CSC345/M45:
Big Data & Machine Learning
(neural network)

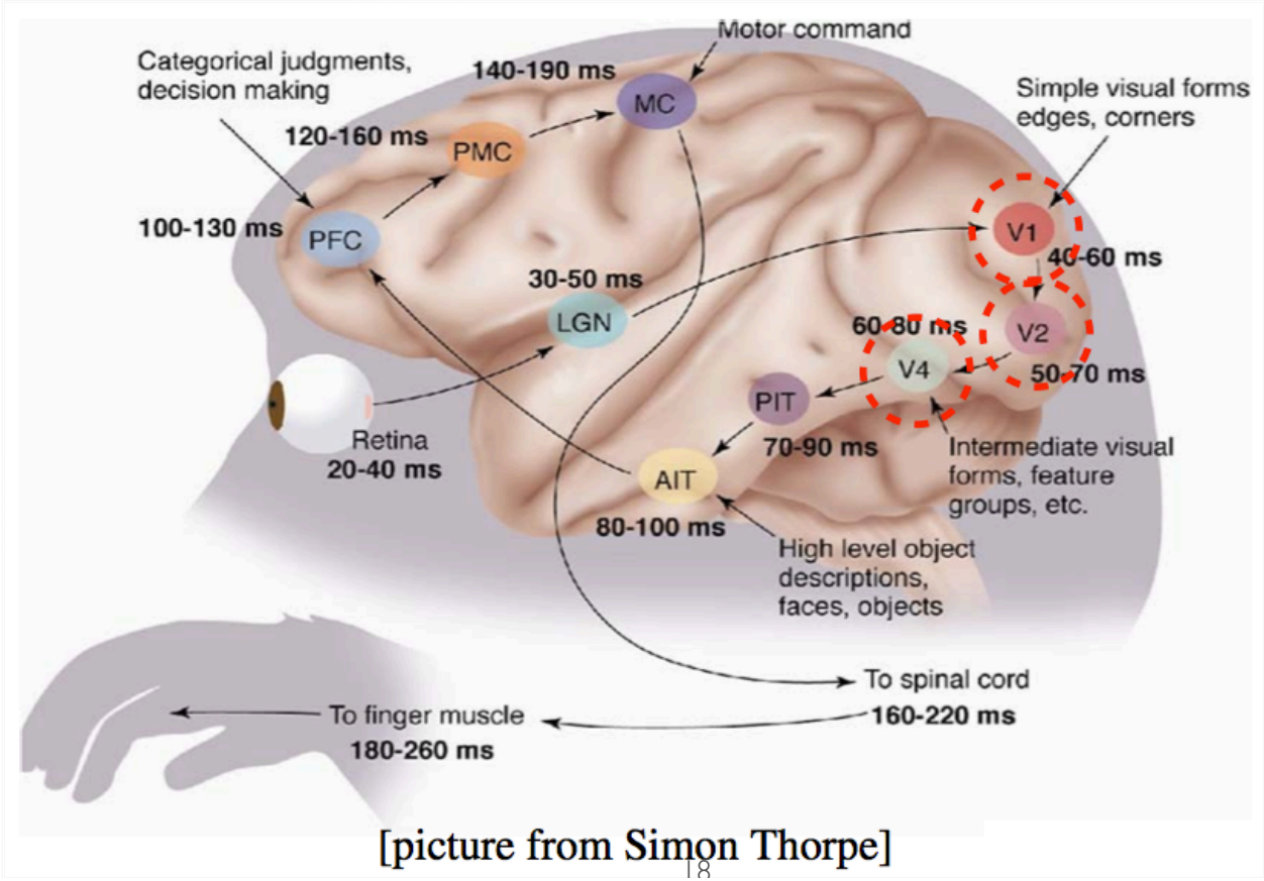
Prof. Xianghua Xie

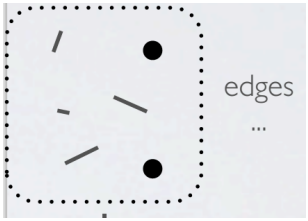
x.xie@swansea.ac.uk
http://csvision.swan.ac.uk
224 Computational Foundry, Bay Campus

“Experiment ”

Human Brain

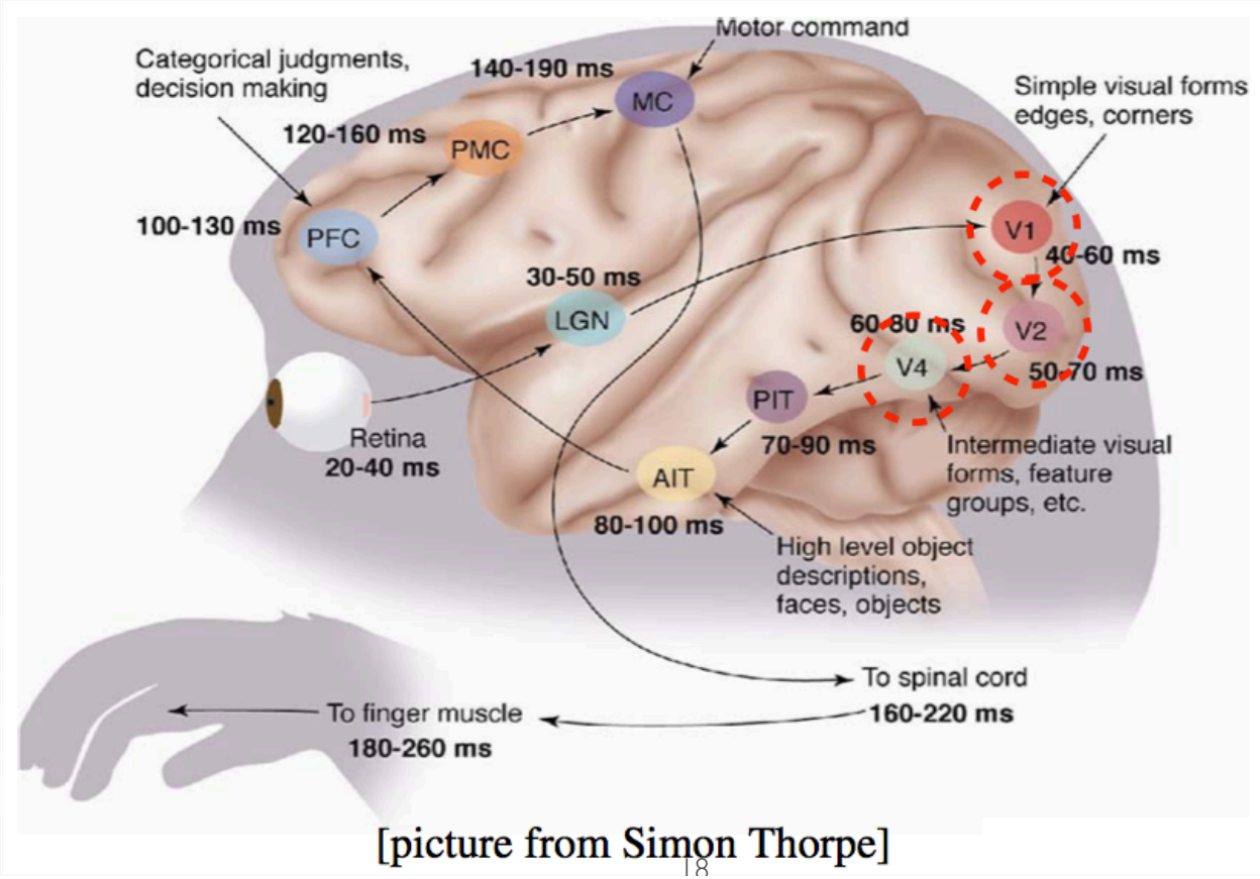
- Visual cortex
  - part of the brain responsible for processing visual information that we get from our retina.

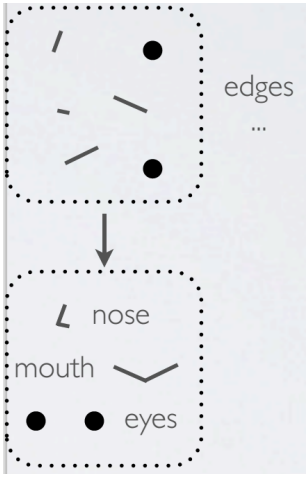




Human Brain

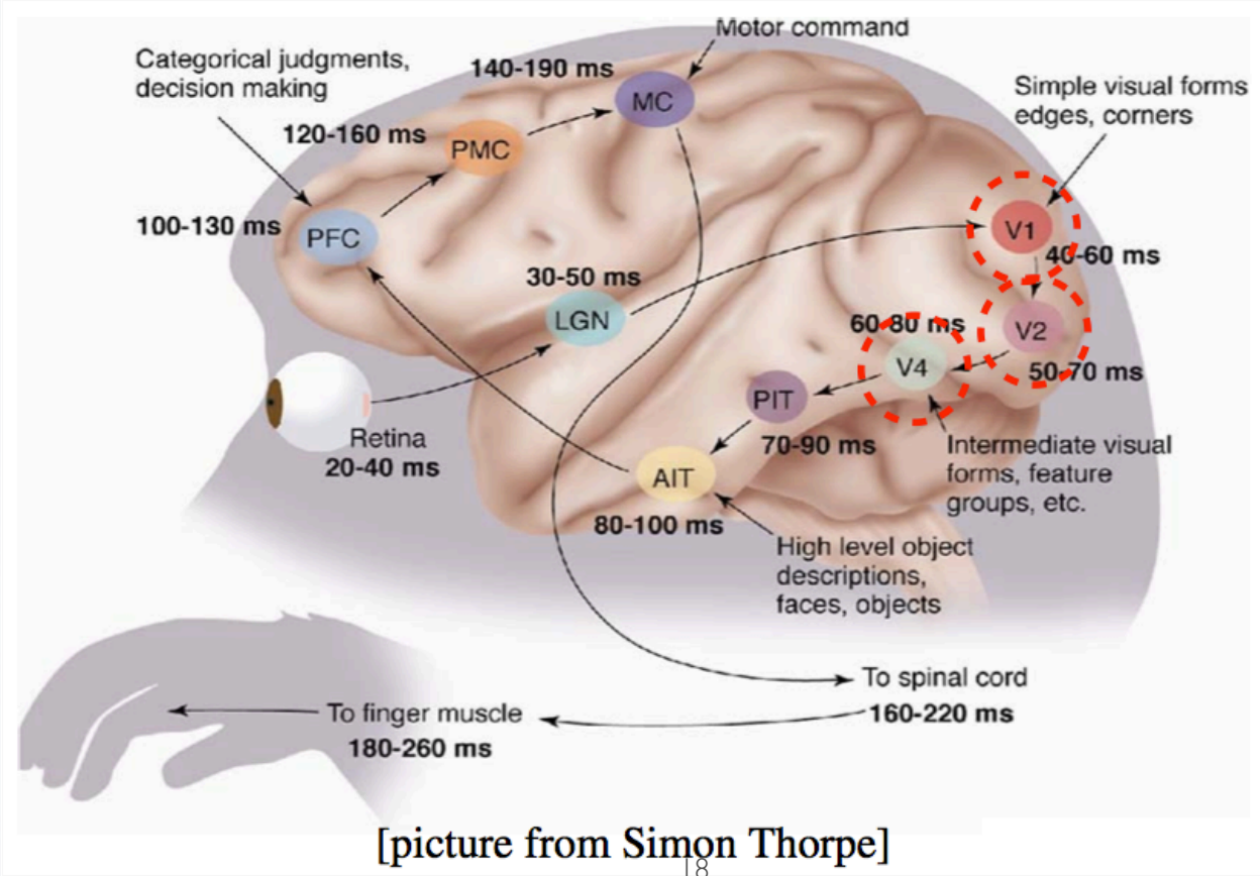
- Visual cortex
  - part of the brain responsible for processing visual information that we get from our retina.

