

Neural Networks and Deep Learning

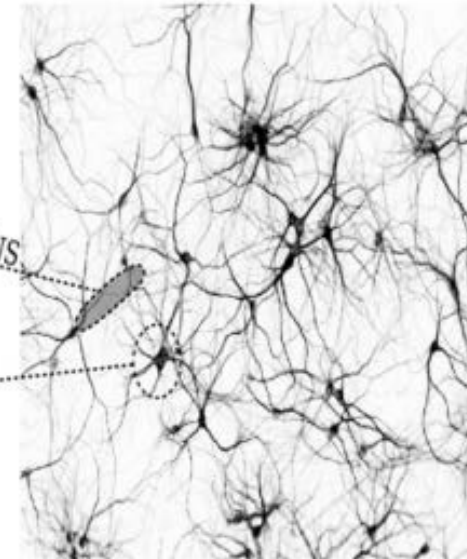
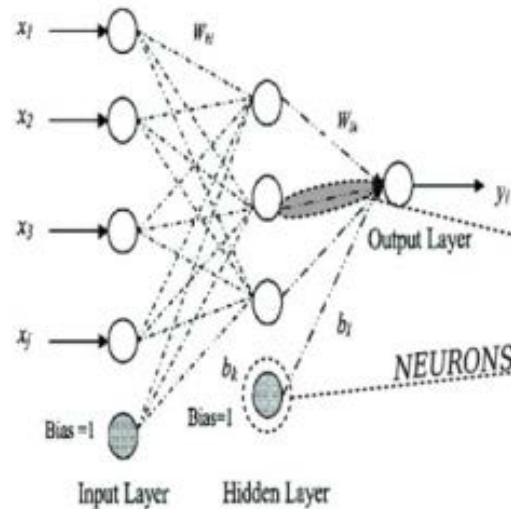
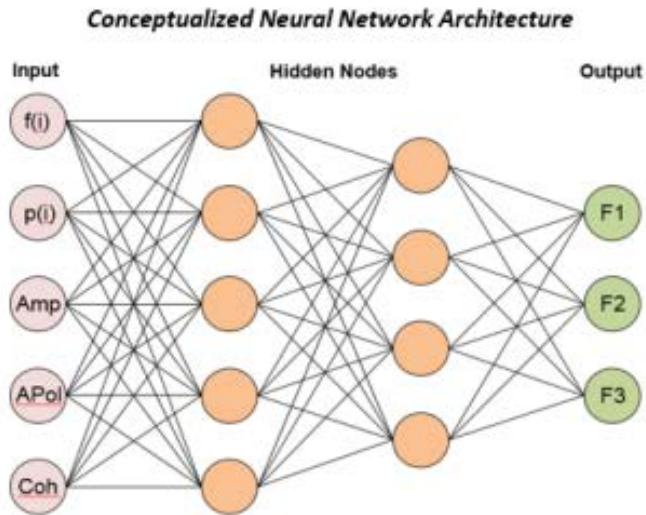
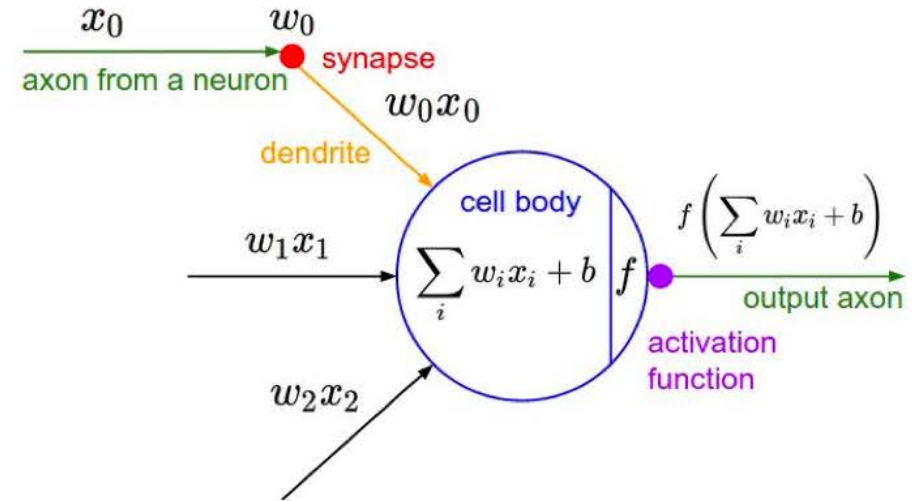
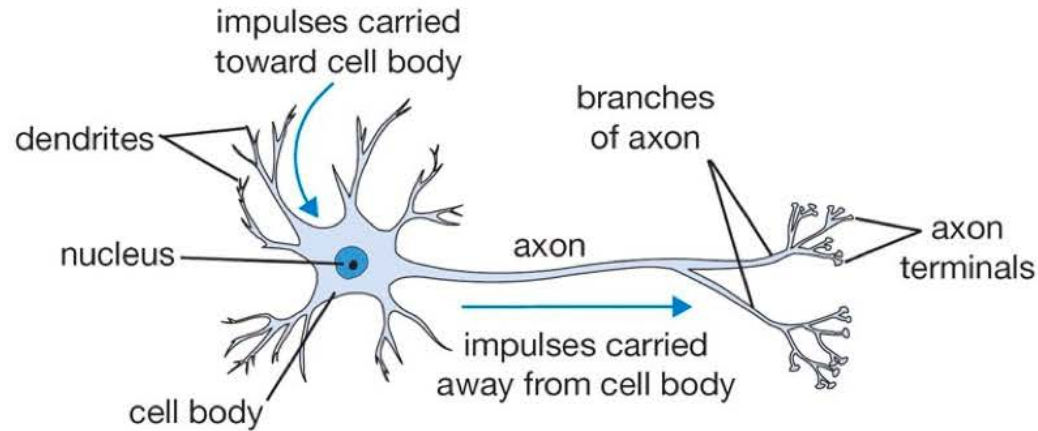
Brain inspired way to learn from patterns

COURSE: CS60045

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg



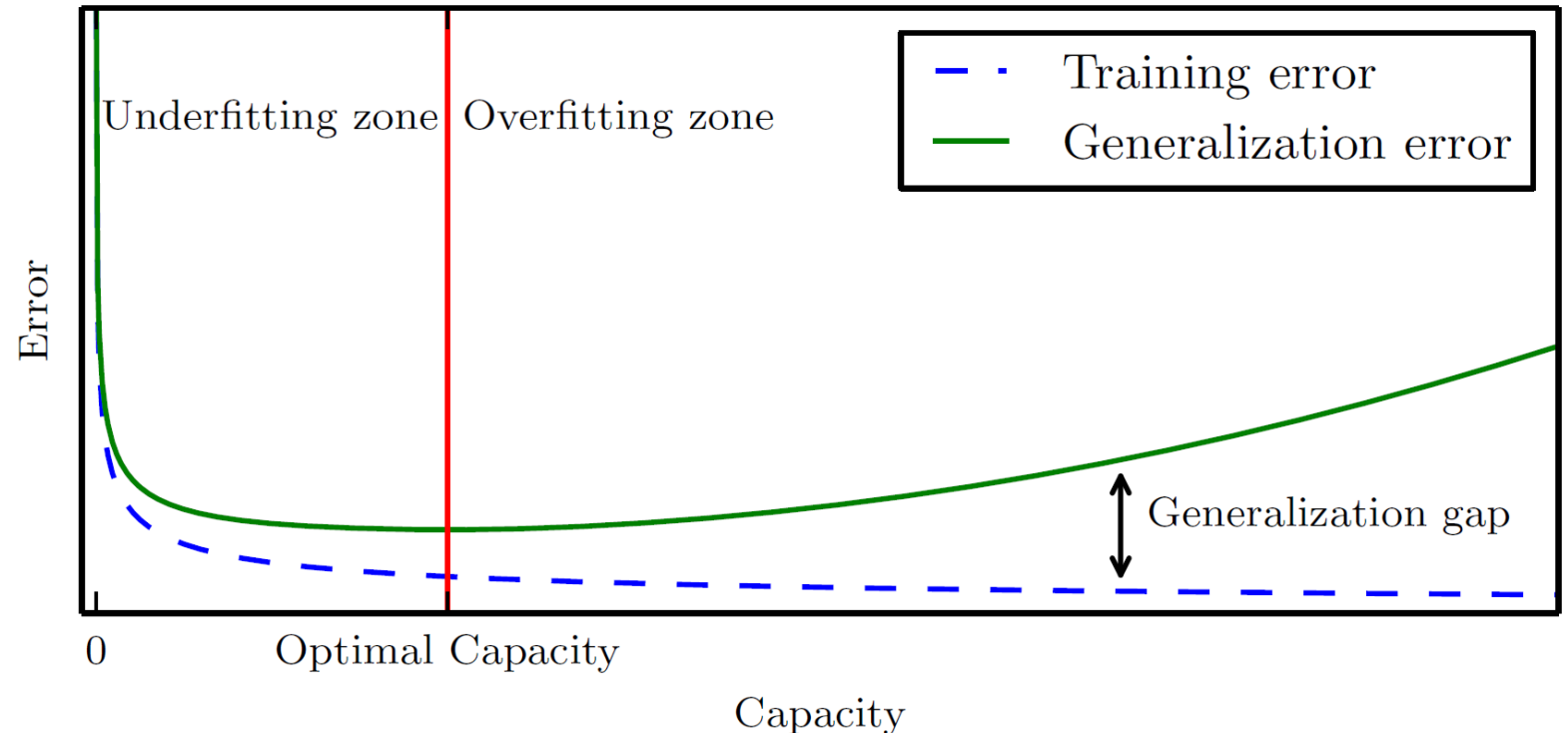
Brain inspired computing



- Simple units
- The power is in the network

Preliminaries

- Deciding the *capacity* of the model
 - Under-fitting, if the capacity is weak
 - Over-fitting, if the capacity is unnecessarily large
- Neural network offers a generic model, which offers:
 - Structural variants, so as to scale up / down the capacity
 - Various types of *activation functions*, which enables the modeling of various types of functions.



