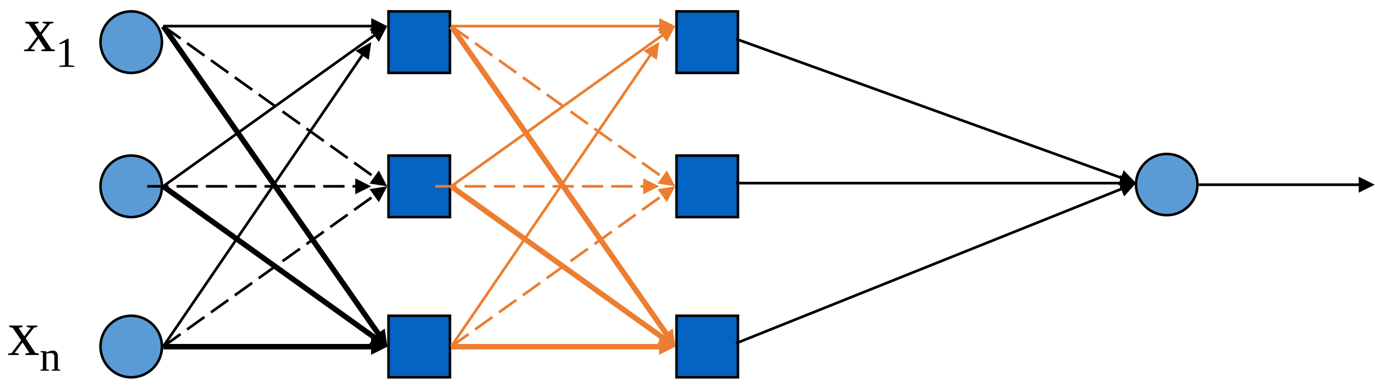
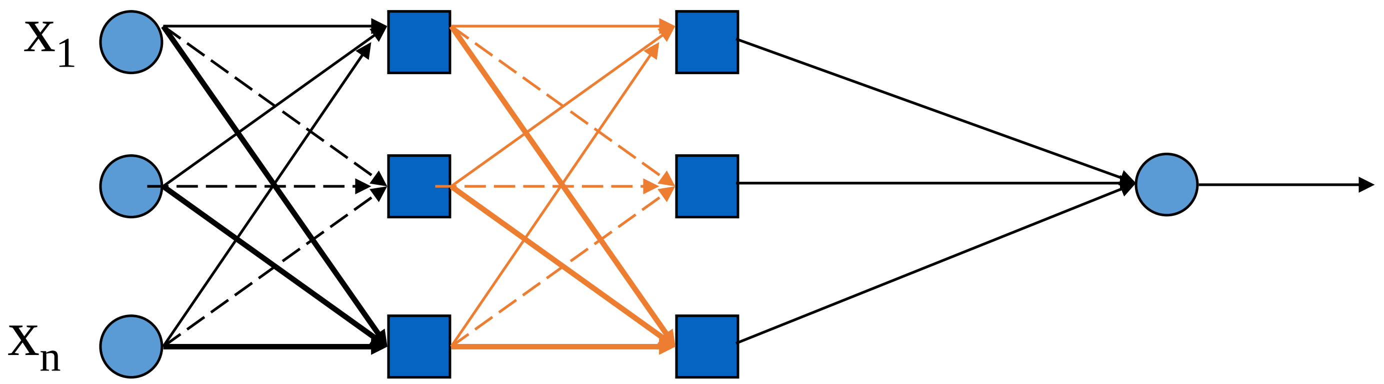
Multi-Layer Perceptron (MLP)



Today we will introduce the MLP and the backpropagation algorithm which is used to train it

MLP used to describe any general feedforward (no recurrent connections) network

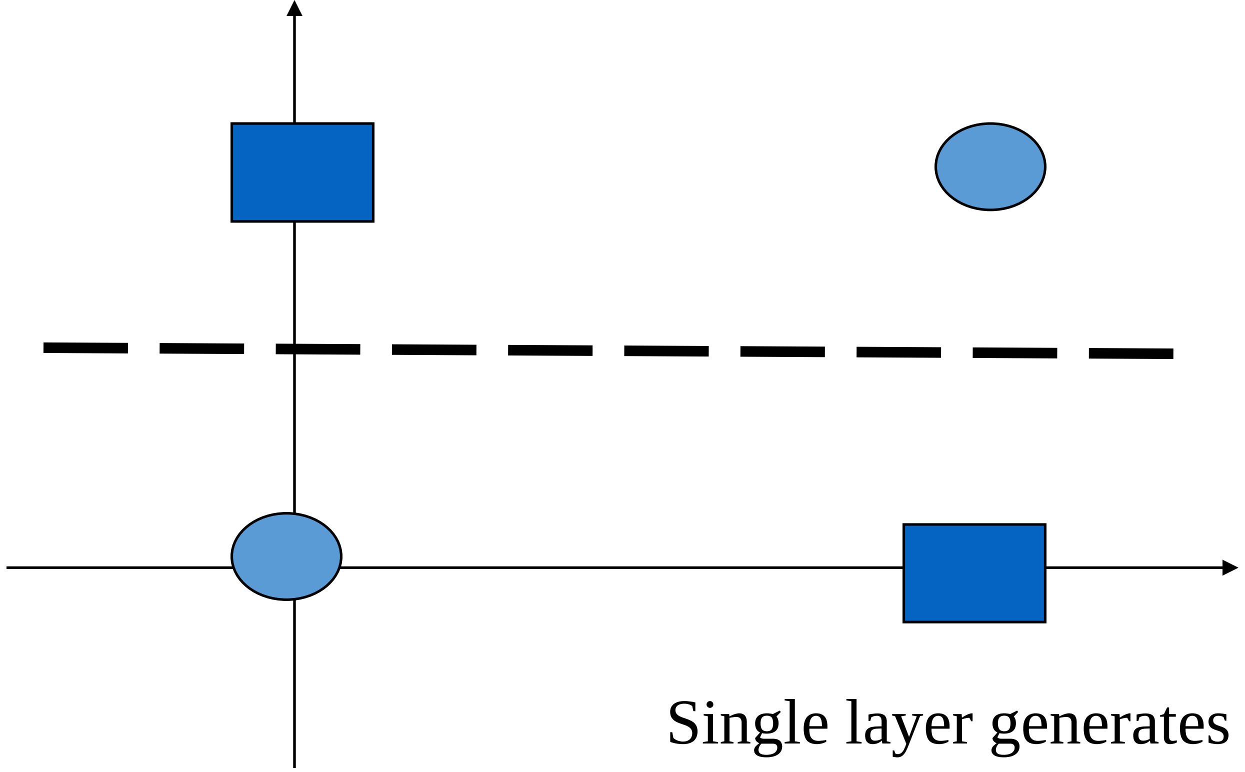
However, we will concentrate on nets with units arranged in layers



1st question:

what do the extra layers gain you? Start with looking at what a single layer can't do

