COSC343: Artificial Intelligence

Lecture 10 : Artificial neural networks

Lech Szymanski

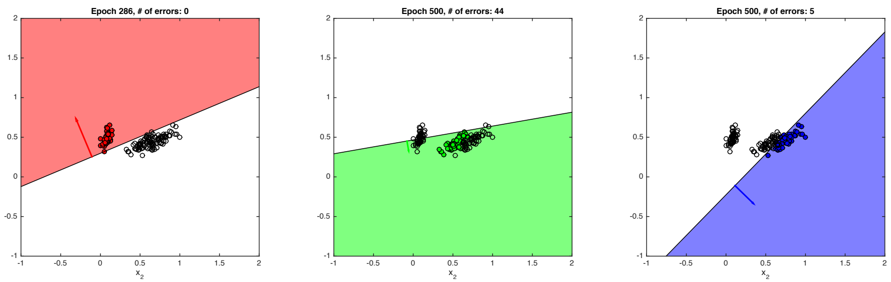
Dept. of Computer Science, University of Otago

In today's lecture

- Multi-layer perceptrons (MLPs)
- Sigmoid activation function
- Backpropagation
- Backpropagated least squares

Recall: Limit of the perceptron

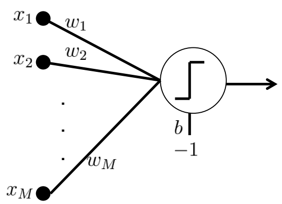
- Perceptron splits data space into two half-spaces
- If the required separation boundary is non-linear, perceptron cannot form a reasonable separation boundary
- But perceptron corresponds to single artificial neurons with many inputs – what if we created a networks of neurons?



Multi-layer perceptrons (MLP)

Minsky and Papert's book *Perpectrons* (1969) set out the following ideas:

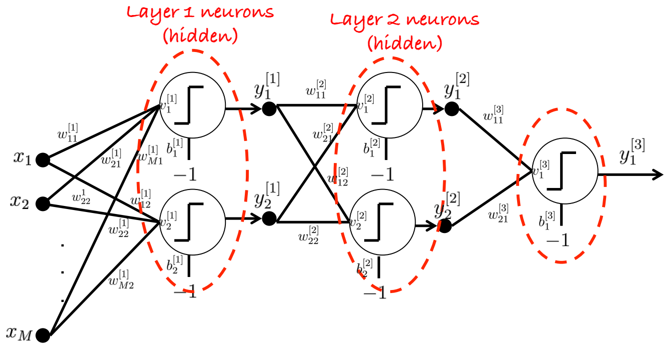
- Multi-layer perceptrons can compute any function
- 1-layer perceptrons can only compute linearly separable functions
- The perceptron learning rule only works for 1-layer perceptrons



Multi-layer perceptrons (MLP)

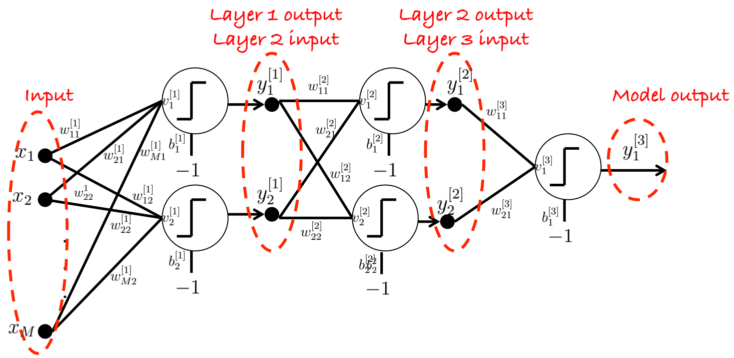
Minsky and Papert's book *Perpectrons* (1969) set out the following ideas:

- Multi-layer perceptrons can compute any function
- 1-layer perceptrons can only compute linearly separable functions
- The perceptron learning rule only works for 1-layer perceptrons



Multi-layer perceptrons (MLP)