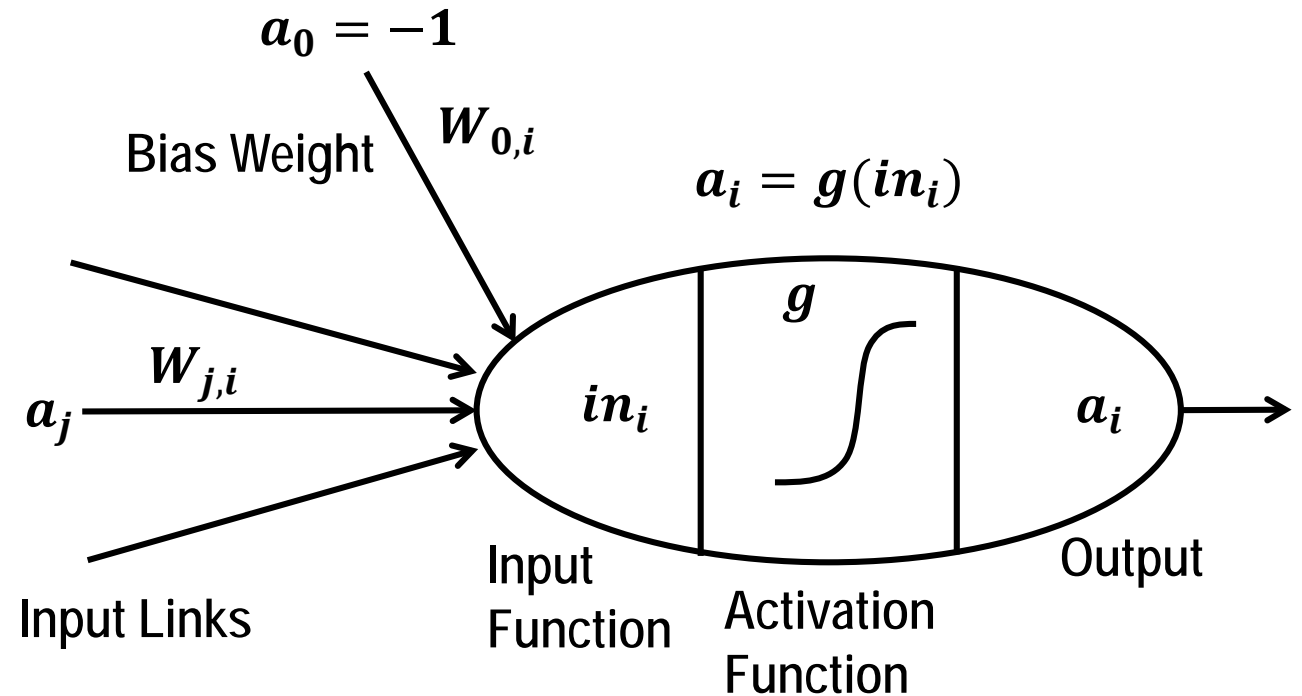
Neural Networks

A neural network consists of a set of nodes (neurons/units) connected by links

- Each link has a numeric weight

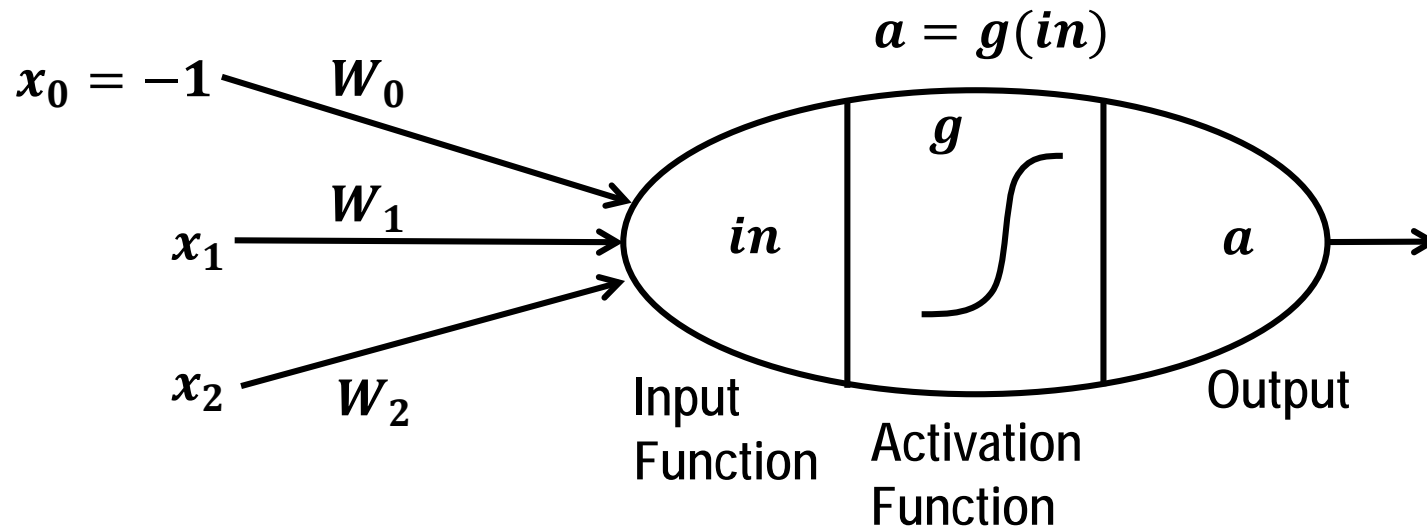
Each unit has:

- a set of input links from other units,
- a set of output links to other units,
- a current activation level, and
- an activation function to compute the activation level in the next time step.



$$in_i = \sum_{j=0}^n W_{j,i} a_j \quad a_i = g(in_i) = g\left(\sum_{j=0}^n W_{j,i} a_j\right)$$

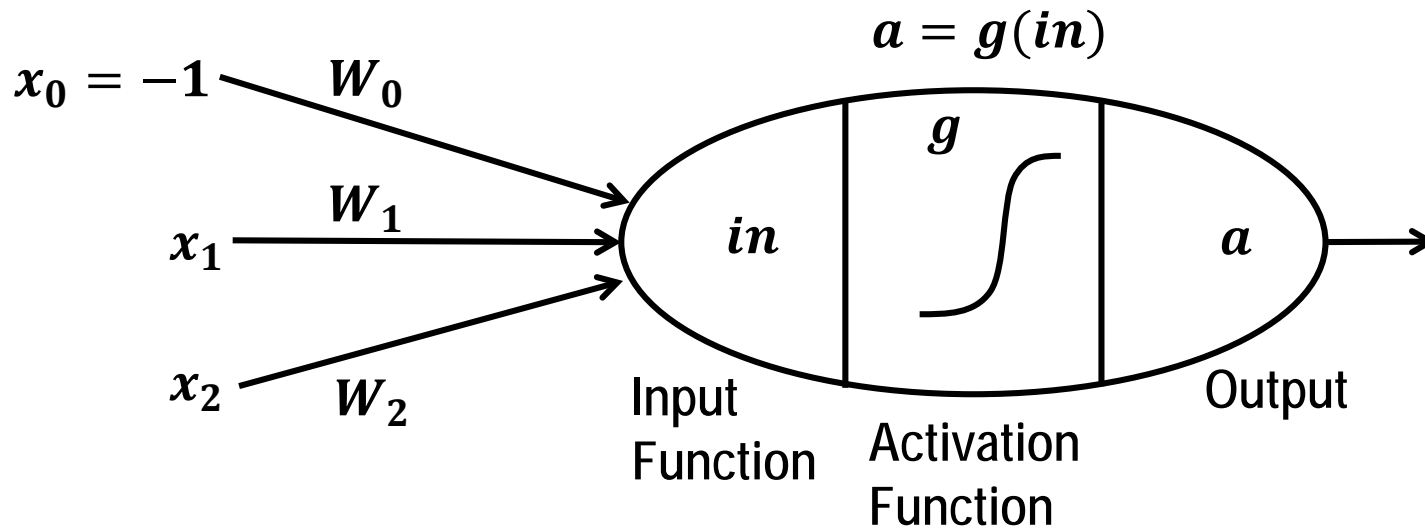
Perceptron



$$in = \sum_{j=0}^2 W_j x_j \quad a = \begin{cases} 0 & \text{if } in \leq 0 \\ 1 & \text{if } in > 0 \end{cases}$$

Studying a perceptron helps us to understand the limitations in capacity and the corresponding inability to model certain types of functions.

Perceptron



Linear Function:

$$in = x_1W_1 + x_2W_2 - W_0$$

$$a = \begin{cases} 0 & \text{if } in \leq 0 \\ 1 & \text{if } in > 0 \end{cases}$$

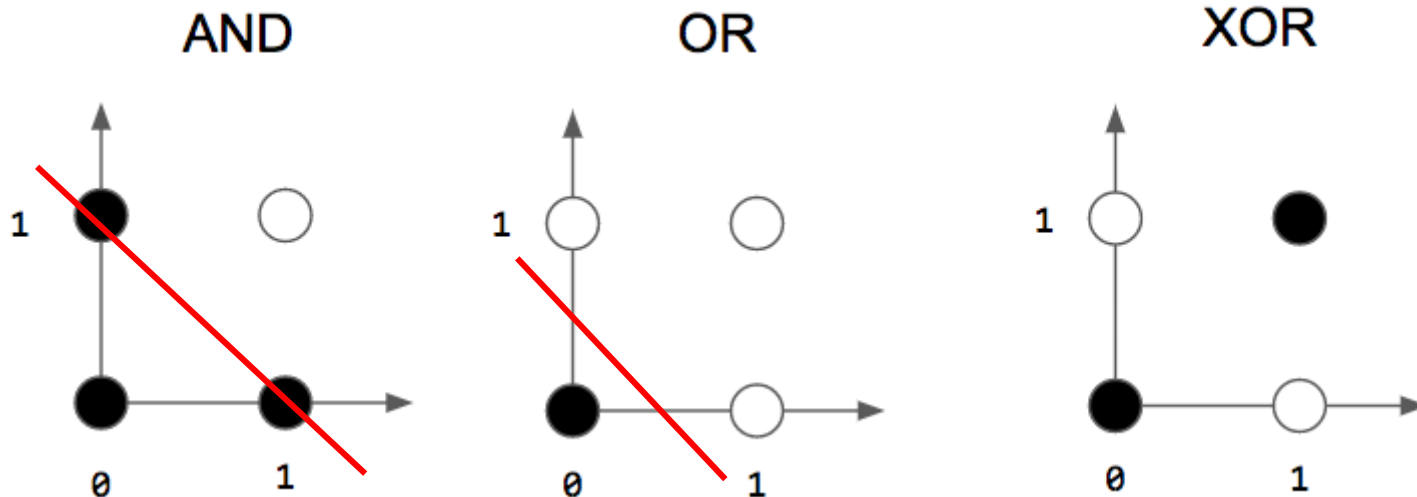
AND: $W_1 = 1, W_2 = 1, W_0 = 1$

$$in = x_1 + x_2 - 1$$

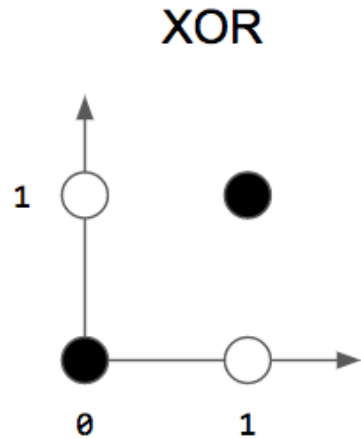
OR: $W_1 = 2, W_2 = 2, W_0 = 1$

$$in = 2x_1 + 2x_2 - 1$$

What about XOR?