XOR problem



Single layer generates a linear decision boundary

XOR (exclusive OR) problem

0+0=0

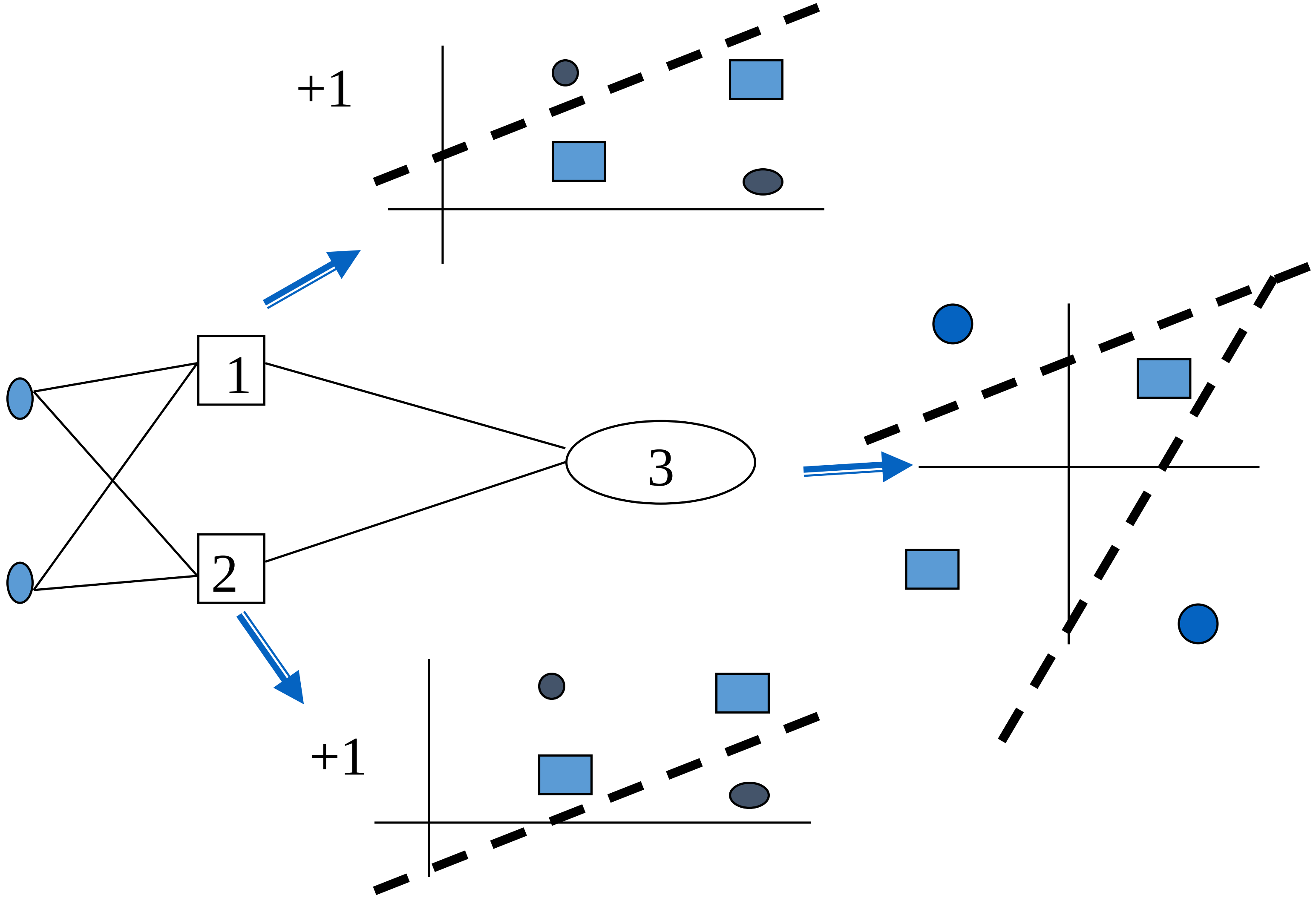
1+1=2=0 mod 2

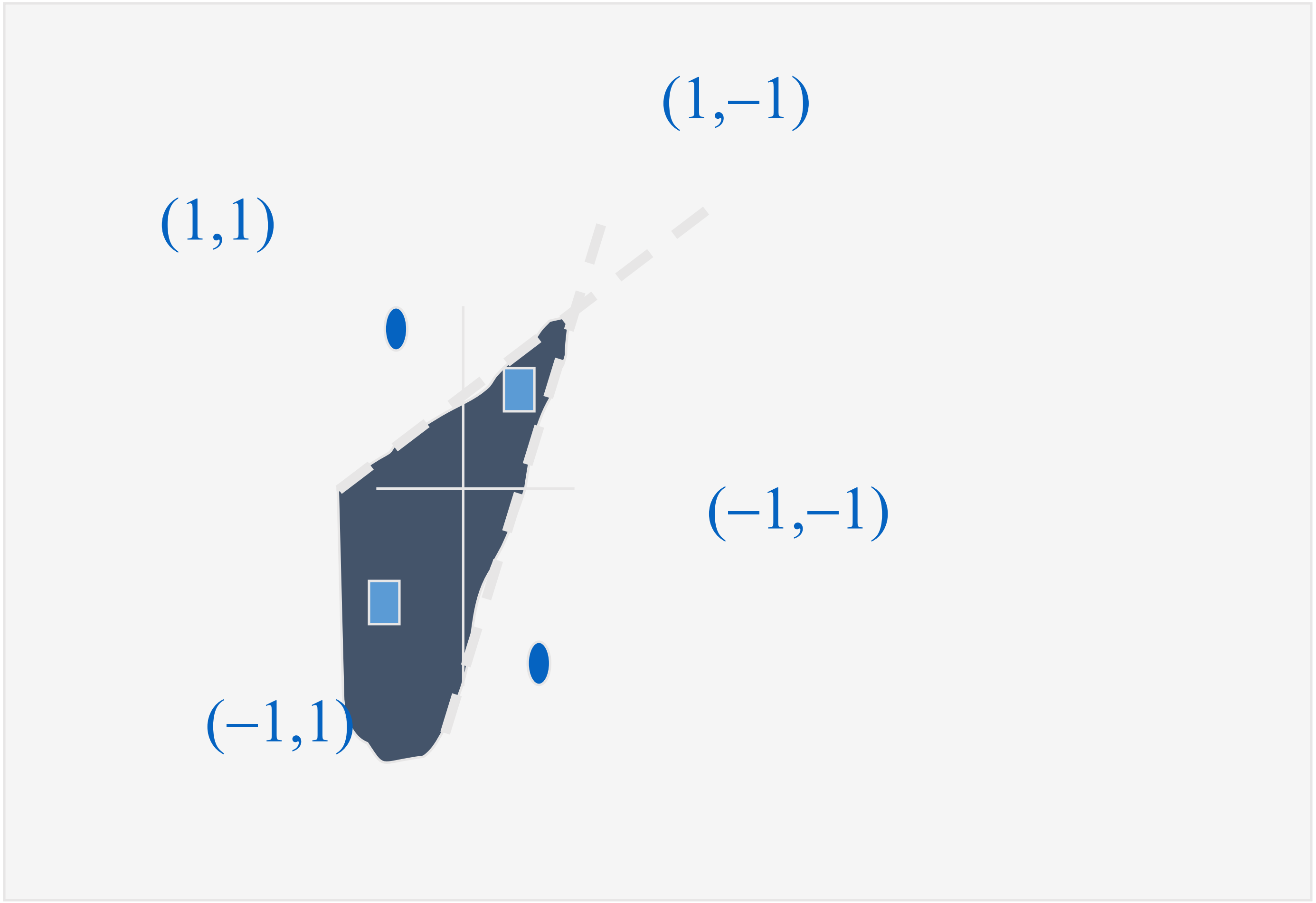
1+0=1

0+1=1

Perceptron does not work here

Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of units

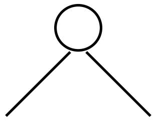
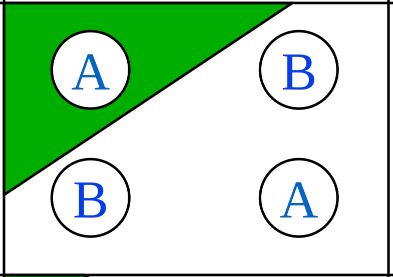
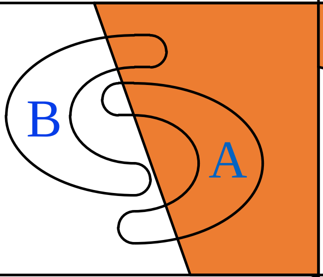
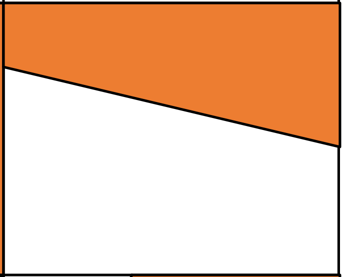
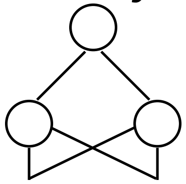
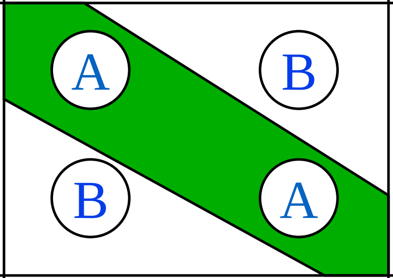
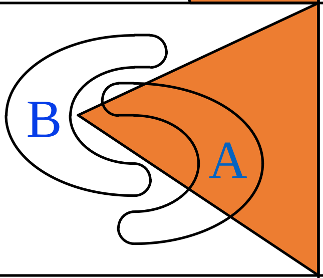
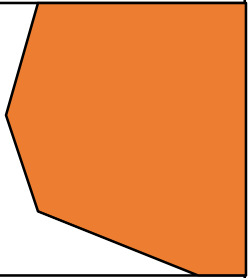
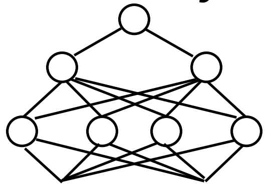
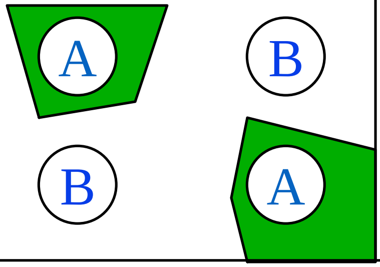
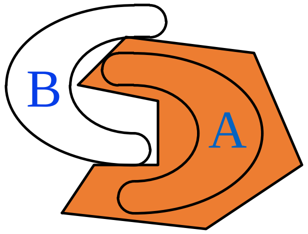
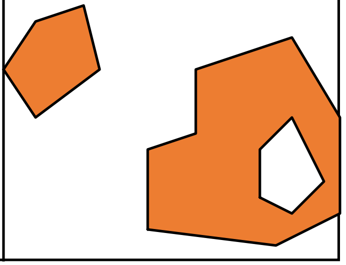




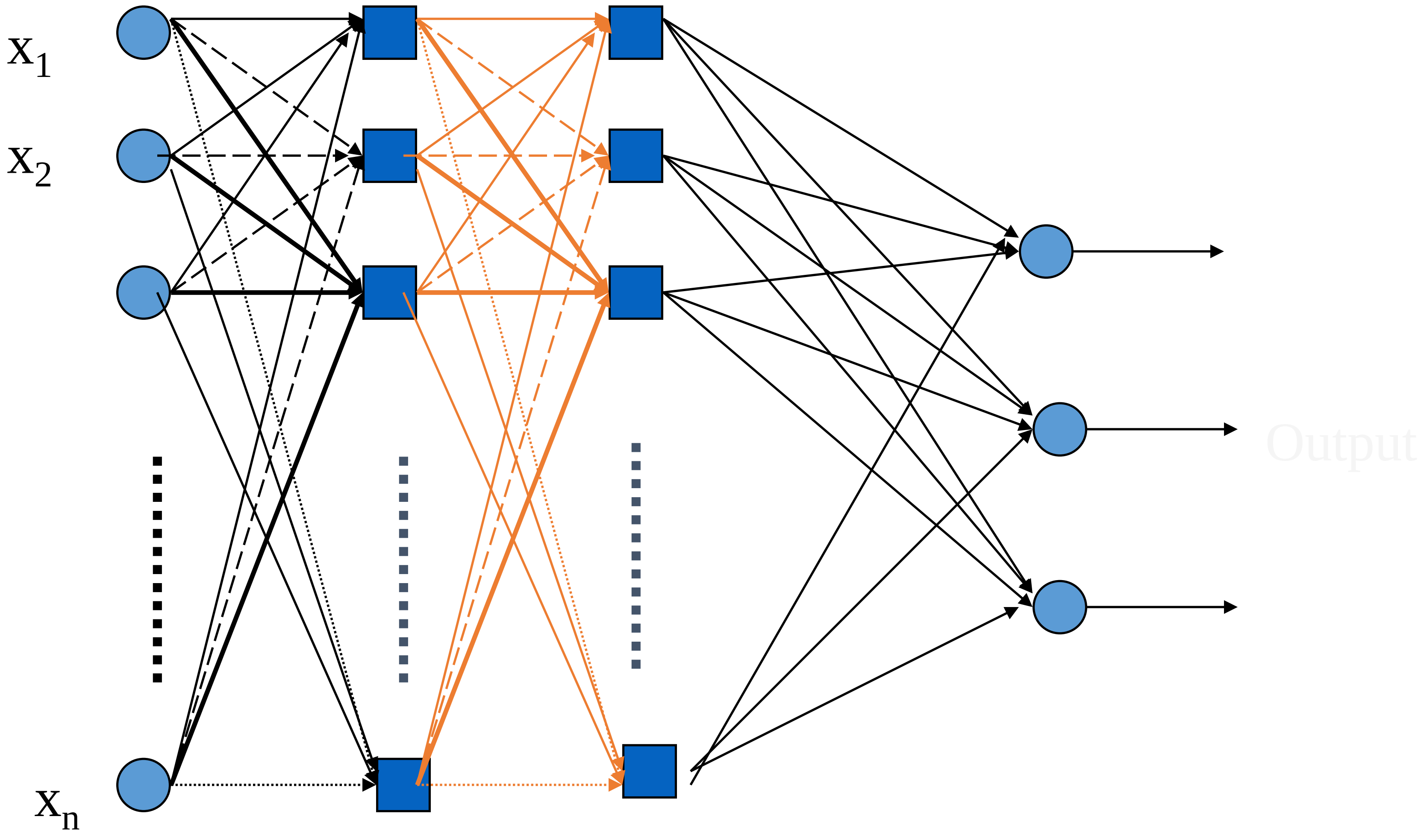
This is a linearly separable problem!

Since for 4 points { (-1,1), (-1,-1), (1,1),(1,-1) } it is always linearly separable if we want to have three points in a class

Non linearly separable problems

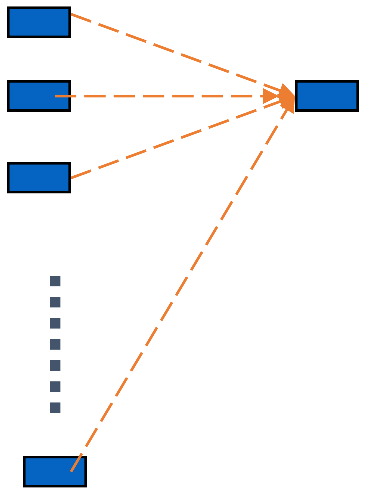
Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shapes*
Single-Layer 	*Half Plane Bounded By Hyperplane*			
Two-Layer 	*Convex Open Or Closed Regions*			
Three-Layer 	Abitrary (Complexity Limited by No. of Nodes)			

Three-layer networks



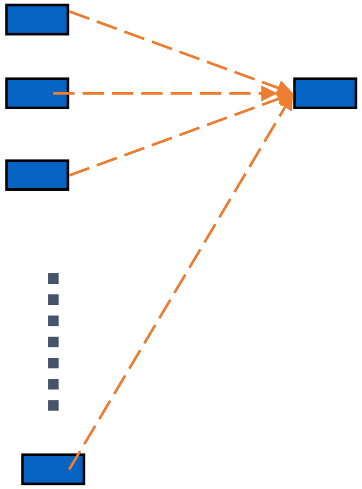
Hidden layers

- No connections within a layer



Each unit is a perceptron

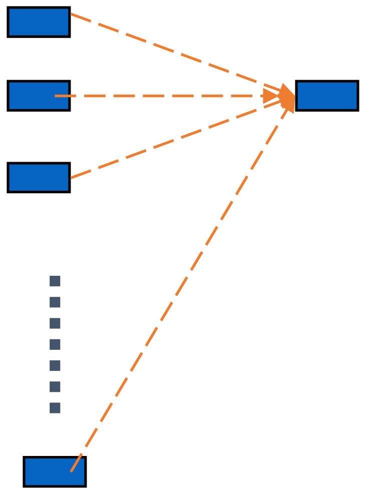
- No connections within a layer
- No direct connections between input and output layers
-



Each unit is a perceptron

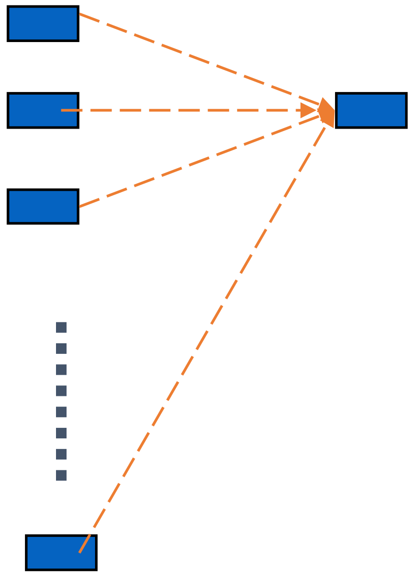
[redacted]

