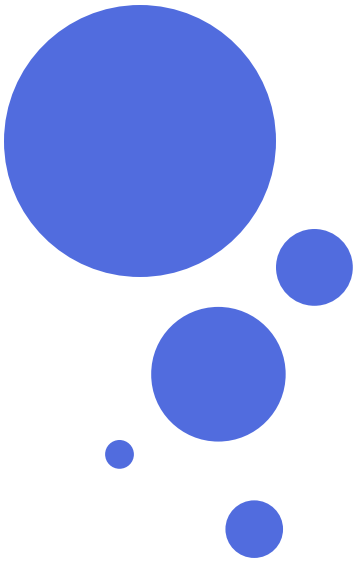

Rensselaer

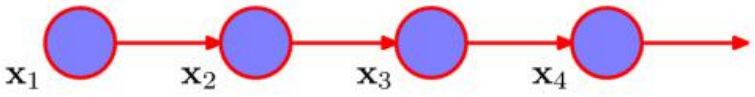


# Lecture 16: Introduction to Neural Networks, Feed-forward Networks and Back-propagation

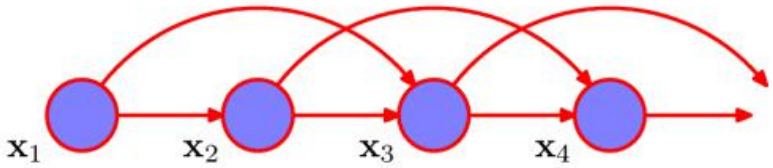
Dr. Chengjiang Long
Computer Vision Researcher at Kitware Inc.
Adjunct Professor at RPI.
Email: **longc3@rpi.edu**

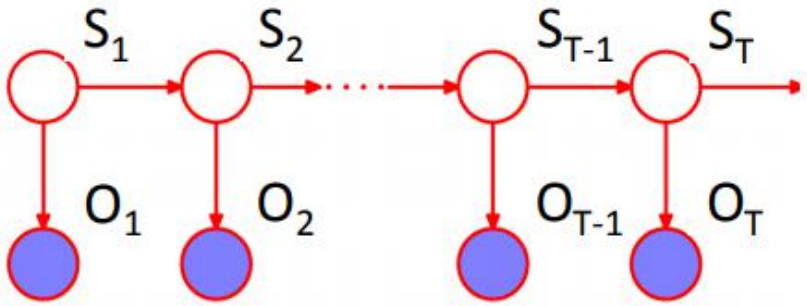
# Recap Previous Lecture

$$p(\mathbf{X}) = \prod_{i=1}^{n} p(X_n|X_{n-1})$$




$$p(\mathbf{X}) = \prod_{i=1}^{n} p(X_n|X_{n-1}, X_{n-2})$$




$$p(\{S_t\}_{t=1}^{T}, \{O_t\}_{t=1}^{T}) = p(S_1) \prod_{t=2}^{T} p(S_t|S_{t-1}) \prod_{t=1}^{T} p(O_t|S_t)$$

# Outline

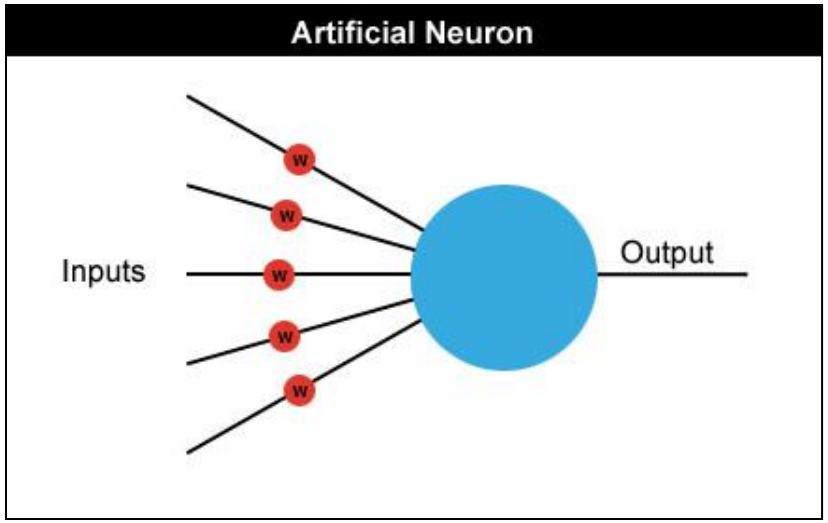
- Introduction to Nueral Networks
- Feed-Forward Networks
  - Single-layer Perceptron (SLP)
  - Multi-layer Perceptron (MLP)
- Back-propagation Learning

# Outline

- **Introduction to Nueral Networks**
- Feed-Forward Networks
  - Single-layer Perceptron (SLP)
  - Multi-layer Perceptron (MLP)
- Back-propagation Learning

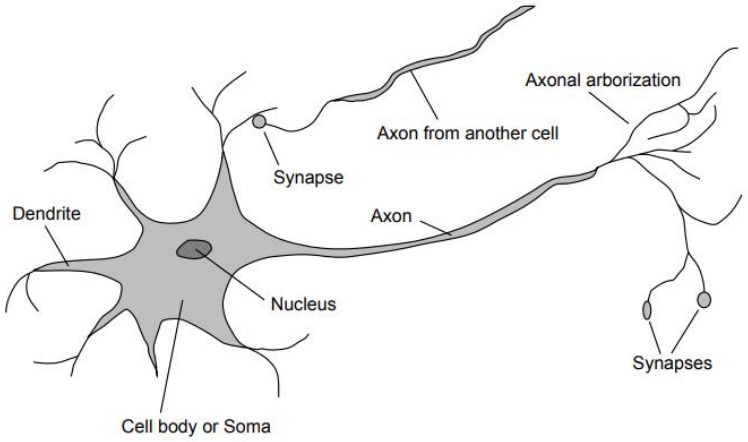
# Neural networks

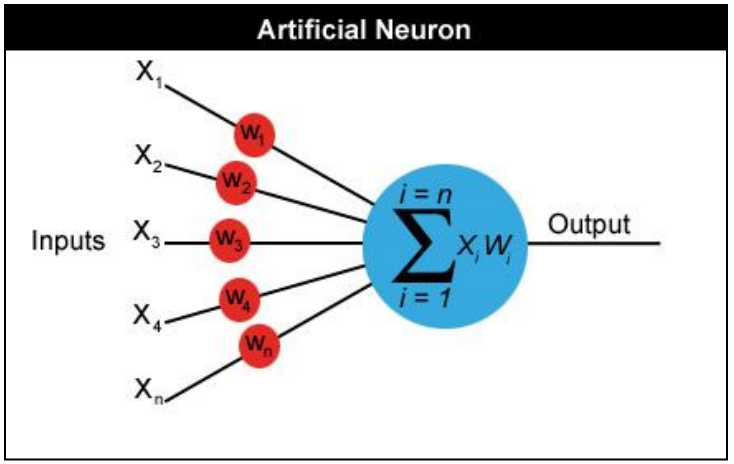
- Neural networks are made up of many artificial neurons.
- Each input into the neuron has its own weight associated with it illustrated by the red circle.
- A weight is simply a floating point number and it's these we adjust when we eventually come to train the network.



# Neural networks

- A neuron can have any number of inputs from one to n, where n is the total number of inputs.
- The inputs may be represented therefore as $x_1, x_2, x_3 \ldots x_n$.
- And the corresponding weights for the inputs as $w_1, w_2, w_3 \ldots w_n$.
- Output $a = x_1w_1 + x_2w_2 + x_3w_3 \ldots + x_nw_n$

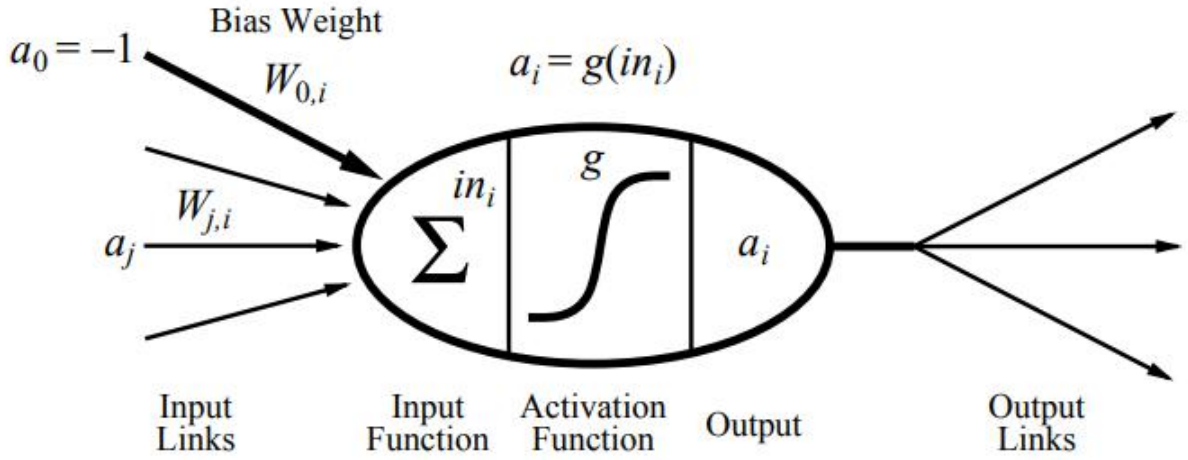


# McCulloch–Pitts "unit"

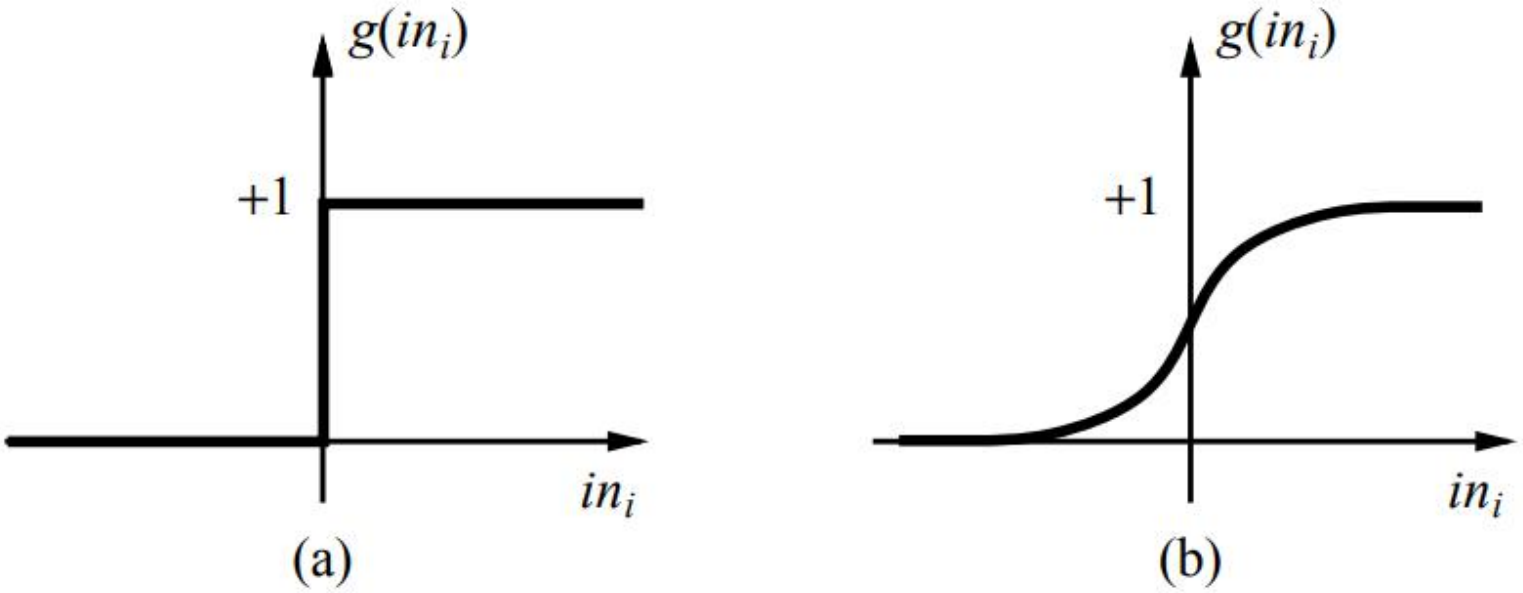
- Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$




- A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

# Activation functions




(a) is a step function or threshold function

(b) is a sigmoid function $1/(1+e^{-x})$

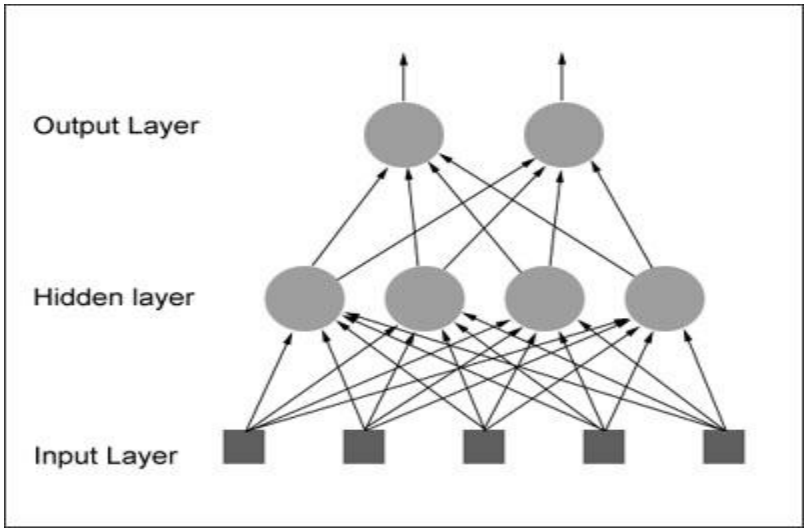
Changing the bias weight $W_{0,i}$ moves the threshold location

# Neural Networks by an Example

- let's design a neural network that will detect the number '4'.
- Given a panel made up of a grid of lights which can be either on or off, we want our neural net to let us know whenever it thinks it sees the character '4'.
- The panel is eight cells square and looks like this:
- the neural net will have 64 inputs, each one representing a particular cell in the panel and a hidden layer consisting of a number of neurons (more on this later) all feeding their output into just one neuron in the output layer

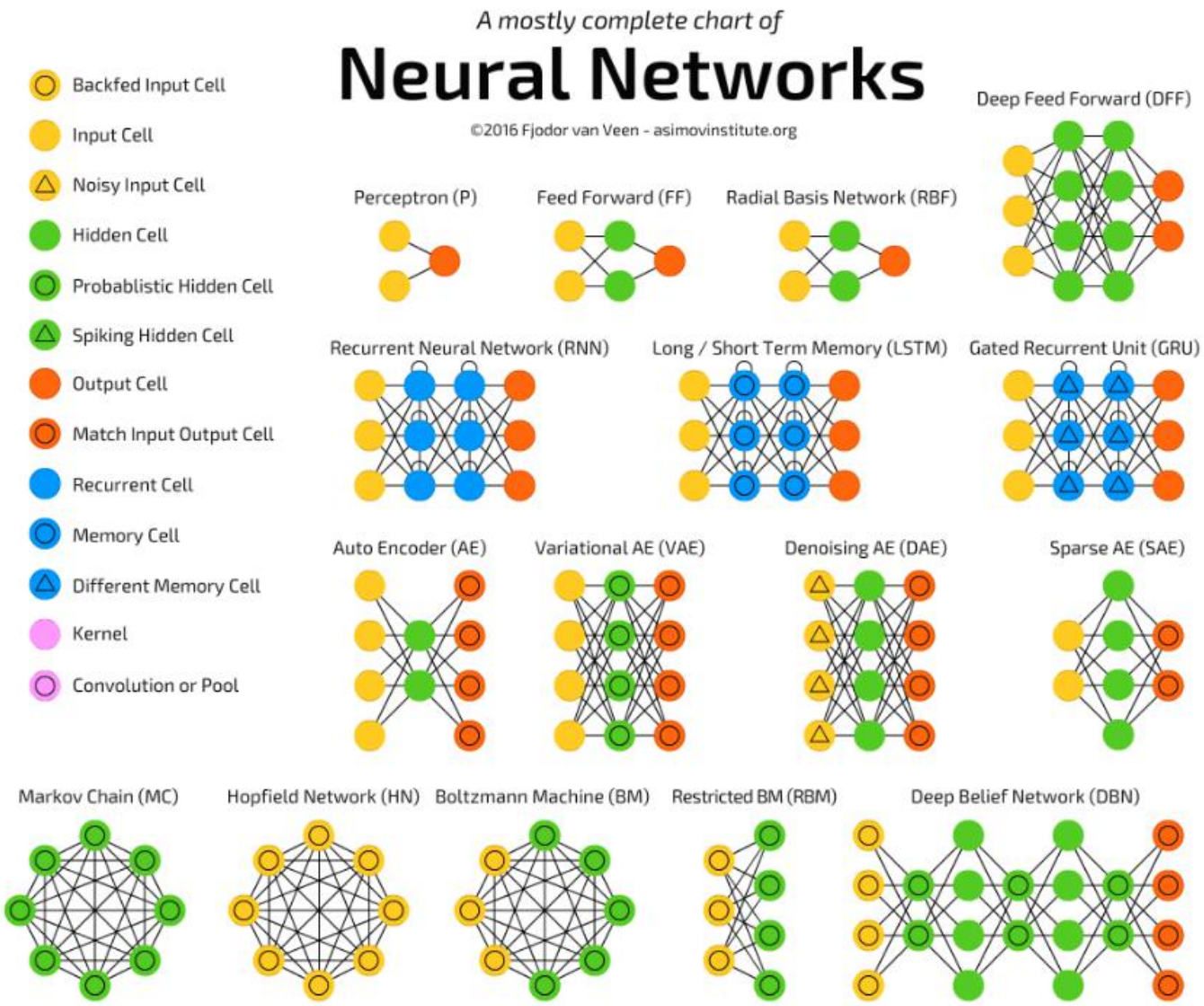
# Neural Networks by an Example

- Initialize the neural net with random weights
- Feed it a series of inputs which represent, in this example, the different panel configurations
- For each configuration we check to see what its output is and adjust the weights accordingly so that whenever it sees something looking like a number 4 it outputs a 1 and for everything else it outputs a zero.