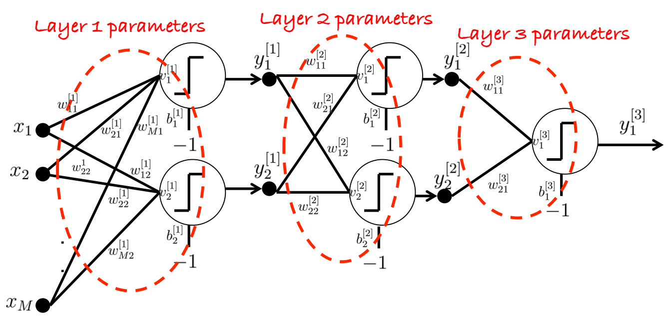
- \Minsky and Papert's book *Perpectrons* (1969) set out the following ideas:
- Multi-layer perceptrons can compute any function
- 1-layer perceptrons can only compute linearly separable functions
- The perceptron learning rule only works for 1-layer perceptrons



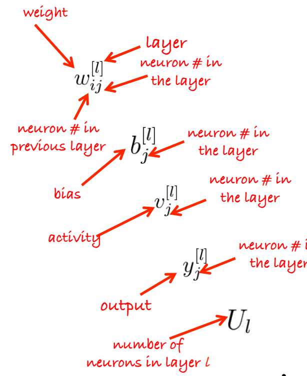
Multi-layer perceptrons (MLP)

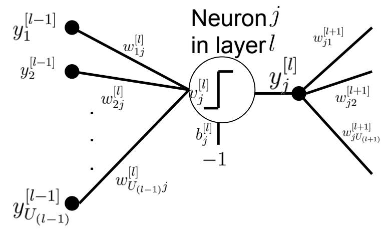
Minsky and Papert's book *Perpectrons* (1969) set out the following ideas:

- Multi-layer perceptrons can compute any function
- 1-layer perceptrons can only compute linearly separable functions
- The perceptron learning rule only works for 1-layer perceptrons



MLP computation





Programmer makes a decision how many layers and how many neurons per layer!

- Activity is the weighted sum of inputs from the previous layer minus the bias $\quad v_j^{[l]} = \sum_{i=1}^{U^{[l-1]}} w_{ij}^{[l]} y_i^{[l-1]} - b_j^{[l]}$
- Output is a function of activity $\quad y_j^{[l]} = f_{\text{hardlim}}(v_j^{[l]}) = \begin{cases} 1 & v_j^{[l]} \geq 0 \\ 0 & \text{otherwise} \end{cases}$

where $y_i^{[0]} = x_i$ and $U_0 = M$

MLP computation (matrix form)

weight

layer

neuron # in the layer

$w_{ij}^{[l]}$

neuron # in previous layer

$b_j^{[l]}$ neuron # in the layer

bias

$v_j^{[l]}$ neuron # in the layer

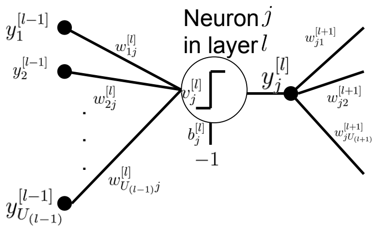
activity

$y_j^{[l]}$ neuron # in the layer

output

U_l

number of neurons in layer l



$$\mathbf{v}_l = \mathbf{W}_l^T \mathbf{y}_{(l-1)} - \mathbf{b}_l$$

weight vector of the 1st neuron in the layer

weight vector of the U_l neuron in the layer

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} \quad \mathbf{v}_l = \begin{bmatrix} v_1^{[l]} \\ \vdots \\ v_{U_l}^{[l]} \end{bmatrix} \quad \mathbf{W}_l = \begin{bmatrix} w_{11}^{[l]} & \cdots & w_{1U_l}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{U_{(l-1)}1}^{[l]} & \cdots & w_{U_{(l-1)}U_l}^{[l]} \end{bmatrix}$$

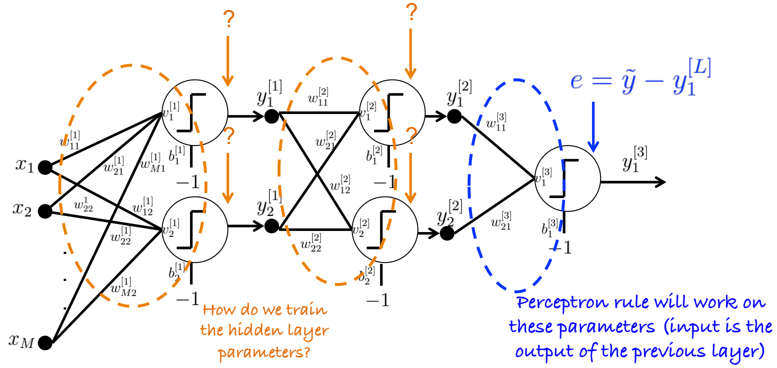
$$\mathbf{y}_l = \begin{bmatrix} y_1^{[l]} \\ \vdots \\ y_{U_l}^{[l]} \end{bmatrix} \quad \mathbf{b}_l = \begin{bmatrix} b_1^{[l]} \\ \vdots \\ b_{U_l}^{[l]} \end{bmatrix} \quad \mathbf{y}_0 = \mathbf{x}$$

