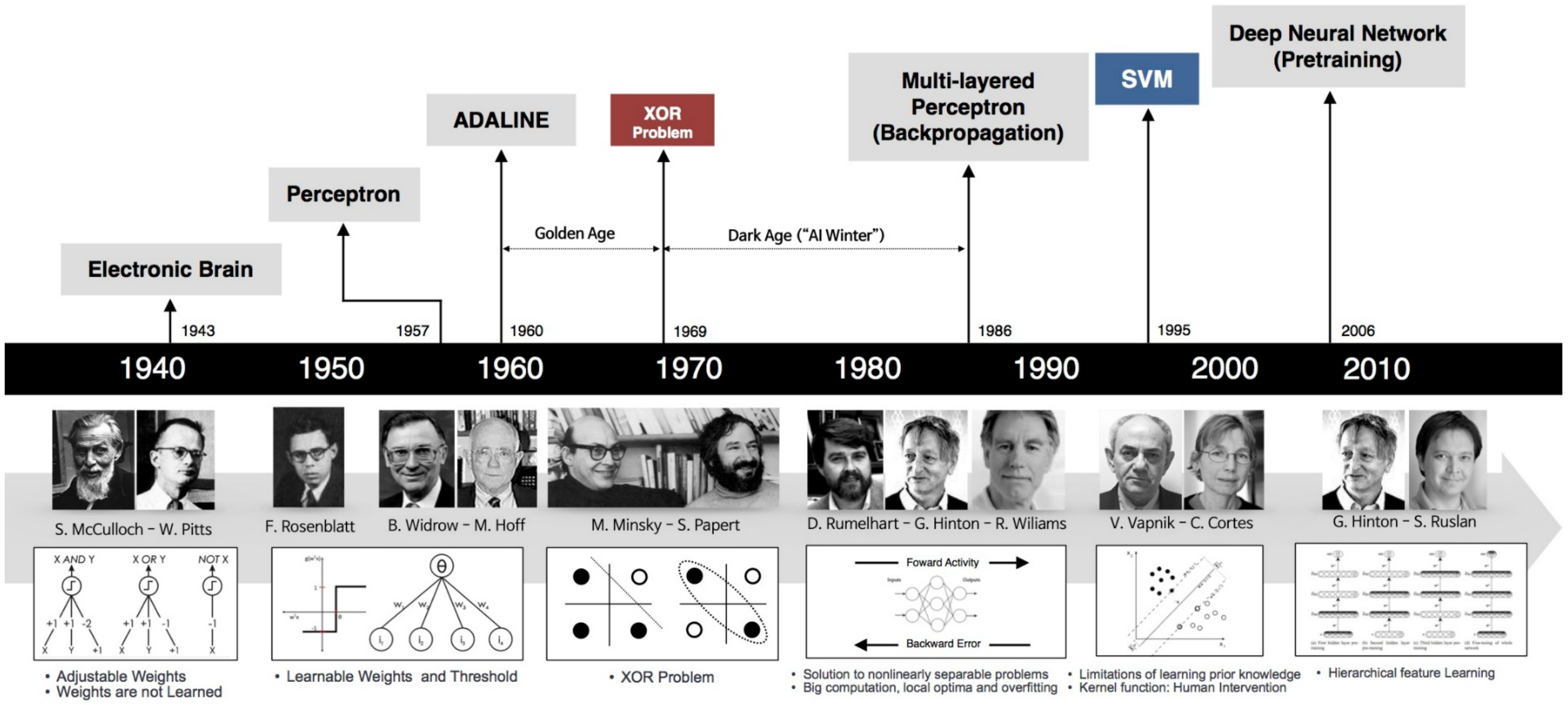
Artificial Neural Networks

A brief history


Electronic Brain
Perceptron
ADALINE
XOR Problem
Multi-layered Perceptron (Backpropagation)
SVM
Deep Neural Network (Pretraining)
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch – W. Pitts
F. Rosenblatt
B. Widrow – M. Hoff
M. Minsky – S. Papert
D. Rumelhart – G. Hinton – R. Wiliams
V. Vapnik – C. Cortes
G. Hinton – S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Artificial Neural Networks

- Humans
  - Neuron switching time ~ .001 s
  - Number of neurons ~ 10^{10}
  - Connections per neuron ~ 10^{10}
  - Scene recognition time ~ 0.1 s
  - 100 inference steps doesn’t seem like enough → much parallel computation
- Properties of Artificial Neural Networks (ANN’s)
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically

When Neural Nets

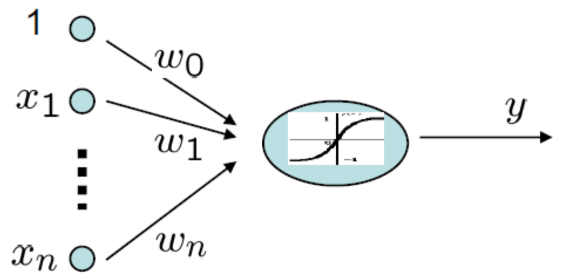
- Input is a high-dimensional discrete or real-data (e.g., raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is not important

Examples

1. Speech phoneme recognition[Waibel]
2. Image classification
3. Financial Prediction
4. Latest: RL / games (with video)

ANNs

- The basis of **neural networks** was developed in the 1940s-1960s
- The idea was to build **mathematical models** that might **"compute" in the same way that neurons** in the brain do
- As a result, neural networks are **biologically inspired**, though many of the **algorithms** developed for them are **not biologically plausible**
- Perform surprisingly well for many tasks



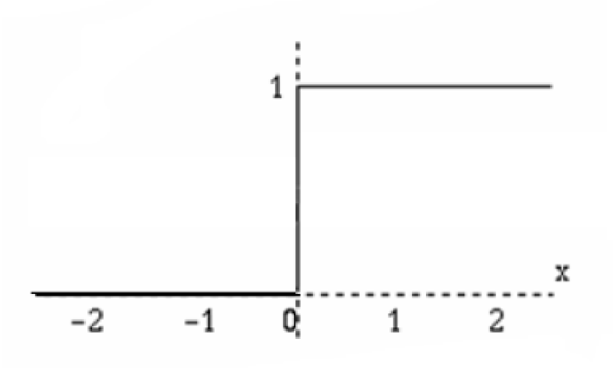
- Receives n inputs (plus a bias term)
- Multiplies each input by its weight
- Applies an activation function to the sum of results
- Outputs result

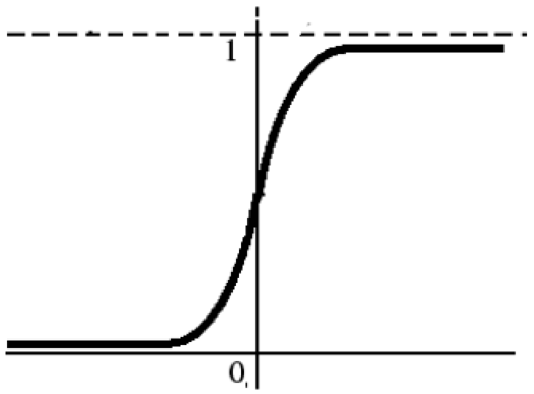
Activation Functions

- Controls whether the neuron is "active" or "inactive"
- Threshold function: outputs 1 if the input is positive and 0 otherwise – perceptron
- Logistic sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Differentiable and as we discussed earlier, this is a good property for learning