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers



Each unit is a perceptron

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units

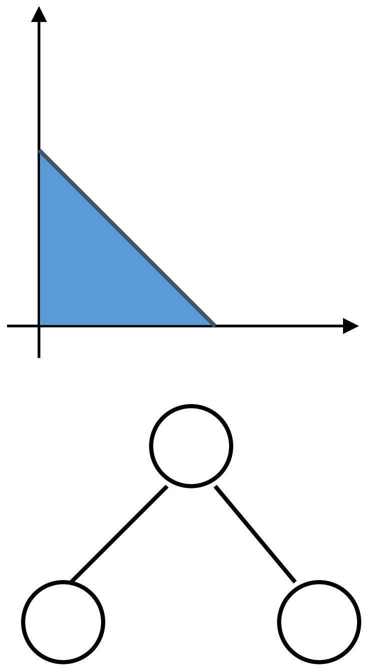


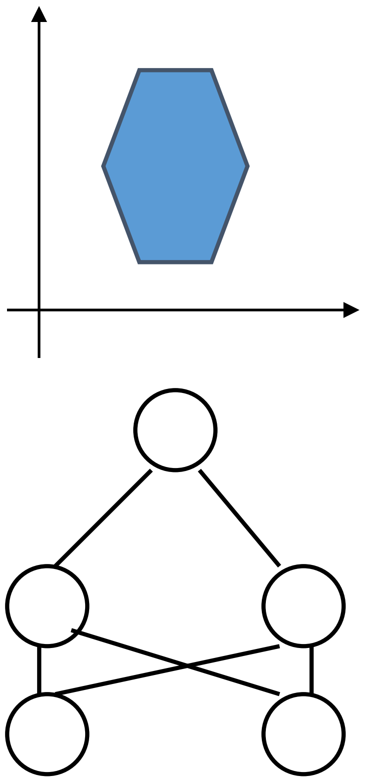
Each unit is a perceptron

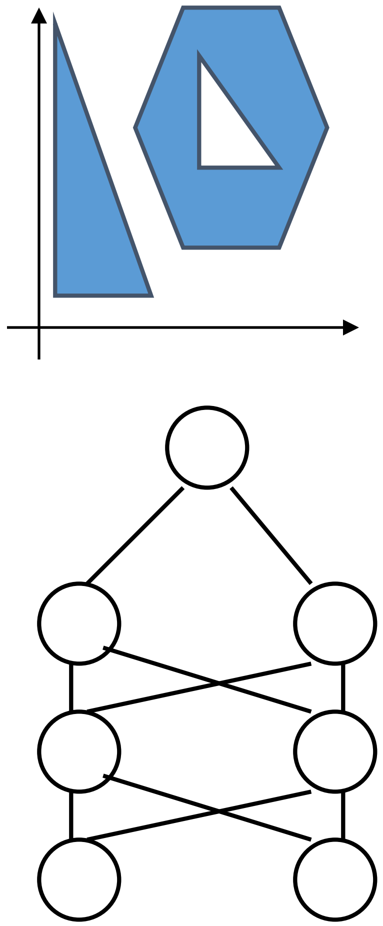
[redacted]

Often include bias as an extra weight

What do each of the layers do?


1st layer draws linear boundaries


2nd layer combines the boundaries


3rd layer can generate arbitrarily complex boundaries

Continued…

- Can also view 2nd layer as using local knowledge while 3rd layer does global
- With sigmoid activation functions can show that a 3 layer net can approximate any function to arbitrary accuracy: property of Universal Approximation
- Proof by thinking of superposition of sigmoid
- Not practically useful as need arbitrarily large number of units but more of an existence proof
- For a 2 layer net, same is true for a 2 layer net providing function is continuous and from one finite dimensional space to another

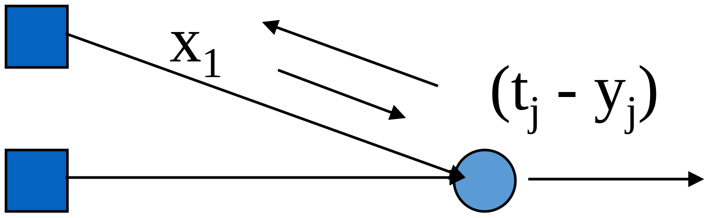
BP Algorithm

gradient descent method

+

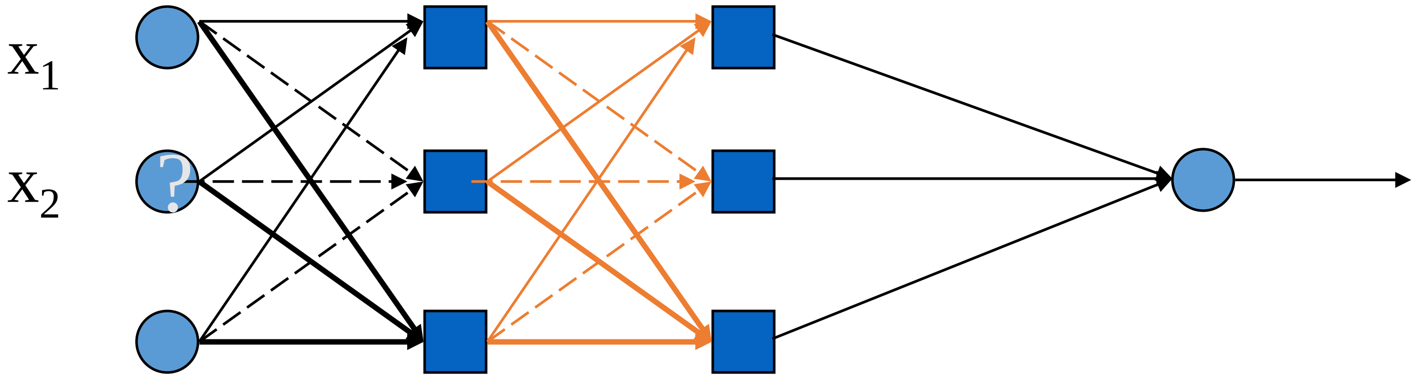
multilayer networks

In the perceptron/single layer nets, we used gradient descent on the error function to find the correct weights:



$$\Delta w_{ji} = (t_j - y_j)\, x_i$$

We see that errors/updates are local to the node ie the change in the weight from node i to output j (w_{ji}) is controlled by the input that travels along the connection and the error signal from output j



•*But with more layers how are the weights for the first 2 layers found when the error is computed for layer 3 only?*

•*There is no direct error signal for the first layers!!!!!*

Credit assignment problem

• Problem of assigning 'credit' or 'blame' to individual elements involved in forming overall response of a learning system (hidden units)

• In neural networks, problem relates to deciding which weights should be altered, by how much and in which direction.

Analogous to deciding how much a weight in the early layer contributes to the output and thus the error

We therefore want to find out how weight w_{ij} affects the error ie we want:

Backpropagation learning algorithm ‘BP’

Solution to credit assignment problem in MLP

Rumelhart, Hinton and Williams (1986)

Forward pass phase:

computes ‘functional signal’, feed-forward propagation of input pattern signals through network

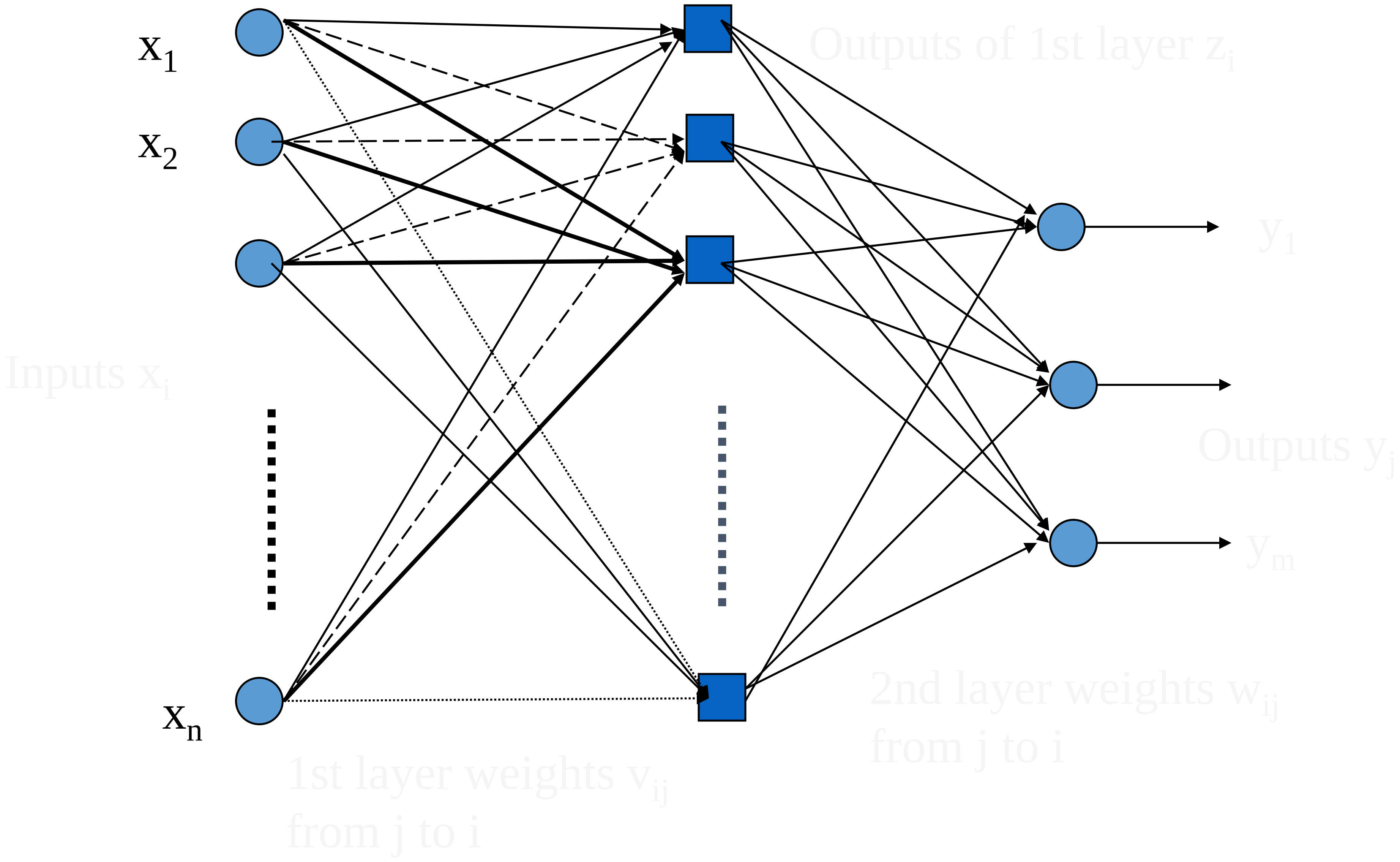
Back Propagation learning algorithm ‘BP’

BP has two phases:

Forward pass phase: computes ‘functional signal’, feed-forward propagation of input pattern signals through network

Backward pass phase: computes ‘error signal’, *propagates* the error *backwards* through network starting at output units (where the error is the difference between actual and desired output values)

Two-layer networks


x_1
x_2
x_n
Inputs x_i
Outputs of 1st layer z_i
y_1
Outputs y_j
y_m
2nd layer weights w_{ij} from j to i
1st layer weights v_{ij} from j to i

We will concentrate on two-layer, but could easily generalize to more layers

$$z_i(t) = g(\Sigma_j \; v_{ij}(t) x_j(t)) \quad \text{at time t}$$
$$= g(u_i(t))$$

$$y_i(t) = g(\Sigma_j \; w_{ij}(t) z_j(t)) \quad \text{at time t}$$
$$= g(a_i(t))$$

a/u known as activation, g the activation function

biases set as extra weights

Forward pass

Weights are fixed during forward and backward pass at time *t*

1. Compute values for hidden units

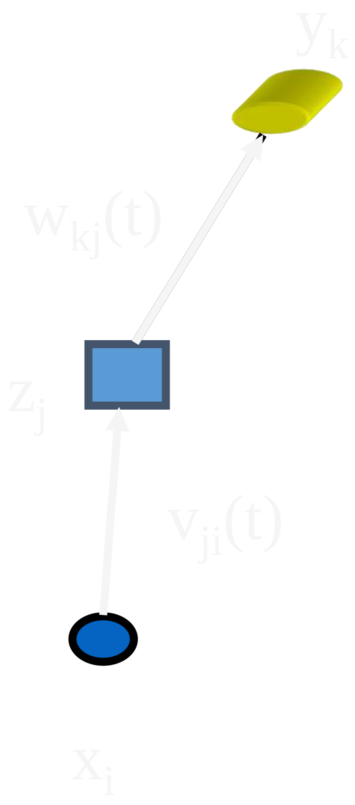
$$u_j(t) = \sum_i v_{ji}(t) x_i(t)$$

$$z_j = g(u_j(t))$$

2. compute values for output units

$$a_k(t) = \sum_j w_{kj}(t) z_j$$

$$y_k = g(a_k(t))$$



Backward Pass

Will use a sum of squares error measure. For each training pattern we have:

[redacted]

where d_k is the target value for dimension k. We want to know how to modify weights in order to decrease E. **Use gradient descent** ie

$$w_{ij}(t+1) - w_{ij}(t) \propto -\frac{\partial E(t)}{\partial w_{ij}(t)}$$

both for hidden units and output units

The partial derivative can be rewritten as product of two terms using chain rule for partial differentiation

$$\frac{\partial E(t)}{\partial a_i(t)} \cdot \frac{\partial a_i(t)}{\partial w_{ij}(t)}$$

both for hidden units and output units

Term A

How error for pattern changes as function of change in network input to unit *j*

Term B

How net input to unit j changes as a function of change in weight *w*

Term B first:

$$\frac{\partial u_i(t)}{\partial v_{ij}(t)} = x_j(t) \qquad \frac{\partial a_i(t)}{\partial w_{ij}(t)} = z_j(t)$$

