CS 412

$$\frac{dE}{dw_{ij}}$$

MAR 3RD – NEURAL NETWORKS

HTF – CHAPTER 11

A Note about Python Classes

```
class NeuralNetwork:

    def _init_(self,x=[[]],y=[],numLayers=2,numNodes=2,eta=0.001,maxIter=10000):
```

To create a new NeuralNetwork object call NeuralNetwork(x,y)

This will create an object for which all other parameters are “defaults”.

If a user does not specify the numLayers, then it will be 2

This prevents the coder from needing to build multiple constructors

NeuralNetwork(x, y,
eta = 1)

X = datapoints
y = values

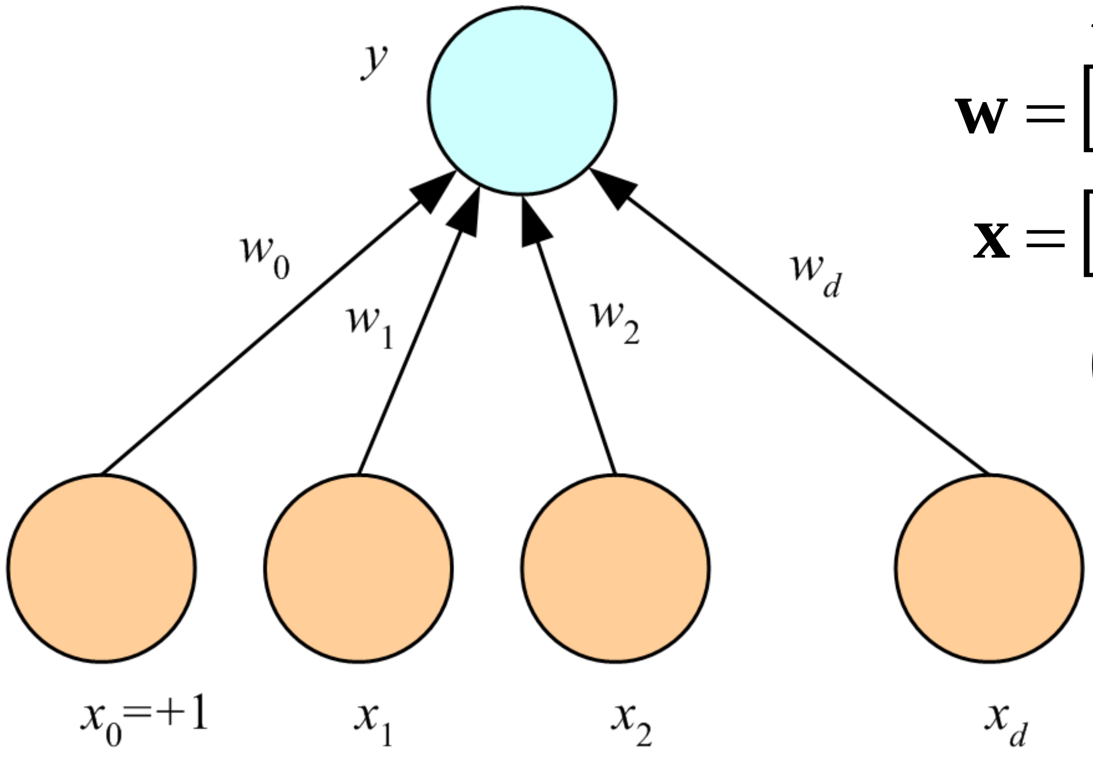
A Note about Python Classes

```
class NeuralNetwork:
        DON'T PUT VARIABLES HERE --- they'll be static for all objects
    attribute = input

    def _init_(self,x=[[]],y=[],numLayers=2,numNodes=2,eta=0.001,maxIter=10000):

    def yourfunction(self,input="default"):
        self.attribute=input // this is how you give a specific object an attribute
```

In the test code, I will only create one instance of the NeuralNetwork, so static variables won't impact the solution, but this is a property of python that you should know

You will, however, likely need to add your own helper functions and attributes, and this will be helpful

Perceptron



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

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

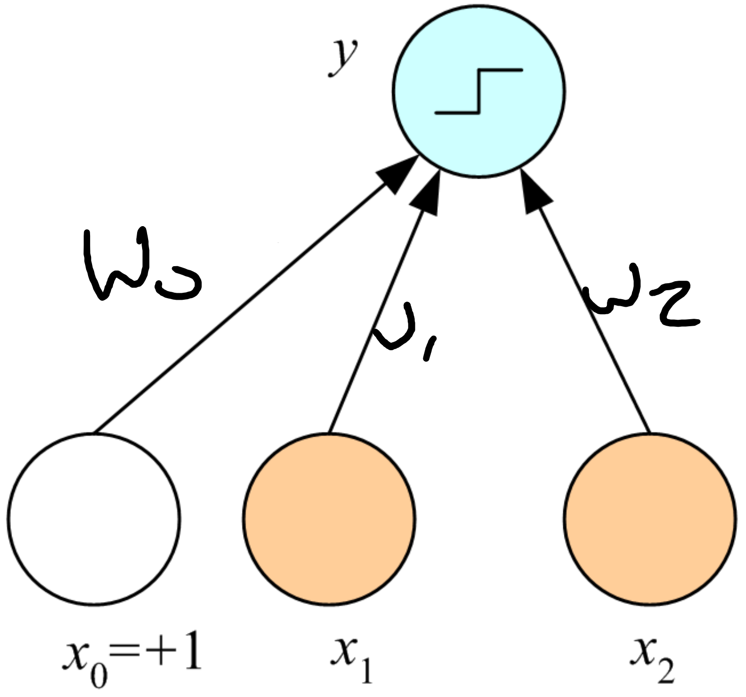
$$\mathbf{x} = \left[1, x_1, \ldots, x_d\right]^T$$

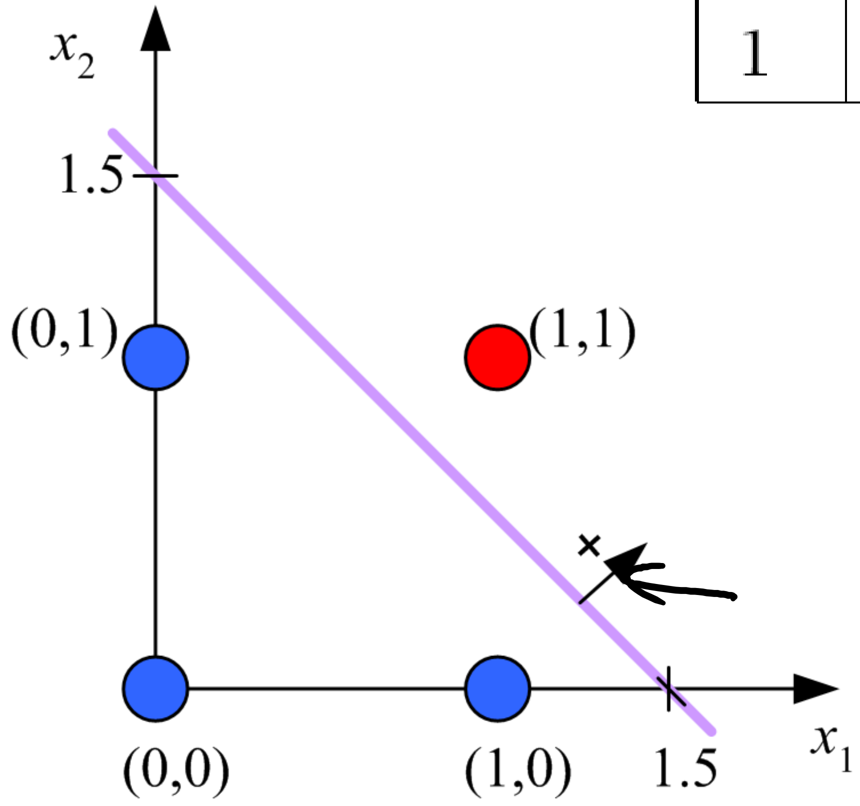
(Rosenblatt, 1962)

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)