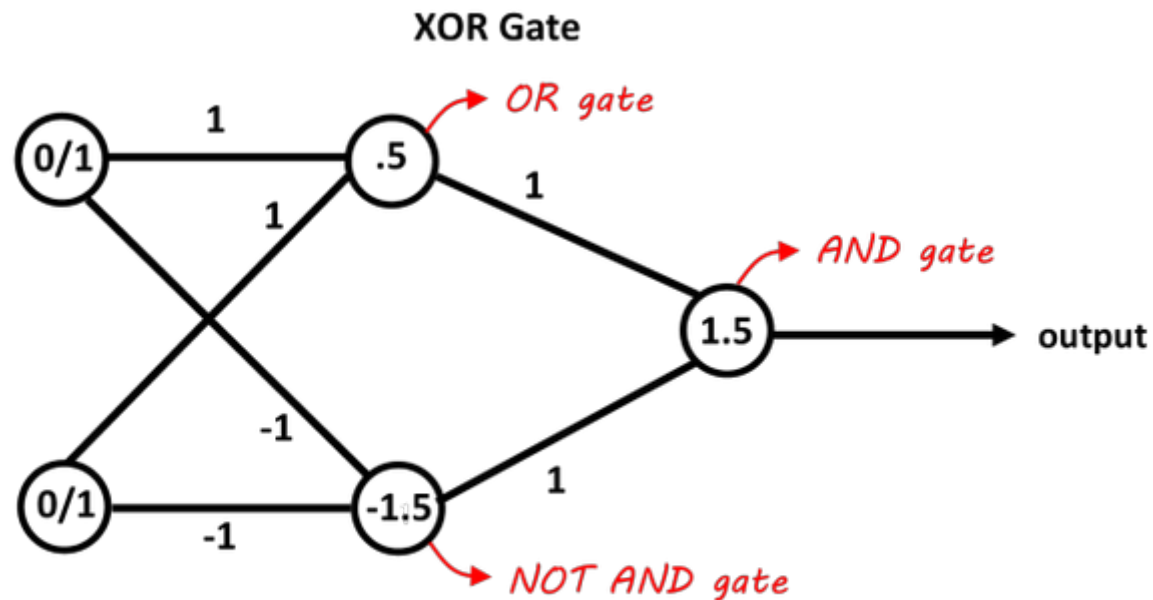


Multiple Layers Increase the Capacity



The black and white dots are not *linearly separable*, that is, no linear function of the following form separates them:

$$in = x_1W_1 + x_2W_2 - W_0$$



With two layers, it is possible to model the XOR function.

Supervised Learning by back-propagating errors

The basic idea:

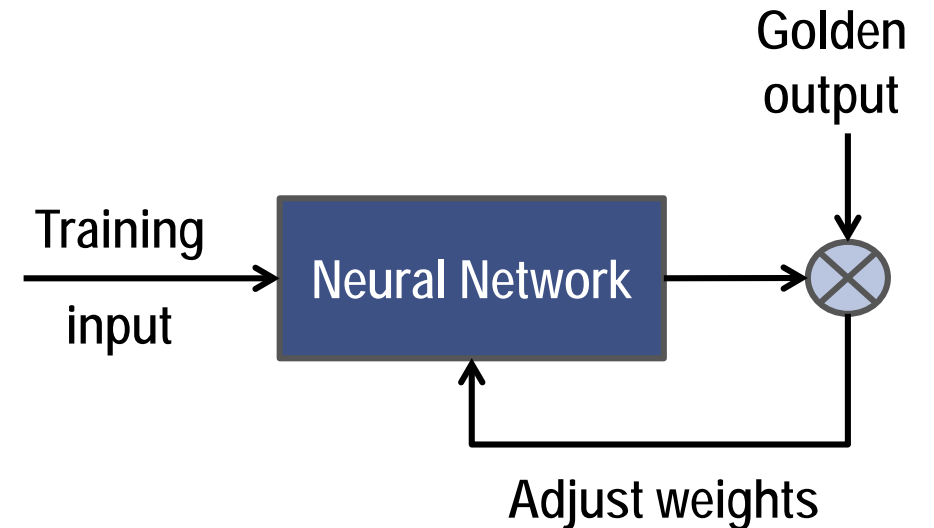
- We compute the output error as:

$$\text{Error} = \text{golden output (y)} - \text{output of network (a)}$$

- The training error function computed over all training data is:

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

- We wish to find values of W_j such that E is minimum over the training data
- For this purpose we may iteratively do the following:
 - Present a training sample to the network
 - Compute the error for this output
 - Factorize the error in proportion to the contribution of the nodes and readjust the weights accordingly



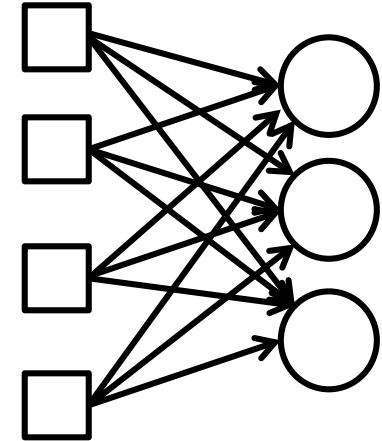
Learning in Single Layered Networks

Idea: Optimize the weights so as to minimize error function:

$$E = \frac{1}{2} \text{Err}^2 = \frac{1}{2} \left(y - g \left(\sum_{j=0}^n W_j x_j \right) \right)^2$$

We can use gradient descent to reduce the squared error by calculating the partial derivative of E with respect to each weight.

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



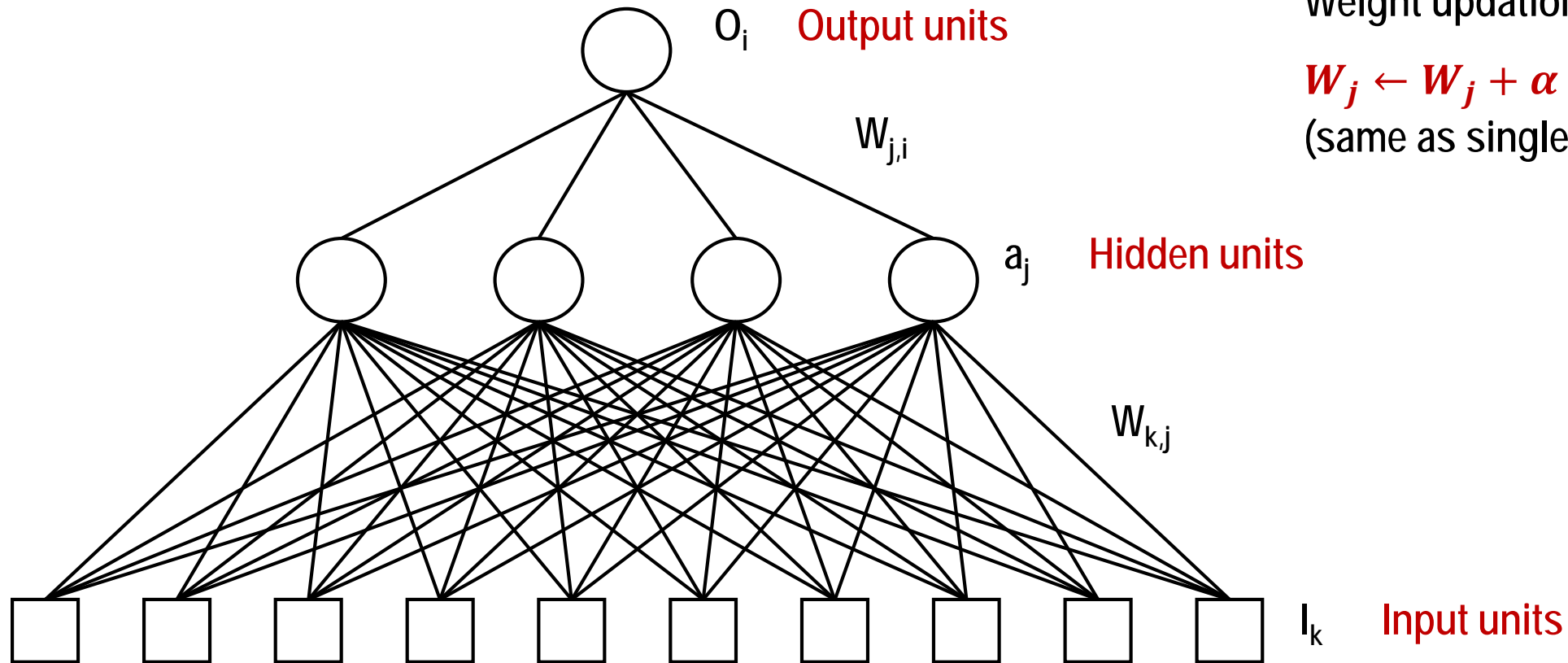
Weight update rule:

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

where α is the learning rate

We purposefully eliminate a **fraction** of the error through the weight adjustment rule, but **not the whole** of it. Why?

Multi-Layer Feed-Forward Network



Weight updation rule at the output layer:

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

(same as single layer)

In multilayer networks, the hidden layers also contribute to the error at the output.

- So the important question is: *How do we revise the hidden layers?*

Back-Propagation Learning

- To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes
- We do an error back-propagation, defining error as $\Delta_i = Err_i \times g'(in_i)$
- The idea is that a hidden node j is *responsible* for some fraction of the error in each of the output nodes to which it connects
- Thus the Δ_j values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer.

- The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

- The update rule for the hidden layers is:
 $W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$

The mathematics behind the updation rule

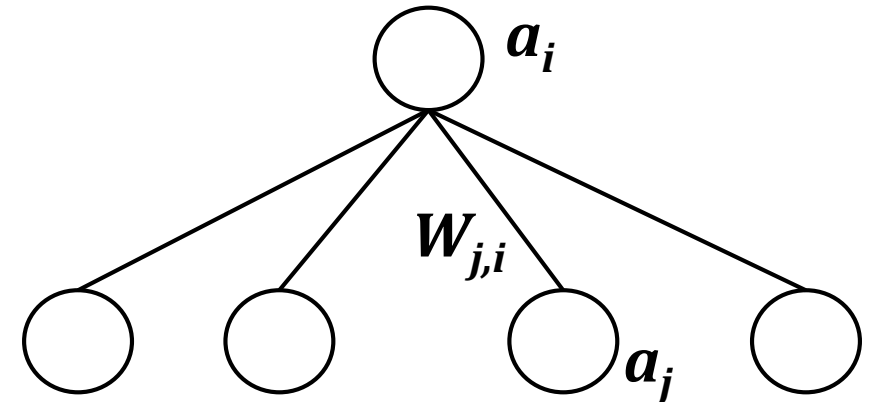
The squared error on a single example is defined as:

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

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $W_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $W_{j,i}$

$$\begin{aligned} \frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i) \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \frac{\partial g(in_i)}{\partial W_{j,i}} = -(y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{j,i}} \\ &= -(y_i - a_i) g'(in_i) \frac{\partial}{\partial W_{j,i}} \left(\sum_j W_{j,i} a_j \right) \\ &= -(y_i - a_i) g'(in_i) a_j = -a_j \Delta_i \end{aligned}$$