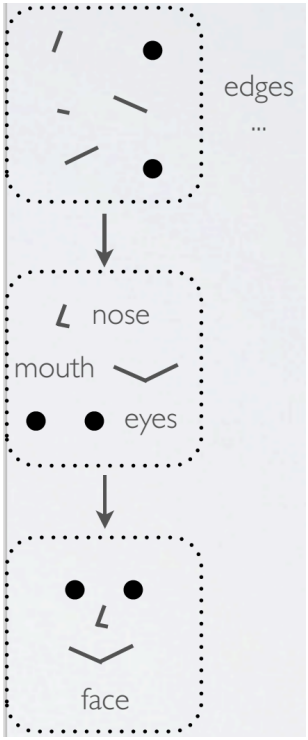




Human Brain

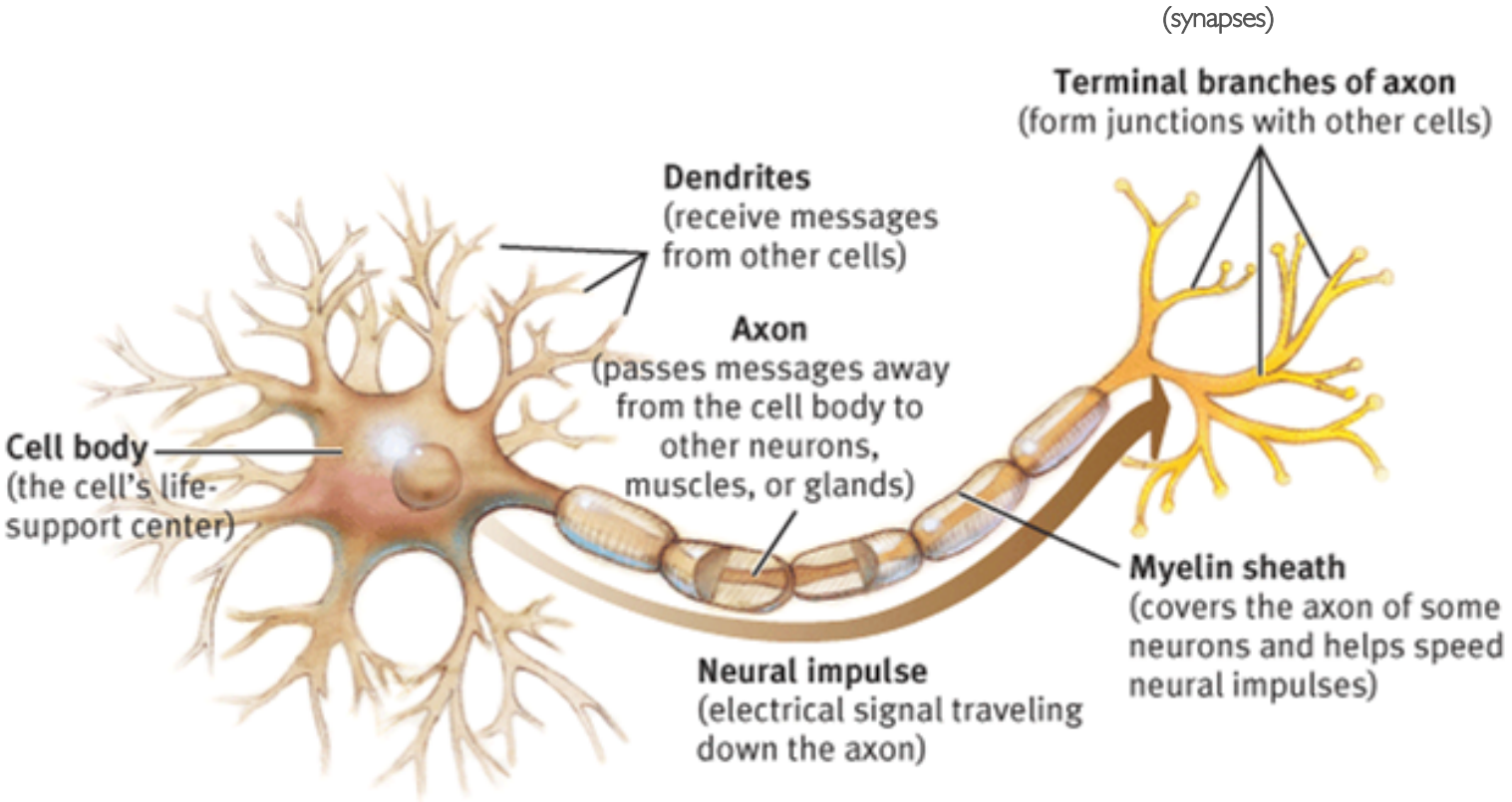
- Visual cortex
  - part of the brain responsible for processing visual information that we get from our retina.





Human Brain

- Number of neurons: 10^{11}; >20 types; 10^{14} synapses
- Neuron switching time: 1~10ms cycle time
- Signals are noisy “spike trains” of electrical potential
- Scene recognition speed: ~0.1s



Artificial Neural Network

- ANN: a bottom-up attempt to model the functionality of brain

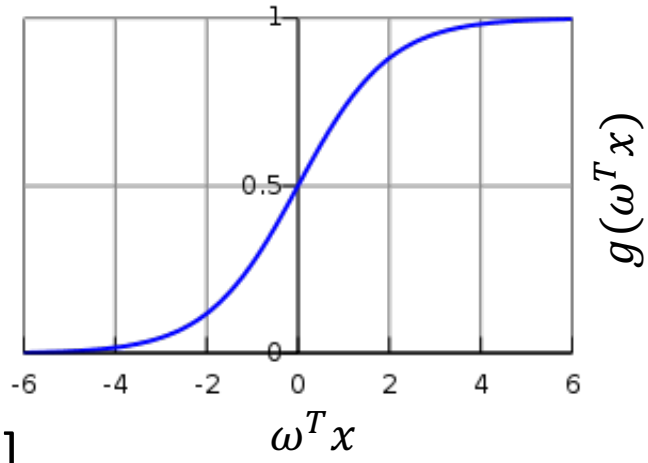
- ANNs are biologically inspired, but not necessarily biologically plausible

- ANN properties
  - Many neuron-like switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process (GPU)
  - Emphasis on tunning weights automatically

Recap: Logistic Regression

- Logistic function
  - Also known as **sigmoid function**

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



  - The dependent variable needs fall in [0,1]

$$0 \leq h_\omega(x) \leq 1$$

  - The regression takes the following form:

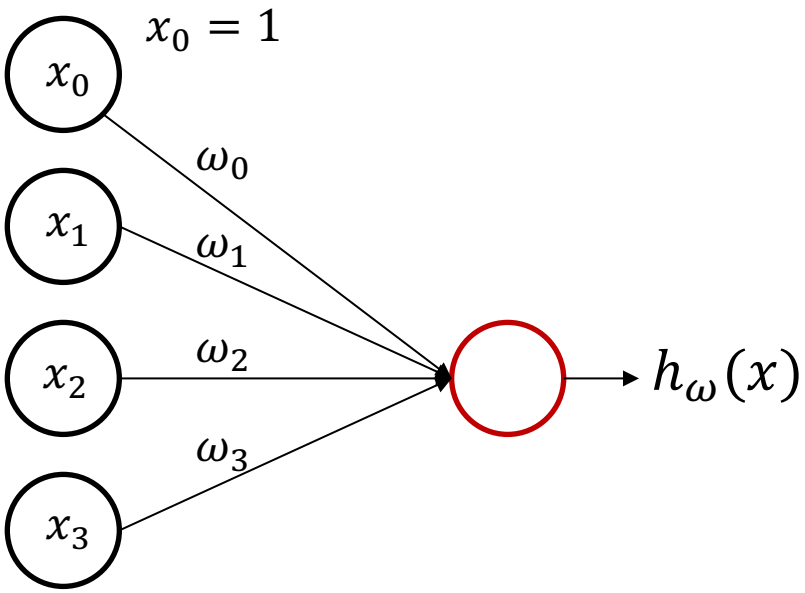
$$h_\omega(x) = g(\omega^T x) = \frac{1}{1 + e^{-\omega^T x}}$$

Logistic Unit

- A graphical representation of a logistic function

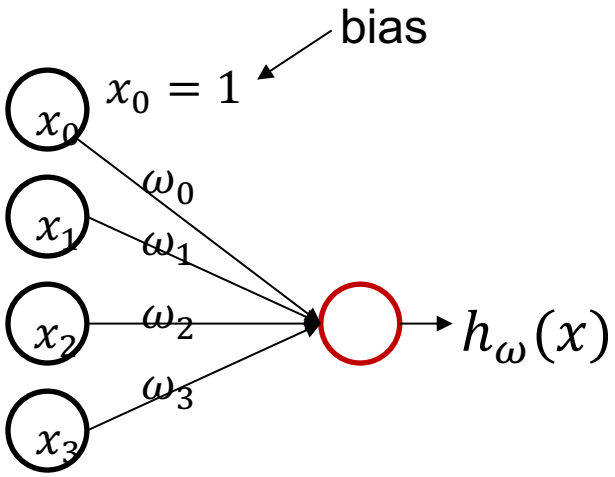
$$h_\omega(x) = g(\omega^T x) = \frac{1}{1 + e^{-\omega^T x}}$$

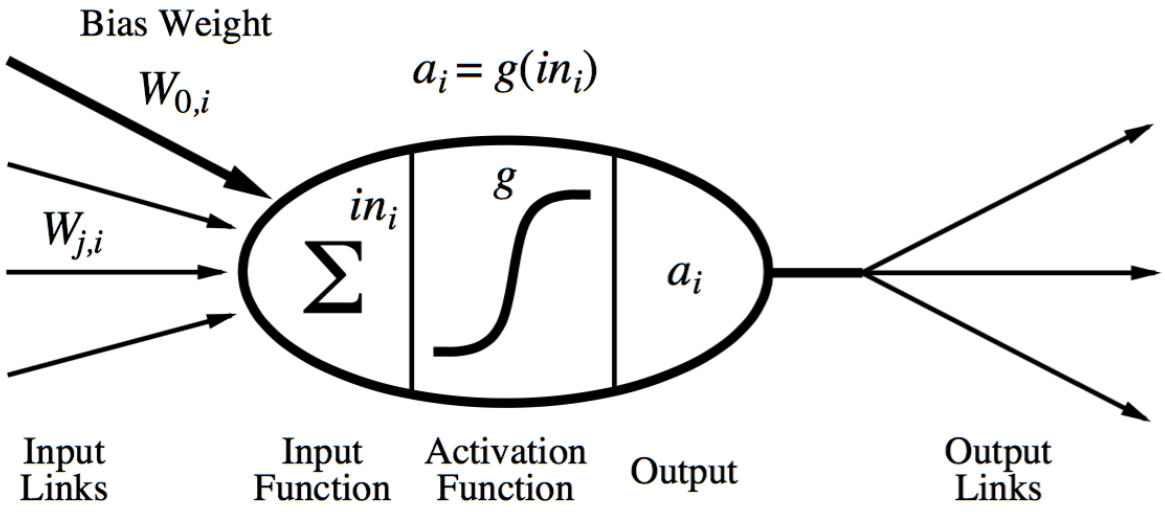
e.g. $\omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$ $x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$



Logistic Unit

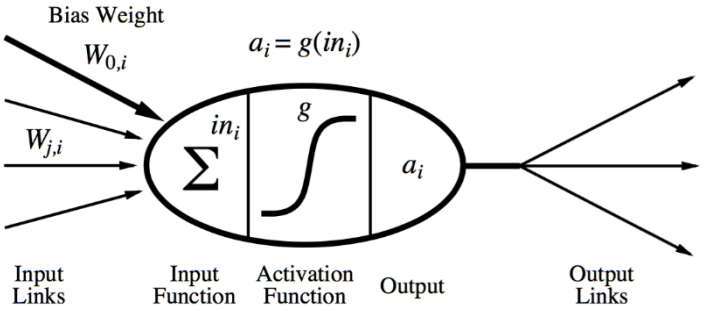
- Neuron model: Perceptron
  - A gloss simplification of real neurons; synthetic interpretation of biological neurons
  - Its purpose is to develop understanding of the capabilities of various networks of such simple units