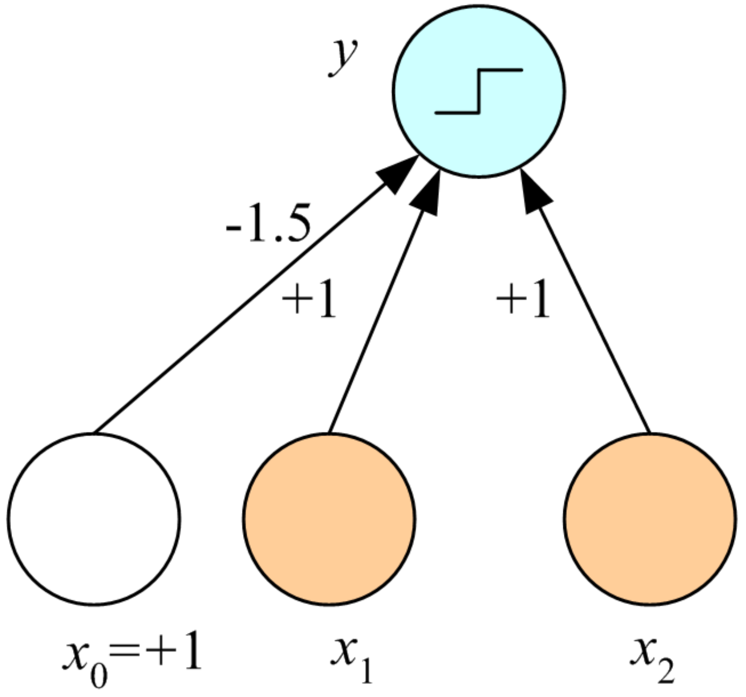


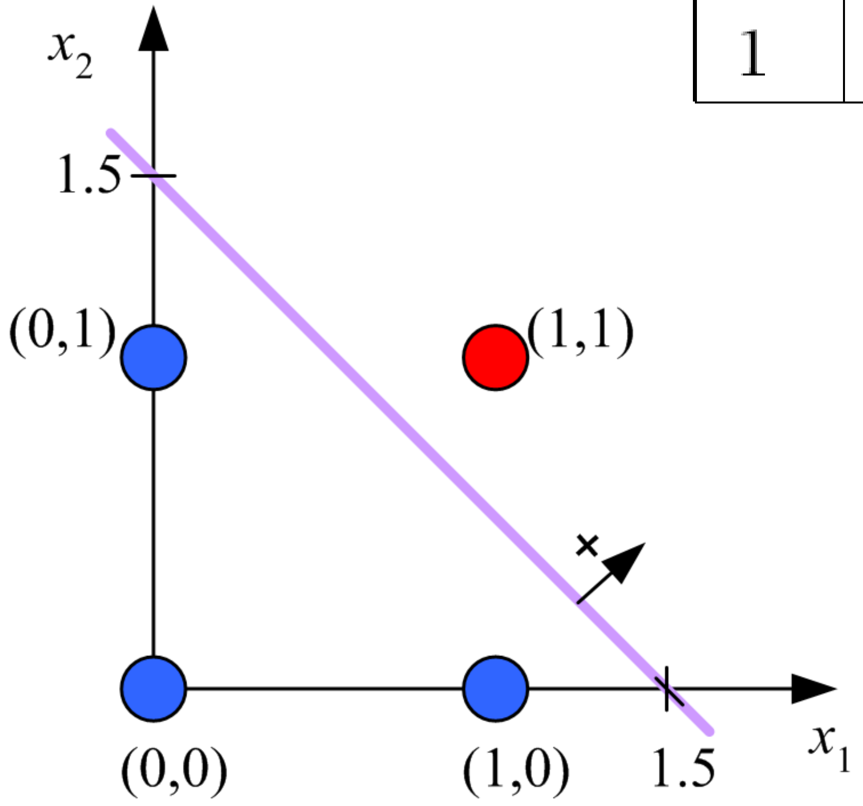


Learning Boolean AND

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)



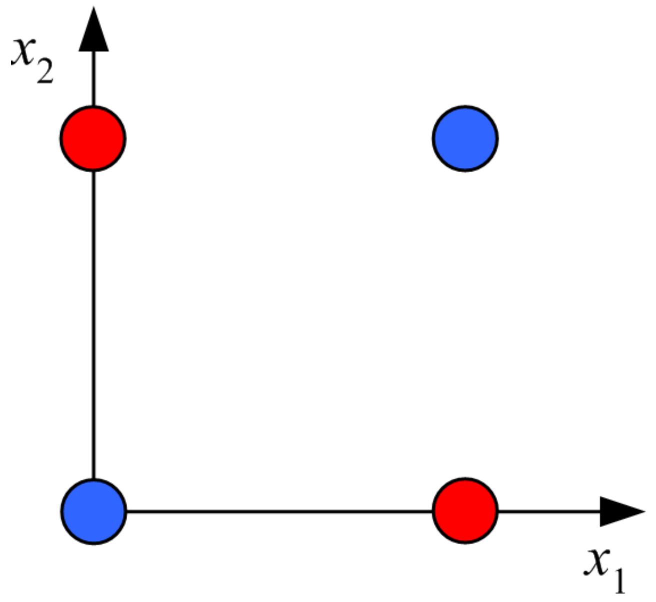


XOR

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

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?

- 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?

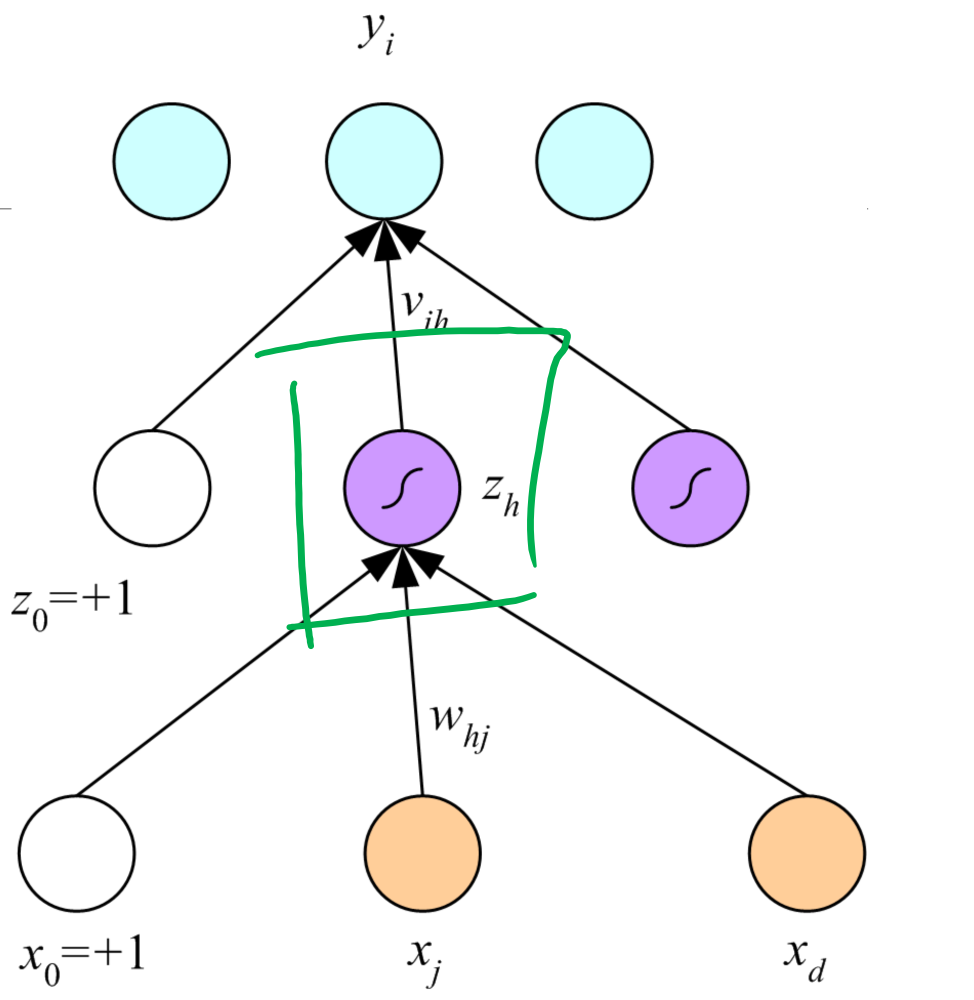
- It can't model non-linear data

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

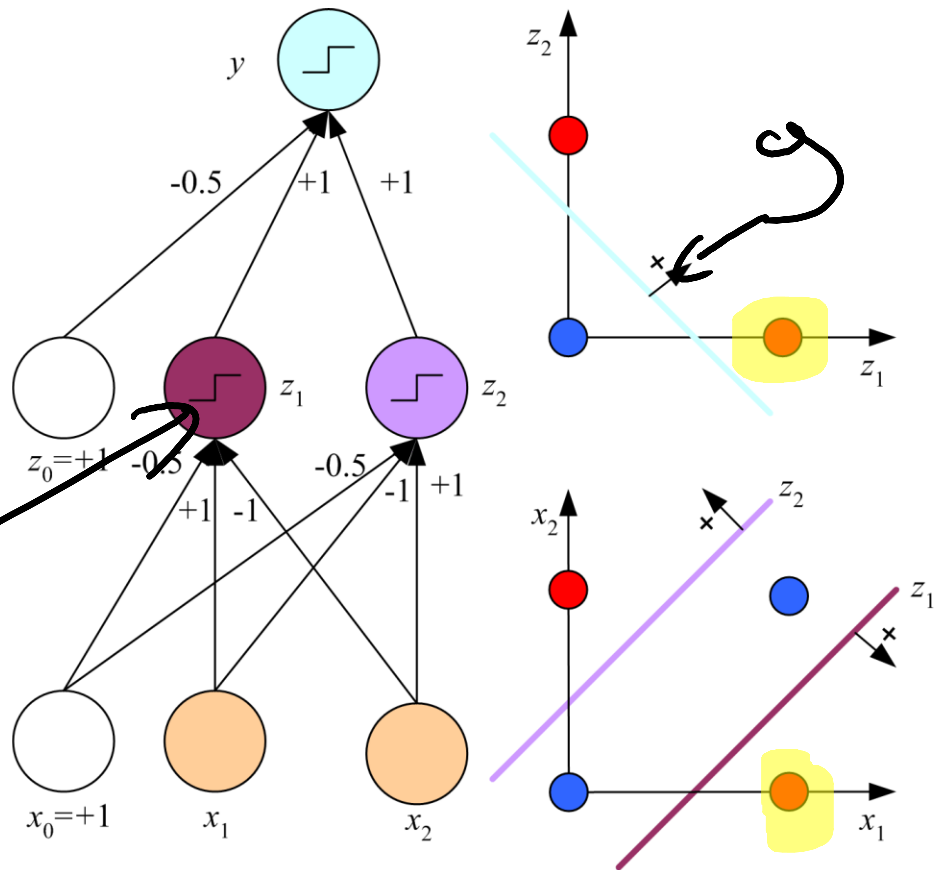
- At each level we have a **regression** model defined by the activation function and **always** a constant w_0



Solves a regression problem

Perceptron

How do we use this to solve the XOR problem?