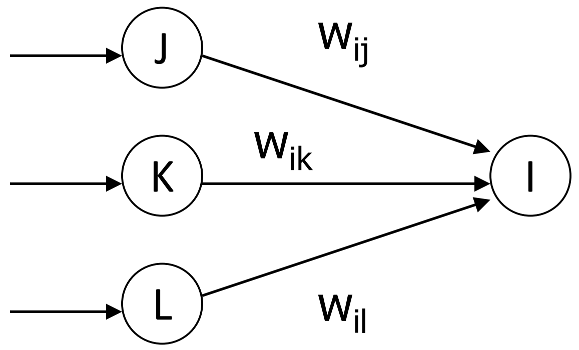


A neural network is a **directed graph** consisting of a collection of neurons (the **nodes**), directed **edges** (each with an associated **weight**), and a collection of **fixed binary inputs**

Connectionism

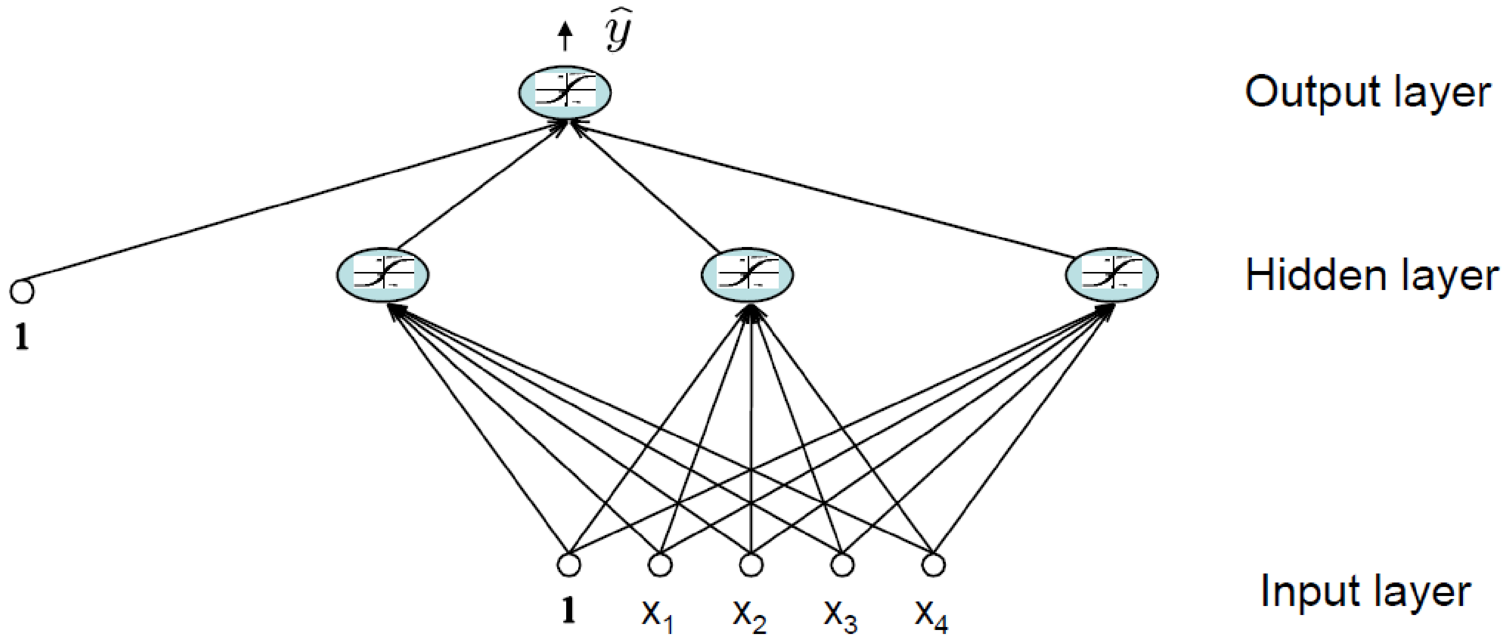
PERCEPTRONS (Rosenblatt 1957)

- earliest work in machine learning
- died out in 1960's (Minsky & Papert book)



$\text{Output}_i = F(W_{ij} \times \text{output}_j + W_{ik} \times \text{output}_k + W_{il} \times \text{output}_l\ )$

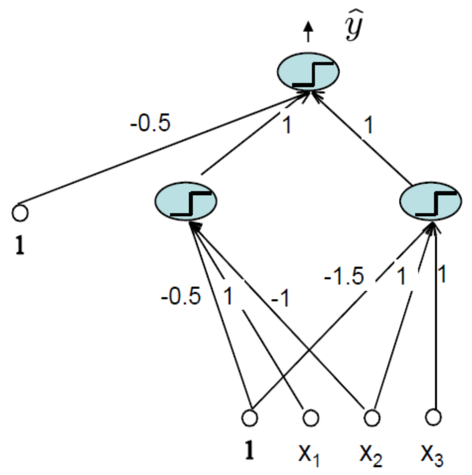
Multi-level Neural Network



- Each layer receives its inputs from the previous layer and forwards its outputs to the next – feed forward structure
- Output layer: sigmoid activation ftn for classification, linear activation ftn for regression problem
- Referred to as a two-layer network (two layer of weights)

Representational Power

- Any Boolean Formula
  - Consider the following boolean formula $(x_1 \wedge \neg x_2) \vee (x_2 \wedge x_3)$



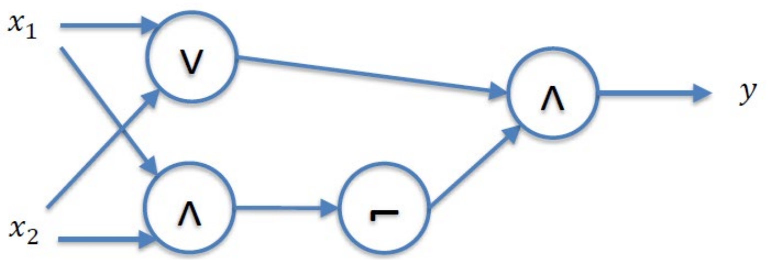
OR units

AND units

Recall that the **xor** function can be written as:

$$x_1 \oplus x_2 = (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$$

Can be expressed by combining **multiple perceptron units with multiple layers**!

