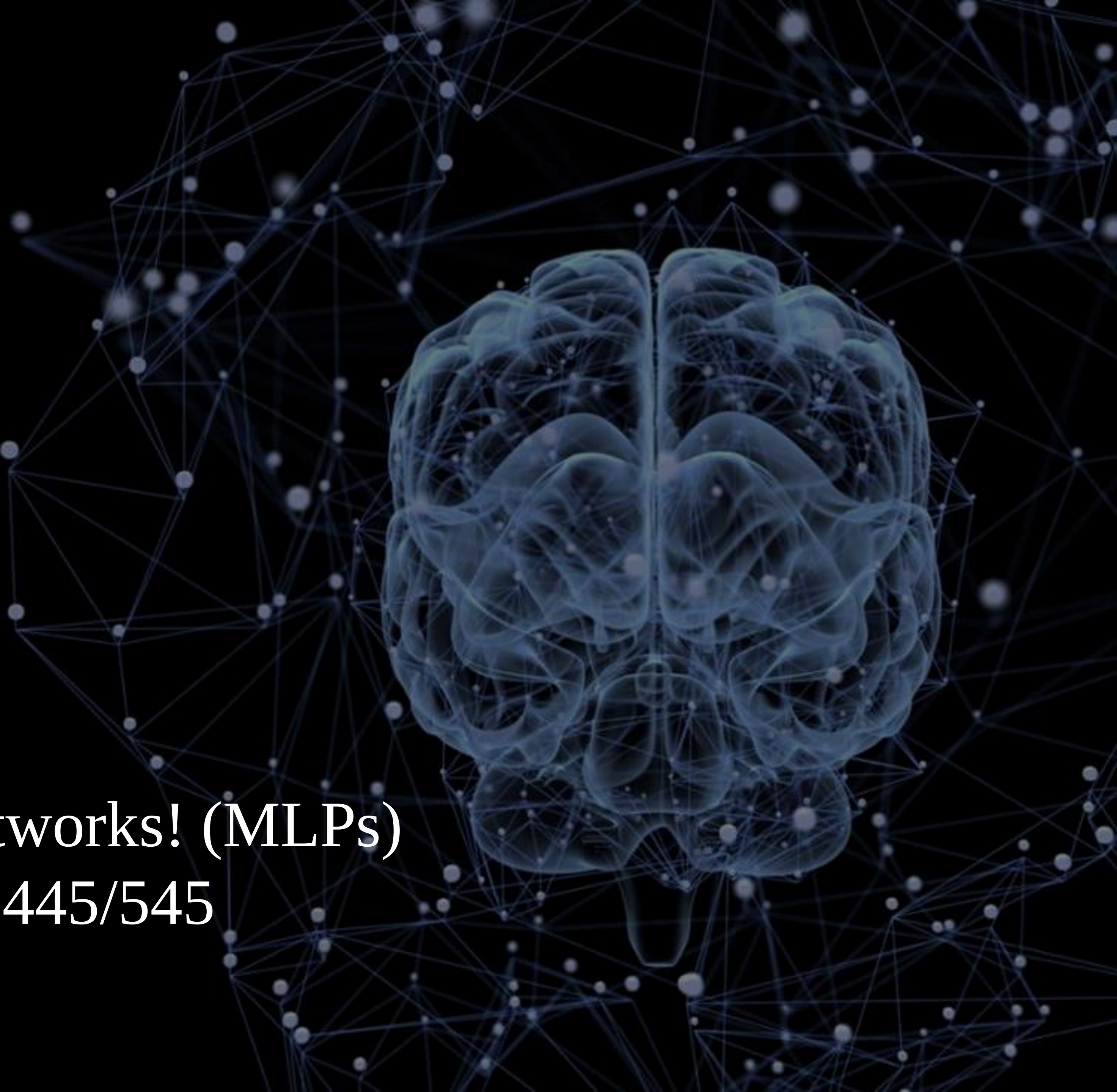
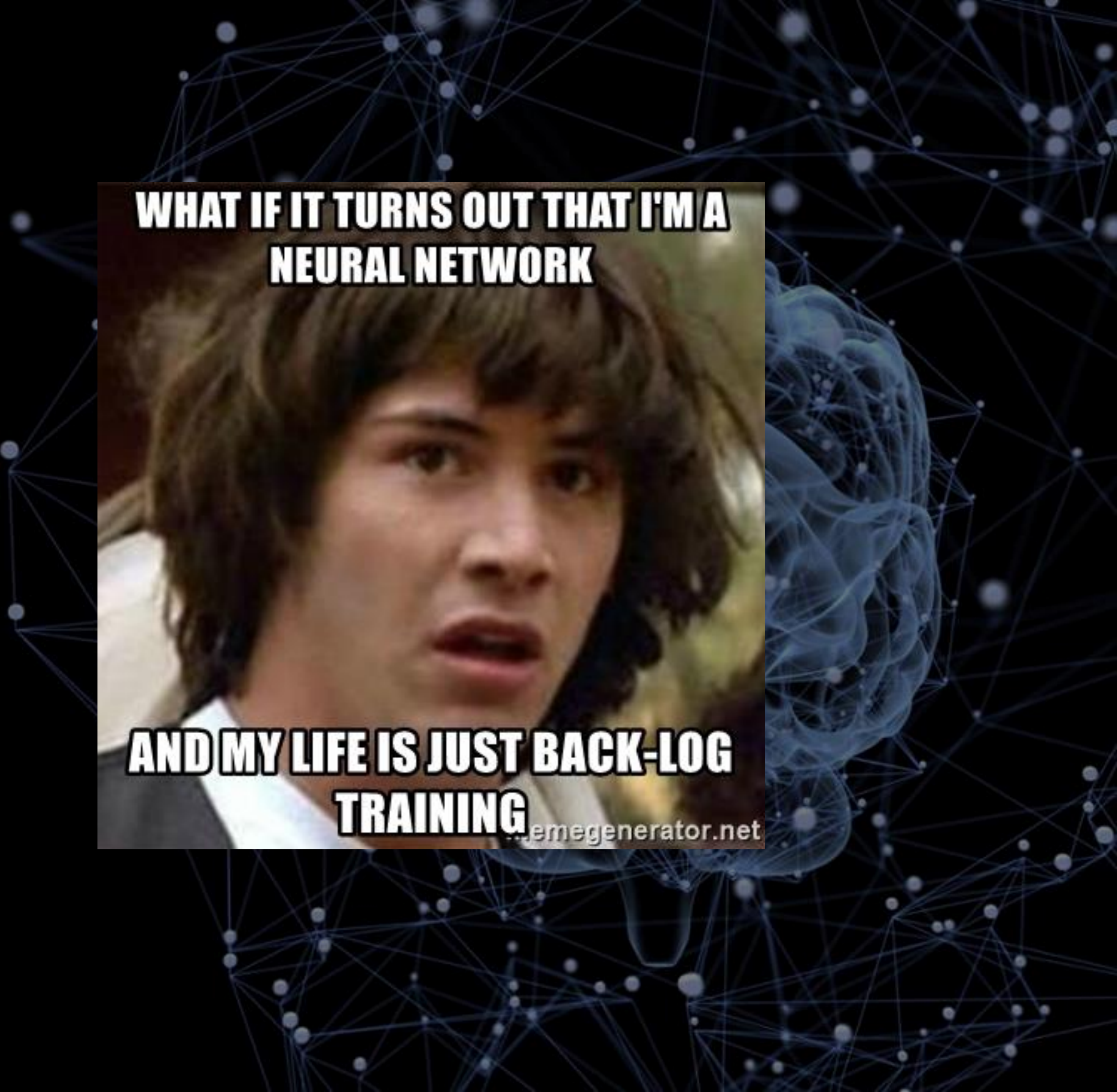
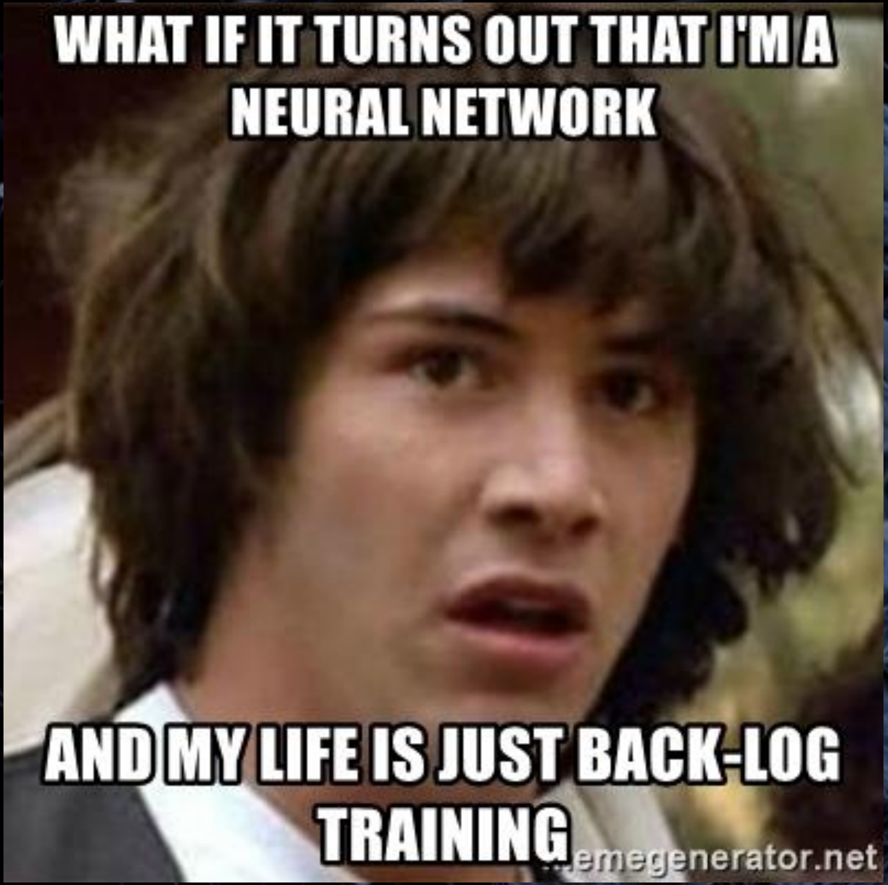


Neural Networks! (MLPs)

CS 445/545




WHAT IF IT TURNS OUT THAT I'M A NEURAL NETWORK
AND MY LIFE IS JUST BACK-LOG TRAINING
...emegenerator.net

What can I do with a NN?

A short list of applications:

(*) Binary classification, 1-in-K classification, regression

(*) General pattern recognition / statistical learning

(*) Character recognition, facial recognition

(*) Computer Vision: image classification, localization, scene recognition, captioning

(*) Signal Processing: noise suppression, signal analysis

(*) Data compression

(*) NLP: machine translation, sentiment analysis

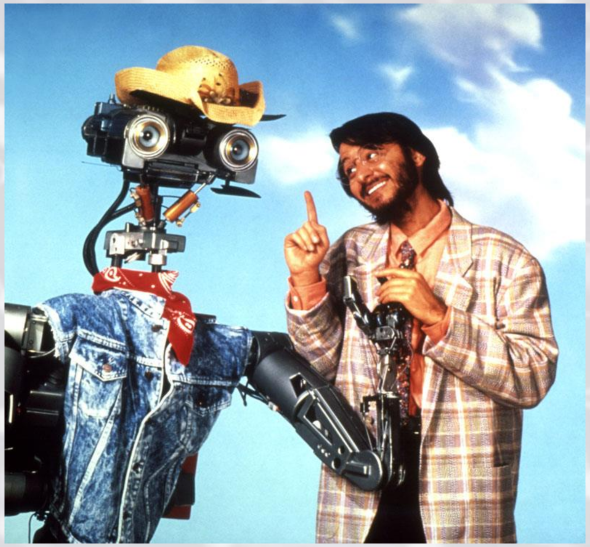
(*) Finance: statistical arbitrage, risk analysis

(*) AI: Q-Learning (reinforcement learning)

(*) Medicine: diagnosis, imaging, genomics

(*) Law: information retrieval

(*) Computational creativity applications



A bit of history

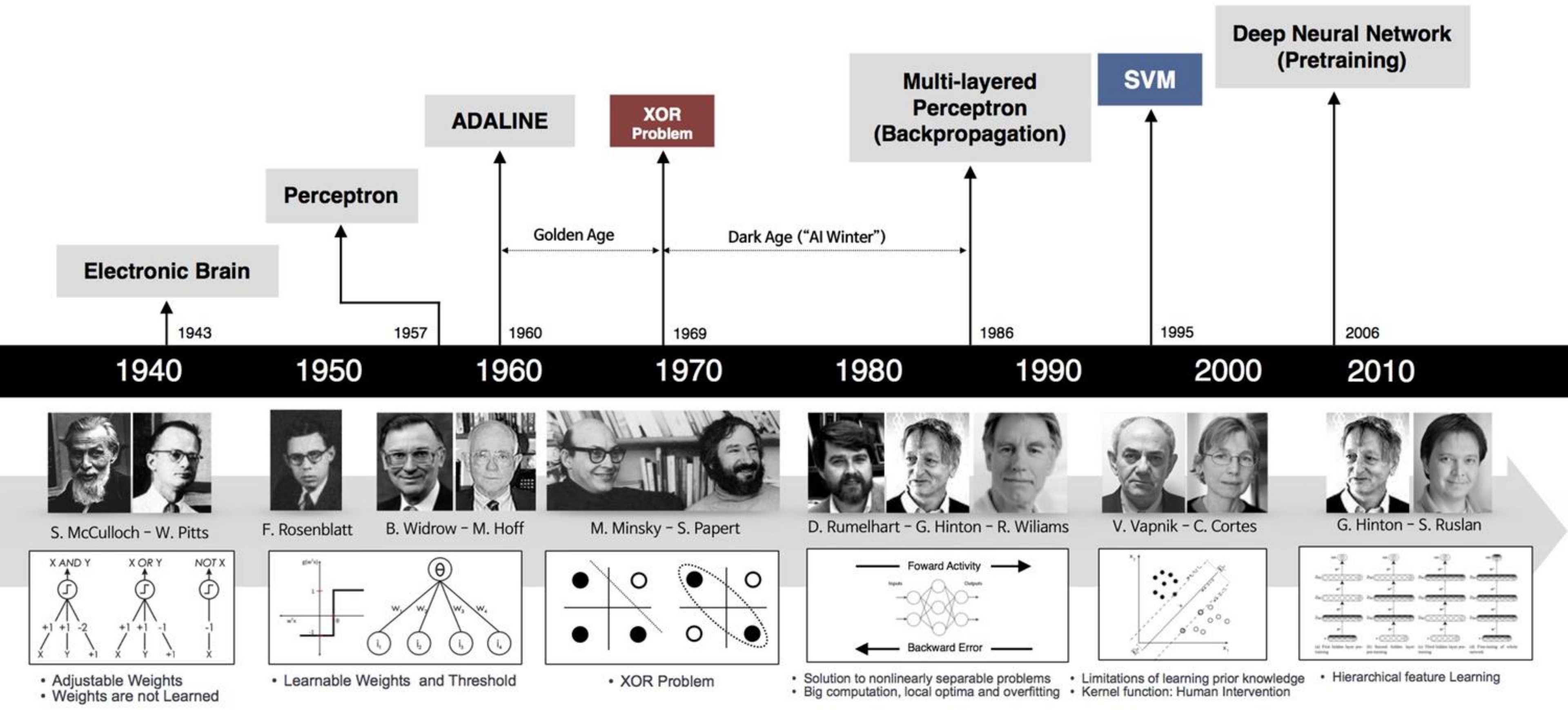
- 1960s: **Rosenblatt** proved that the perceptron learning rule converges to correct weights in a finite number of steps, provided the training examples are linearly separable.

- 1969: **Minsky and Papert** proved that perceptrons cannot represent non-linearly separable target functions.

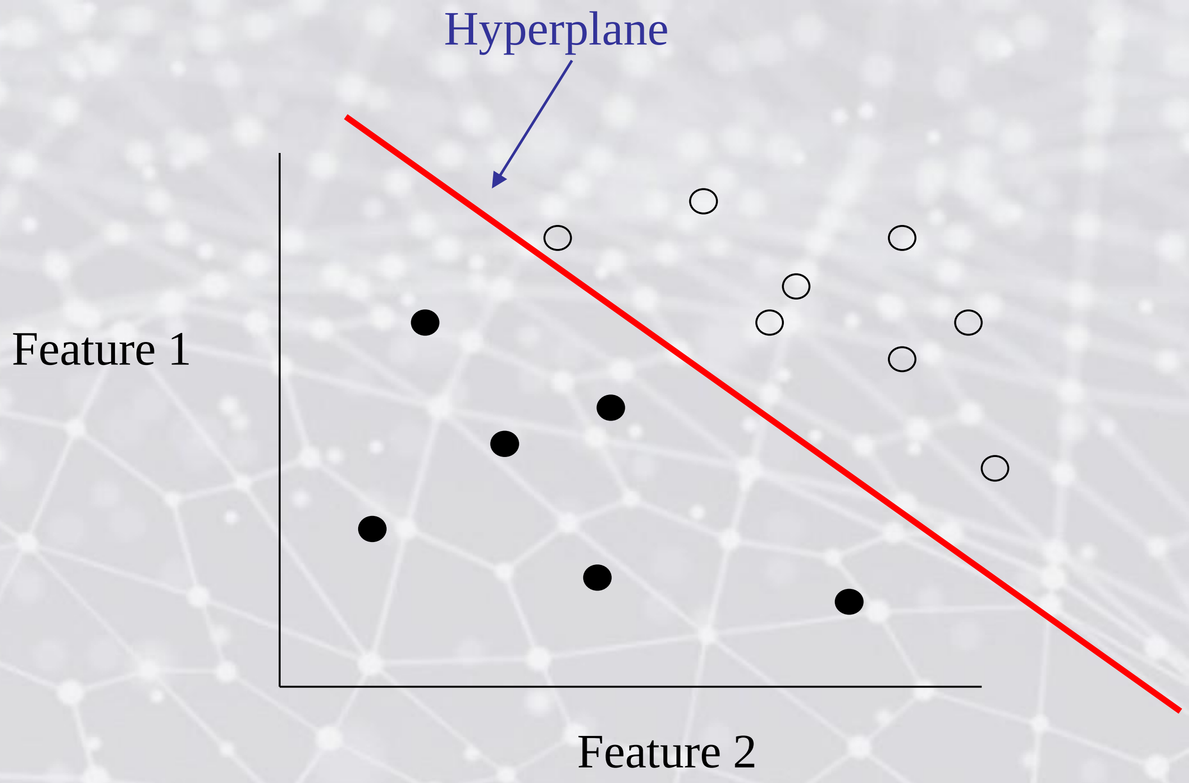
- However, they showed that adding a fully connected hidden layer makes the network more powerful.
  - I.e., Multi-layer neural networks can represent non-linear decision surfaces.
- Later it was shown that by using continuous activation functions (rather than thresholds), a fully connected network with a single hidden layer can in principle represent any function.

- 1986: "rediscovery" of backprop algorithm: **Hinton** et al.

A bit of 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

Linear separability



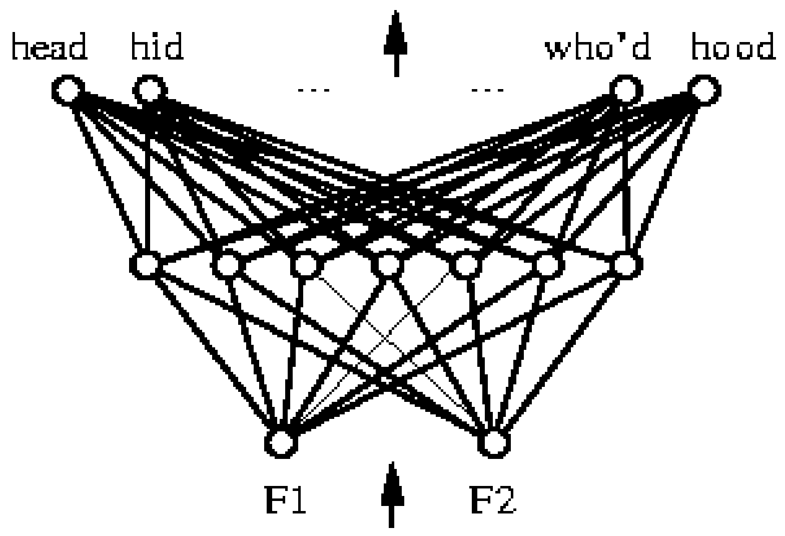
In 2D:

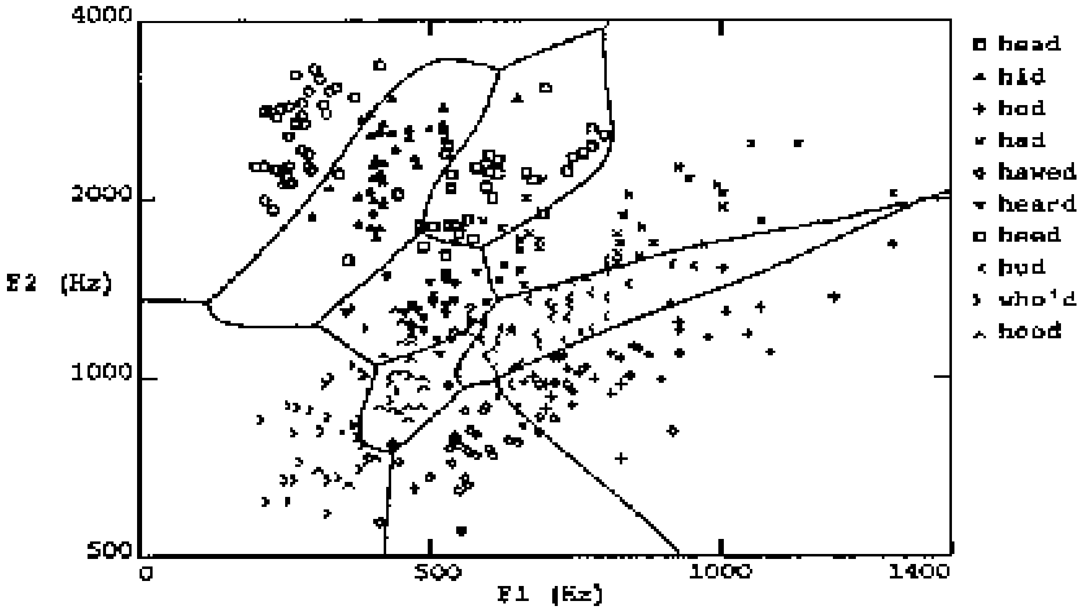
$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

A perceptron can separate data that is linearly separable.

