CS 412

FEB 20TH – NEURAL NETWORKS

HTF – CHAPTER 11

Neural Networks

Networks of processing units (neurons) with connections (synapses) between them

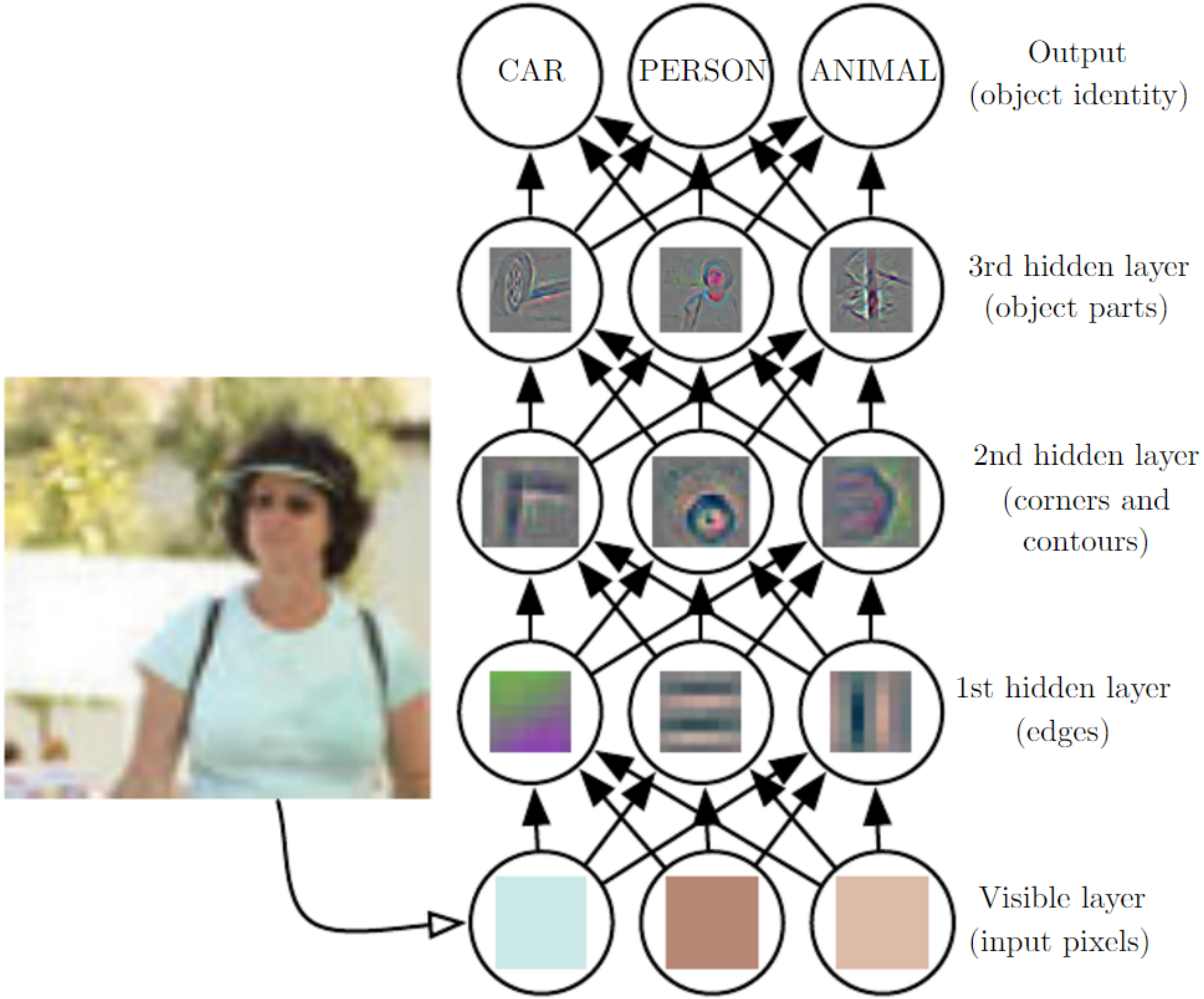
Large number of neurons: 10^{10}

Large connectitivity: 10^5

Parallel processing

Distributed computation/memory

Robust to noise, failures


CAR
PERSON
ANIMAL
Output
(object identity)
3rd hidden layer
(object parts)
2nd hidden layer
(corners and
contours)
1st hidden layer
(edges)
Visible layer
(input pixels)

Understanding the Brain

Levels of analysis for an information processing system such as sorting (Marr, 1982)

1. Computational theory: goal of computation and abstract definition of the task
2. Representation and algorithm: how to represent input and output, and how to transform from input to output
3. Hardware implementation

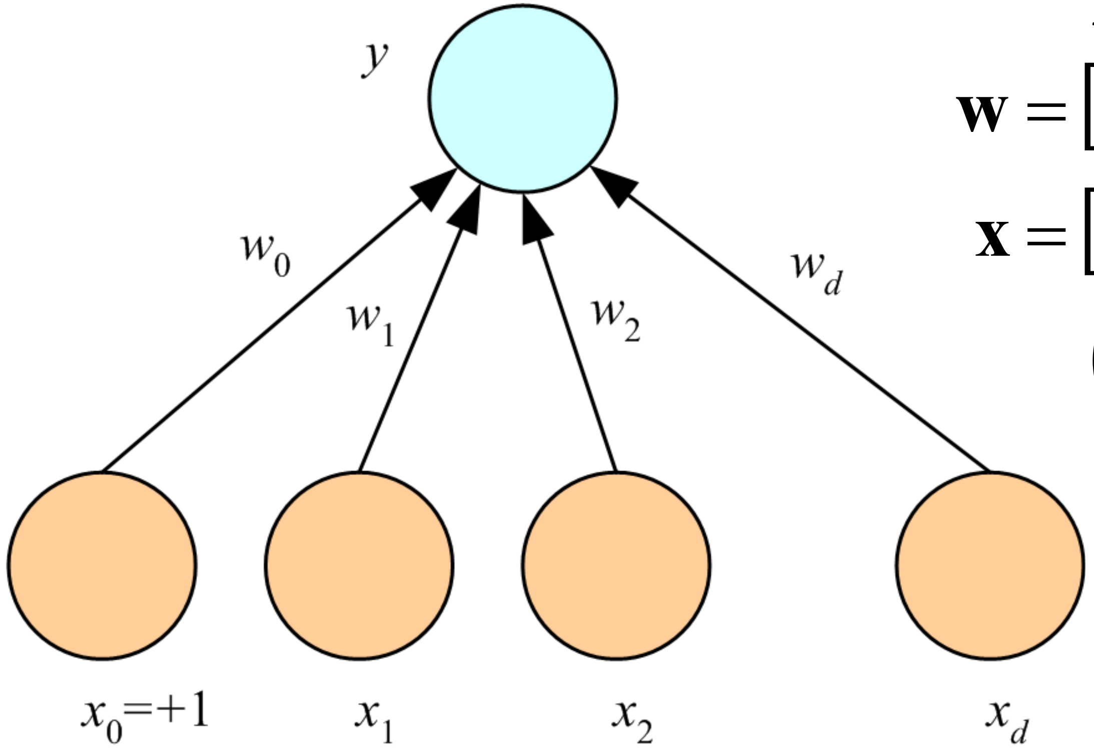
Reverse engineering: From hardware to theory

Parallel processing: SIMD vs MIMD

Neural net: SIMD with modifiable local memory

Learning: Update by training/experience

Perceptron



$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{w} = \left[w_0, w_1, \ldots, w_d\right]^T$$

$$\mathbf{x} = \left[1, x_1, \ldots, x_d\right]^T$$

(Rosenblatt, 1962)

Perceptron

What is the single-layer perceptron?

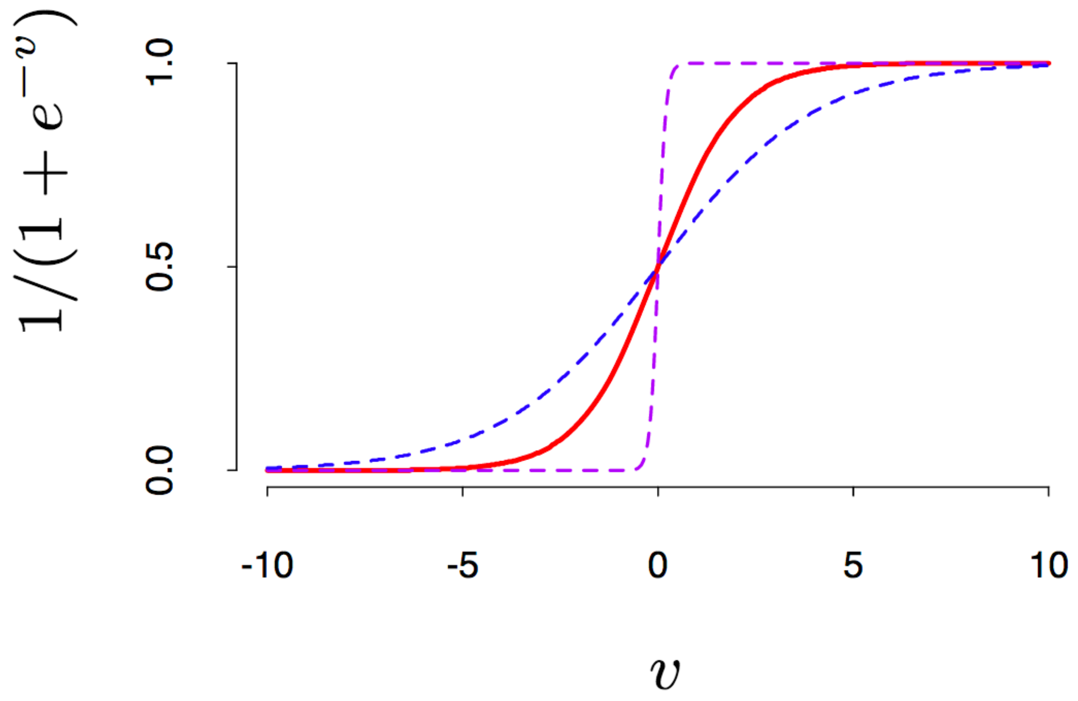
- Just the linear discriminator
- No support vector constraints

How do we train it?

- Stochastic gradient descent
  - Small changes based on the data – minimizing loss (what loss should we minimize?)
  - Update = learning factor*(DesiredOutput – Actual Output) * Input

$$\Delta w_{ij}^{t} = \eta\left(r_i^t - y_i^t\right)x_j^t$$

  - Descent is moderated by our learning factor (eta)



Perceptron

How do we make a regression model into a classification model?