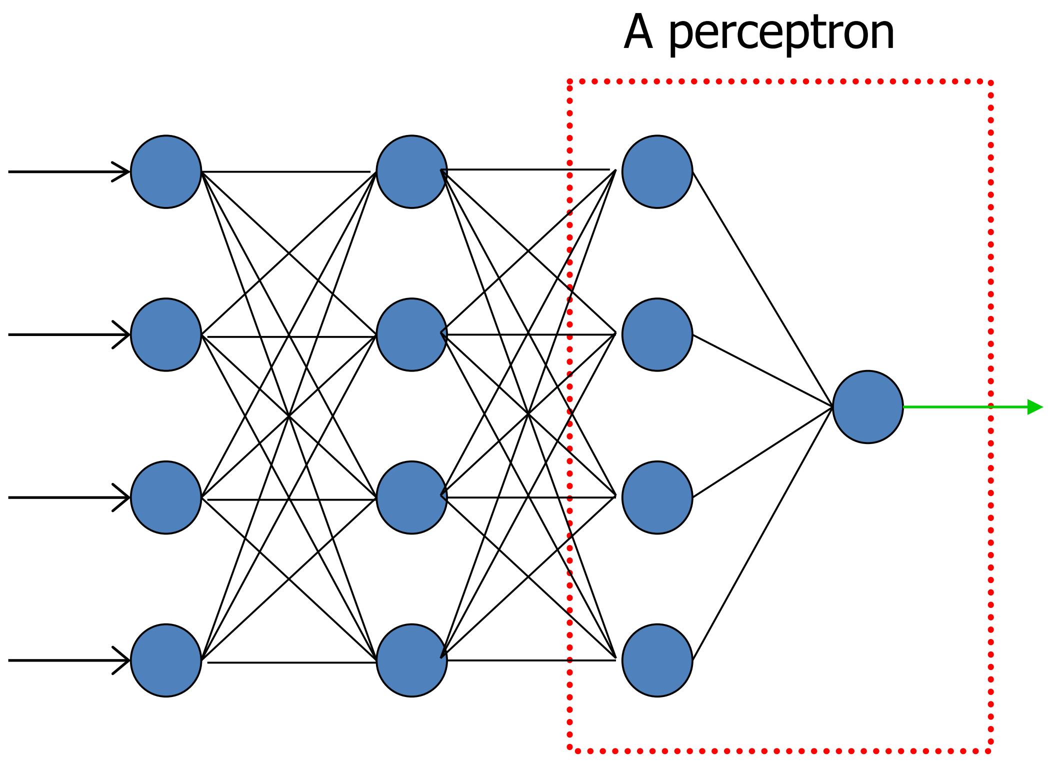


- Universal approximator
  - A two layer (linear output) network can approximate any continuous function on a compact input domain to arbitrary accuracy given a sufficiently large number of hidden units

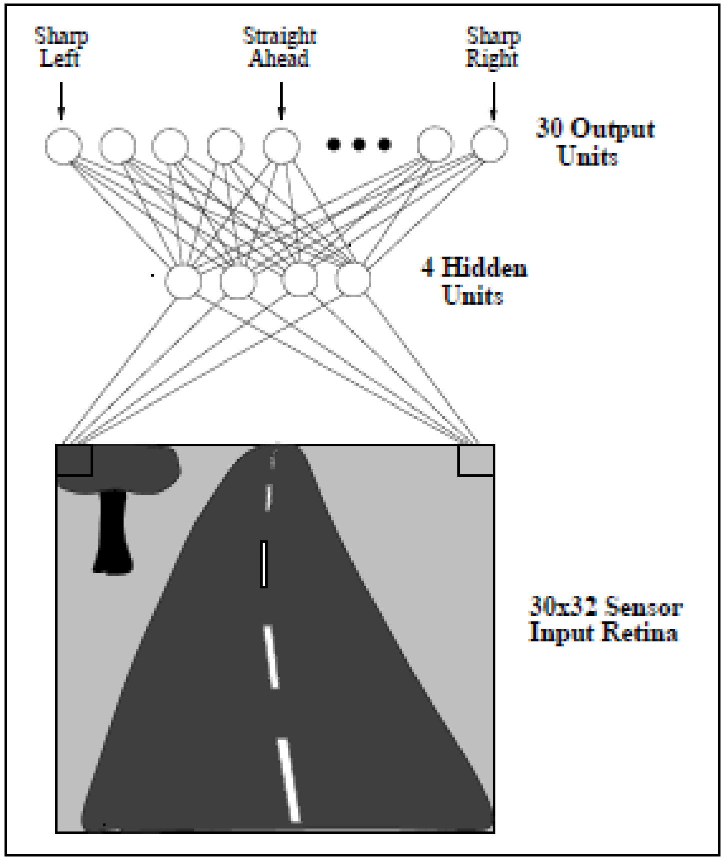
Hidden Units

<u>One View</u>

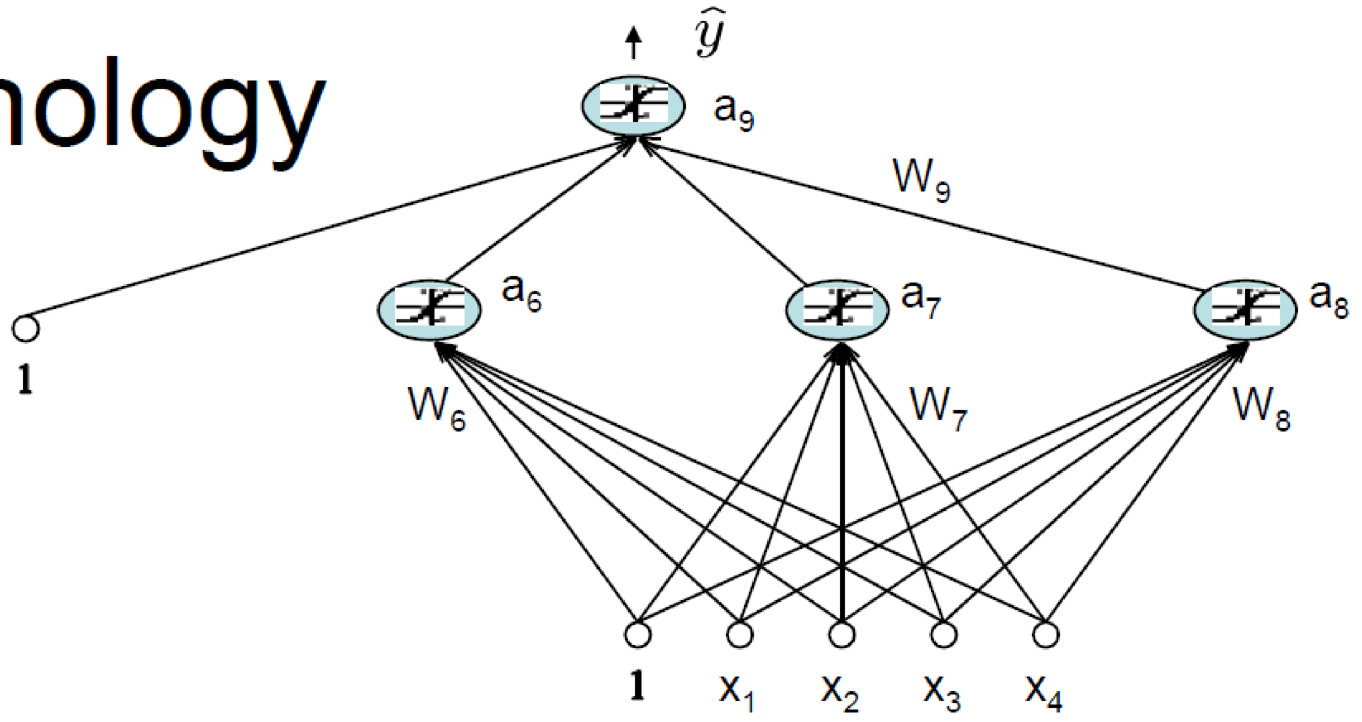
Allow a system to create its own <u>internal representation</u> – for which problem solving is easy