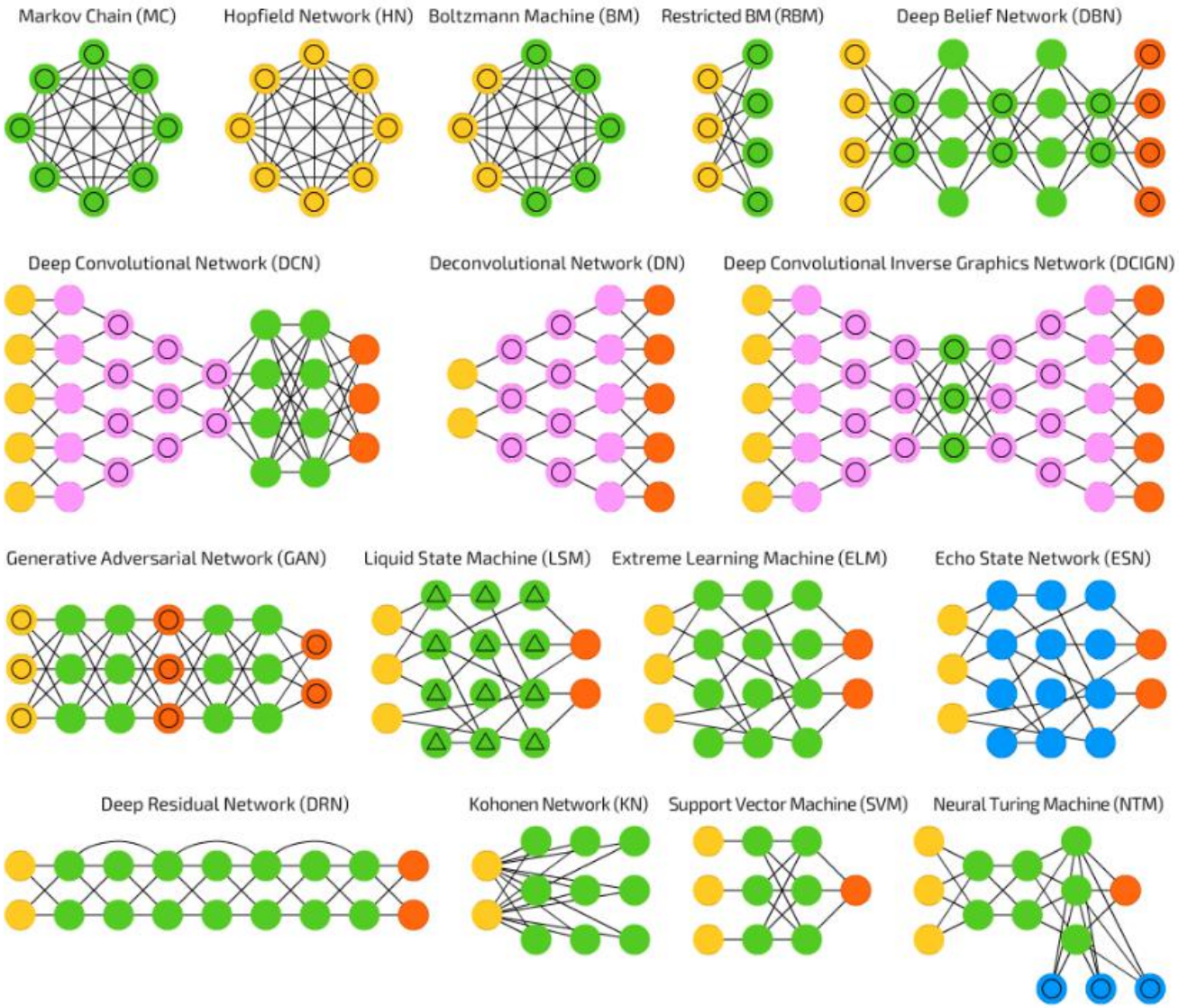



http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

# A Cheat Sheet of Neural Network Architectures



# A Cheat Sheet of Neural Network Architectures

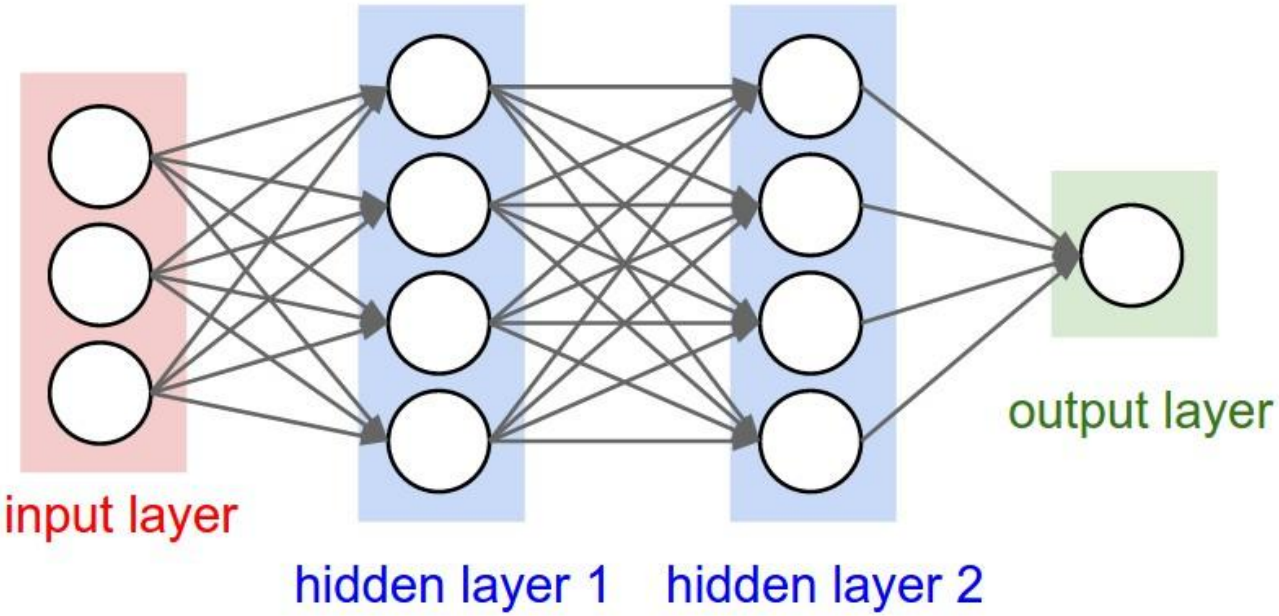


# Outline

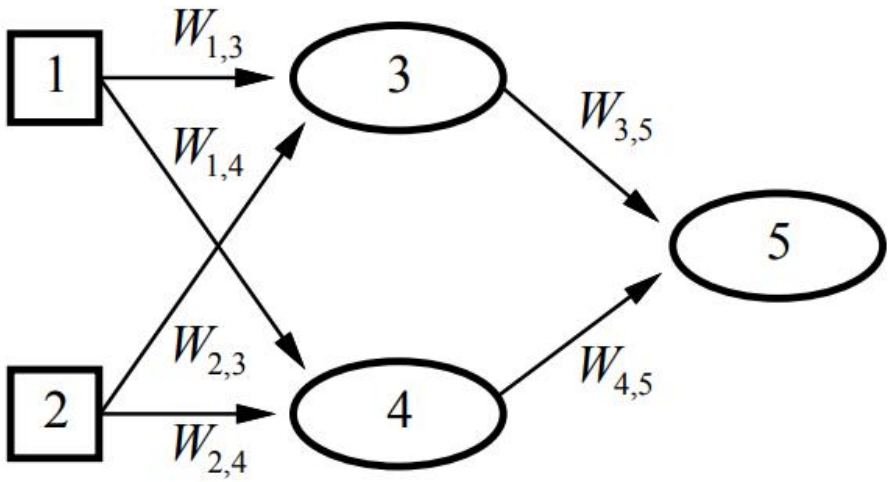
- Introduction to Nueral Networks
- **Feed-Forward Networks**
  - Single-layer Perceptron (SLP)
  - Multi-layer Perceptron (MLP)
- Back-propagation Learning

# Network structures

- Feed–forward networks:
  - single–layer perceptrons
  - multi–layer perceptrons
- Feed–forward networks implement functions, have no internal state



# Feed-forward example




- Feed–forward network = a parameterized family of nonlinear functions:

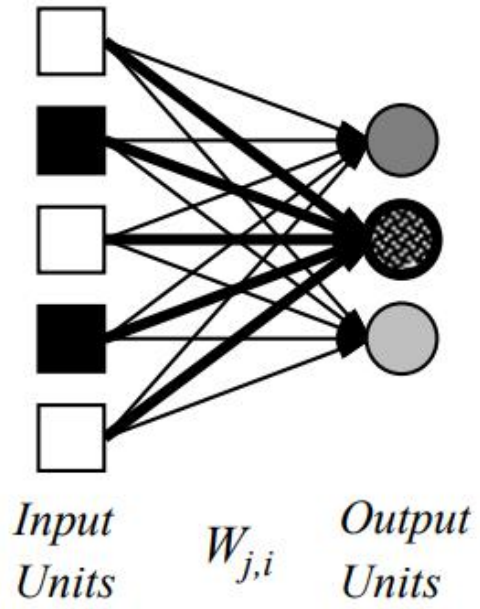
$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

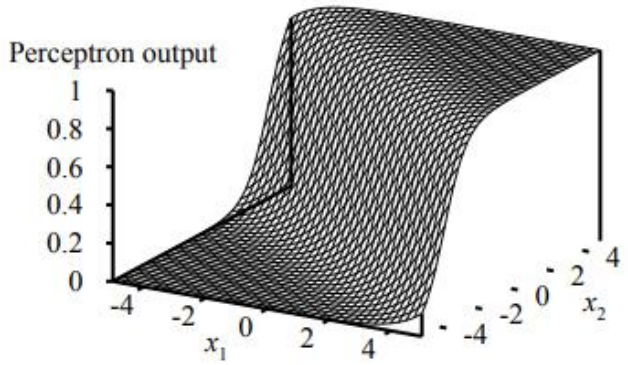
- Adjusting weights changes the function: do learning this way!

# Outline

- Introduction to Nueral Networks
- Feed-Forward Networks
  - **Single-layer Perceptron (SLP)**
  - Multi-layer Perceptron (MLP)
- Back-propagation Learning

# Single-layer perceptrons

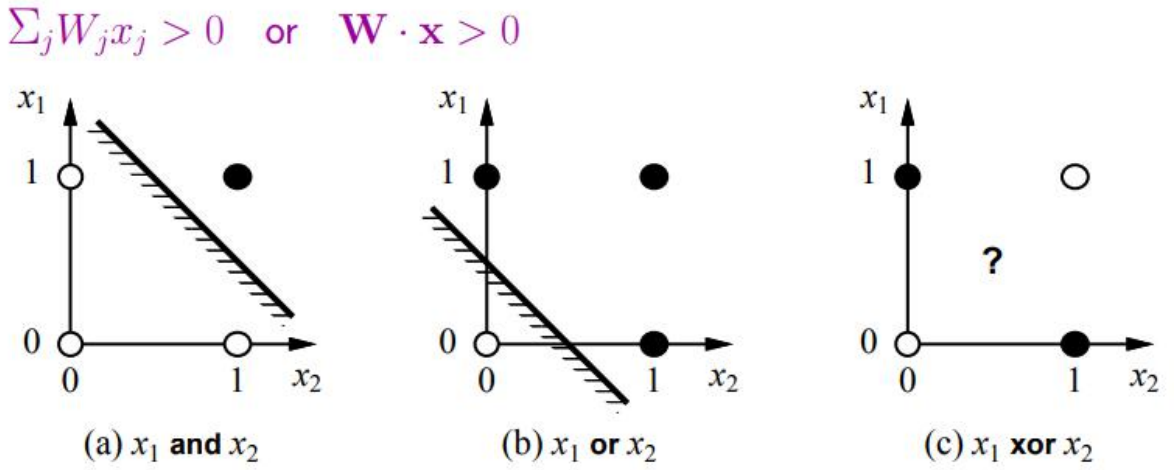



- Output units all operate separately—no shared weights
- Adjusting weights moves the location, orientation, and steepness of cliff

# Expressiveness of perceptrons

- Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)
- Can represent AND, OR, NOT, majority, etc., but not XOR
- Represents a linear separator in input space:




Minsky & Papert (1969) pricked the neural network balloon

# Perceptron learning

Learn by adjusting weights to reduce error on training set

The squared error for an example with input $\mathbf{x}$ and true output $y$ is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 \ ,$$

Perform optimization search by gradient descent:

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

Simple weight update rule:

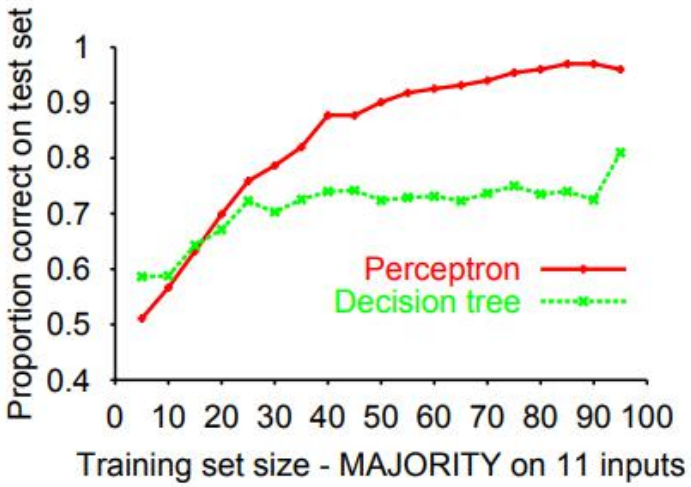
$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