- Activation function (here: sigmoid)
- Like logistic regression, there is no unique solution, so we also have to consider the rate at which the sigmoid transitions, this is the **activation rate**, s (here: ½,1,10)

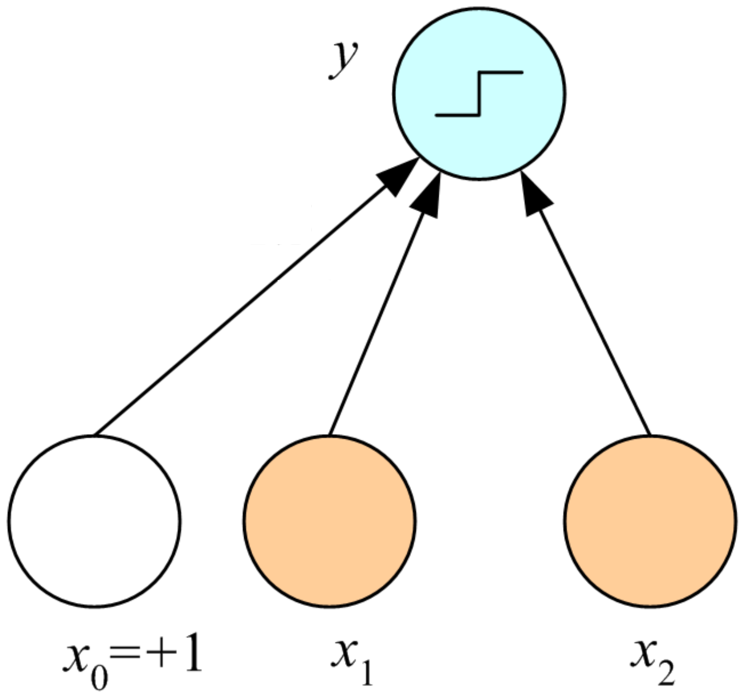
Why do we prefer this to the sign function?

- They are differentiable and non-linear

Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

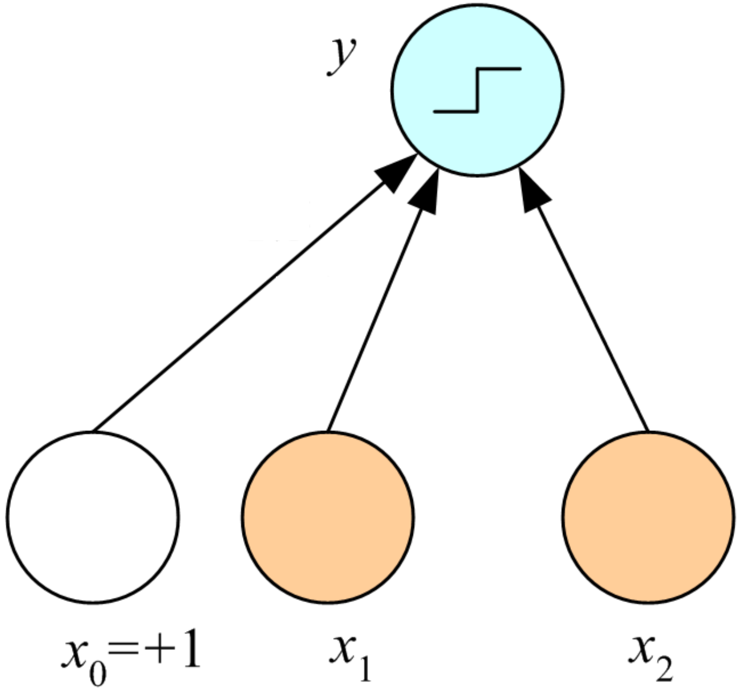
What are the weights for this perceptron?
(purple line)

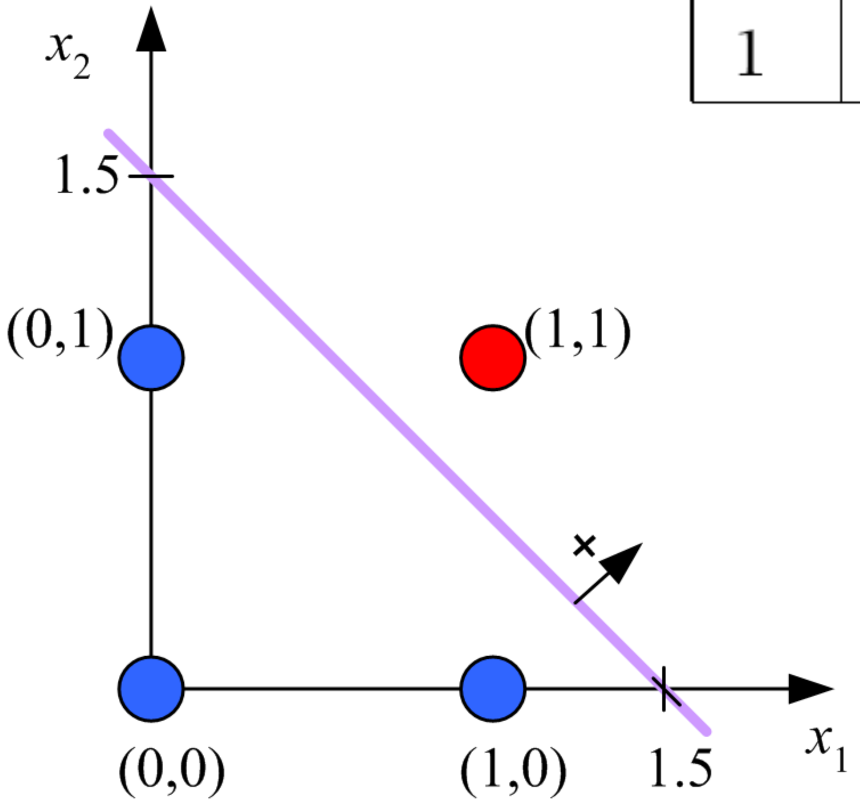


Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

What are the weights for this perceptron?
(purple line)

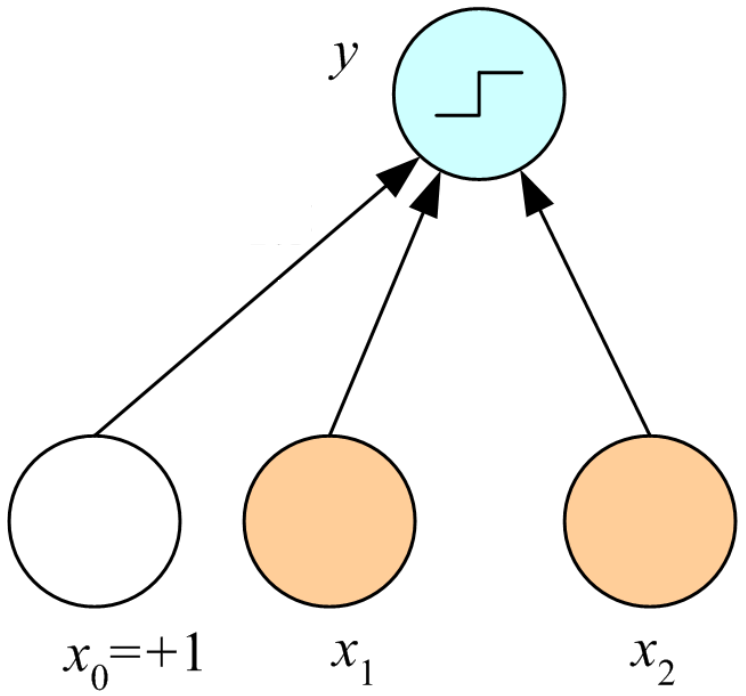


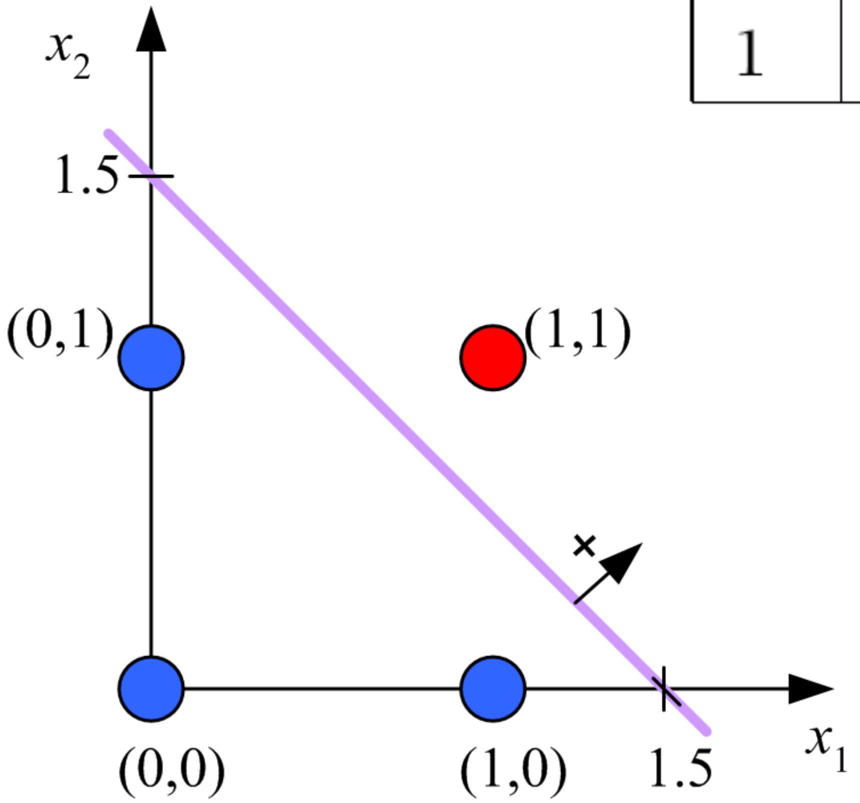


Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

What are the weights for this perceptron?
(purple line)