ALVINN drives 70 mph on highways



Terminology



- $X=[1, x_1, x_2, x_3, x_4]^T$ – the input vector with the bias term
- $A=[1, a_6, a_7, a_8]^T$ – the output of the hidden layer with the bias term
- W_i represents the weight vector leading to node *i*
- $w_{i,j}$ represents the weight connecting from the *j*th node to the *i*th node
  - $w_{9,6}$ is the weight connecting from a_6 to a_9
- We will use σ to represent the activation function (generally a sigmoid function), so

$$\hat{y} = \sigma(W_9 \cdot [1, a_6, a_7, a_8]^T) = \sigma(W_9 \cdot [1, \sigma(W_6 \cdot X), \sigma(W_7 \cdot X), \sigma(W_8 \cdot X)]^T)$$

Mean Squared Error

- We adjust the weights of the neural network to minimize the mean squared error (MSE) on training set.

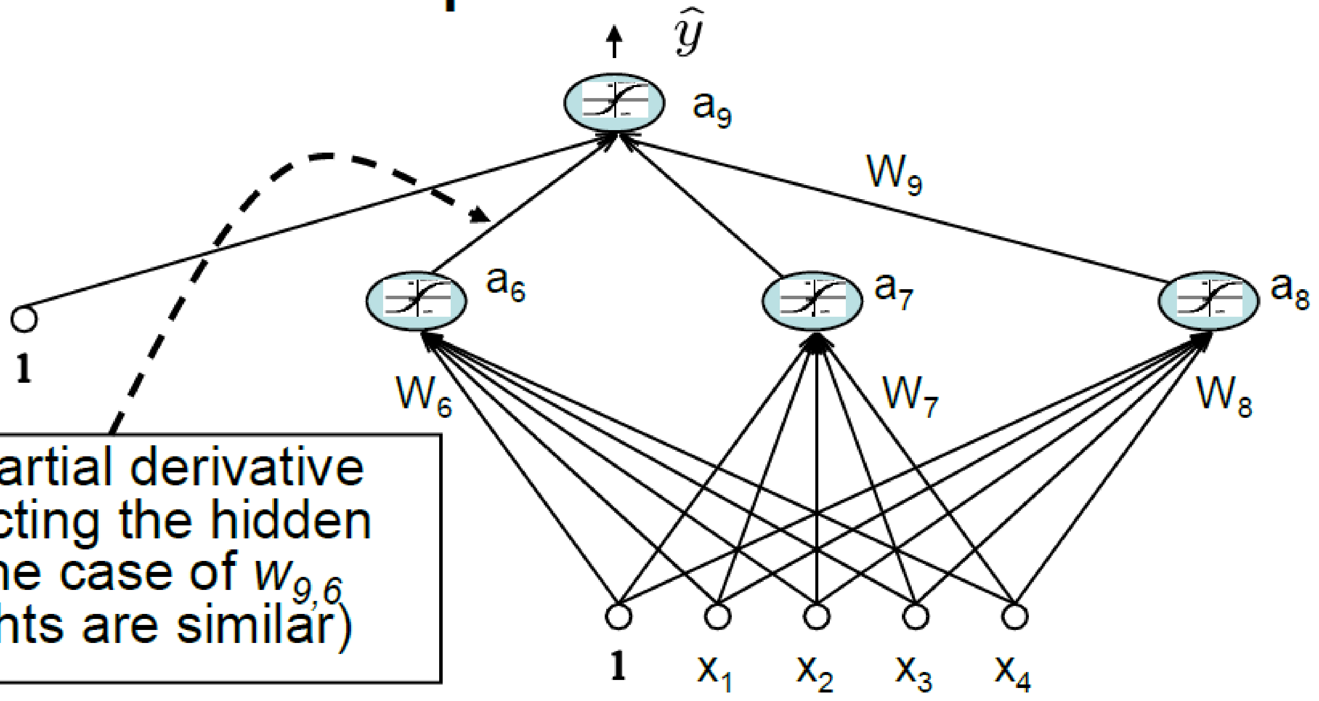
$$J(W) = \frac{1}{2}\sum_{i=1}^{N}(\widehat{y}^i - y^i)^2$$

