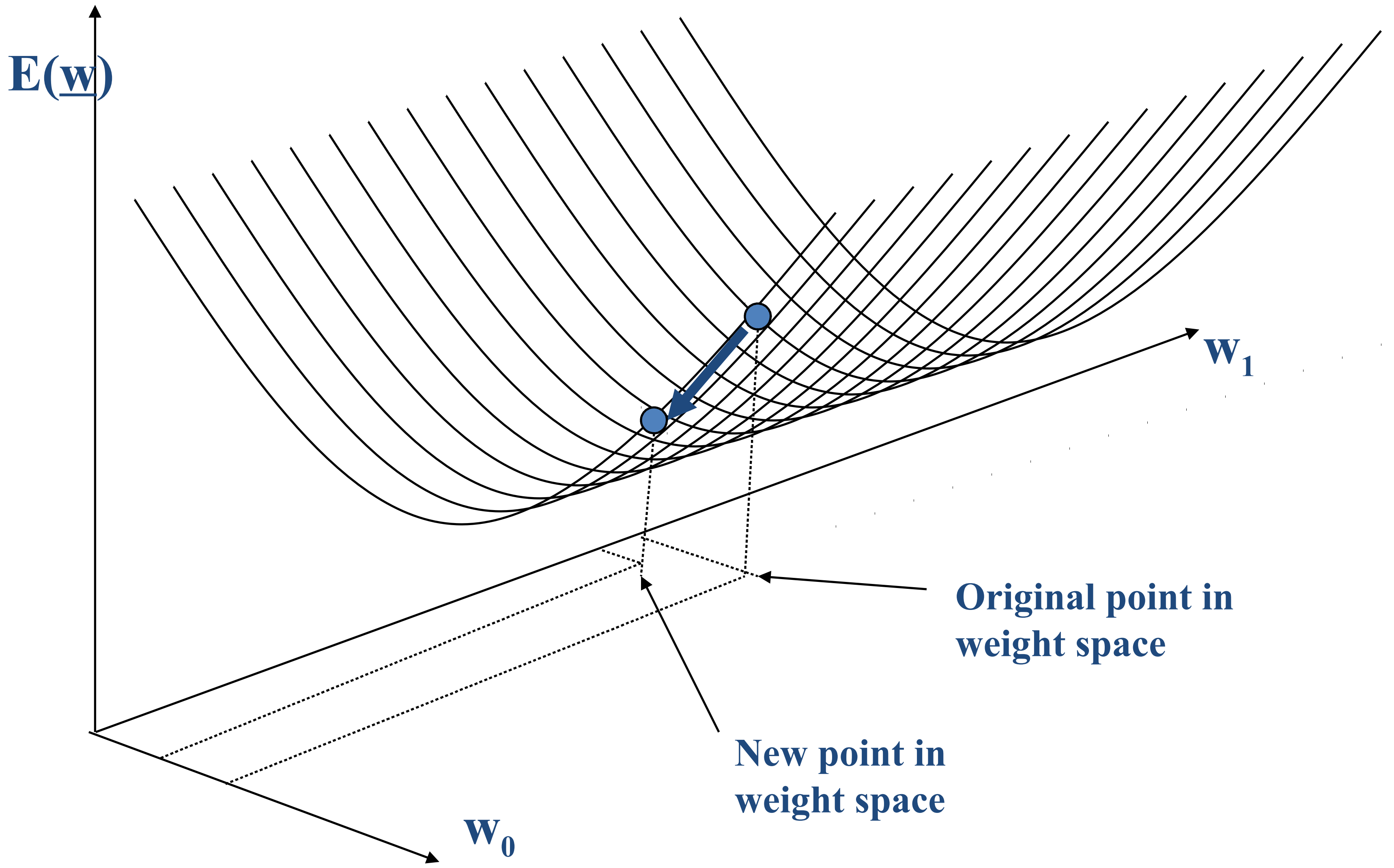


Illustration of Gradient Descent



Direction of steepest descent = direction of negative gradient

Illustration of Gradient Descent

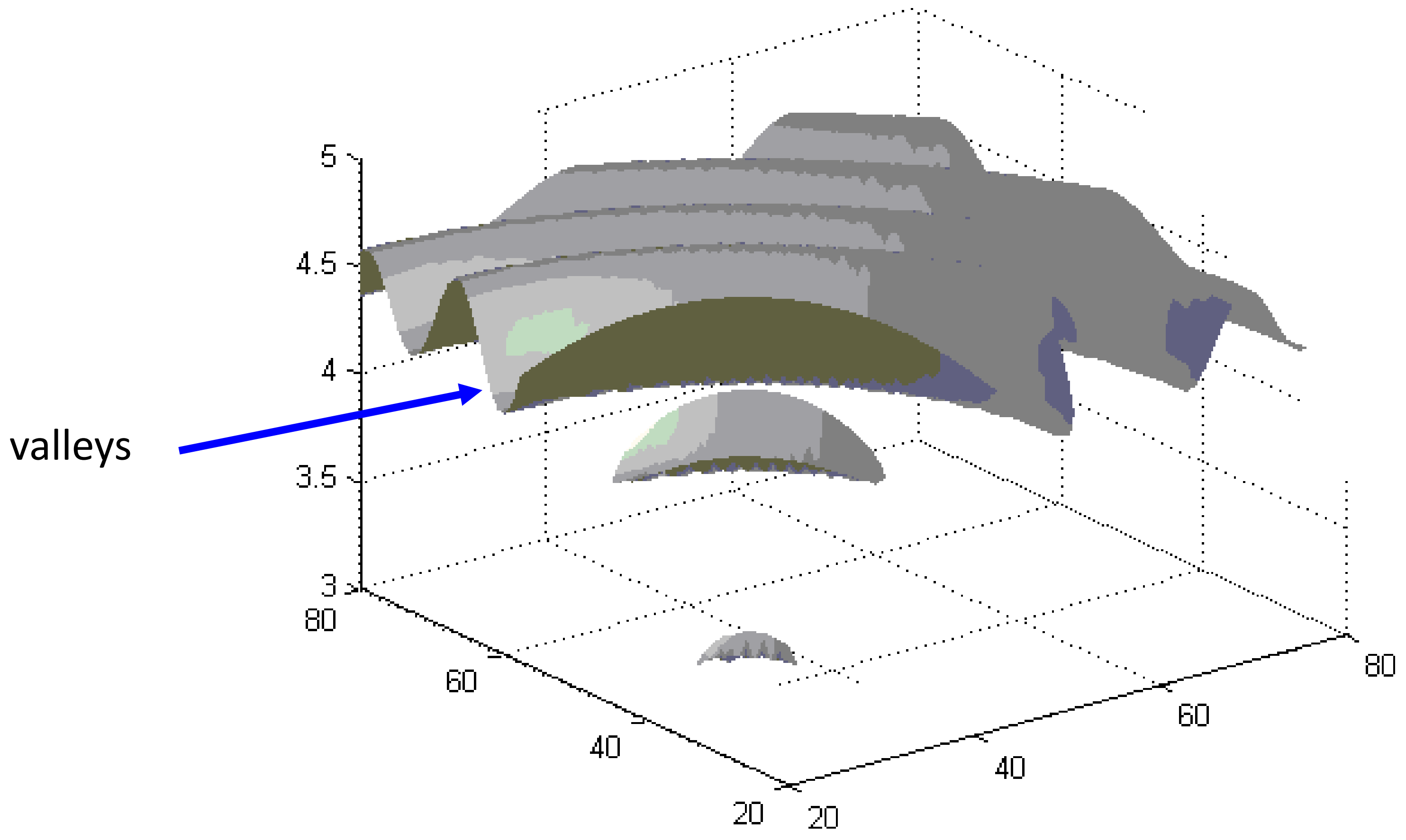


General Gradient Descent Algorithm

- Define an objective function E($\underline{w}$)

- Algorithm:
  - pick an initial set of weights $\underline{w}$, e.g. randomly
  - evaluate $\nabla \mathbf{E}(\underline{\mathbf{w}})$ at $\underline{w}$
    - note: this can be done numerically or in closed form
  - update all the weights
    - $\underline{\mathbf{w}}_{\mathbf{new}} = \underline{\mathbf{w}}_{\mathbf{old}} - \eta \nabla \mathbf{E}(\underline{\mathbf{w}})$
  - check if $\nabla \mathbf{E}(\underline{\mathbf{w}})$ is approximately 0
    - if so, we have converged to a "flat minimum"
    - if not, we move again in weight space

Multi-layer networks often have complex error surfaces (e.g. plateaus, long valleys etc.) with no single minimum.



Selecting initial weight values

- Choice of initial weight values is important as this decides starting position in weight space. That is, how far away from global minimum

- Aim is to select weight values which produce midrange function signals (when the sigmoid function is saturated, its derivative is very close to zero, so weights don't get updated)

- One idea is to select weight values randomly form uniform probability distribution

Question: What is the effect of learning rate?

Momentum

Method of reducing problems of instability while increasing the rate of convergence.