Term A   Let

$$\delta_i(t) = -\frac{\partial E(t)}{\partial u_i(t)}, \Delta_i(t) = -\frac{\partial E(t)}{\partial a_i(t)}$$

(error terms). Can evaluate these by chain rule:

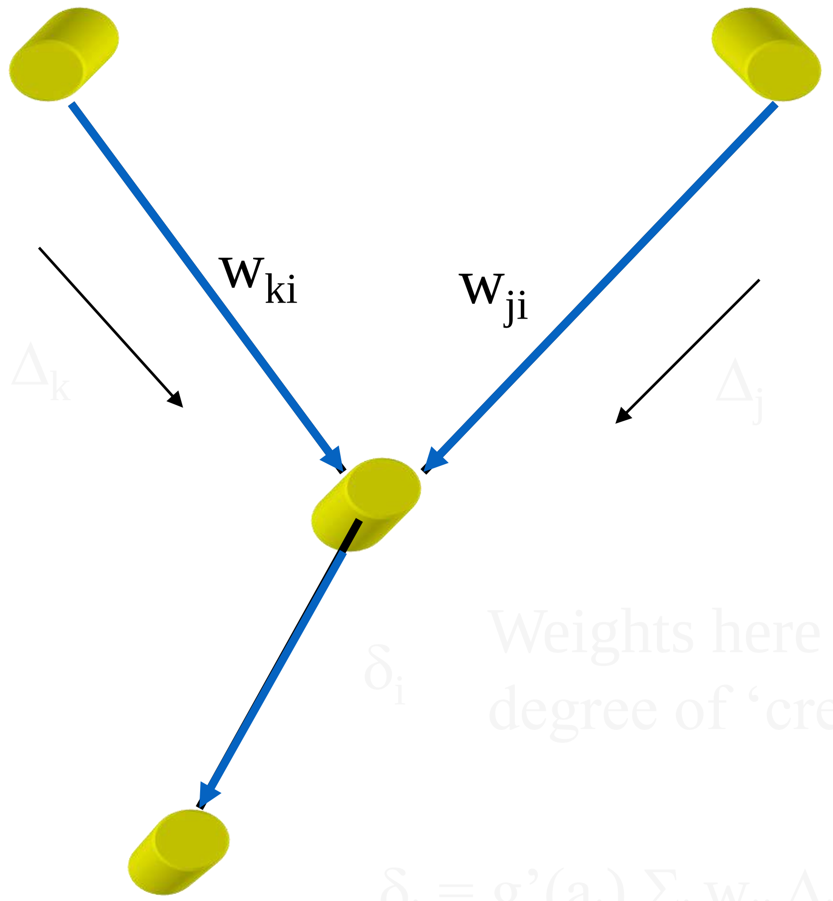
For output units we therefore have:

[redacted]

For hidden units must use the chain rule:

[redacted]

Backward Pass



Weights here can be viewed as providing degree of ‘credit’ or ‘blame’ to hidden units

$\delta_i = g'(a_i) \Sigma_j w_{ji} \Delta_j$

Combining A+B gives

[redacted]

So to achieve gradient descent in E should change weights by

$$v_{ij}(t+1)-v_{ij}(t) = \eta\, \delta_i (t)\, x_j (n)$$

$$w_{ij}(t+1)-w_{ij}(t) = \eta\, \Delta_i (t)\, z_j (t)$$

Where η is the learning rate parameter ($0 < \eta <=1$)

Summary

Weight updates are local

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$

$$w_{ij}(t+1) - w_{ij}(t) = \eta \Delta_i(t) z_j(t)$$

output unit

$$w_{ij}(t+1) - w_{ij}(t) = \eta \Delta_i(t) z_j(t)$$
$$= \eta (d_i(t) - y_i(t)) g'(a_i(t)) z_j(t)$$

hidden unit

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$
$$= \eta g'(u_i(t)) x_j(t) \sum_k \Delta_k(t) w_{ki}$$

Algorithm (sequential)

1. Apply an input vector and calculate all activations, a and u
2. Evaluate Δ_k for all output units via:

[redacted]

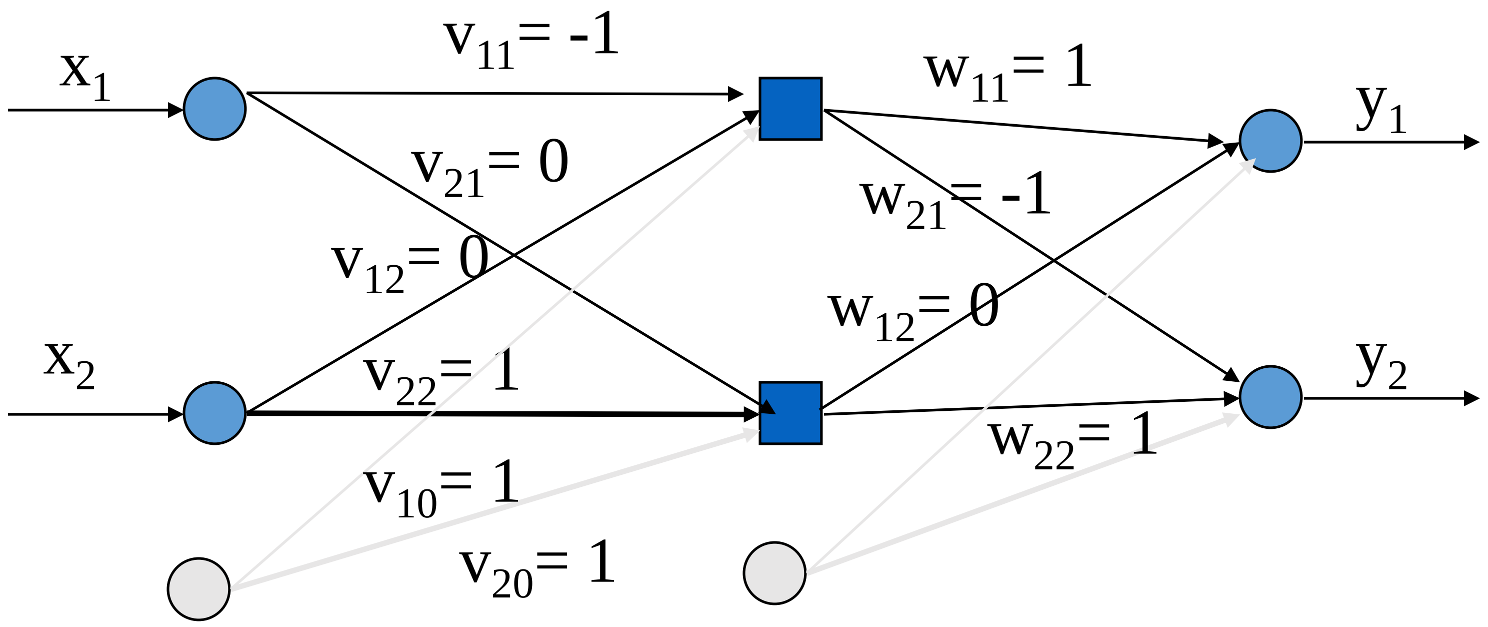
3. Backpropagate Δ_k(s) to get error terms δ for hidden layers using:

[redacted]

4. Evaluate changes using:

[redacted]

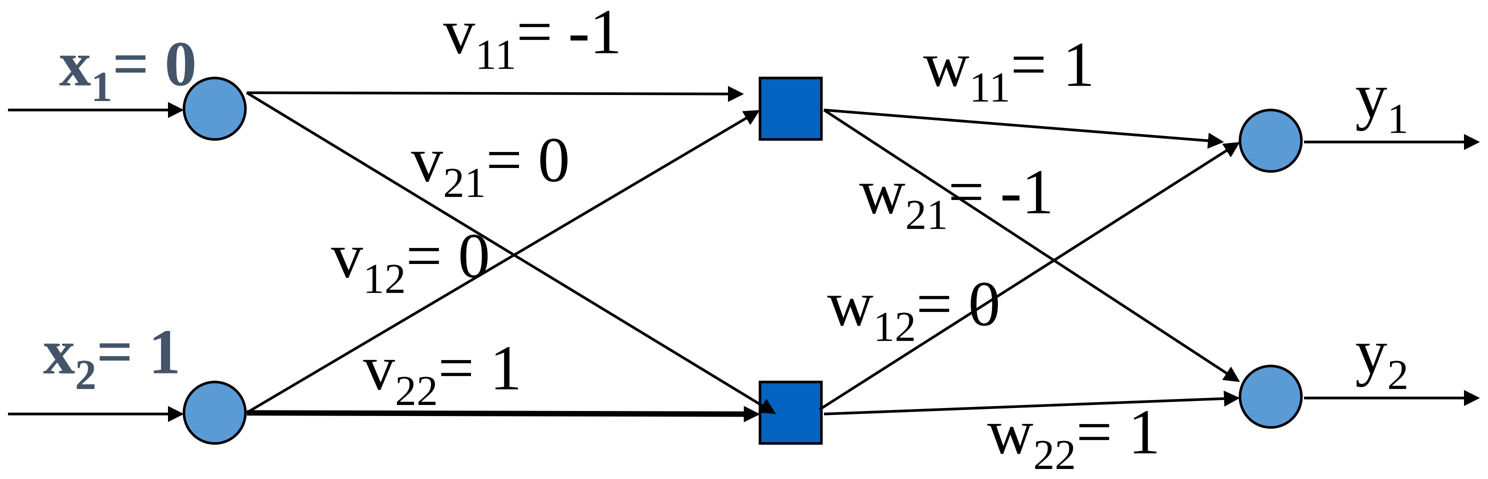
Once weight changes are computed for all units, weights are updated at the same time (bias included as weights here). An example:



Use identity activation function (ie g(a) = a)

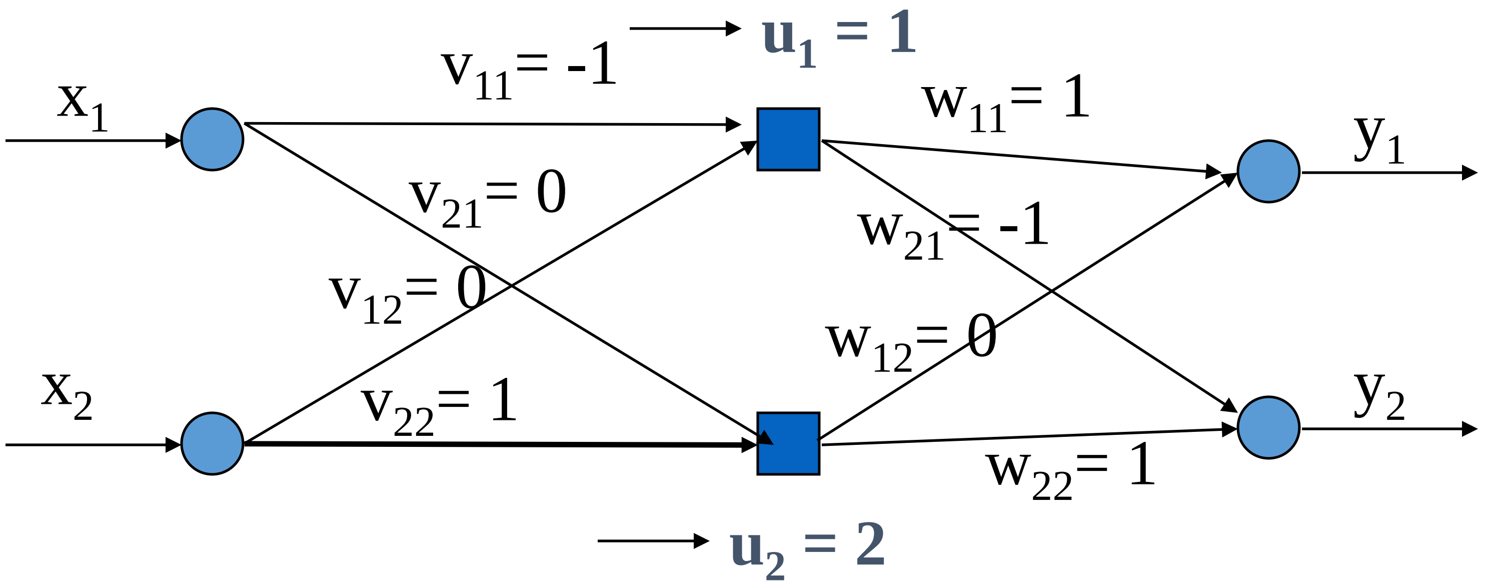
All biases set to 1. Will not draw them for clarity.

Learning rate $\eta = 0.1$



Have input [0 1] with target [1 0].