$$J_i(W) = \frac{1}{2}(\widehat{y}^i - y^i)^2$$

- **Useful Fact**: the derivative of the sigmoid activation function is

$$\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

Gradient Descent: Output Unit

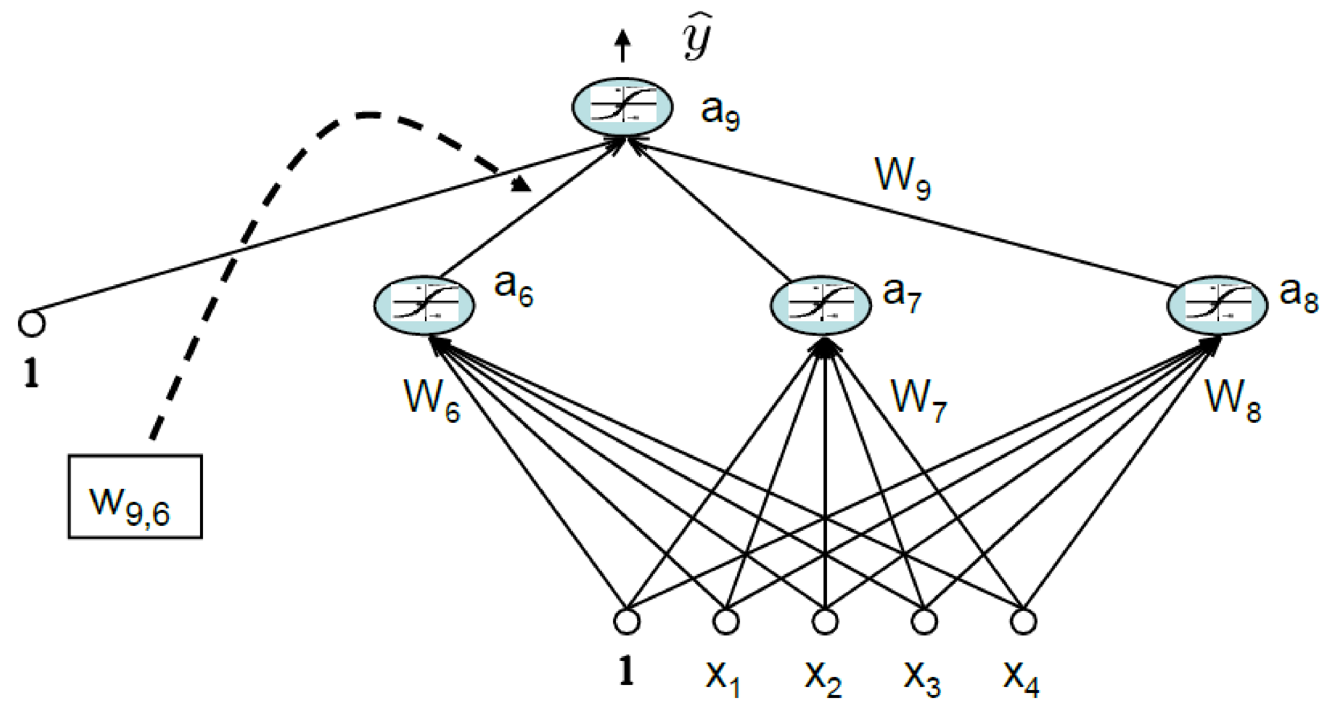


Lets compute the partial derivative wrt a weight connecting the hidden to output layer. In the case of $w_{9,6}$ we get: (other weights are similar)

hidden layer activation for i'th example

$$
\begin{aligned}
\frac{\partial J_i(W)}{\partial w_{9,6}} &= \frac{\partial}{\partial w_{9,6}} \frac{1}{2}(\hat{y}^i - y^i)^2 \\
&= \frac{1}{2} \cdot 2 \cdot (\hat{y}^i - y^i) \cdot \frac{\partial}{\partial w_{9,6}} (\sigma(W_9 \cdot A^i) - y^i) \\
&= (\hat{y}^i - y^i) \cdot \sigma(W_9 \cdot A^i)(1 - \sigma(W_9 \cdot A^i)) \cdot \frac{\partial}{\partial w_{9,6}} W_9 \cdot A^i \\
&= (\hat{y}^i - y^i)\hat{y}^i(1 - \hat{y}^i) \cdot a_6^i
\end{aligned}
$$

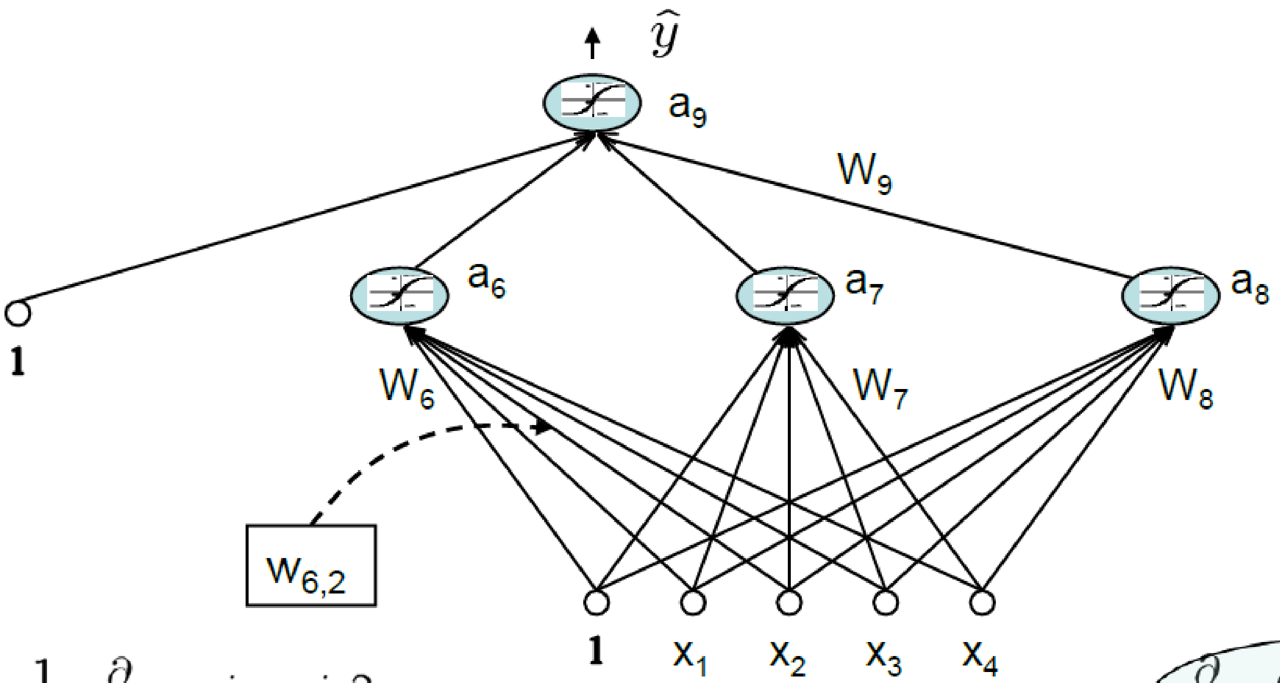
The Delta Rule



- Define $\delta_9^i = (\hat{y}^i - y^i)\hat{y}^i(1-\hat{y}^i)$

  then
$$\begin{aligned}\frac{\partial J_i(W)}{\partial w_{9,6}} &= (\hat{y}^i - y^i)\hat{y}^i(1-\hat{y}^i)\cdot a_6^i \\ &= \delta_9^i \cdot a_6^i\end{aligned}$$