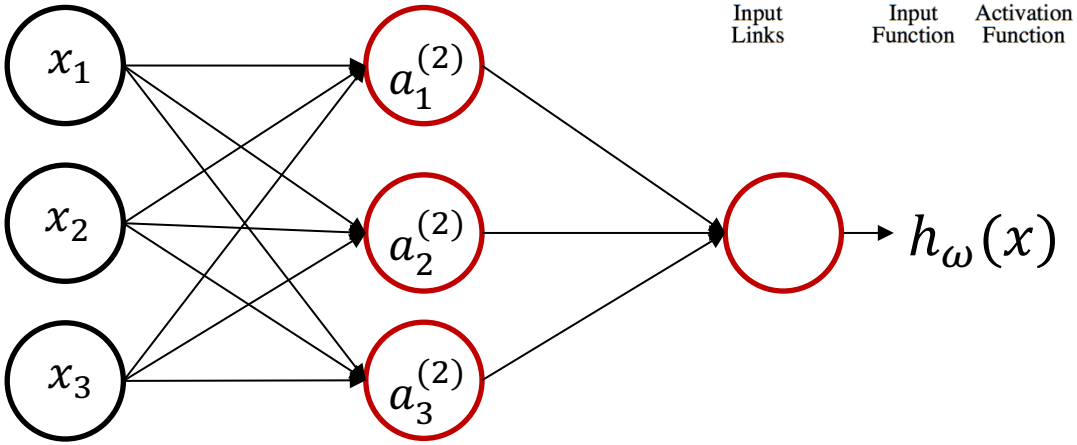




Neural Network Definition

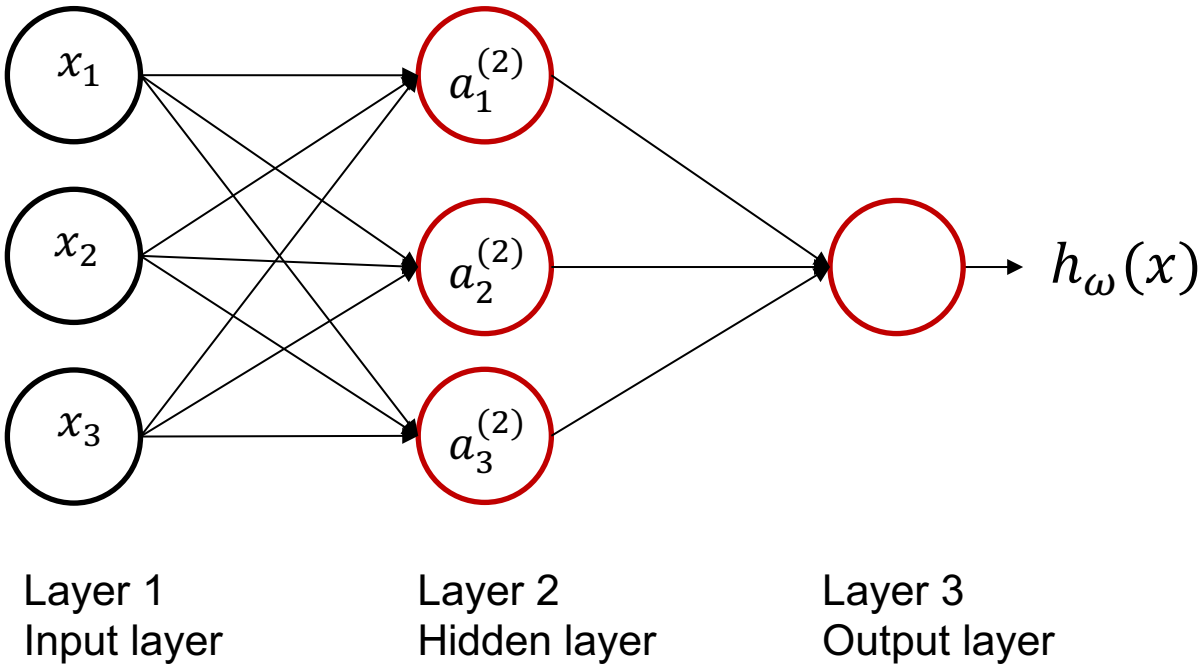
- A neural network is characterized by
  1. its pattern of connections between the neurons (called its *architecture),*
  2. its method of determining the weights on the connections (called its *training,* or *learning, algorithm),* and
  3. its *activation function*





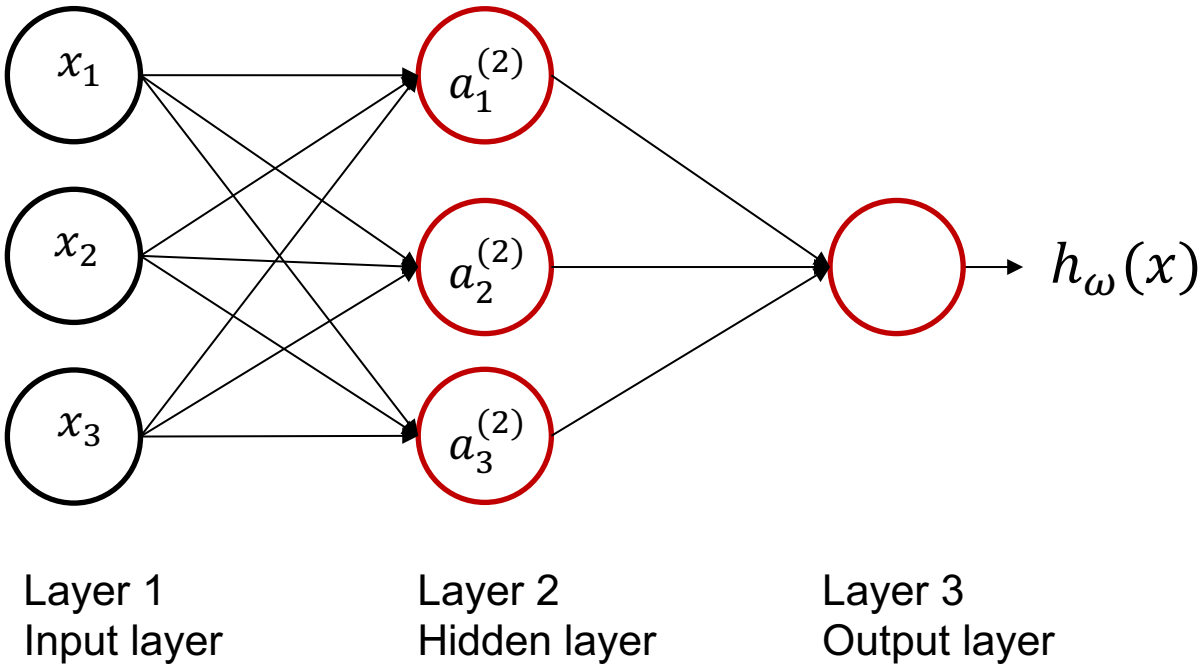
Neural Network

- Graphical representation
  - Typically, we don't draw the bias units.



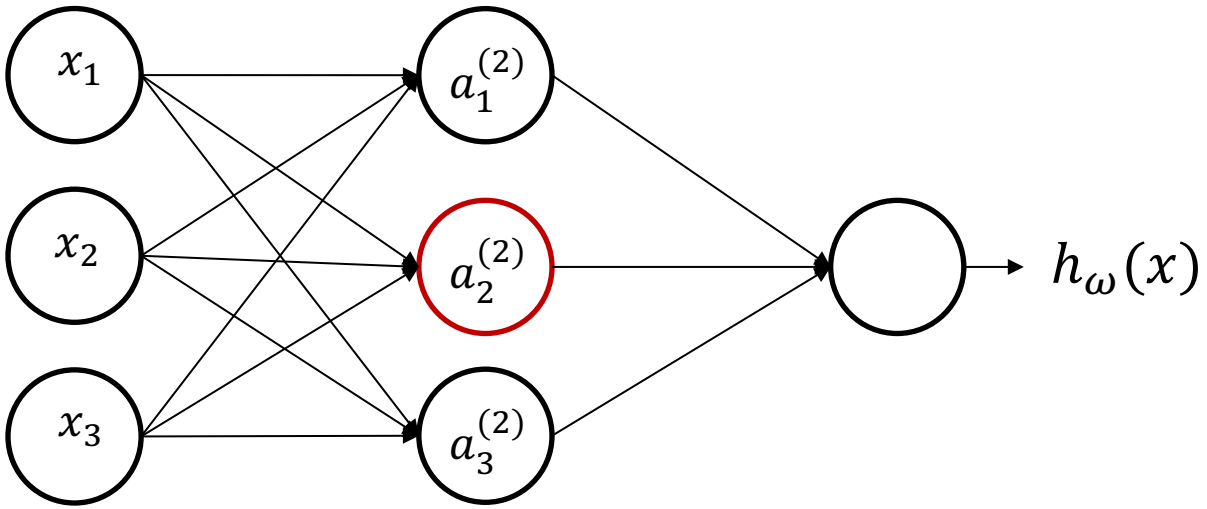
Neural Network

- Learning its own features
  - For all nodes beyond layer 2, each neuron takes output from other neurons as input
  - Those outputs however are learned results of sub-neural networks



Neural Network: Representation

- $a_i^{(j)}$ "activation" of unit i in layer j
- $\Omega^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$
  - If network has n_j nodes in layer j and n_{j+1} nodes in layer $j+1$, then $\Omega^{(j)}$ will be of dimension of $n_{j+1} \times (n_j + 1)$

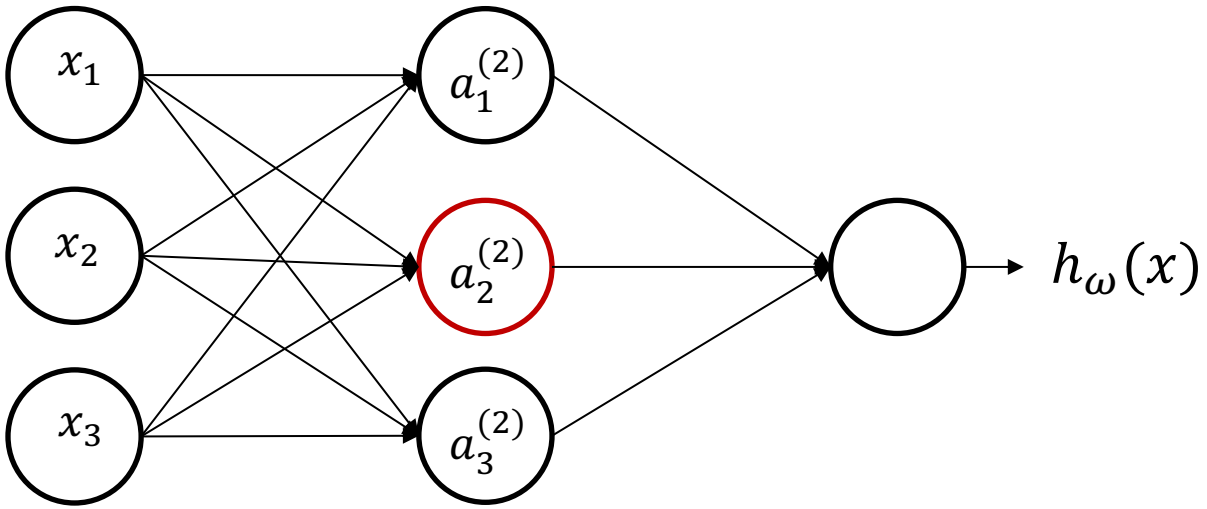


Neural Network: Representation

- $a_i^{(j)}$ "activation" of unit i in layer j
- $\Omega^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$
  - If network has n_j nodes in layer j and n_{j+1} nodes in layer $j+1$, then $\Omega^{(j)}$ will be of dimension of $n_{j+1} \times (n_j + 1)$

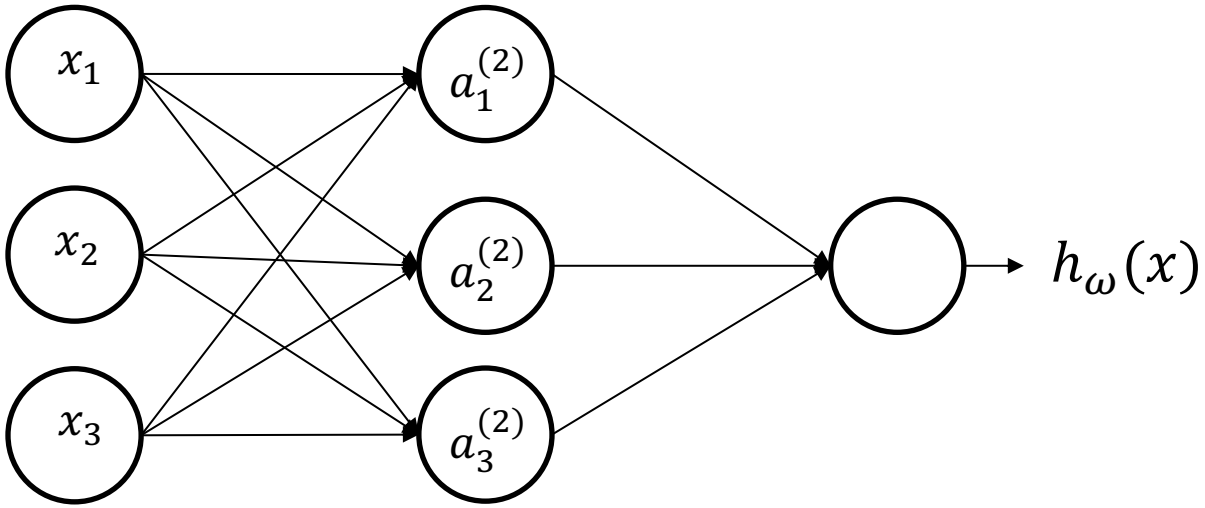
e.g. $a_2^{(2)} = g(\Omega_{20}^{(1)} x_0 + \Omega_{21}^{(1)} x_1 + \Omega_{22}^{(1)} x_2 + \Omega_{23}^{(1)} x_3)$

$h_\omega(x) = a_1^{(3)} = g(\Omega_{10}^{(2)} a_0^{(2)} + \Omega_{11}^{(2)} a_1^{(2)} + \Omega_{12}^{(2)} a_2^{(2)} + \Omega_{13}^{(2)} a_3^{(2)})$



Two-class Classification Problem

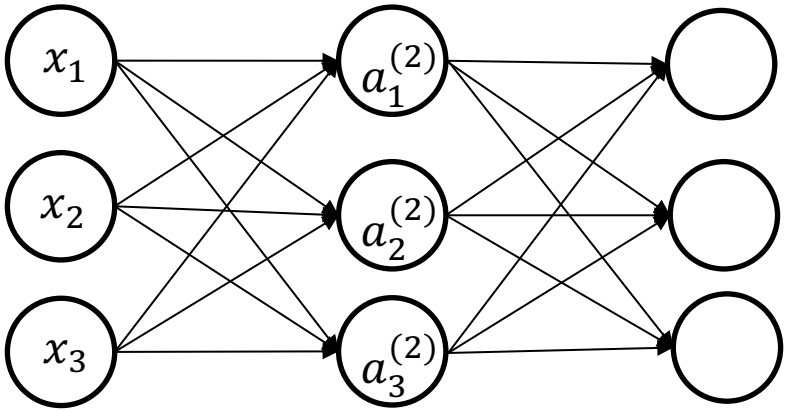
- Binary classification
  - A single output unit/node is sufficient:



    - Output >=0.5, then y=1;
    - Output <0.5, then y=0.
  - Training data is prepared as such:
    - $(x_{(1)}, y_1)$, $(x_{(2)}, y_2)$, …
    - $x_{(i)} = (x_1, x_2, x_3)^T$, $y_i = 0$ or 1

Multi-class Classification Problem

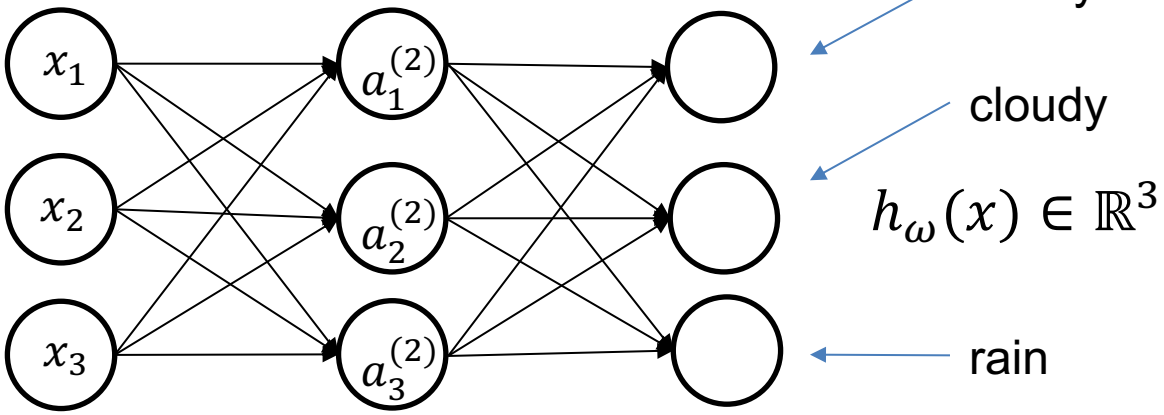
- Output >=3
  - E.g. weather prediction: sunny, cloudy, rain…
  - Multiple output nodes are required



$h_\omega(x) \in \mathbb{R}^3$

Multi-class Classification Problem

- Output >=3
  - E.g. weather prediction: sunny, cloudy, rain…
  - Multiple output nodes are required

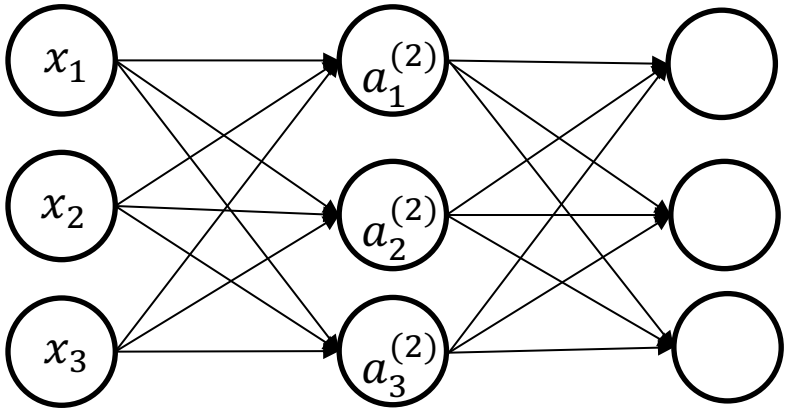


  - The training data is prepared as such
    - $(x_{(1)}, y_1), (x_{(2)}, y_2), \ldots$ Note, y_i is a vector
    - $x_{(i)} = (x_1, x_2, x_3)$, $y_i =$ $\begin{pmatrix}1\\0\\0\end{pmatrix}$, $\begin{pmatrix}0\\1\\0\end{pmatrix}$, or $\begin{pmatrix}0\\0\\1\end{pmatrix}$

sunny cloudy rain

Neural Network: learning and weight optimisation

- Forward propagation
  - During forward propagation through a network, the output (activation) of a given node is a function of its inputs.
  - The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node.
  - Neural network learns using its own (learned) features



$h_\omega(x) \in \mathbb{R}^3$

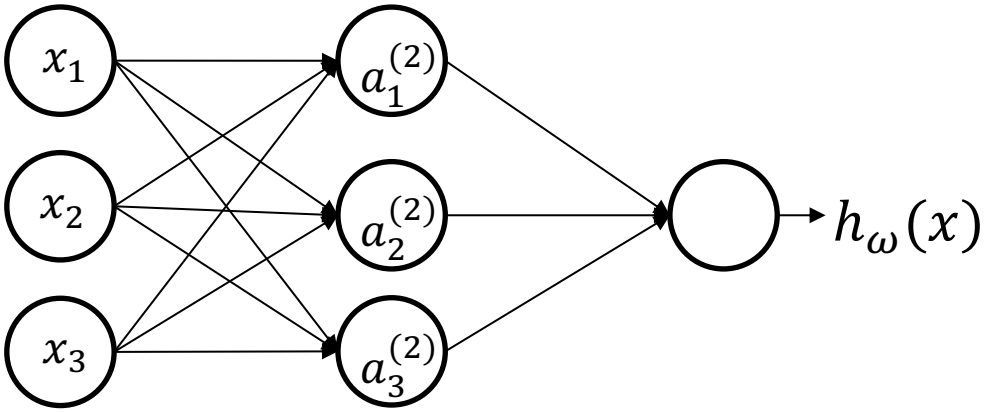
Neural Network: learning and weight optimisation

- Backward propagation
  - **backward propagation of errors**
  - a common method of training NN, in conjunction with an optimisation method, such as **gradient descent**
  - the method calculates the gradient of a loss function with respect to all the weights in the network
  - the gradient is fed to the optimisation method which in turn uses it to **update the weights**, in an attempt to **minimise the loss function**

Neural Network: learning and weight optimisation

- Backward propagation
  - **backward propagation of errors**
  - a common method of training NN, in conjunction with an optimisation method, such as **gradient descent**
  - the method calculates the gradient of a loss function with respect to all the weights in the network
  - the gradient is fed to the optimisation method which in turn uses it to **update the weights**, in an attempt to **minimise the loss function**
  - requirements
    - requires a known, desired output for each input value in order to calculate the loss function gradient
    - requires that the activation function used by the artificial neurons (or "nodes") be differentiable

Forward Propagation (this slide is optional)



$a^{(1)} = x$ $\quad a^{(2)} = g(z^{(2)})$ $\quad a^{(3)} = g(z^{(3)})$

$z^{(j)}$ denotes input to jth layer

$a^{(1)}$: $\quad x = (x_0, x_1, x_2, x_3)^T$

$a^{(2)}$:

$$a_1^{(2)} = g(\Omega_{10}^{(1)} x_0 + \Omega_{11}^{(1)} x_1 + \Omega_{12}^{(1)} x_2 + \Omega_{13}^{(1)} x_3) = g(z_1^{(2)})$$
$$a_2^{(2)} = g(\Omega_{20}^{(1)} x_0 + \Omega_{21}^{(1)} x_1 + \Omega_{22}^{(1)} x_2 + \Omega_{23}^{(1)} x_3) = g(z_2^{(2)})$$
$$a_3^{(2)} = g(\Omega_{30}^{(1)} x_0 + \Omega_{31}^{(1)} x_1 + \Omega_{32}^{(1)} x_2 + \Omega_{33}^{(1)} x_3) = g(z_3^{(2)})$$

$$z^{(2)} = \begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{pmatrix} = \Omega^{(1)} a^{(1)}$$

$a^{(3)}$: $\quad a_1^{(3)} = h_\omega(x) = g(\Omega_{10}^{(2)} a_0^{(2)} + \Omega_{11}^{(2)} a_1^{(2)} + \Omega_{12}^{(2)} a_2^{(2)} + \Omega_{13}^{(2)} a_3^{(2)}) = g(z_1^{(3)})$

$$z^{(3)} = \Omega^{(2)} a^{(2)}$$

Cost Function

- Recall: logistic regression cost function

$$E(\omega) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log h_\omega(x_i) + (1 - y_i)\log(1 - h_\omega(x_i))] + \frac{\lambda}{2N}\sum_{i=1}^{N}\omega_i^2$$

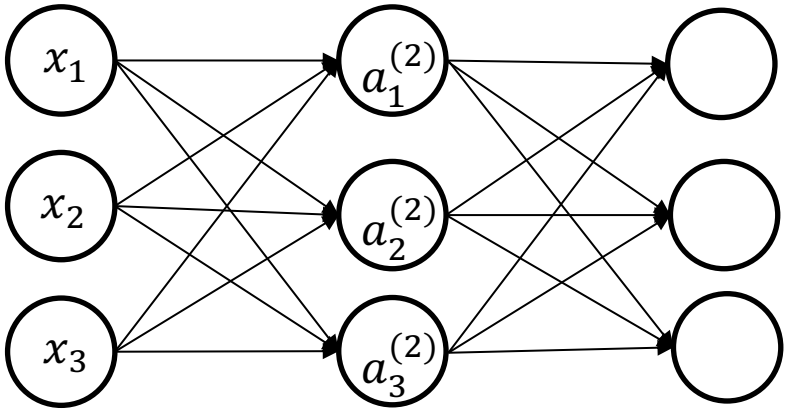
Cost Function

- Recall: logistic regression cost function

$$E(\omega) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log h_\omega(x_i) + (1-y_i)\log(1-h_\omega(x_i))] + \frac{\lambda}{2N}\sum_{i=1}^{N}\omega_i^2$$

- Neural network cost function
  - K is the number of output nodes
  - s_l is number of nodes in layer l

$$E(\Omega) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_{i,k}\log\left(h_\Omega(x_i)\right)_k + \left(1-y_{i,k}\right)\log\left(1-h_\Omega(x_i)\right)_k\right] + \frac{\lambda}{2N}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Omega_{ji}^{(l)})^2$$



Cost Function

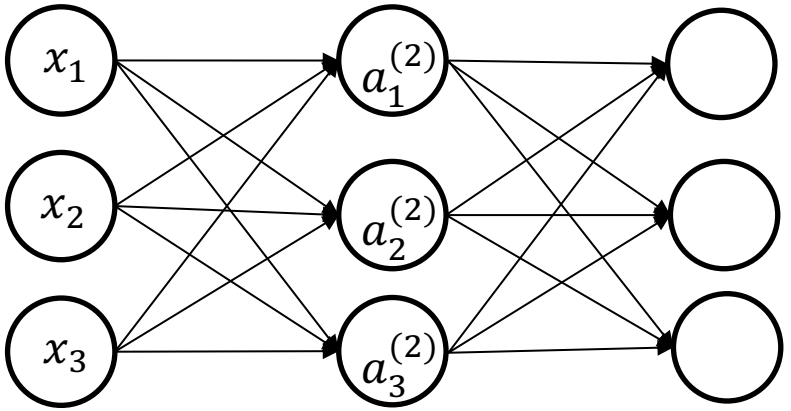
- Recall: logistic regression cost function

$$E(\omega) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log h_\omega(x_i) + (1-y_i)\log(1-h_\omega(x_i))] + \frac{\lambda}{2N}\sum_{i=1}^{m}\omega_i^2$$

- Neural network cost function
  - K is the number of output nodes
  - s_l is number of nodes in layer l

$$E(\Omega) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_{i,k}\log\left(h_\Omega(x_i)\right)_k + \left(1-y_{i,k}\right)\log\left(1-h_\Omega(x_i)\right)_k\right] + \frac{\lambda}{2N}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Omega_{ji}^{(l)})^2$$

- The first term essentially computes the average sum of logistic regression of the output nodes
- The second term is the regularisation term: sum of squares of all weights except bias weights.



Cost Function: optimisation

- Cost function:

$$E(\Omega) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_{i,k}\log\left(h_\Omega(x_i)\right)_k + \left(1-y_{i,k}\right)\log\left(1-h_\Omega(x_i)\right)_k\right] + \frac{\lambda}{2N}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Omega_{ji}^{(l)})^2$$

- Optimisation: $\min_{\Omega} E(\Omega)$

- Gradient descent:
  - Compute the partial derivative w.r.t. each parameter/weight

$$\frac{\partial}{\partial \Omega_{ji}^{(l)}} E(\Omega)$$

    - Note, $h_\Omega(x_i)$ is differentiable (so is $\log\left(h_\Omega(x_i)\right)_k$).