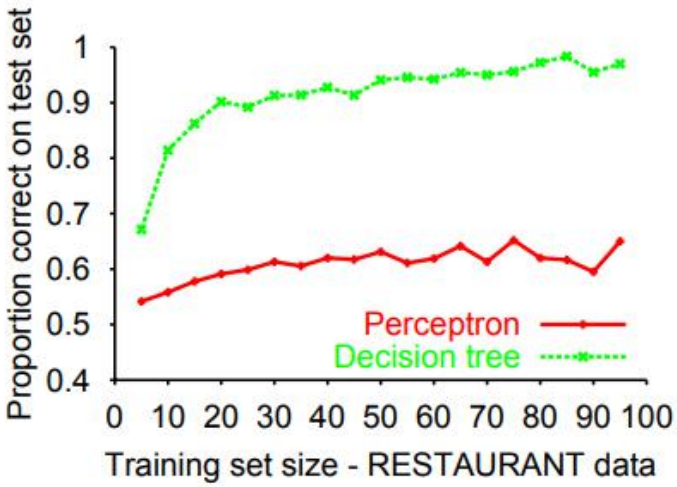
E.g., +ve error $\Rightarrow$ increase network output

$\Rightarrow$ increase weights on +ve inputs, decrease on -ve inputs

# Perceptron learning contd.

- Perceptron learning rule converges to a consistent function **for any linearly separable data set**



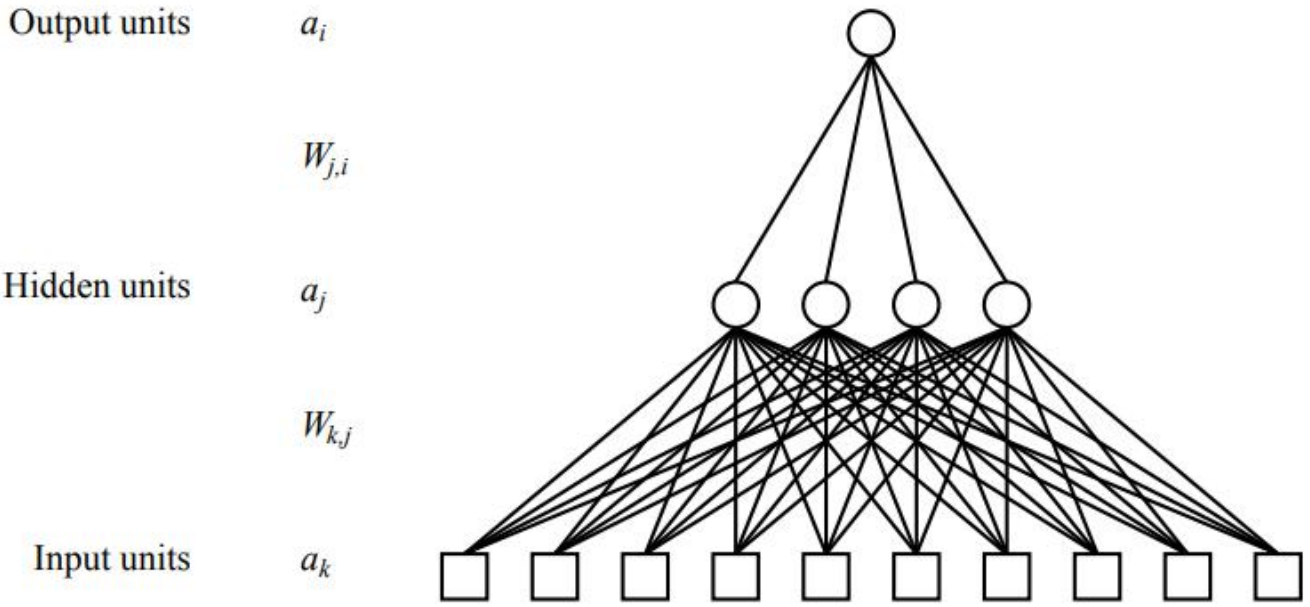

Perceptron learns majority function easily, DTL is hopeless
DTL learns restaurant function easily, perceptron cannot represent it.

# Outline

- Introduction to Nueral Networks
- Feed-Forward Networks
  - Single-layer Perceptron (SLP)
  - **Multi-layer Perceptron (MLP)**
- Back-propagation Learning

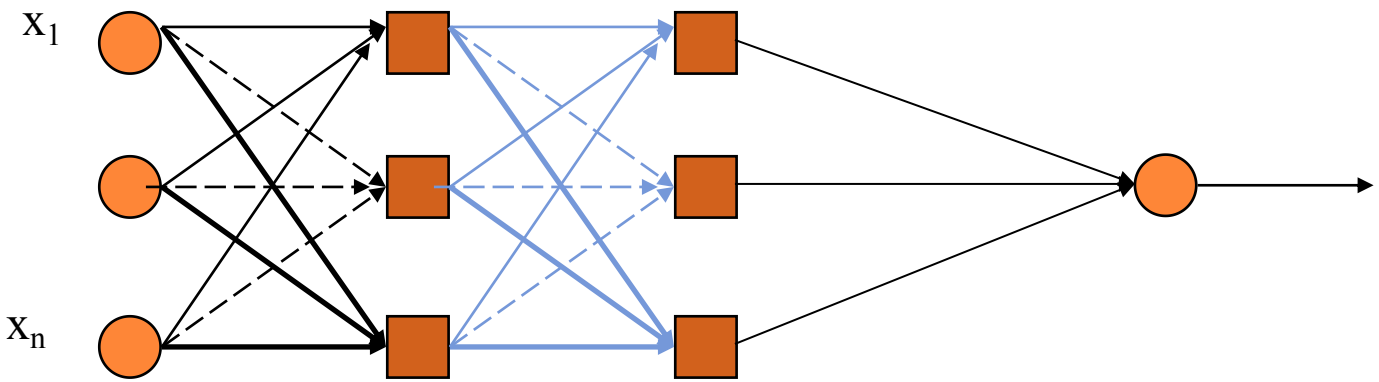
# Multilayer perceptrons

- Layers are usually fully connected, and numbers of hidden units typically chosen by hand

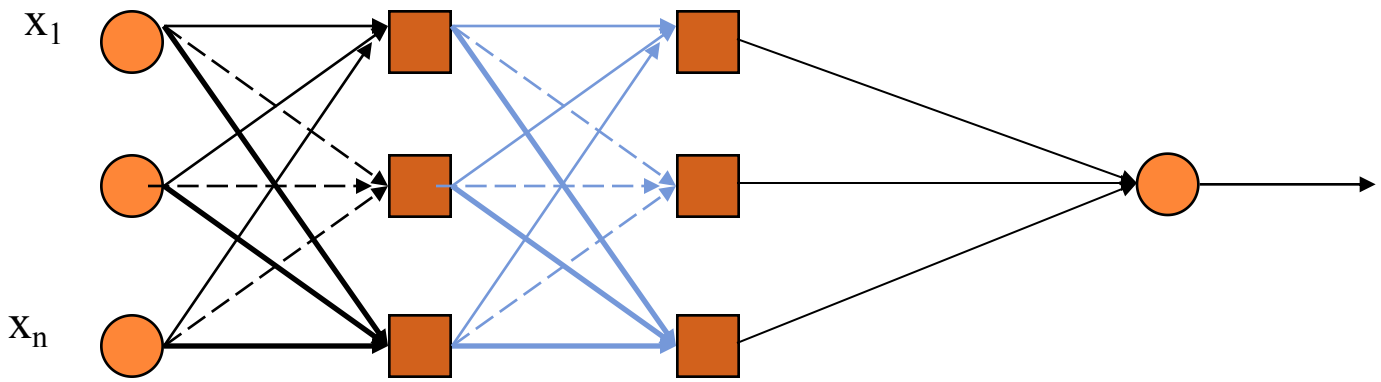


# Multi-Layer Perceptron

- We will introduce the MLP and the backpropagation algorithm which is used to train it
- MLP used to describe any general feedforward (no recurrent connections) network
- However, we will concentrate on nets with units arranged in layers



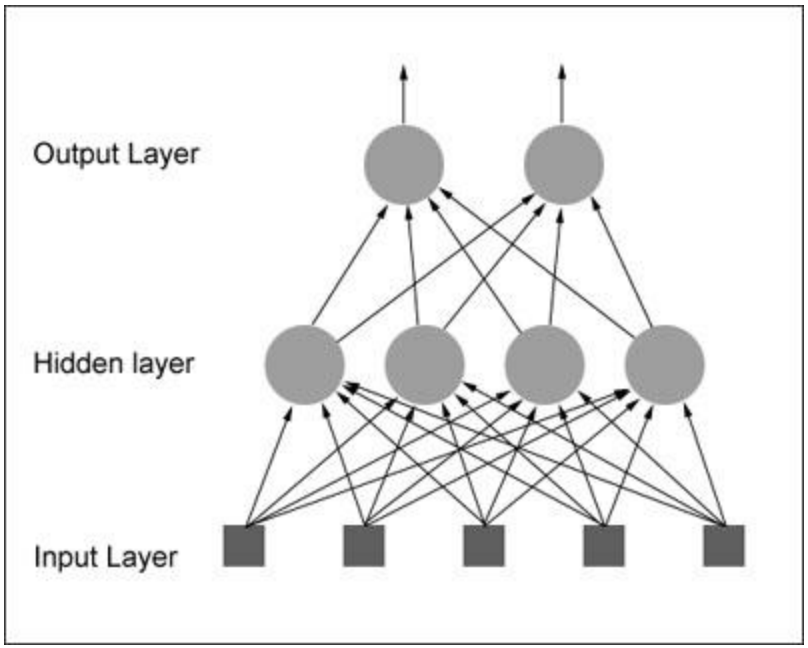
# Multi-Layer Perceptron




- Different books refer to the above as either 4 layer (no. of layers of neurons) or 3 layer (no. of layers of adaptive weights). We will follow the latter convention.
- What do the extra layers gain you? Start with looking at what a single layer can't do.

# How do we actually *use* an artificial neuron?

- Feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.
- There can be any number of hidden layers within a feedforward network.
- The number of neurons can be completely arbitrary.

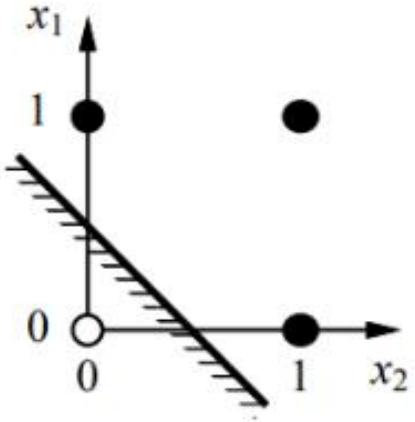


# Perceptron Learning Theorem

- *Recap*: A perceptron (threshold unit) can *learn* anything that it can *represent* (i.e. anything separable with a hyperplane)

Logical OR Function

| *x1* | *x2* | *y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

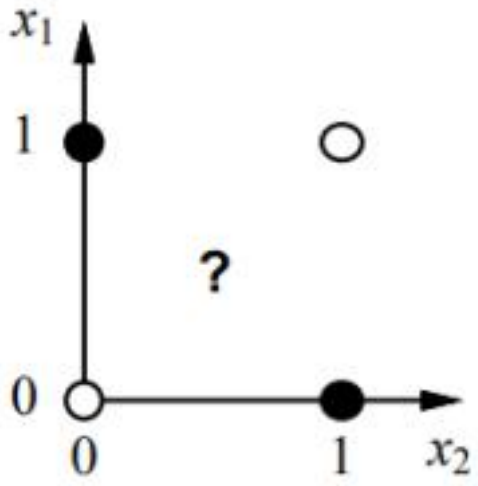


# The Exclusive OR problem

A Perceptron cannot represent Exclusive OR since it is not linearly separable.

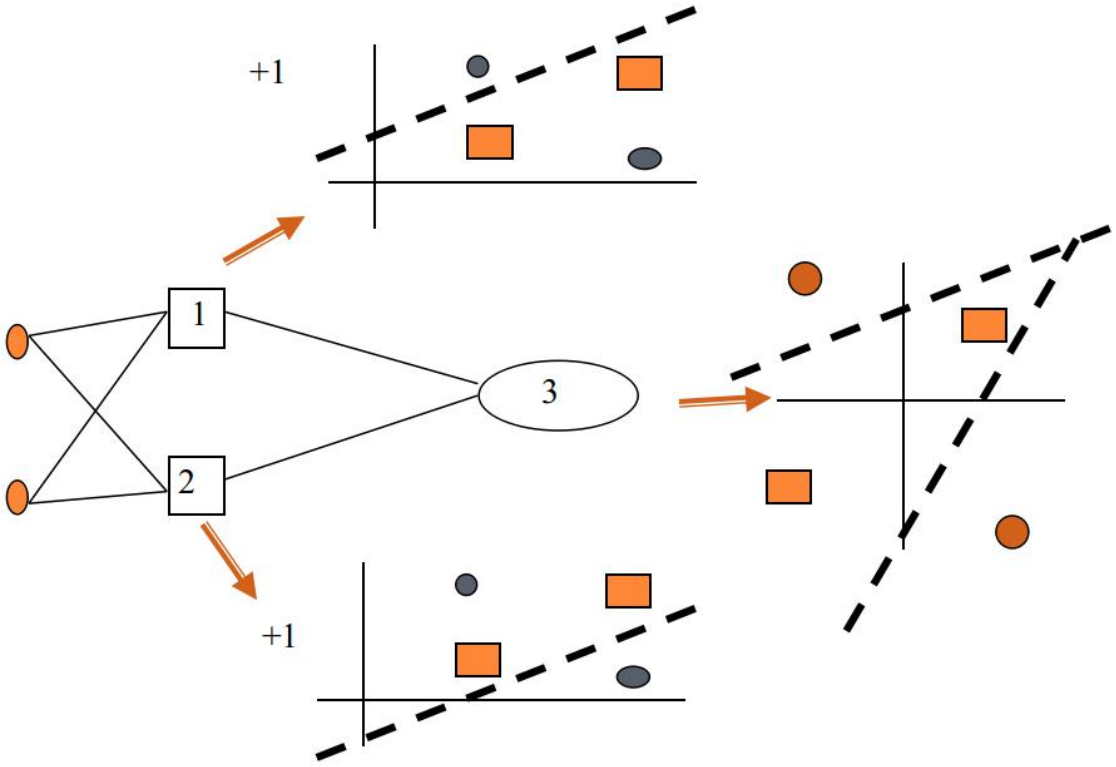
Logical XOR Function

| *x1* | *x2* | *y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



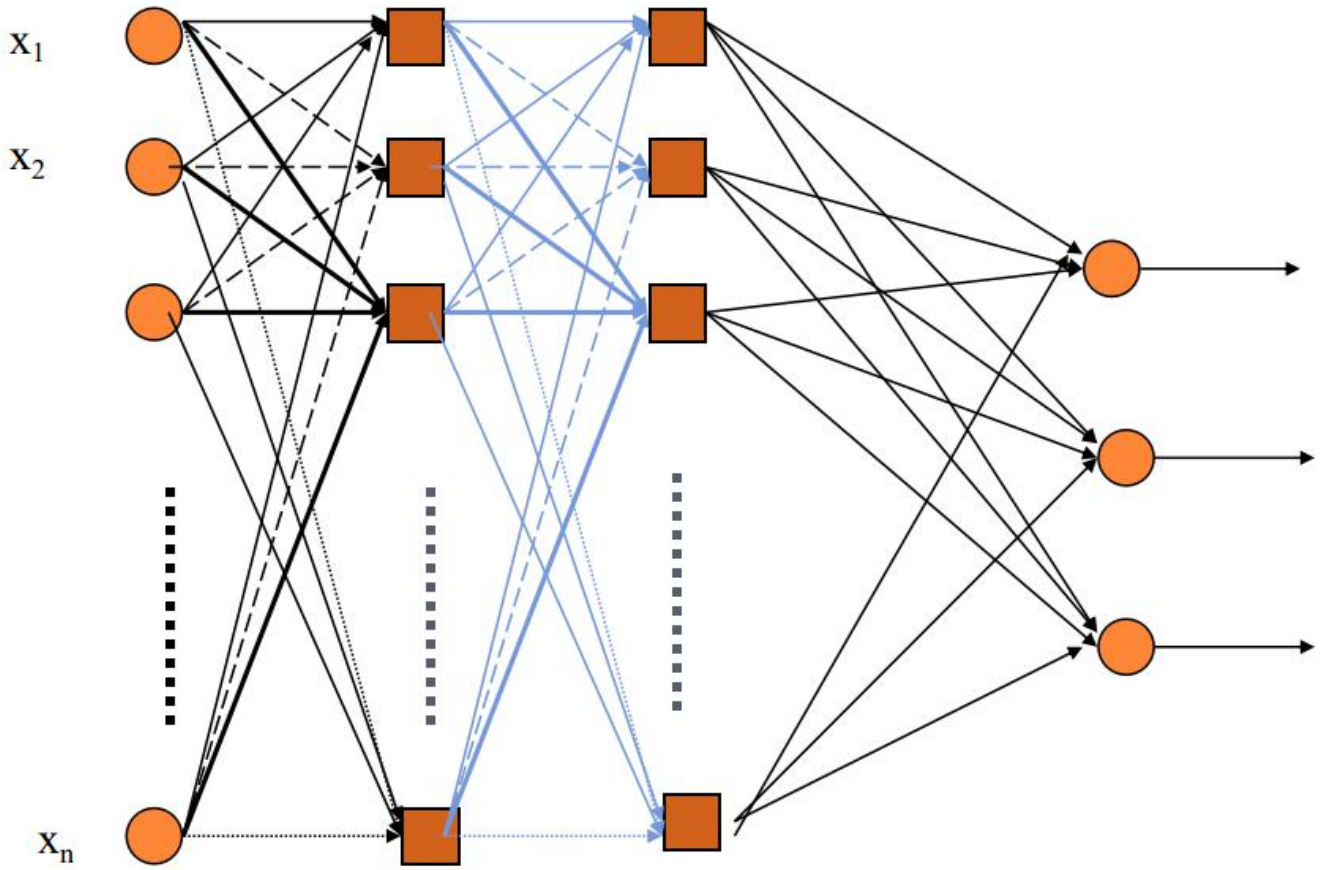
# Piecewise linear classification using an MLP

- Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of Units. Piecewise linear classification using an MLP with threshold (perceptron) units



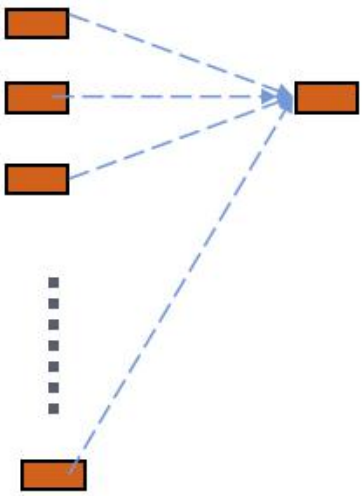
# Piecewise linear classification using an MLP

## Three-layer networks



# Properties of architecture

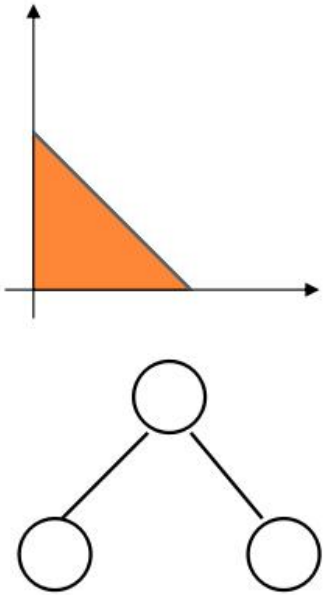
- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units



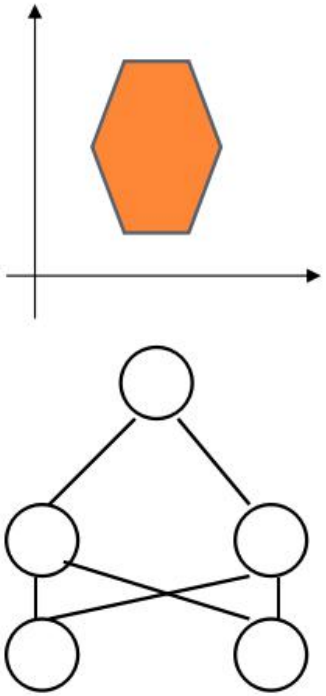
Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

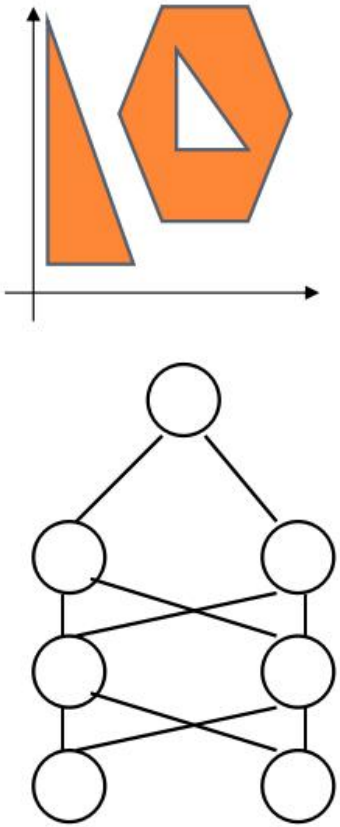
# What do each of the layers do?



1st layer draws linear boundaries



2nd layer combines the boundaries



3rd layer can generate arbitrarily complex boundaries

# Outline

- Introduction to Nueral Networks
- Feed-Forward Networks
  - Single-layer Perceptron (SLP)
  - Multi-layer Perceptron (MLP)
- **Back-propagation Learning**

# Backpropagation learning algorithm BP

- Solution to credit assignment problem in MLP. *Rumelhart, Hinton and Williams (1986) (*though actually invented earlier in a PhD thesis relating to economics)
- **BP has two phases**:

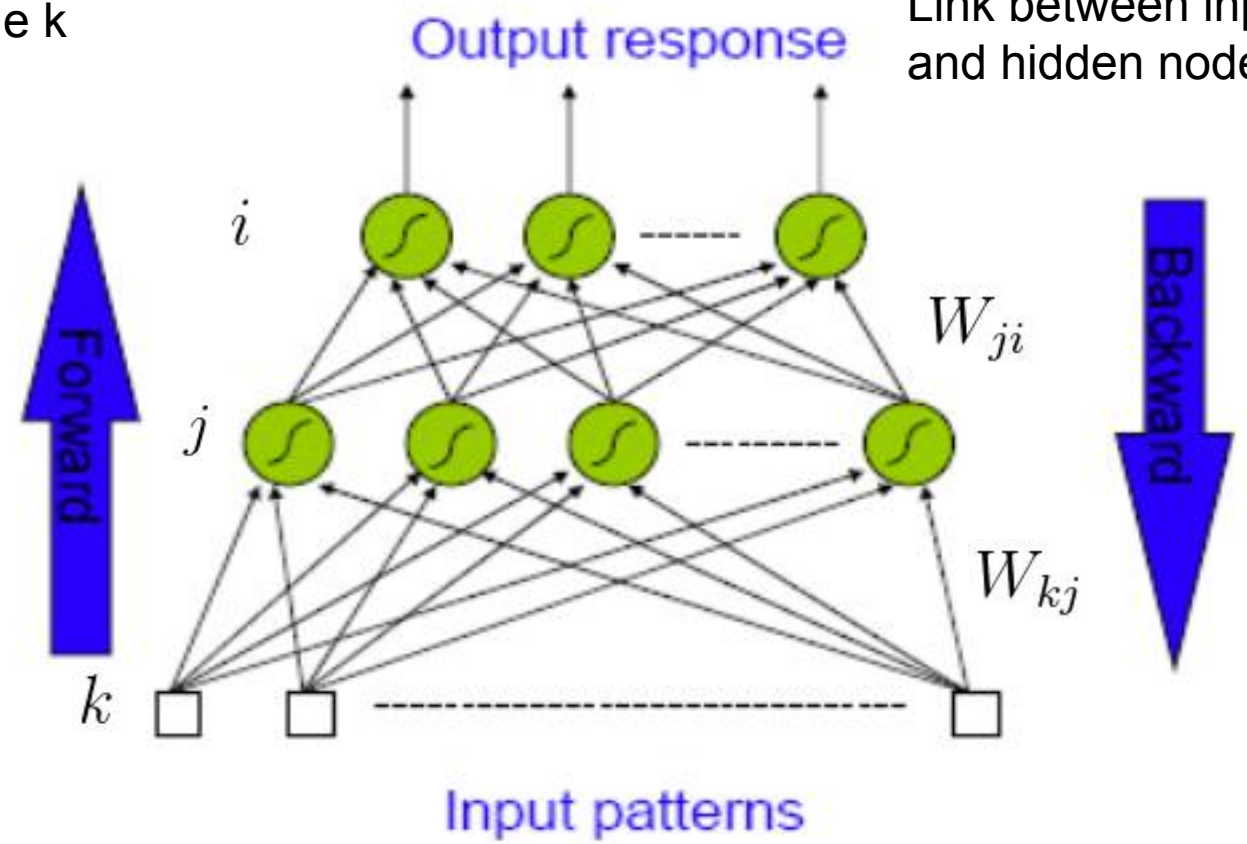
**Forward pass phase**: computes 'functional signal', feed forward propagation of input pattern signals through network.

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

# Conceptually: Forward Activity - Backward Error

Output node i
Hidden node j
Input node k

Link between hidden node j and output node i: Wji
Link between input node k and hidden node j: Wkj



# Back-propagation derivation

- The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - o_i)^2$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\delta E}{\delta W_{kj}} &= -(y_i - o_i)\frac{\delta o_i}{\delta W_{ji}} = -(y_i - o_i)\frac{\delta g(in_i)}{\delta W_{ji}} \\
&= -(y_i - o_i)g'(in_i)\frac{\delta in_i}{\delta W_{ji}} = -(y_i - o_i)g'(in_i)\frac{\delta}{\delta W_{ji}}(\sum_j W_{ji}o_j) \\
&= -(y_i - o_i)g'(in_i)o_j \\
&= -\sigma_i o_j
\end{aligned}$$

# Back-propagation derivation contd.

$$\frac{\delta E}{\delta W_{kj}} = -\sum_i (y_i - o_i)\frac{\delta o_i}{\delta W_{kj}} = -\sum_i (y_i - o_i)\frac{\delta g(in_i)}{\delta W_{kj}}$$

$$= -\sum_i (y_i - o_i) g'(in_i)\frac{\delta in_i}{\delta W_{kj}} = -\sum_i (y_i - o_i) g'(in_i)\frac{\delta}{\delta W_{kj}}(\sum_j W_{ji} o_j)$$

$$= -\sum_i \sigma_i \frac{\delta}{\delta W_{kj}}(\sum_j W_{ji} o_j) = -\sum_i \sigma_i W_{ji}\frac{\delta o_j}{\delta W_{kj}}$$

$$= -\sum_i \sigma_i W_{ji}\frac{\delta g(in_j)}{\delta W_{kj}} = -\sum_i \sigma_i W_{ji} g'(in_j)\frac{\delta in_j}{\delta W_{kj}}$$

$$= -\sum_i \sigma_i W_{ji} g'(in_j)\frac{\delta}{\delta W_{kj}}(\sum_k W_{kj} o_k) = -\sum_i \sigma_i W_{ji} g'(in_j) o_k$$

$$= -\sigma_j o_k$$

# Back-propagation Learning

- Output layer: same as the single-layer perceptron

$$W_{ji} = W_{ji} + \eta \sigma_i o_j$$

where $\sigma_i = -(y_i - o_i) g'(in_i)$

- Hidden layer: back-propagation the error from the output layer.

$$\sigma_j = -\sum_i \sigma_i W_{ji} g'(in_j)$$

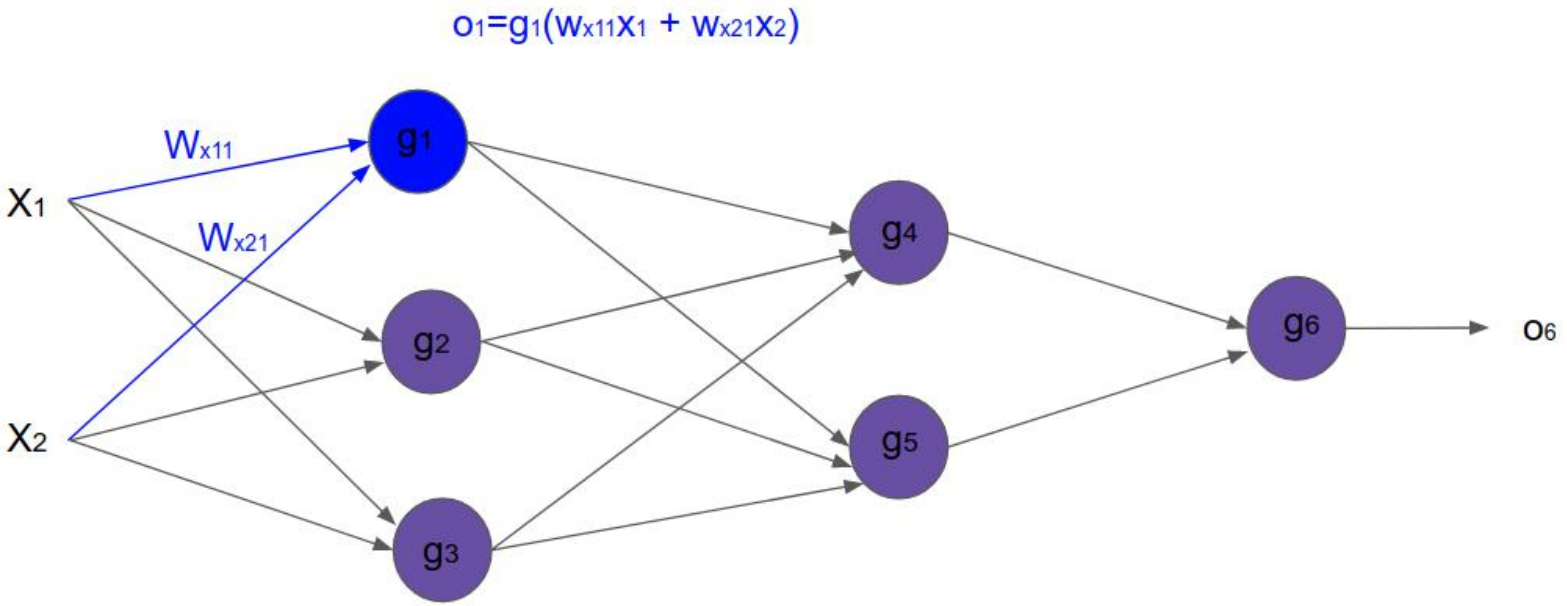
- Upadate rules for weights in the hidden layers.

$$W_{kj} = W_{kj} + \eta \sigma_j o_k$$

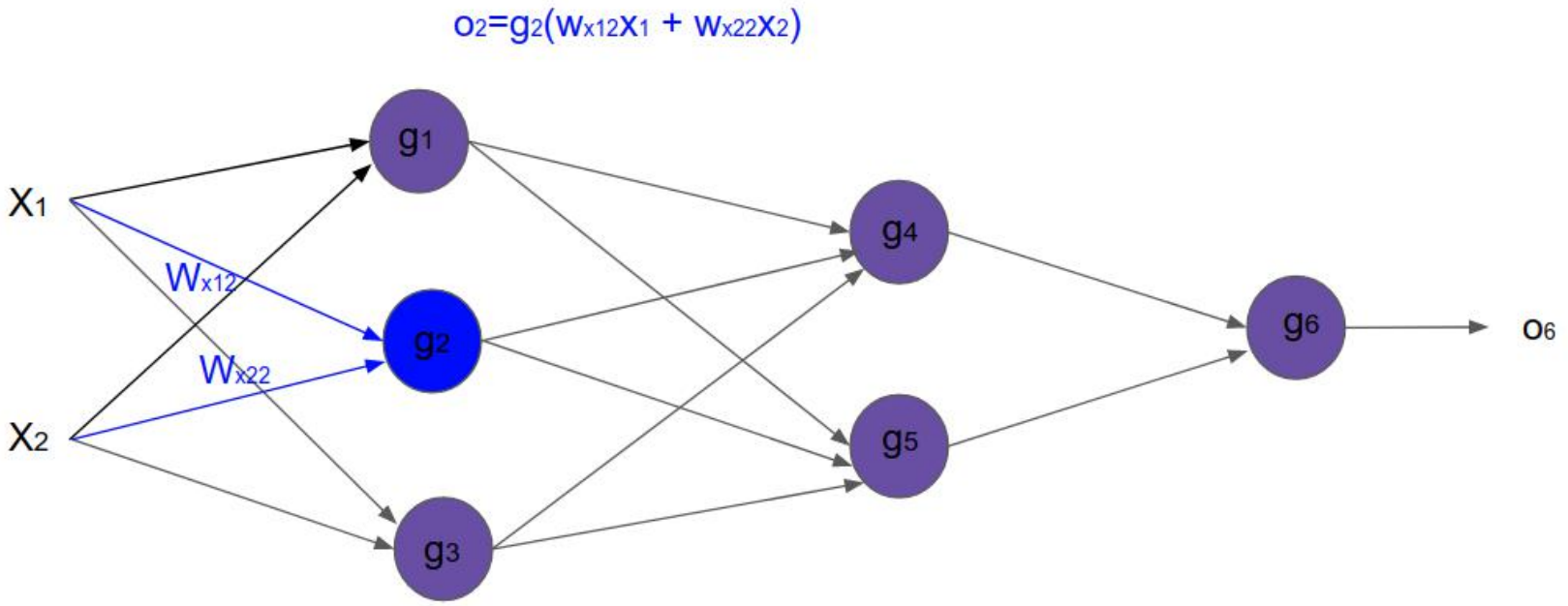
(Most neuroscientists deny that back-propagation occurs in the brain)