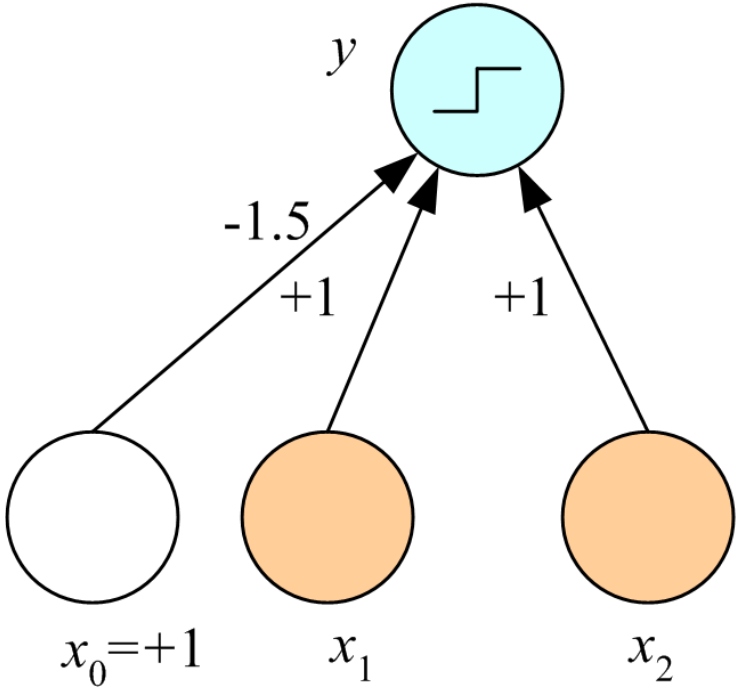


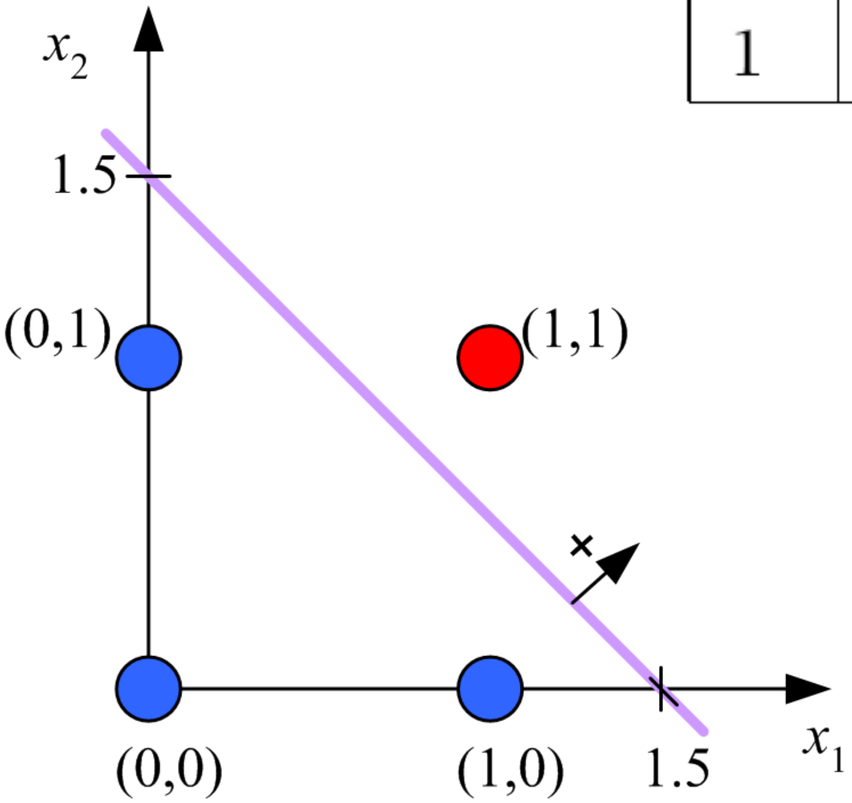


Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

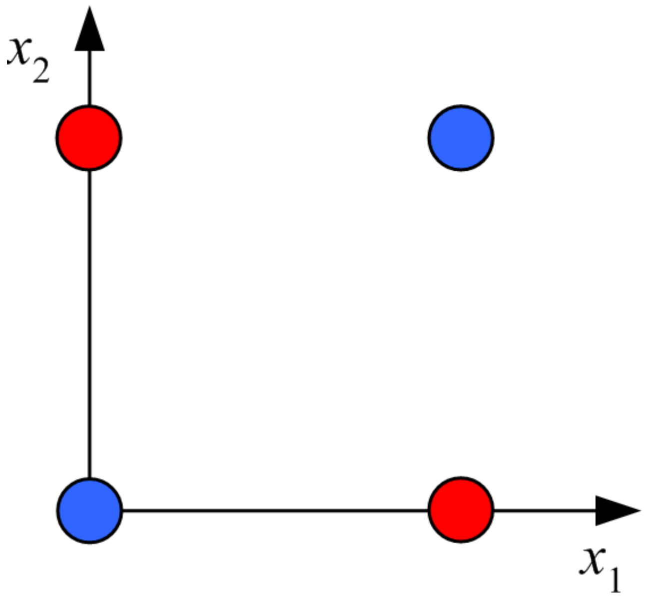
What are the weights for this perceptron?
(purple line)





XOR

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

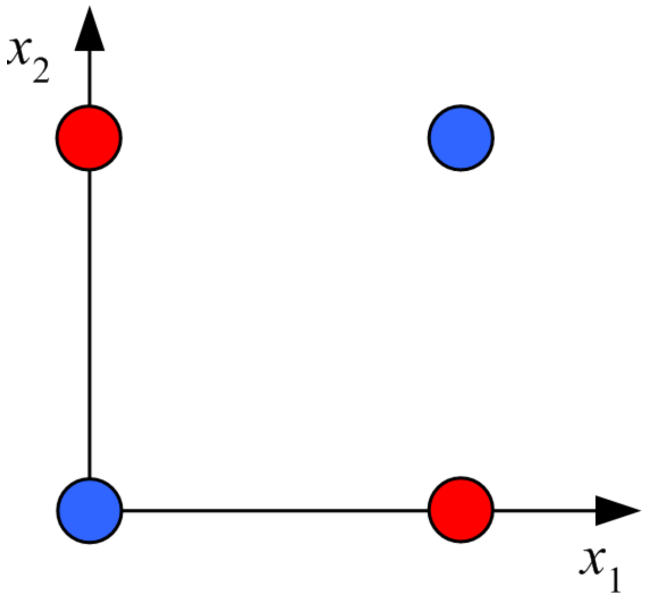


XOR

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

No w_0, w_1, w_2 satisfy:

$$
\begin{aligned}
& & & w_0 &&\leq 0 \\
& & w_2 + {} & w_0 &&> 0 \\
w_1 + {} & & & w_0 &&> 0 \\
w_1 + {} & & w_2 + {} & w_0 &&\leq 0
\end{aligned}
$$



Perceptron

What is the problem with the simple perceptron?

Perceptron

What is the problem with the simple perceptron?

- It can't model non-linear data

How do we fix this?

Perceptron

What is the problem with the simple perceptron?

- It can't model non-linear data

How do we fix this?

- SVM fixed this by using the kernel methods
- Can the perceptron?

Perceptron

What is the problem with the simple perceptron?

- It can't model non-linear data

How do we fix this?

- SVM fixed this by using the kernel methods
- Can the perceptron? **Yes**, but that's not what the neural network does

Perceptron

What is the problem with the simple perceptron?

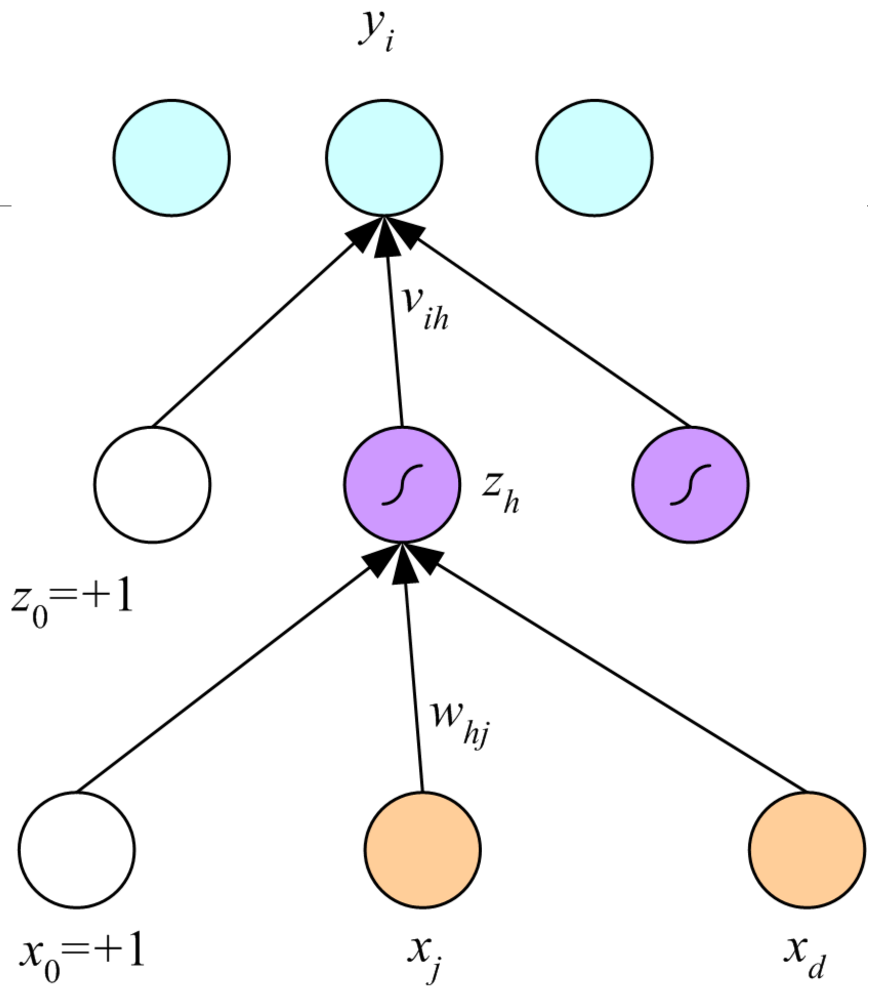
- It can't model non-linear data

How do we fix this?

- SVM fixed this by using the kernel methods
- Can the perceptron? **Yes**, but that's not what the neural network does

Let's add multiple layers to the perceptron

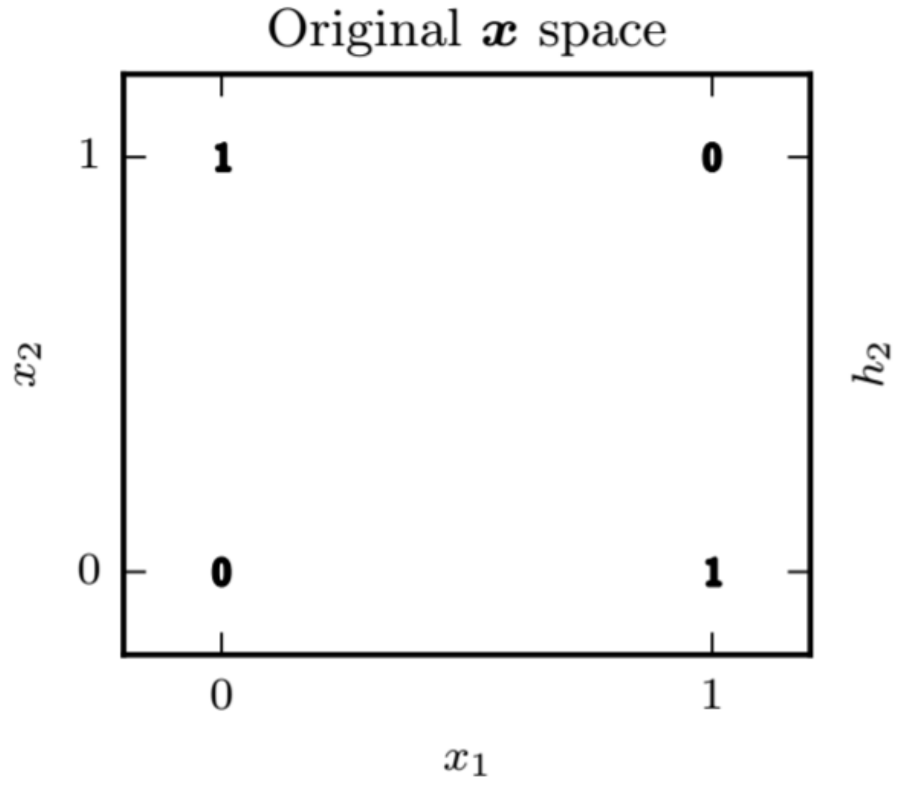
- At each level we have a **regression** model defined by the activation function and **always** a constant w_0



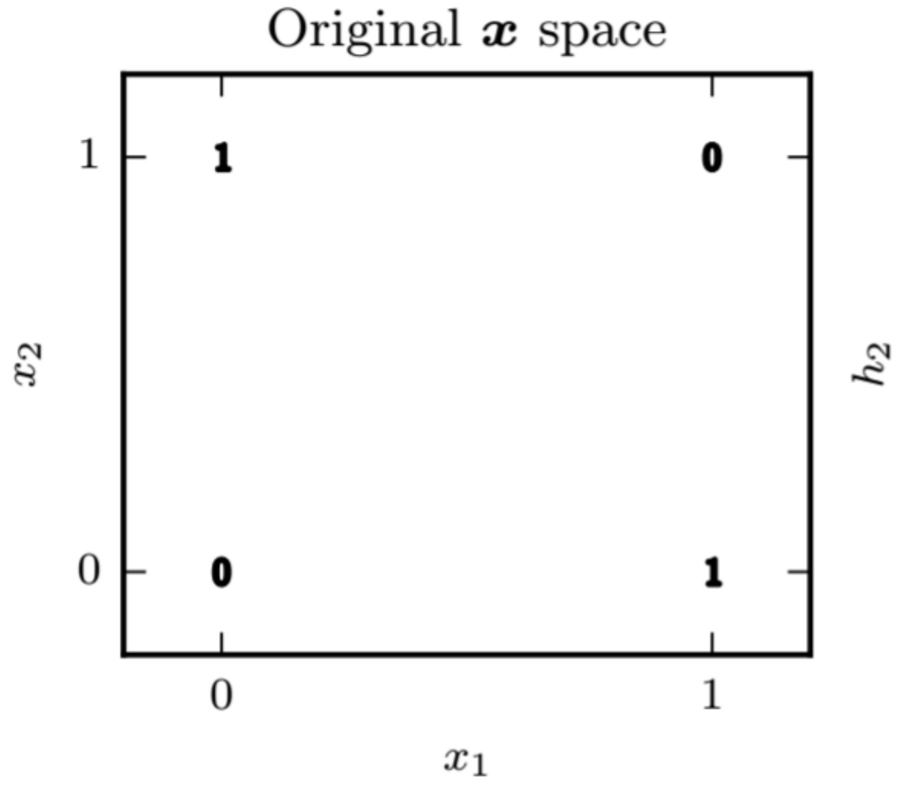
Perceptron

How do we use this to solve the XOR problem?