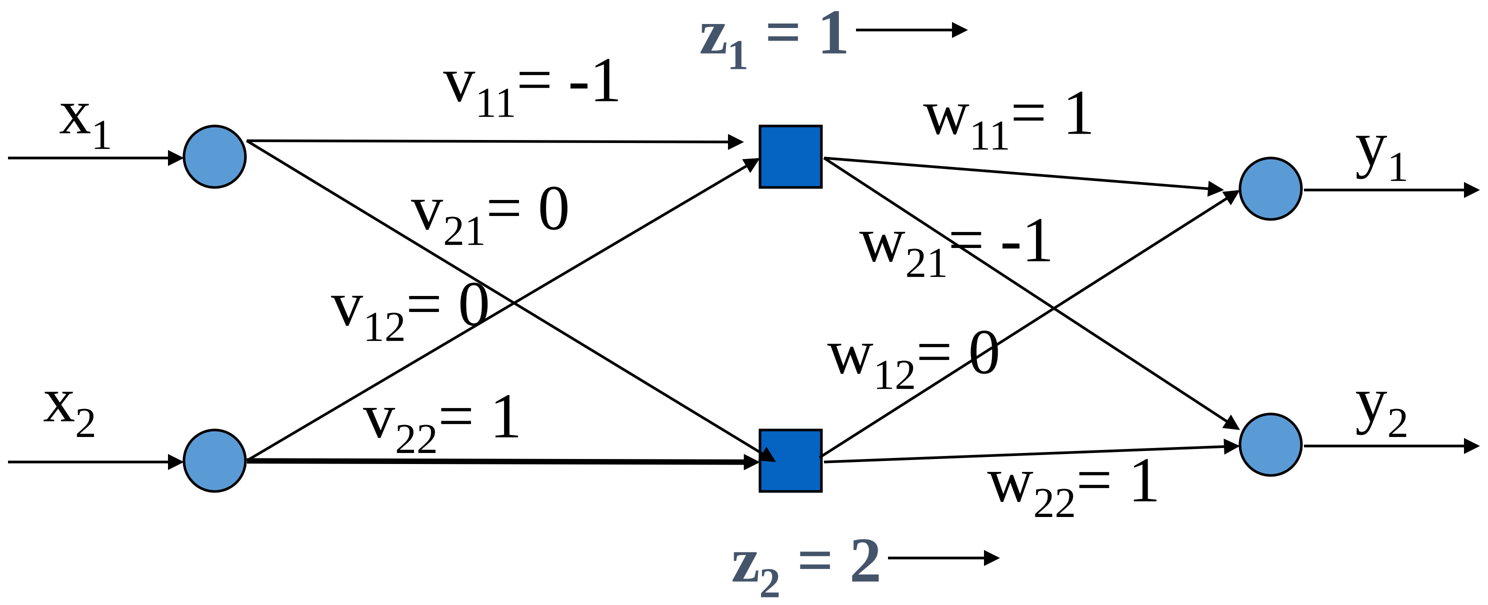
Forward pass. Calculate 1st layer activations:



$u_1 = -1x0 + 0x1 +1 = 1$

$u_2 = 0x0 + 1x1 +1 = 2$

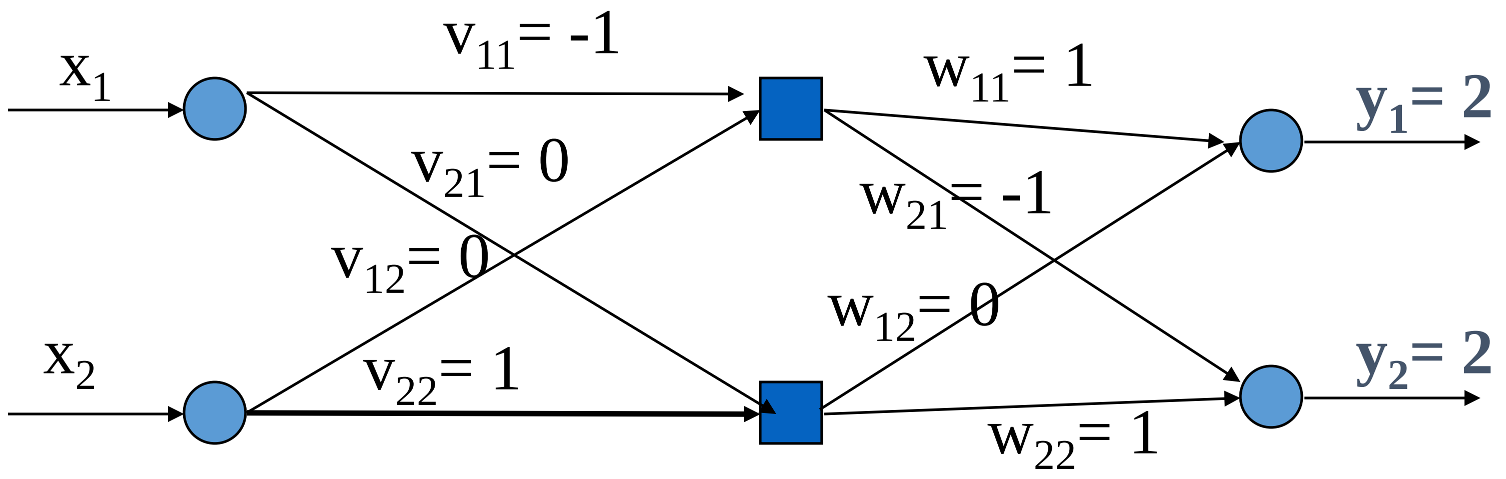
Calculate first layer outputs by passing activations thru activation functions



$z_1 = g(u_1) = 1$

$z_2 = g(u_2) = 2$

Calculate 2nd layer outputs (weighted sum thru activation functions):

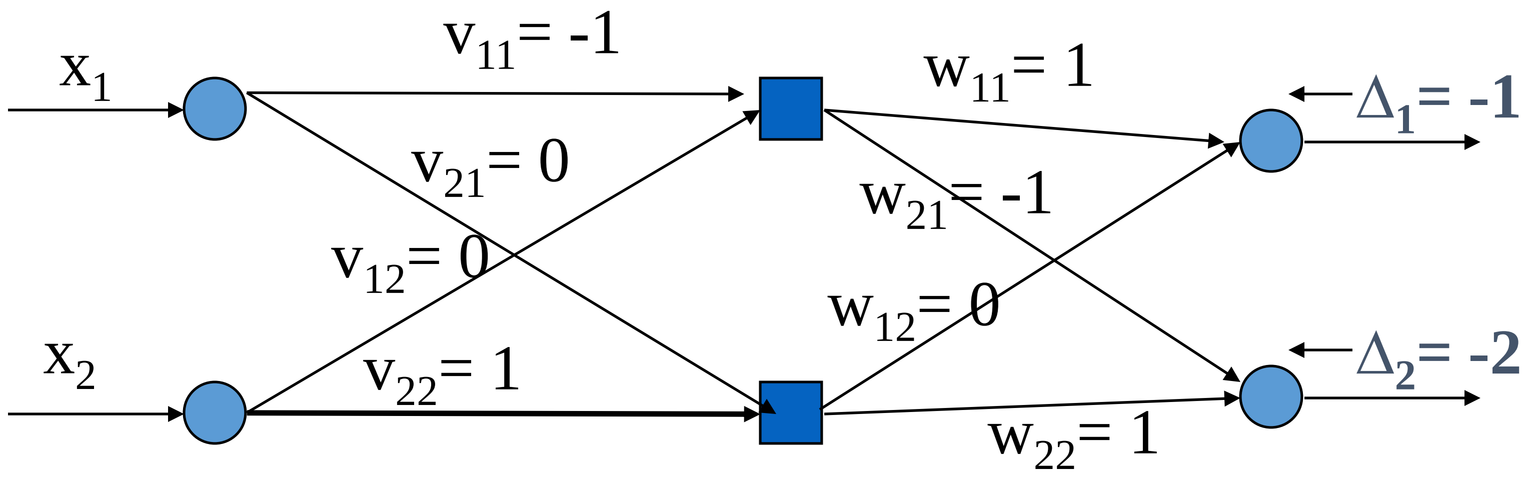


$y_1 = a_1 = 1x1 + 0x2 +1 = 2$

$y_2 = a_2 = -1x1 + 1x2 +1 = 2$

Backward pass:

[redacted]



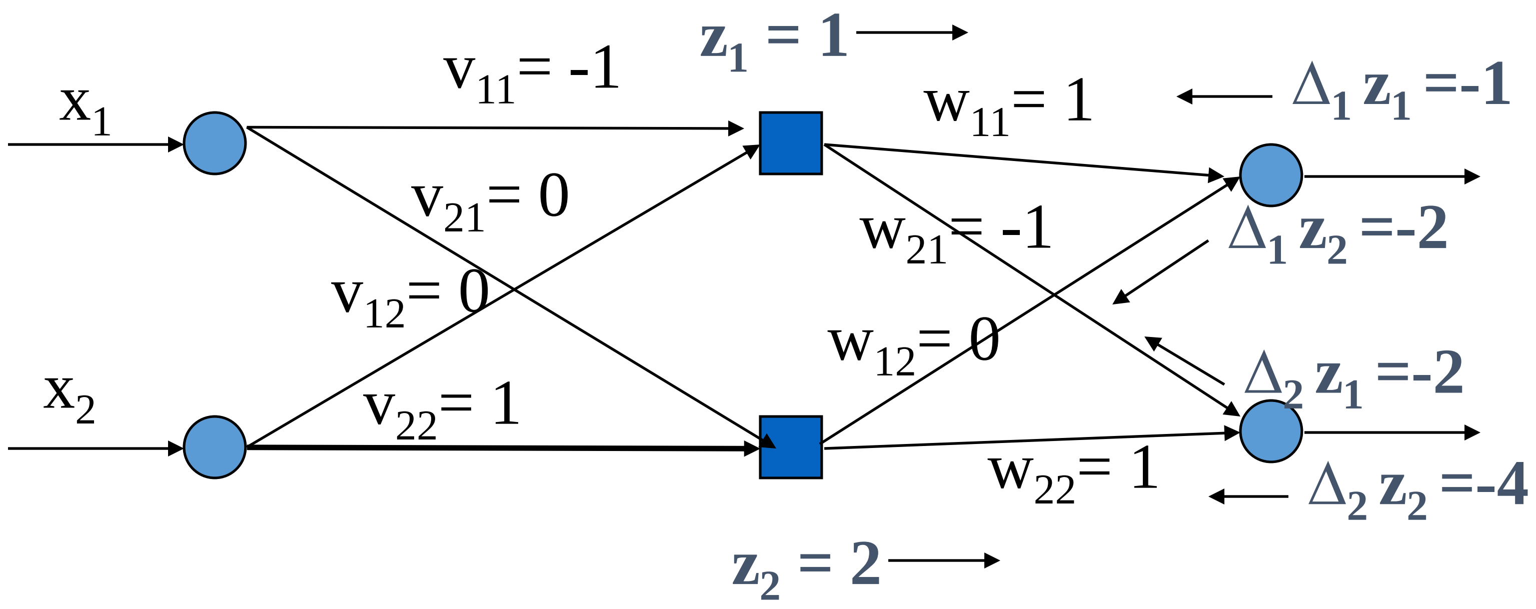
Target =[1, 0] so $d_1 = 1$ and $d_2 = 0$

So:

$\Delta_1 = (d_1 - y_1)= 1 – 2 = -1$

$\Delta_2 = (d_2 - y_2)= 0 – 2 = -2$

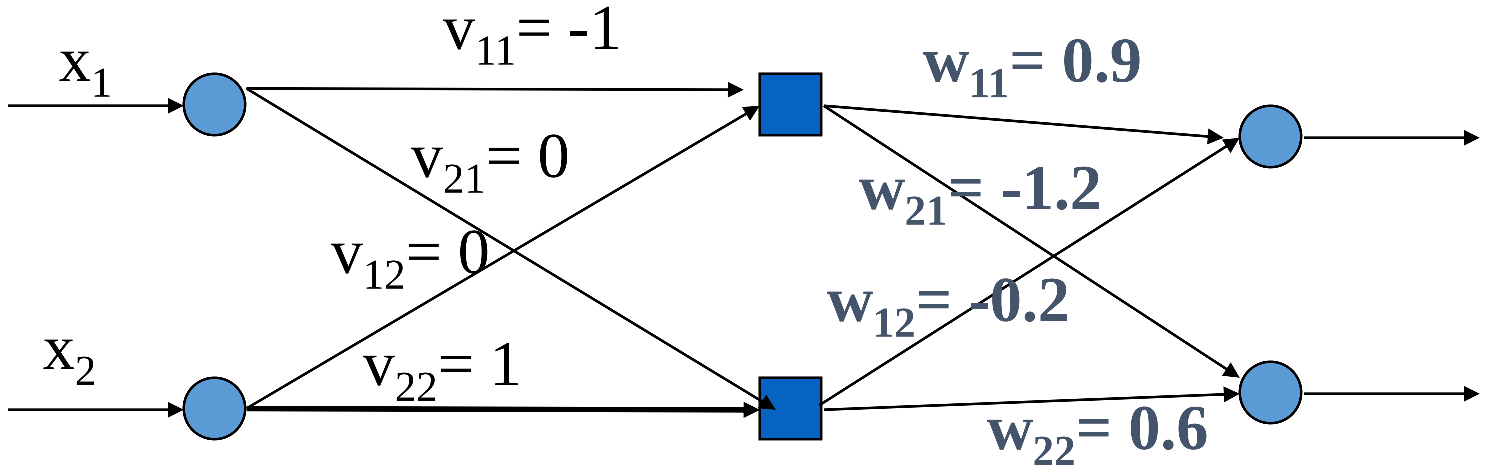
Calculate weight changes for 1st layer (cf perceptron learning):