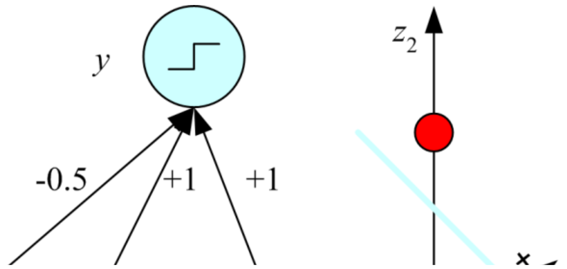


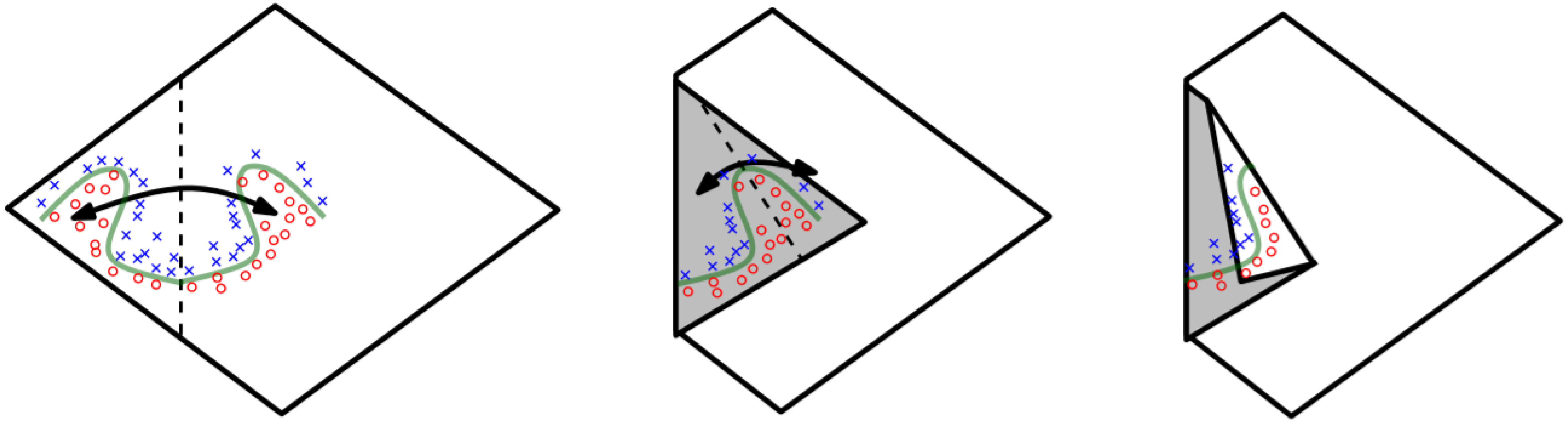
Step activation function

Perceptron

How do we use this to solve the XOR problem?

- What is happening here?





How to Train

how many layers

- Training to minimize sum-squared error

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

$g(x) =$ activation function

- What type of neural network do we think this is?
  - How do we minimize it?

$$\frac{\partial L}{\partial w_k}$$

How to Train

$$\frac{\partial L}{\partial w}$$

- Training to minimize sum-squared error

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

- What type of neural network do we think this is?
  - How do we minimize it?
    - Stochastic gradient descent --- what do we need?

$$\frac{\partial \ell(W)}{\partial w_k} =$$

$$\alpha \Delta w_k$$

How to Train

- Training to minimize sum-squared error

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

- What type of neural network do we think this is?
  - How do we minimize it?
    - Stochastic gradient descent --- what do we need?

$$\frac{\partial \ell(W)}{\partial w_k} \;=\; \underbrace{-\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]}_{\text{CR1}} \; \overbrace{x_k^j}^{\text{Chain Rule application 3}} \; \underbrace{g'(w_0 + \sum_i w_i x_i^j)}_{\text{CR2}}$$

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

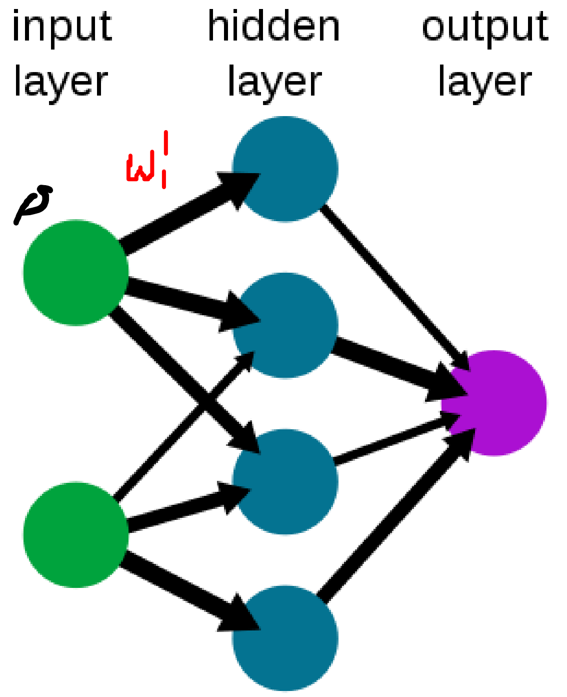
$$\ell(W) = \frac{1}{2}\sum_j [y^j - out(\mathbf{x}^j)]^2$$

$$out(\mathbf{x}) = g\left(\sum_{k'} w_{k'} \underbrace{g(\sum_{i'} w_{i'}^{k'} x_{i'})}_{\text{outputs of previous layer}}\right)$$

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

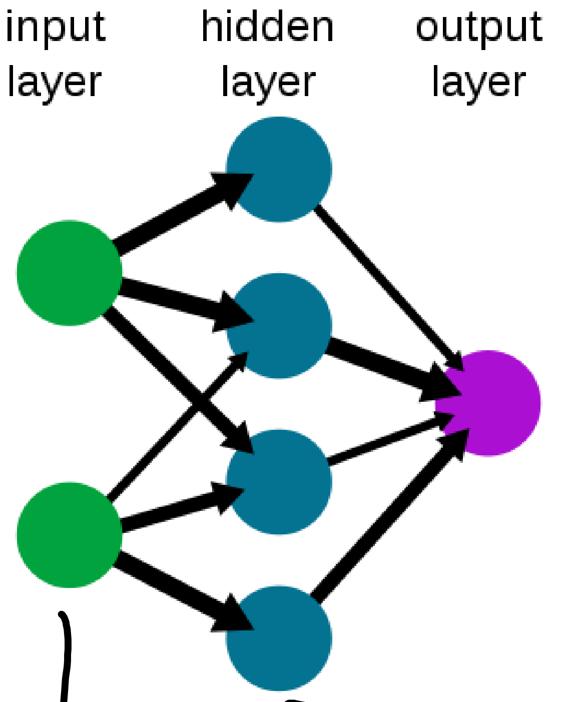
Dropped w_0 to make derivation simpler

A simple neural network



Back-propagation Algorithm

A simple neural network



i features

k hidden layers

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

input vector

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

$g'(z) = z(1-z)$

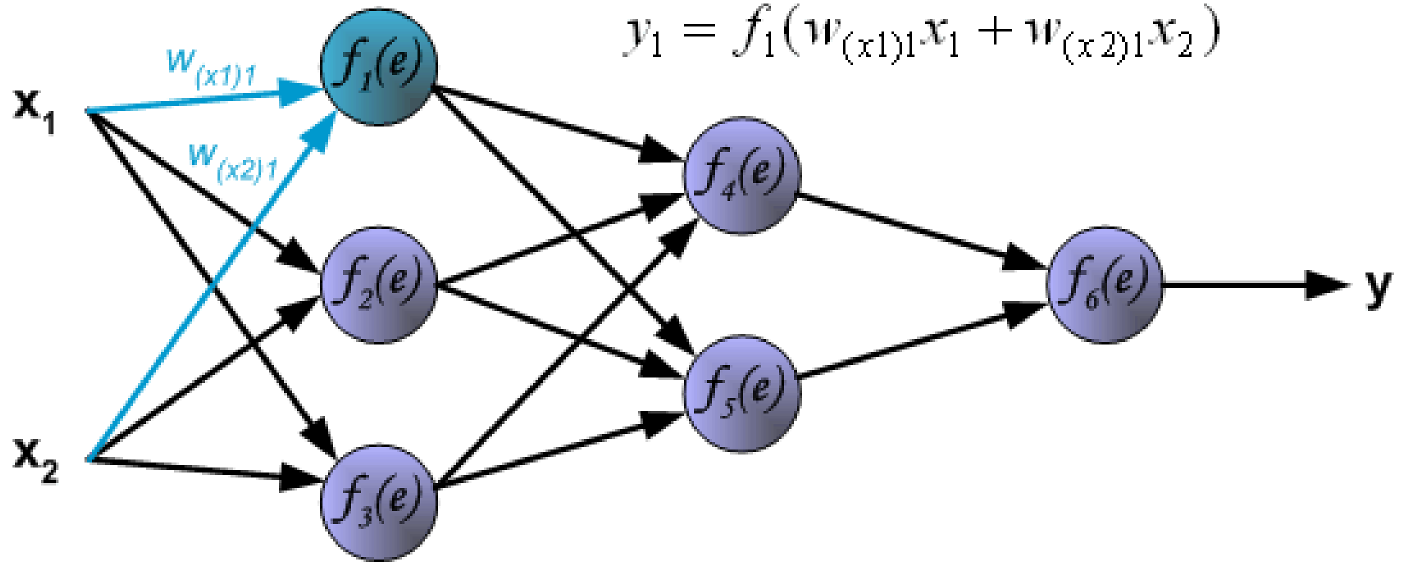
(Rummelhart et al. 1986)