XOR

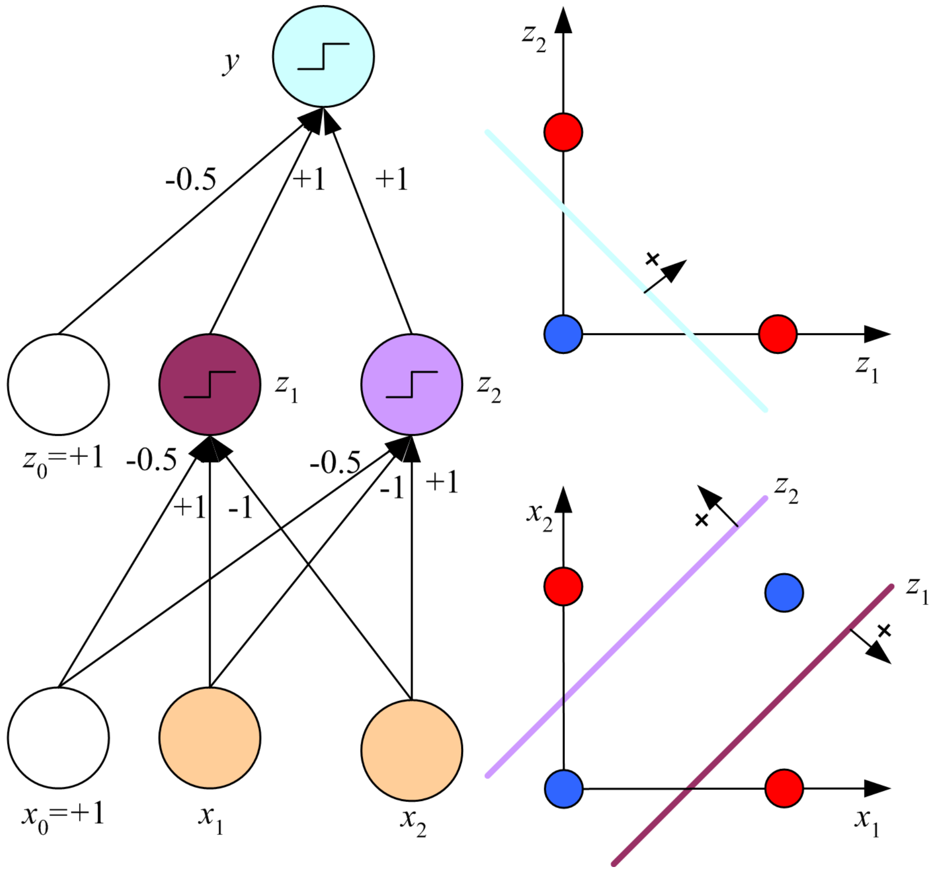


XOR



Perceptron

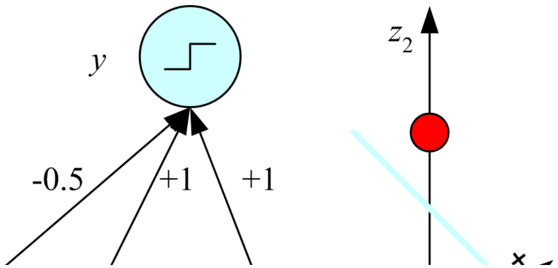
How do we use this to solve the XOR problem?

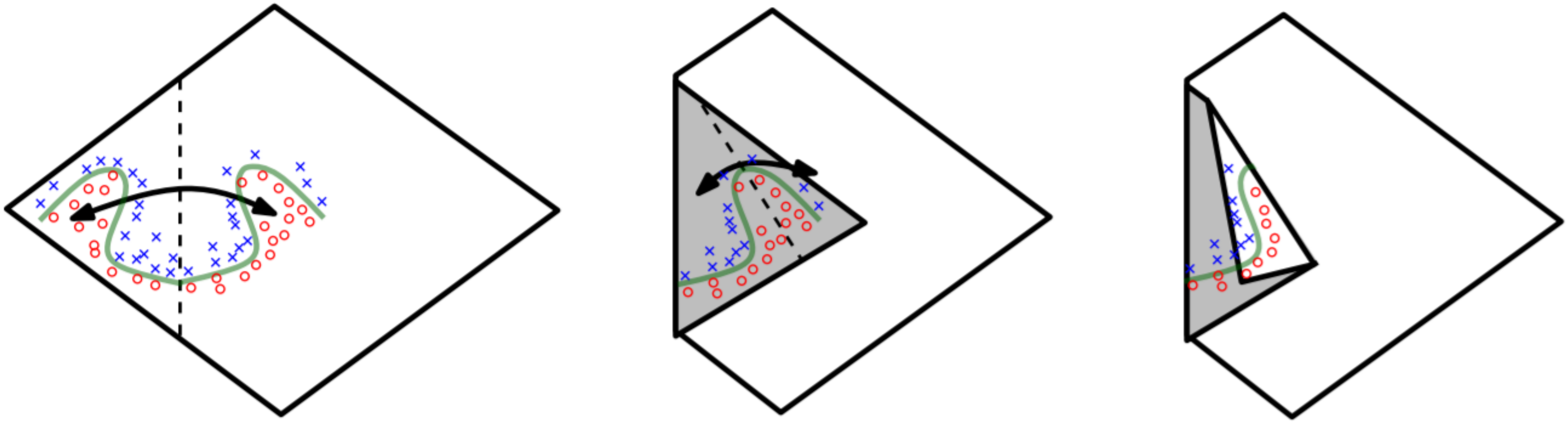


Perceptron

How do we use this to solve the XOR problem?

- What is happening here?





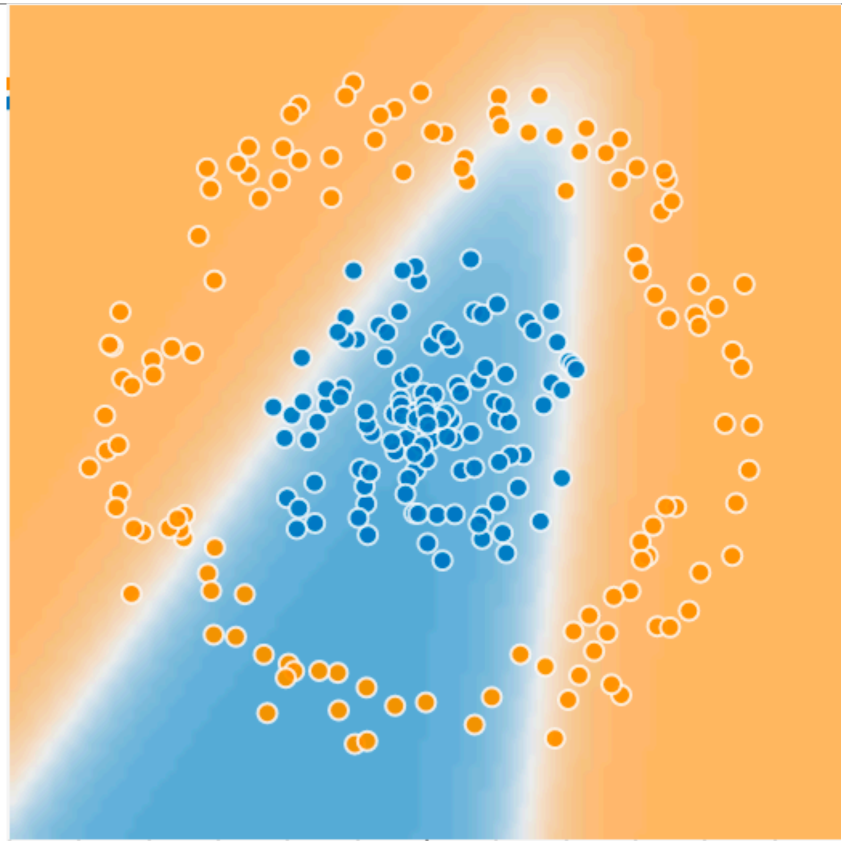
How to train?

We (hopefully) have some idea of how a particular set of weights causes the neural network to make a decision

- We have some vector of inputs X_d that are all fed as parameters to some number of nodes
- Each of these nodes outputs a sigmoid function to the next hidden layer
- This process eventually leads to the final layer, which makes the final prediction

How to train?

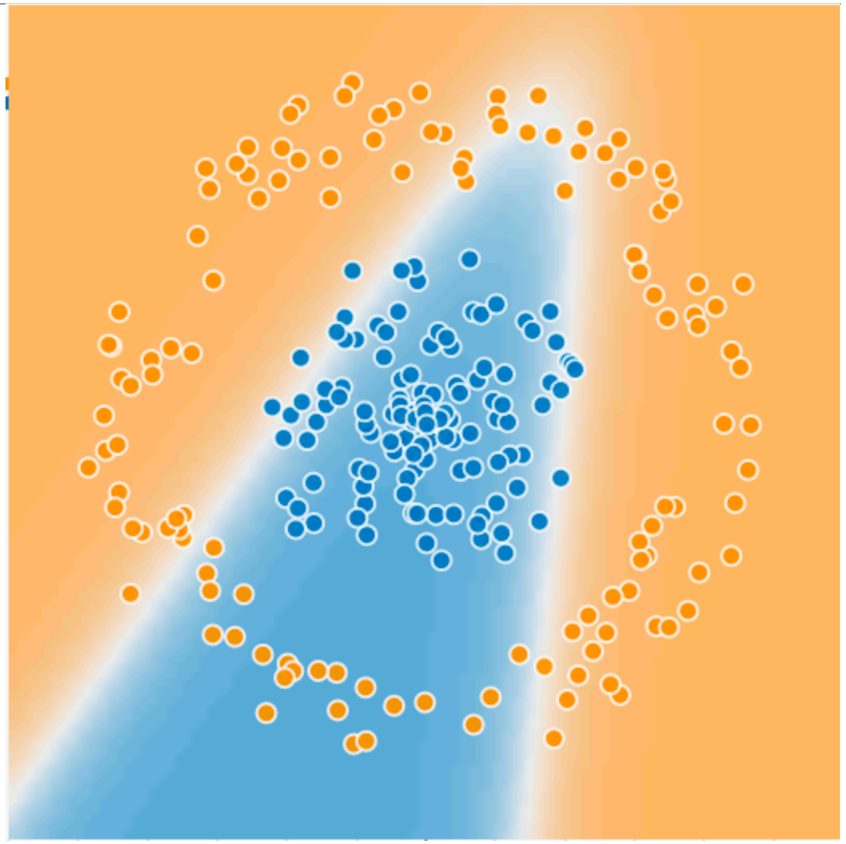
What is the structure of the neural network that produced this decision region?