# Learning Algorithm: Backpropagation

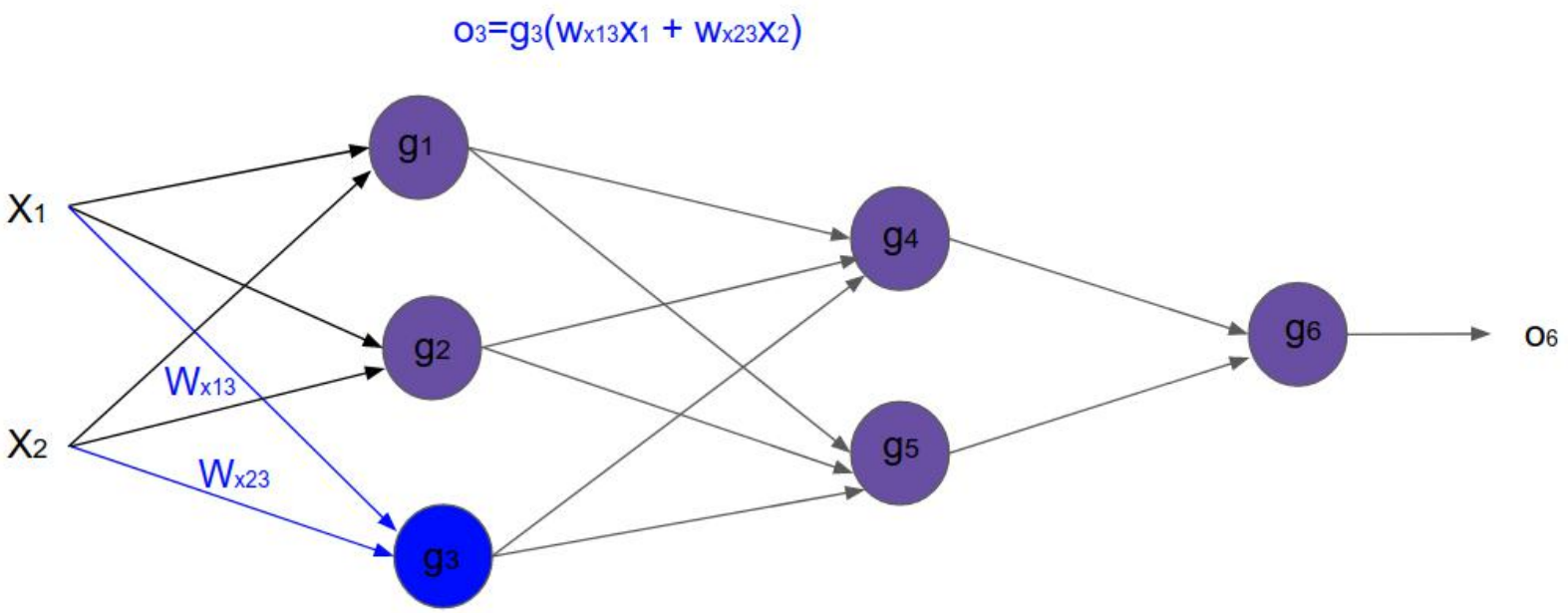
- Pictures below illustrate how signal is propagating through the network, Symbols $w_{(xm)n}$ represent weights of connections between network input $x_m$ and neuron $n$ in input layer.Symbols $o_n$ represents output signal of neuron $n$.



# Learning Algorithm: Backpropagation

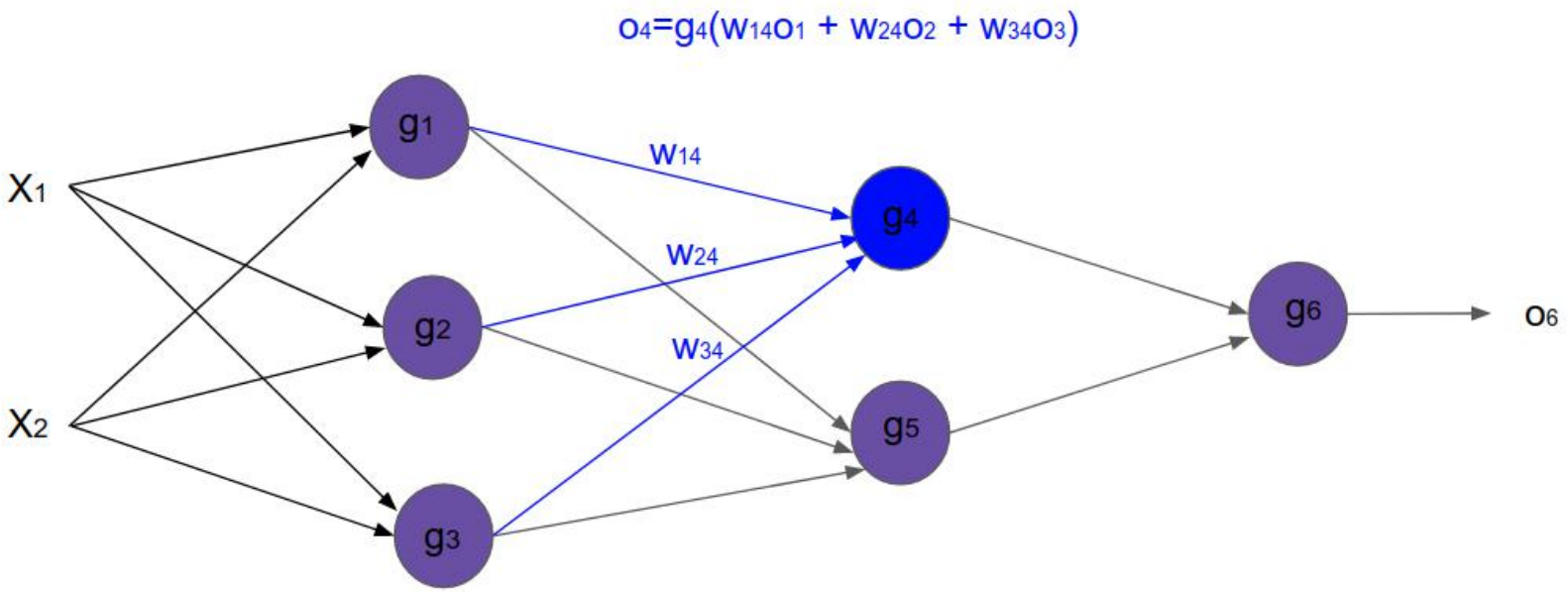


# Learning Algorithm: Backpropagation

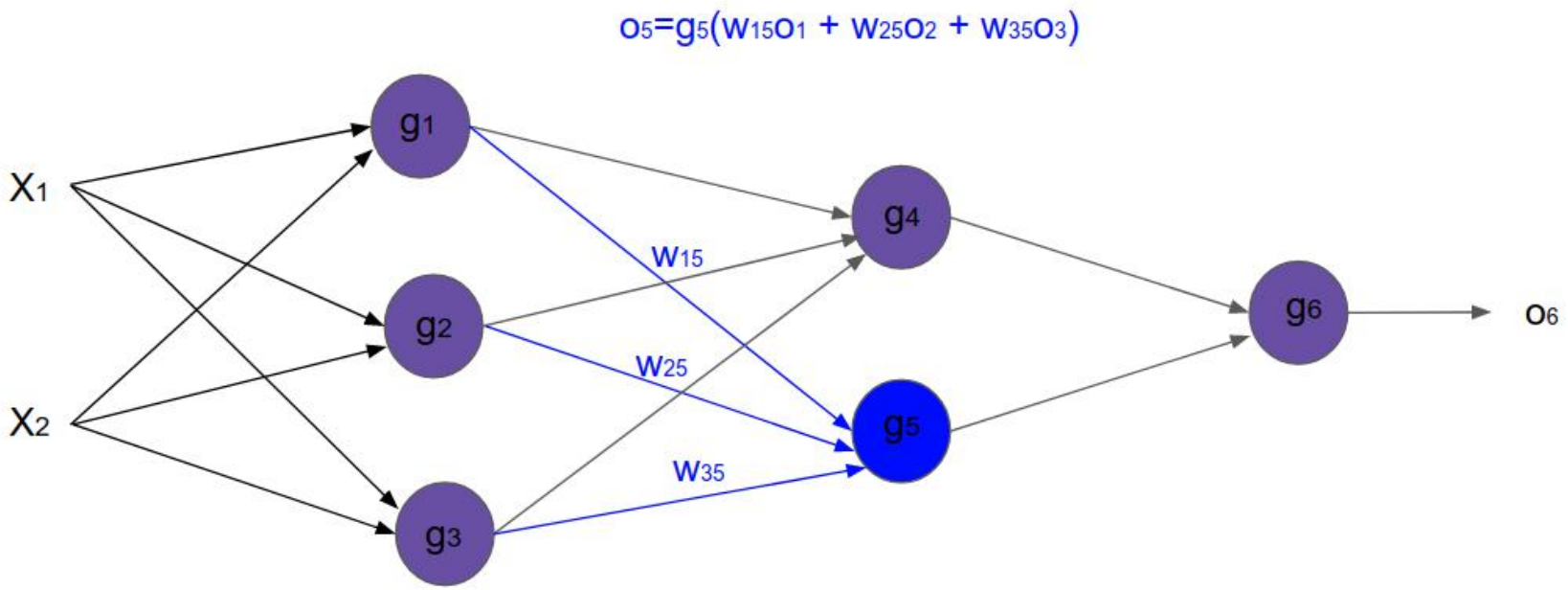


# Learning Algorithm: Backpropagation

- 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.

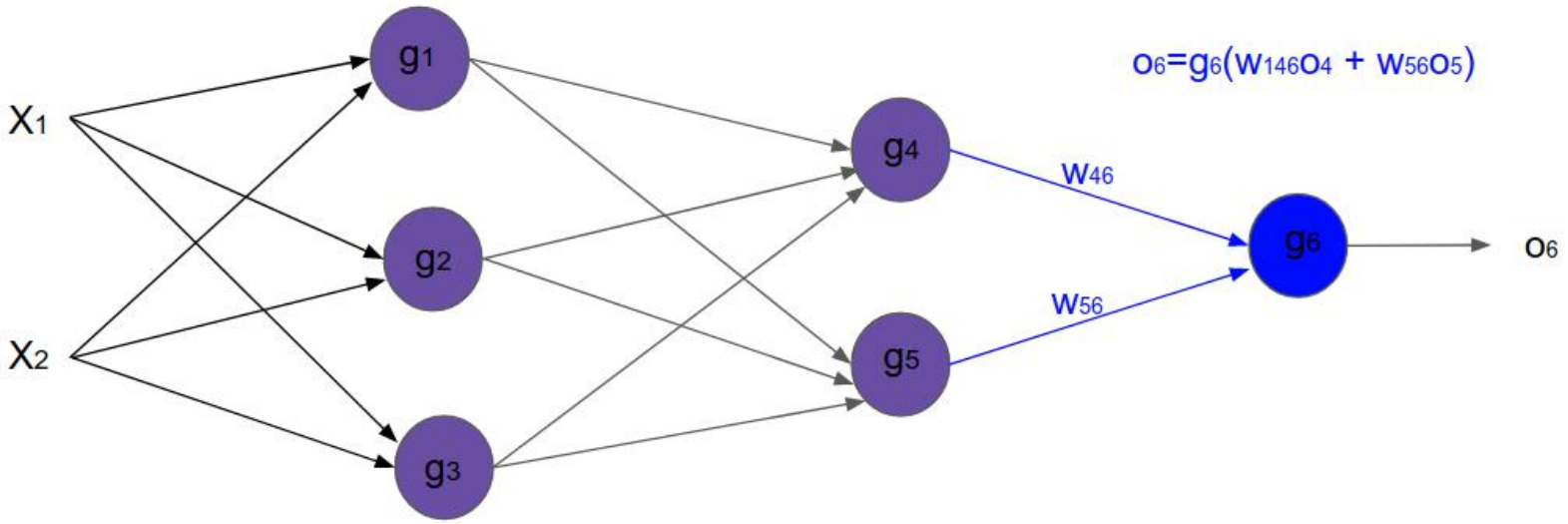


# Learning Algorithm: Backpropagation



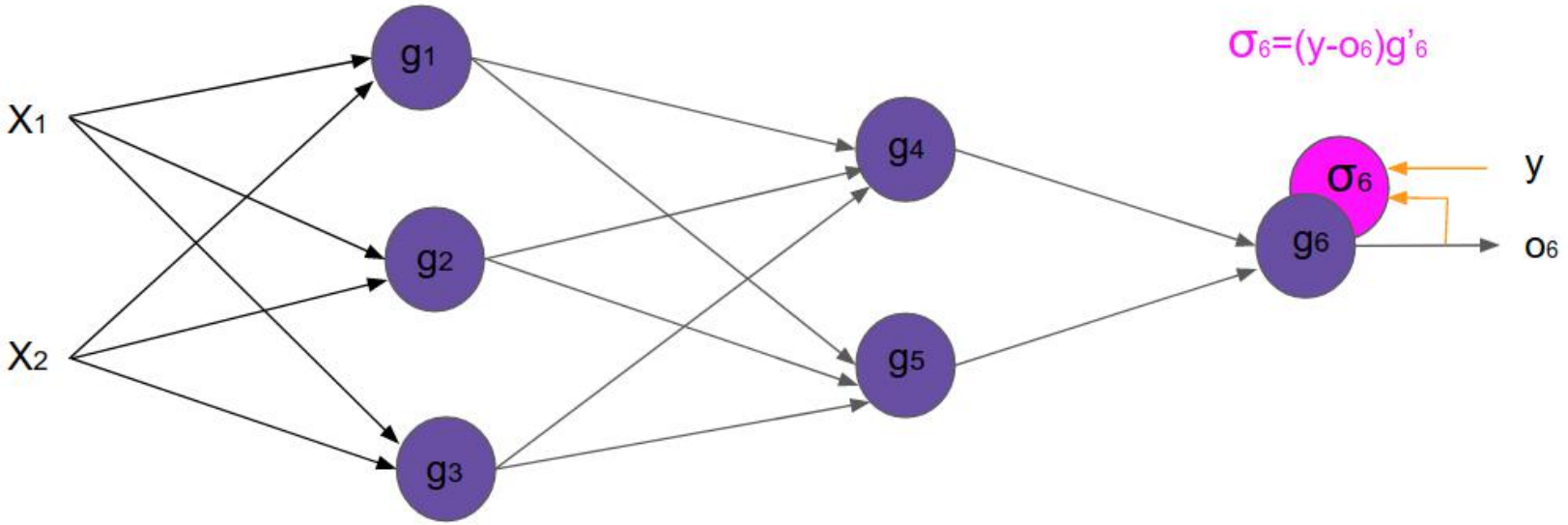
# Learning Algorithm: Backpropagation

- Propagation of signals through the output layer.