Multi-layer neural network example





Decision regions of a multilayer feedforward network. (From T. M. Mitchell, *Machine Learning)*

The network was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had", "hid")

The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound.

The 10 network outputs correspond to the 10 possible vowel sounds.

- **Good news:** Adding hidden layer allows more target functions to be represented.

- **Bad news:** No algorithm for learning in multi-layered networks, and no convergence theorem!

- Quote from Minsky and Papert's book, *Perceptrons* (1969):

  "[The perceptron] has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile."

- Two major problems they saw were:

  1. How can the learning algorithm apportion credit (or blame) to individual weights for incorrect classifications depending on a (sometimes) large number of weights?

  2. How can such a network learn useful higher-order features?

- **Good news:** Successful credit-apportionment learning algorithms developed soon afterwards (e.g., back-propagation).

- **Bad news:** However, in multi-layer networks, there is no guarantee of convergence to minimal error weight vector.

But in practice, multi-layer networks often work very well.

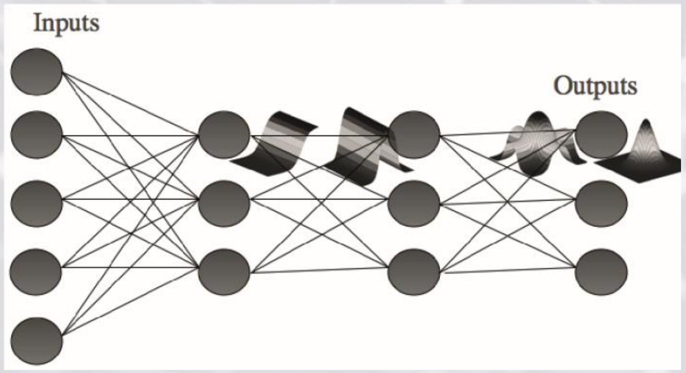
Summary

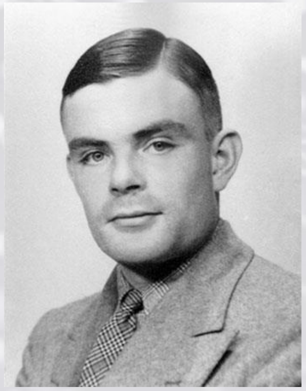
- Perceptrons can only be 100% accurate only on linearly separable problems.

- Multi-layer networks (often called *multi-layer perceptrons*, or *MLPs*) can represent any target function.

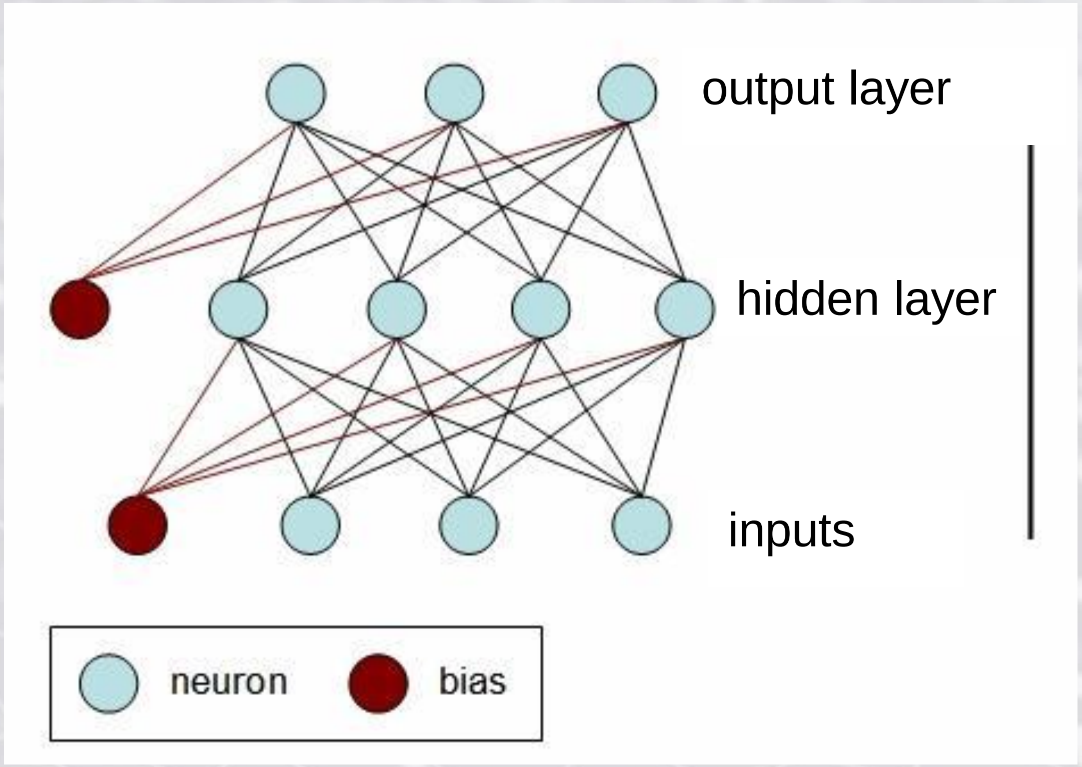
- However, in multi-layer networks, there is no guarantee of convergence to minimal error weight vector.

- One can show, mathematically, that one hidden layer is sufficient to approximate any function to arbitrary accuracy with a NN. This is known as the **Universal Approximation Theorem (1989)** (we say: "NNs are universal function approximators"); RNNs are *Turing Complete*.





A “two”-layer neural network



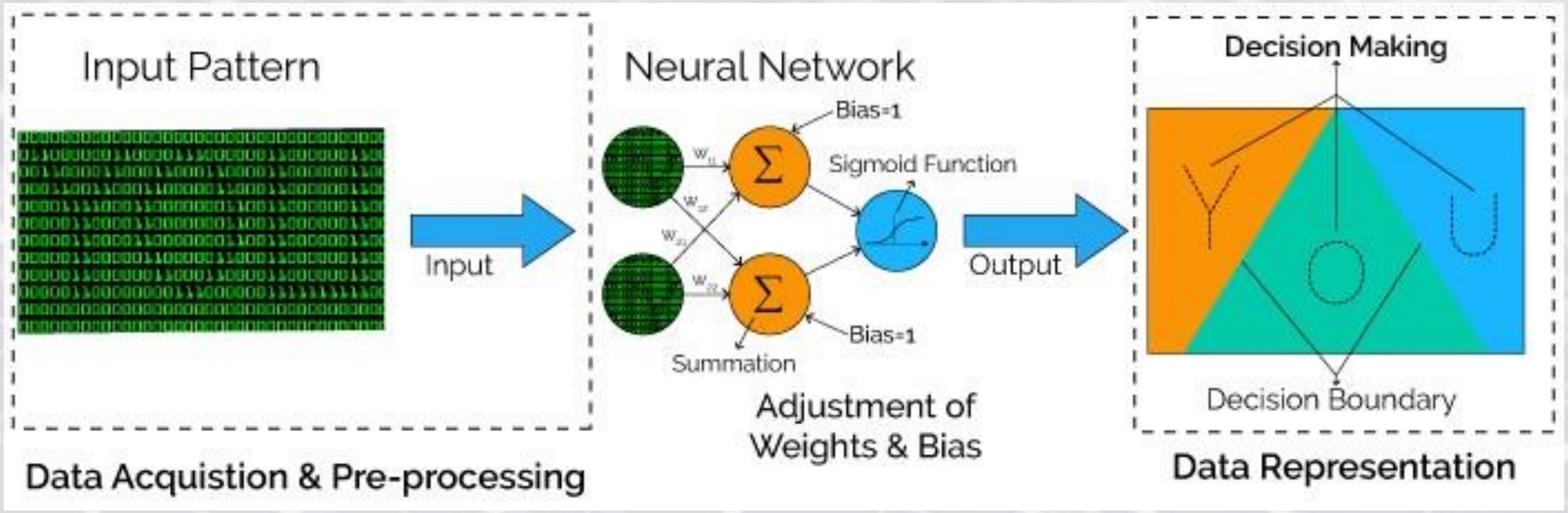
(activation represents classification)

(internal representation)

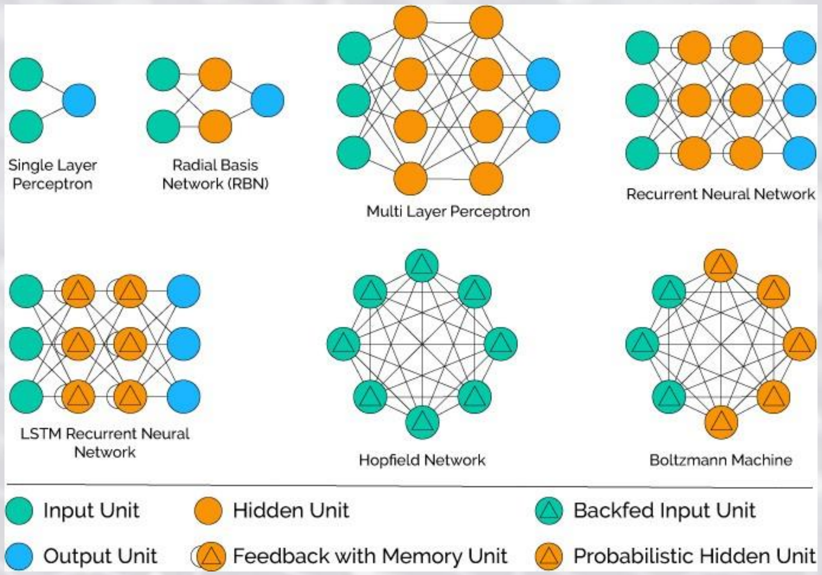
(activations represent feature vector for one training example)

•**Input layer** — It contains those units (artificial neurons) which receive input from the outside world on which network will learn, recognize about or otherwise process.
•**Output layer** — It contains units that respond to the information about how it’s learned any task.
•**Hidden layer** — These units are in between input and output layers. The job of hidden layer is to transform the input into something that output unit can use in some way.
Most neural networks are fully connected that means to say each hidden neuron is fully connected to the every neuron in its previous layer(input) and to the next layer (output) layer.

Classification Pipeline



Different Types of Neural Networks



Perceptron — Neural Network having two input units and one output units with no hidden layers. These are also known as 'single layer perceptrons.

Radial Basis Function Network — These networks are similar to the feed forward neural network except radial basis function is used as activation function of these neurons.

Multilayer Perceptron — These networks use more than one hidden layer of neurons, unlike single layer perceptron. These are also known as deep feedforward neural networks.

Recurrent Neural Network — Type of neural network in which hidden layer neurons has self-connections. Recurrent neural networks possess memory. At any instance, hidden layer neuron receives activation from the lower layer as well as it previous activation value.

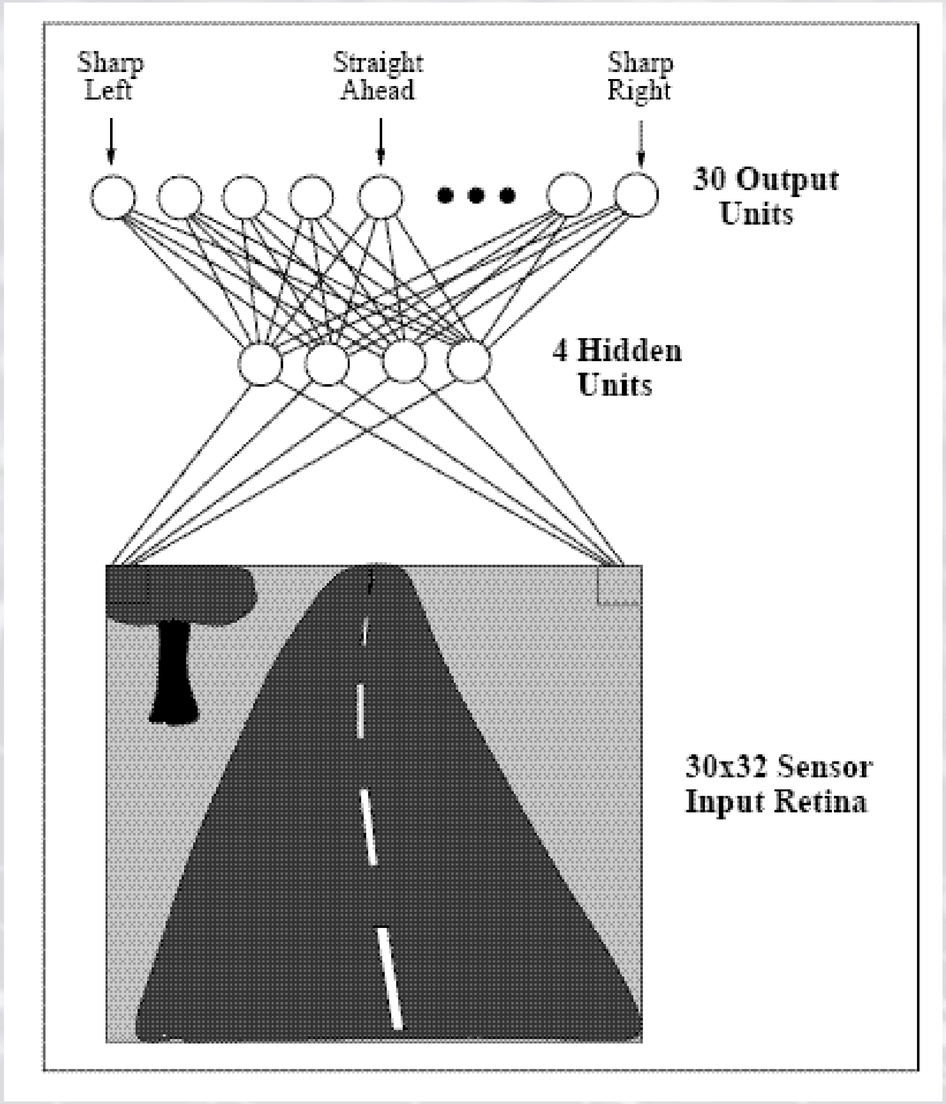
Long /Short Term Memory Network (LSTM) — Type of neural network in which memory cell is incorporated inside hidden layer neurons is called LSTM network.

Convolutional Neural Network — Get a complete overview of Convolutional Neural Networks through our blog Log Analytics with Machine Learning and Deep Learning.

Example: ALVINN
(Pomerleau, 1993)

- ALVINN learns to drive an autonomous vehicle at normal speeds on public highways.

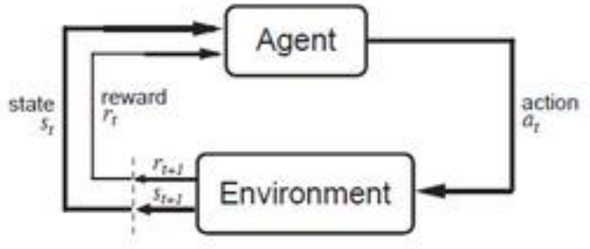
- Input: 30 x 32 grid of pixel intensities from camera



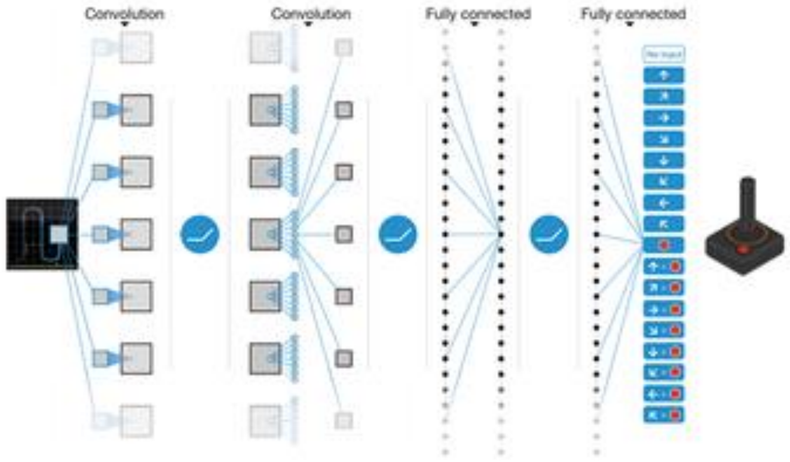
(Note: bias units and weights not shown)

Each output unit correspond to a particular steering direction. The most highly activated one gives the direction to steer.

Example: DeepMind (Deep Q learning for Atari, 2014)



AlphaGo



Activation functions

g( ) - transfer functions, nonlinearities, units

- They act as a **threshold**

Desirable properties

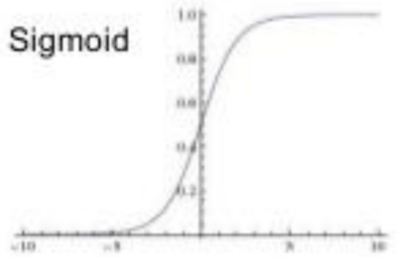
- Mostly smooth, continuous, differentiable
- Fairly linear

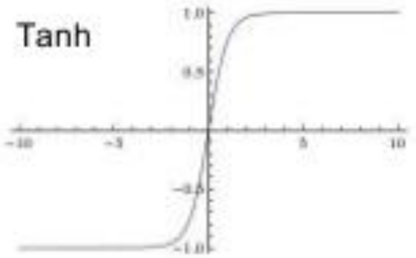
Common nonlinearities

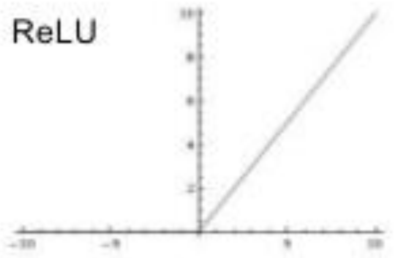
- Sigmoid
- Tanh
- ReLU = max(0, x)

Why do we need them?

If we only use linear layers we are only able to learn linear transformations of our input.





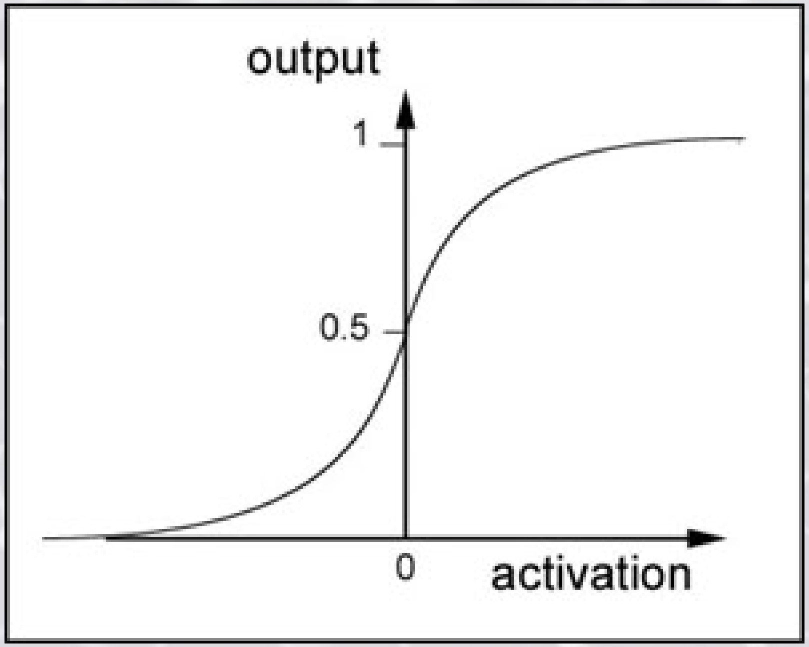


- Advantages of sigmoid function: nonlinear, differentiable, has real-valued outputs, and approximates a threshold function.

Sigmoid activation function:

$$o = \sigma(\mathbf{w} \cdot \mathbf{x}), \quad \text{where} \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

$\mathbf{w} \cdot \mathbf{x}$	$\sigma(\mathbf{w} \cdot \mathbf{x})$
-2	.12
-1.5	.18
-1	.27
-.5	.38
0	.50
.5	.62
1	.73
1.5	.82
2	.88



- The derivative of the sigmoid activation function is easily expressed in terms of the function itself:

$$\frac{ds(z)}{dz} = s(z) \times (1 - s(z))$$

This is useful in deriving the back-propagation algorithm.