How to train?

What is the structure of the neural network that produced this decision region?

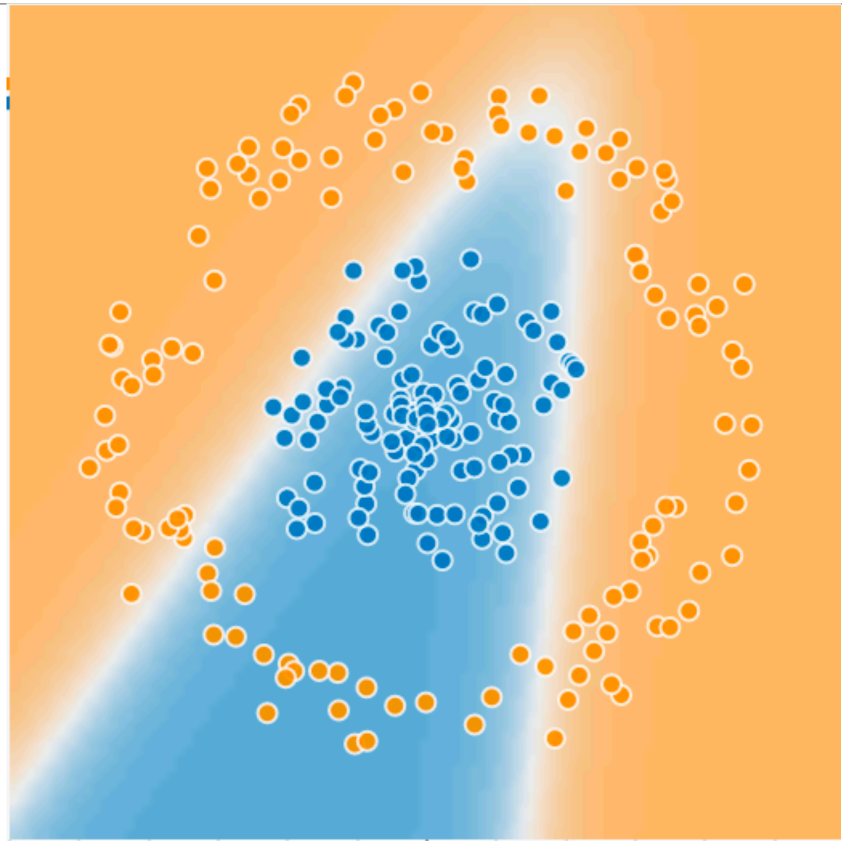
- Blue is positive, orange is negative
- What do the white regions represent?



How to train?

What is the structure of the neural network that produced this decision region?

- Blue is positive, orange is negative
- What do the white regions represent?

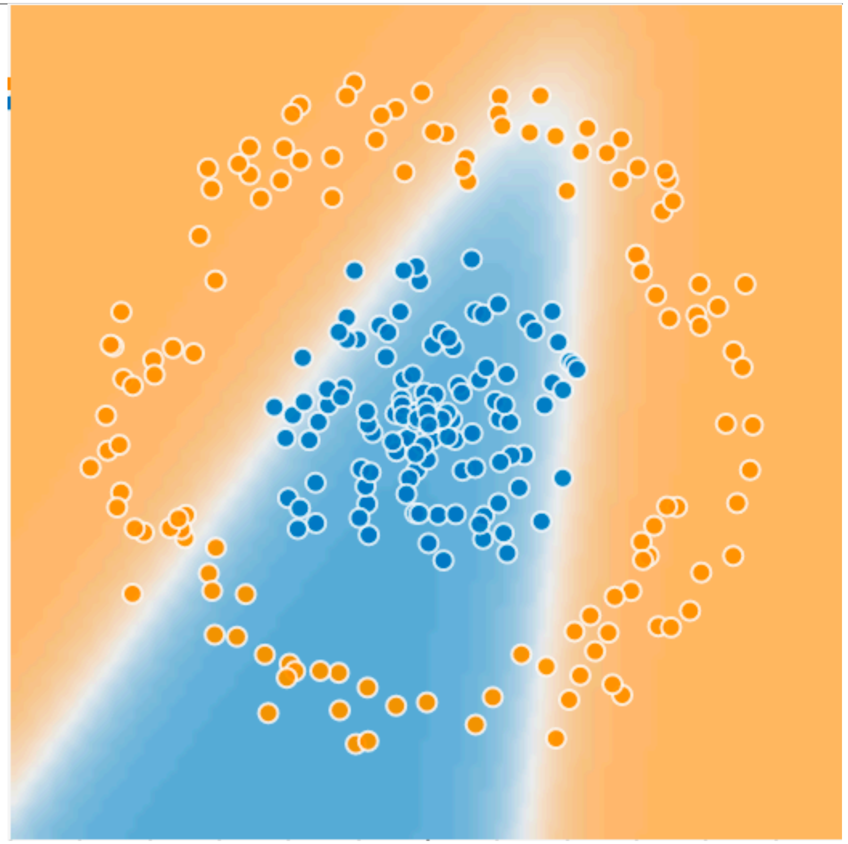


Two lines from two middle “hidden nodes” with sigmoid behavior

What might the weights look like for each of these nodes?

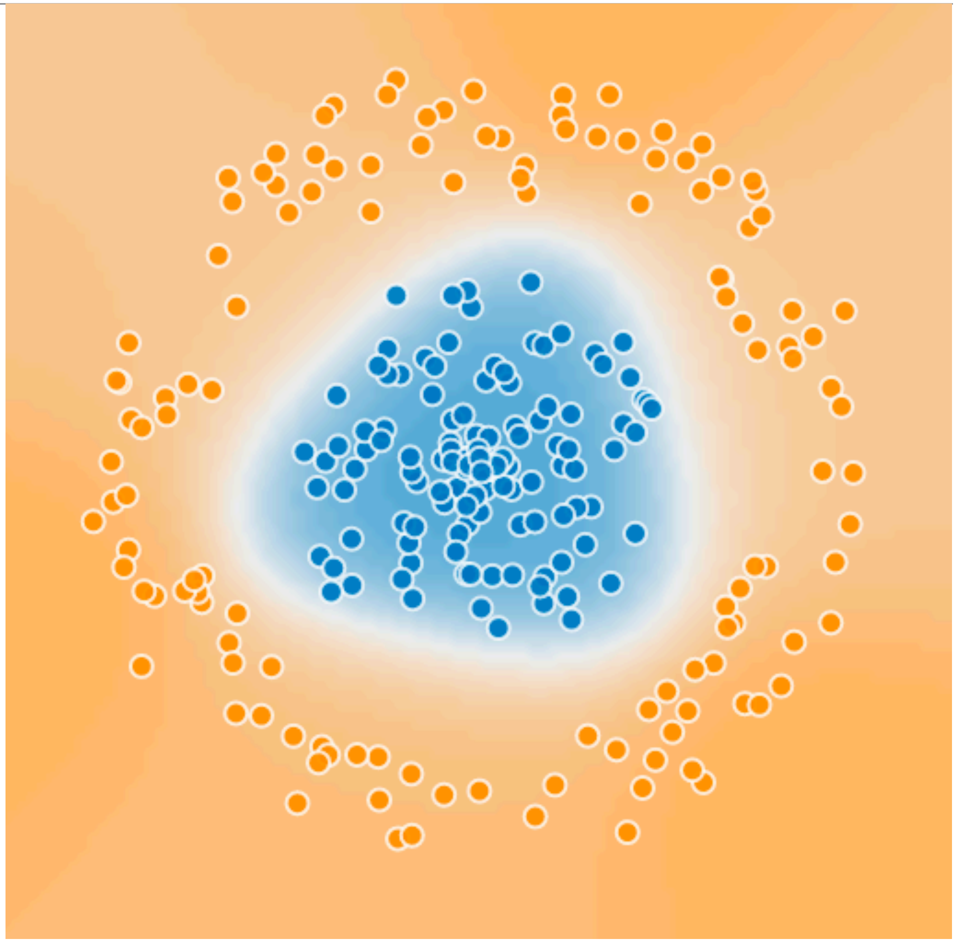
How to train?

Which would help more? Increasing the number of nodes in our hidden layer or increasing the number of hidden layers?



How to train?

With three internal nodes, we can now generate three linear models to separate the data.



Algorithm 8.4: Perceptron algorithm

Input: linearly separable data set $\mathbf{x}_i \in \mathbb{R}^D$, $y_i \in \{-1, +1\}$ for $i = 1 : N$;

Initialize $\boldsymbol{\theta}_0$;

$k \leftarrow 0$;

repeat

    $k \leftarrow k + 1$;

    $i \leftarrow k \bmod N$;

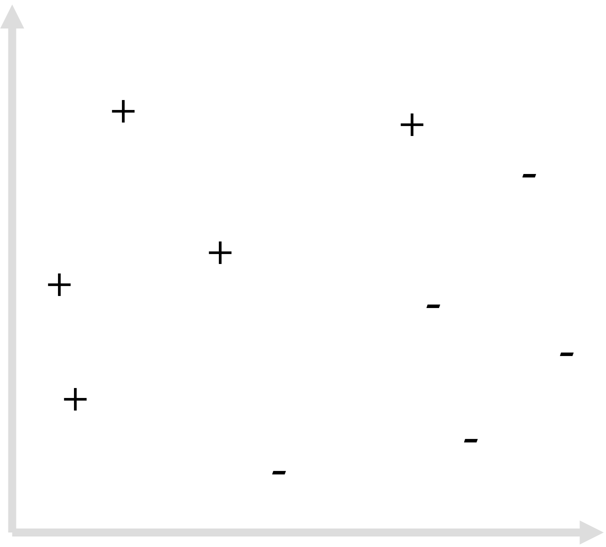
    if $\hat{y}_i \neq y_i$ **then**

        $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + y_i \mathbf{x}_i$

    else

        no-op

until *converged*;



What's wrong with this approach?

It's an easy way to separate linear data. But what's wrong?

What's wrong with this approach?

It's an easy way to separate linear data. But what's wrong?

- Data isn't always linearly separable
- Nodes of the neural network work in conjunction with each other
  - This approach would mean that all internal nodes convert to the same point!

What’s wrong with this approach?

It’s an easy way to separate linear data. But what’s wrong?

- Data isn’t always linearly separable
- Nodes of the neural network work in conjunction with each other
  - This approach would mean that all internal nodes convert to the same point!

How do we get around this?

What’s wrong with this approach?

It’s an easy way to separate linear data. But what’s wrong?

- Data isn’t always linearly separable
- Nodes of the neural network work in conjunction with each other
  - This approach would mean that all internal nodes convert to the same point!

How do we get around this?