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

Back-prop: find local optima

2 layers

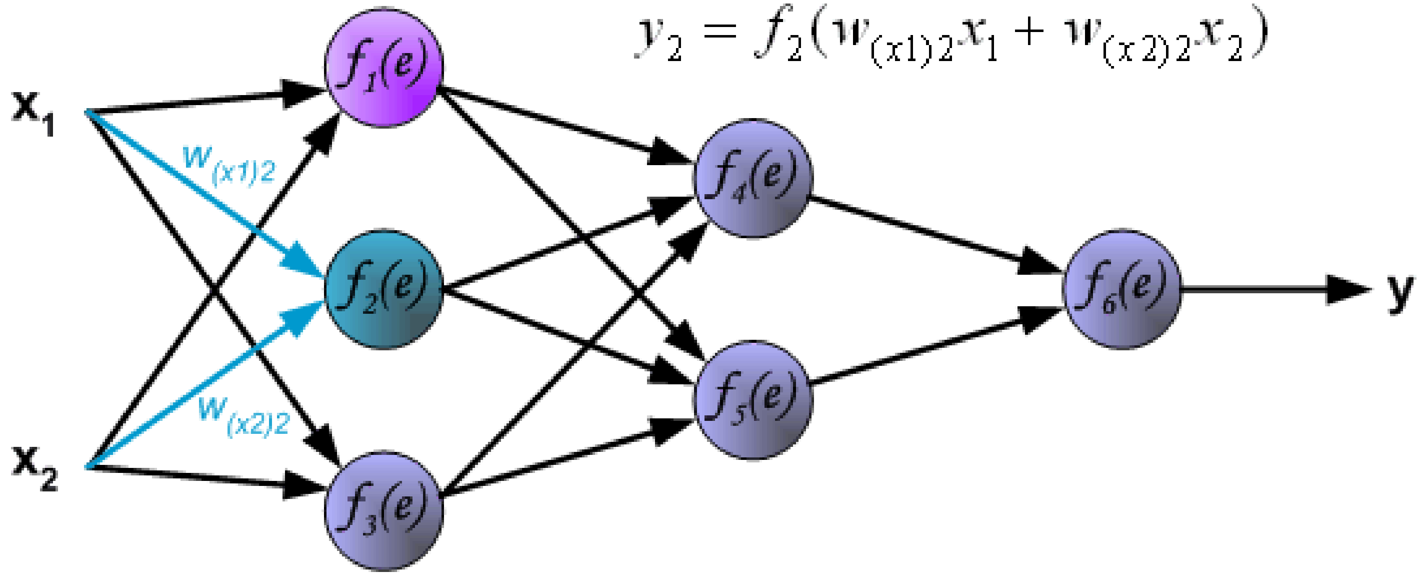
Learning Algorithm: Backpropagation



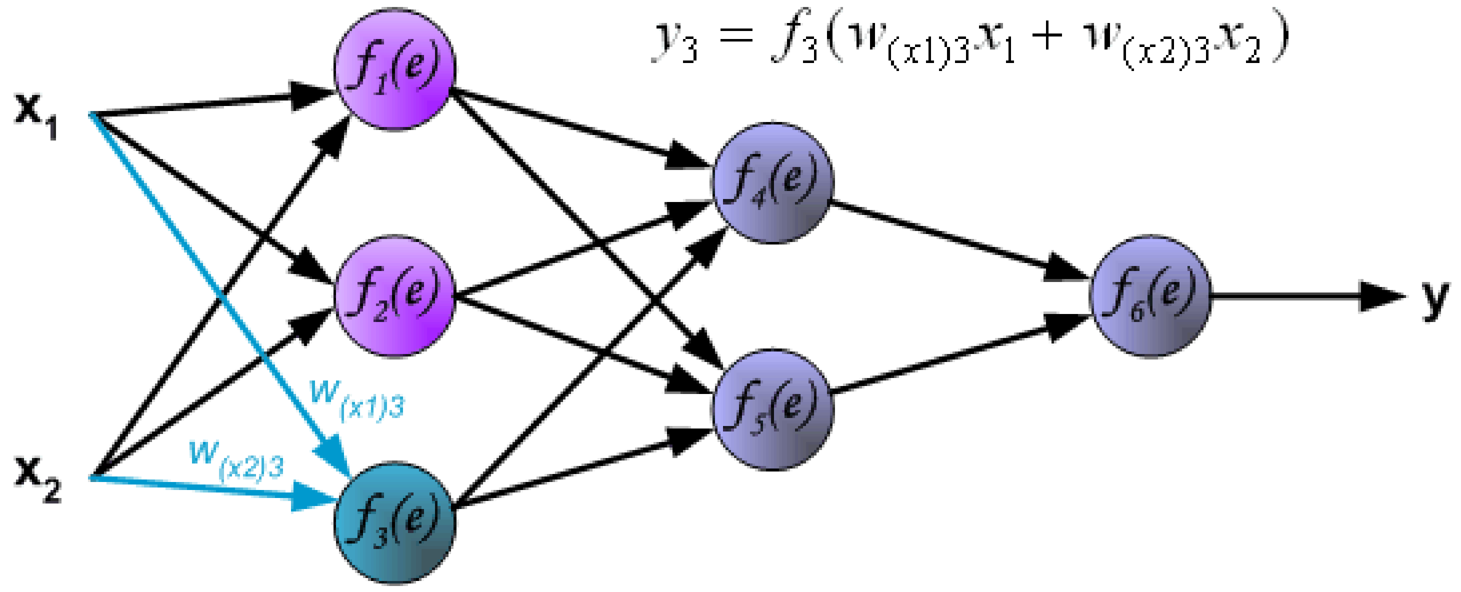
Step 1:
feed forward w/ point or set of points

Learning Algorithm: Backpropagation

$$y_2 = f_2(w_{(x1)2}x_1 + w_{(x2)2}x_2)$$



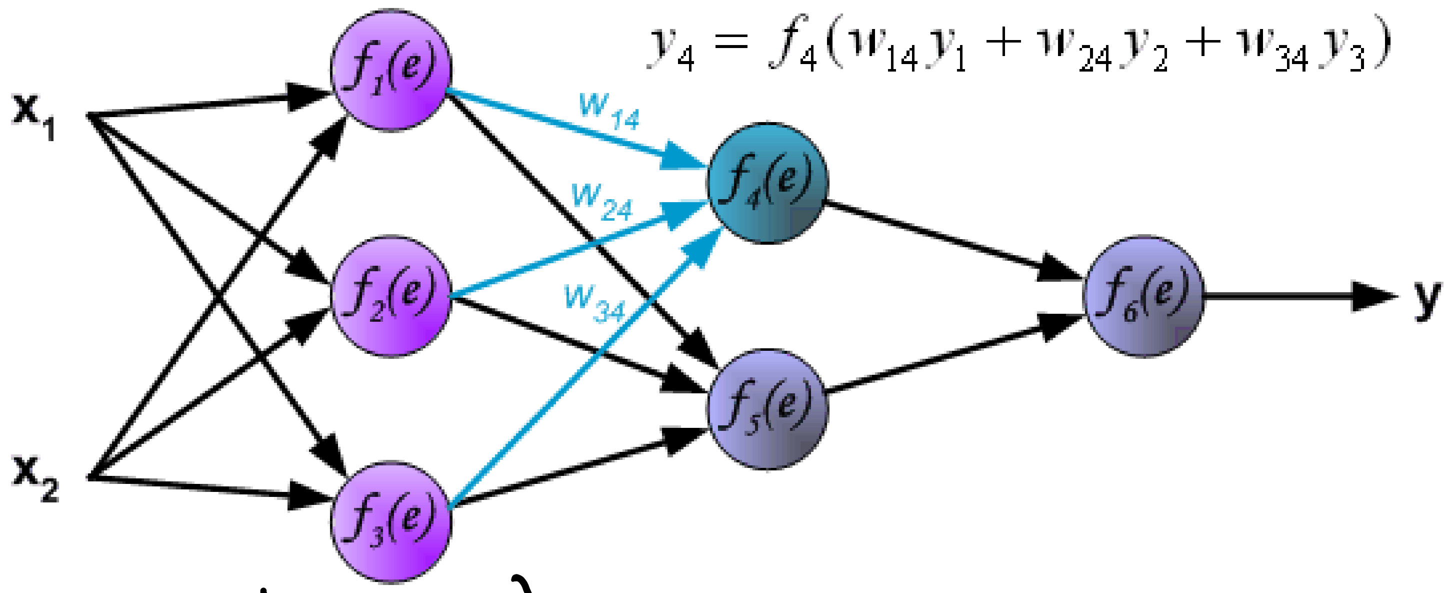
Learning Algorithm: Backpropagation



Learning Algorithm: Backpropagation

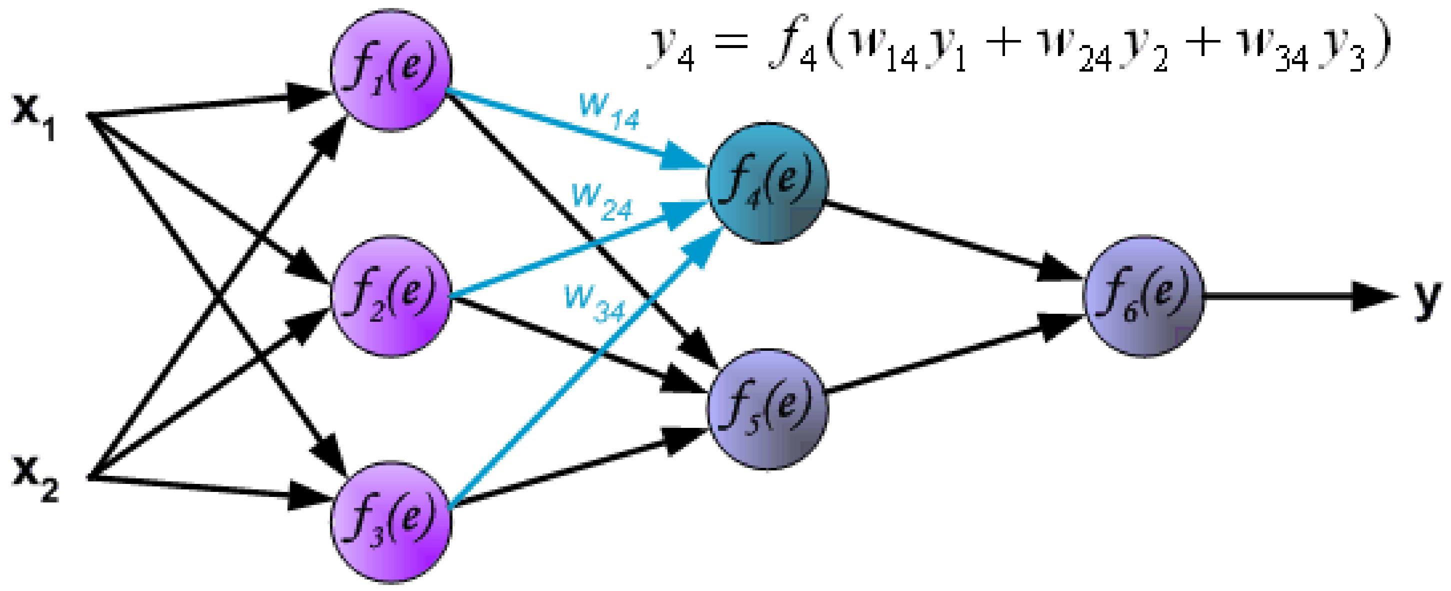
Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron *m* and input of neuron *n* in the next layer.