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

$$\frac{ds}{dz} = -1(1+e^{-z})^{-2}\frac{d}{dz}(1+e^{-z})$$

$$= -\frac{1}{(1+e^{-z})^2}\left(-e^{-z}\right)$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

$$s(z)\cdot(1-s(z))$$

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

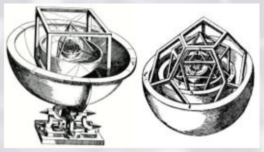
$$= \left(\frac{1}{1+e^{-z}}\right)-\left(\frac{1}{1+e^{-z}}\right)^2$$

$$= \left(\frac{1}{1+e^{-z}}\right)-\left(\frac{1}{(1+e^{-z})^2}\right)$$

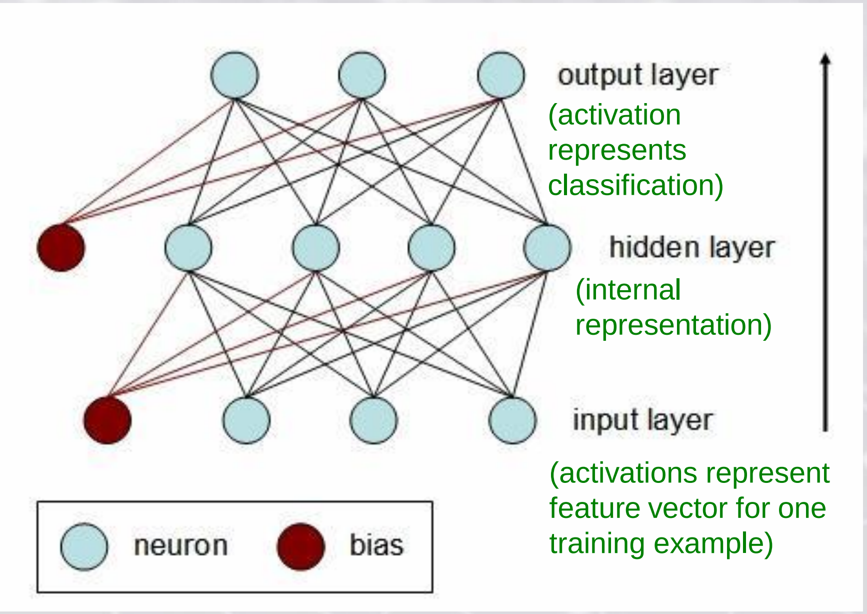
$$= \left(\frac{1+e^{-z}}{(1+e^{-z})^2}\right)-\left(\frac{1}{(1+e^{-z})^2}\right)$$

$$= \frac{e^{-z}}{(1+e^{-z})^2}$$

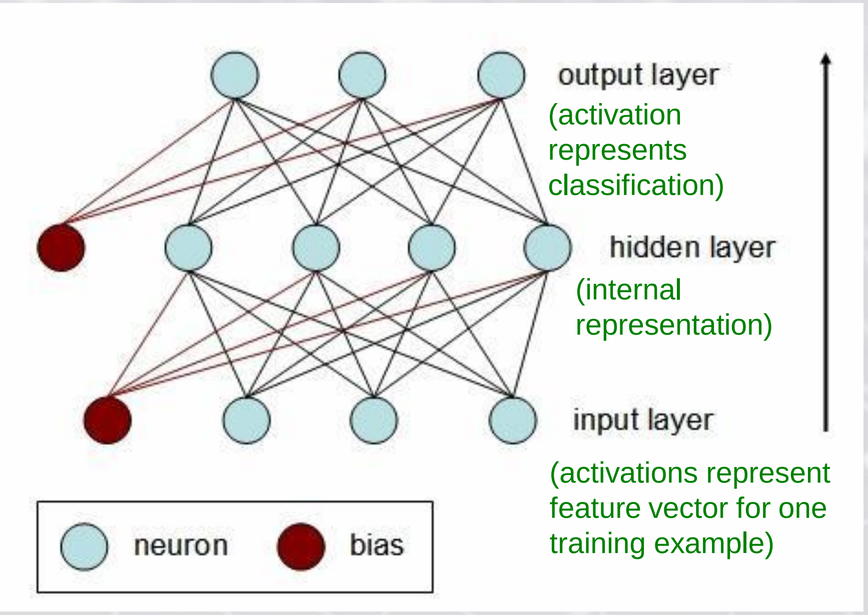
And thus the math Gods said… $\dfrac{ds(z)}{dz} = s(z)\times(1-s(z))$



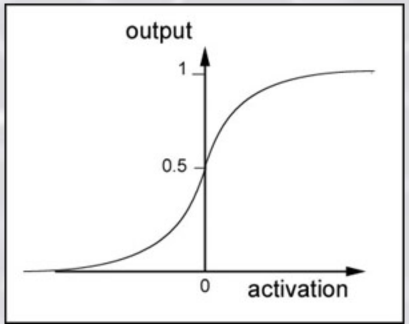
Neural network notation


output layer
(activation represents classification)
hidden layer
(internal representation)
input layer
(activations represent feature vector for one training example)
neuron
bias

Neural network notation


output layer
(activation represents classification)
hidden layer
(internal representation)
input layer
(activations represent feature vector for one training example)
neuron
bias

Sigmoid function:



Neural network notation

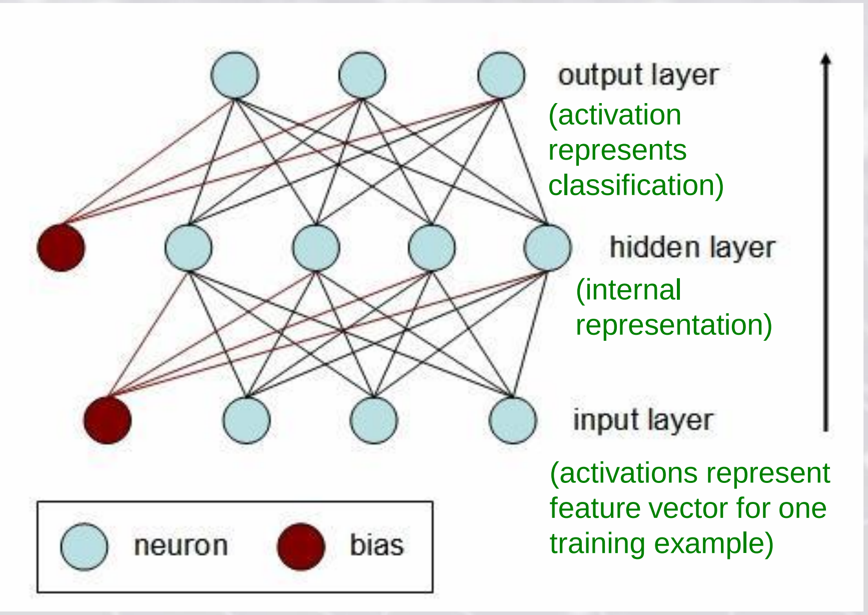
x_i : activation of **input** node *i*.

h_j : activation of **hidden** node *j*.

o_k : activation of **output** node *k*.

w_{ji} : weight from node *i* to node *j*.

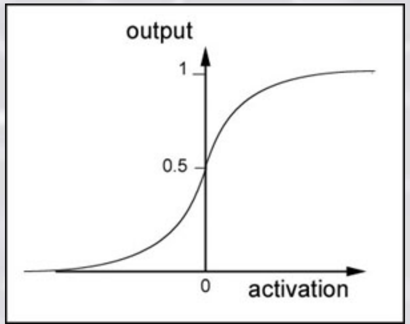
σ : sigmoid function.


output layer
(activation represents classification)
hidden layer
(internal representation)
input layer
(activations represent feature vector for one training example)
neuron
bias

For each node *j* in hidden layer,

$$h_j = s\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0} \right)$$

Sigmoid function:



For each node *k* in output layer,

$$o_k = s\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0} \right)$$

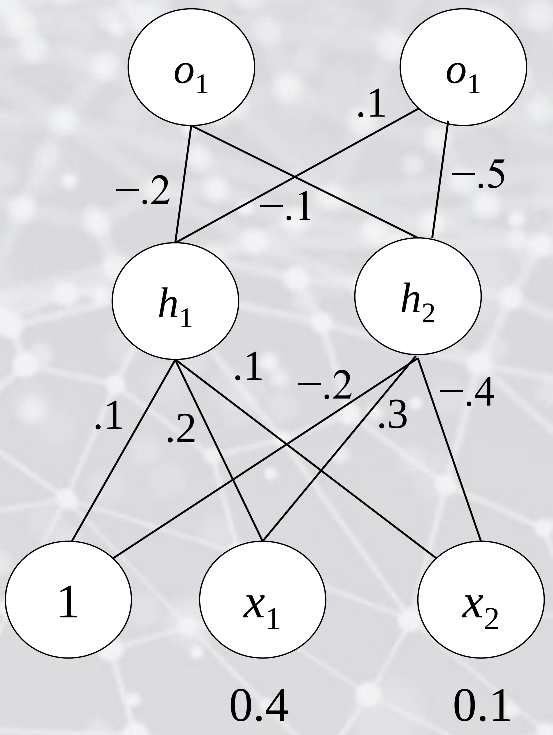
Classification with a two-layer neural network
(“<u>Forward propagation</u>”)

Assume two-layer networks (i.e., one hidden layer):

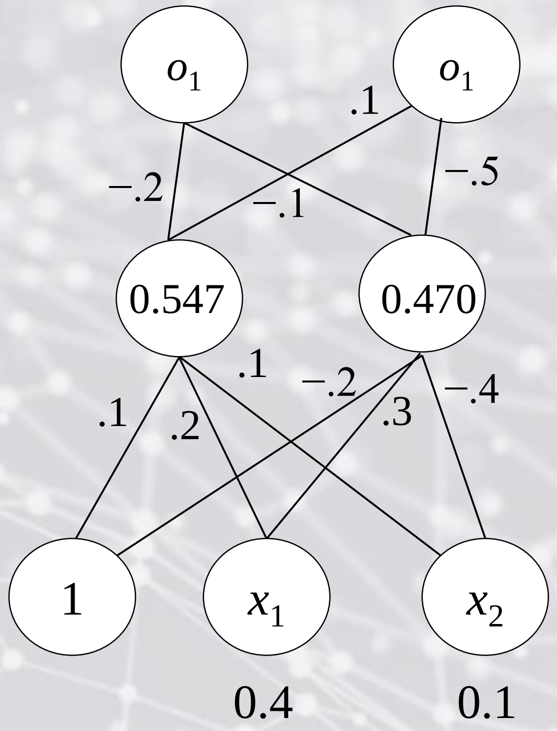
1. Present input to the input layer.
2. <u>Forward propagate</u> the activations times the weights to each node in the <u>hidden layer</u>.
3. Apply activation function (sigmoid) to sum of weights times inputs to each hidden unit.
4. Forward propagate the activations times weights from the hidden layer to the <u>output layer</u>.
5. Apply activation function (sigmoid) to sum of weights times inputs to each output unit.
6. Interpret the output layer as a classification.

Simple Example

Input:



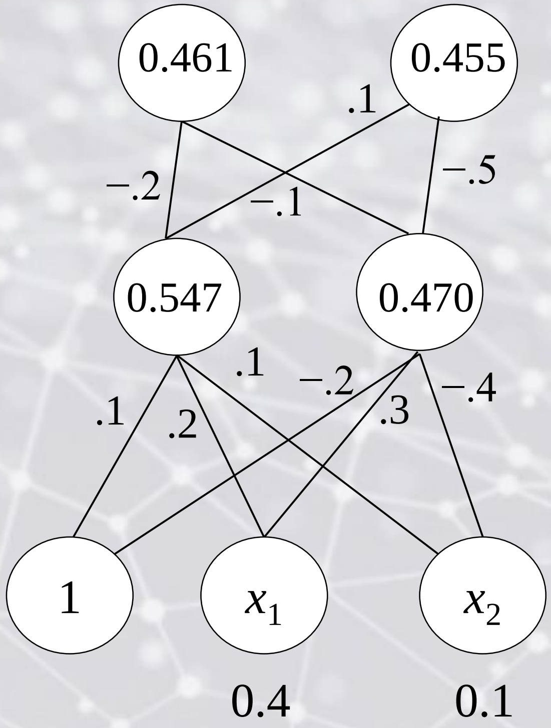
Hidden Layer:



$$h_1 = \sigma\big((.1)(1) + (.2)(.4) + (.1)(.1)\big) = \sigma(.19) = .547$$

$$h_2 = \sigma\big((-.2)(1) + (.3)(.4) + (-.4)(.1)\big) = \sigma(-.12) = .470$$

Output Layer:

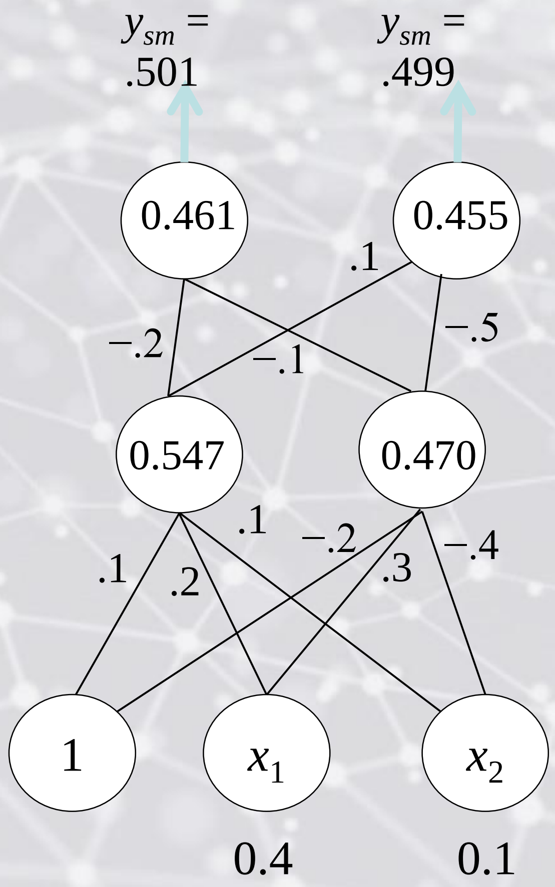


$$o_1 = \sigma\big((-.2)(.547)+(-.1)(.470)\big) = \sigma(-.1564) = .461$$

$$o_2 = \sigma\big((.1)(.547)+(-.5)(.470)\big) = \sigma(-.180) = .455$$

“Softmax” operation

Often used to turn output values into a <u>probability distribution</u>



$$y_{sm}(o_i) = \frac{e^{o_i}}{\sum_{k=1}^{K} e^{o_k}},$$

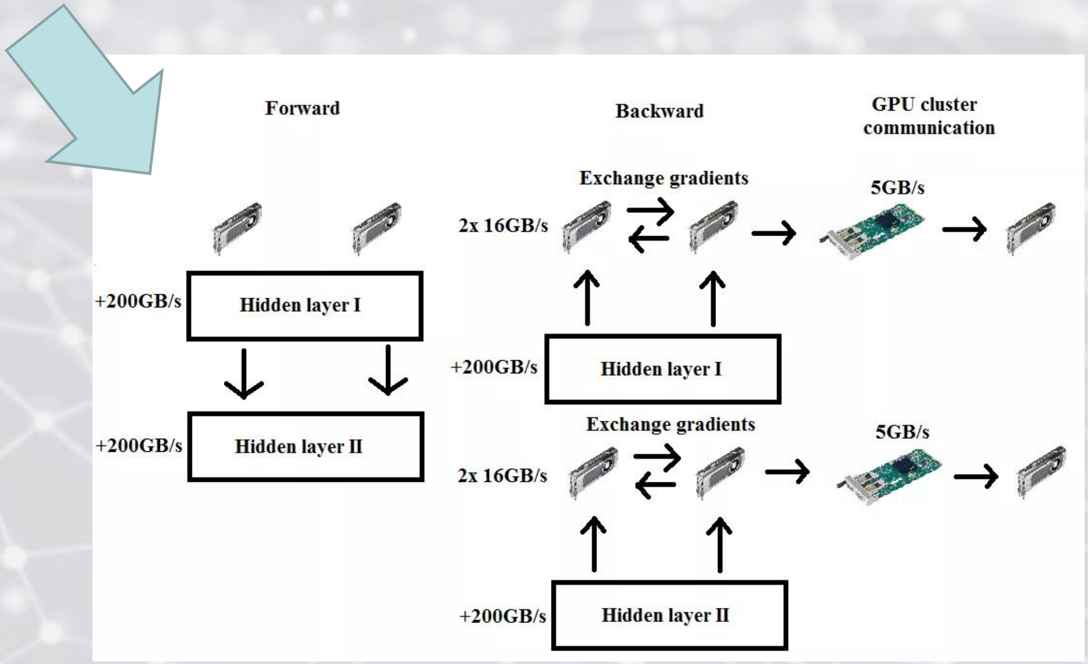
where K is the number of output units.

What kinds of problems are suitable for neural networks?

- Have sufficient training data
- Long training times are acceptable
- Not necessary for humans to understand learned target function or hypothesis

Advantages of neural networks

- Designed to be parallelized (e.g. split minibatches, use GPUs)



- Robust on noisy training data

- Fast to evaluate new examples

Training a multi-layer neural network

Repeat for a given number of epochs or until accuracy on training data is acceptable:

For each training example:

1. Present input to the input layer.
2. **Forward propagate** the activations times the weights to each node in the hidden layer.
3. **Forward propagate** the activations times weights from the hidden layer to the output layer.
4. At each output unit, **determine the error**.
5. Run the <u>**back-propagation** algorithm one layer at a time</u> to update all weights in the network.

Training a multilayer neural network with back-propagation (stochastic gradient descent)

- Suppose training example has form (**x**, **t**)

  (i.e., both input and target are vectors).

- Error (or "loss") E is sum-squared error over all output units:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{k \in \text{output layer}} (t_k - o_k)^2$$

- Goal of learning is to minimize the mean sum-squared error over the training set.

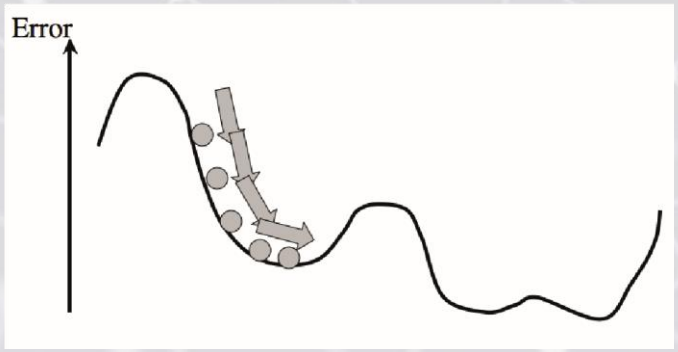
Training a multilayer neural network with back-propagation (stochastic gradient descent)

- Idea -- Minimize sum-of-squares error

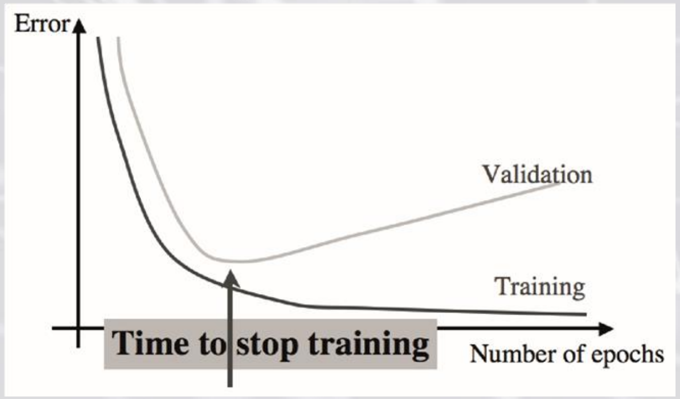
$$E(\mathbf{w}) = \frac{1}{2} \sum_{k \in \text{output layer}} (t_k - o_k)^2$$

over the entire training data set.

- Note that we "tune" the parameters of the NN (the weights) during training.

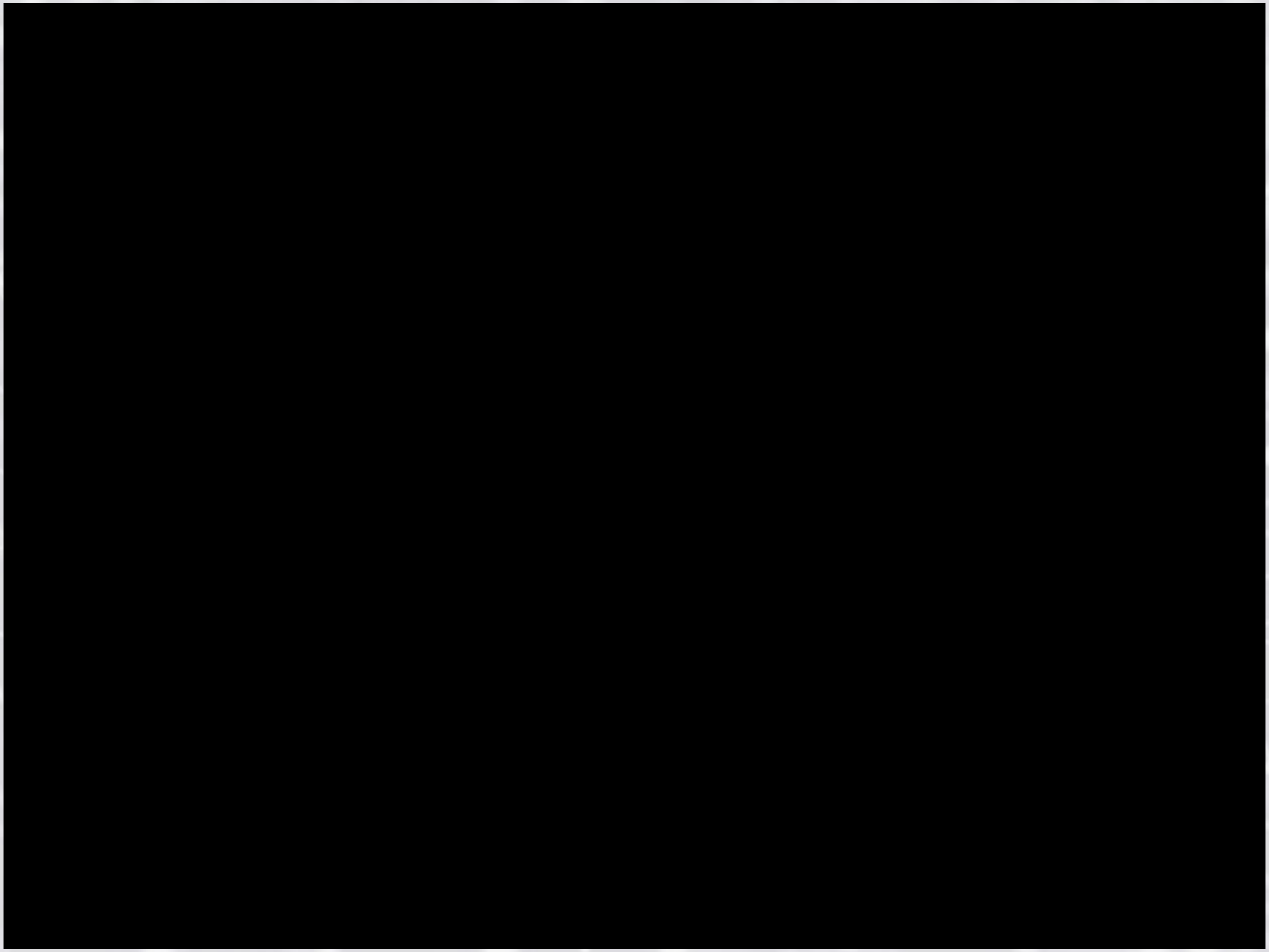


The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.