- Random weights
- Feedback between nodes

How to Train

- Training to minimize sum-squared error

$$\ell(W) \;=\; \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial \ell(W)}{\partial w_k} \;=\; -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]\; x_k^j\; g'(w_0 + \sum_i w_i x_i^j)$$

Gradient descent for 1-hidden layer – Back-propagation: Computing $\frac{\partial \ell(W)}{\partial w_i^k}$

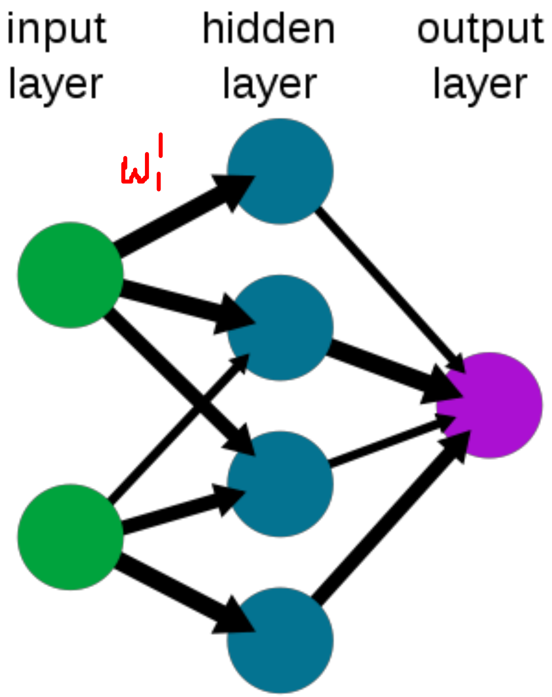
$$\ell(W) = \frac{1}{2}\sum_j [y^j - out(\mathbf{x}^j)]^2$$

$$out(\mathbf{x}) = g\left(\sum_{k'} w_{k'} g(\sum_{i'} w_{i'}^{k'} x_{i'})\right)$$

$$\frac{\partial \ell(W)}{\partial w_i^k} = \sum_{j=1}^{m} -[y - out(\mathbf{x}^j)]\frac{\partial out(\mathbf{x}^j)}{\partial w_i^k}$$

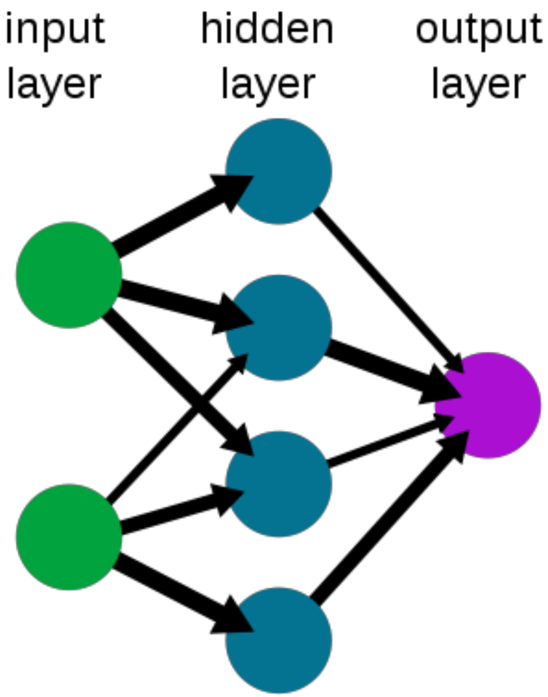
Dropped w_0 to make derivation simpler

A simple neural network



Back-propagation Algorithm

A simple neural network



$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$

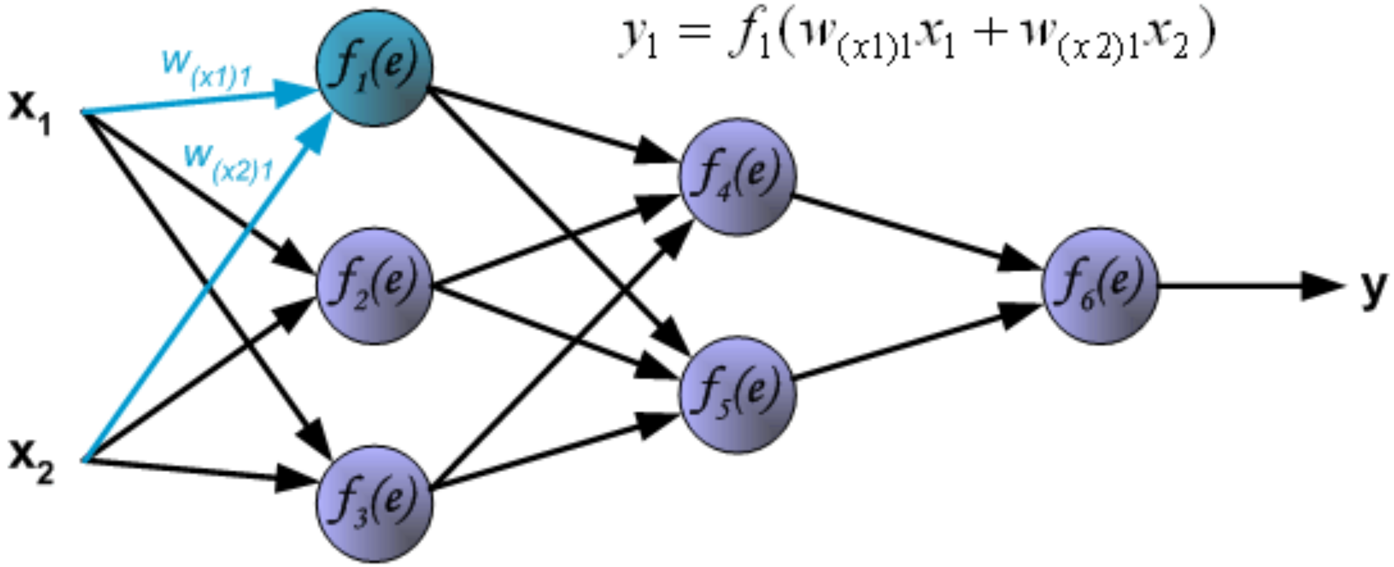
Sigmoid function: g(z) = $\frac{1}{1 + exp(-z)}$

(Rummelhart et al. 1986)