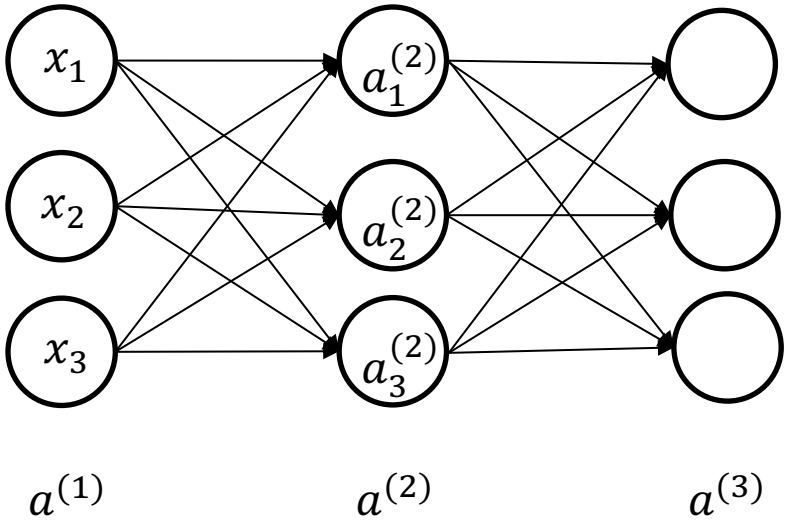
Forward-Backward Propagation (this slide is optional)

- Forward propagation
  - Given one training sample (x, y), using the example NN:

$$a^{(1)} = \left(a_1^{(1)}, a_2^{(1)}, a_3^{(1)}\right)^T = x = (x_1, x_2, x_3)^T$$

$$a^{(2)} = g(\Omega^{(1)} a^{(1)})$$

$$a^{(3)} = g((\Omega^{(2)} a^{(2)}) = h_\Omega(x)$$



$$\Omega^{(1)} = \begin{pmatrix} \Omega_{11}^{(1)}, \Omega_{12}^{(1)}, \Omega_{13}^{(1)} \\ \Omega_{21}^{(1)}, \Omega_{22}^{(1)}, \Omega_{23}^{(1)} \\ \Omega_{31}^{(1)}, \Omega_{32}^{(1)}, \Omega_{33}^{(1)} \end{pmatrix}$$

For simplicity, we omitted bias terms in forward propagation

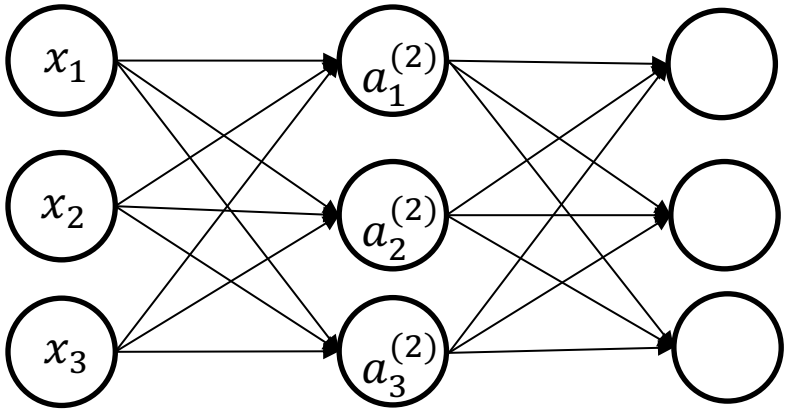
Forward-Backward Propagation (this slide is optional)

- Backward propagation of errors
  - Given one training sample (x, y), using the example NN:
  - Error is propagated from the last layer of the NN to the input layer

$$\delta_1^{(3)} = a_1^{(3)} - y_1$$
$$\delta_2^{(3)} = a_2^{(3)} - y_2$$
$$\delta_3^{(3)} = a_3^{(3)} - y_3$$

Output layer

$$\delta^{(3)} = a^{(3)} - y$$



$$\delta^{(2)} = (\Omega^{(2)})^T \delta^{(3)} \circ g'(z^{(2)})$$

$z^{(2)}$ denotes input to the 2nd layer ($\Omega^{(1)} a^{(1)}$)

How about $\delta^{(1)}$?

$$\Omega^{(2)} = \begin{pmatrix} \Omega_{11}^{(2)}, \Omega_{12}^{(2)}, \Omega_{13}^{(2)} \\ \Omega_{21}^{(2)}, \Omega_{22}^{(2)}, \Omega_{23}^{(2)} \\ \Omega_{31}^{(2)}, \Omega_{32}^{(2)}, \Omega_{33}^{(2)} \end{pmatrix}$$

Weight Initialisation

- The weights of the neural network needs to be initialised
  - Training samples are then propagated forward and errors (differences between predictions and groundtruth) are propagated backward
  - This is an iterative process
- The logistic function is the same for each node
  - Its derivative is thus the same

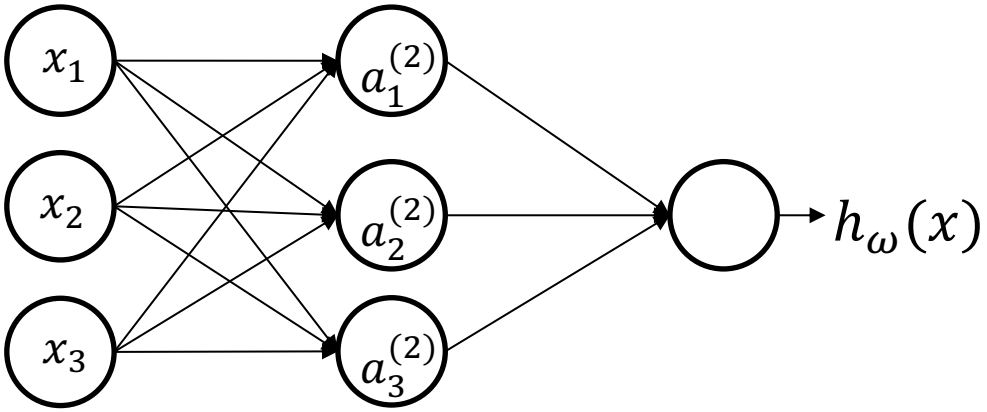
Weight Initialisation

- The weights of the neural network needs to be initialised
  - Training samples are then propagated forward and errors (differences between predictions and groundtruth) are propagated backward
  - This is an iterative process
- The logistic function is the same for each node
  - Its derivative is thus the same
- Zero initialisation
  - All weights all set zero to start with
  - This however is problematic, since all the nodes will behave exactly the same

Weight Initialisation

- The weights of the neural network needs to be initialised
  - Training samples are then propagated forward and errors (differences between predictions and groundtruth) are propagated backward
  - This is an iterative process
- The logistic function is the same for each node
  - Its derivative is thus the same
- Zero initialisation
  - All weights all set zero to start with
  - This however is problematic, since all the nodes will behave exactly the same
- Random initialisation
  - This breaks the symmetry and every node computes differently

RECAP: Forward Propagation



$a^{(1)} = x$ $\quad a^{(2)} = g(z^{(2)})$ $\quad a^{(3)} = g(z^{(3)})$

$z^{(j)}$ denotes input to jth layer

$a^{(1)}$: $\quad x = (x_0, x_1, x_2, x_3)^T$

$a^{(2)}$:

$$a_1^{(2)} = g(\Omega_{10}^{(1)} x_0 + \Omega_{11}^{(1)} x_1 + \Omega_{12}^{(1)} x_2 + \Omega_{13}^{(1)} x_3) = g(z_1^{(2)})$$
$$a_2^{(2)} = g(\Omega_{20}^{(1)} x_0 + \Omega_{21}^{(1)} x_1 + \Omega_{22}^{(1)} x_2 + \Omega_{23}^{(1)} x_3) = g(z_2^{(2)})$$
$$a_3^{(2)} = g(\Omega_{30}^{(1)} x_0 + \Omega_{31}^{(1)} x_1 + \Omega_{32}^{(1)} x_2 + \Omega_{33}^{(1)} x_3) = g(z_3^{(2)})$$

$$z^{(2)} = \begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{pmatrix} = \Omega^{(1)} a^{(1)}$$

$a^{(3)}$: $\quad a_1^{(3)} = h_\omega(x) = g(\Omega_{10}^{(2)} a_0^{(2)} + \Omega_{11}^{(2)} a_1^{(2)} + \Omega_{12}^{(2)} a_2^{(2)} + \Omega_{13}^{(2)} a_3^{(2)}) = g(z_1^{(3)})$

$$z^{(3)} = \Omega^{(2)} a^{(2)}$$

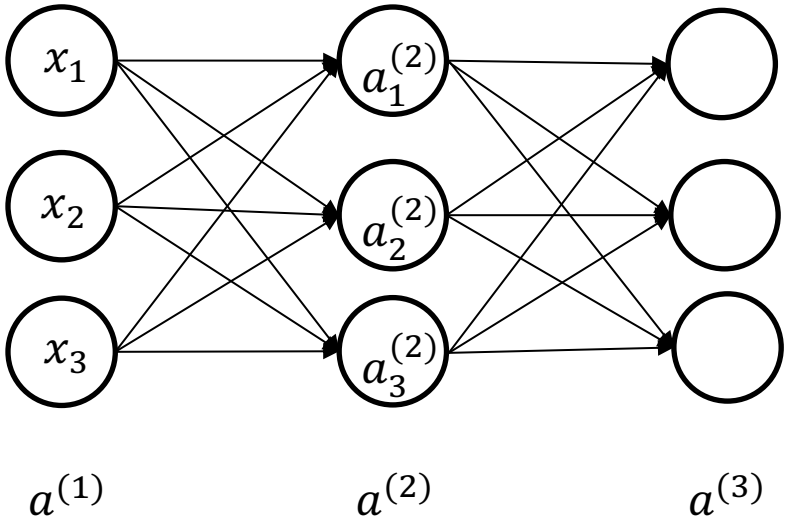
RECAP: Forward-Backward Propagation

- Backward propagation of errors
  - Given one training sample (x, y), using the example NN:
  - Error is propagated from the last layer of the NN to the input layer

$$\delta_1^{(3)} = a_1^{(3)} - y_1$$
$$\delta_2^{(3)} = a_2^{(3)} - y_2$$
$$\delta_3^{(3)} = a_3^{(3)} - y_3$$

Output layer

$$\delta^{(3)} = a^{(3)} - y$$

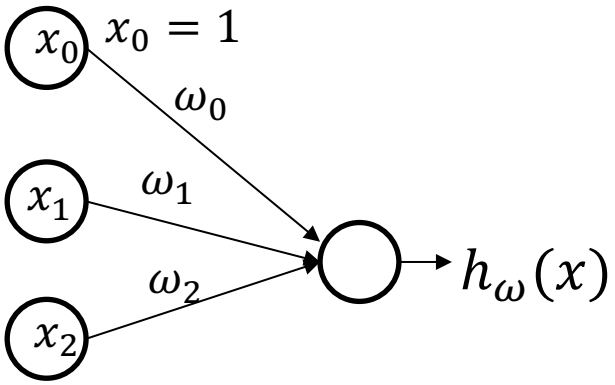


$$\delta^{(2)} = (\Omega^{(2)})^T \delta^{(3)} \circ g'(z^{(2)})$$

$z^{(2)}$ denotes input to the 2nd layer ($\Omega^{(1)} a^{(1)}$)

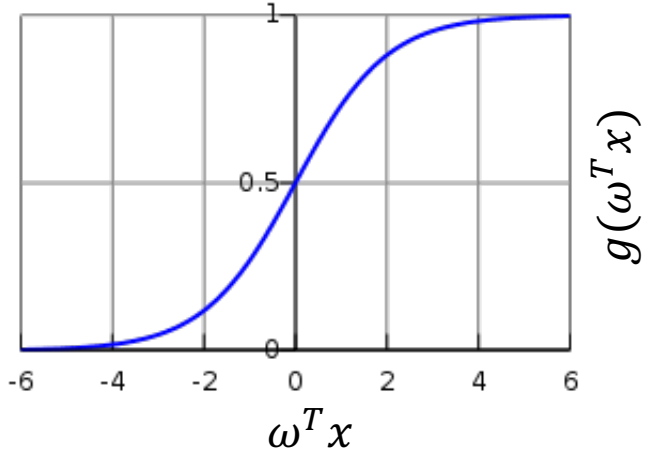
Approximate Logical Operators

- Logical AND
  - "A and B" is true only if A is true and B is true
  - x is binary input {0,1}



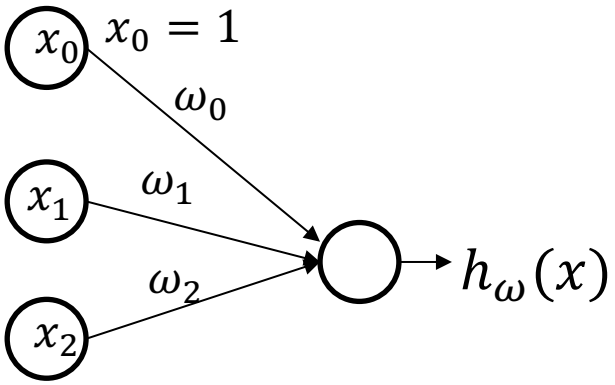
x_1	x_2	x_1 AND x_2
1	1	1
1	0	0
0	1	0
0	0	0