Geoffrey Hinton: NN training with MNIST

Aiva: AI Composed Music (2017)



Later in the slides we will **derive the back-propagation equations** (you can also find a derivation in the text).

The derivation can be somewhat challenging, however, you only need **one basic tool to derive them**: multi-variate differentiation (e.g. *chain rule*, *partial derivatives*).

For now, let's just walk through the basic algorithm.

Backpropagation algorithm

(Stochastic Gradient Descent)

- Initialize the network weights **w** to small random numbers (e.g., between −0.05 and 0.05).

- Until the termination condition is met, Do:
  - For each (**x**,**t**) ∈ training set, Do:
    1. *Propagate the input forward:*
       - Input **x** to the network and compute the activation h_j of each hidden unit j.
       - Compute the activation o_k of each output unit k.

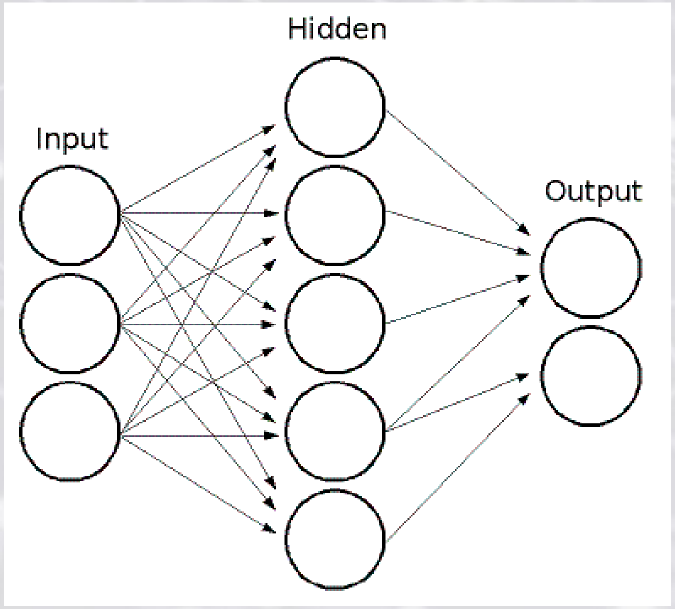
2. *Calculate error terms*

For each **output** unit *k*, calculate error term δ_k :

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

For each **hidden** unit *j*, calculate error term δ_j:

$$d_j \leftarrow h_j(1 - h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$



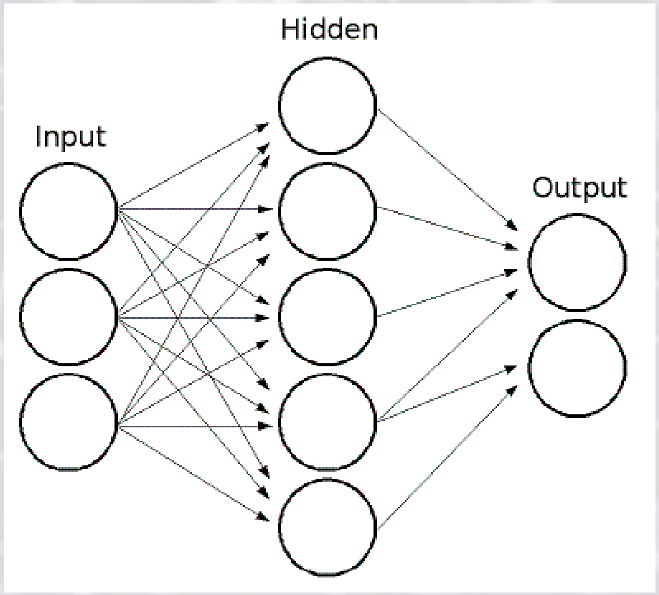
2. *Calculate error terms*

For each **output** unit *k*, calculate error term δ_k :

$$\boldsymbol{\delta}_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

For each **hidden** unit *j*, calculate error term δ_j:

$$d_j \leftarrow h_j(1 - h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$



3. Update weights

Hidden to Output layer: For each weight w_{kj}

$$w_{kj} \leftarrow w_{kj} + Dw_{kj}$$

where

$$Dw_{kj} = hd_k h_j$$

Input to Hidden layer: For each weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + Dw_{ji}$$

where

$$Dw_{ji} = hd_j x_i$$

Backpropagation Algorithm (BP)

- **Forwards Phase**: compute the activation of each neuron in the hidden layers and outputs using:

$$h_j = s\left(\sum_{i \in \text{input layer}} w_{ji} x_i + w_{j0} \right) \qquad o_k = s\left(\sum_{j \in \text{hidden layer}} w_{kj} h_j + w_{k0} \right)$$

- **Backwards pass**
- Compute the error at the output using: $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
- Compute the error at the hidden layer(s) using: $d_j \leftarrow h_j(1 - h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k \right)$
- Update the output layer weights using: $w_{kj} \leftarrow w_{kj} + Dw_{kj}$

  where $Dw_{kj} = hd_k h_j$
- Update the hidden layer weights using: $w_{ji} \leftarrow w_{ji} + Dw_{ji}$

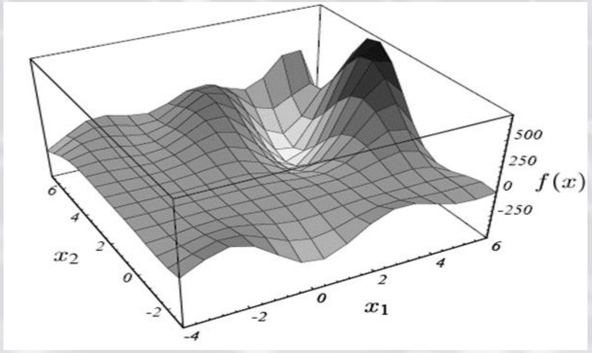
  where $Dw_{ji} = hd_j x_i$
- (If using *sequential updating*) randomize the order of the input vectors so that you don't train in exactly the same order each iteration.

Training Time

- The Aim is to balance between Generalization & Memorization (Minimizing cost function is not necessarily good idea).
  - Using two (or three) disjoint sets:
    - Training-Testing Sets
    - Training-Testing-Validation Sets
  - As long as the error for the training-testing set decreases, training continues (unless max # iterations achieved).
  - When the error begins to increase , the net is starting to memorize.

Some Pros and Cons of BP

- Connectionism
  - Biological Issues
    - No excitatory or inhibitory for real neurons
    - No Global connection in MLP
    - No backward propagation in real neurons
  - Useful in parallel hardware implementation
- Computational Efficiency
  - Learning Algorithm is said to be **computationally efficient** , when its complexity is polynomial.
  - The BP algorithm is computationally efficient.
    - In MLP with a total of W weights, its complexity is linear in W
- Local Minima
  - Presence of local minima is a significant issue, particularly for high dimensional data.



Batch (or "True") Gradient Descent: Change weights only after averaging gradients from all training examples:

Weights from hidden units to output units:

$$\Delta w_{kj} = \eta \frac{1}{M} \sum_{m=1}^{M} \delta_k^m h_j^m$$

Weights from input units to hidden units:

$$\Delta w_{ji} = \eta \frac{1}{M} \sum_{m=1}^{M} \delta_j^m x_i^m$$

Mini-Batch Gradient Descent: Change weights only after averaging gradients from a subset of B training examples:

At each iteration *t*: Get next subset of B training examples, B_t , until all examples have been processed.

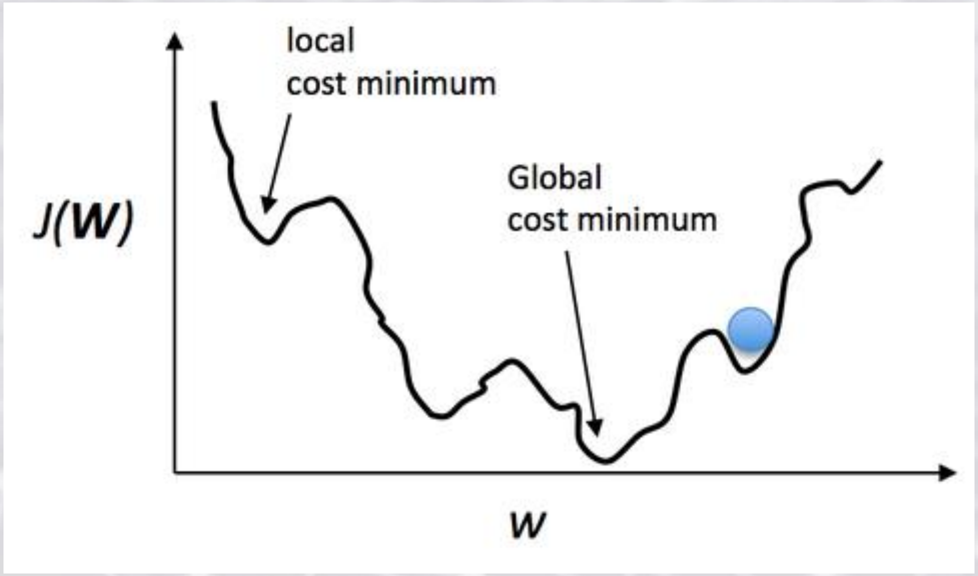
Weights from hidden units to output units:

$$\Delta w_{kj} = \eta \frac{1}{B} \sum_{m \in B_t} \delta_k^m h_j^m$$

Weights from input units to hidden units:

$$\Delta w_{ji} = \eta \frac{1}{B} \sum_{m \in B_t} \delta_j^m x_i^m$$

Local Minima, Momentum, etc.



- Recall that BP is an instance of "hill climbing" (e.g. gradient descent). With non-convex problems we are not guaranteed to settle into a global minimum.
- If we think of the analogy of a ball rolling down a hill, we can consider giving the ball some "weight" by implementing a **momentum term**.
- The purpose of the momentum term is to mitigate the instance of getting "stuck" in a local minimum (i.e. a "valley") and to avoid performance oscillations during training.

Momentum

Introduce a *momentum term*, in which change in weight is dependent on past weight change:

$$\Delta w_{kj}^{t} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1} \qquad \text{(hidden-to-output)}$$

$$\Delta w_{ji}^{t} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1} \qquad \text{(input-to-hidden)}$$

where *t* is the iteration through the main loop of back-propagation.

α is a parameter between 0 and 1; α determines the “strength” of the momentum term.

The idea is to <u>keep weight changes moving in the same direction</u>.

Update weights, with momentum

Hidden to Output layer: For each weight *w_{kj}*

$$w_{kj} \leftarrow w_{kj} + \mathrm{D}w_{kj}$$

where

$$\Delta w_{kj} = \eta \delta_k h_j \boxed{+ \alpha \Delta w_{kj}^{t-1}}$$

Input to Hidden layer: For each weight *w_{ji}*

$$w_{ji} \leftarrow w_{ji} + \mathrm{D}w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_i \boxed{+ \alpha \Delta w_{ji}^{t-1}}$$

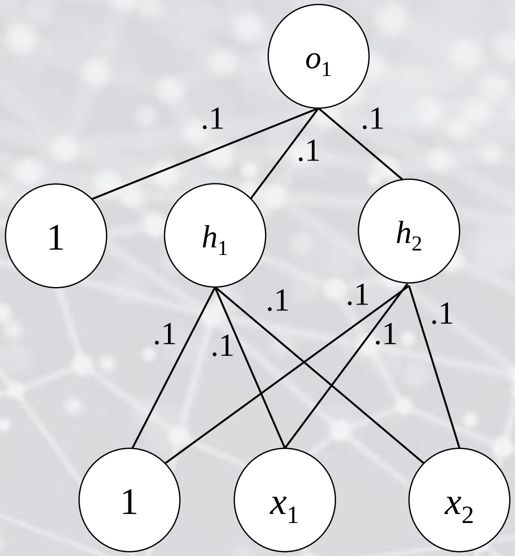
Backprop Example

Training set:

1	**0**	**Label: 0.9**
0	**1**	**Label: -.3**

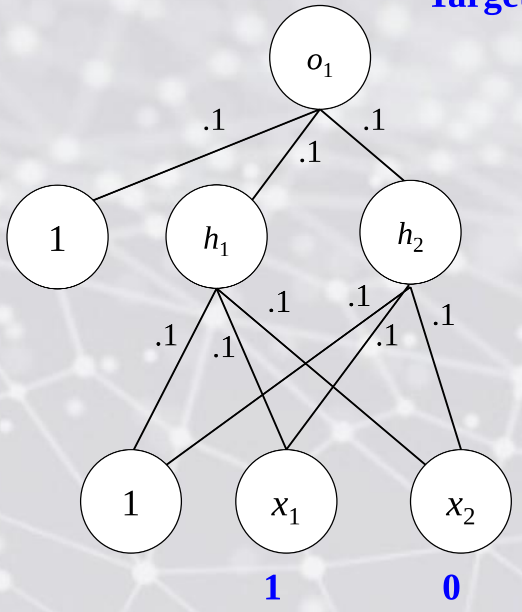
Test set:

1	**1**	**Label: .8**



Training set:				Test set:		
1	0	Label: .9		1	1	Label: .8
0	1	Label: -.3				

Target: .9



Training set:

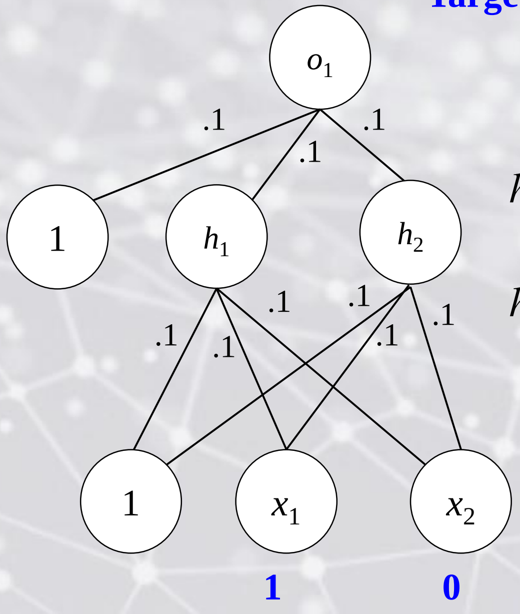
1 0 Label: .9

0 1 Label: -.3

Test set:

1 1 Label: .8

Target: .9



$$h_1 = \sigma\left((1)(.1)+(1)(.1)+(0)(.1)\right) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

$$h_2 = \sigma\left((1)(.1)+(1)(.1)+(0)(.1)\right) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

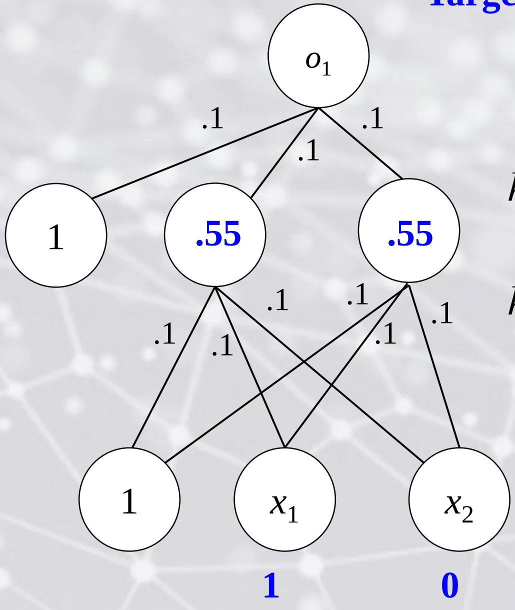
Training set:

1	**0**	**Label: .9**
0	**1**	**Label: -.3**

Test set:

1	**1**	**Label: .8**

Target: .9



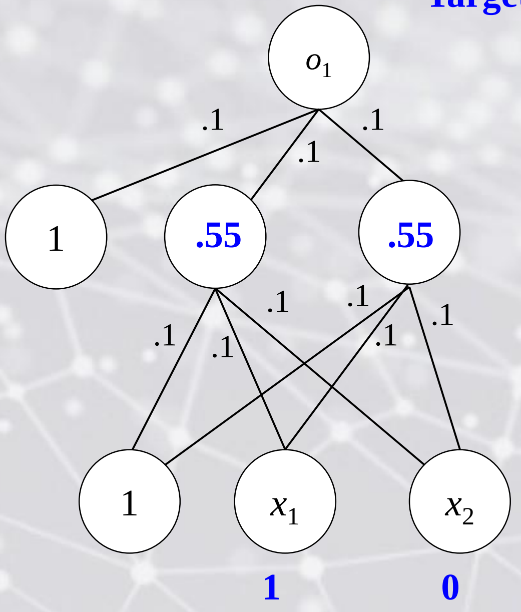
$$h_1 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

$$h_2 = \sigma\big((1)(.1)+(1)(.1)+(0)(.1)\big) = \sigma(.2) = \frac{1}{1+e^{-.2}} = .55$$

"Forward Phase" – hidden layers

Training set:			**Test set:**		
1	**0**	**Label: .9**	**1**	**1**	**Label: .8**
0	**1**	**Label: -.3**			

Target: .9

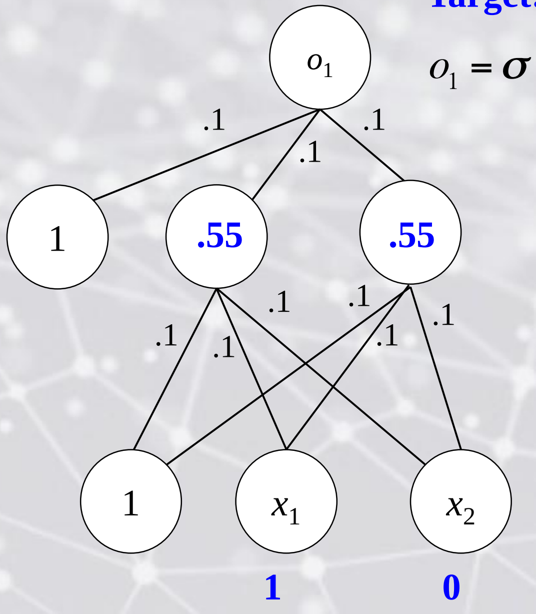


Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8



Target: .9

$$o_1 = \sigma\big((1)(.1) + (.55)(.1) + (.55)(.1)\big) = \sigma(.21) = \frac{1}{1+e^{-.21}} = .552$$

"Forward Phase" – output layer

Training set:

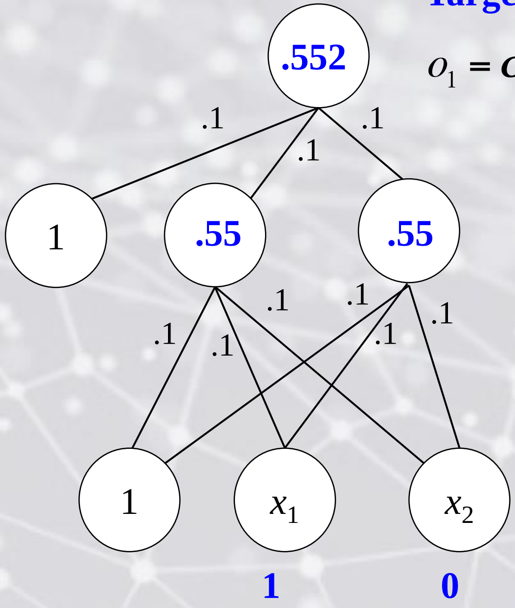
1	**0**	**Label: .9**
0	**1**	**Label: -.3**

Test set:

1	**1**	**Label: .8**

Target: .9

$$o_1 = \sigma\left((1)(.1) + (.55)(.1) + (.55)(.1)\right) = \sigma(.21) = \frac{1}{1+e^{-.21}} = .552$$



"Forward Phase" – output layer

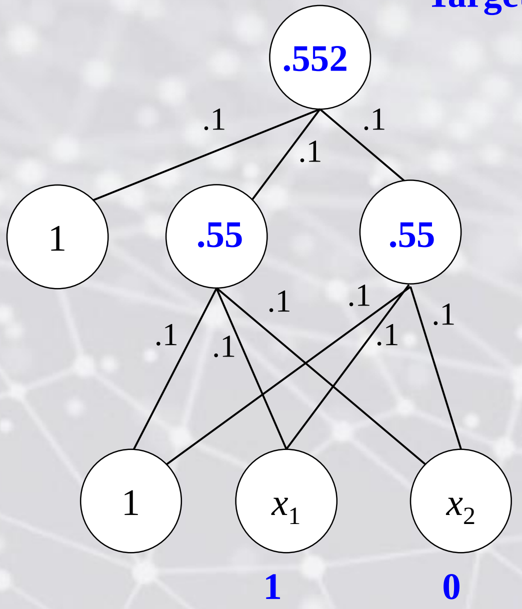
Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8

Target: .9



Output weight Updates

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

Hidden weight Updates

$$d_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, d_k\right)$$

"Backward Phase"

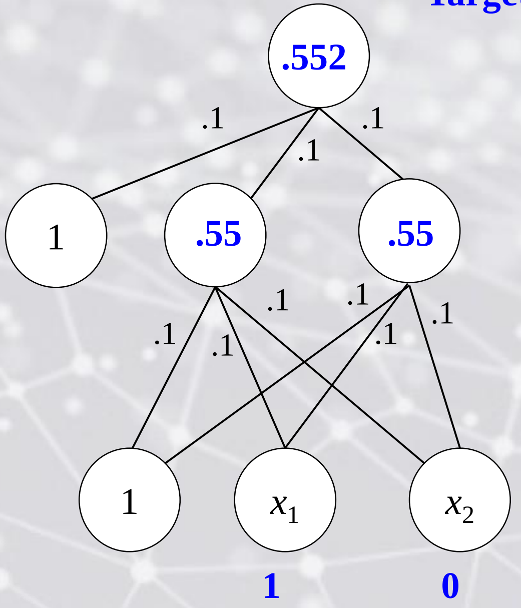
Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8