[redacted]

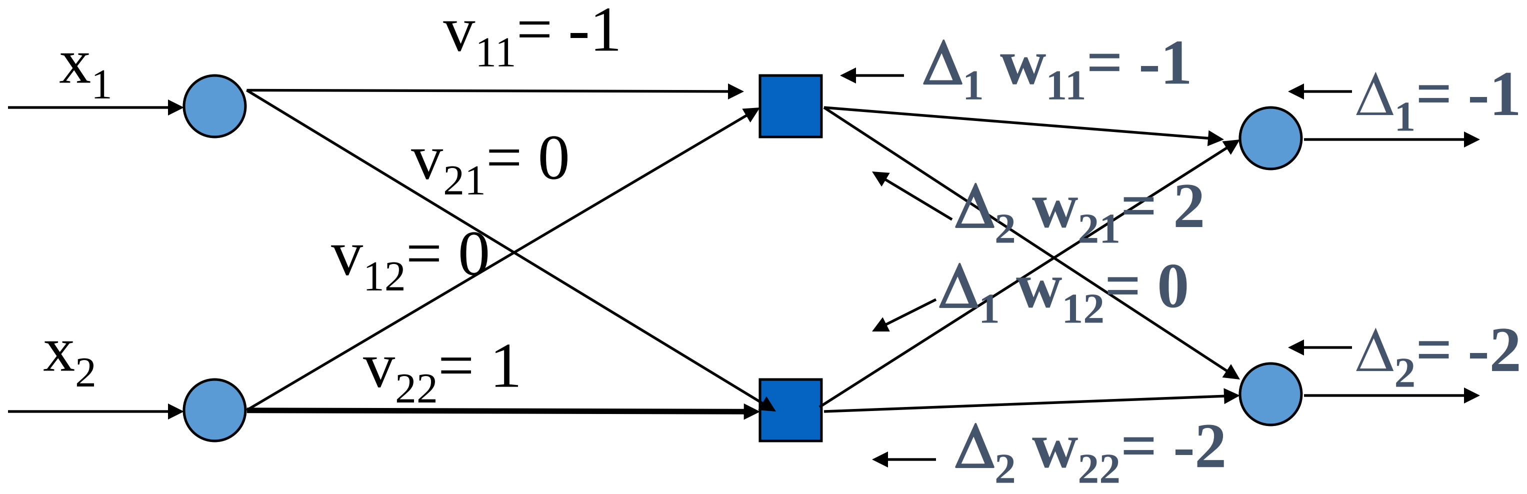
Weight changes will be:

[redacted]



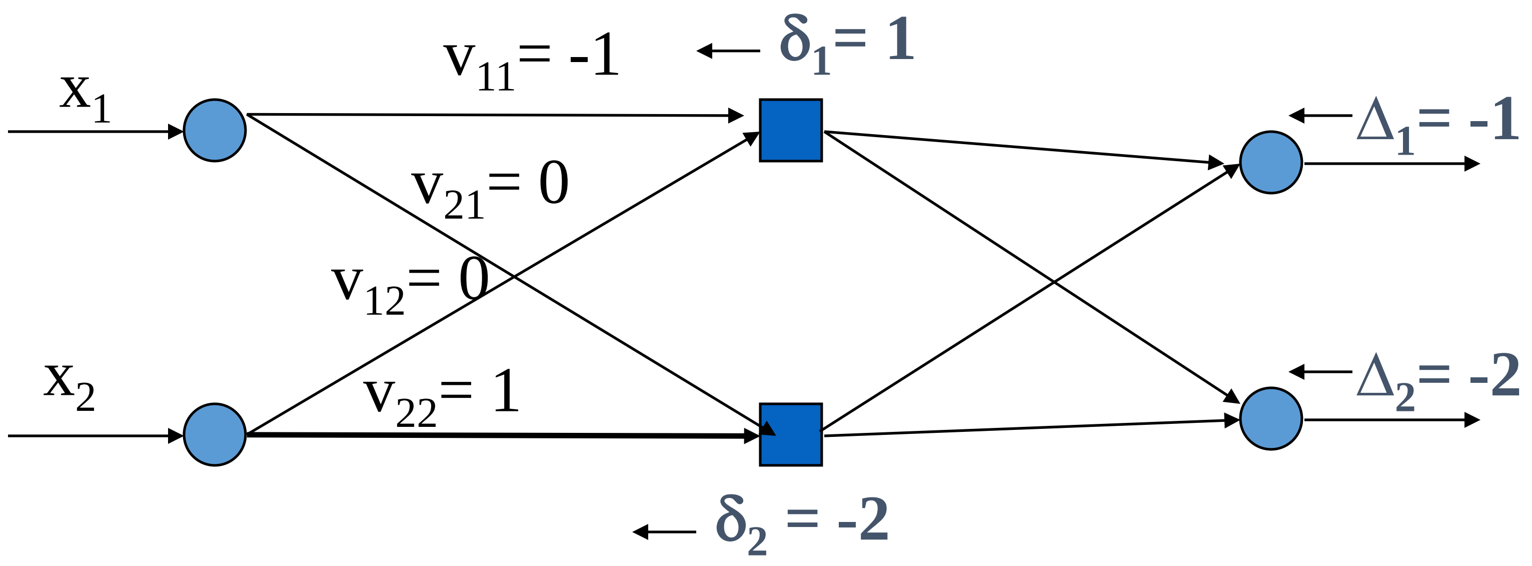
But first must calculate δ's:

[redacted]



Δ’s propagate back:

[redacted]

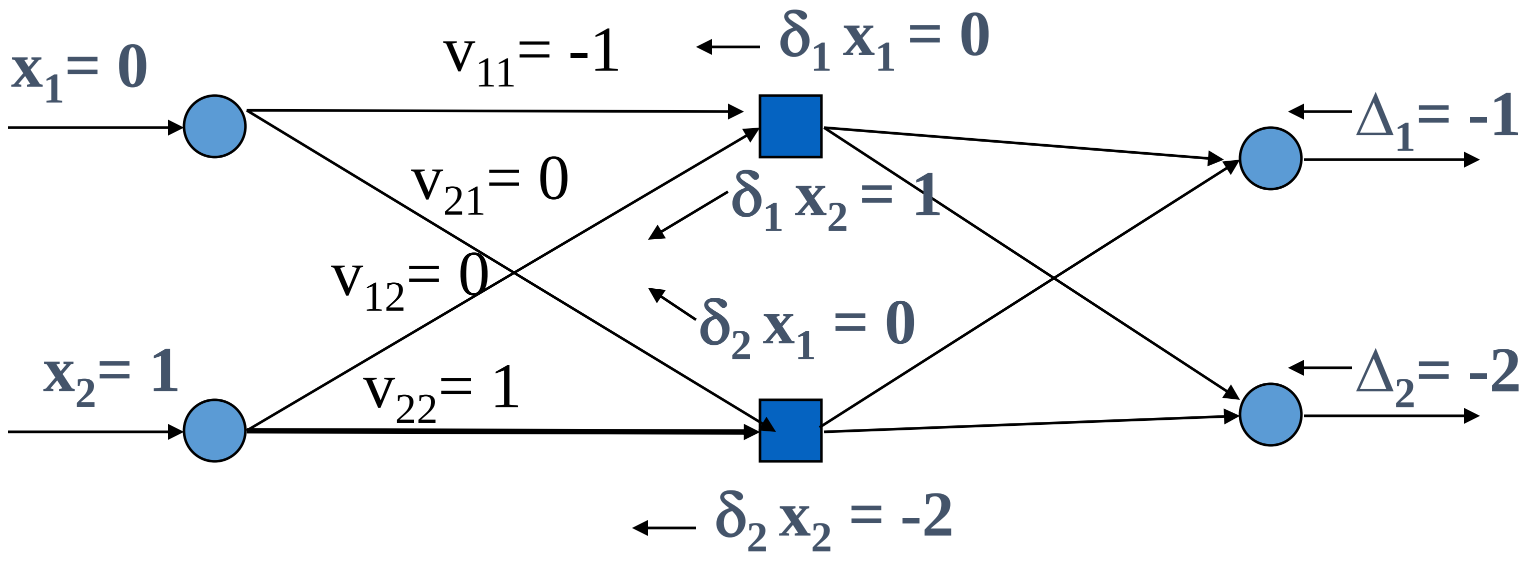


$\delta_1 = -1 + 2 = 1$

$\delta_2 = 0 - 2 = -2$

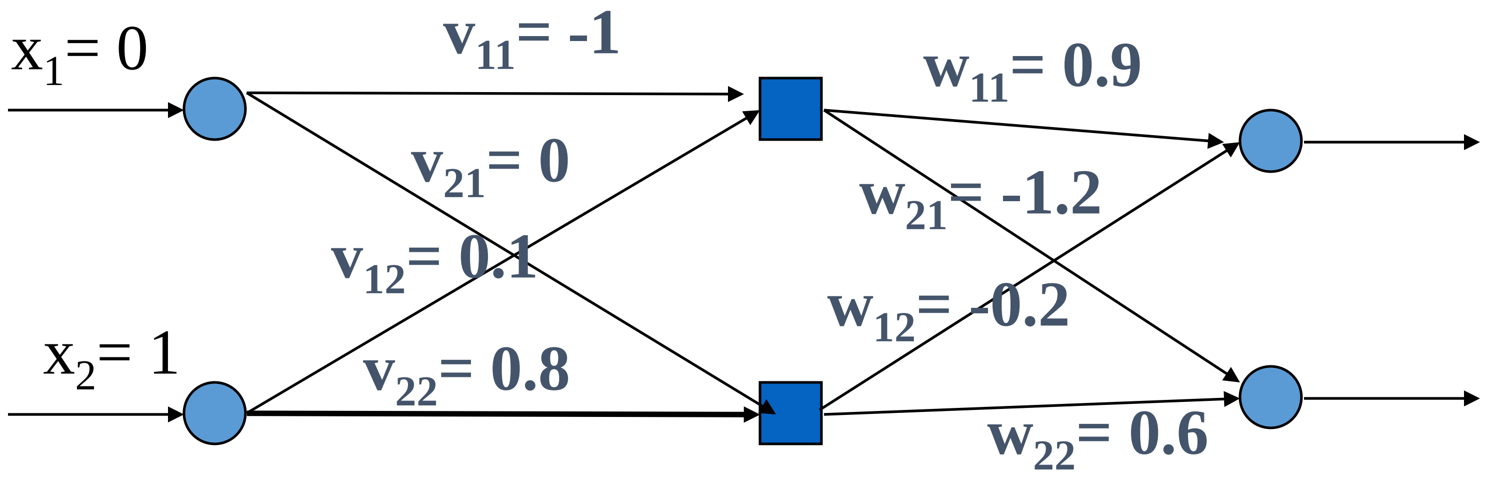
And are multiplied by inputs:

[redacted]



Finally change weights:

[redacted]