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$



Approximate Logical Operators

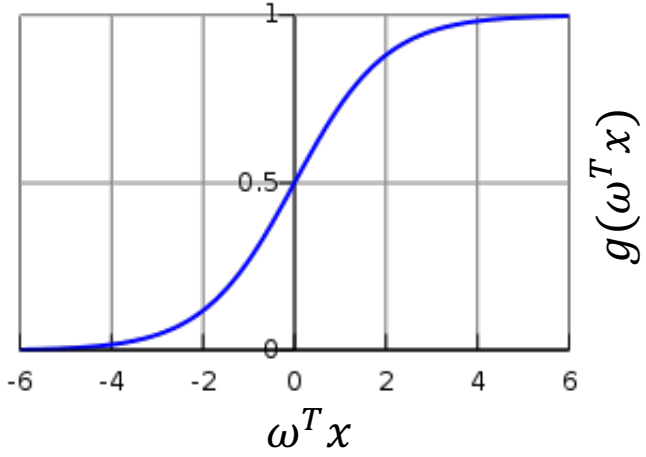
- Logical AND
  - "A and B" is true only if A is true and B is true
  - x is binary input {0,1}



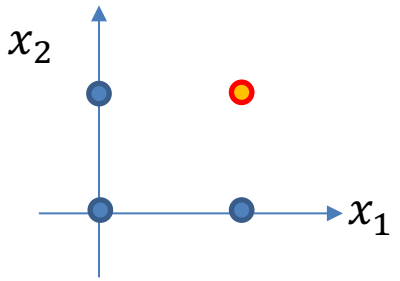
x_1	x_2	x_1 AND x_2
1	1	1
1	0	0
0	1	0
0	0	0

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

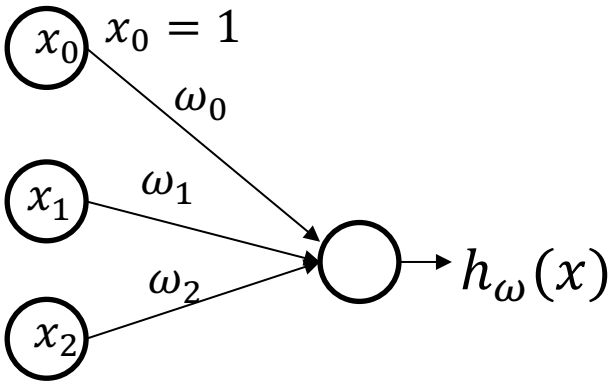
e.g. $\omega = (-30, 20, 20)^T$



Approximate Logical Operators



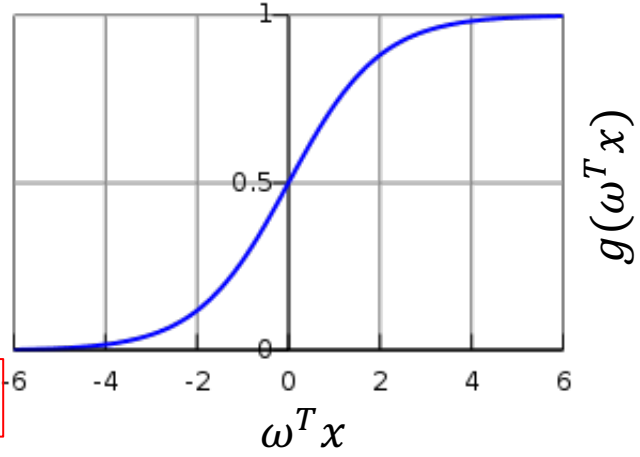
- Logical AND
  - "A and B" is true only if A is true and B is true
  - x is binary input {0,1}



x_1	x_2	x_1 AND x_2
1	1	1
1	0	0
0	1	0
0	0	0

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

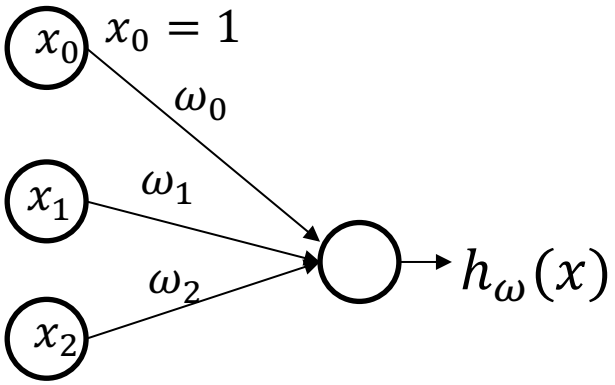
e.g. $\omega = (-30, 20, 20)^T$



large negative weight for bias, equal weight for input

Approximate Logical Operators

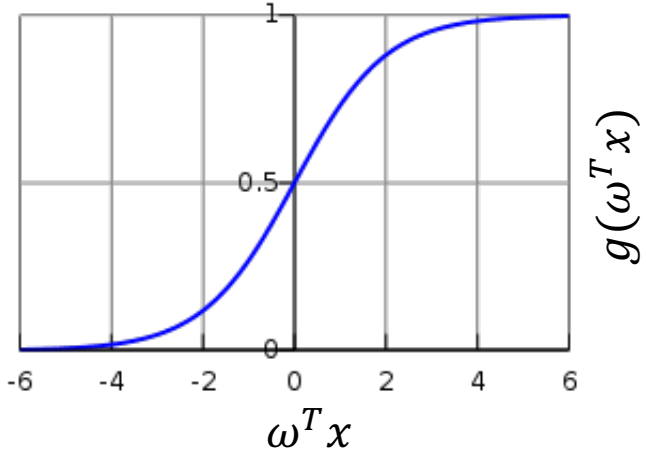
- Logical OR
  - "A or B" is true if A is true, or if B is true, or if both A and B are true
  - x is binary input {0,1}



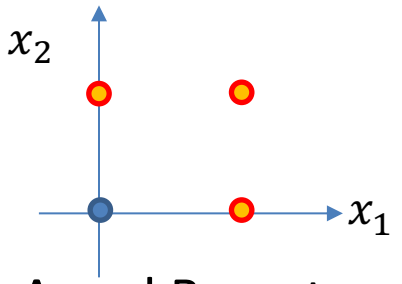
x_1	x_2	x_1 OR x_2
1	1	1
1	0	1
0	1	1
0	0	0

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

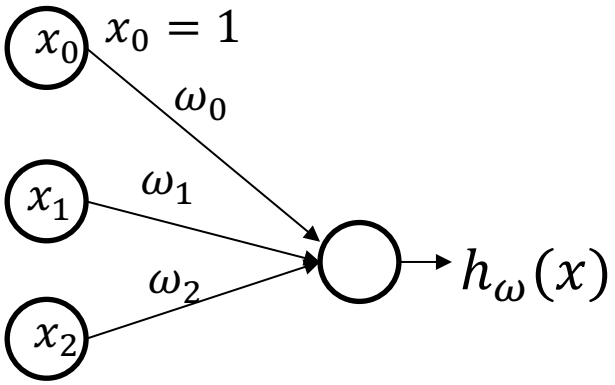
e.g. $\omega = (-10, 20, 20)^T$



Approximate Logical Operators



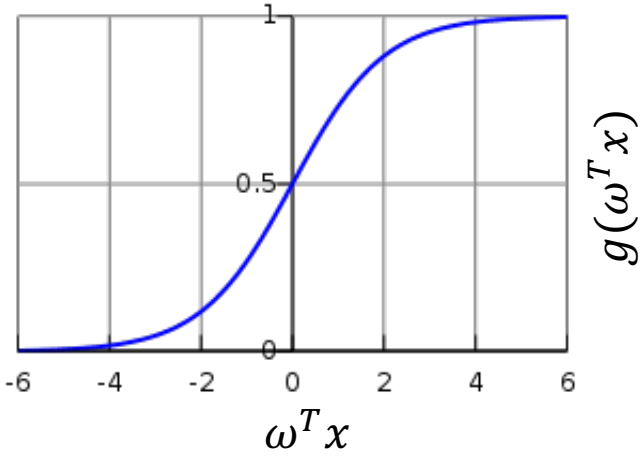
- Logical OR
  - "A or B" is true if A is true, or if B is true, or if both A and B are true
  - x is binary input {0,1}



x_1	x_2	x_1 OR x_2
1	1	1
1	0	1
0	1	1
0	0	0

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

e.g. $\omega = (-10, 20, 20)^T$

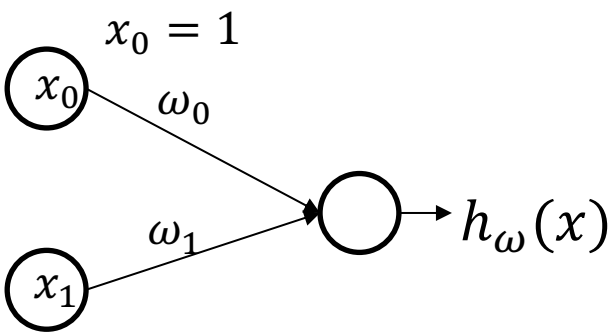


negative weight for bias, large equal weight for input

Approximate Logical Operators

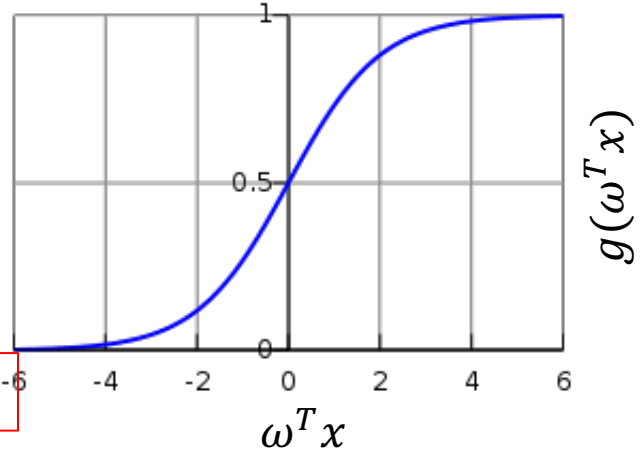
- Negate NOT
  - "NOT A" is true if A is false
  - x is binary input {0,1}

x_1	NOT x_1
1	0
0	1



$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1)$$

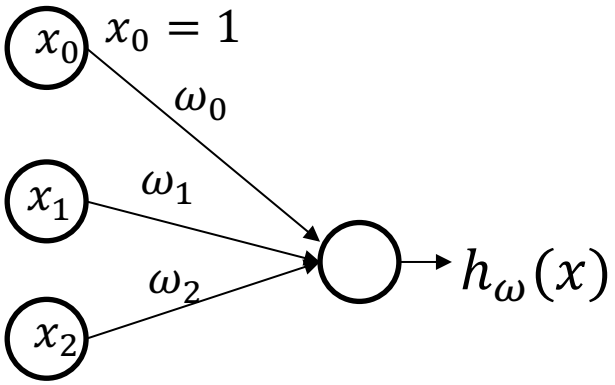
e.g. $\omega = (10, -20)^T$



positive weight for bias, large negative weight for input

Approximate Logical Operators

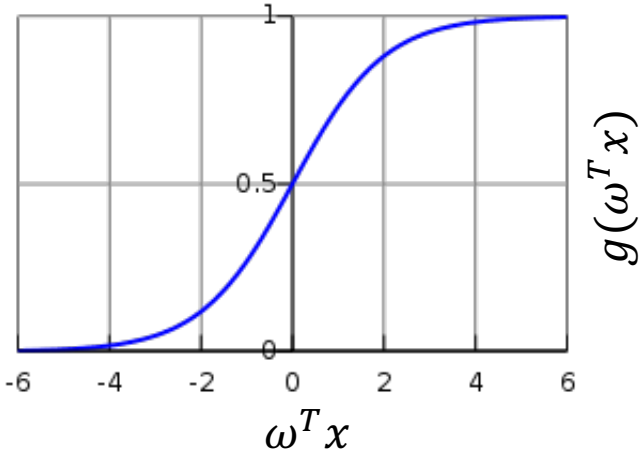
- (NOT x_1) AND (NOT x_2)
  - Only true if both x1 and x2 are false
  - x is binary input {0,1}



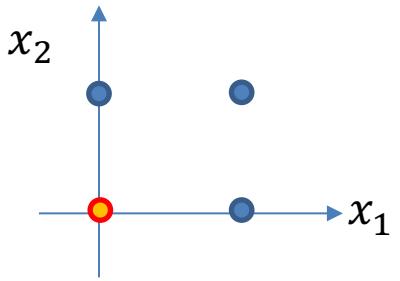
x_1	x_2	(NOT x_1) AND (NOT x_2)
1	1	0
1	0	0
0	1	0
0	0	1