Programmer makes a decision how many layers and how many neurons per layer!

MLP learning

Recall the perceptron learning (delta) rule:

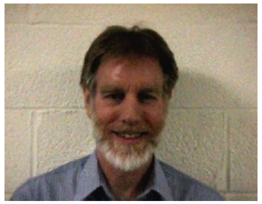
- Weights change by the value of input times the resulting error
- Bias changes by negative value of error (same rule as above with -1 input)



MLP learning

Is there a learning rule that explains how to update the weights of hidden units in a multi-layer perceptron?

Yes!



- First discovered by mathematicians (e.g. Bryson and Ho, 1969)...
- First applied to neural networks by Werbos (1981)...
- Made famous by Rumelhart, Hinton and Williams (1986).

Backpropagation

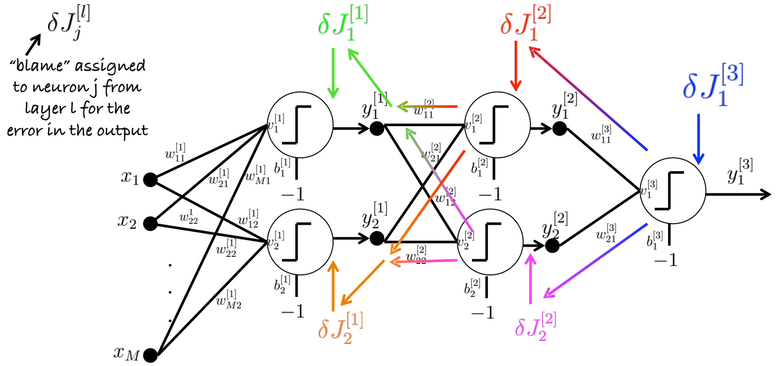
The basic idea behind **error backpropagation** is to take the error associated with the output unit, and *distribute* it amongst the units which provided input.

The input units which are connected with strongest weights need to take more 'responsibility' for the error.

MLP learning

The basic idea behind **error backpropagation** is to take the *blame* associated with the error in an output unit, and *distribute* it amongst the units which provided input.

The input units which are connected with strongest weights need to take more 'responsibility' for the error.



Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right)$$

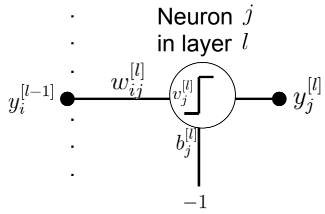
Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha \Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha \Delta b_j^{[l]}$$

where...



The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

where...

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right)$$

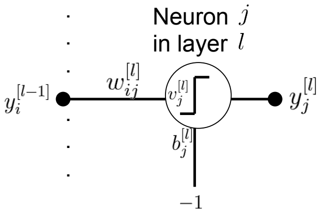
Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha \Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha \Delta b_j^{[l]}$$

where...



The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

Output of neuron i from the previous layer

Derivative of the activation function

Blame on neuron j

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

Derivative of the activation function

Blame on neuron j

where...

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

Sum over neurons in layer l

Weight connecting neuron i in layer l-1 to neuron j in layer l

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right)$$

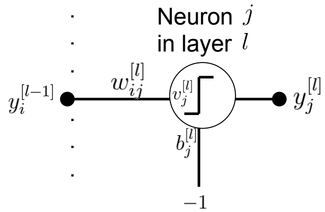
Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha \Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha \Delta b_j^{[l]}$$

where...



The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

Where does this derivative come from?

where...