$$y_4 = f_4(w_{14}y_1 + w_{24}y_2 + w_{34}y_3)$$

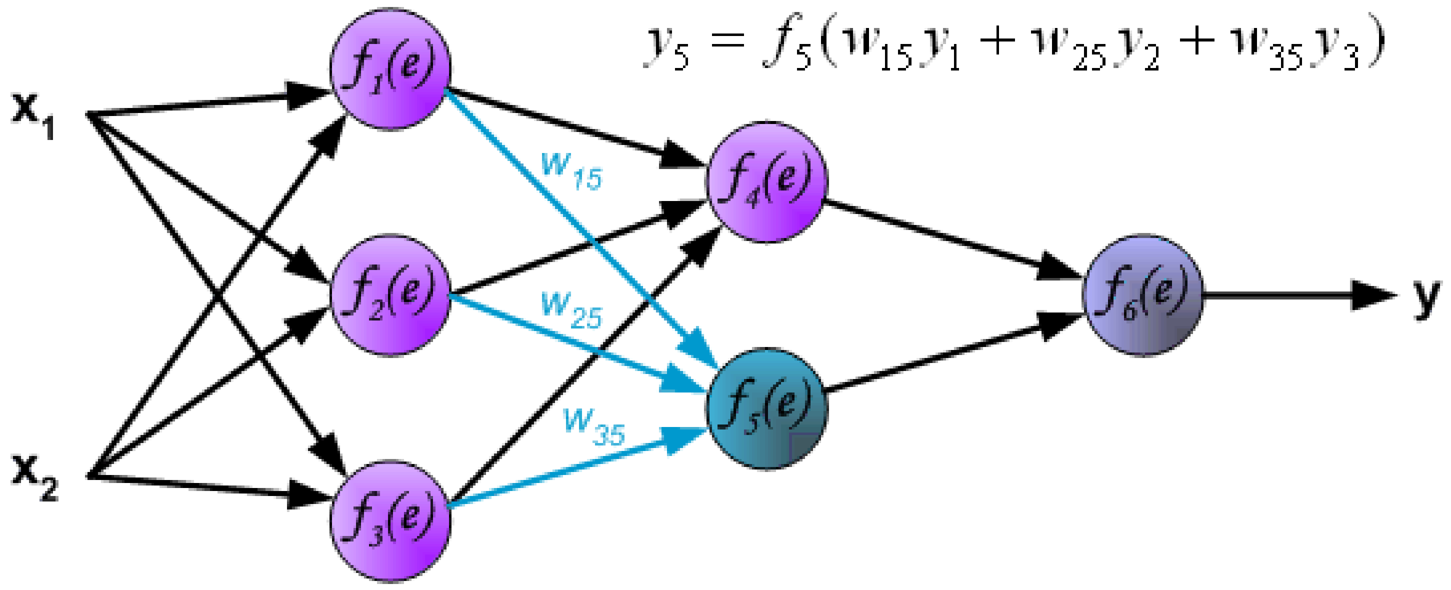


$$\underbrace{\qquad\qquad}_{\text{feed forward}}$$

Learning Algorithm: Backpropagation

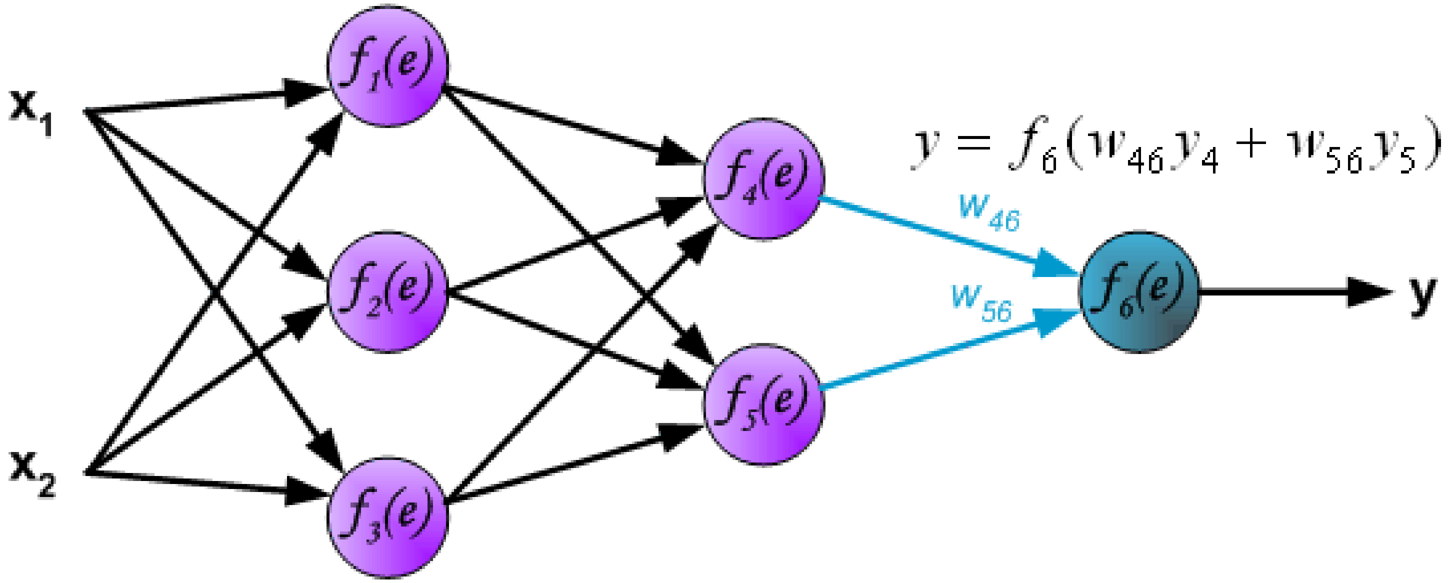


Learning Algorithm: Backpropagation



Learning Algorithm: Backpropagation

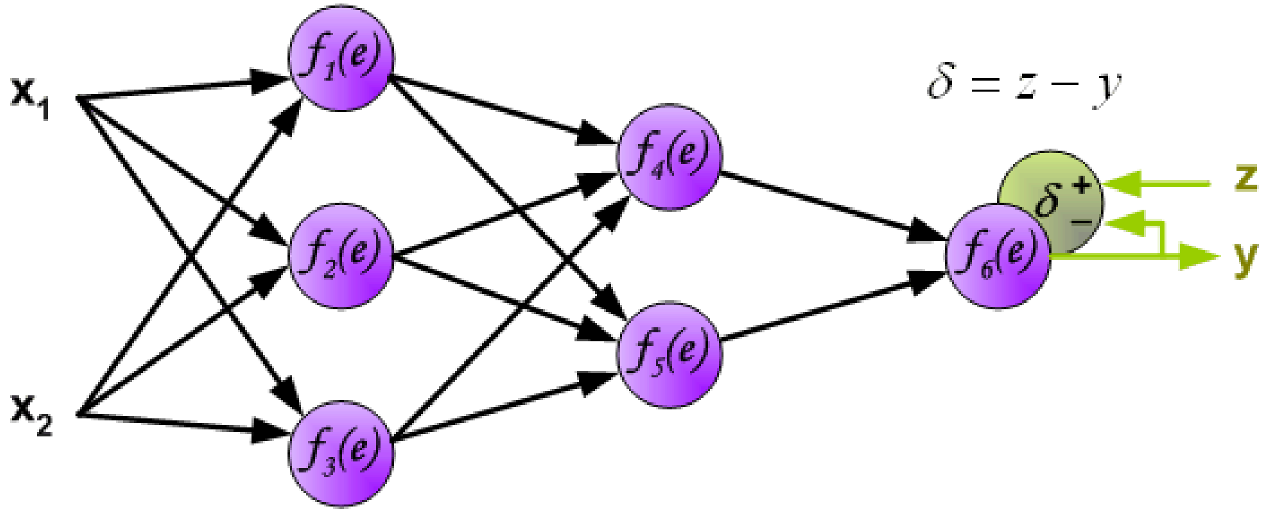
Propagation of signals through the output layer.



Learning Algorithm: Backpropagation

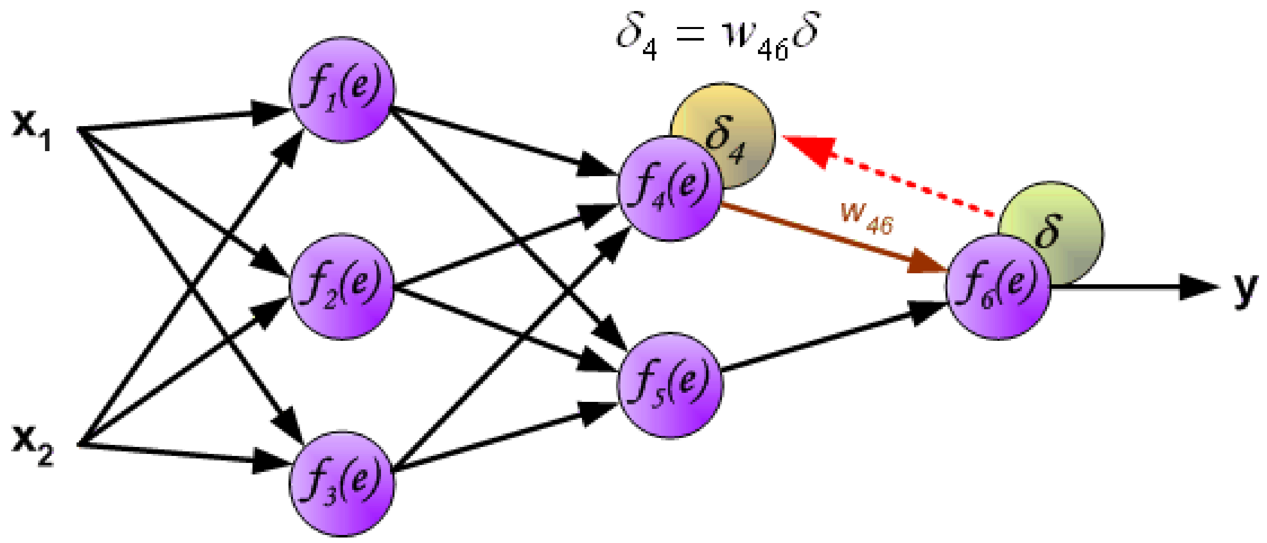
In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron

Cross-entropy error



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.