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

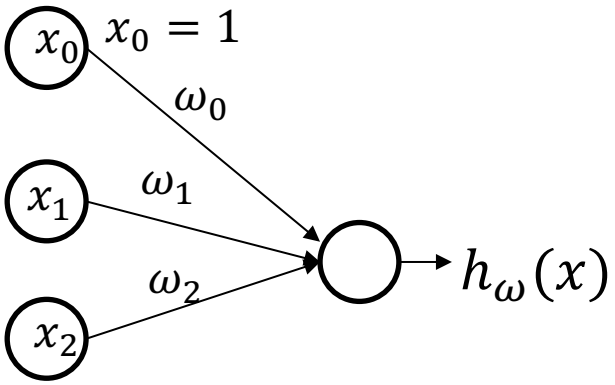
e.g. $\omega = (10, -20, -20)^T$



Approximate Logical Operators



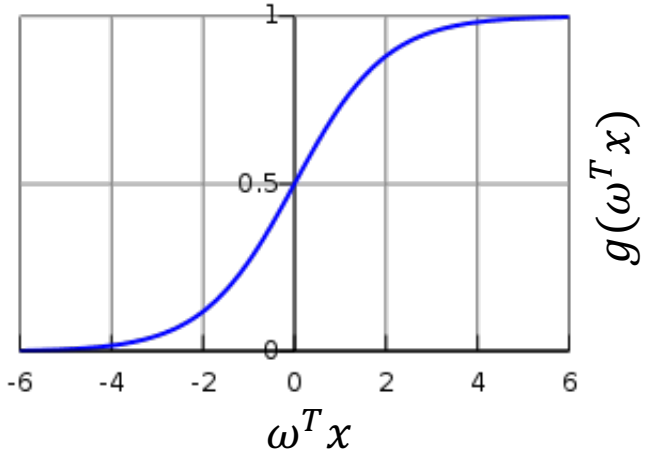
- (NOT x_1) AND (NOT x_2)
  - Only true if both x1 and x2 are false
  - x is binary input {0,1}



x_1	x_2	(NOT x_1) AND (NOT x_2)
1	1	0
1	0	0
0	1	0
0	0	1

$$h_\omega(x) = g(\omega^T x) = g(\omega_0 + \omega_1 x_1 + \omega_2 x_2)$$

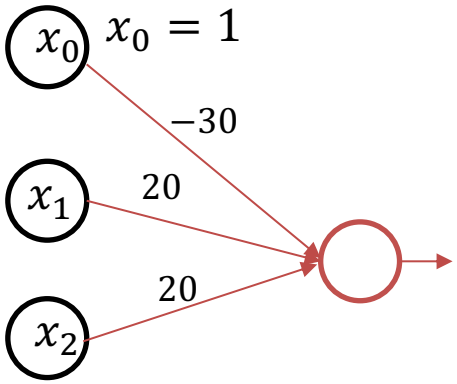
e.g. $\omega = (10, -20, -20)^T$



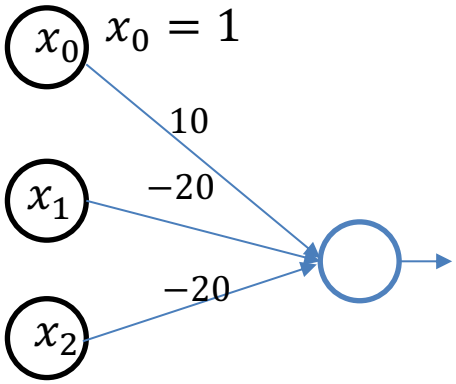
This is the same as NOT(x_1 OR x_2); hence the weights are the negate of OR.

Approximate Logical Operators

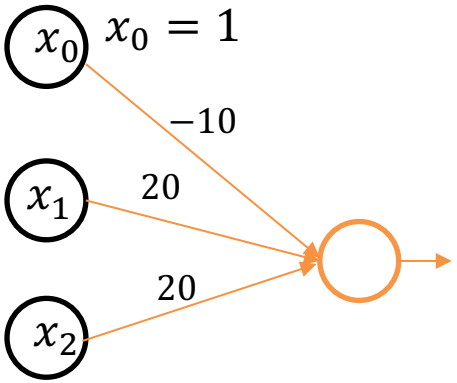
- x_1 XNOR x_2
  - (x_1 AND x_2) OR ((NOT x_1) AND (NOT x_2))
  - x is binary input {0,1}



x_1 AND x_2



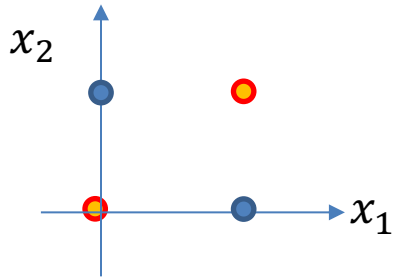
(NOT x_1) AND (NOT x_2)



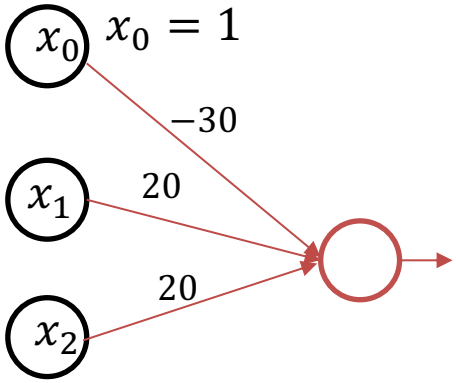
x_1 OR x_2

x_1	x_2	x_1 AND x_2	(NOT x_1) AND (NOT x_2)	x_1 XNOR x_2
1	1	1	0	1
1	0	0	0	0
0	1	0	0	0
0	0	0	1	1

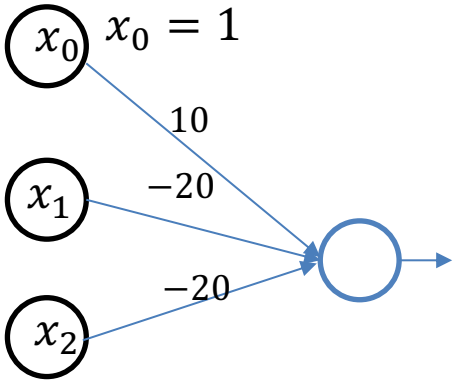
Approximate Logical Operators



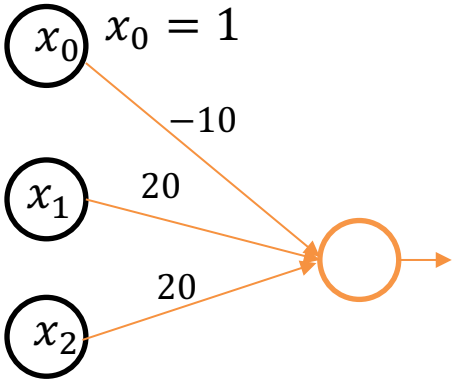
- x_1 XNOR x_2
  - (x_1 AND x_2) OR ((NOT x_1) AND (NOT x_2))
  - x is binary input {0,1}



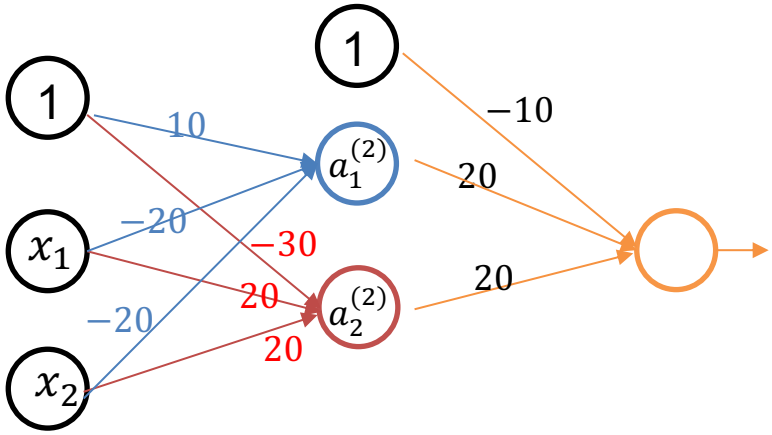
x_1 AND x_2



(NOT x_1) AND (NOT x_2)



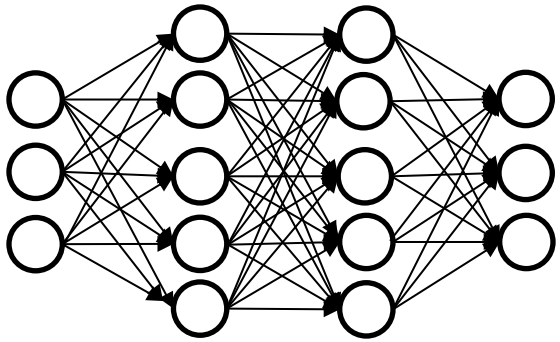
x_1 OR x_2



x_1	x_2	x_1 AND x_2	(NOT x_1) AND (NOT x_2)	x_1 XNOR x_2
1	1	1	0	1
1	0	0	0	0
0	1	0	0	0
0	0	0	1	1

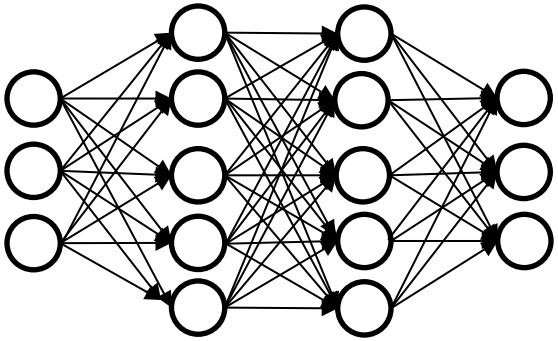
Neural Network Architecture

- NN architecture refers to the connectivity patterns between neurons
  - Number of input nodes
    - Depends on the dimensionality of the input data (feature)
  - Number of output nodes
    - Depends on the number of classes in the case of classification



Neural Network Architecture cont.

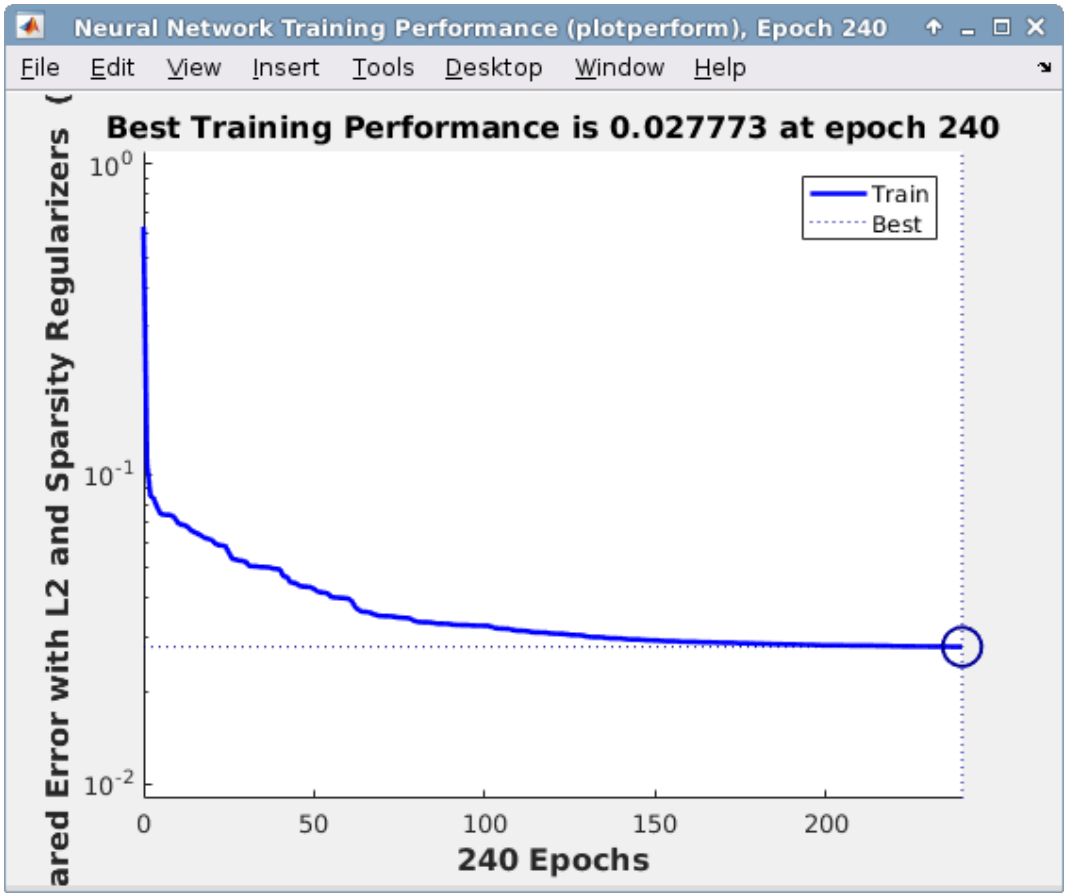
- NN architecture refers to the connectivity patterns between neurons
  - Number of hidden layers
    - This requires careful consideration (and sometimes trial and error)
    - >1 hidden layer: *may* be considered as deep NN
    - >10 hidden layers: very deep NN
    - Generally, more hidden layers more powerful NN becomes; also more computationally expensive and more difficult to train



  - Here, we are only concerned with feed-forward NN
    - There is no loop
  - There are however other types of NN, e.g. recurrent NN

Neural Network Training

- Training errors are expected to decrease through iterative forward- and backward- propagations.
  - Can take some time to converge