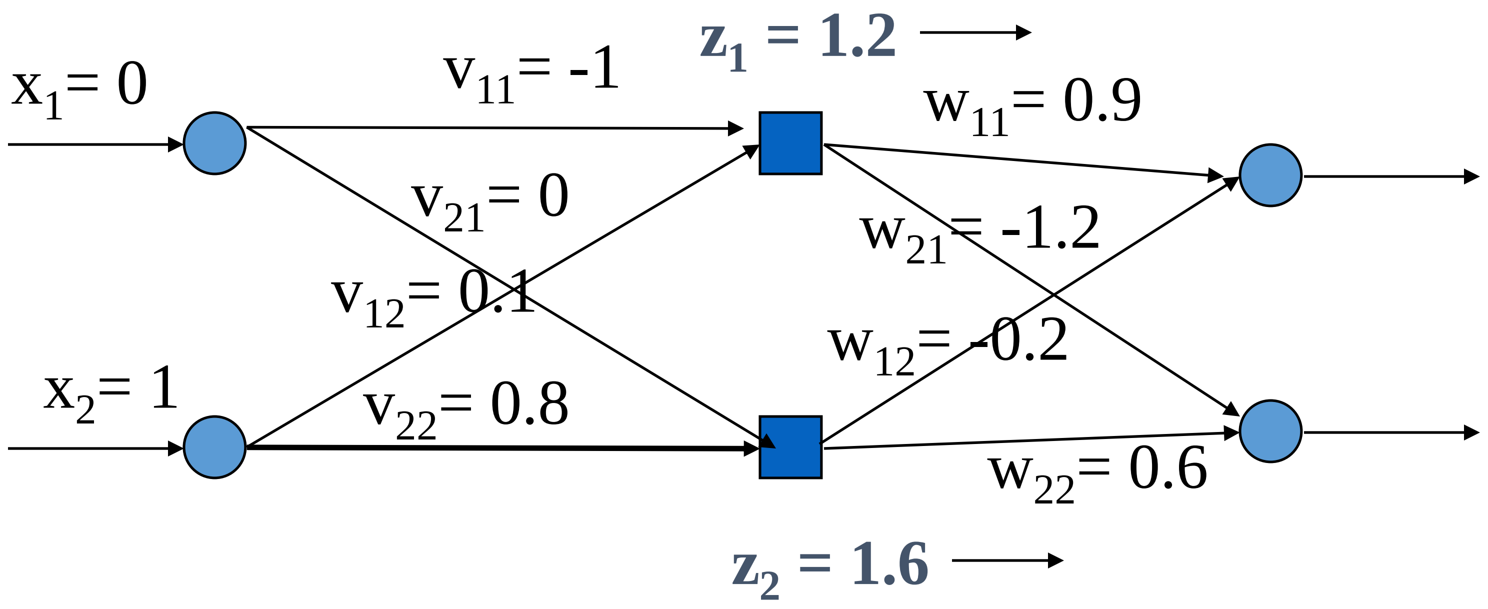


Note that the weights multiplied by the zero input are unchanged as they do not contribute to the error

We have also changed biases (not shown)

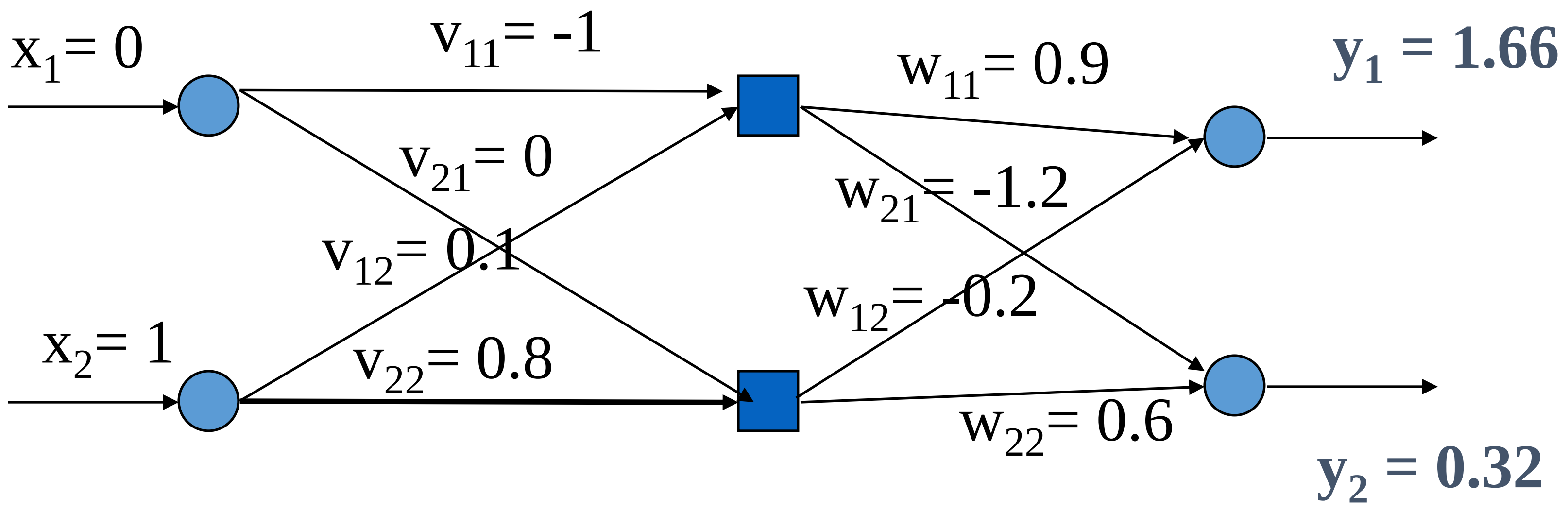
Now go forward again (would normally use a new input vector):

[redacted]



Now go forward again (would normally use a new input vector):

[redacted]



Outputs now closer to target value [1, 0]

Activation Functions

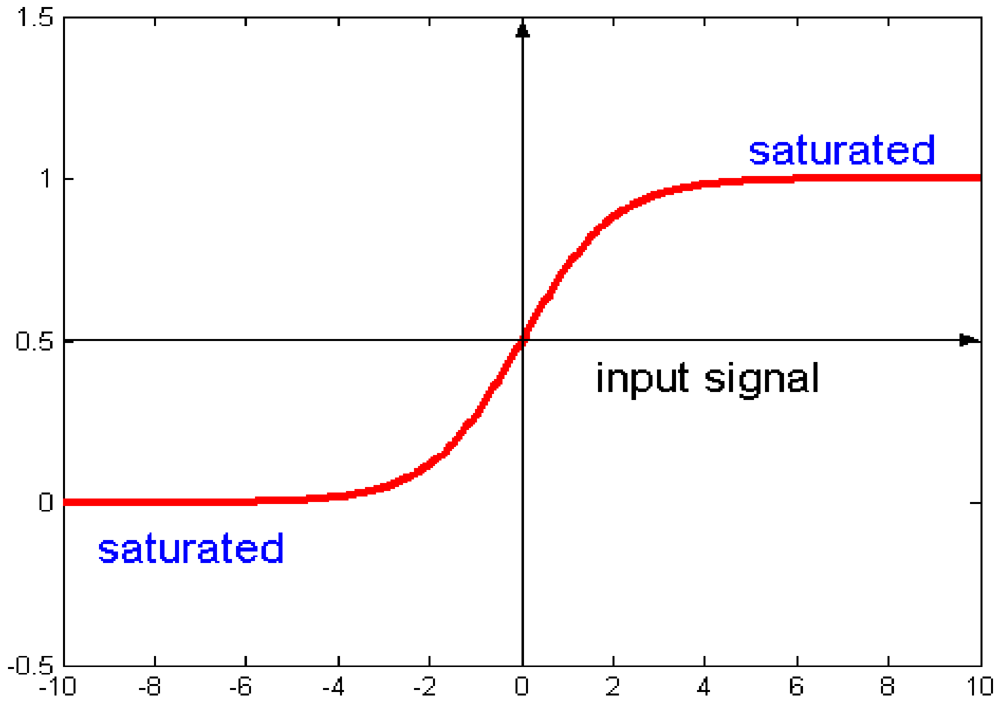
How does the activation function affect the changes?

[redacted]

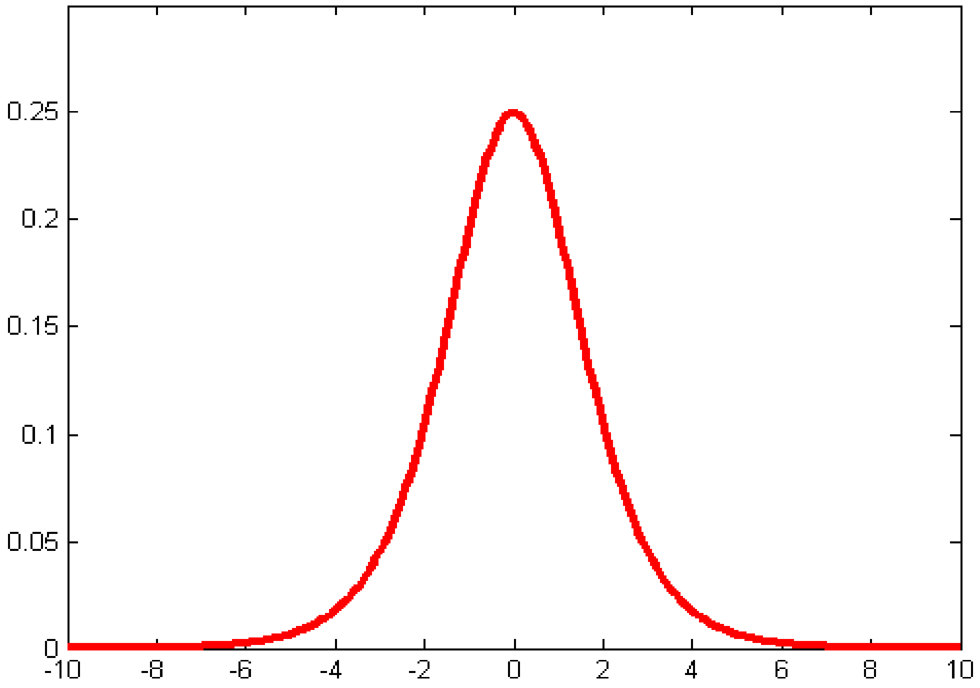
[redacted]

Where: [redacted]

[redacted]


saturated
saturated
input signal
1.5
1
0.5
0
-0.5
-10
-8
-6
-4
-2
0
2
4
6
8
10

[redacted]



[redacted]

[redacted]

Summary of (sequential) BP learning algorithm

$\boxed{\eta}$

present input pattern to input units
compute functional signal for hidden units
compute functional signal for output units

present Target response to output units
computer error signal for output units
compute error signal for hidden units
update all weights at same time
increment n to n+1 and select next input and target

- Sequential mode (on-line, stochastic, or per-pattern) Weights updated after each pattern is presented

- Batch mode (off-line or per -epoch). Calculate the derivatives/wieght changes for each pattern in the training set. Calculate total change by summing imdividual changes

Sequential mode

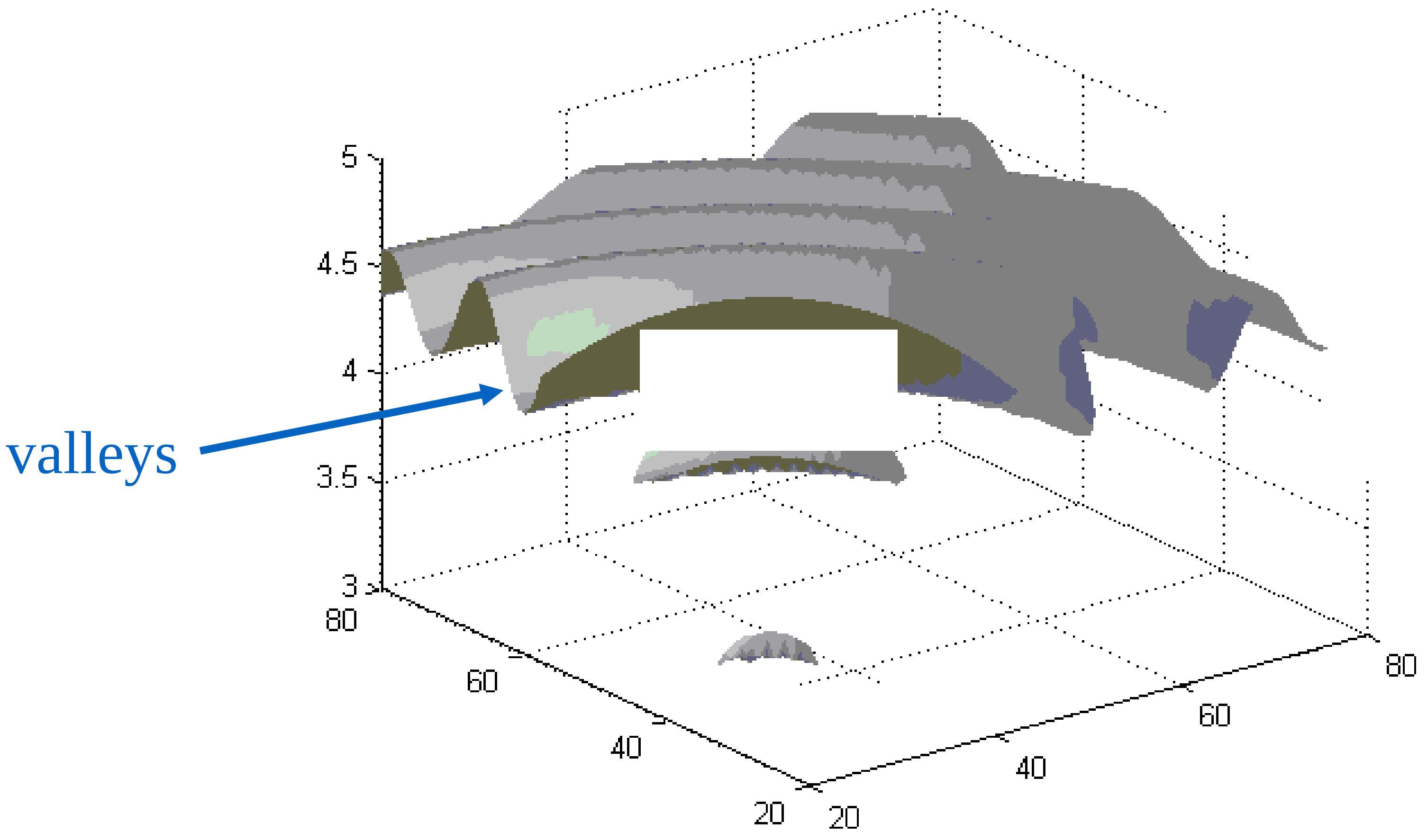
- Less storage for each weighted connection
- Random order of presentation and updating per pattern means search of weight space is stochastic--reducing risk of local minima
- Able to take advantage of any redundancy in training set (i.e.. same pattern occurs more than once in training set, esp. for large difficult training sets)
- Simpler to implement