Target: .9



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = \left(.55\right)(.45)(.1)\left(.086\right) = .002$$

$$\delta_{j=2} = \left(.55\right)(.45)(.1)\left(.086\right) = .002$$

"Backward Phase"

$$\boldsymbol{\delta}_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

Output weight Updates

$$\delta_j \leftarrow h_j(1 - h_j)\left(\sum_{k \in \text{output units}} w_{kj}\, \delta_k\right)$$

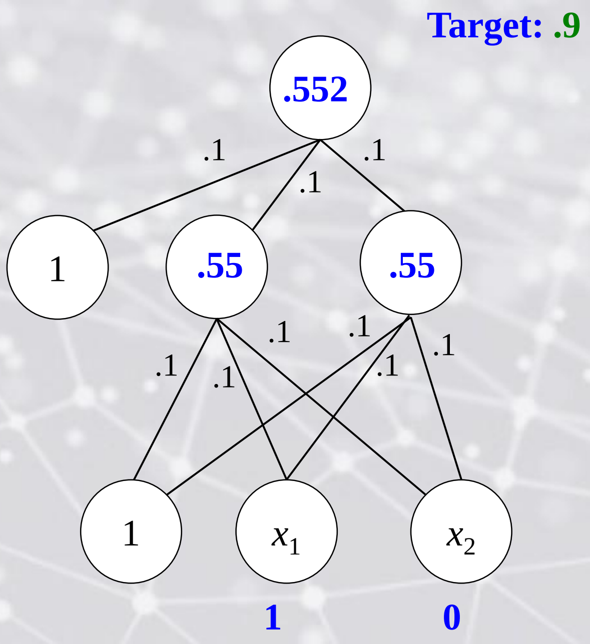
Hidden weight Updates

Training set:

1	**0**	**Label: .9**
0	**1**	**Label: -.3**

Test set:

1	**1**	**Label: .8**



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)\,(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)\,(.086) = .002$$

"Backward Phase"

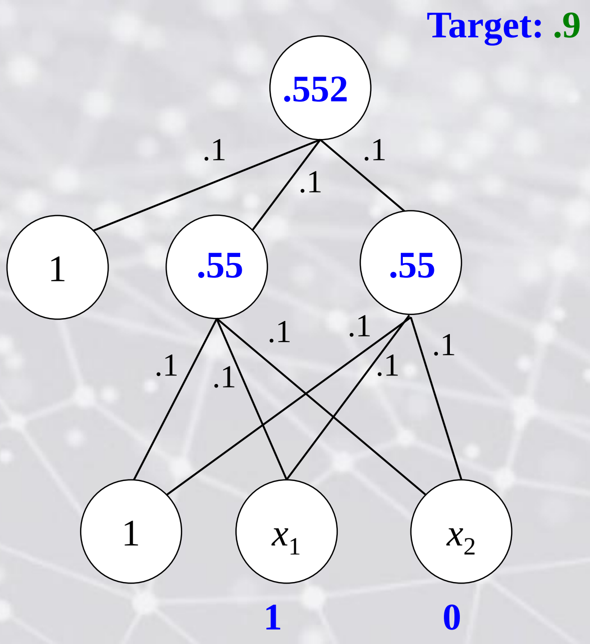
$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

Output weight Updates

$$\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k \in \text{output units}} w_{kj}\,\delta_k\right)$$

Hidden weight Updates

Training set:			Test set:		
1	0	Label: Positive	1	1	Label: Positive
0	1	Label: Negative			



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\boxed{\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}}$$

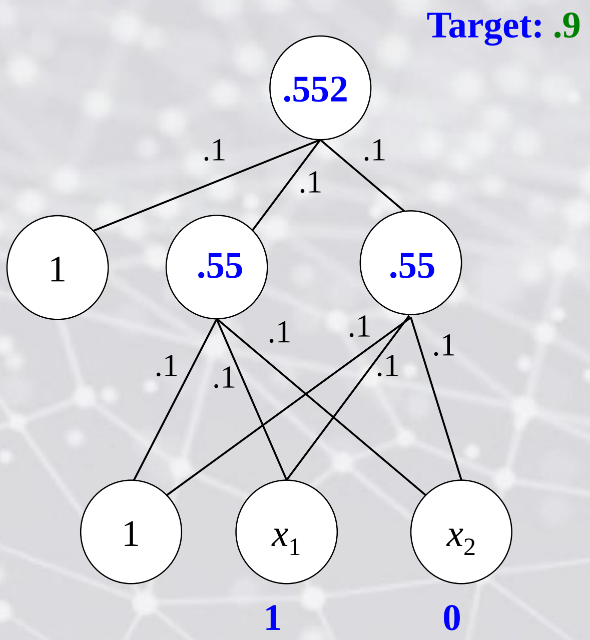
Update** hidden-to-output weights **(learning rate = 0.2; momentum = 0.9):

Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\boxed{\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}}$$

Update ***hidden-to-output weights*** **(*learning rate* = 0.2; *momentum* = 0.9): Hidden unit j=1**

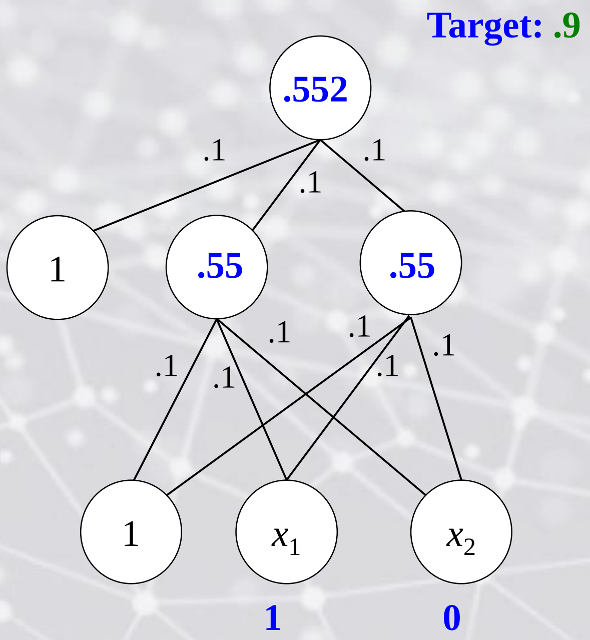
$$\Delta w^1_{k=1,j=0} = (.2)(.086)(1) + (.9)(0) = .0172$$

$$\Delta w^1_{k=1,j=1} = (.2)(.086)(.55) + (.9)(0) = .0095$$

$$\Delta w^1_{k=1,j=2} = (.2)(.086)(.55) + (.9)(0) = .0095$$

Training set:		
1	0	Label: .9
0	1	Label: -.3

Test set:		
1	1	Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\boxed{\Delta w_{kj} = \eta \delta_k h_j + \alpha \Delta w_{kj}^{t-1}}$$

Update hidden-to-output weights (learning rate = 0.2; momentum = 0.9):

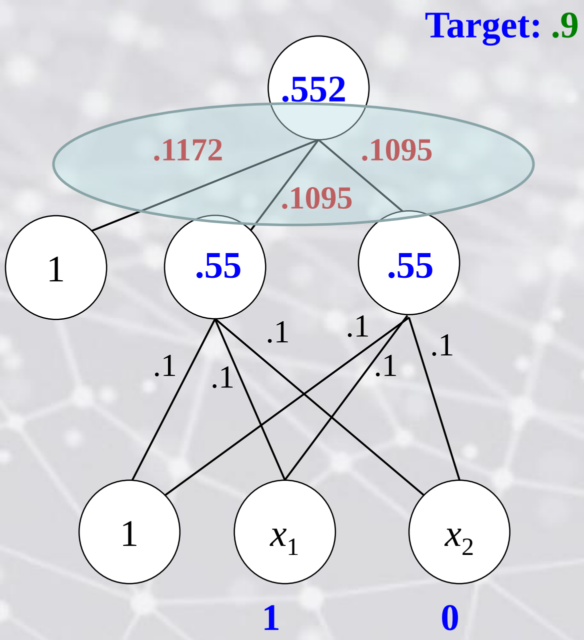
$$\Delta w^1_{k=1,j=0} = (.2)(.086)(1) + (.9)(0) = .0172 \qquad w^1_{k=1,j=0} = .1 + .0172 = .1172$$

$$\Delta w^1_{k=1,j=1} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=1} = .1 + .0095 = .1095$$

$$\Delta w^1_{k=1,j=2} = (.2)(.086)(.55) + (.9)(0) = .0095 \qquad w^1_{k=1,j=2} = .1 + .0095 = .1095$$

Training set:		
1	**0**	**Label: .9**
0	**1**	**Label: -.3**

Test set:		
1	**1**	**Label: .8**



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

Update** hidden-to-output weights **(learning rate = 0.2;** momentum **= 0.9):

$$\Delta w^1_{k=1,j=0} = (.2)(.086)(1) + (.9)(0) = .0172$$

$$w^1_{k=1,j=0} = .1 + .0172 = .1172$$

$$\Delta w^1_{k=1,j=1} = (.2)(.086)(.55) + (.9)(0) = .0095$$

$$w^1_{k=1,j=1} = .1 + .0095 = .1095$$

$$\Delta w^1_{k=1,j=2} = (.2)(.086)(.55) + (.9)(0) = .0095$$

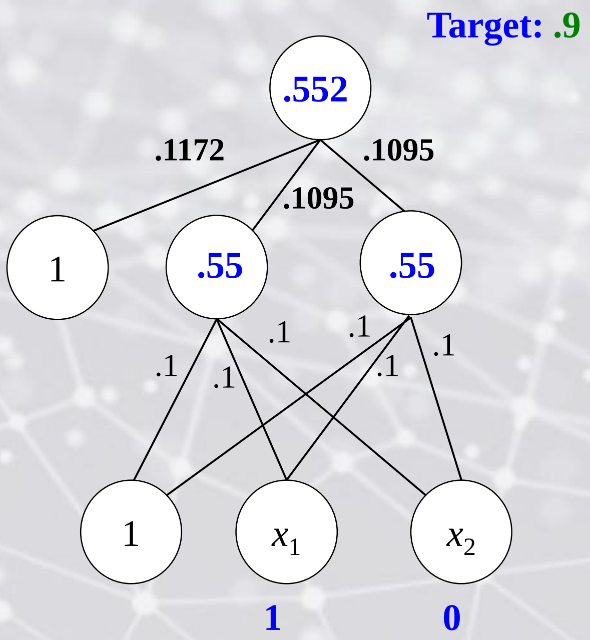
$$w^1_{k=1,j=2} = .1 + .0095 = .1095$$

Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)\left(.086\right) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)\left(.086\right) = .002$$

$$\boxed{\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}}$$

Update ***input-to-hidden weights*** **(*learning rate* = 0.2; *momentum* = 0.9):**

$$\Delta w^1_{j=1,i=0} = (.2)\left(.002\right)(1) + (.9)(0) = .0004$$

$$w^1_{j=1,i=0} = .1 + .0004 = .1004$$

$$\Delta w^1_{j=1,i=1} = (.2)\left(.002\right)(1) + (.9)(0) = .0004$$

$$w^1_{j=1,i=1} = .1 + .0004 = .1004$$

$$\text{D}w^1_{j=1,i=2} = (.2)\left(.002\right)(0) + (.9)(0) = 0$$

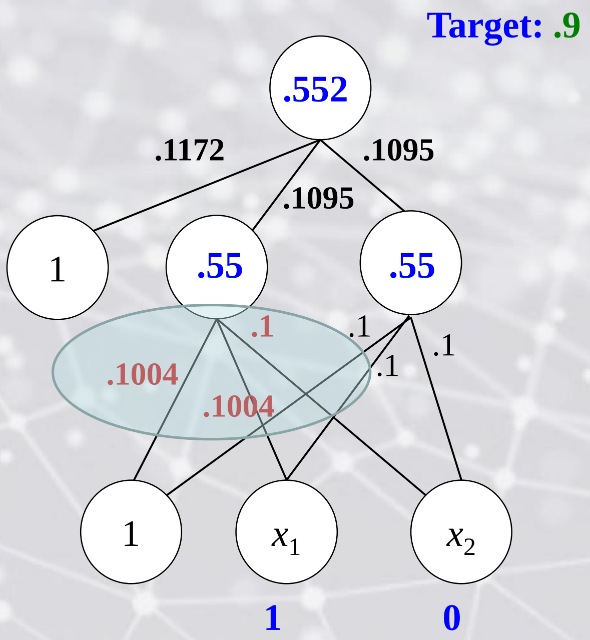
$$w^1_{j=1,i=2} = .1$$

Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\boxed{\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}}$$

Update ***input-to-hidden weights*** **(*learning rate* = 0.2; *momentum* = 0.9):**

$$\Delta w^1_{j=1,i=0} = (.2)(.002)(1) + (.9)(0) = .0004$$

$$w^1_{j=1,i=0} = .1 + .0004 = .1004$$

$$\Delta w^1_{j=1,i=1} = (.2)(.002)(1) + (.9)(0) = .0004$$

$$w^1_{j=1,i=1} = .1 + .0004 = .1004$$

$$\text{D}w^1_{j=1,i=2} = (.2)(.002)(0) + (.9)(0) = 0$$

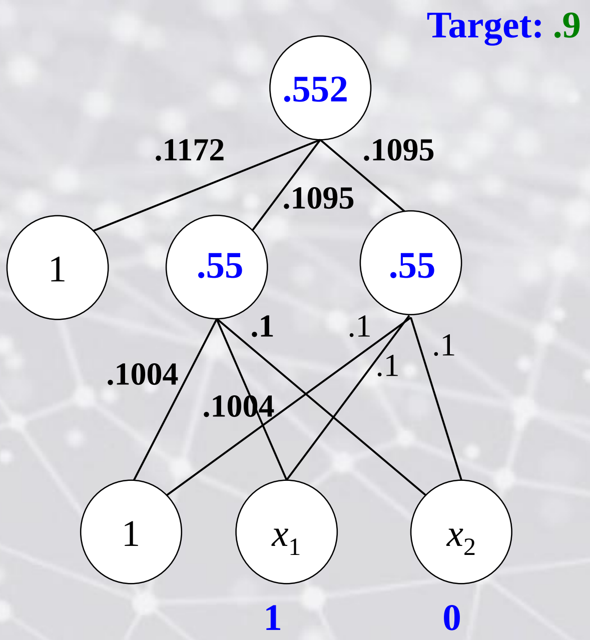
$$w^1_{j=1,i=2} = .1$$

Training set:

1	0	Label: .9
0	1	Label: -.3

Test set:

1	1	Label: .8



Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9-.552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

$$\boxed{\Delta w_{ji} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1}}$$

Update input-to-hidden weights (learning rate = 0.2; momentum = 0.9): **Hidden unit j=2**

$$\Delta w^1_{j=2,i=0} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=2,i=0} = .1 + .0004 = .1004$$

$$\Delta w^1_{j=2,i=1} = (.2)(.002)(1) + (.9)(0) = .0004 \qquad w^1_{j=2,i=1} = .1 + .0004 = .1004$$

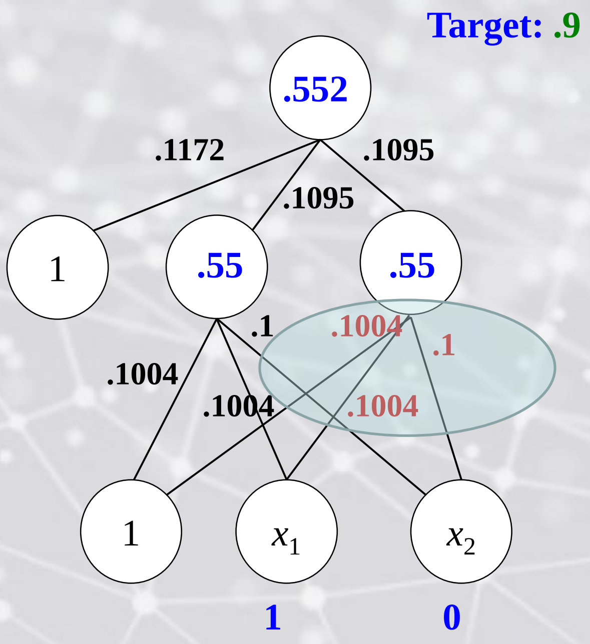
$$\Delta w^1_{j=2,i=2} = (.2)(.002)(0) + (.9)(0) = 0 \qquad w^1_{j=2,i=2} = .1$$

Training set:

1	**0**	**Label: .9**
0	**1**	**Label: -.3**

Test set:

1	**1**	**Label: .8**



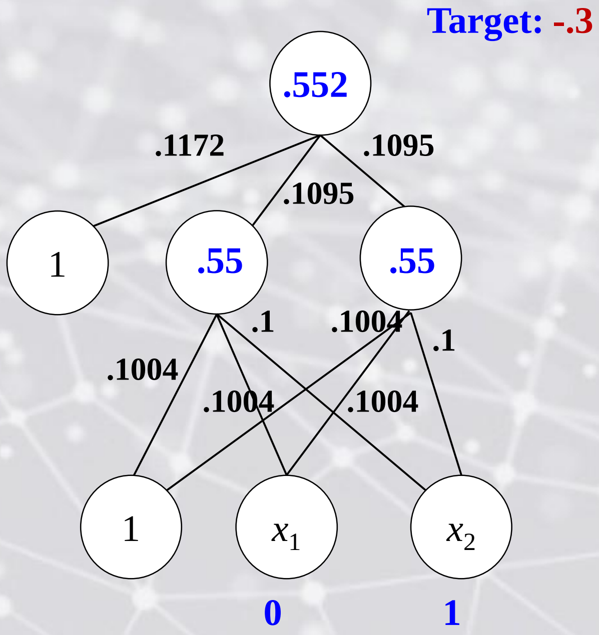
Calculate error terms:

$$\delta_{k=1} = .552(.448)(.9 - .552) = .086$$

$$\delta_{j=1} = (.55)(.45)(.1)(.086) = .002$$

$$\delta_{j=2} = (.55)(.45)(.1)(.086) = .002$$

Training set:			Test set:		
1	0	Label: .9	1	1	Label: .8
0	1	Label: -.3			

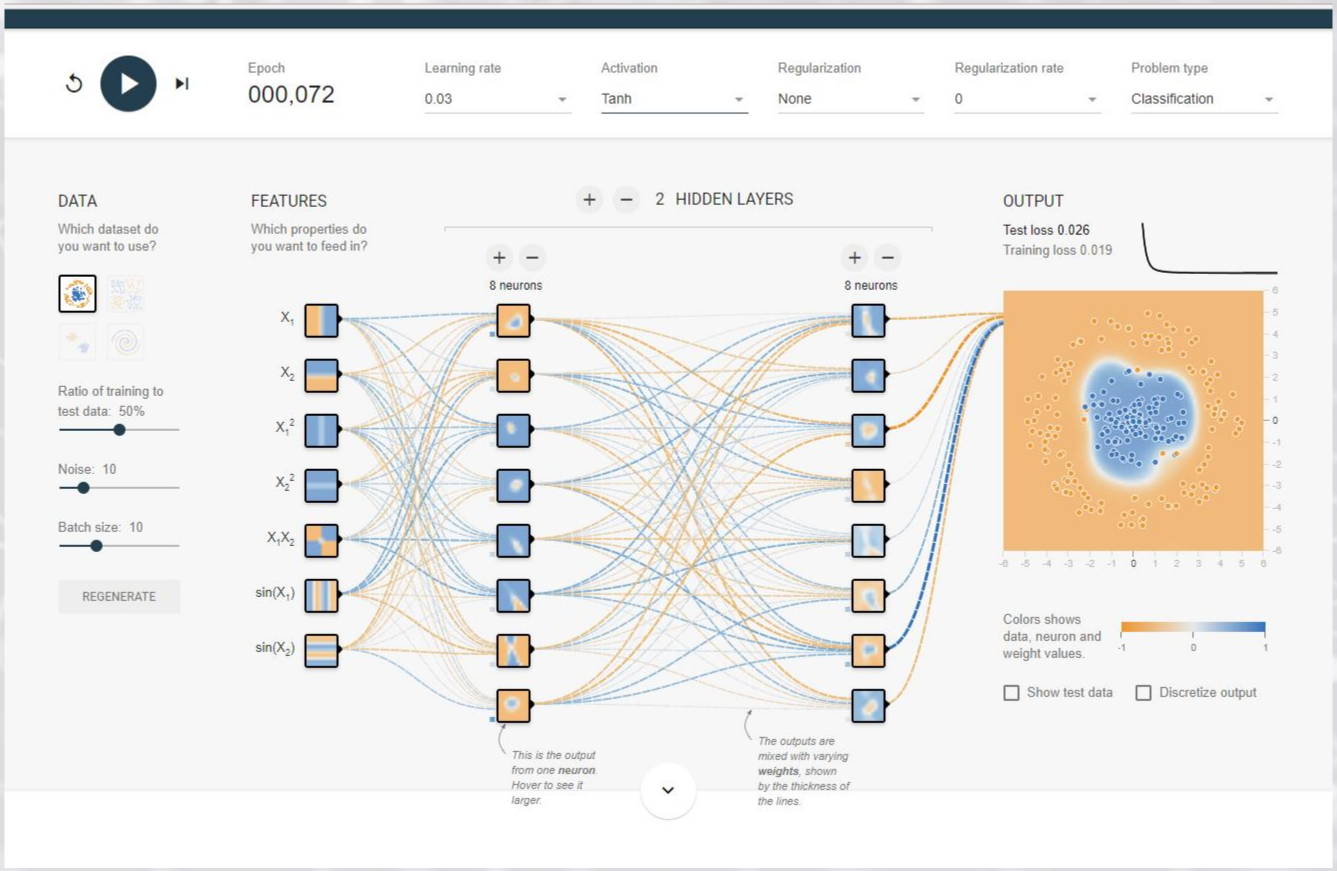


Note: This is time step 2, so momentum term will be nonzero…

Another detailed backprop example:
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Play with a Neural Network

http://playground.tensorflow.org/



DATA
Which dataset do you want to use?
FEATURES
Which properties do you want to feed in?
+ − 2 HIDDEN LAYERS
8 neurons
8 neurons
OUTPUT
REGENERATE
Colors shows data, neuron and weight values.
Show test data
Discretize output
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.