Batch Gradient Descent

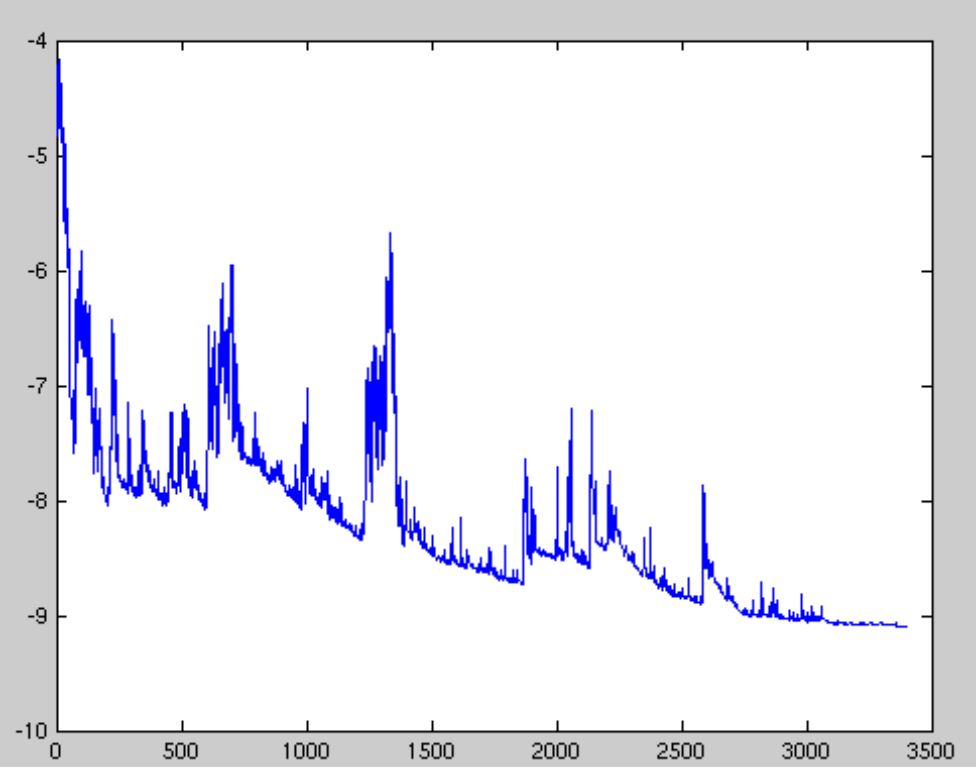
- Take logistic regression as an example:
  - Cost function:

$$E(\omega) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log h_\omega(x_i) + (1-y_i)\log(1-h_\omega(x_i))] + \frac{\lambda}{2N}\omega^T\omega$$

  - To find the parameters: $\min_{\omega} E(\omega)$
  - Gradient descent: $\omega_j := \omega_j - \alpha \frac{\partial}{\partial \omega_j} E(\omega)$
    - This is an iterative process
    - For each iteration, the partial derivative calculation $\frac{\partial}{\partial \omega_j}$ involves enumeration across the whole dataset (N samples)
    - And thus this type of gradient descent is referred to as **batch gradient descent**
    - This can be computationally expensive, when N is large.

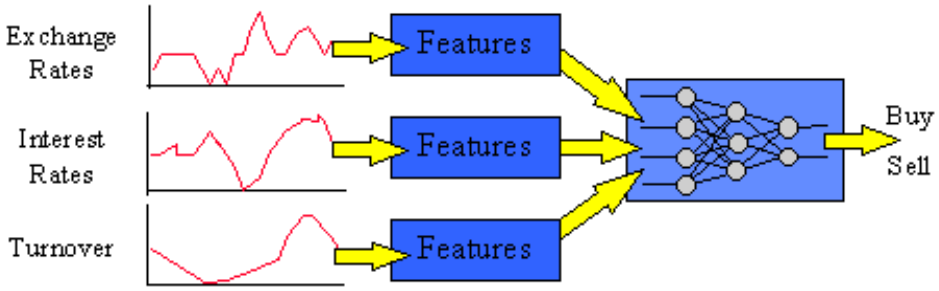
Stochastic Gradient Descent

- Instead of summing over all data points, which is computationally expensive in dealing with large data sets
  - First, randomise the dataset through shuffling
  - Perform gradient descent for each data sample
    - Sometimes, small subset of data samples



Example Applications

- Stock market prediction
  - Predict trend based on key indicators
  - Case study:



A major Japanese securities company used neural computing in order to develop better prediction models. A neural network was trained on 33 months' worth of historical data. This data contained a variety of economic indicators such as turnover, previous share values, interest rates and exchange rates. The network was able to learn the complex relations between the indicators and how they contribute to the overall prediction. Once trained it was then in a position to make predictions based on "live" economic indicators.

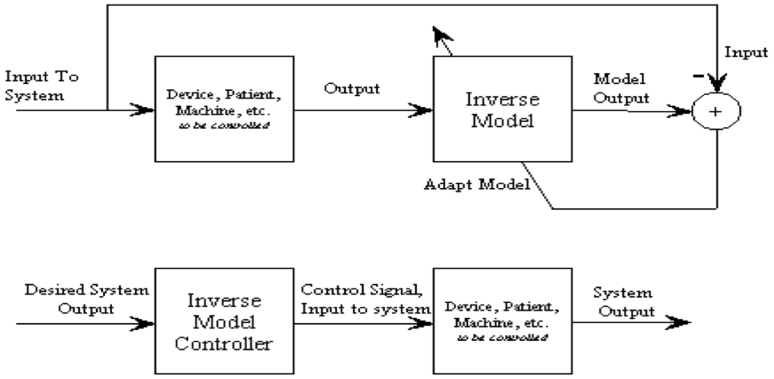
The neural network-based system is able to make faster and more accurate predictions than before. It is also more flexible since it can be retrained at any time in order to accommodate changes in stock market trading conditions. Overall the system outperforms statistical methods by a factor of 19%. The system can therefore make a considerable difference on returns.

- Same methodology can be applied to
  - Predict sales and consumer trends
  - Utility supply and demand
  - Predict yield (e.g. farming)
  - Predict trajectory (e.g. object tracking)

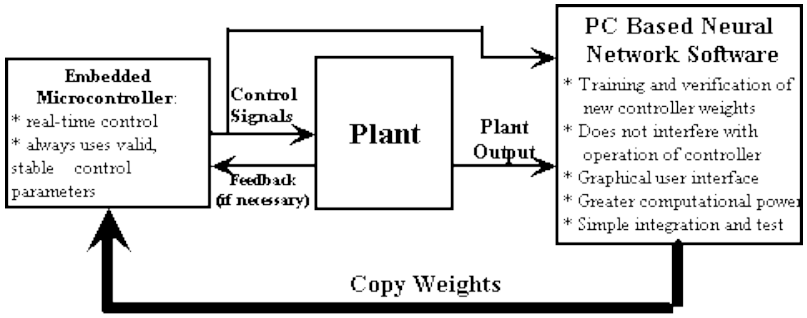
Several of the case studies are from: George Papadourakis

Example Applications

- Adaptive inverse control
  - Adaptive control (e.g. automated systems, including vehicles, robots and plants)
  - Case study:



Block diagram for neural network adaptive control

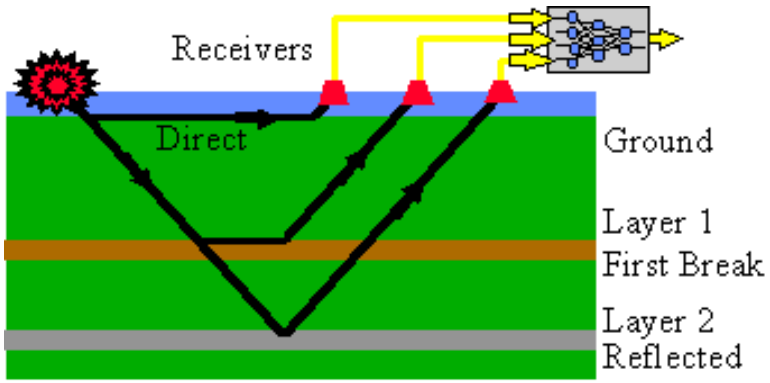


A machine learning system for adaptive control

NNs can be used in adaptive control applications. The top block diagram shows the training of the inverse model. Essentially, the neural network is learning to recreate the input that created the current output of the plant. Once properly trained, the inverse model (which is another NN) can be used to control the plant since it can create the necessary control signals to create the desired system output.

Example Applications

- Oil exploitation
  - Detect the right signal
  - Case study:



A neural network was trained on a set of traces selected from a representative set of seismic records, each of which had their first break signals highlighted by an expert.

The vast quantities of seismic data involved are cluttered with noise and are highly dependent on the location being investigated. Classical statistical analysis techniques lose their effectiveness when the data is noisy and comes from an environment not previously encountered. Even a small improvement in correctly identifying first break signals could result in a considerable return on investment.

The neural network achieves better than 95 % accuracy, easily outperforming existing manual and computer-based methods. As well as being more accurate, the system also achieves an 88% improvement in the time taken to identify first break signals. Considerable cost savings have been made as a result.

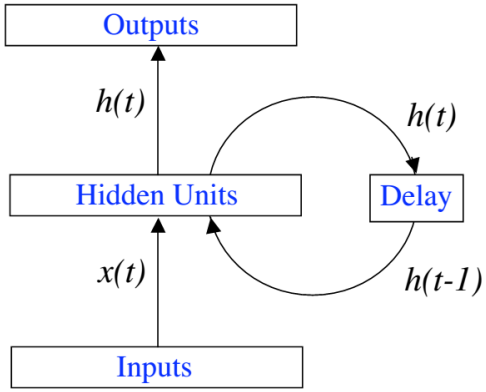
- This can also applied to:
  - Medical scanner signal recovery
  - Radio astronomy signal analysis
  - Defence radar signal analysis

Example Applications

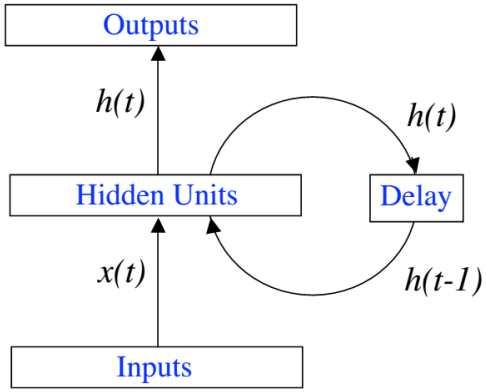
- There are many many more
  - Chemical manufacturing (getting the right mixture)
  - Environmental control
  - Power demand analysis
  - Automated inspection (detecting defects)
  - Speech analysis (e.g. recognition)
  - Machine translation
  - …

Recurrent NN Example

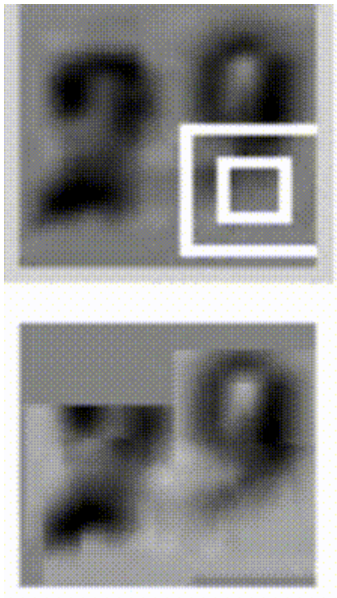
- R-NN
  - Multilayer NN with previous hidden unit activations feedin back into the network along with the input
  - Dynamically learn the context it needs to solve the probler
  - Has a “memory” which captures information about what has been calculated so far



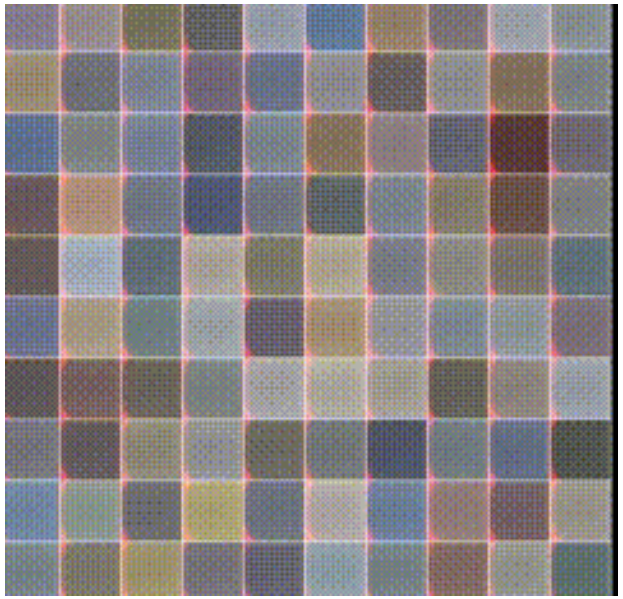
Recurrent NN Example



- R-NN
  - Multilayer NN with previous hidden unit activations feedin back into the network along with the input
  - Dynamically learn the context it needs to solve the probler
  - Has a “memory” which captures information about what has been calculated so far
- Example: predict the next letter to write a document
  - Sequential processing in absence of sequences



R-NN learns to read house numbers



R-NN learns to paint house numbers