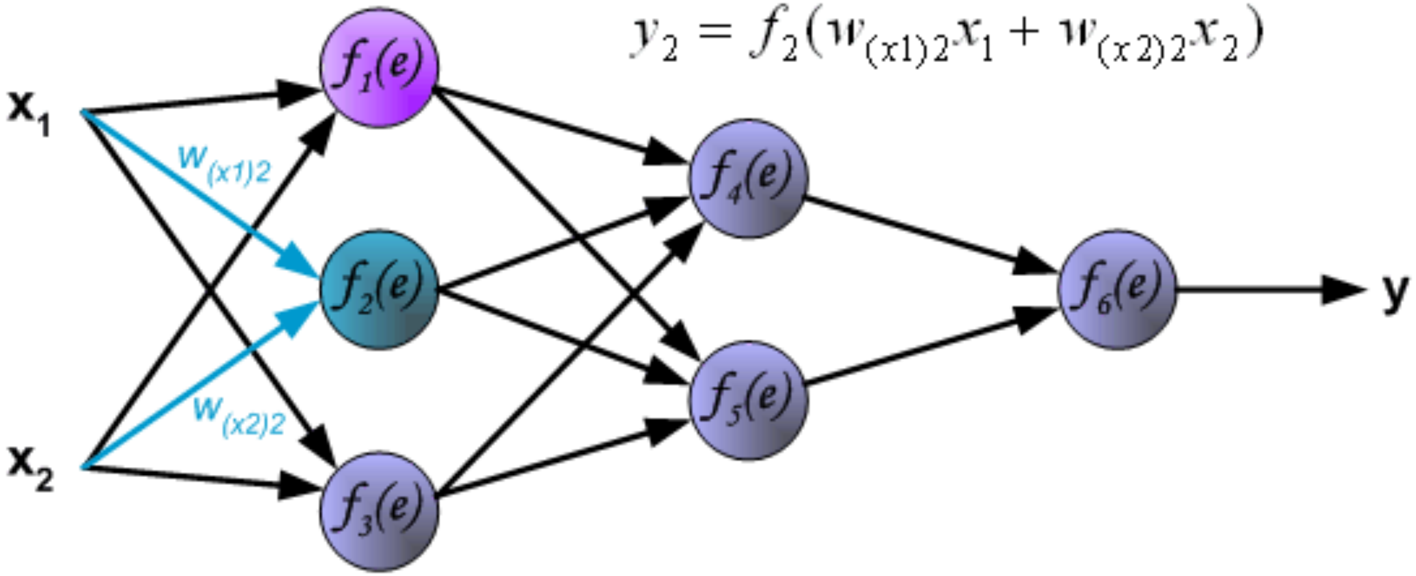
Non-convex function of '**w**'s

Back-prop: find local optima

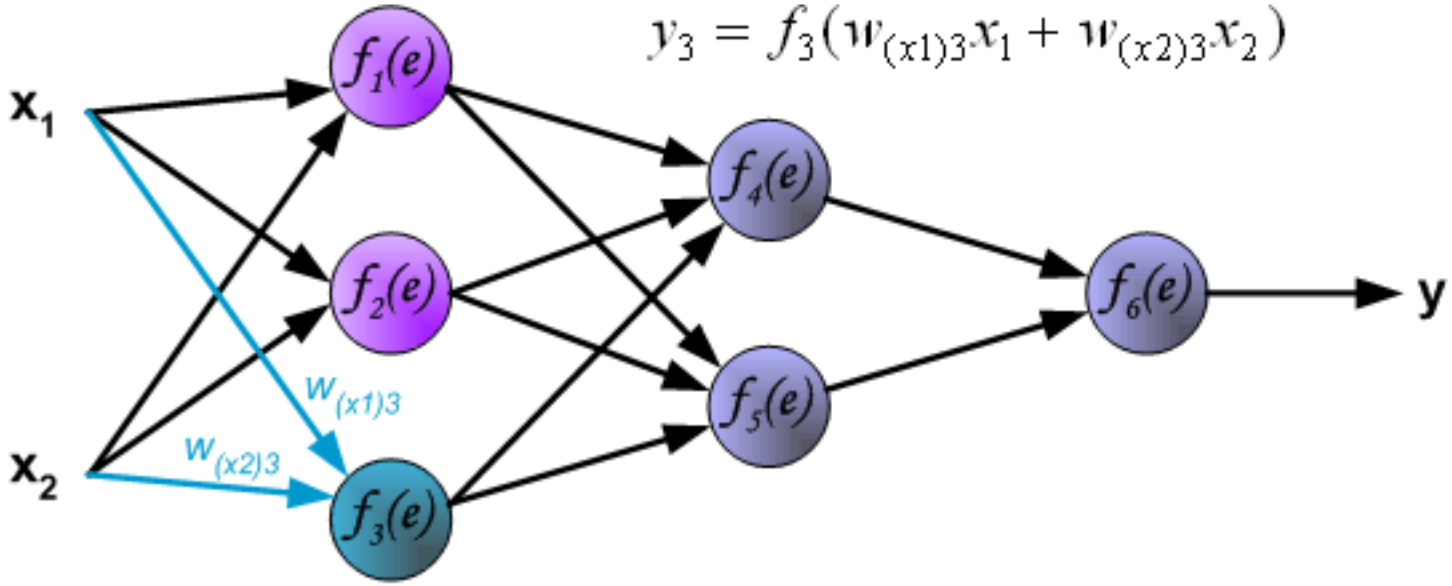
Learning Algorithm: Backpropagation



Learning Algorithm: Backpropagation

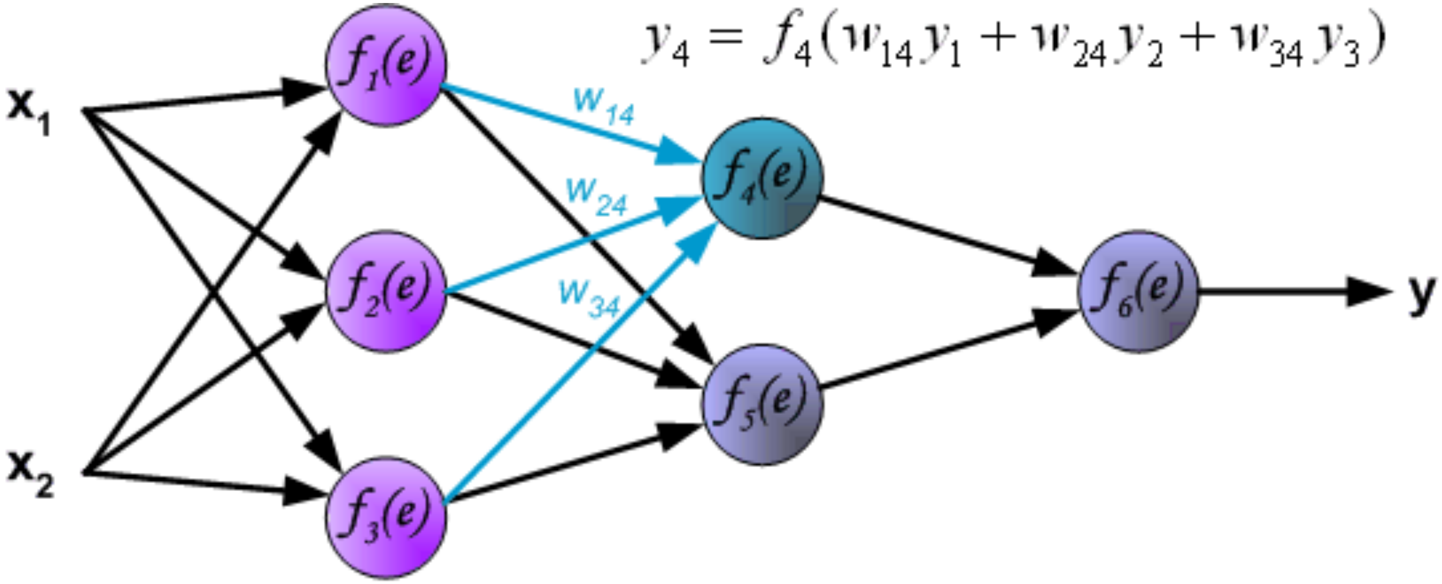


Learning Algorithm: Backpropagation

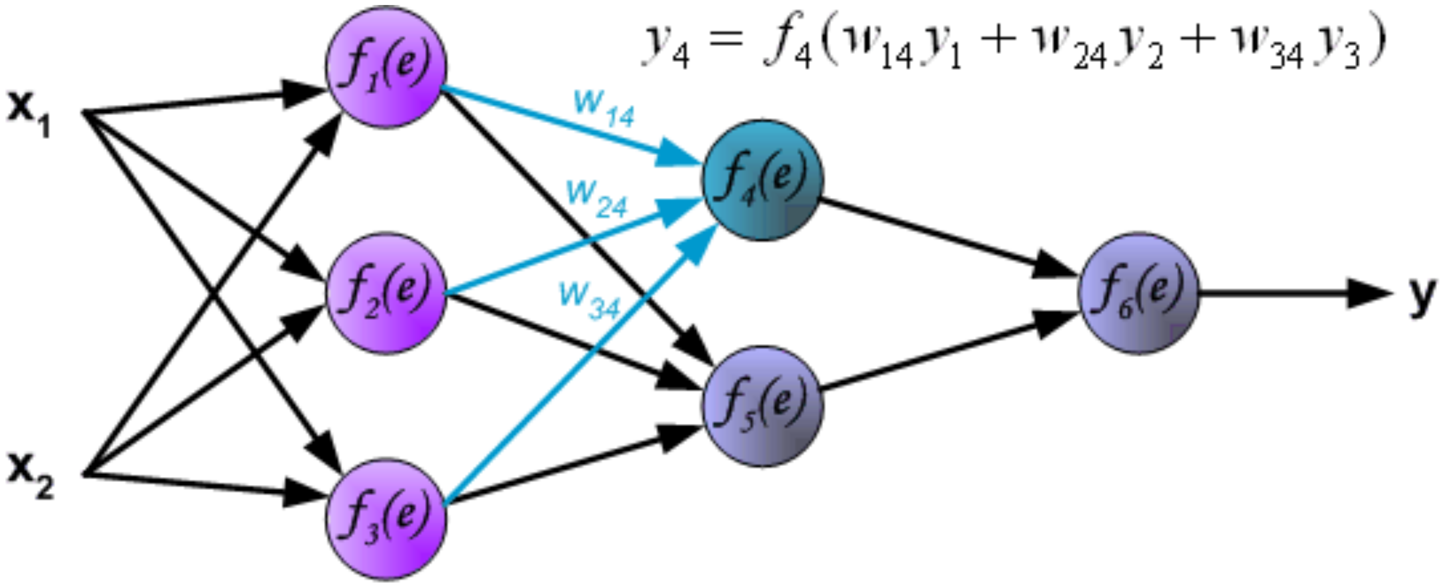


Learning Algorithm: Backpropagation

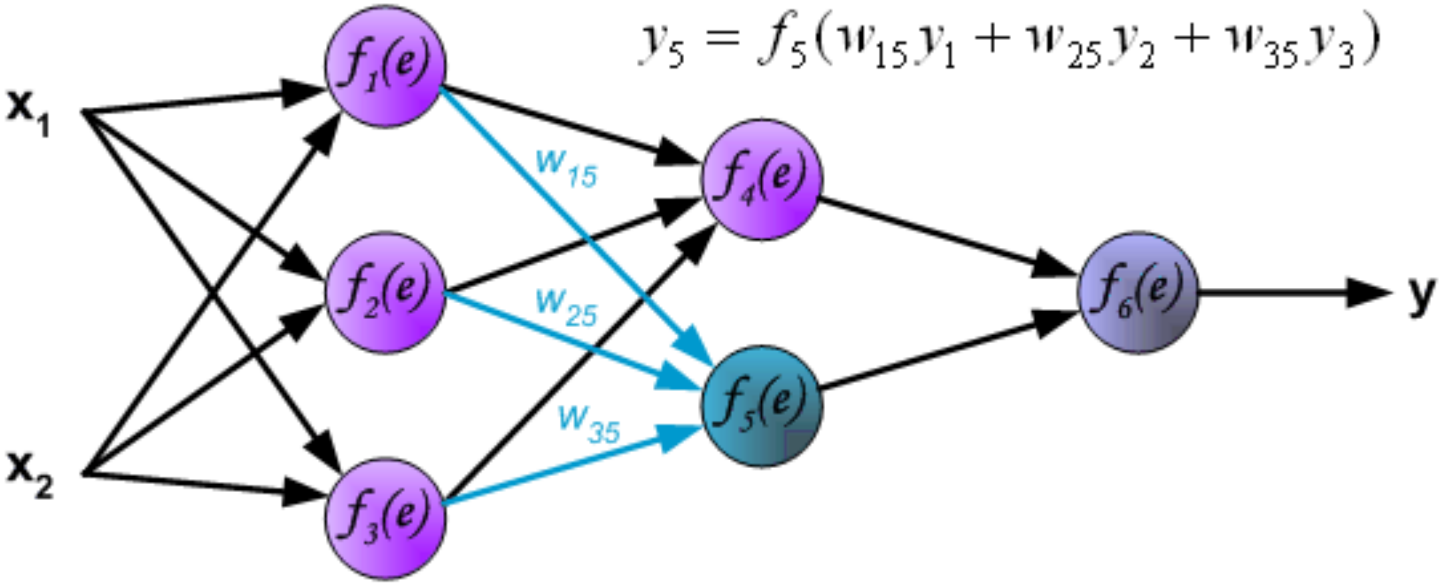
Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron m and input of neuron n in the next layer.



Learning Algorithm: Backpropagation

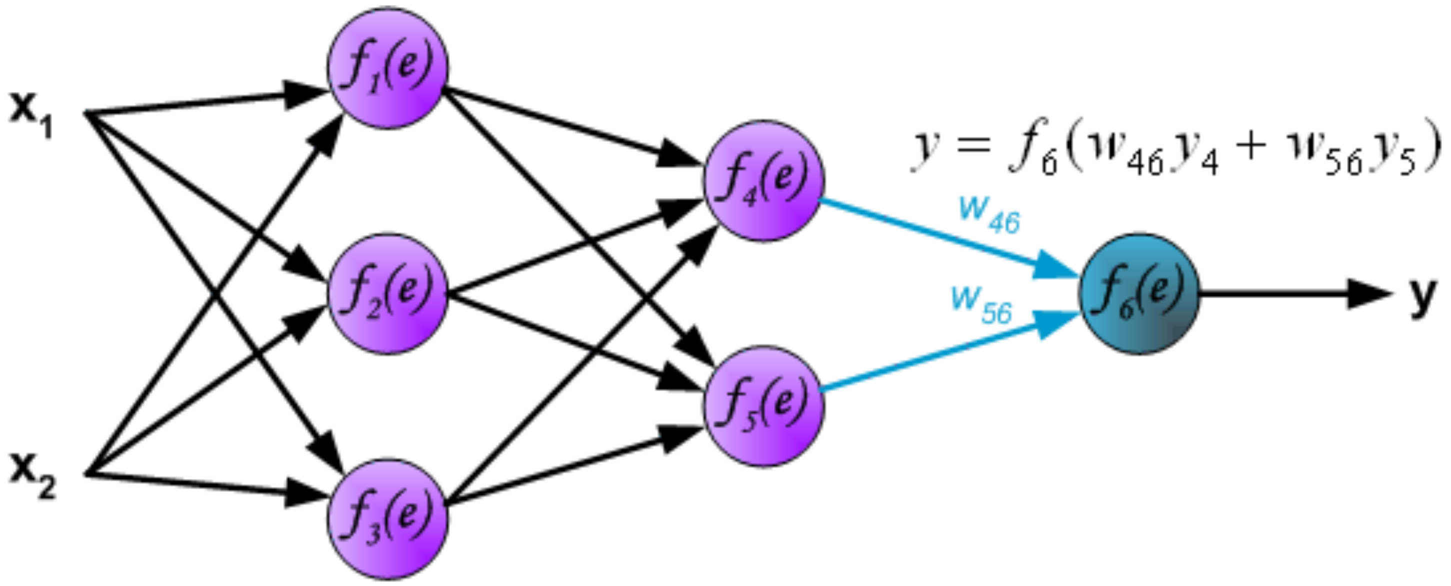


Learning Algorithm: Backpropagation



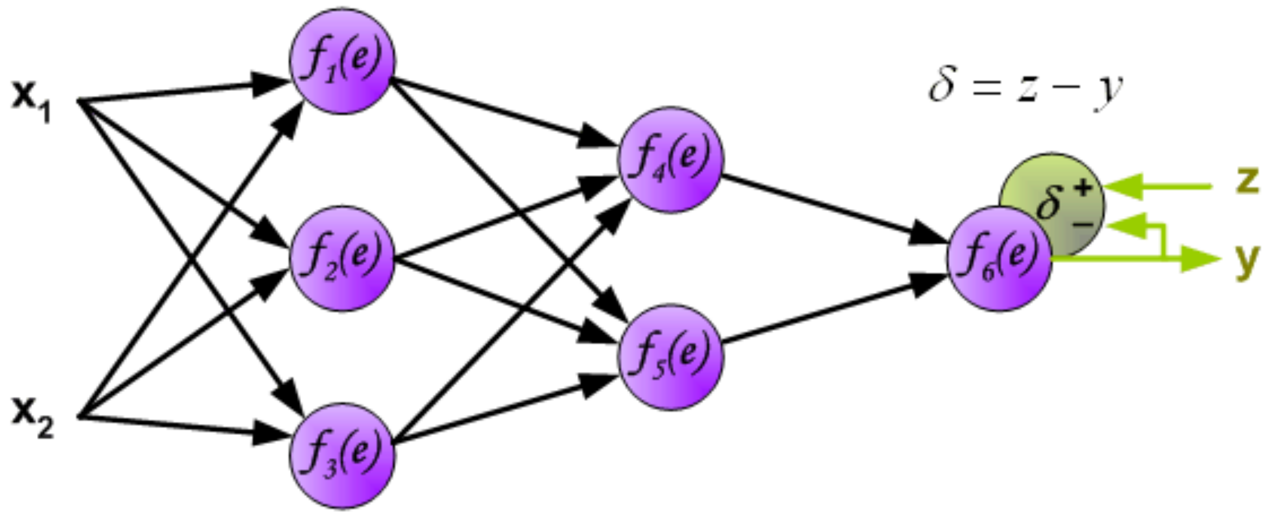
Learning Algorithm: Backpropagation

Propagation of signals through the output layer.