Hard limiting function has no derivative

$$y_j^{[l]} = f_{\text{hardlim}}(v_j^{[l]}) = \begin{cases} 1 & v_j^{[l]} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial y_j^{[l]}}{\partial v_j^{[l]}} = \ ?$$

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]} \frac{dy_j^{[l]}}{dv_j^{[l]}} \delta J_j^{[l]}$$

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

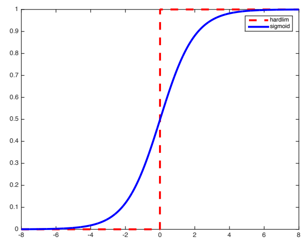
$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

Logistic sigmoid activation

Logistic sigmoid function

$$y_j = f_{\text{logsig}}(v_j) = \frac{1}{1+exp^{-v_j}}$$

$$\frac{dy_j}{dv_j} = y_j(1-y_j)$$

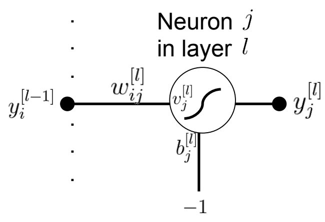


v

Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right)$$



Sigmoid function has a derivative

$$y_j^{[l]} = f_{\text{logsig}}(v_j^{[l]}) = \frac{1}{1+\exp^{-v_j^{[l]}}}$$

$$\frac{\partial y_j^{[l]}}{\partial v_j^{[l]}} = y_j^{[l]}(1-y_j^{[l]})$$

Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha\Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha\Delta b_j^{[l]}$$

where...

The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

where...

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

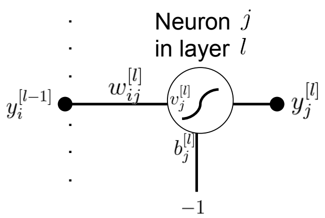
The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right)$$



Sigmoid function has a derivative

$$y_j^{[l]} = f_{\text{logsig}}(v_j^{[l]}) = \frac{1}{1+\exp^{-v_j^{[l]}}}$$

$$\frac{\partial y_j^{[l]}}{\partial v_j^{[l]}} = y_j^{[l]}(1-y_j^{[l]})$$

Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha\Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha\Delta b_j^{[l]}$$

where...

The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

Where does this derivative come from?

where...

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

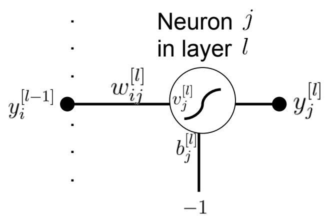
$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right) = \frac{1}{2}\left(y_j^{[L]} - \tilde{y}_j\right)^2$$

Let's use least squares error



Sigmoid function has a derivative

$$y_j^{[l]} = f_{\text{logsig}}(v_j^{[l]}) = \frac{1}{1+\exp^{-v_j^{[l]}}}$$

$$\frac{\partial y_j^{[l]}}{\partial v_j^{[l]}} = y_j^{[l]}(1-y_j^{[l]})$$

Steepest gradient descent update for weight connecting input i with neuron j in layer l is :

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha\Delta w_{ij}^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is :

$$b_j^{[l]} = b_j^{[l]} - \alpha\Delta b_j^{[l]}$$

where...

The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

Where does this derivative come from?

where...

The cost blame for neuron i in layer l-1 is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}}$$

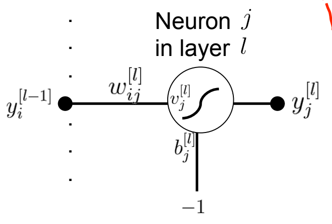
Backpropagation

Let's assume that we are minimising some cost function that depends on the network output and the desired output

$$J\left(y_j^{[L]}, \tilde{y}_j\right) = \frac{1}{2}\left(y_j^{[L]} - \tilde{y}_j\right)^2$$

Let's use least squares error

Then...



Sigmoid function has a derivative

$$y_j^{[l]} = f_{\text{logsig}}(v_j^{[l]}) = \frac{1}{1+\exp^{-v_j^{[l]}}}$$

$$\frac{\partial y_j^{[l]}}{\partial v_j^{[l]}} = y_j^{[l]}(1-y_j^{[l]})$$

Steepest gradient descent update for weight connecting input i with neuron j in layer l is:

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha\Delta w_{ij}^{[l]}$$

where...