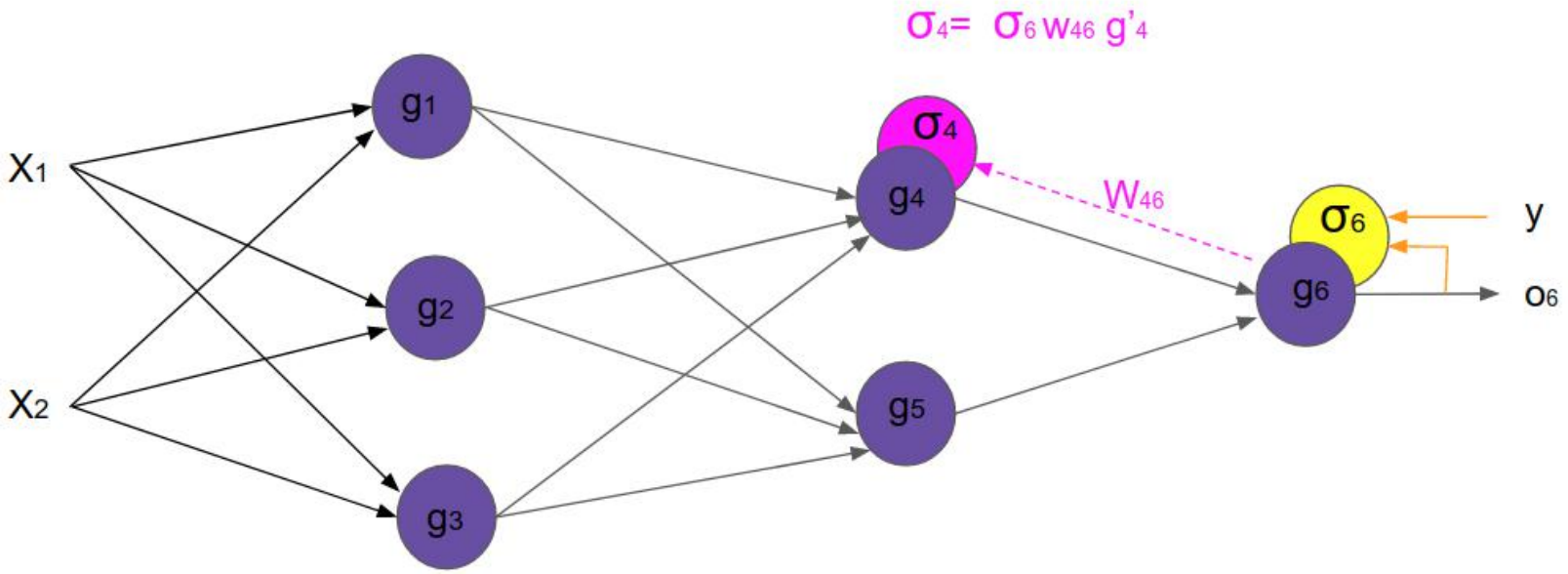
# Learning Algorithm: Backpropagation

- 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 of output layer neuron



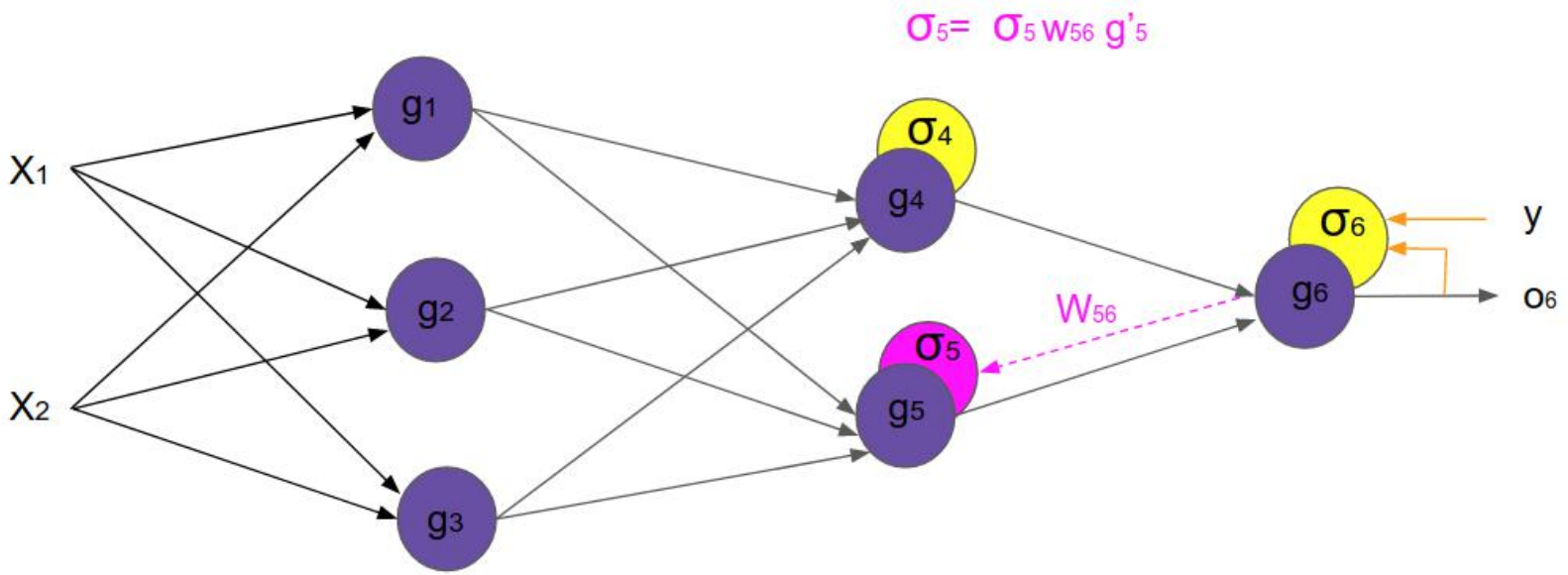
# Learning Algorithm: Backpropagation

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



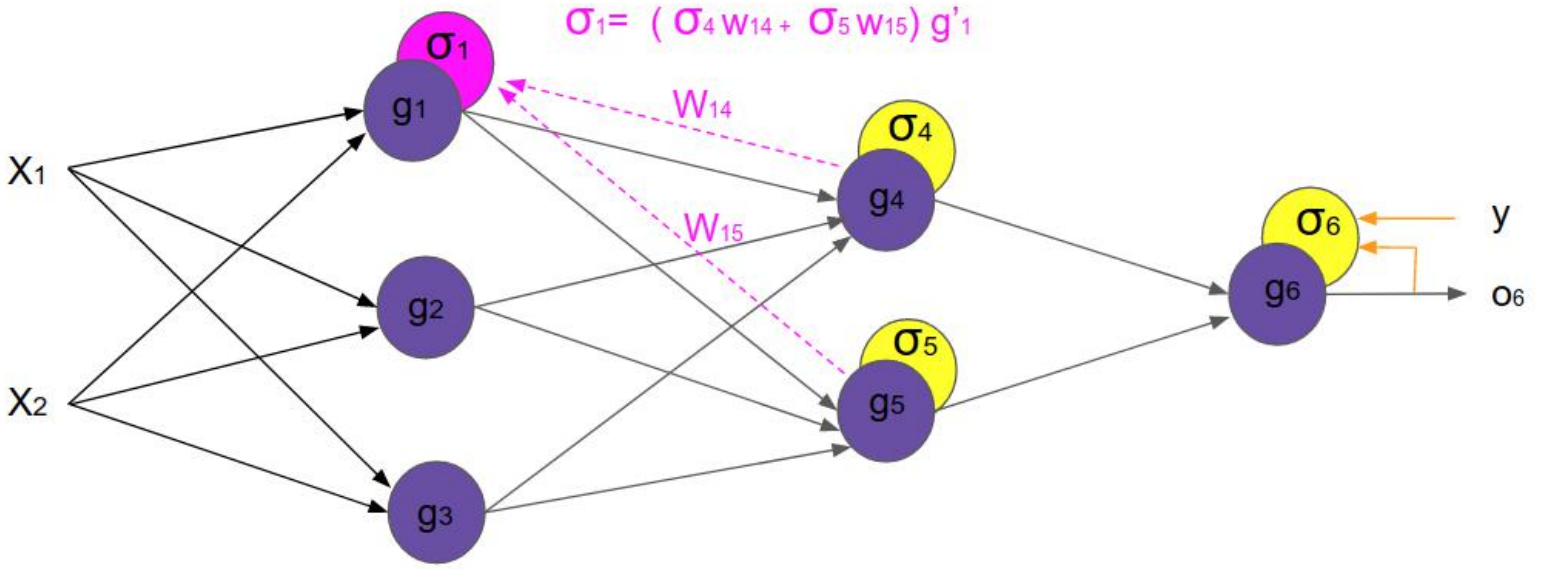
# Learning Algorithm: Backpropagation

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



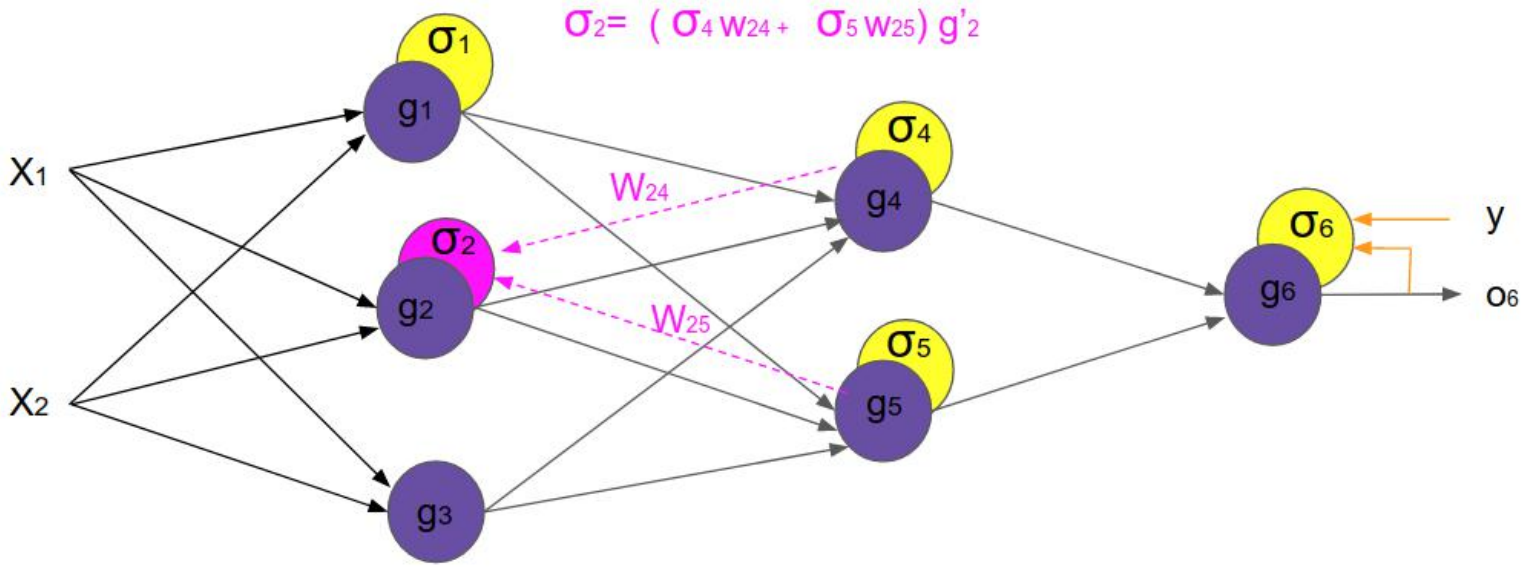
# Learning Algorithm: Backpropagation

- 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:



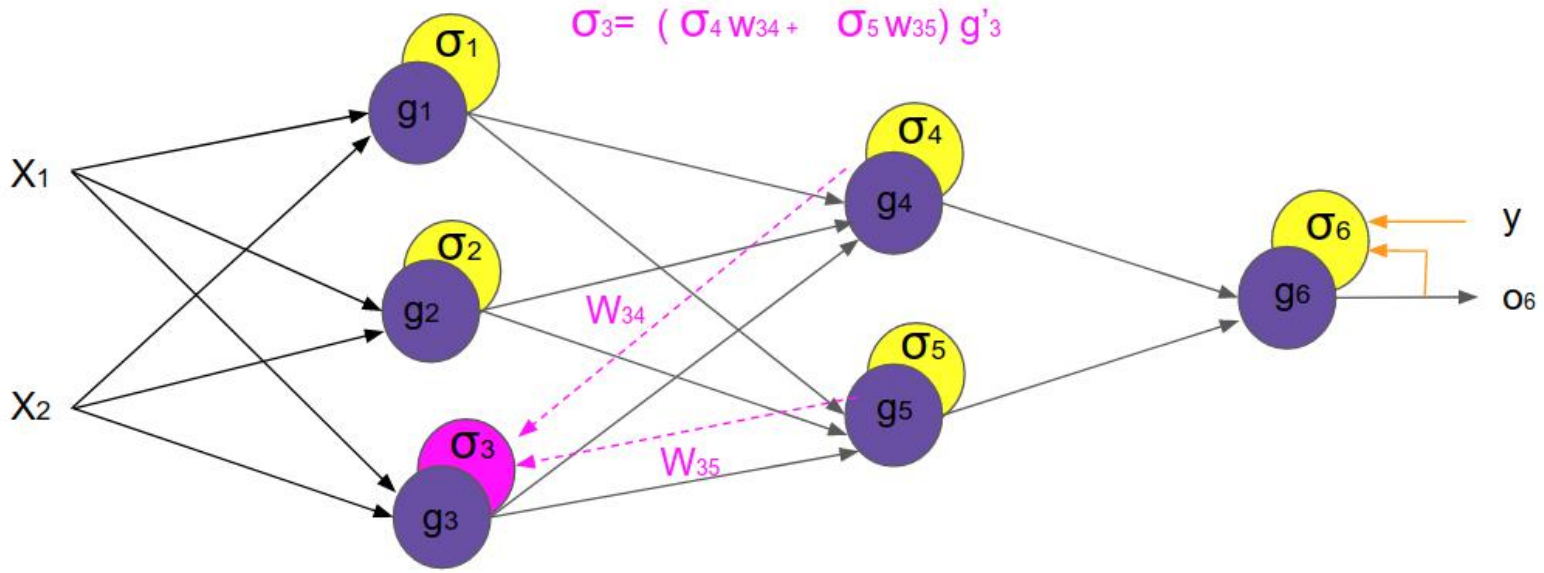
# Learning Algorithm: Backpropagation

- 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:



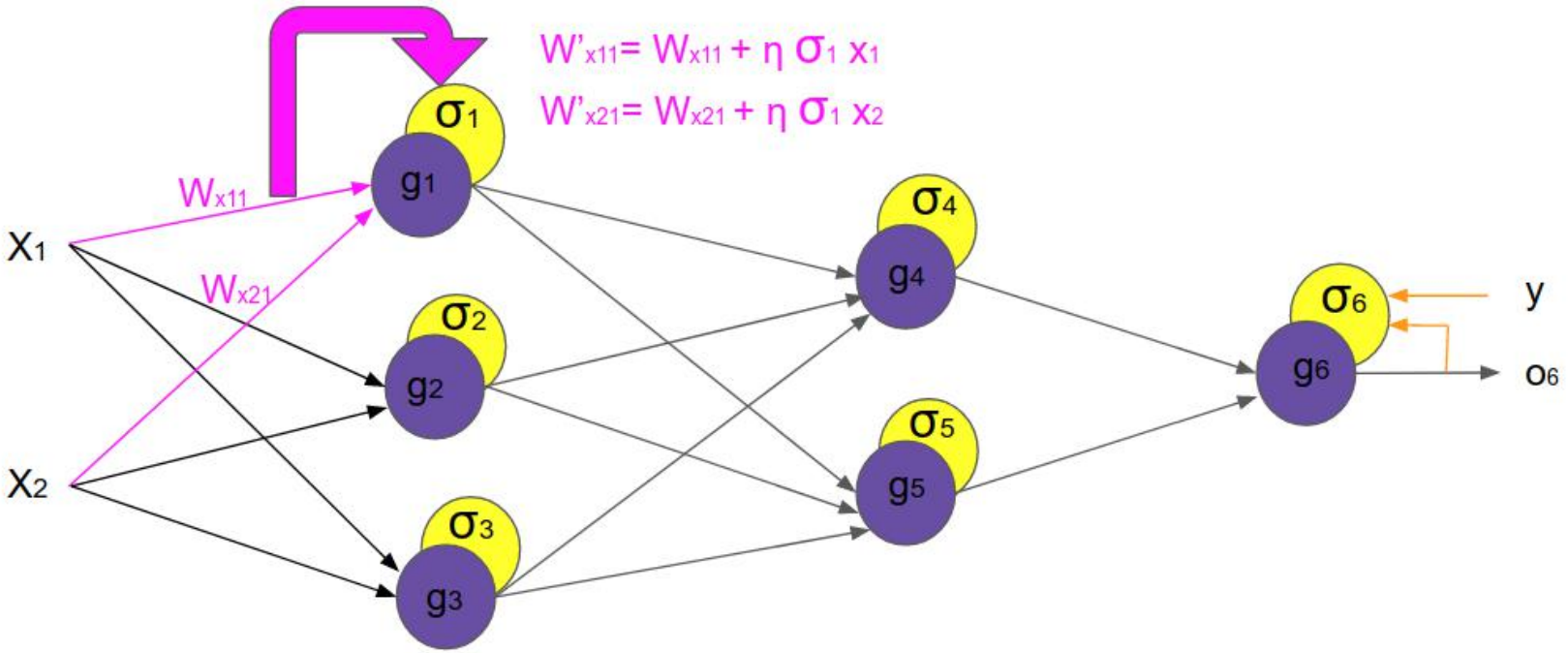
# Learning Algorithm: Backpropagation

- 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:



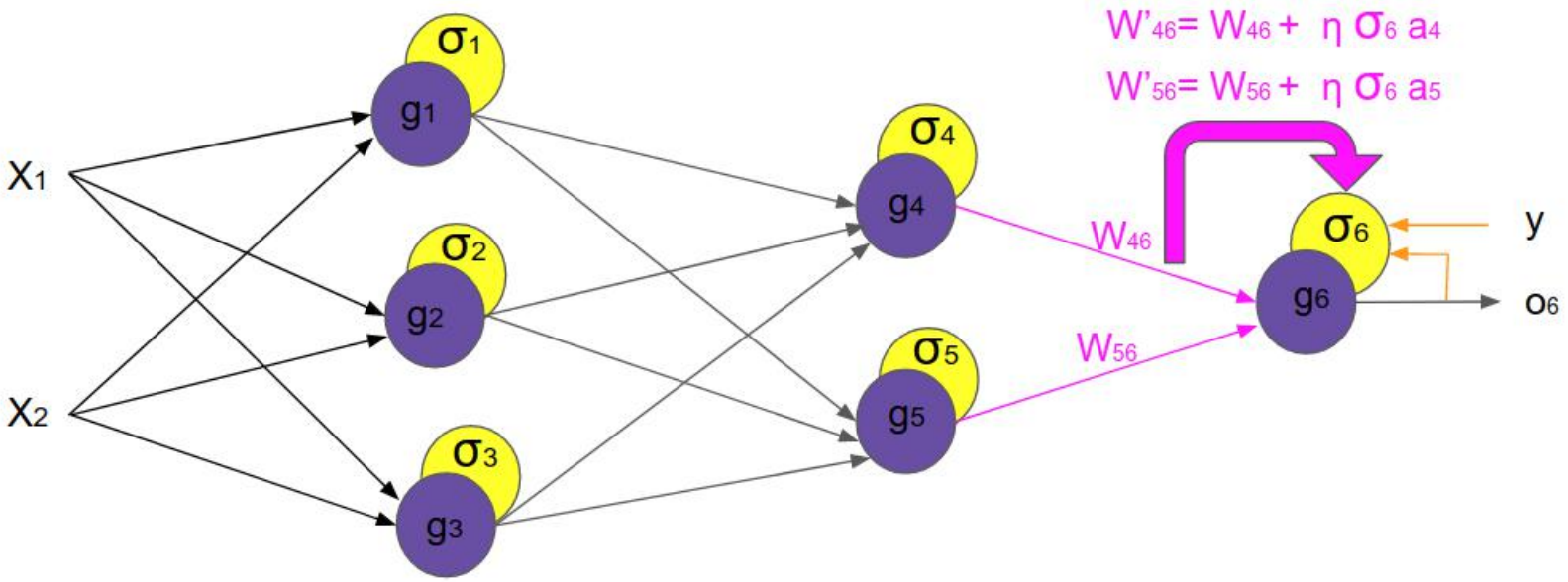
# Learning Algorithm: Backpropagation

- When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified.



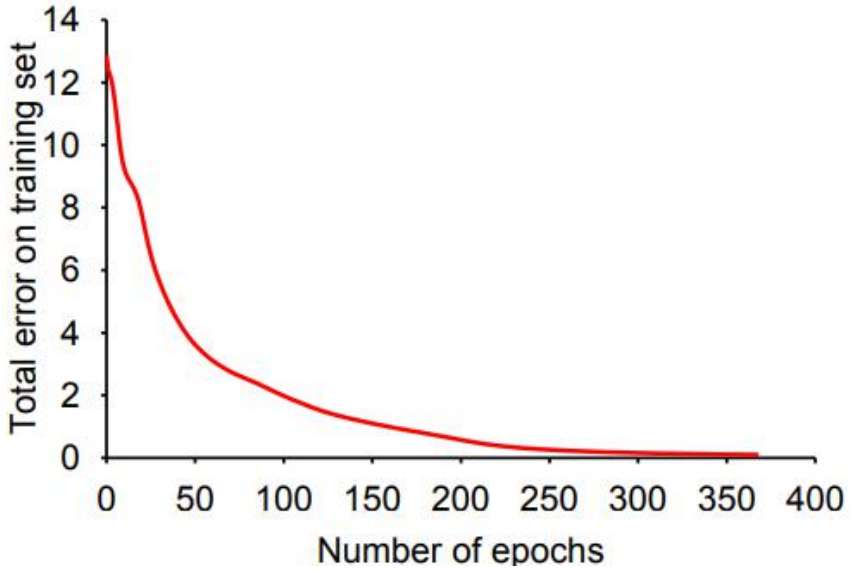
# Learning Algorithm: Backpropagation

- When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified.



# Back-propagation learning contd.

- At each epoch, sum gradient updates for all examples and apply
- Training curve for $100$ restaurant examples: finds exact fit



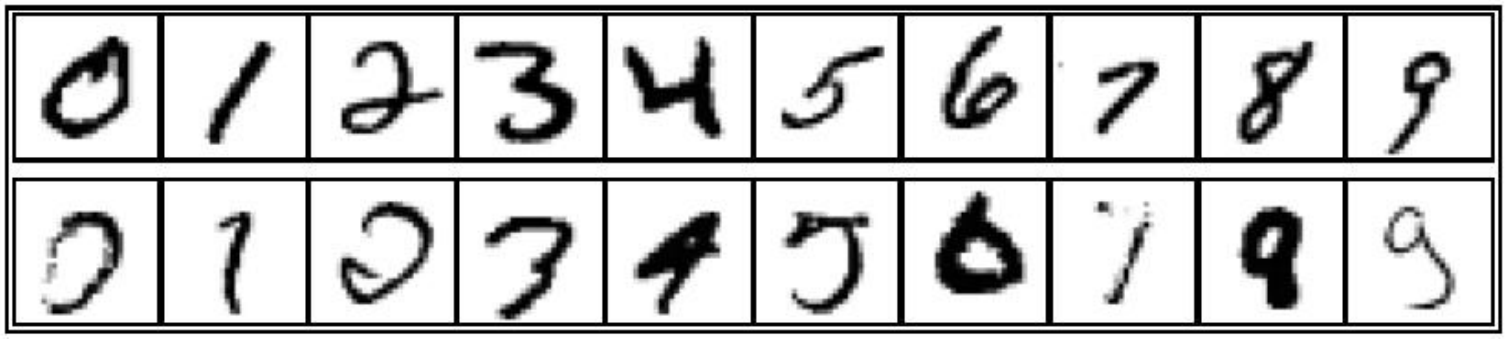

Typical problems: slow convergence, local minima

# Back-propagation learning contd.

- Learning curve for MLP with 4 hidden units:


MLPs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

# Handwritten digit recognition



- $3-$nearest neighbor $= 2.4\%$ error
- $400-300-10$ unit MLP $= 1.6\%$ error
- LeNet: $768-192-30-10$ unit MLP $= 0.9\%$ error
- Current best (kernel machines, vision algorithms) $\approx 0.6\%$ error

# Forward Propagation of Activity

- Step 1: Initialise weights at random, choose a learning rate $\eta$
- Until network is trained:
- For each training example i.e. input pattern and target output(s):
- Step 2: Do forward pass through net (with fixed weights) to produce output(s)
  - i.e., in Forward Direction, layer by layer:
    - Inputs applied
    - Multiplied by weights
    - Summed
    - 'Squashed' by sigmoid activation function
    - Output passed to each neuron in next layer
  - Repeat above until network output(s) produced

# Back-propagation of error

- Compute error (delta or local gradient) for each output unit $\delta k$
- Layer-by-layer, compute error (delta or local gradient) for each hidden unit $\delta j$ by backpropagating errors (as shown previously)

Step 4: Next, update all the weights $\Delta wij$
By gradient descent, and go back to Step 2

- The overall MLP learning algorithm, involving forward pass and backpropagation of error (until the network training completion), is known as the Generalised Delta Rule (GDR), or more commonly, the Back Propagation (BP) algorithm

# Training

- This was a single iteration of back-prop
- Training requires many iterations with many training examples or *epochs* (one epoch is entire presentation of complete training set)
- It can be slow !
- Note that computation in MLP is local (with respect to each neuron)
- Parallel computation implementation is also possible

# Training and testing data

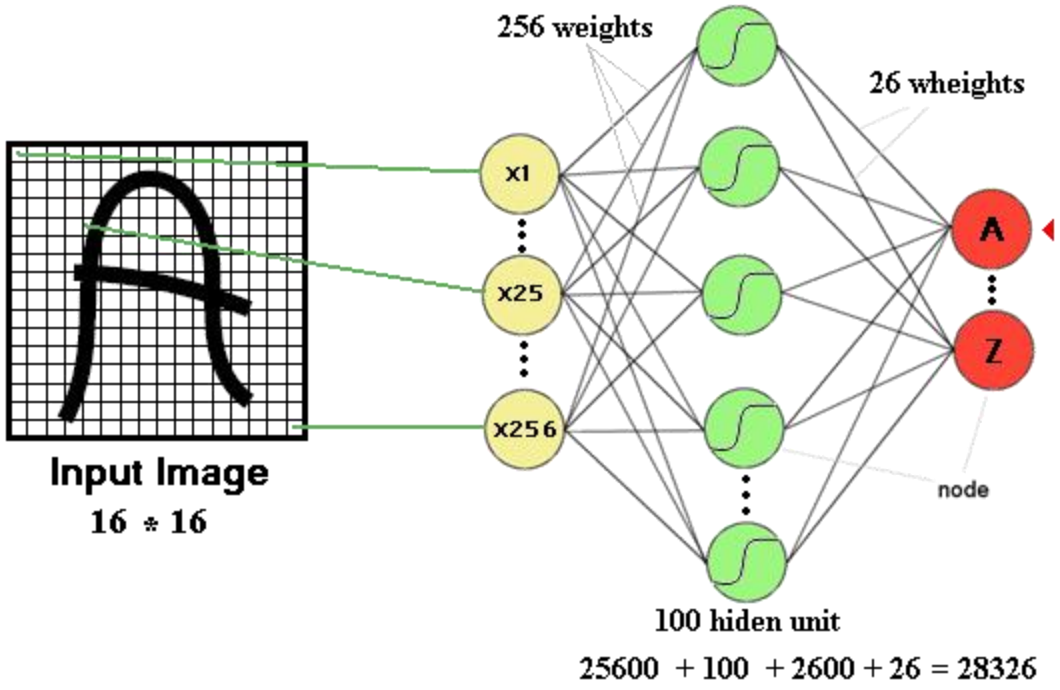
- How many examples ?
  - The more the merrier !
- Disjoint training and testing data sets
  - learn from training data but evaluate performance (generalization ability) on unseen test data
- **Aim**: minimize error on *test* data

# Summary

- Most brains have lots of neurons; each neuron ≈ linear–threshold unit.
- Perceptrons (one–layer networks) insufficiently expressive
- Multi–layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back–propagation
- Many applications: speech, driving, handwriting, fraud detection, etc.
- Engineering, cognitive modelling, and neural system modelling subfields have largely diverged.