Adding a term to weight update equation to hold history of previous weight updates:

$$\Delta w(t) = -\eta \frac{\partial E}{\partial w} + \alpha \Delta w(t-1)$$

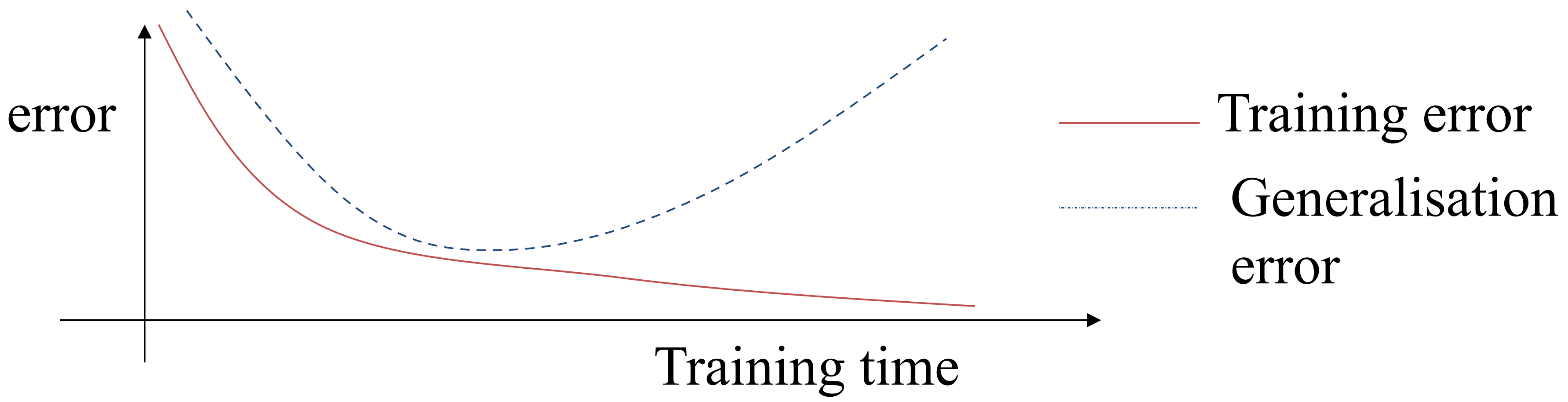
Momentum cont'd

α is momentum constant and determines how much recent history influences the next update.

Effect of momentum term:

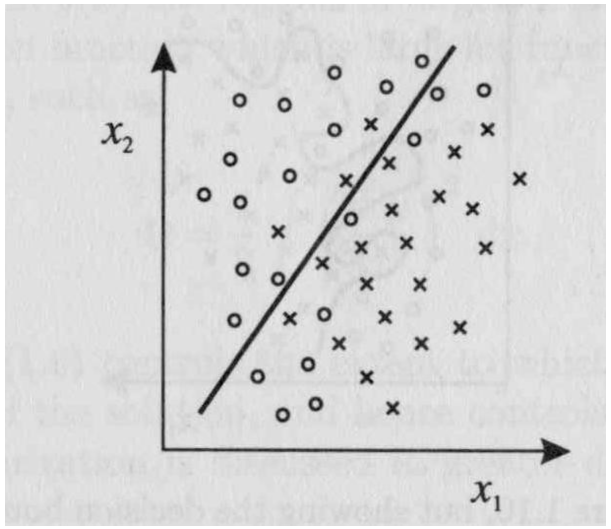
If weight changes tend to have the same sign, the momentum builds up and larger steps are taken.

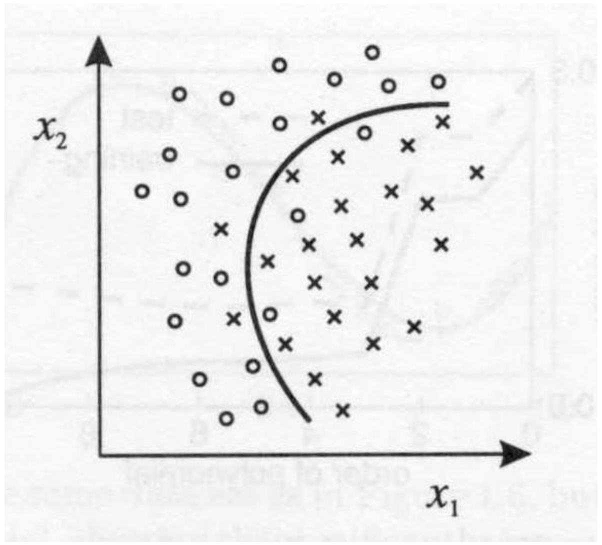
If weight changes tend to have opposing signs, the momentum term decreases and gradient descent slows down to reduce oscillations (stabilises).