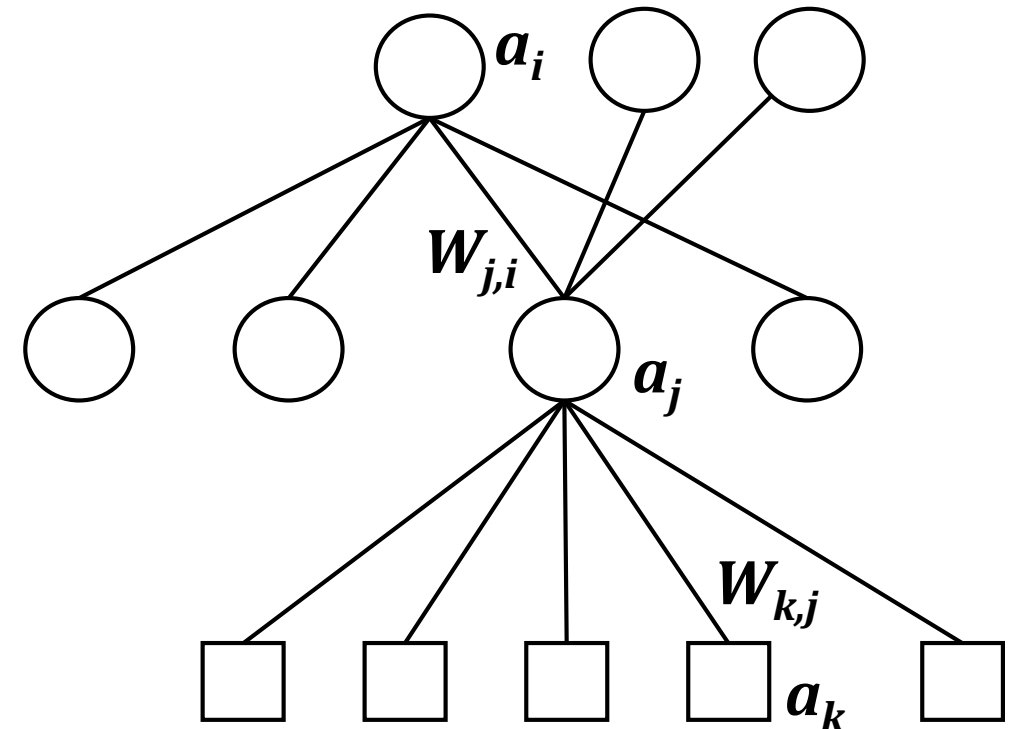
$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$



The mathematics contd.

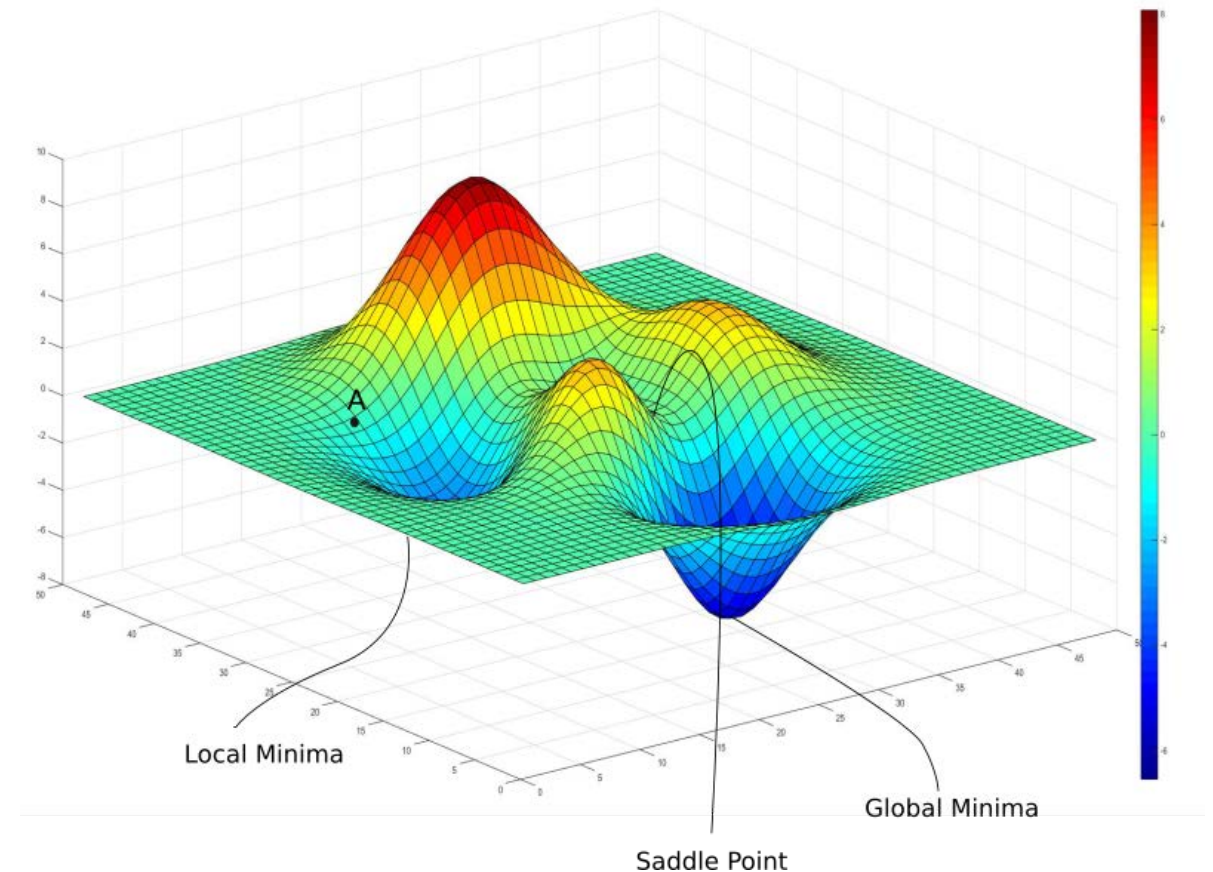
$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= - \sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} = - \sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left(\sum_j W_{j,i} a_j \right) = - \sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = - \sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} \left(\sum_k W_{k,j} a_k \right) \\ &= - \sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j\end{aligned}$$

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$



Problems with this Learning

- The weight updation rules define a single step of gradient descent
- Gradient descent may reach a local minima
 - The minimum training error reached at the end of training is not the best
- The final network is not explainable. We do not know what the network has learned.
 - For a single layer network, the error can be explained in terms of the inputs and the weights
 - In a multi-layer network, the hidden layers do not make any sense to the end user

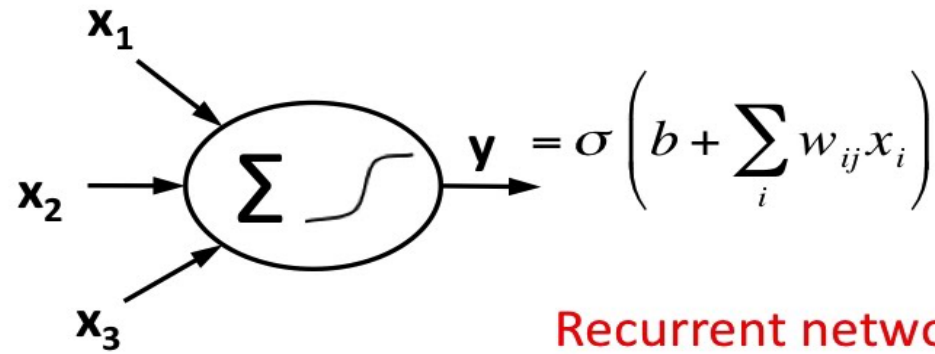


Convolutional and Recurrent Neural Networks

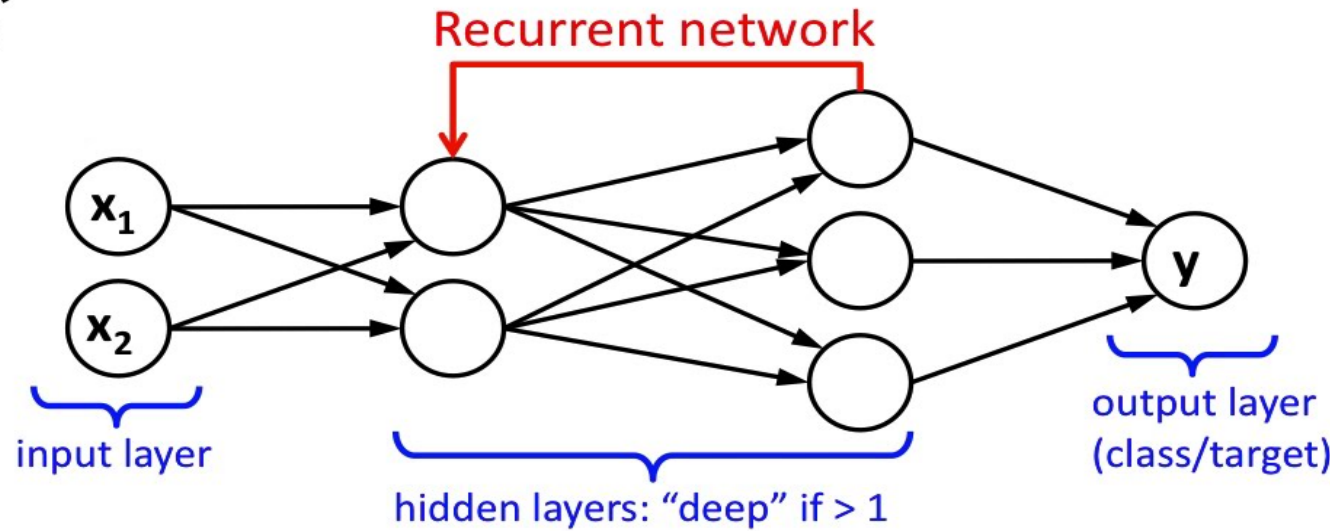
- *Convolution is useful for learning artifacts that have a small locality of reference*
- *Recurrence is useful for learning sequences*

Types of Neural Networks

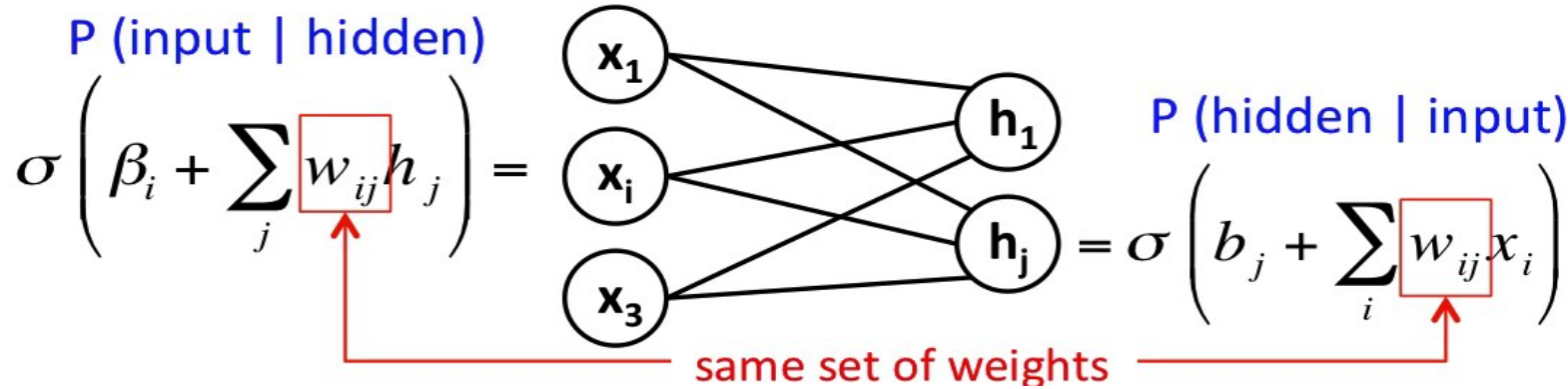
Activation Functions	
Name	Formula
Identity	$A(x) = x$
Sigmoid	$A(x) = \frac{1}{1 + e^{-x}}$
Tanh	$A(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Step	$A(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$



Single neuron: perceptron,
linear / logistic regression



Feed-forward network
(no cycles) -- non-linear
classification & regression



Symmetric (RBM)
unsupervised, trained
to maximize likelihood
of input data
a mixture model

The Convolution Operation

Suppose we are tracking the location of a spaceship with a laser sensor.