Epoch 000,072
Learning rate 0.03
Activation Tanh
Regularization None
Regularization rate 0
Problem type Classification

Ratio of training to test data: 50%
Noise: 10
Batch size: 10

Test loss 0.026
Training loss 0.019

Example: Face recognition

(From T. M. Mitchell, *Machine Learning*, Chapter 4)

Code (C) and data at http://www.cs.cmu.edu/~tom/faces.html

- **Task:** classify camera images of various people in various poses.

- **Data:** Photos, varying:
  - Facial expression: *happy, sad, angry, neutral*
  - Direction person is facing: *left, right, straight ahead, up*
  - Wearing sunglasses?: *yes, no*

Within these, variation in background, clothes, position of face for a given person.

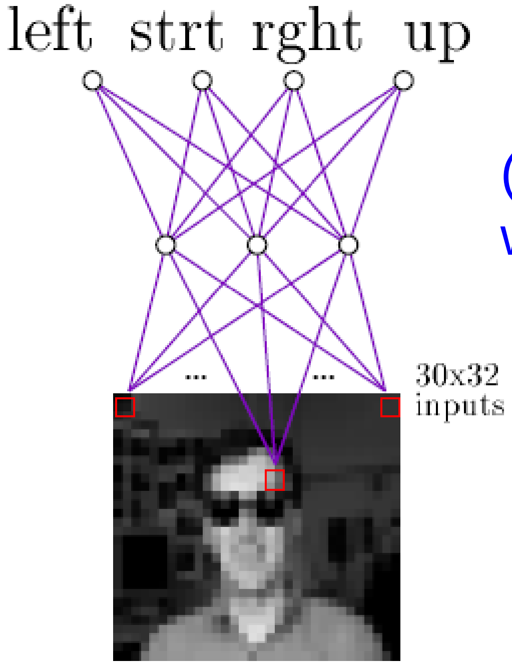
an2i_left_angry_open_4

an2i_right_sad_sunglasses_4

glickman_left_angry_open_4

Design Choices

- Input encoding
- Output encoding
- Network topology
- Learning rate
- Momentum



(Note: bias unit and weights not shown)

Typical input images

- Preprocessing of photo:
  - Create 30x32 coarse resolution version of 120x128 image
  - This makes size of neural network more manageable

- Input to neural network:
  - Photo is encoded as 30x32 = 960 pixel intensity values, scaled to be in [0,1]
  - One input unit per pixel

- Output units:
  - Encode classification of input photo

- Possible target functions for neural network:
  - Direction person is facing
  - Identity of person
  - Gender of person
  - Facial expression
  - etc.

- As an example, consider target of “direction person is facing”.

Target function

- Target function is:
  - Output unit should have activation 0.9 if it corresponds to correct classification
  - Otherwise output unit should have activation 0.1
- Use these values instead of 1 and 0, since sigmoid units can't produce 1 and 0 activation.

Other parameters

- Learning rate $\eta = 0.3$

- Momentum $\alpha = 0.3$

- If these are set too high, training fails to converge on network with acceptable error over training set.

- If these are set too low, training takes much longer.

Training

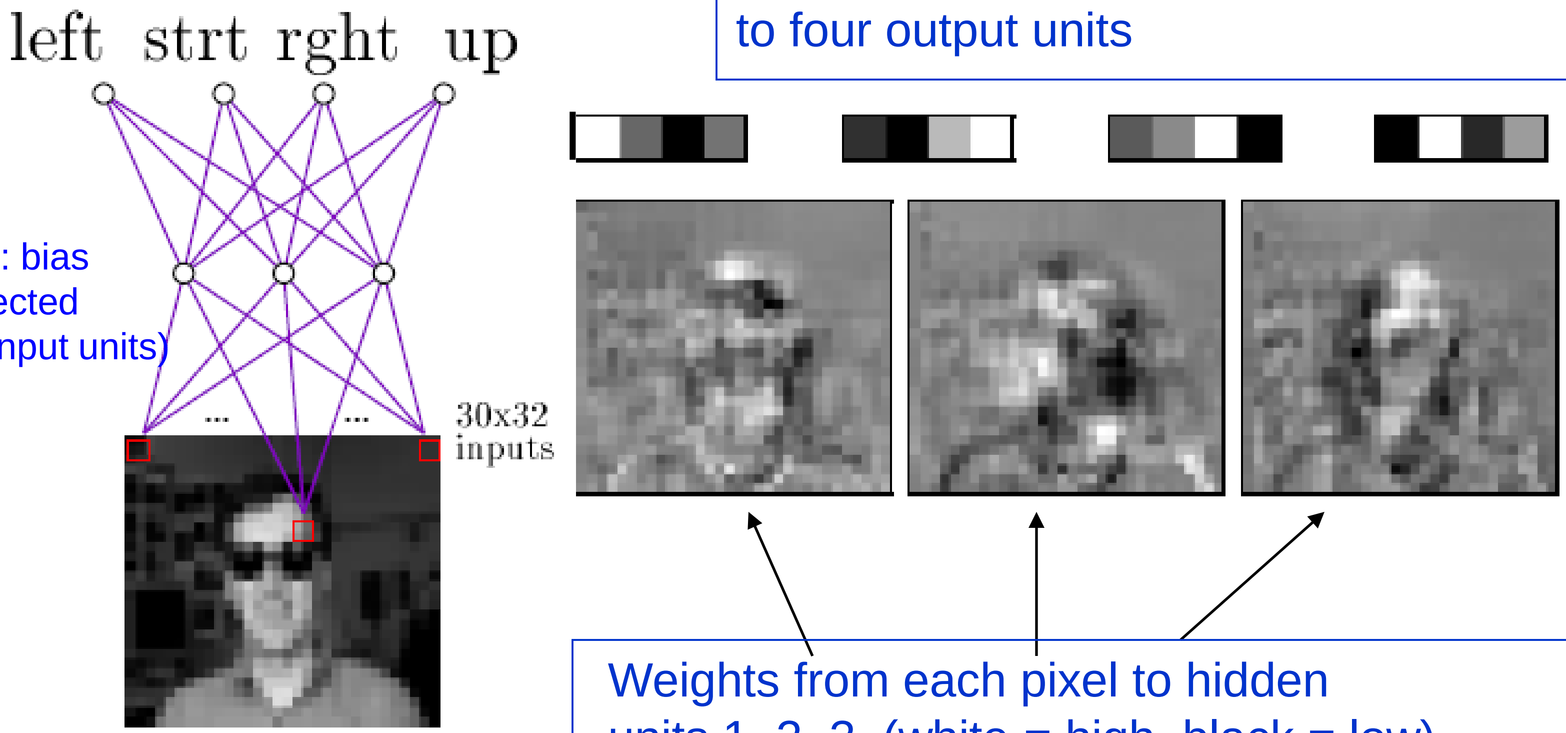
- For maximum number of epochs:
  - For each training example
    - Input photo to network
    - Propagate activations to output units
    - Determine error in output units
    - Adjust weights using back-propagation algorithm

- Demo of code

(from http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html)

Weights from each hidden unit to four output units

Not shown: bias unit (connected to all non-input units)



Weights from each pixel to hidden units 1, 2, 3 (white = high, black = low)

After 100 epochs of training.
(From T. M. Mitchell, Machine Learning)

- Hidden unit 2 has high positive weights from right side of face.
- If person is looking to the right, this will cause unit 2 to have high activation.
- Output unit *right* has high positive weight from hidden unit 2.

Understanding weight values

- After training:
  - Weights from input to hidden layer: high positive in certain facial regions

  - “Right” output unit has strong positive from second hidden unit, strong negative from third hidden unit.
    - Second hidden unit has positive weights on right side of face (aligns with bright skin of person turned to right) and negative weights on top of head (aligns with dark hair of person turned to right)

    - Third hidden unit has negative weights on right side of face, so will output value close to zero for person turned to right.

Hidden Units

- Two few – can’t represent target function

- Too many – leads to overfitting

Use “cross-validation” to decide number of hidden units.

Weight decay

- Modify error function to add a penalty for magnitude of weight vector, to decrease *overfitting*.

- This modifies weight update formula (with momentum) to:

$$\Delta w_{ji}^{t} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{t-1} - \lambda w_{ji}^{t-1}$$

where λ is a parameter between 0 and 1.

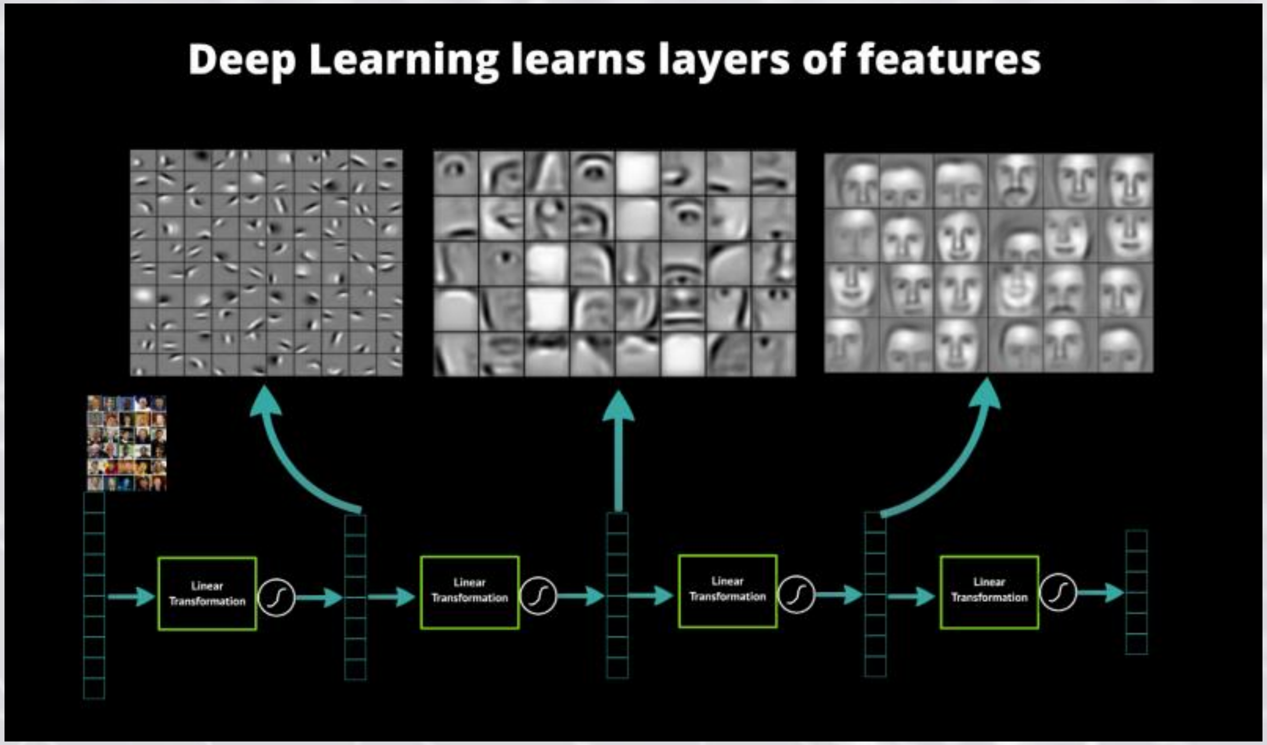
This kind of penalty is called "regularization"; common to use **L1 or L2 regularization** (references norm used for "penalty" term – if you're a "Bayesian" this is equivalent to putting a prior centered at zero on the network weights).

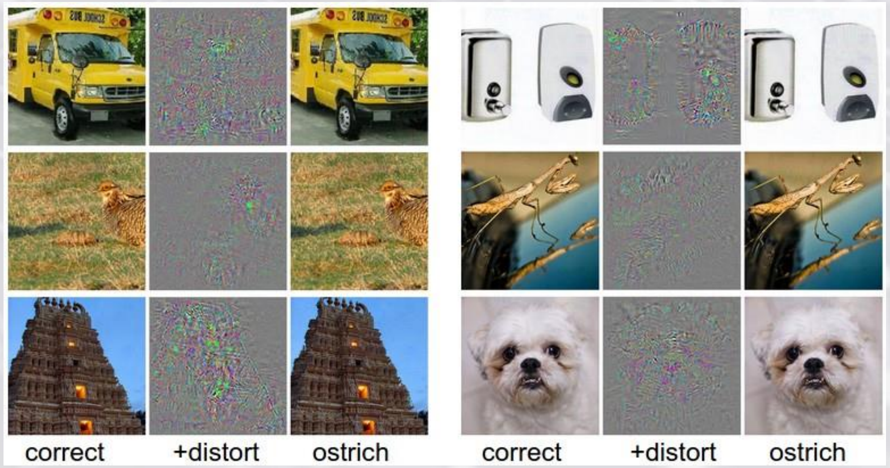
Other important topics for NNs

Please feel free to talk to me about any of these more advanced topics or investigate them on your own:

- Q: What does a NN “learn”? Should it be treated as a black box or should it be interpretable?
- Deep Learning and Hierarchical Models Learning
- Other methods for training the weights; different loss functions and optimization techniques (e.g. GAs)
- Different architectures: RNNs, CNN, LSTMs, etc.
- Sparse models (e.g. using “dropout”)
- Adversarial Networks/Generative Networks

Other important topics for NNs



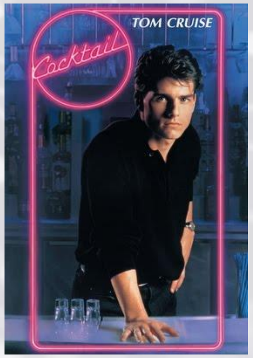


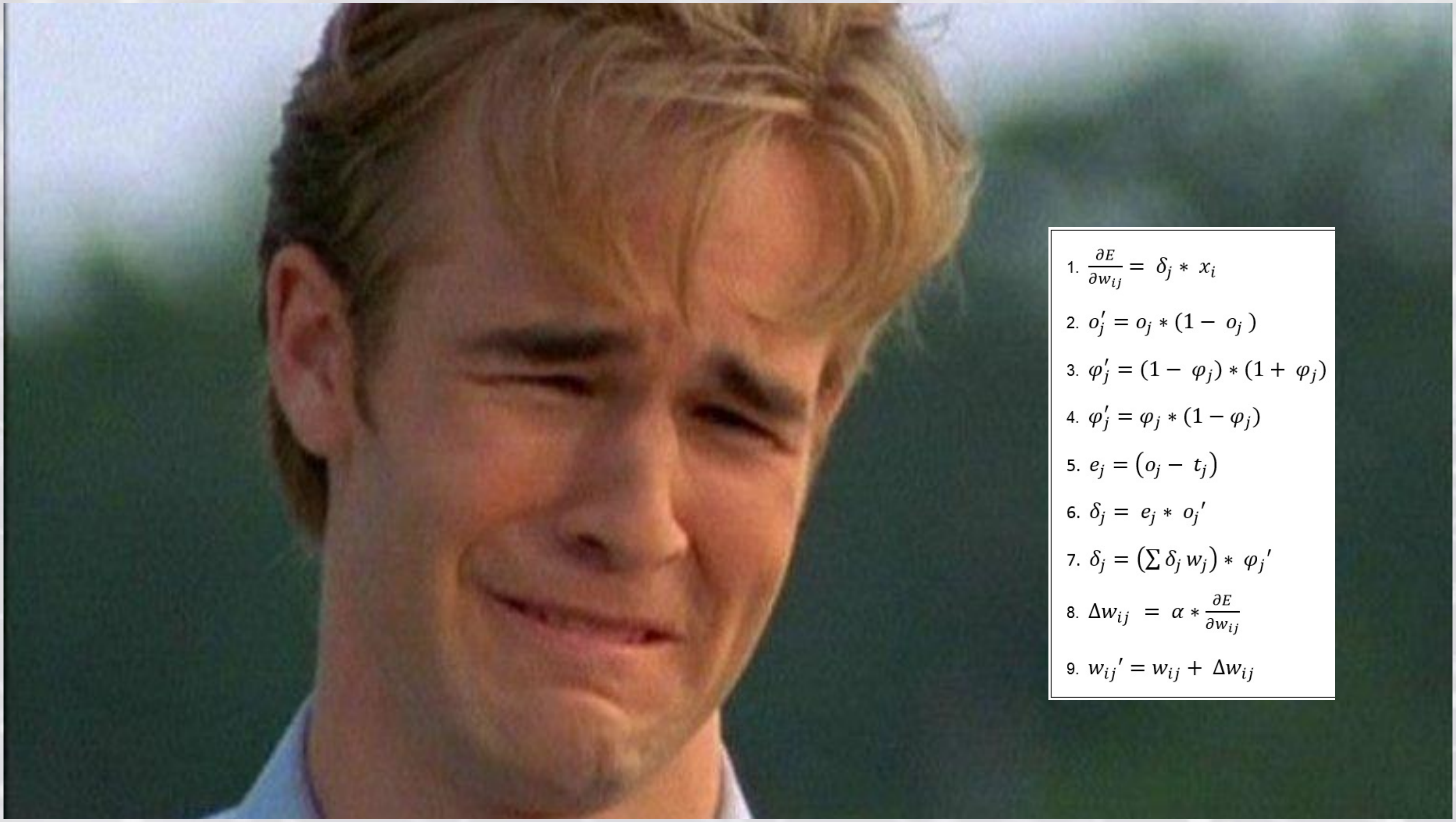
⚠ WARNING

THIS POST CONTAINS MATH

(*) Here is a derivation (later slides contain a derivation with visuals) of BP (note our text also has a derivation pp. 101-108). Time permitting, I'll walk us through this. If you require further details don't hesitate to ask for help.

(*) "*Will this be on the exam?*" **No**, but understanding the material at this level makes you a better person – moreover, it will make your friends envious, your mother will love you more, and strangers at cocktail parties will be drawn to you like a magnet. You're welcome.





1. $\frac{\partial E}{\partial w_{ij}} = \delta_j * x_i$
2. $o_j' = o_j * (1 - o_j)$
3. $\varphi_j' = (1 - \varphi_j) * (1 + \varphi_j)$
4. $\varphi_j' = \varphi_j * (1 - \varphi_j)$
5. $e_j = (o_j - t_j)$
6. $\delta_j = e_j * o_j'$
7. $\delta_j = \left(\sum \delta_j w_j\right) * \varphi_j'$
8. $\Delta w_{ij} = \alpha * \frac{\partial E}{\partial w_{ij}}$
9. $w_{ij}' = w_{ij} + \Delta w_{ij}$

Backprop Derivation

What do we need to derive the backpropagation (BP) algorithm?

Only a basic knowledge of differential Calculus!

(*) Recall that we will use BP to update weights in both the hidden layer(s) and the output layer of our NN.

(*) We use the chain rule to "propagate" the error back through the network (following the "forward phase").

backprop is just the chain rule
backprop is just the chain rule + memoization
backprop is an instance of the method of Lagrange multipliers, which uses efficient block-coordinate steps on the multipliers & intermediate variables!
That's why it's correct, linear time and linear space.
backprop is an instance of the method of Lagrange multipliers, [...] which gives me the freedom to optimize & compute it with a million other methods!
One such method is reverse-mode (backprop), another is forward-mode, but actually there is a huge spectrum of methods that fall out of this elegant formulation; each give interesting algs with different time-space tradeoffs for gradient computation.
I can run optimization directly on the Lagrange dual - moving away from the de facto block-coordinate scheme.
I can even support cyclic computations thanks to this wonder unified view!

obligatory backprop meme

Backprop Derivation

(*) We'll call the current input **x** (a vector) and the output **y**; the activation function (throughout the network) will be denoted g(·).

(*) For simplicity, let's assume the NN contains only a single hidden layer (BP extends naturally for more layers); denote the weights of the network **v** and **w**, for the first and second layers respectively.

(*) Recall that "learning" entails tuning the weights of the network.

Backprop Derivation

(*) We wish to minimize the error function:

$$E(w) = \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 = \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik}x_i\right) - t_k\right]^2$$

Where **y** is the output, t is the target; **N** is the data set size and **L** is the number of nodes (in a given layer).

(*) We use *gradient descent*. In particular, we wish to know how the error function changes with respect to the different weights:

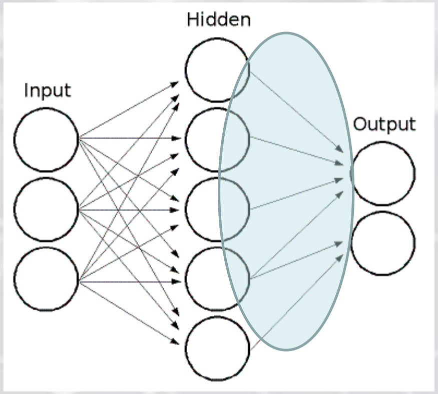
$$\frac{\partial E}{\partial w_{\varsigma\kappa}}$$

*Note: $j = \zeta, k = \kappa$ are fixed indices.

Backprop Derivation

(*) Let $a = g(h) = \frac{1}{1+e^{-h}}$ (the sigmoid function); recall that: $\boxed{a' = a(1-a)}$

(*) Using the *chain rule*, we have:



$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa} \frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

where: $h_\kappa = \sum_{j=0}^{M} w_{j\kappa} a_\zeta$

The input to output-layer neuron κ

The equation above says that **the error at the output changes as we vary the second-layer weights** as a function of the <u>error change with respect to the input to the output neurons</u> and the <u>change in the input with respect to the weights</u>.