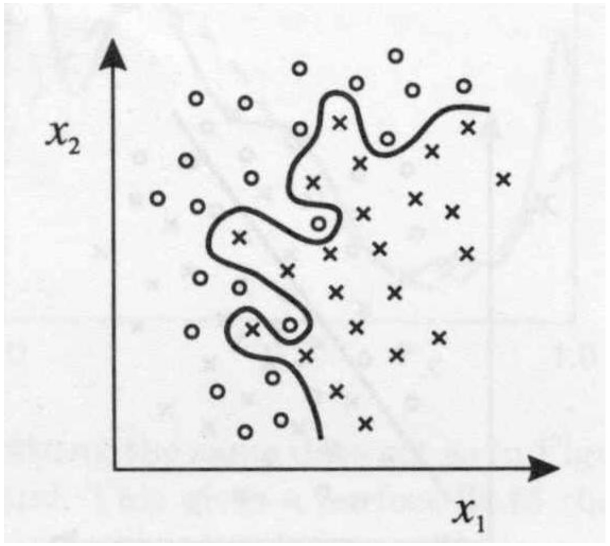


Typically, though error on training set will decrease as training continues generalisation error (error on unseen data) hitts a minimum then increases (cf model complexity etc)

Therefore want more complex stopping criterion







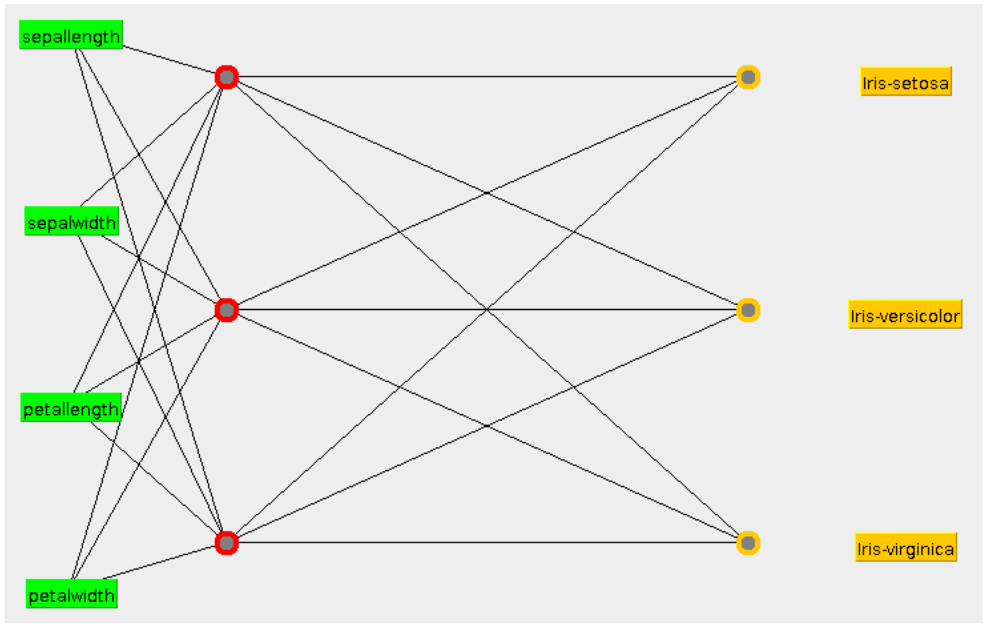
Overfitting can occur in classification problems as well: A model with too much flexibility does not generalise well resulting in a non-smooth decision boundary.

Somewhat like giving a system enough capacity (nodes/layers) to ‘remember’ all training points: no need to generalise. Less capacity => it must generalise to be able to model training data

Network architecture for a typical classification task

- The numbers of inputs and outputs is determined by the problem.
- Usually one hidden layer is enough for most classification tasks. Therefore, there will be a column of input nodes (equal to the number of attributes in the problem), a column of hidden neurons, and a column of output neurons. This means that there are two sets (columns) of weights: one for the hidden layer and the other for the output layer.

4 attributes



3 Classes

A Rough Guideline for network size

- Best is to have as few hidden layers/nodes as possible
  - Forces better generalization
  - Fewer weights to be found
- Number of nodes in the hidden layer:
  - Make the best guess you can (e.g. a number between the number of input and output neurons)
  - If training is unsuccessful try more hidden nodes
  - If training is successful try fewer hidden nodes
  - Inspect weights after training. Nodes which have small weights can probably be eliminated