- Our laser sensor produces a single output $x(t)$, the position of the spaceship at time t
- Suppose that our laser sensor is somewhat noisy, and therefore we wish to take the average of multiple measurements.
- More recent measurements have more weight, so we need a weighting function $w(a)$, which returns the weight of measurement taken at the past time, a .

$$s(t) = \int x(a)w(t - a)da = (x * w)(t)$$

This operation is called *convolution*. The first argument, $x()$, is called the *input*, and the second argument, $w()$, is called the *kernel*.

Discrete Convolution

If we assume that x and w are defined only on integer t , we can define discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$




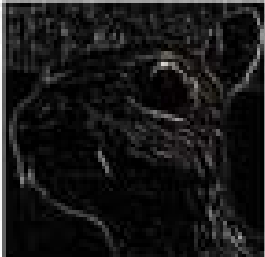

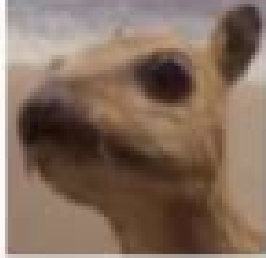
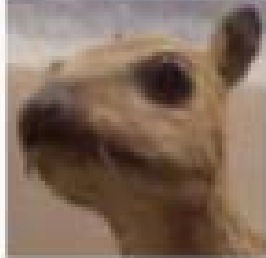
Convolution can also be defined over more than one axis at a time. For example, if we use a two dimensional image I as our input, we may want to use a two dimensional kernel:

$$s(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i-m, j-n)$$

Convolution is commutative, that is, we can also write (by replacing m by $i-m$ and n by $j-n$):

$$s(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n)$$

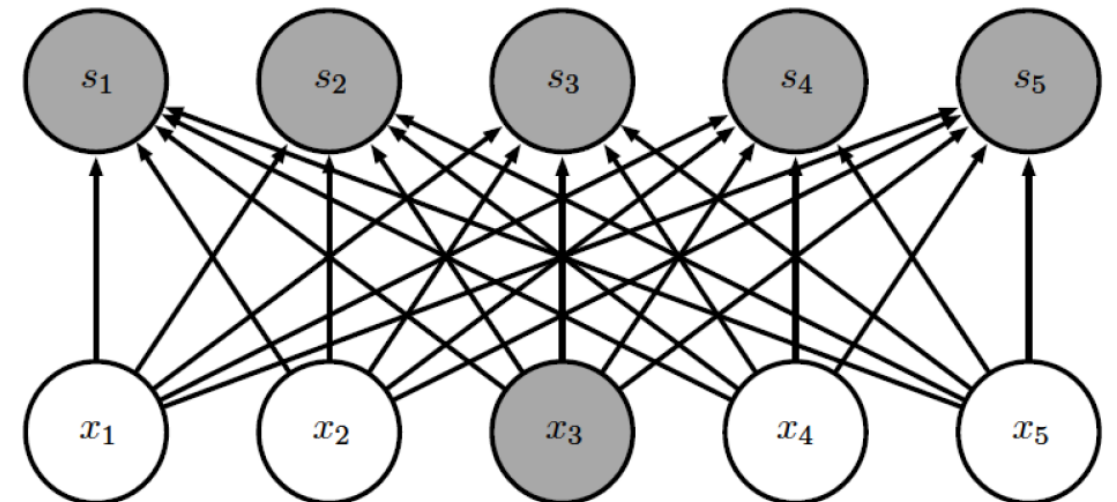
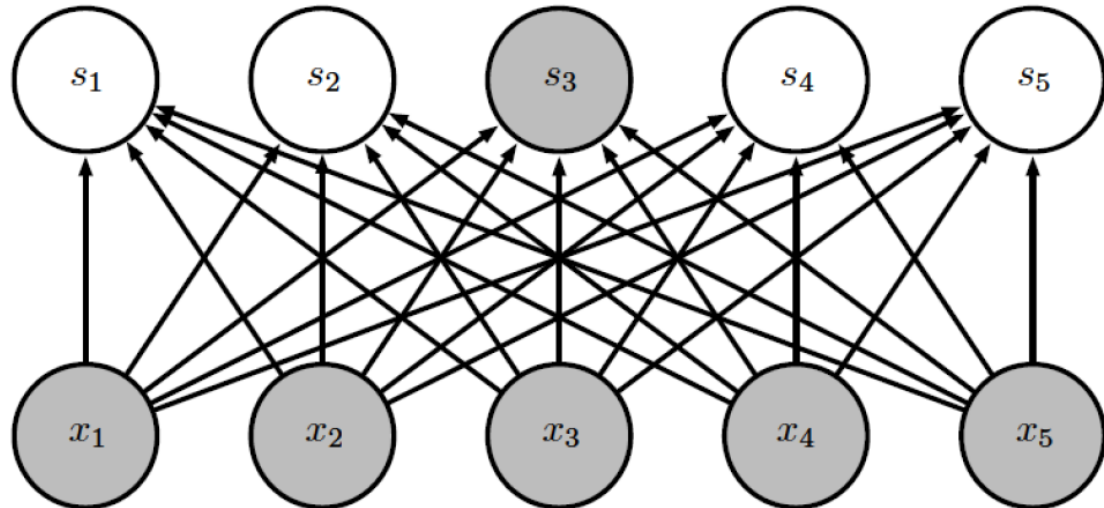
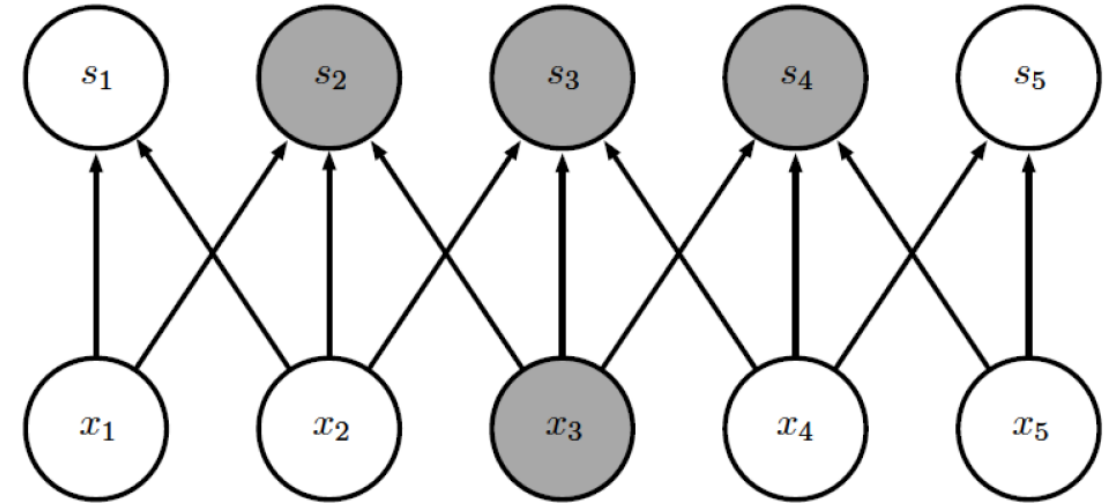
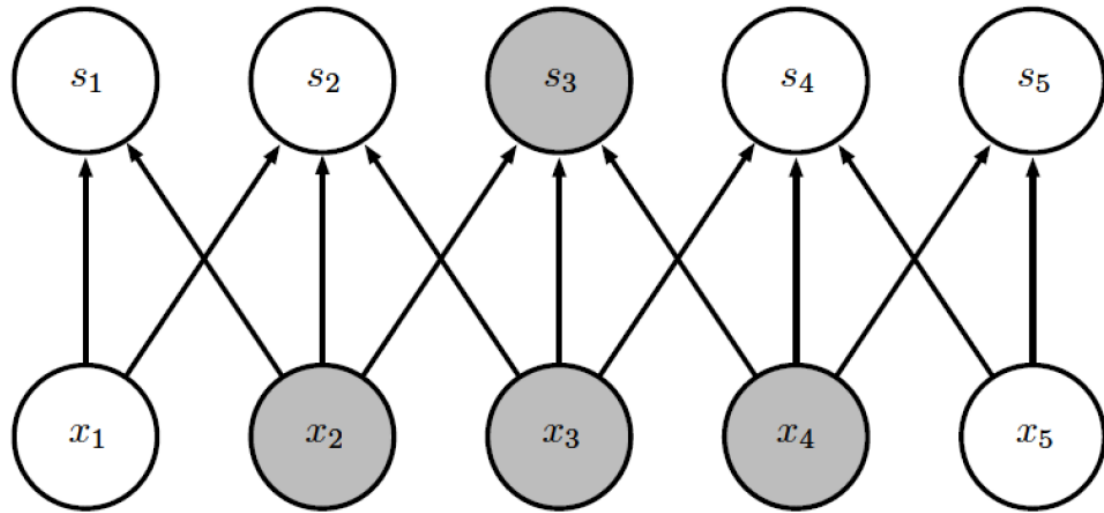
Convolution Networks help us to learn image filters