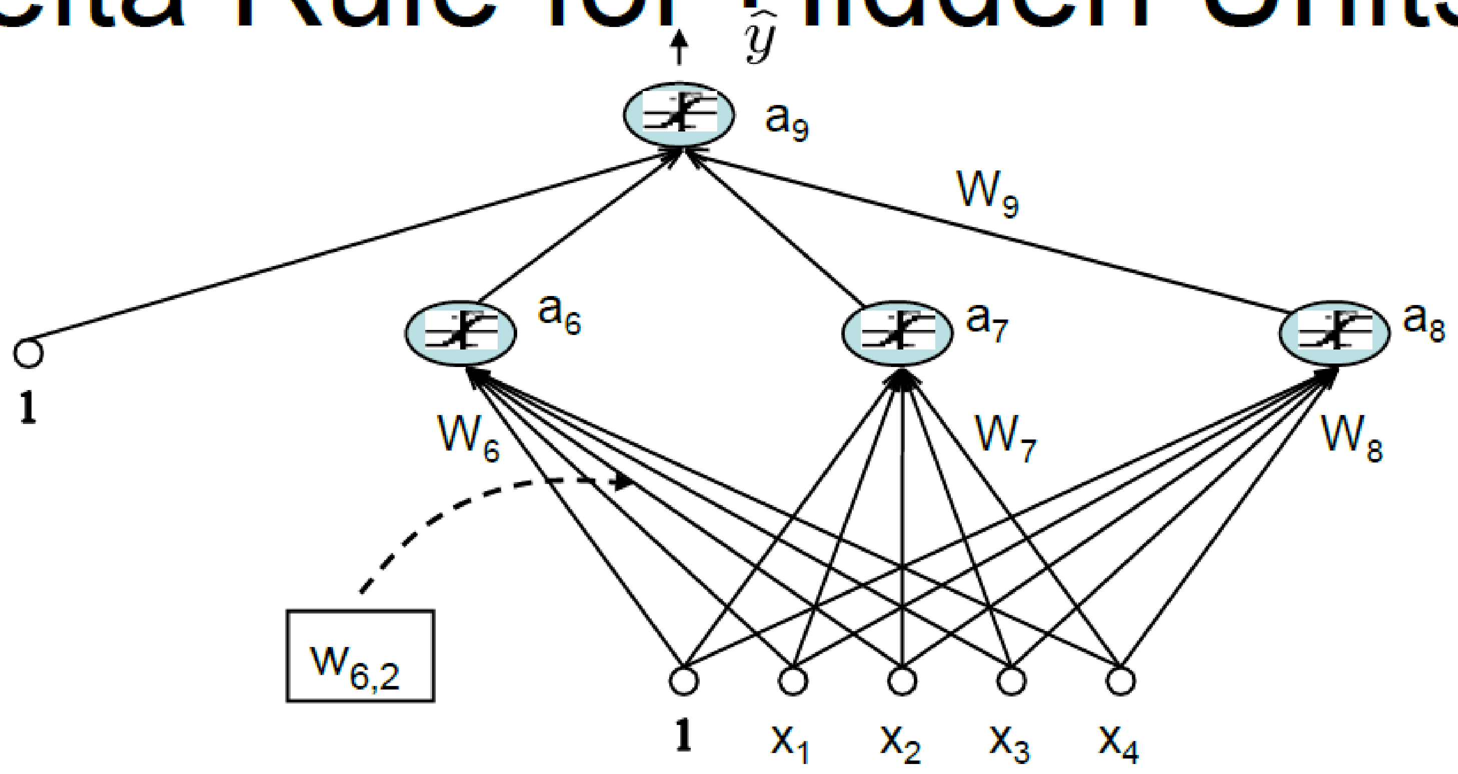
Derivation: Hidden Units



$$
\begin{aligned}
\frac{\partial J_i(W)}{\partial w_{6,2}} &= \frac{1}{2}\frac{\partial}{\partial w_{6,2}}(\hat{y}^i - y^i)^2 \\
&= (\hat{y}^i - y^i)\cdot\sigma(W_9\cdot A^i)(1-\sigma(W_9\cdot A^i))\cdot\frac{\partial}{\partial w_{6,2}}(W_9\cdot A^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot\frac{\partial}{\partial w_{6,2}}\sigma(W_6\cdot X^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot\sigma(W_6\cdot X^i)(1-\sigma(W_6\cdot X^i))\cdot\frac{\partial}{\partial w_{6,2}}(W_6\cdot X^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot a_6(1-a_6)\cdot x_2^i
\end{aligned}
$$

$$\frac{\partial}{\partial w_{6,2}}(W_9\cdot A^i) = \frac{\partial}{\partial w_{6,2}}(w_{9,6}\cdot a_6^i)$$

Delta Rule for Hidden Units



Define $\delta_6^i = \delta_9^i \cdot w_{9,6} \cdot a_6^i(1 - a_6^i)$

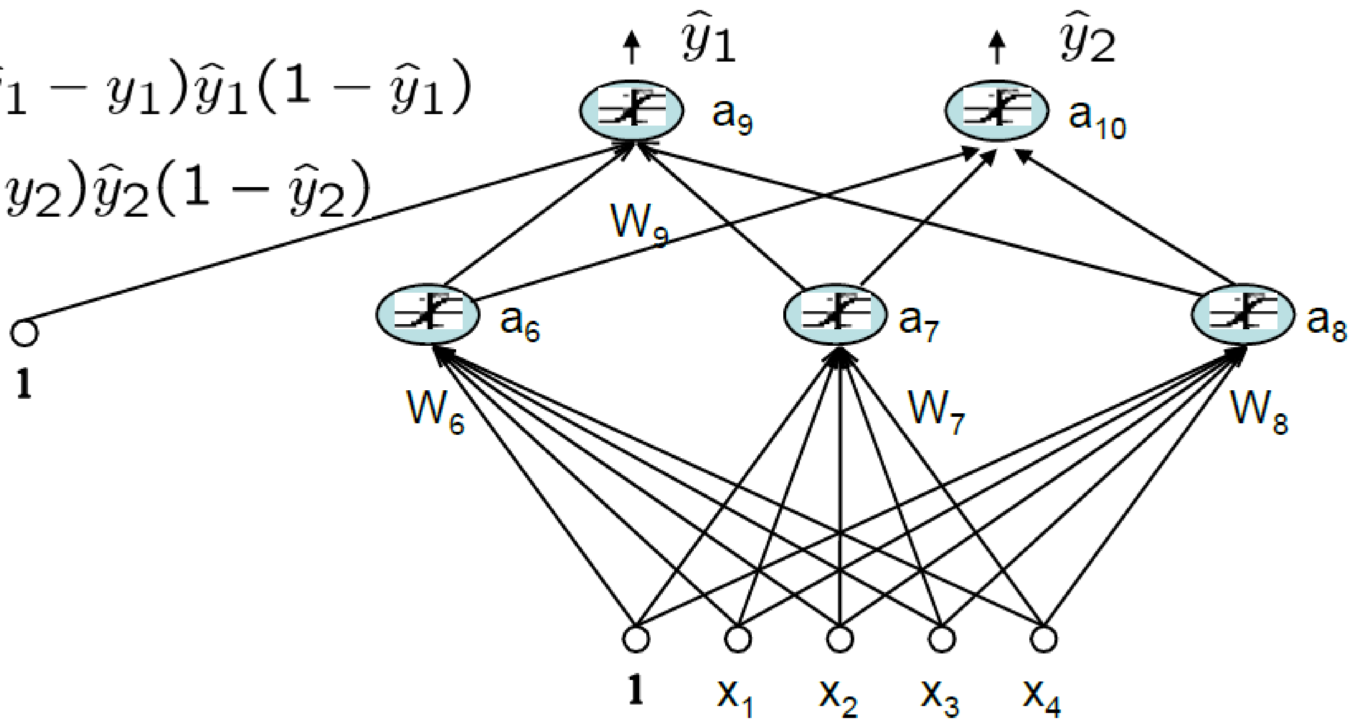
and rewrite as

$$\frac{\partial J_i(W)}{\partial w_{6,2}} = \delta_6^i \cdot x_2^i.$$

Networks with Multiple Output Units

$$\delta_9 = (\hat{y}_1 - y_1)\hat{y}_1(1 - \hat{y}_1)$$

$$\delta_{10} = (\hat{y}_2 - y_2)\hat{y}_2(1 - \hat{y}_2)$$



- We get a separate contribution to the gradient from each output unit.
- Hence, for input-to-hidden weights, we must sum up the contributions:

$$\delta_6 = a_6(1 - a_6)(w_{9,6}\delta_9 + w_{10,6}\delta_{10})$$

Backpropagation Algorithm

- **Forward Pass** Given X, compute a_u and $\hat{y}_v$ for hidden units u and output units v.
- **Compute Errors** Compute $\varepsilon_v = (\hat{y}_v - y_v)$ for each output unit v
- **Compute Output Deltas** $\delta_v = (\hat{y}_v - y_v)\, \hat{y}_v(1 - \hat{y}_v)$
- **Compute Hidden Deltas** $\delta_u = a_u(1 - a_u) \sum_v w_{v,u}\, \delta_v$
- **Compute Gradient**
  - Compute $\frac{\partial J_i}{\partial w_{v,u}} = \delta_v a_u^i$ for hidden-to-output weights.
  - Compute $\frac{\partial J_i}{\partial w_{u,j}} = \delta_u x_j^i$ for input-to-hidden weights.

$\downarrow$ Backpropagate error from layer to layer

- **For each weight take a gradient step**

$$w_{u,v} := w_{u,v} - \eta \frac{\partial J_i}{\partial w_{u,v}}$$

Backpropagation Training

- Create a three layer network with N hidden units, fully connect the network and assign small random weights
- Until all training examples produce correct output or MSE ceases to decrease
  - For all training examples, do

    Begin Epoch

      For each training example do

      - Compute the network output
      - Compute the error
      - Backpropagate this error from layer to layer and adjust weights to decrease this error

    End Epoch

Gradient Descent

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

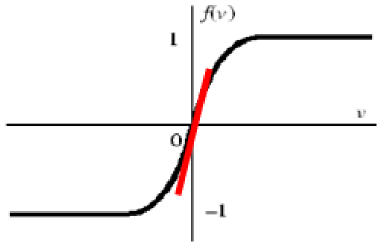
Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Proper Initialization

- Start in the "linear" regions

  keep all weights near zero, so that all sigmoid units are in their linear regions. Otherwise nodes can be initialized into flat regions of the sigmoid causing for very small gradients



- Break symmetry
  - If we start with the weights all equal, what will happen?
  - Ensure that each hidden unit has different input weights so that the hidden units move in different directions.
- Set each weight to a random number in the range

$$[-1, +1] \times \frac{1}{\sqrt{\text{fan-in}}}.$$

Where "fan-in" of weight $w_{v,u}$ is the number of inputs to unit v.

Batch, Online and Momentum Factor

- Batch. Sum the $\nabla_W J_i(W)$ for each example *i*. Then take a gradient descent step.
- Online. Take a gradient descent step with each $\nabla_W J_i(W)$ as it is computed (this is the algorithm we described)
- Momentum factor. Make the *t+1*-th update dependent on the *t*-th update

$$\Delta W^{(t+1)} = \nabla_W J(W^t)$$

$$\Downarrow$$

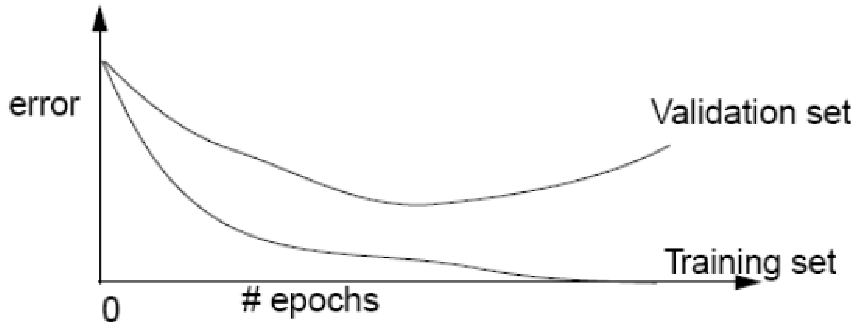
$$\Delta W^{(t+1)} = \alpha \Delta W^{(t)} + \nabla_W J(W^t)$$

α is called the momentum factor, and typically take values in the range [0.7, 0.95]

This tends to keep weight moving in the same direction and improves convergence.

Overtraining Prevention

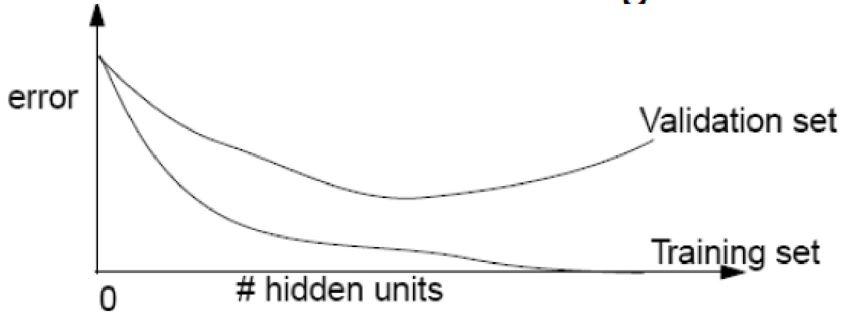
- Running too many epochs may overtrain the network and result in overfitting. Tries too hard to exactly match the training data.



- Keep a validation set and test accuracy after every epoch. Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond this.
- To avoid losing training data to validation:
  - Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance.
    - We will discuss cross-validation later in the course
  - Train on the full data set using this many epochs to produce the final result.

Over-fitting Prevention

- Too few hidden units prevent the system from adequately fitting the data and learning the concept.
- Too many hidden units leads to over-fitting.



- Can also use a validation set or cross-validation to decide an appropriate number of hidden units.
- Another approach to preventing over-fitting is ***weight decay***, in which we multiply all weights by some fraction between 0 and 1 after each epoch.
  - Encourages smaller weights and less complex hypotheses.
  - Equivalent to including an additive penalty in the error function proportional to the sum of the squares of the weights of the network.