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (2)nd factor:

Why?

$$\frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}} = \frac{\partial \sum_{j=0}^{M} w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = \sum_{j=0}^{M} \frac{\partial w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = a_{\zeta}$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa} \frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\boxed{\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}} = a_\zeta}$$

(*) Consider the (2)nd factor:

$$\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}} = \frac{\partial \sum_{j=0}^{M} w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = \sum_{j=0}^{M} \frac{\partial w_{j\kappa} a_j}{\partial w_{\varsigma\kappa}} = a_\zeta$$

(*) Last step holds because $\frac{\partial w_{j\kappa}}{\partial w_{\varsigma\kappa}} = 0$, except in the case: $j = \zeta$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa} \frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (1)st factor, which we short-hand as follows:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa}$$

(*) By the chain rule, we have:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa} \frac{\partial y_\kappa}{\partial h_\kappa}$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_{\kappa}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

(*) Consider the (1)st factor, which we short-hand as follows:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}}$$

(*) By the chain rule, we have:

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_{\kappa}} = \frac{\partial E}{\partial y_{\kappa}} \frac{\partial y_{\kappa}}{\partial h_{\kappa}}$$

(*) Also, note that: $y_{\kappa} = g\left(h_{\kappa}^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_{j}^{hidden}\right)$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \boxed{\frac{\partial E}{\partial h_{\kappa}}} \frac{\partial h_{\kappa}}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa} \frac{\partial y_\kappa}{\partial h_\kappa}$$

$$y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} \frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa}$$

$$y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)}\boxed{\left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right]} g'\left(h_\kappa^{output}\right)$$

$$E(w) = \frac{1}{2}\sum_{k=1}^{N}\left(y_k - t_k\right)^2 = \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik} x_i\right) - t_k\right]^2$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\delta_O(\kappa) = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)}\frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right] g'\left(h_\kappa^{output}\right)$$

$$= \left(g\left(h_\kappa^{output}\right) - t_k\right) g'\left(h_\kappa^{output}\right)$$

Backprop Derivation

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \boxed{\frac{\partial E}{\partial h_\kappa}} \frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$\delta_O(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa} \frac{\partial y_\kappa}{\partial h_\kappa} \qquad y_\kappa = g\left(h_\kappa^{output}\right) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{hidden}\right)$$

Continuing…

$$\boxed{\delta_O(\kappa)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} \frac{\partial g\left(h_\kappa^{output}\right)}{\partial\left(h_\kappa^{output}\right)} = \frac{\partial E}{\partial g\left(h_\kappa^{output}\right)} g'\left(h_\kappa^{output}\right)$$

$$= \frac{\partial}{\partial g\left(h_\kappa^{output}\right)} \left[\frac{1}{2}\sum_{k=1}^{N}\left(g\left(h_\kappa^{output}\right) - t_k\right)^2\right] g'\left(h_\kappa^{output}\right)$$

$$= \left(g\left(h_\kappa^{output}\right) - t_k\right) g'\left(h_\kappa^{output}\right) = \boxed{\left(y_k - t_k\right) g'\left(h_\kappa^{output}\right)}$$

Backprop Derivation

In Summary…

$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$= \boxed{\left(y_k - t_k\right) g'\left(h_\kappa^{output}\right)} \boxed{a_\zeta}$$

Backprop Derivation

In Summary…

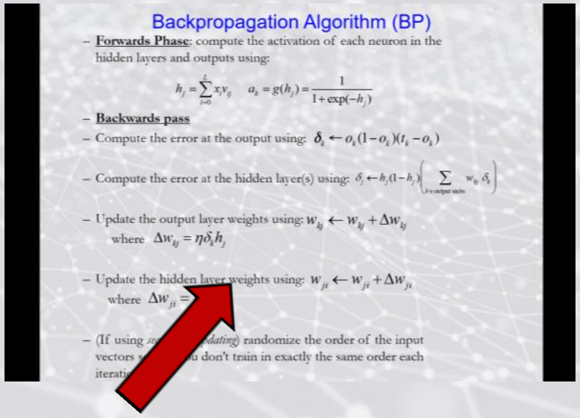
$$\frac{\partial E}{\partial w_{\varsigma\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\varsigma\kappa}}$$

$$= \boxed{\left(y_k - t_k\right) g'\left(h_\kappa^{output}\right)}\boxed{a_\zeta}$$

(*) Recall, a "gradient descent" based weight update has the form:

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta\left(y_k - t_k\right) g'\left(h_\kappa^{output}\right) a_\zeta$$



Backpropagation Algorithm (BP)

– **Forwards Phase**: compute the activation of each neuron in the hidden layers and outputs using:

$$h_j = \sum_{i=0}^{L} x_i v_{ij} \quad a_k = g(h_j) = \frac{1}{1+\exp(-h_j)}$$

– **Backwards pass**

– Compute the error at the output using: $\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$

– Compute the error at the hidden layer(s) using: $\delta_j \leftarrow h_j(1-h_j)\left(\sum_{k\in\text{output units}} w_{kj}\delta_k\right)$

– Update the output layer weights using: $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$
where $\Delta w_{kj} = \eta \delta_k h_j$

– Update the hidden layer weights using: $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
where $\Delta w_{ji} =$

– (If using se [illegible] dating) randomize the order of the input vectors s [illegible] u don't train in exactly the same order each iterati [illegible]

(*) Cool, but this doesn't look like the formulas for BP you showed us before.

Backprop Derivation

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) g'\left(h_\kappa^{output} \right) a_\zeta$$

(*) Recall that g is the sigmoid! So what's our new formula?

Backprop Derivation

$$w_{\varsigma\kappa} \leftarrow w_{\varsigma\kappa} - \eta \frac{\partial E}{\partial w_{\varsigma\kappa}}$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) g'\left(h_\kappa^{output} \right) a_\zeta$$

$$= w_{\varsigma\kappa} - \eta \left(y_k - t_k \right) y_k \left(1 - y_k \right) a_\zeta$$

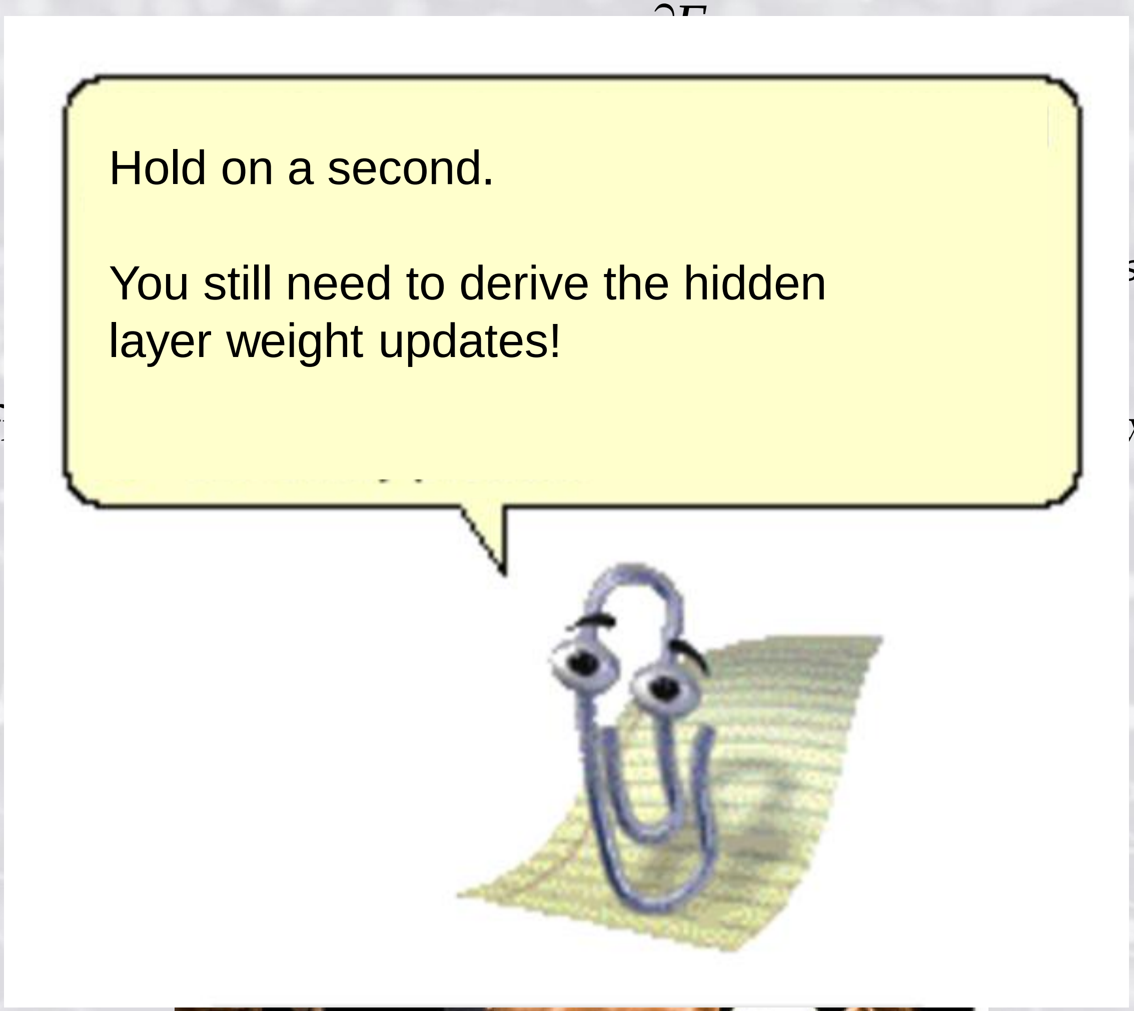
$$\frac{ds(z)}{dz} = s(z) \times (1 - s(z))$$

(*) This is the final formula for the BP update for the output layer weights!

Backprop Derivation

$s(z)\times(1 - s(z))$

(*) This is the f weights!



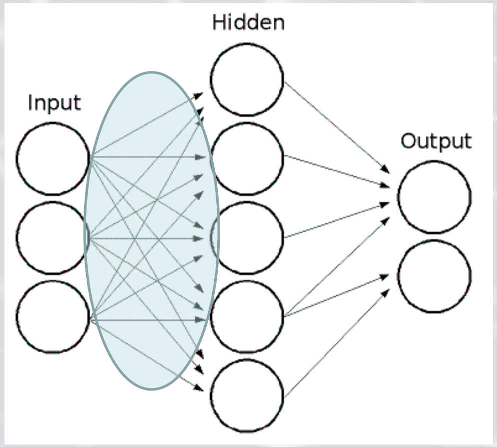
Backprop Derivation

(*)*Short version* of **input-to-hidden layer weight updates** for BP:

We compute: $\delta_h(\zeta) = \frac{\partial E}{\partial h_\zeta^{hidden}} = \sum_{k=1}^{N} \frac{\partial E}{\partial h_k^{output}} \frac{\partial h_k^{output}}{\partial h_\zeta^{hidden}} = \sum_{k=1}^{N} \delta_O(k) \frac{\partial h_k^{output}}{\partial h_\zeta^{hidden}}$

(*) This formula comes from the fact that each hidden node contributes to the activation of all the output nodes, and so we need to consider all of these contributions.

(*) From here, using the chain rule, differential properties of the sigmoid and the NN topology, it is not difficult (as we did before, analogously), to show:



$$\delta_h(\zeta) = a_\zeta(1 - a_\zeta) \sum_{k=1}^{N} \delta_O(\kappa) w_\zeta$$

Backprop Derivation

$$\delta_h(\zeta) = a_\zeta(1-a_\zeta)\sum_{k=1}^{N}\delta_O(\kappa)w_\zeta$$

(*)This yields the following update rule for vertex v_ι:

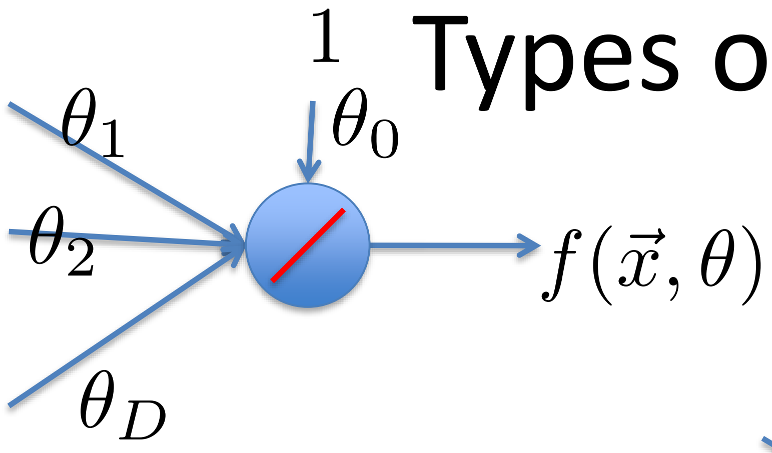
$$v_\iota \leftarrow v_\iota - \eta\frac{\partial E}{\partial v_\iota}$$

$$= v_\iota - \eta a_\zeta(1-a_\zeta)\left(\sum_{k=1}^{N}\delta_O(\kappa)w_\zeta\right)x_\iota$$

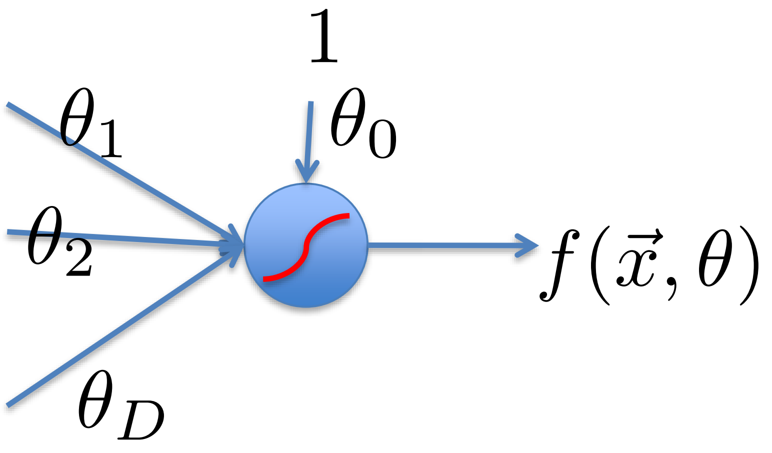
Derivation complete! (at least for NNs with one hidden layer)

Backprop Derivation Redux (with visuals)

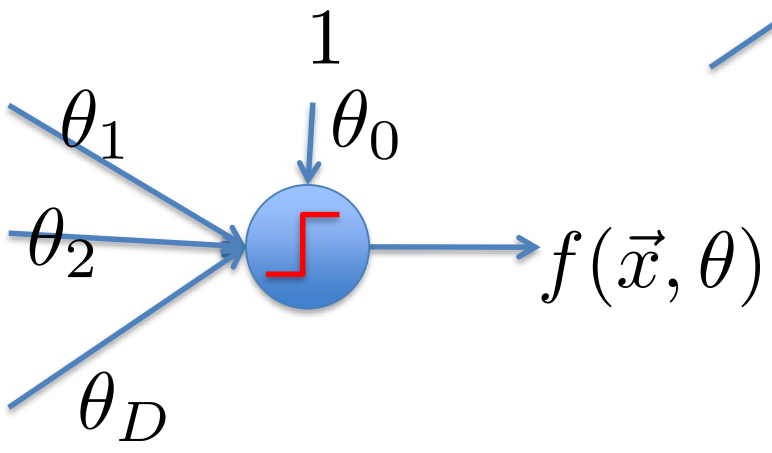
Types of Neurons



Linear Neuron



Logistic Neuron

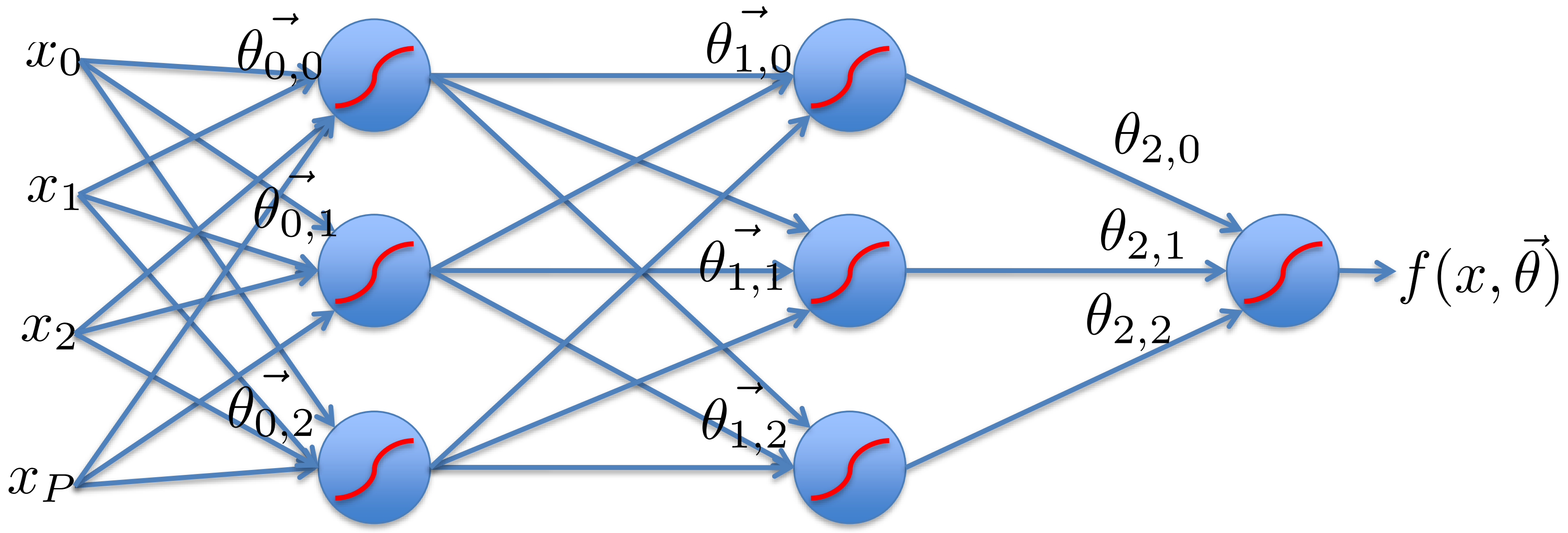


Perceptron

Potentially more. Require a convex loss function for gradient descent training.

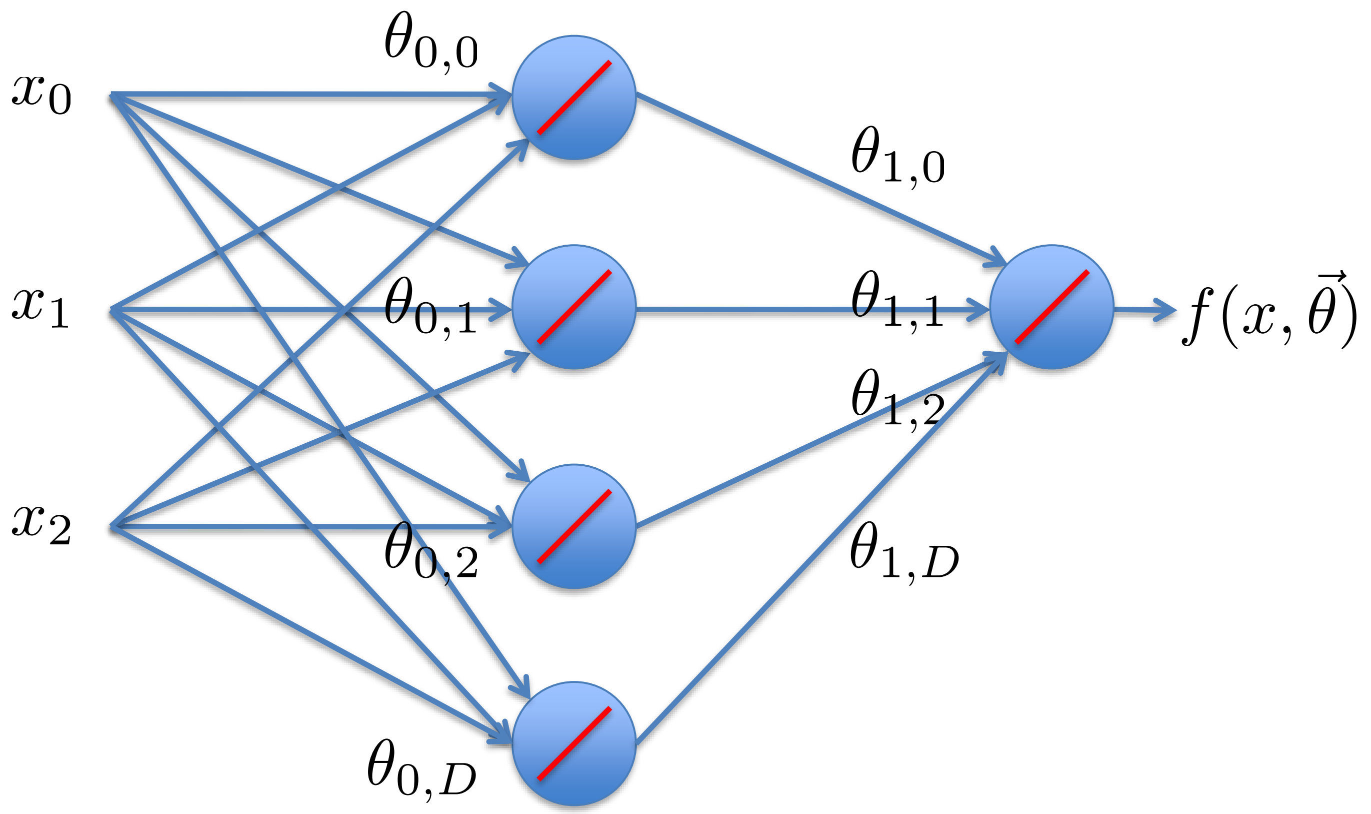
Multilayer Networks

- Cascade Neurons together
- The output from one layer is the input to the next
- Each Layer has its own sets of weights



Linear Regression Neural Networks

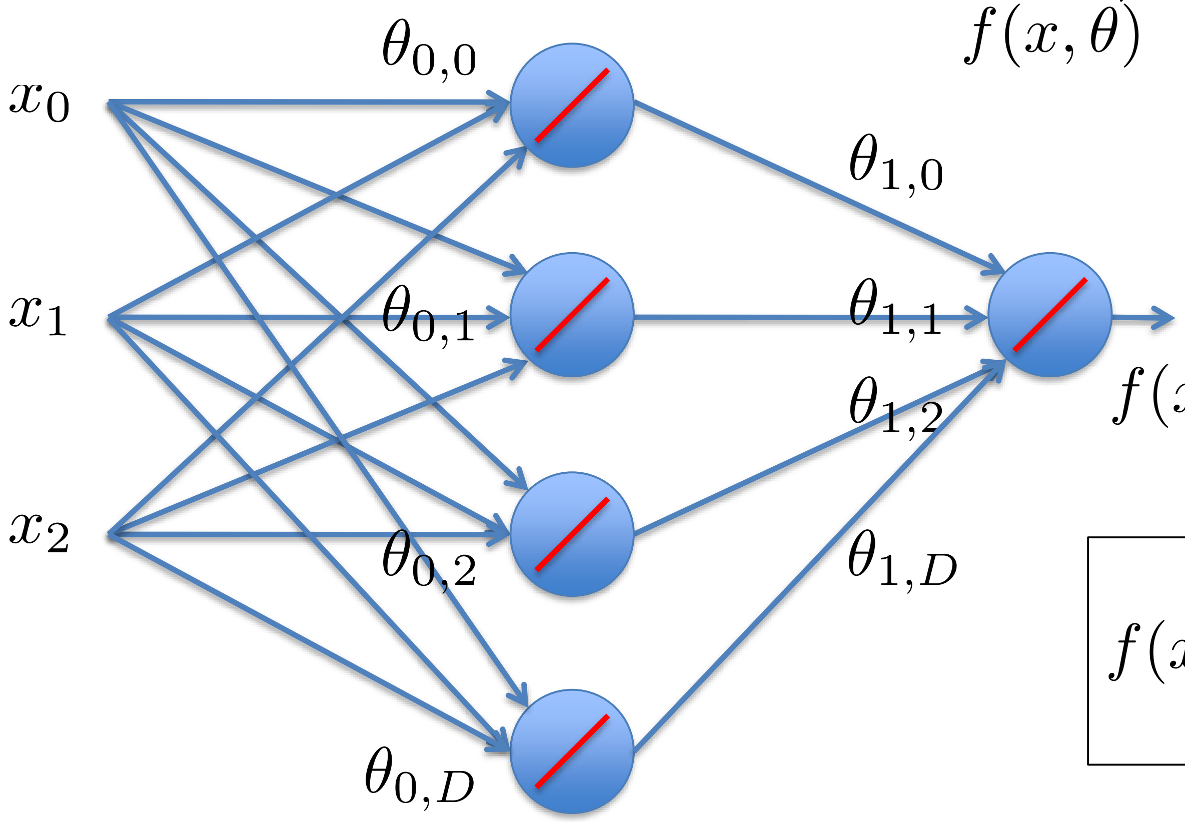
- What happens when we arrange **linear neurons** in a multilayer network?