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

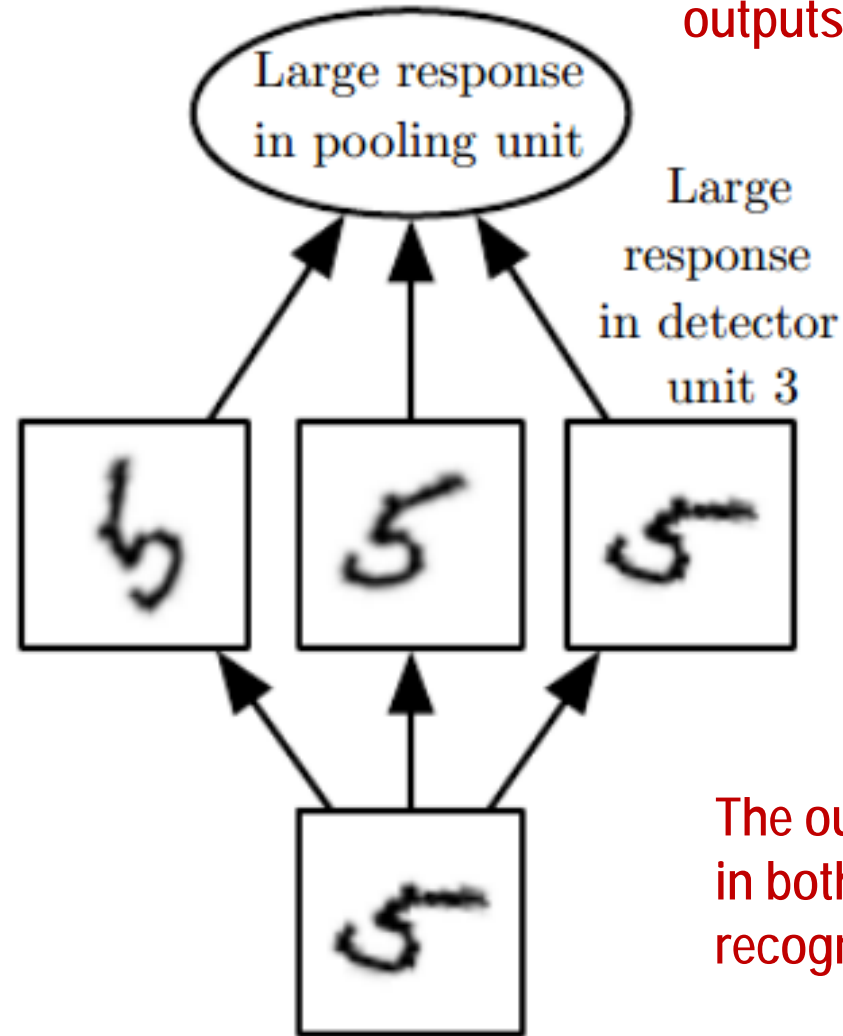
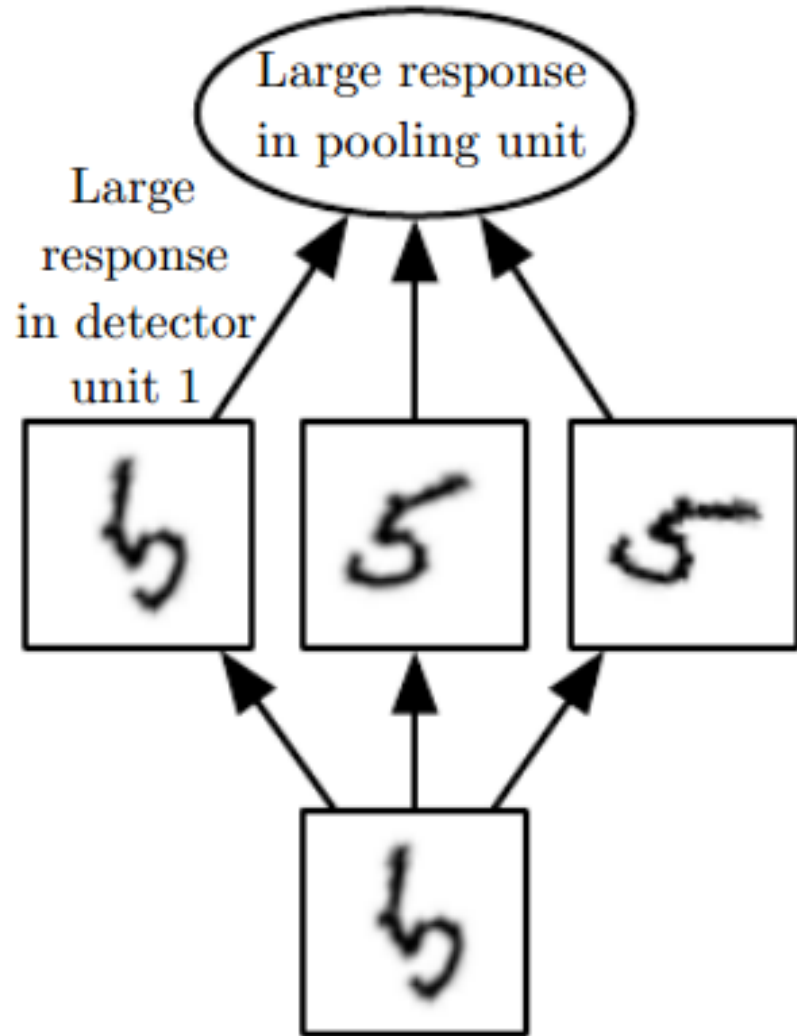
Machine learning can be used to learn these filters.

- The weights of a convolutional network are learned
- How does the network look like?

If kernel width is small, the network will be sparse



Convolution and Pooling



A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs

Set of three learned filters

The output of pooling unit is the same in both cases. Hence both the 5s are recognized.

Sequence Modeling: Recurrent and Recursive Networks

- Recurrent Neural Networks (RNNs) are a family of neural networks for processing sequential data
- Recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization
 - Most recurrent networks can also process sequences of variable length
- The key idea behind RNNs is *parameter sharing*
 - For example, in a dynamical system, the parameters of the transfer function do not change with time
 - Therefore we can use the same part of the neural network over and over again

Unfolding Computation

Consider a dynamical system:

$$\mathbf{s}^{(t)} = \mathbf{f}(\mathbf{s}^{(t-1)}; \boldsymbol{\theta})$$

where $\mathbf{s}^{(t)}$ is the state at time t and $\boldsymbol{\theta}$ is the set of parameters of \mathbf{f}

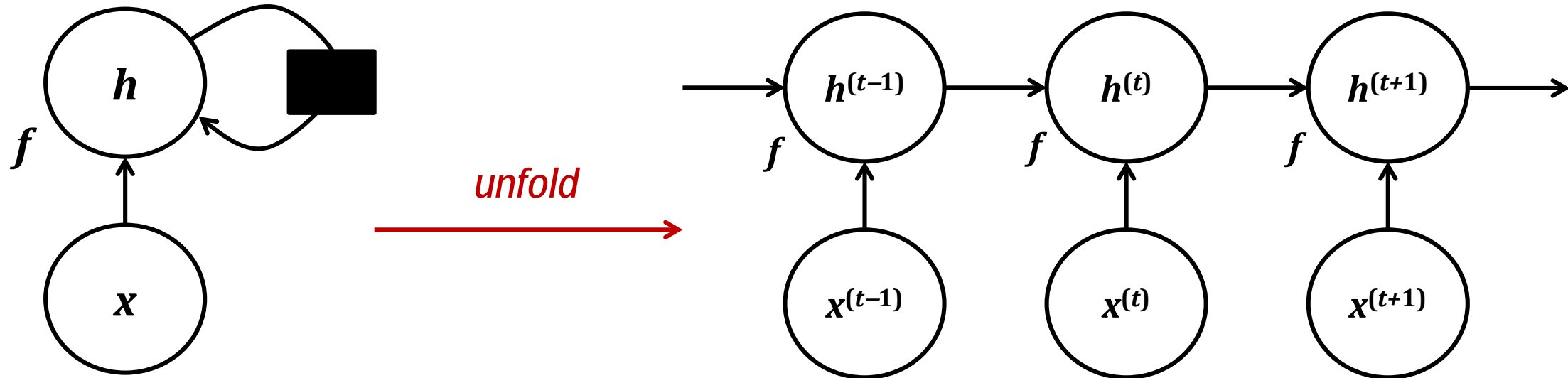
- The state after a finite number of steps can be obtained by applying the definition recursively. For example, after 3 steps:

$$\mathbf{s}^{(3)} = \mathbf{f}(\mathbf{s}^{(2)}; \boldsymbol{\theta}) = \mathbf{f}(\mathbf{f}(\mathbf{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})$$

- For a dynamical system driven by an external input signal $\mathbf{x}^{(t)}$:

$$\mathbf{s}^{(t)} = \mathbf{f}(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta})$$

Unfolding computation and Recurrent Network



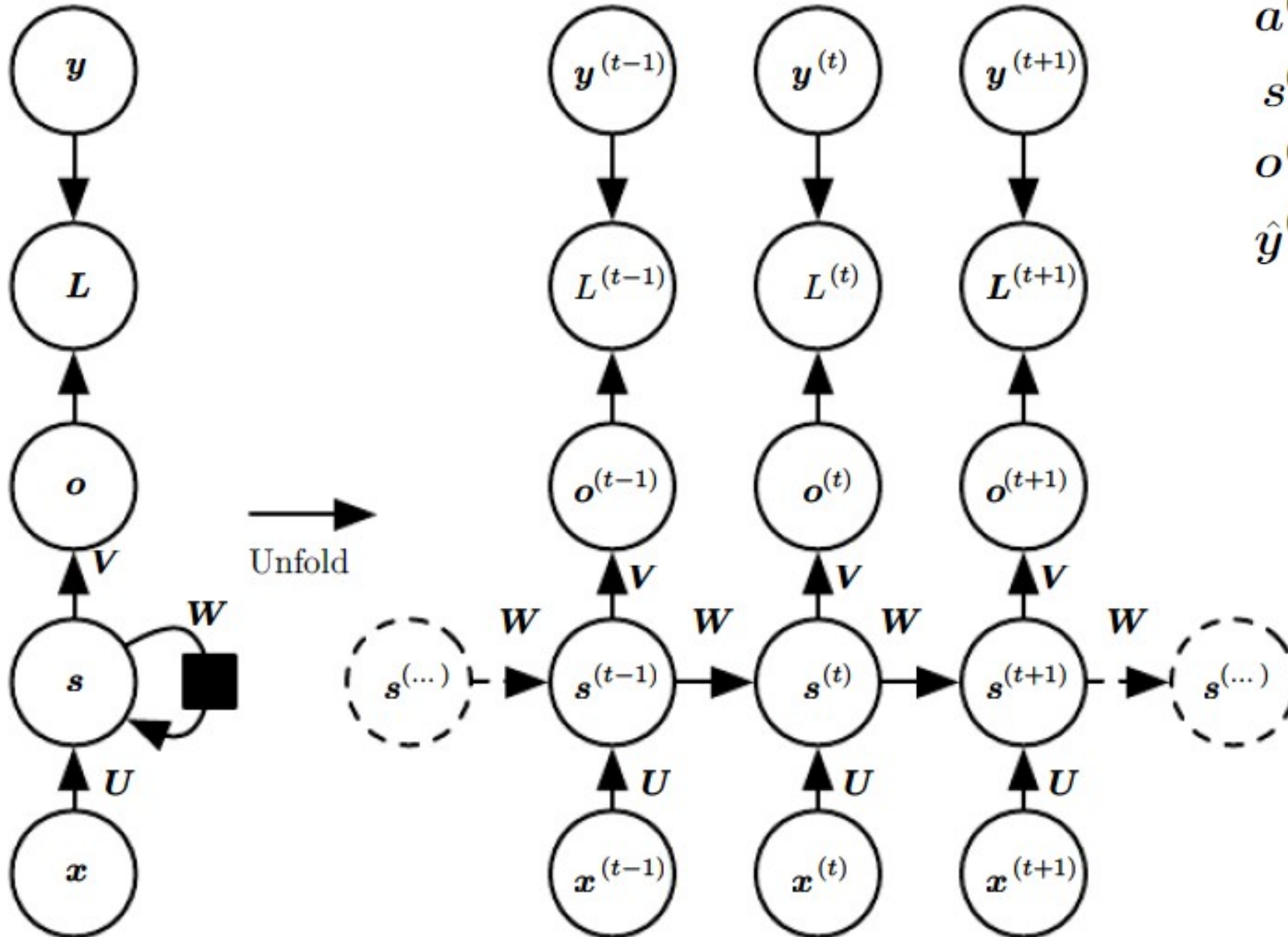
$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

- Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states
- It is possible to use the *same* transition function f with the same parameters at each step

Useful topologies of RNNs

- RNNs that produce an output at each time step and have recurrent connections between hidden units
- RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step
- RNNs with recurrent connections between hidden units, that read an entire sequence and then produce a single output

RNN with hidden-hidden feedback

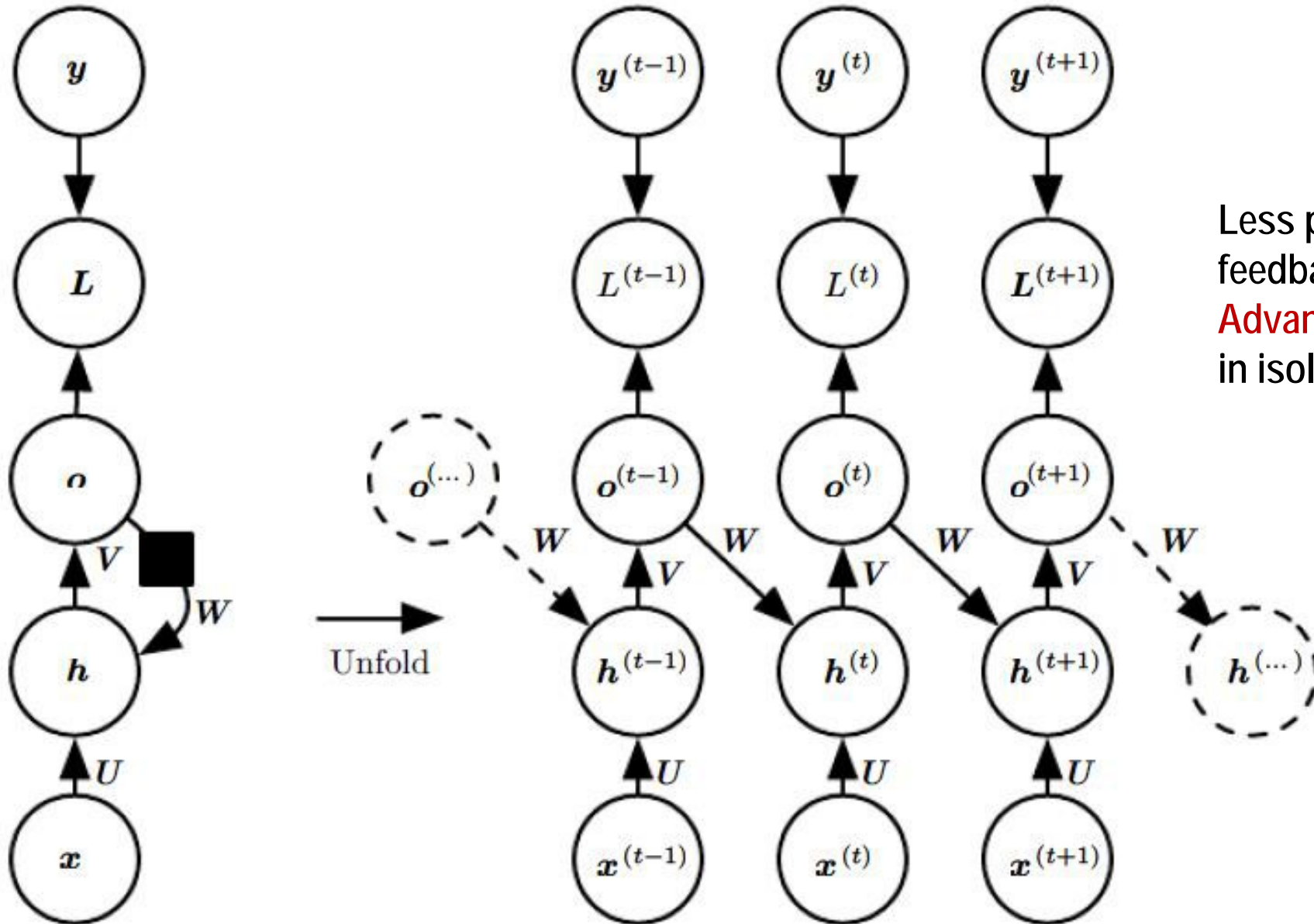


$$\begin{aligned} a^{(t)} &= b + Ws^{(t-1)} + Ux^{(t)} \\ s^{(t)} &= \tanh(a^{(t)}) \\ o^{(t)} &= c + Vs^{(t)} \\ \hat{y}^{(t)} &= \text{softmax}(o^{(t)}) \end{aligned}$$

RNN with hidden-hidden feedback is *universal*. Any function computable by a Turing machine can be computed by such a RNN of finite size (weights can have infinite precision).

Figure from *Deep Learning*,
Goodfellow, Bengio and Courville

RNN with output-hidden feedback



Less powerful than the hidden-hidden feedback model.
Advantage: Each time step can be trained in isolation (why?)

Figure from *Deep Learning*,
Goodfellow, Bengio and Courville

RNN with output only at the end

Can be used to summarize a sequence and produce a fixed-size representation to be used as an input for further processing

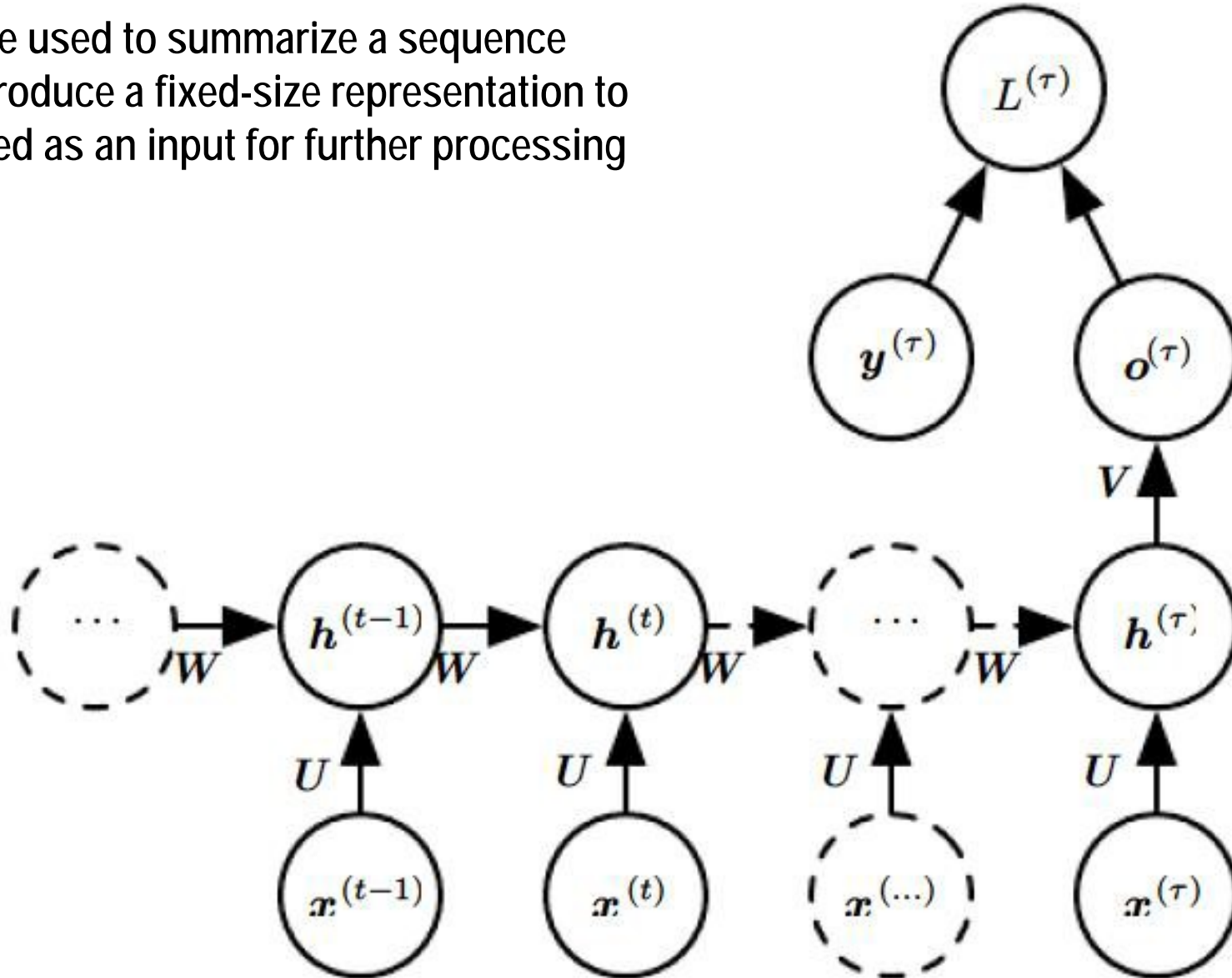


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Boltzmann Machines

A Boltzmann machine is a network of units with an *energy* defined for the overall network. Its units produce binary results. The global energy, E , is:

$$E = -\left(\sum_{i < j} w_{ij} s_i s_j + \sum_i \theta_i s_i\right)$$

where:

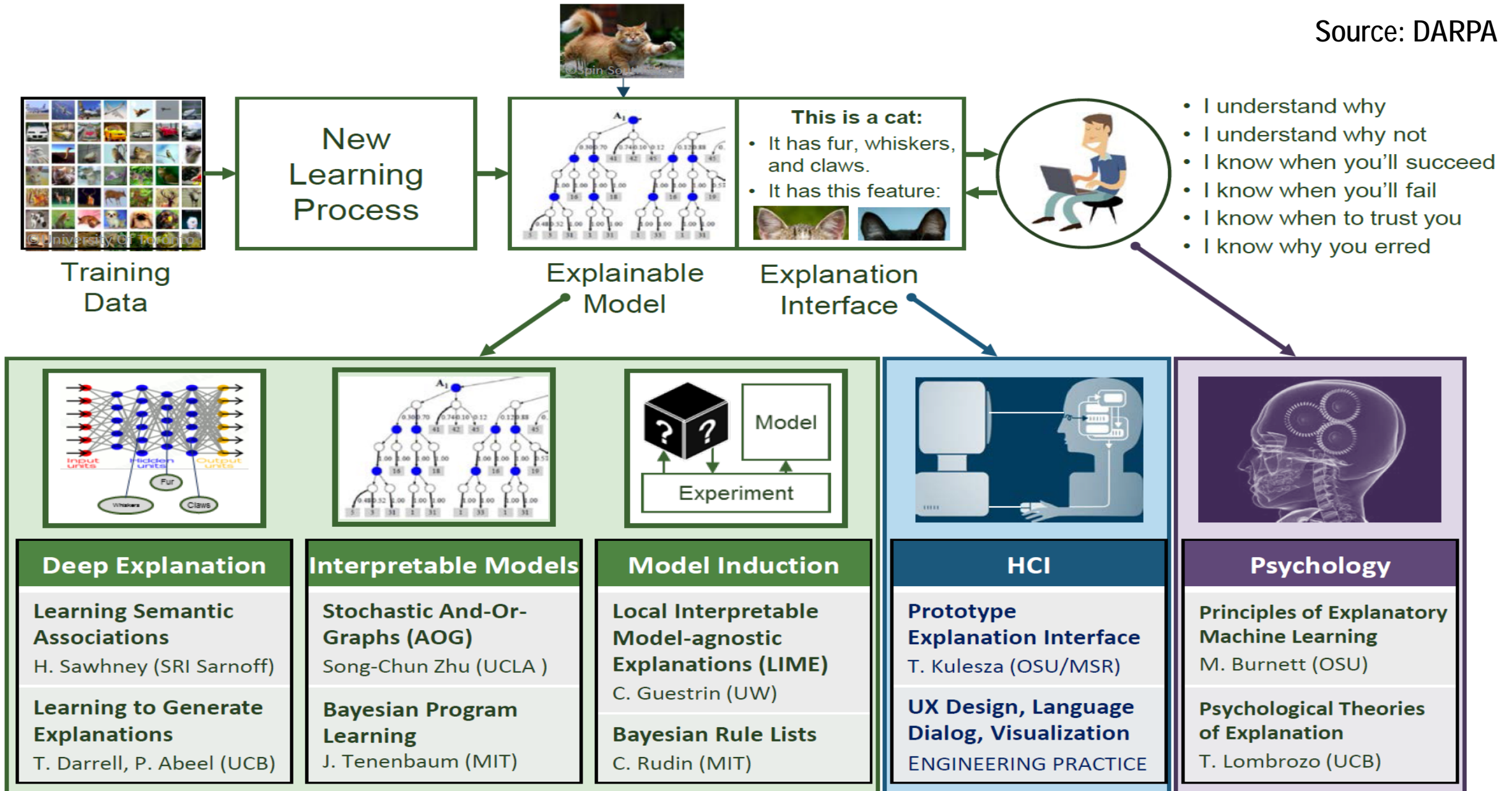
- w_{ij} is the connection strength between unit j and unit i .
- s_i is the state, $s_i \in \{0, 1\}$, of unit i
- θ_i is the bias of unit i in the global energy function. ($-\theta_i$ is the activation threshold for the unit)

$$\Delta E_i = \sum_{j > i} w_{ij} s_j + \sum_{j < i} w_{ji} s_j + \theta_i$$

- From this we obtain (the scalar T is called the *temperature*):

$$p_{i=on} = \frac{1}{1 + \exp\left(-\frac{\Delta E_i}{T}\right)}$$

Source: DARPA



The ML problem in regression

What is the function $f(\cdot)$?

Solution: *This is where the different ML methods come in*

- Linear model: $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$
- Linear basis functions: $f(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$
 - Where $\boldsymbol{\phi}(\mathbf{x}) = [\boldsymbol{\phi}_0(\mathbf{x}) \boldsymbol{\phi}_1(\mathbf{x}) \dots \boldsymbol{\phi}_L(\mathbf{x})]^T$ and $\boldsymbol{\phi}_l(\mathbf{x})$ is the basis function.
 - Choices for the basis function:
 - Powers of x : $\boldsymbol{\phi}_l(\mathbf{x}) = x^l$
 - Gaussian / Sigmoidal / Fourier / ...
- Neural networks
- ...

Classification

Given training data set with:

- Input values: $\mathbf{x}_n = [x_1 \ x_2 \ \dots \ x_M]^T$ for $n = 1 \dots N$.
- Output class labels, for example:
 - 0/1 or $-1/+1$ for binary classification problems
 - 1 ... K for multi-class classification problems
 - 1-of-K coding scheme:

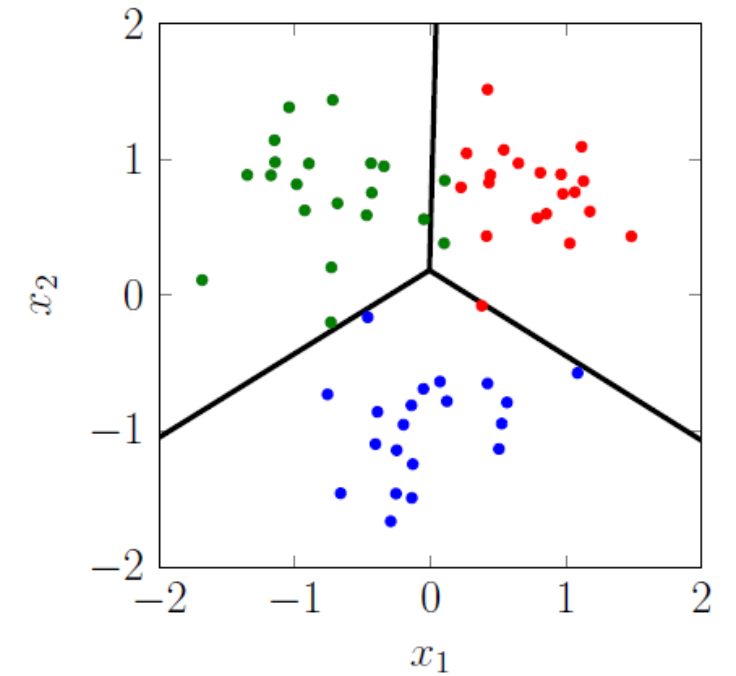
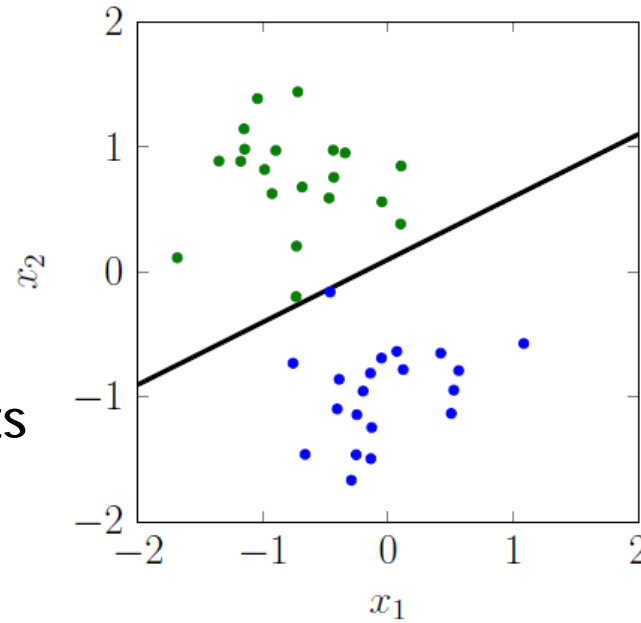
$$\mathbf{y} = [0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0]^T$$

where, if \mathbf{x}_n belongs to class k , then the k^{th} bit is 1 and all others are 0.

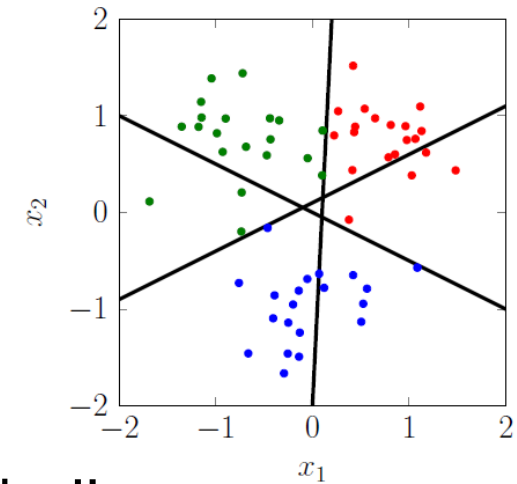
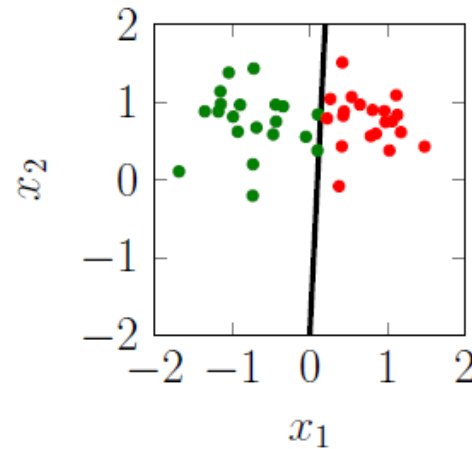
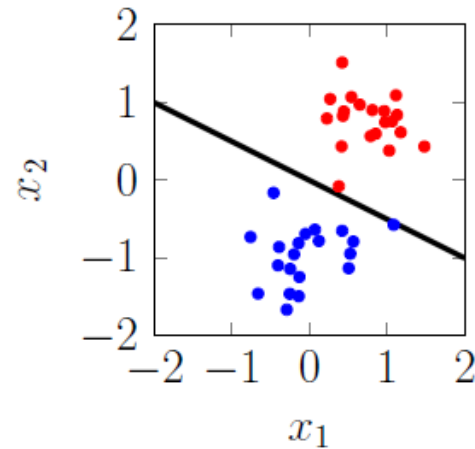
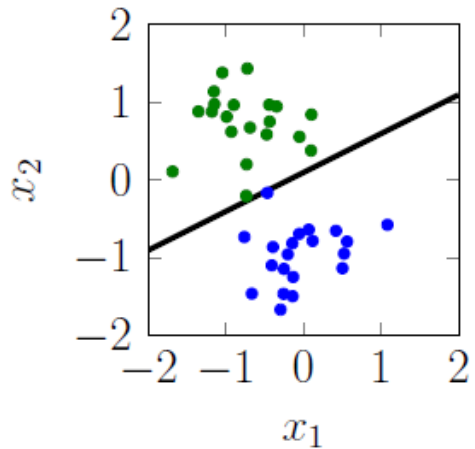
Objective: Predict the output class for new, unknown inputs $\hat{\mathbf{x}}_m$.

Classification strategies

Linear discriminants
(2-class classifiers)



K-class discriminant



Combining 2-class classifiers to obtain multi-class classifiers is a bad idea !!