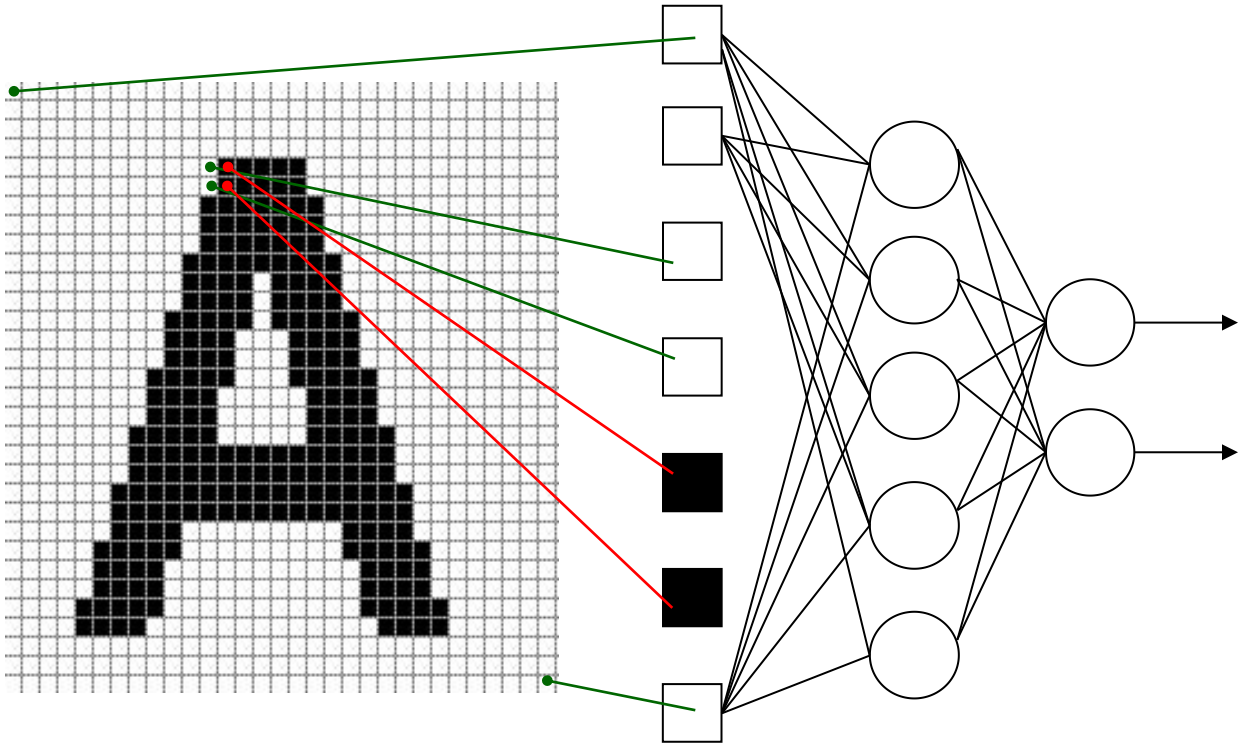
# Drawbacks of previous neural networks

- The number of trainable parameters becomes extremely large

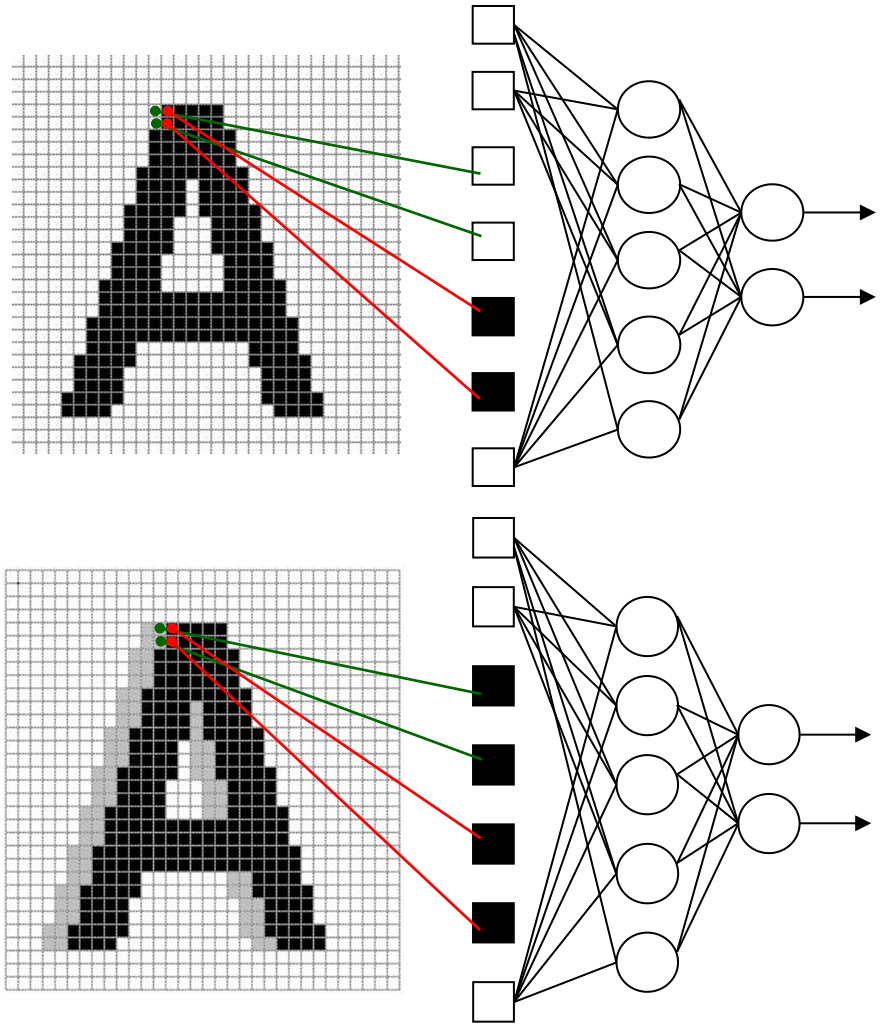


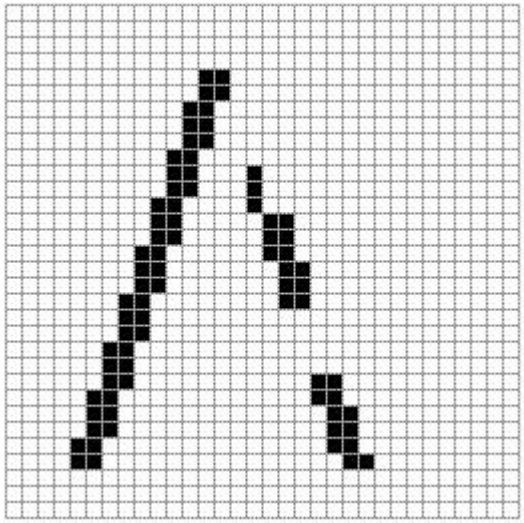
# Drawbacks of previous neural networks

- Little or no invariance to shifting, scaling, and other forms of distortion



# Drawbacks of previous neural networks





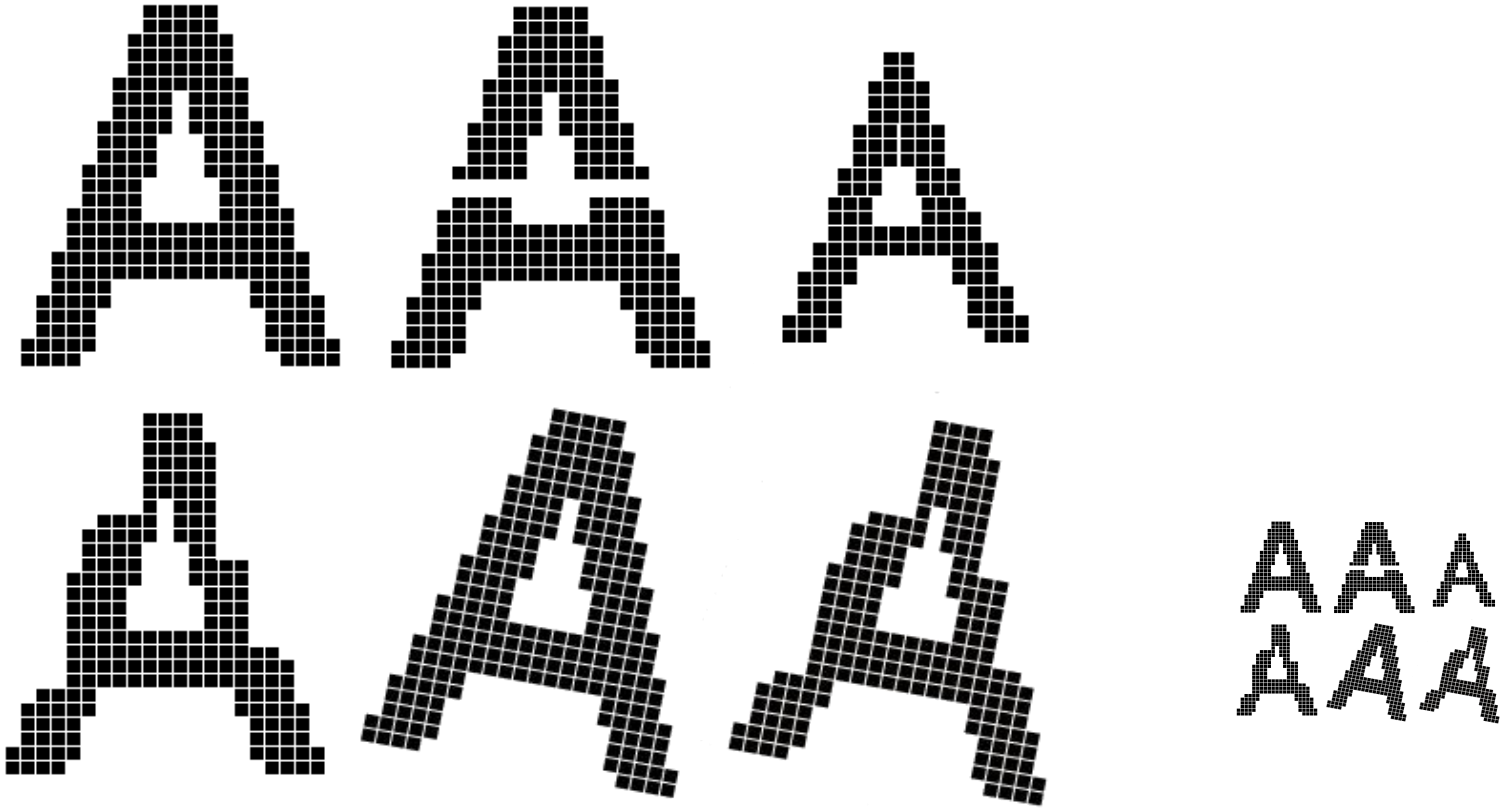
154 input change from 2 shift left

77 : black to white

77 : white to black

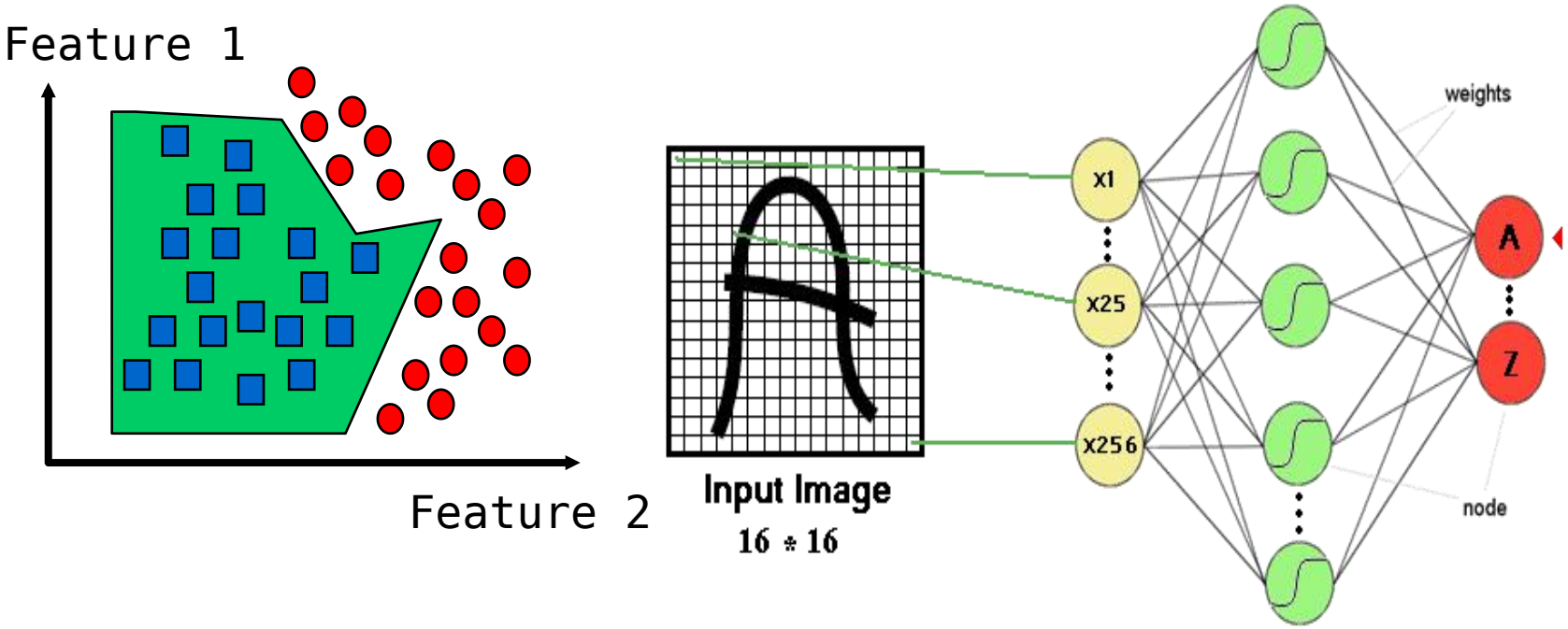
# Drawbacks of previous neural networks

- scaling, and other forms of distortion



# Drawbacks of previous neural networks

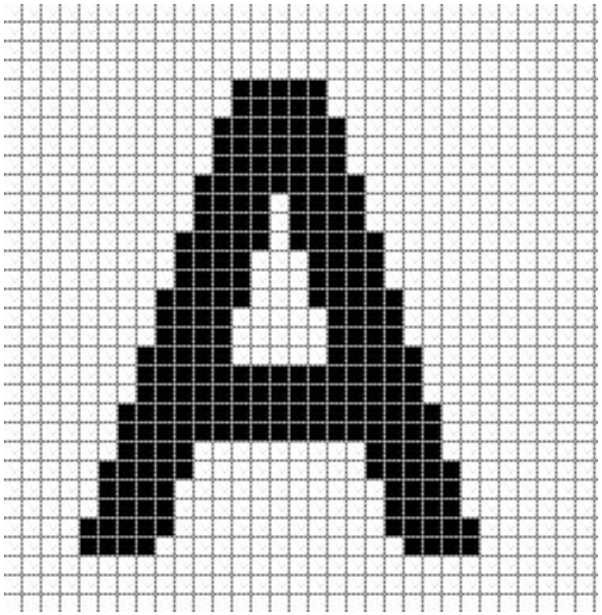
- The topology of the input data is completely ignored work with raw data.



# Drawbacks of previous neural networks

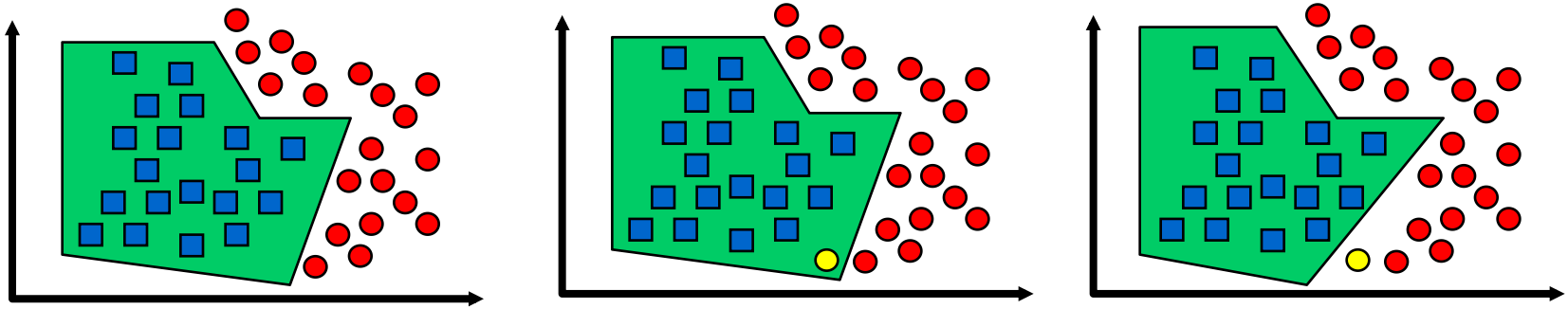
Black and white patterns: $2^{32*32} = 2^{1024}$

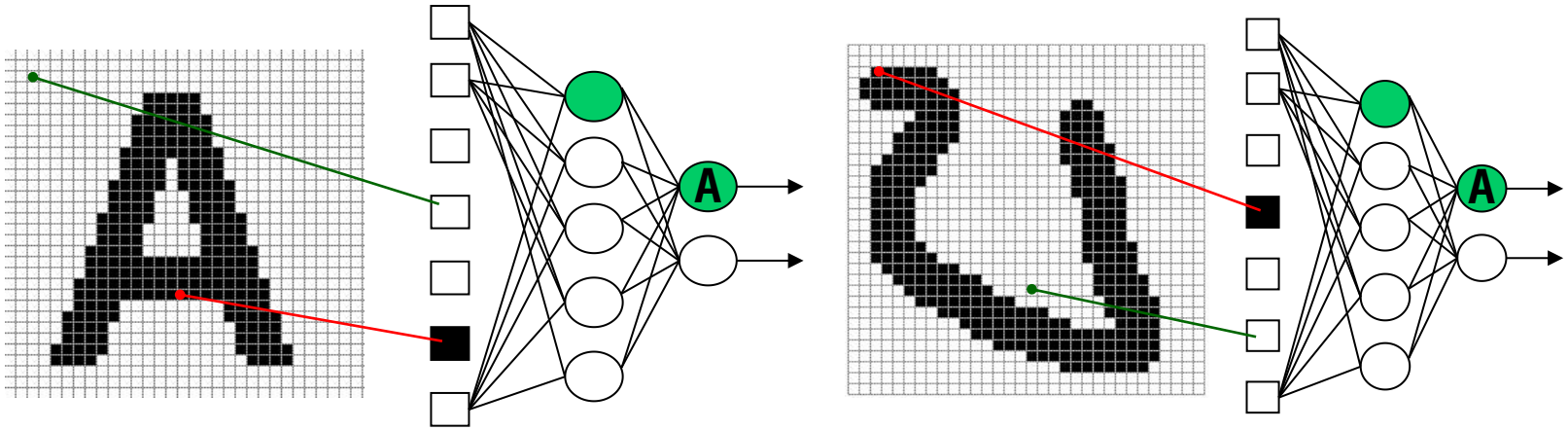
Gray scale patterns : $256^{32*32} = 256^{1024}$



**32 * 32 input image**

# Drawbacks of previous neural networks





# Improvement

- Fully connected network of sufficient size can produce outputs that are invariant with respect to such variations.
  - Training time
  - Network size
  - Free parameters

# Q & A