ANNs as Universal Approximators

- Boolean Functions
  - Need one layer of hidden units to represent exactly
- Continuous Functions
  - Approximation to arbitrarily small error with one (possibly quite 'wide') layer of hidden units
- Arbitrary Functions
  - Any function can be approximated to arbitrary precision with two layers of hidden units
  - Overfitting a huge problem with two or more layers of hidden units

ANN Training summary

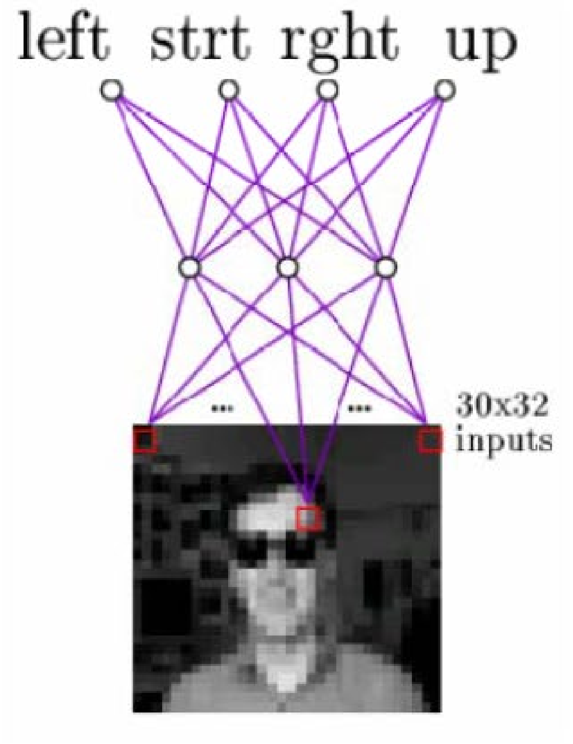
- Compute weighted sums to make decisions
- Use gradient descent to adjust weights in order to reduce error

  Only find <u>local</u> minima, though
- Good accuracy
- Slow training

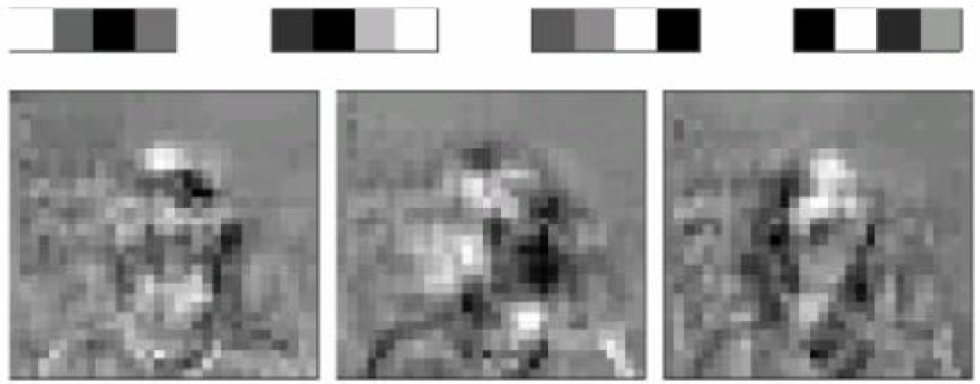
  Conjugate gradient is one way to speedup
- Learned models hard to interpret

Summary

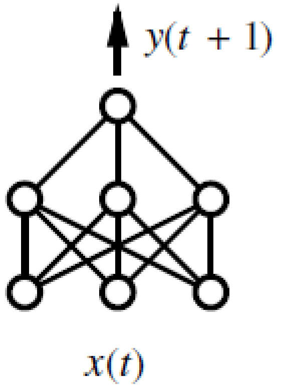
- Neural Networks can represent complex decision boundaries
  - Variable size. Any boolean function can be represented. Hidden units can be interpreted as new features
- Learning Algorithms for neural networks
  - Local Search.
  - Batch or Online
- Because of the large number of parameter
  - Learning can be slow
  - Prone to overfitting
  - Training is somewhat of an art



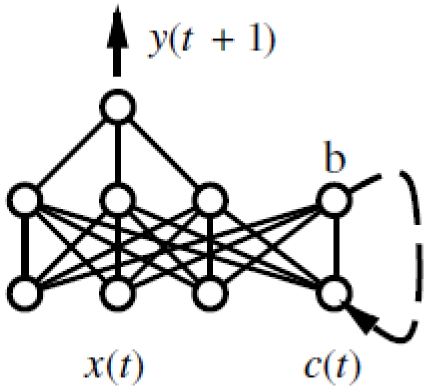
Learned Weights



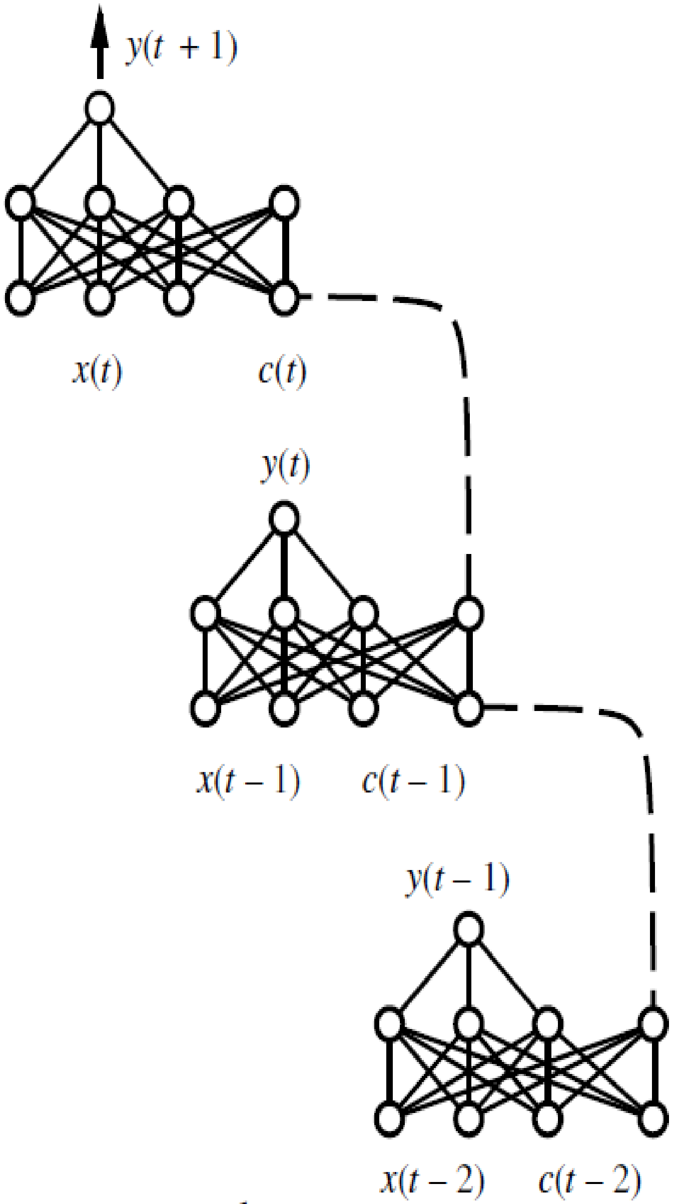
Typical input images



(*a*) Feedforward network



(*b*) Recurrent network



(*c*) Recurrent network unfolded in time