Batch mode:

- Faster learning than sequential mode
- Easier from theoretical viewpoint
- Easier to parallelise

Dynamics of BP learning

$$\boxed{\frac{1}{2}\sum_{k=1}^{p}(d_k(t)-O_k(t))^2}$$


valleys
5
4.5
4
3.5
3
80
60
40
20
20
40
60
80

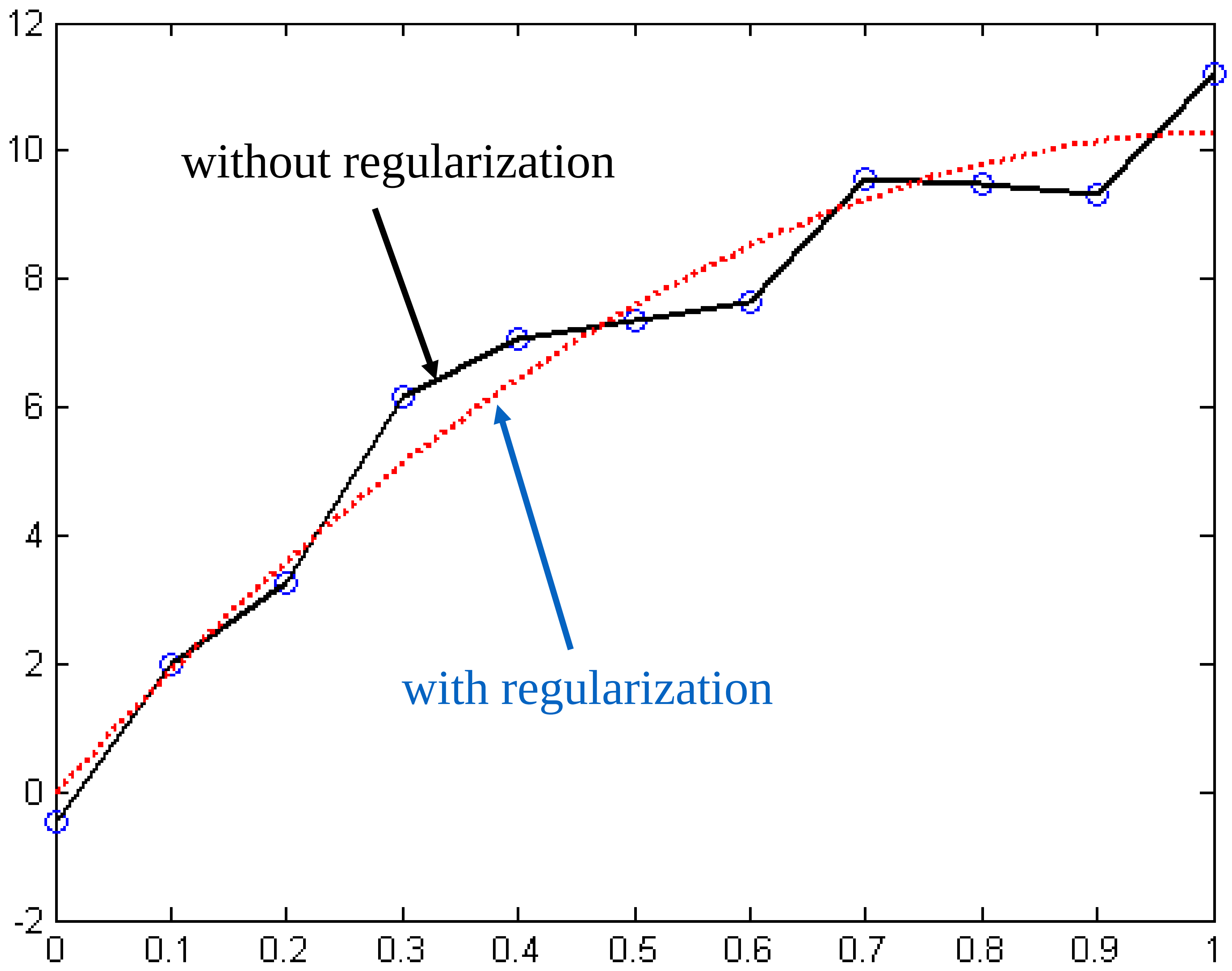
Selecting initial weight values

•

Regularization – a way of reducing variance (taking less notice of data)

$\lambda E_R(F)$

penalty term: require that the solution should be smooth, etc. Eg



without regularization
with regularization
12
10
8
6
4
2
0
-2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Momentum

$$\boxed{\begin{aligned} w_{ij}(n+1) - w_{ij}(n) &= \eta \delta_j(n) y_i(n) \\ &\quad + \alpha[w_{ij}(n) - w_{ij}(n-1)] \end{aligned}}$$

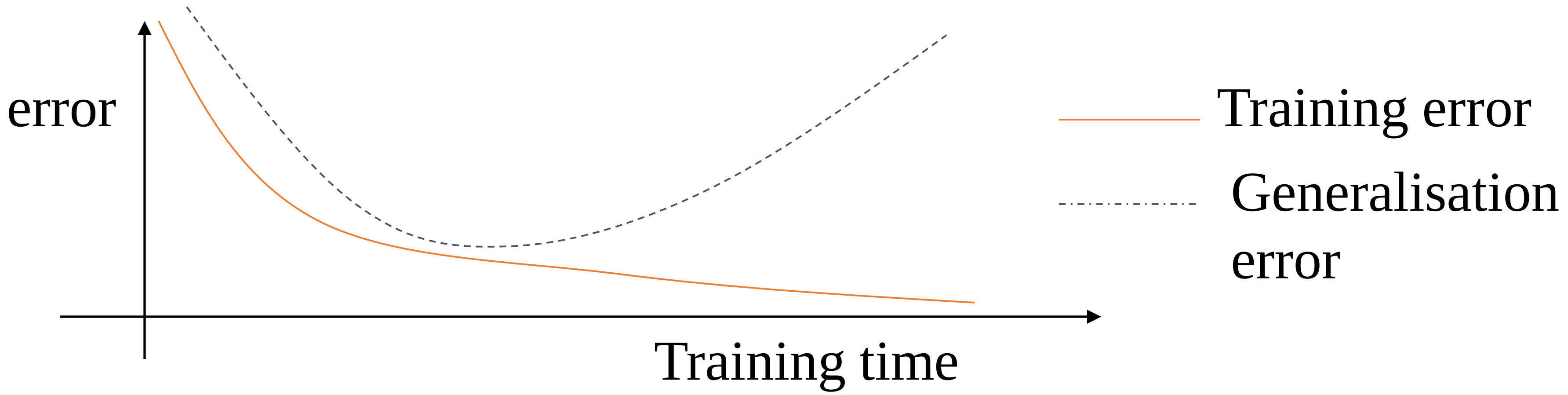
α is momentum constant and controls how much notice is taken of recent history

Effect of momentum term

- If weight changes tend to have same sign
  momentum terms increases and gradient decrease
  speed up convergence on shallow gradient
- If weight changes tend have opposing signs
  momentum term decreases and gradient descent slows to reduce oscillations (stablizes)
- Can help escape being trapped in local minima

[redacted]

However, aim is for new patterns to be classified correctly



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

Cross-validation

- Training data set

- Validation data

- Test data set

Evaluation of generalisation error ie network performance

Early stopping of learning to minimize the training error and validation error

Universal Function Approximation

Universal Approximation Theorem

The backpropagation algorithm

Learning in multi-layered networks

- Networks with one or more hidden layers are necessary to represent complex mappings
- In such a network the basic delta learning law is insufficient
  - It only defines how to update weights in output units (uses T-O)
- To update hidden node weights, we have to define *their* error
- This is achieved by the *Backpropagation* algorithm

The Backpropagation process

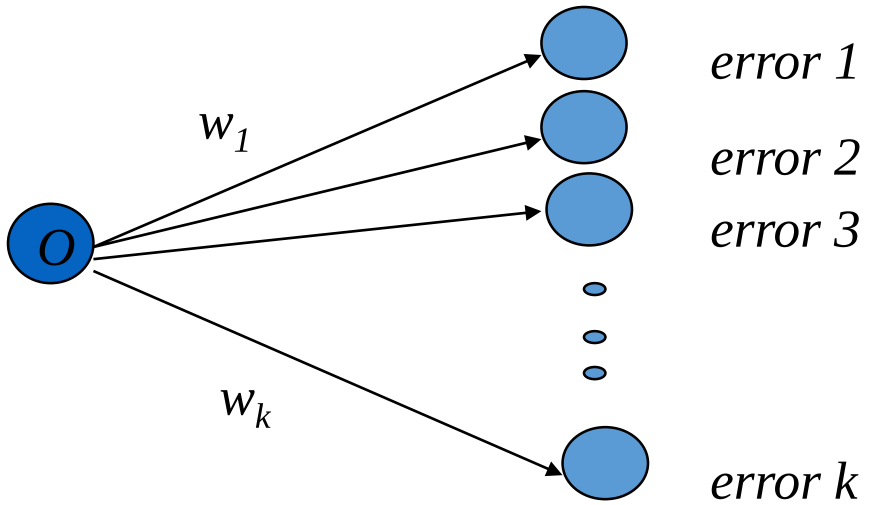
- Inputs are fed through the network in the usual way
  - this is the *forward pass*
- Output layer weights are adjusted based on errors...

  … then weights in the previous layer are adjusted …

  … and so on back to the first layer
  - this is the backwards pass (or *backpropagation*)
- Errors determined in a layer are used to determine those in the *previous* layer

Illustrating the error contribution

- A hidden node is partially ‘credited’ for errors in the next layer
  - these errors are created in the forward pass



$$error\_contribution = w_1 * error\ 1 + \ldots + w_k * error\ k$$

The backpropagation algorithm

- A *backpropagation network* is
  - a multi-layered feed-forward network
  - using the sigmoid response activation function
- *Backpropagation algorithm*

```
1. Initialise all network weights to small random numbers (between -0.05 and 0.05)
2.  begin iteration
        for each training example do:
            propagate input to output layer;
            from output layer, back propagate errors;
            update weights
    end epoch
3. If termination condition is met, stop else goto 2
```

Termination conditions

- Many thousands of iterations (epochs or cycles) may be necessary to learn a classification mapping
  - The more complex the mapping to be learnt, the more cycles will be required
- Several termination conditions are used:
  - stop after a given number of cycles
  - stop when the error on the training examples (or on a separate validation set) falls below some agreed level
- Stopping too soon results in *underfitting*, too late in *overfitting*

Backpropagation as a search

- Learning is a search for a network weight vector to implement the required mapping
- The search is hill-climbing or rather *descending* called *steepest gradient descent*
    - The heuristic used is the total of $(T\text{-}O)^2$ over all examples fed in a single cycle
    - the weight update produces the greatest fall in overall error for the size of step
- As with all hill-climbing there is the danger of sticking in local minima

Problems with the search

- The size of step is controlled by the learning rate parameter
  - This must be tuned for individual problems
  - If the step is too large search becomes inefficient
- The error surface tends to have
  - extensive flat areas
  - troughs with very little slope
- It can be difficult to reduce error in such regions
  - Weights have to move large distances and it can be hard to determine the right direction
  - High numerical accuracy is required, e.g. 32-bit floating point
  - On the bright side there tend to be many global minima and few local minima

The trained network

- After learning, Backpropagation may be used as a classifier:
  - Descriptions of new examples are fed into the network and the class is read from the output layer
  - For 1-out-of-N output representations, exact values of 0 and 1 will not usually be obtained
- *Sensitivity analysis* (using test data) determines which attributes are most important for classification
  - An attribute is regarded as important if small changes in its value affect the classification