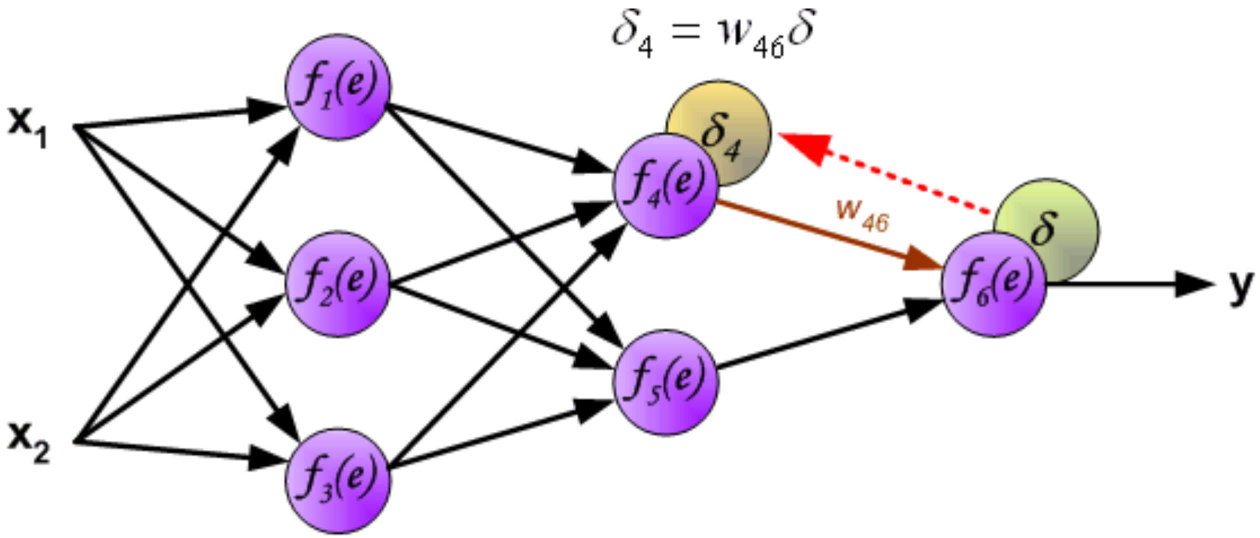
Learning Algorithm: Backpropagation

In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron



Learning Algorithm: Backpropagation

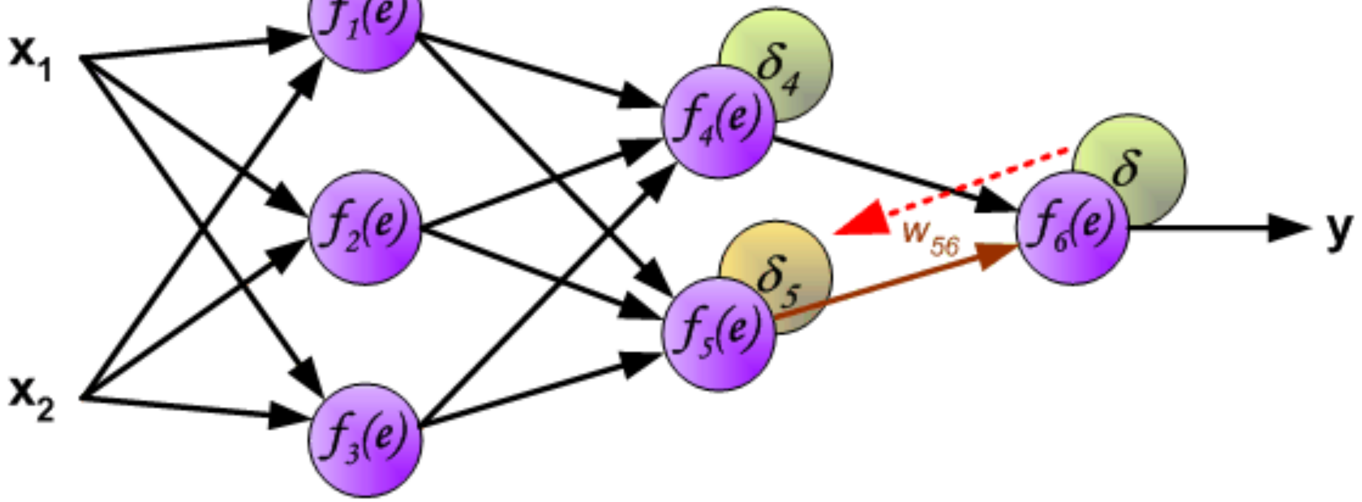
The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.



Learning Algorithm: Backpropagation

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

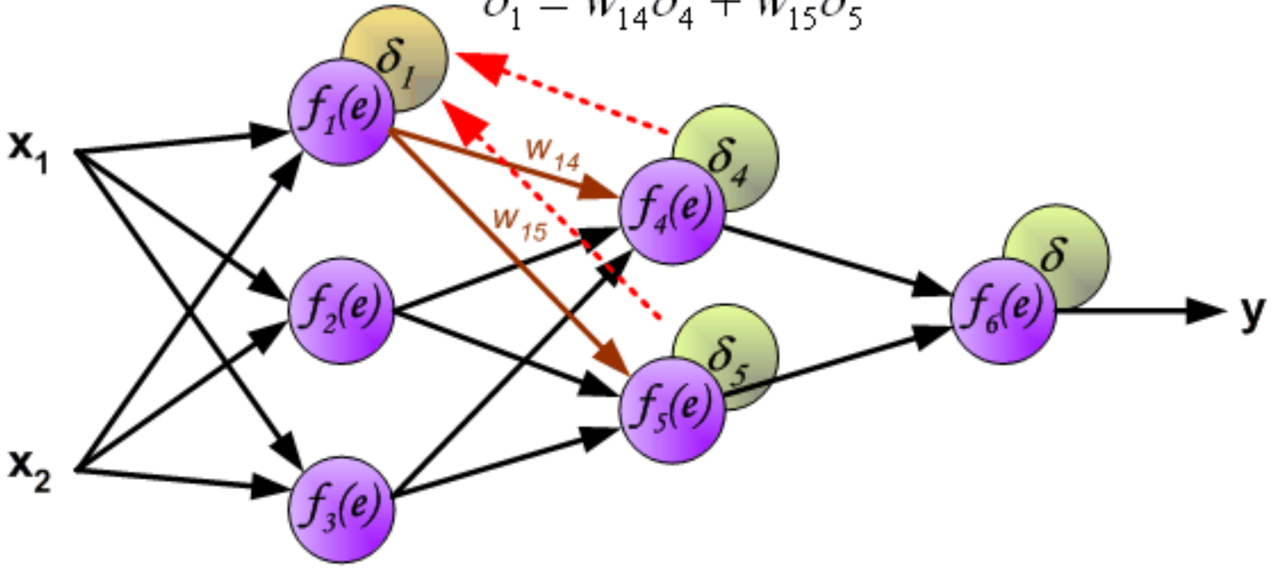
$$\delta_5 = w_{56}\delta$$



Learning Algorithm: Backpropagation

The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:

$$\delta_1 = w_{14}\delta_4 + w_{15}\delta_5$$

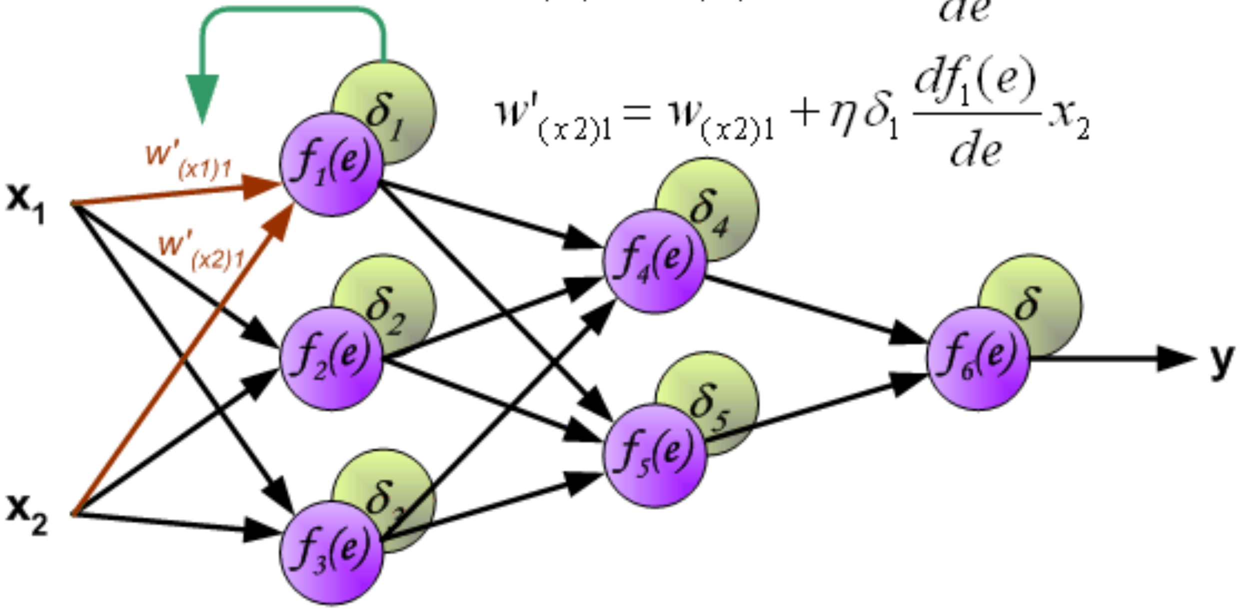


Learning Algorithm: Backpropagation

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified)

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$



Convergence of backprop

Perceptron leads to convex optimization

- Gradient descent reaches **global minima**

Multilayer neural nets **not convex**

- Gradient descent could get stuck in local minima
- Hard to set learning rate
- Selecting number of hidden units and layers = fuzzy process
- Nonetheless, neural nets are one of the most used ML approaches