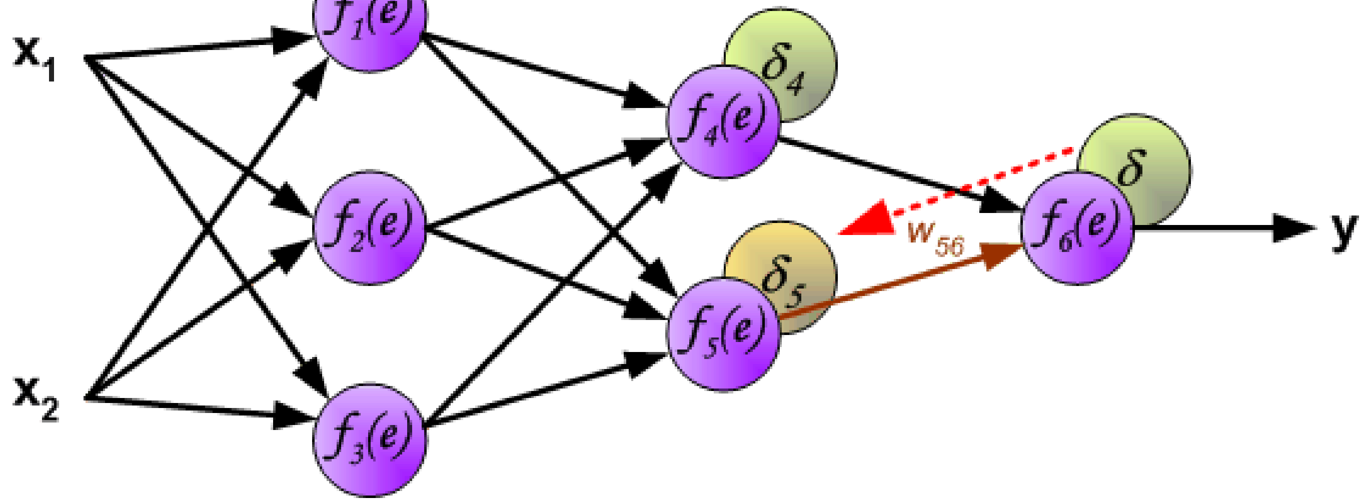


Learning Algorithm: Backpropagation

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.

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

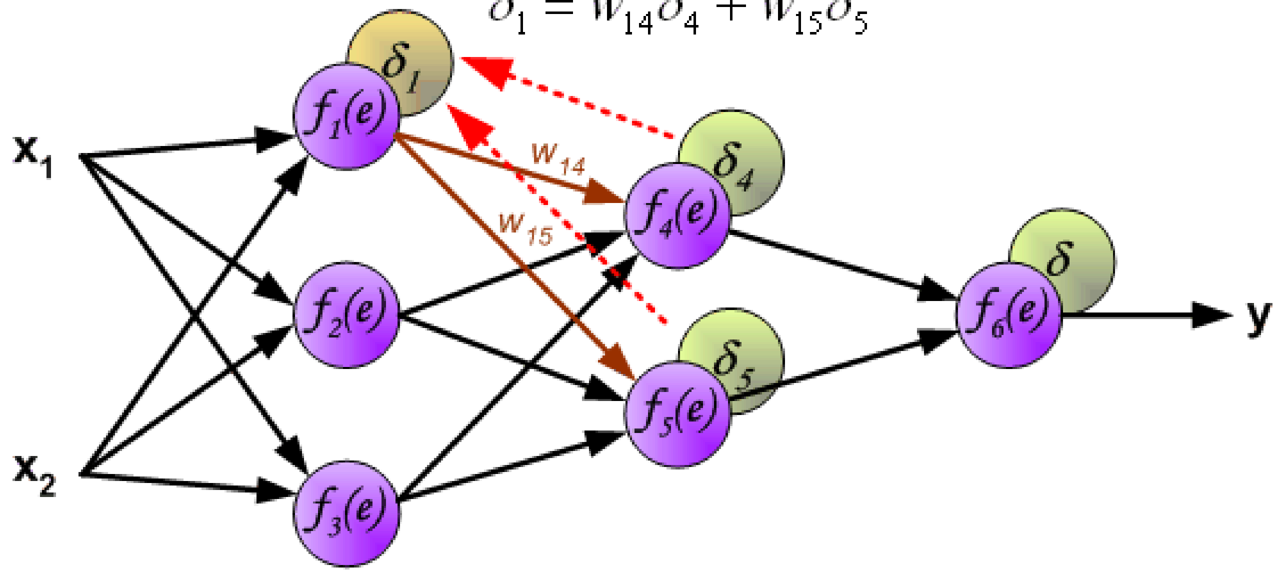
$$\delta_5 = w_{56}\delta$$



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:

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

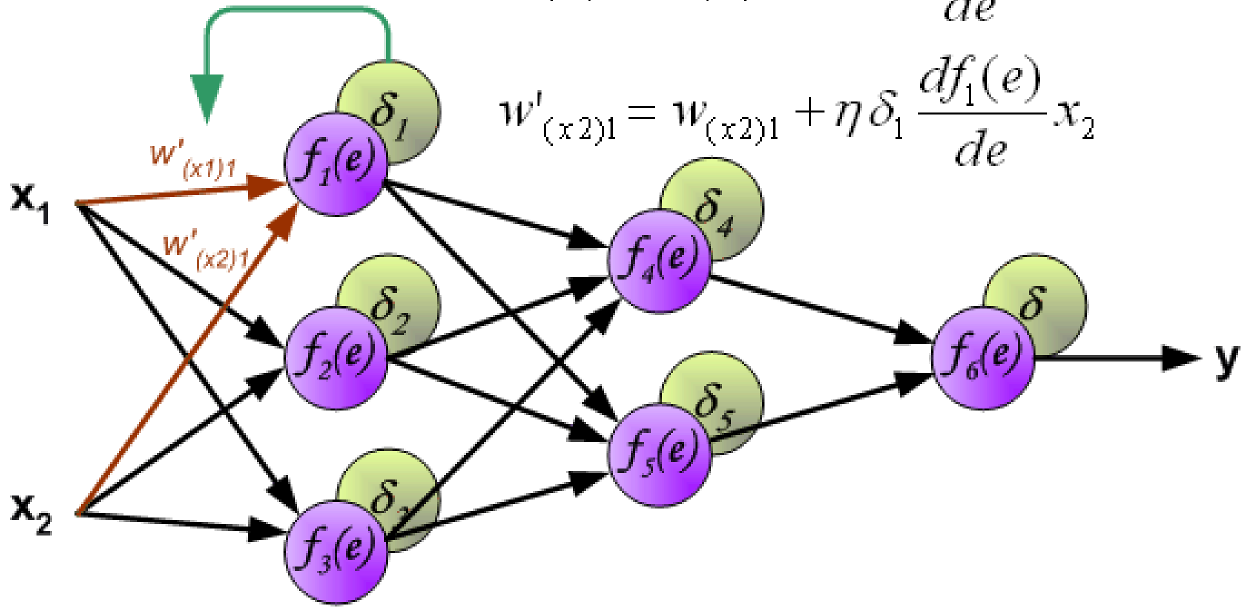


Learning Algorithm: Backpropagation

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)

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

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



Definitions

For each neuron (*j*), the output of that neuron is

$$o_j = g(\sum_{k=1}^{n} w_{kj} o_k)$$

For the first layer of neurons, o_k is just the input variables x_k

Definitions

For each neuron (*j*), the output of that neuron is

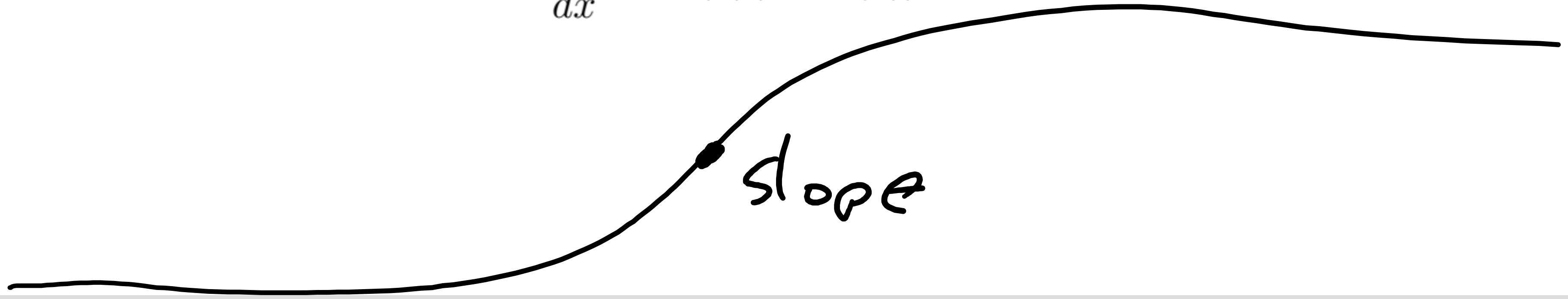
$$o_j = g(\sum_{k=1}^{n} w_{kj} o_k)$$

To find the correct direction for our gradient descent, we need to find the partial derivative of the error with respect to each weight.

- This will depend on our activation function
- By selecting the sigmoid as our activation function g(x), we get the following derivative

$$\frac{dg(x)}{dx} = g(x)(1 - g(x))$$

$0.5 \quad 0.5 = 0.25$



Finding the direction of descent

Then, to find how each weight impacts the final error, given by the squared error function for each sample:

$$E = \frac{1}{2}(y - f(x)^2)$$

We then need to find $\frac{\partial E}{\partial w_{ij}}$ in order to determine the direction of descent. We can a calculate this by:

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj}o_k \longleftarrow$$ before the activation function

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

$$\frac{\partial E}{\partial o_j} =$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j} = \sum_{\ell \in L} \left(\frac{\partial E}{\partial \mathrm{net}_\ell} \frac{\partial \mathrm{net}_\ell}{\partial o_j} \right) = \sum_{\ell \in L} \left(\frac{\partial E}{\partial o_\ell} \frac{\partial o_\ell}{\partial \mathrm{net}_\ell} w_{j\ell} \right)$$

$$\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j} = \sum_{\ell \in L} \left(\frac{\partial E}{\partial \mathrm{net}_\ell} \frac{\partial \mathrm{net}_\ell}{\partial o_j} \right) = \sum_{\ell \in L} \left(\frac{\partial E}{\partial o_\ell} \frac{\partial o_\ell}{\partial \mathrm{net}_\ell} w_{j\ell} \right)$$

$$\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

- This is for the arbitrary neuron *j*, yet it depends on other neurons *l*
- This is the sensitivity from the back propagation! $\longleftarrow$ this is our δ_j value
- Think about the error for the output neuron

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j} = \sum_{\ell \in L} \left(\frac{\partial E}{\partial \text{net}_\ell} \frac{\partial \text{net}_\ell}{\partial o_j} \right) = \sum_{\ell \in L} \left(\frac{\partial E}{\partial o_\ell} \frac{\partial o_\ell}{\partial \text{net}_\ell} w_{j\ell} \right)$$

$$\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

- This is for the arbitrary neuron *j*, yet it depends on other neurons *l*
- This is the sensitivity from the back propagation!
- Think about the error for the output neuron

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y} \frac{1}{2}(t-y)^2 = y - t \longleftarrow \text{this is our output error}$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial o_j}{\partial net_j} =$$

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

Finding the direction of descent

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

What are each of these terms?