The change in the weight connecting input i with neuron j in layer l is:

$$\Delta w_{ij}^{[l]} = y_i^{[l-1]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

The cost blame for neuron i in layer $l-1$ is:

$$\delta J_i^{[l-1]} = \sum_{j=1}^{U_l} w_{ij}^{[l]}\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

Steepest gradient descent update for bias on neuron j in layer l is:

$$b_j^{[l]} = b_j^{[l]} - \alpha\Delta b_j^{[l]}$$

The change in the bias for neuron j in layer l is:

$$\Delta b_j^{[l]} = -\frac{dy_j^{[l]}}{dv_j^{[l]}}\delta J_j^{[l]}$$

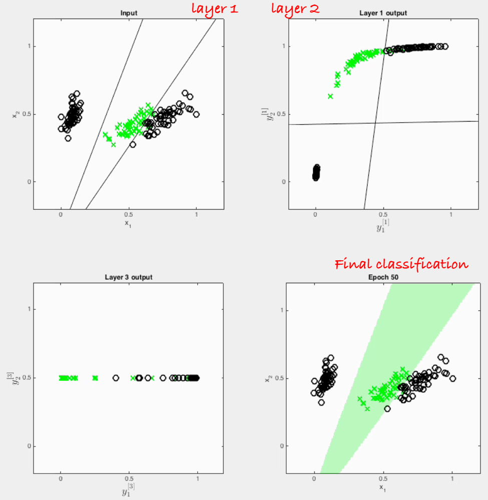
where...

The cost blame for neuron j in the output layer is a derivative of the overall cost with respect to the neuron's output:

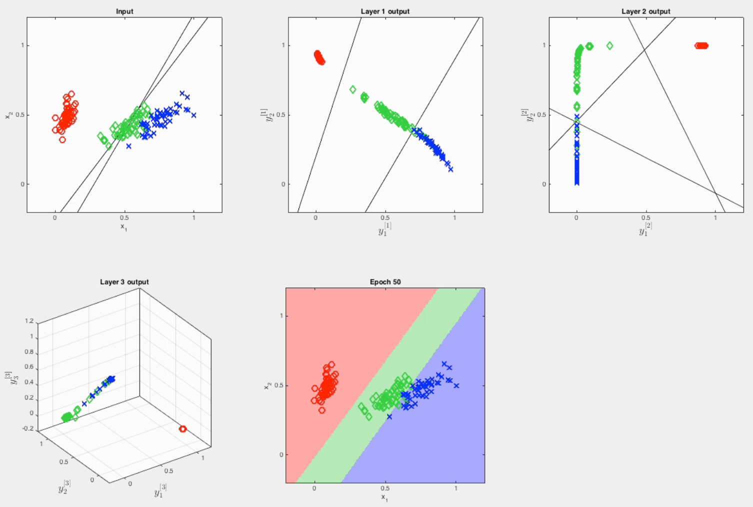
$$\delta J_j^{[L]} = \frac{dJ\left(y_j^{[L]}, \tilde{y}_j\right)}{dy_j^{[L]}} = (y_j^{[L]} - \tilde{y}_j)$$

An example: a 2-2-2-1 neural network

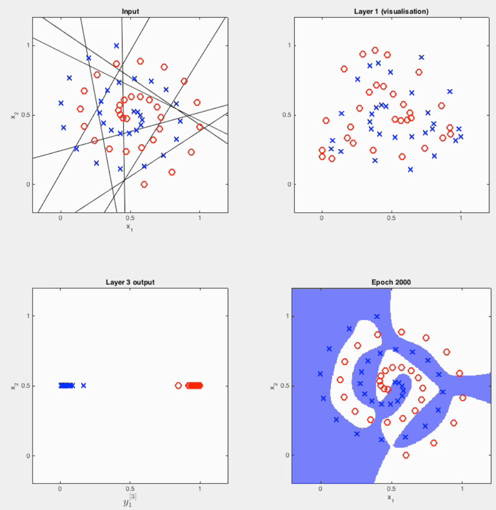
#inputs #nuerons layer 1 #nuerons layer 2 #output neurons



An example: a 2-2-2-3 neural network



An example: a 2-8-8-1 neural network



Summary and reading

- Multi-layer perceptron is a universal function approximator
  - In theory it can model anything
  - In practice, it's not obvious what the best architecture is
- Backpropagation allows training of hidden weights and biases
  - Requires differentiable activation functions – sigmoid is a very common choice

Reading for the lecture: AIMA Chapter 18 Sections 7.1,7.2

Reading for next lecture: No reading