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j} g(net_j) = g(net_j)(1 - g(net_j))$$

When g(x) is the sigmoid (a big advantage of using the sigmoid!)

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

$$\frac{\partial \mathrm{net}_j}{\partial w_{ij}} =$$

$$\frac{\partial xy}{\delta x} =$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

$$\frac{\partial \mathrm{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) =$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj}o_k$$

$$\frac{\partial \mathrm{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj}o_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij}o_i = o_i.$$

A nice and straightforward derivative here

Bringing it all together, we get

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} =$$

Finding the direction of descent

What are each of these terms?

$$\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$net_j = \sum_{k=1}^{n} w_{kj} o_k$$

$$\frac{\partial \mathrm{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij} o_i = o_i .$$

A nice and straightforward derivative here

Bringing it all together, we get

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} = \begin{cases} (o_j - t_j) o_j (1 - o_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} w_{j\ell} \delta_\ell) o_j (1 - o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

Notice that $\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$ is here included in the sensitivity delta

Finding the direction of descent

What should the final change in our weights be then?

Finding the direction of descent

What should the final change in our weights be then?

outputs

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \delta_j o_i$$

calculate sensitivities

(plus activation derivative)

Finding the direction of descent

What should the final change in our weights be then?

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \delta_j o_i$$

Where:

- η : learning rate
- δ_j : the back-propogation factor (the sensitivity) for neuron j
- o_i : the output of the neuron *i* (which is input into j with weight w_{ij})

This means you need to store or save all of this information in your Neural Network code

1.) Feedforward $\rightarrow$ outputs o_i

2.) Backprop $\rightarrow$ sensitivities δ_j

3.) Change the weights $\rightarrow -\eta \delta_j o_i$

Back propagation

This is considered one learning iteration: we adjust all of our weights in one iteration

- Can be multiple data points in one “batch”
- Can be one single point at a time $\leftarrow$ Simpler

What are some strengths?

Back propagation

This is considered one learning iteration: we adjust all of our weights in one iteration

- Can be multiple data points in one "batch"
- Can be one single point at a time

What are some strengths?

- Sigmoid gives us some advantages as far as differentials
- We adjust based on new feedback
  - We can actually adjust partially based on new points (no need to retrain the whole network with new data)
- With random start weights, hopefully each neuron finds its own novel trend in the data
-

Back propagation

This is considered one learning iteration: we adjust all of our weights in one iteration

- Can be multiple data points in one "batch"
- Can be one single point at a time

What are some strengths?

- Sigmoid gives us some advantages as far as differentials
- We adjust based on new feedback
  - We can actually adjust partially based on new points (no need to retrain the whole network with new data)
- With random start weights, hopefully each neuron finds its own novel trend in the data

Weaknesses?

-

Back propagation

This is considered one learning iteration: we adjust all of our weights in one iteration

- Can be multiple data points in one "batch"
- Can be one single point at a time

What are some strengths?

- Sigmoid gives us some advantages as far as differentials
- We adjust based on new feedback
  - We can actually adjust partially based on new points (no need to retrain the whole network with new data)
- With random start weights, hopefully each neuron finds its own novel trend in the data

Weaknesses?

- Gradient descent finds local minima
  - May want to train several neural networks with random start weights
  - Keep the best (or some k best)
- Has difficulty with plateaus (regions where there is little change in the output)
- Harder to fit "deep" neural networks accurately

where to stop

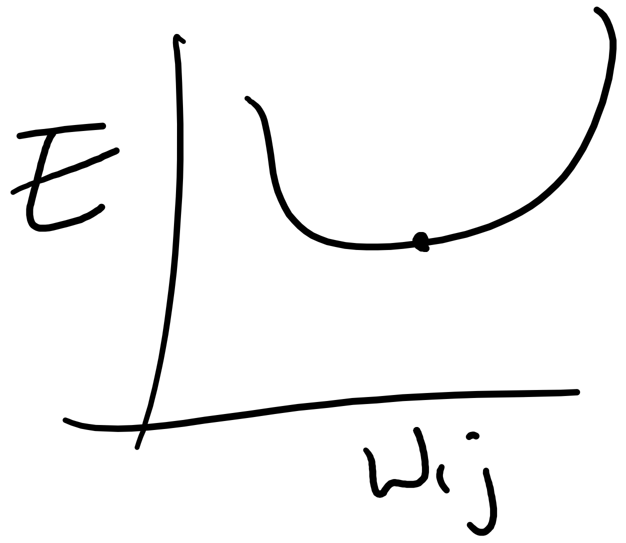
Convergence of backprop

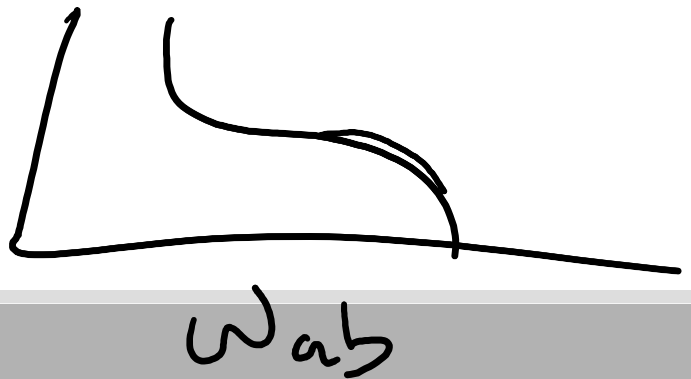
Simple Perceptron leads to convex optimization

- Gradient descent reaches **global** ~~**minima**~~ minimum

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





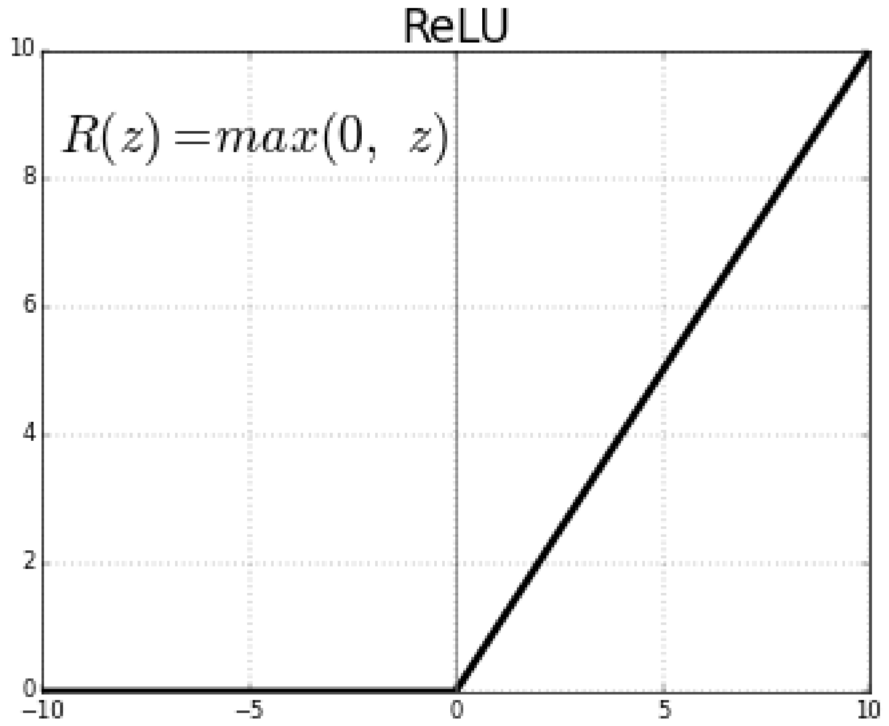
Exam Questions

Neural Networks:

- For the written neural networks problems, we will use the ReLU activation function
  - It's easier to calculate
  - It is common in practice
- Keep this in mind, because this will be the outputs of the hidden nodes

What is its derivative?

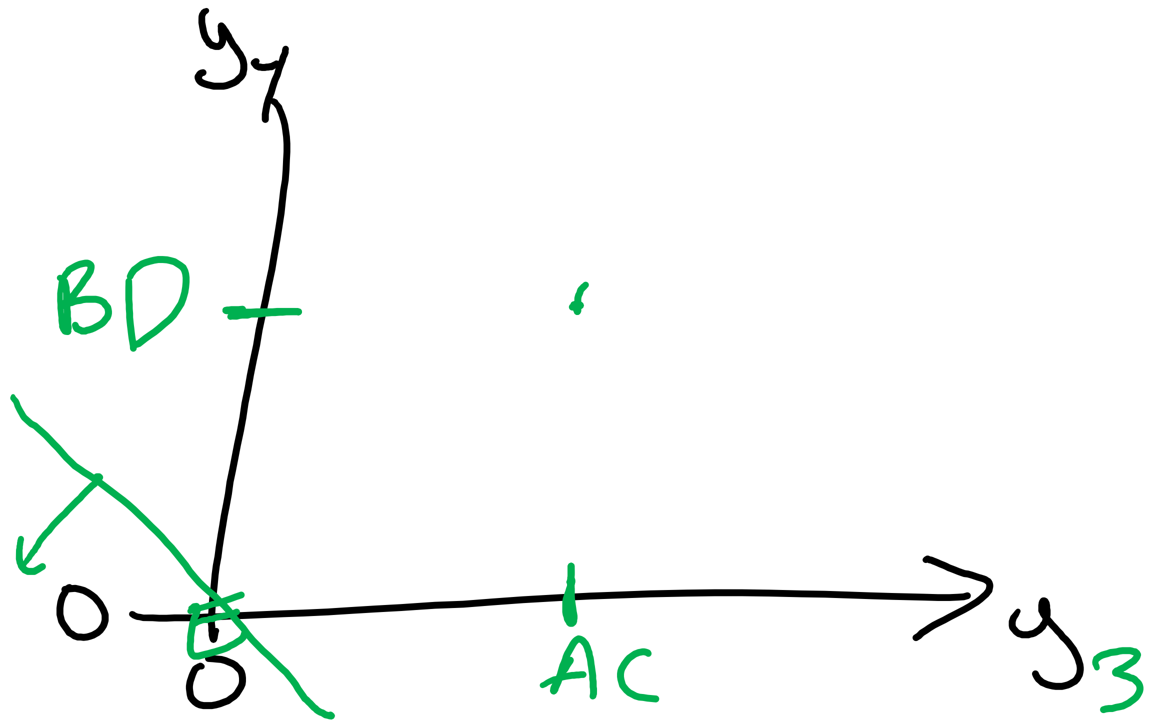
Step function

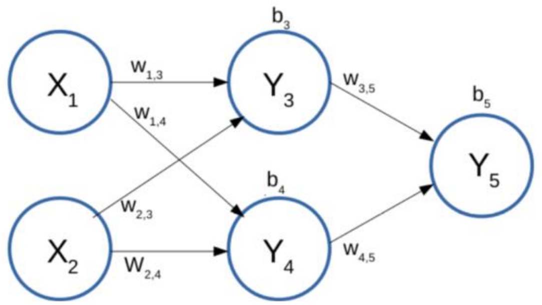


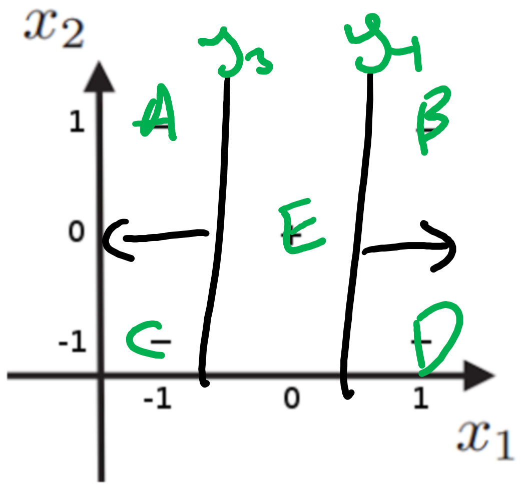
Exam Questions

Given the following neural network, and weights/offsets in {-1,0,1} fit the following data such that all of the negative values have output 0 and the positive point has positive output

Use ReLU as your activation function



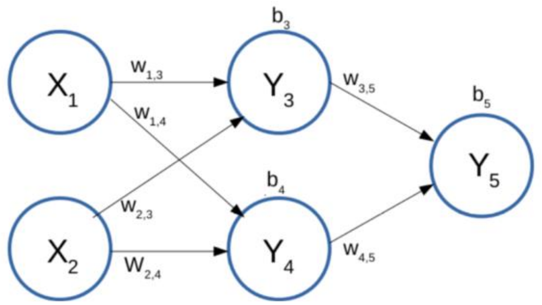


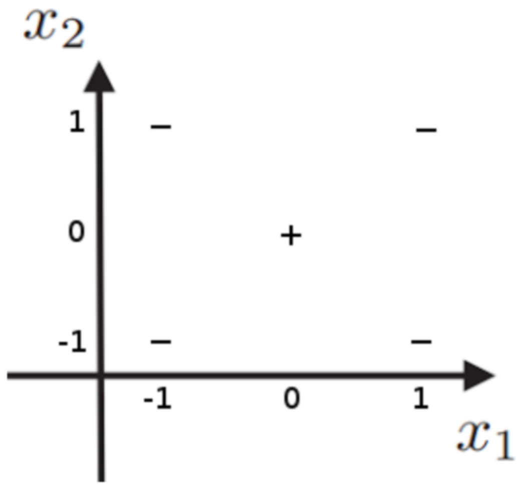


Exam Questions

Given the following neural network, and weights/offsets in {-1,0,1} fit the following data such that all of the negative values have output 0 and the positive point has positive output

Use ReLU as your activation function





Exam Questions

Given the following neural network, and weights/offsets in {-1,0,1} fit the following data such that all of the negative values have output 0 and the positive point has positive output

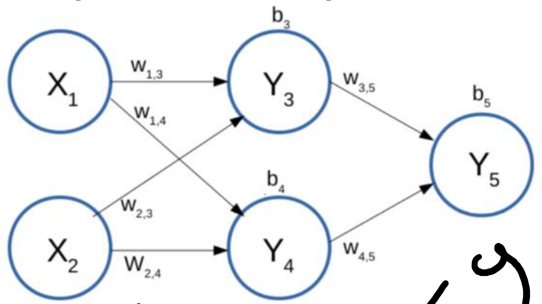
Use ReLU as your activation function

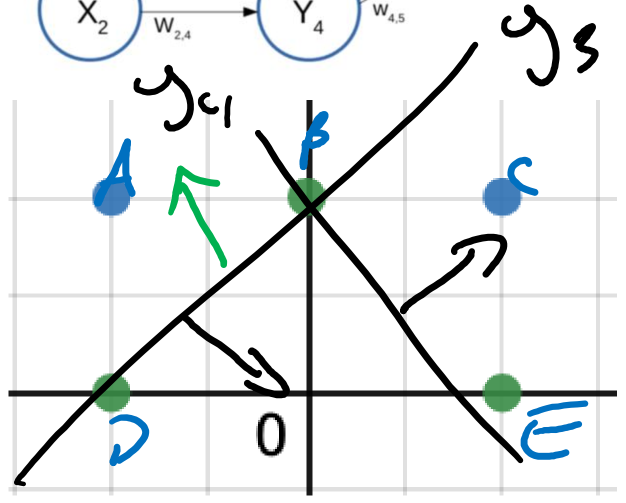
y_4 y_3

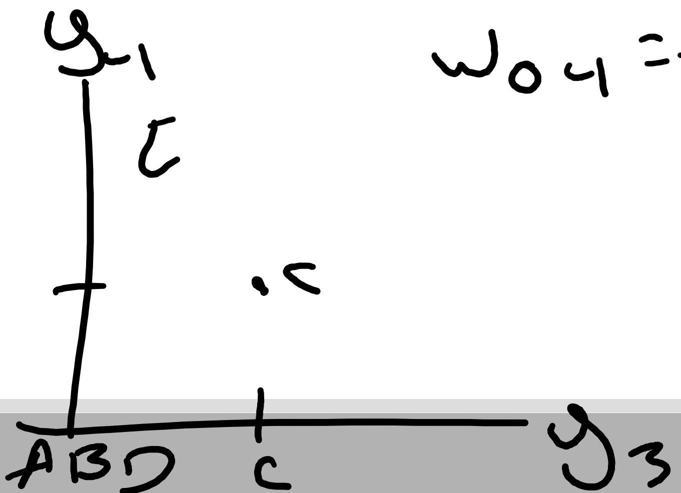
$w_{14} = 1$

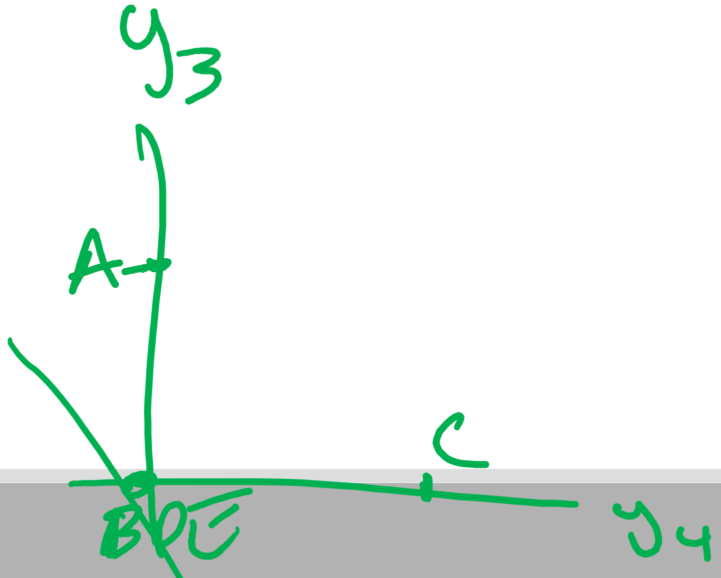
$w_{24} = 1$

$w_{04} = -1$





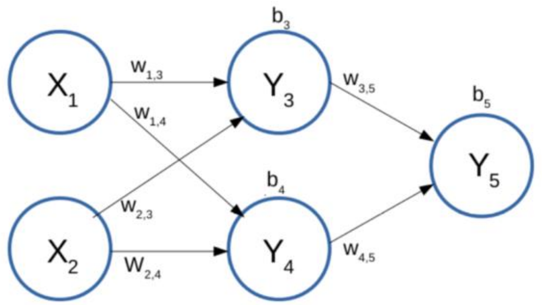


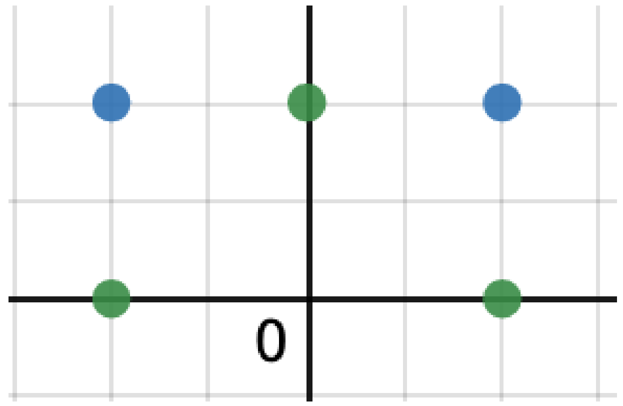


Exam Questions

Given the following neural network, and weights/offsets in {-1,0,1} fit the following data such that all of the negative values have output 0 and the positive point has positive output

Use ReLU as your activation function





Exam Questions

Draw your decision boundaries.