Linear Regression Neural Networks

- Nothing special happens.
  - The product of two linear transformations is itself a linear transformation.

$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i} \sum_{n=0}^{N-1} \theta_{0,i,n} x_n$$

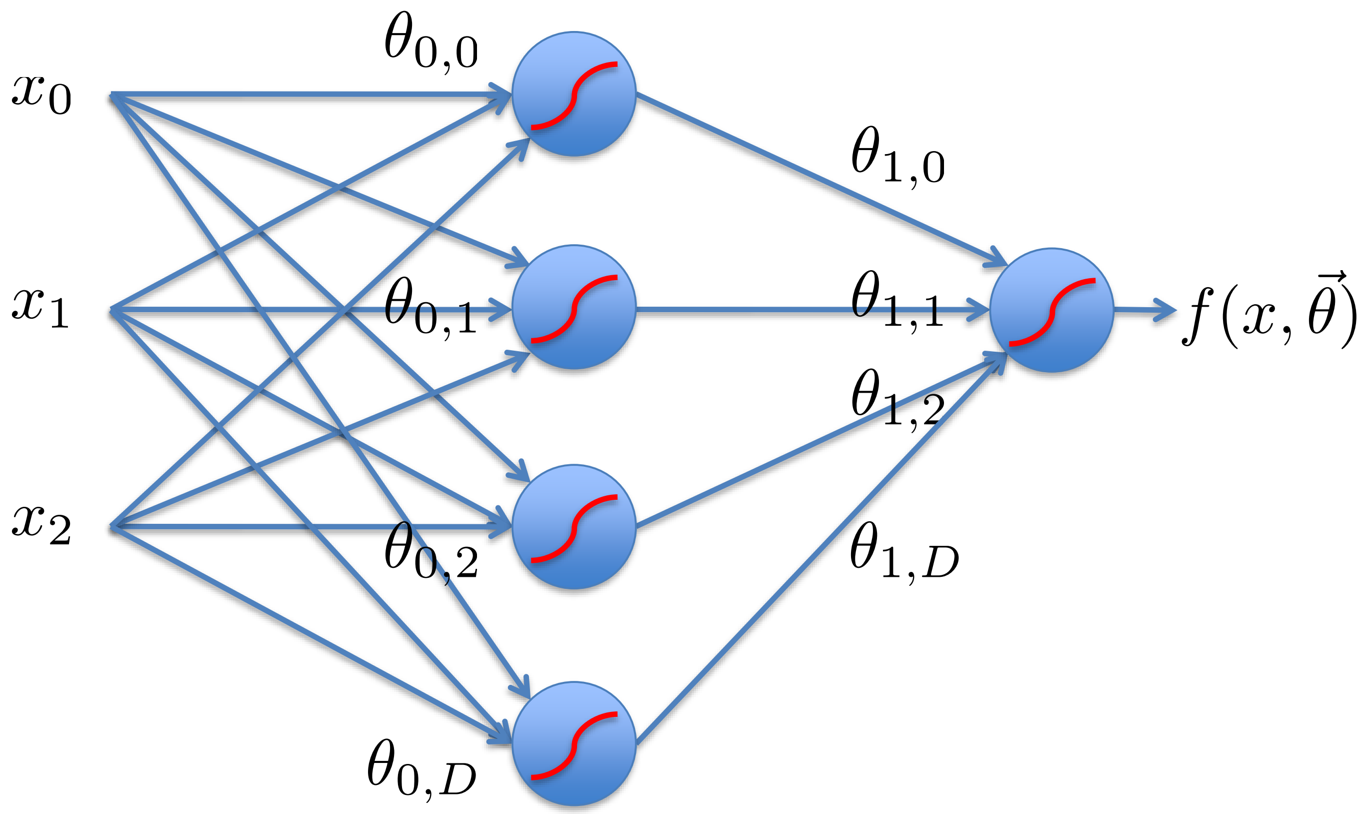


$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i}[\theta_{0,i}^T \vec{x}]$$

$$\boxed{f(x,\vec{\theta}) = \sum_{i=0}^{D} [\hat{\theta}_i^T \vec{x}]}$$

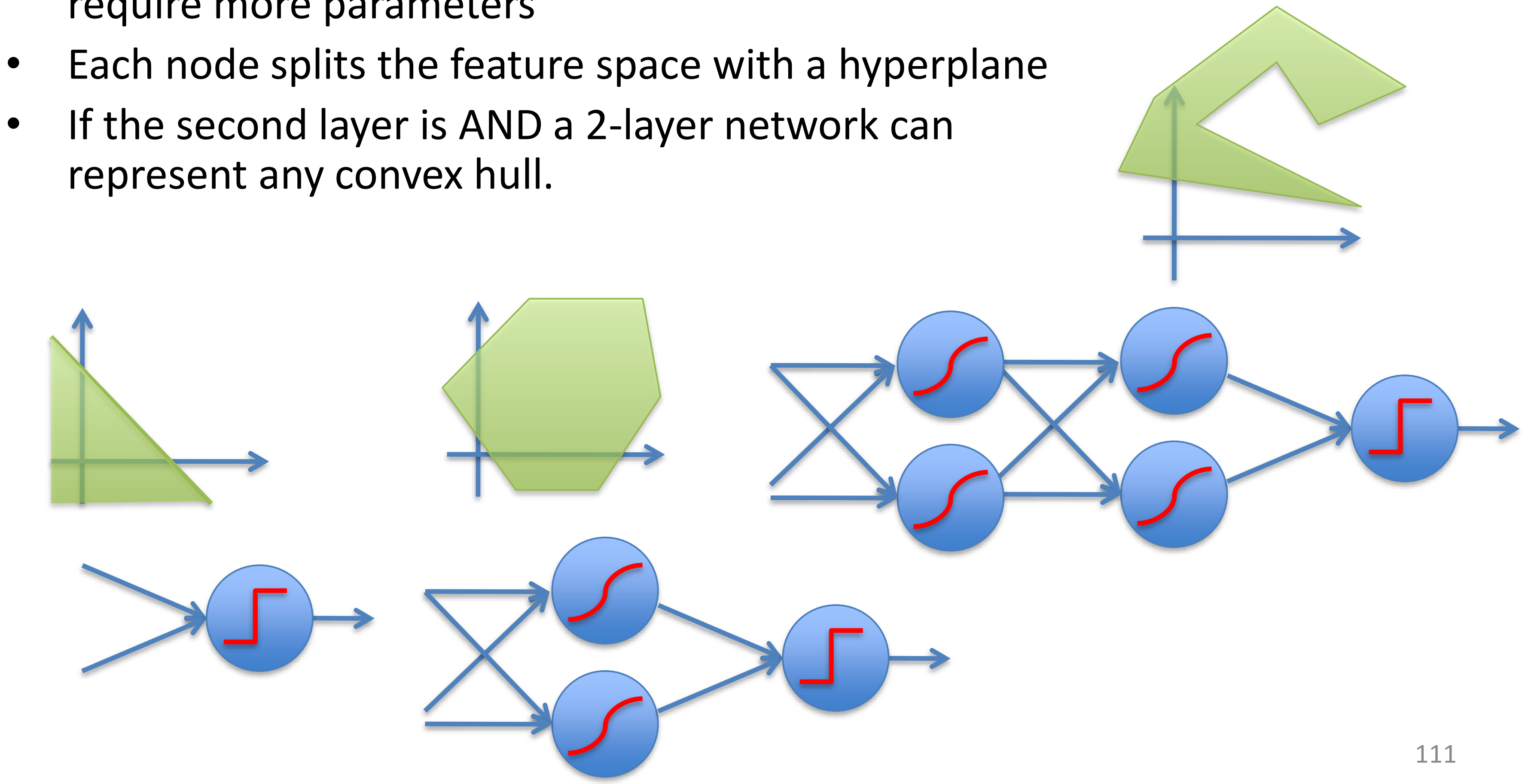
Neural Networks

- We want to introduce non-linearities to the network.
  - Non-linearities allow a network to identify complex regions in space



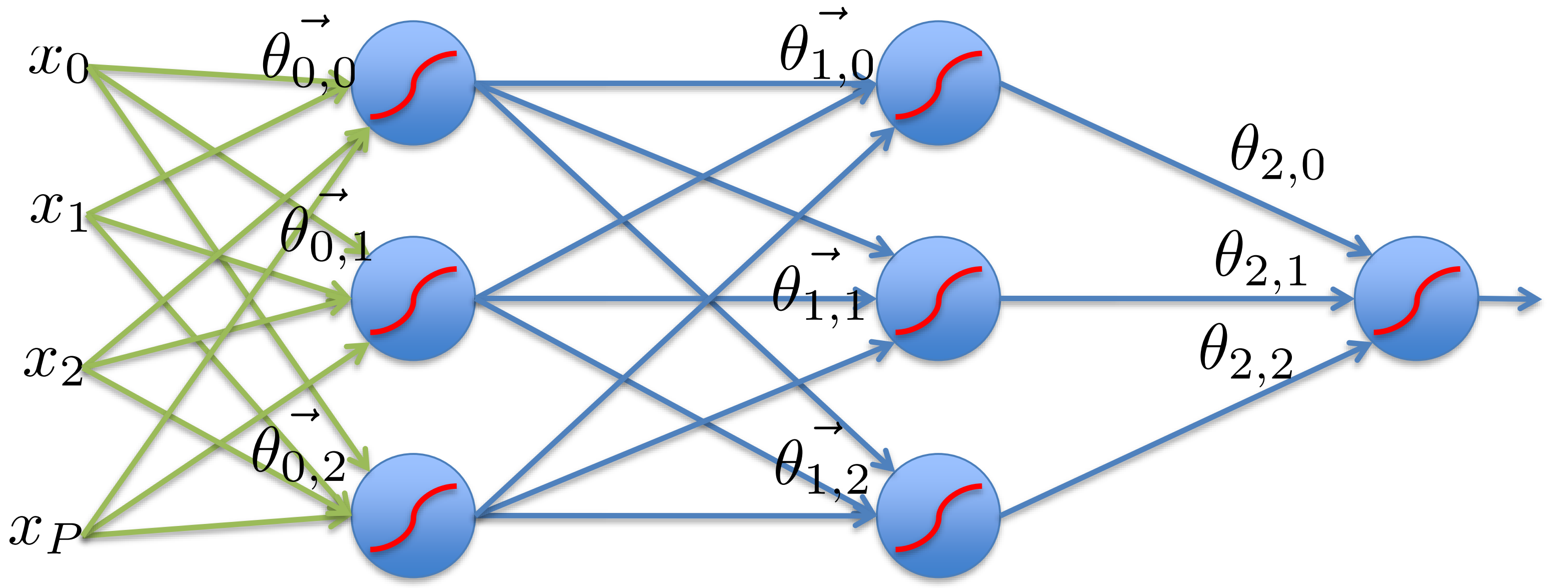
Linear Separability

- 1-layer cannot handle XOR
- More layers can handle more complicated spaces – but require more parameters
- Each node splits the feature space with a hyperplane
- If the second layer is AND a 2-layer network can represent any convex hull.



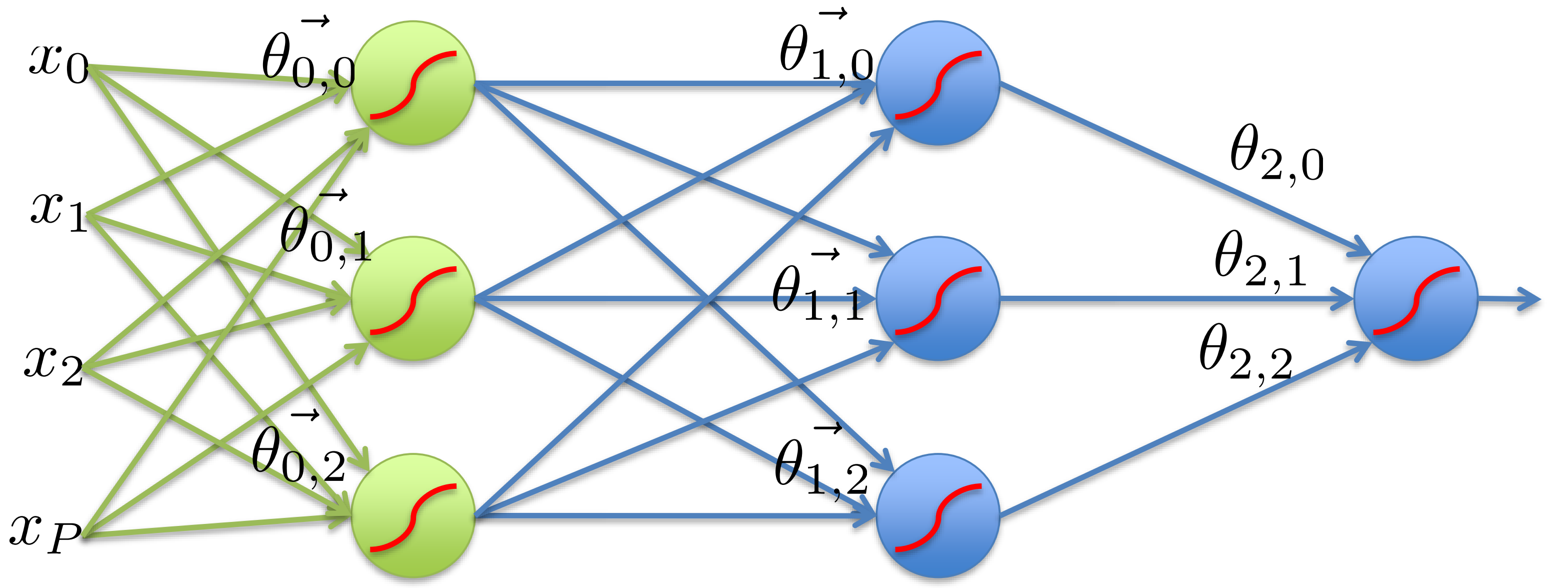
Feed-Forward Networks

- Predictions are fed forward through the network to classify



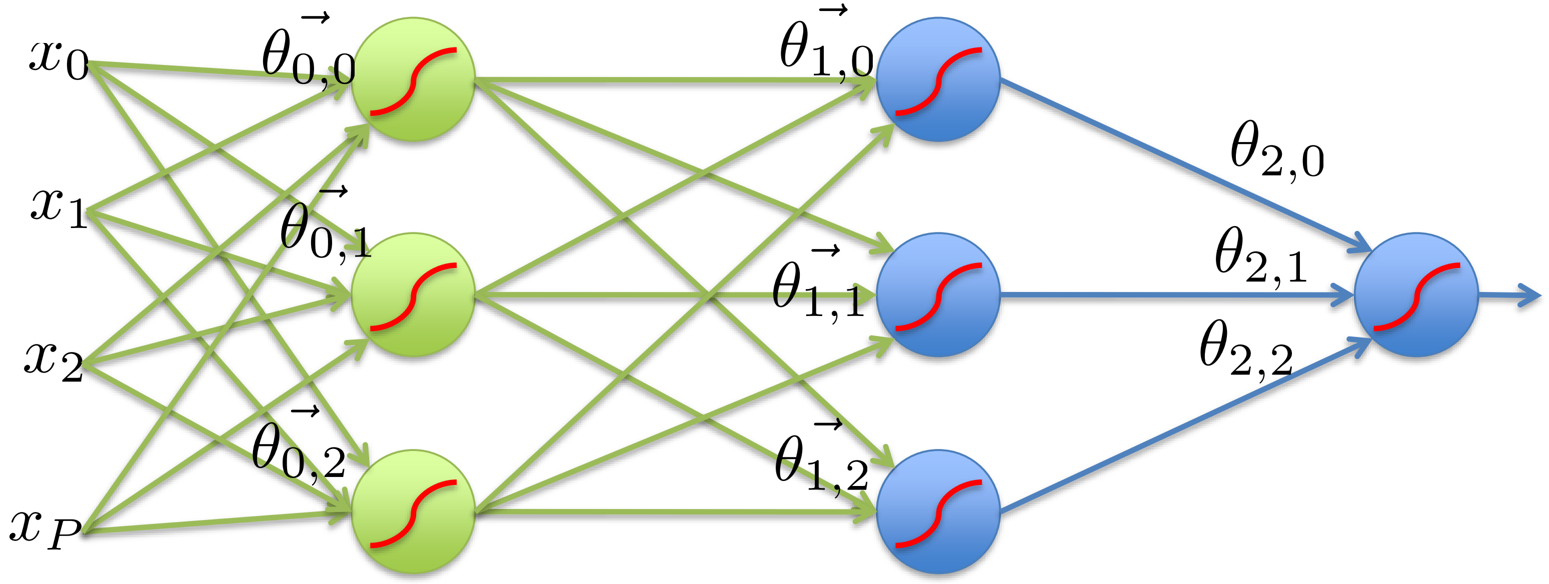
Feed-Forward Networks

- Predictions are fed forward through the network to classify



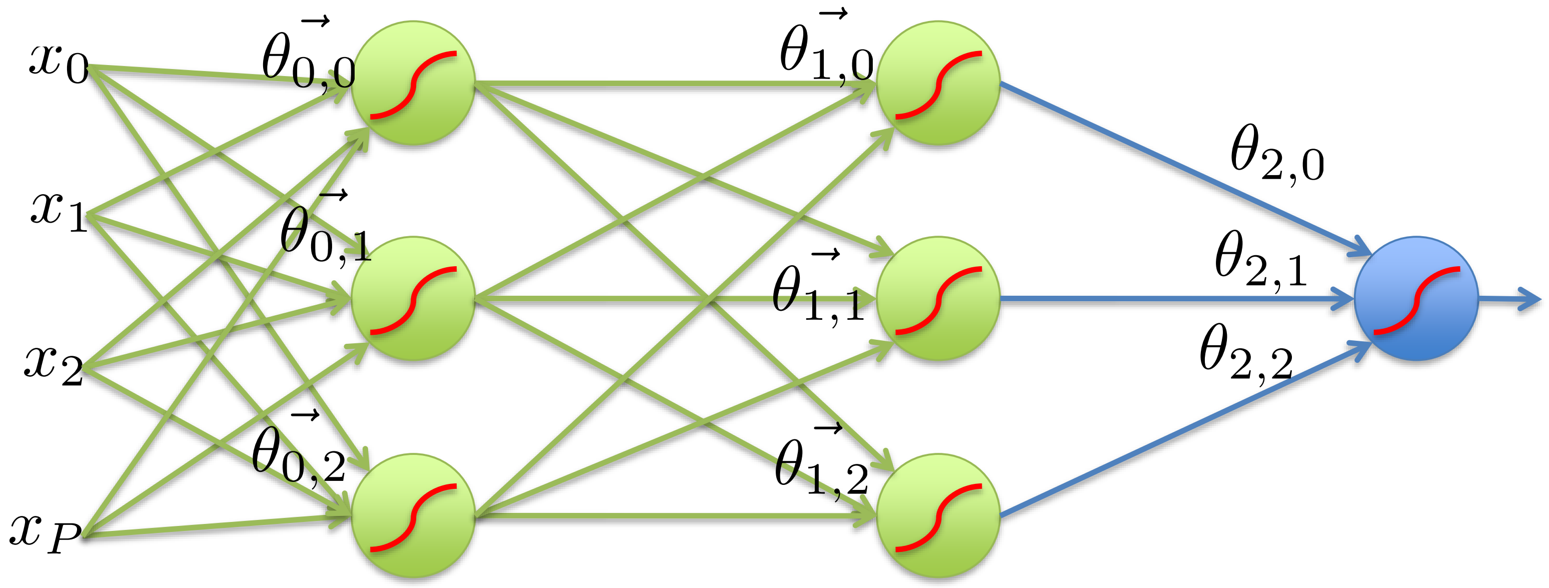
Feed-Forward Networks

- Predictions are fed forward through the network to classify



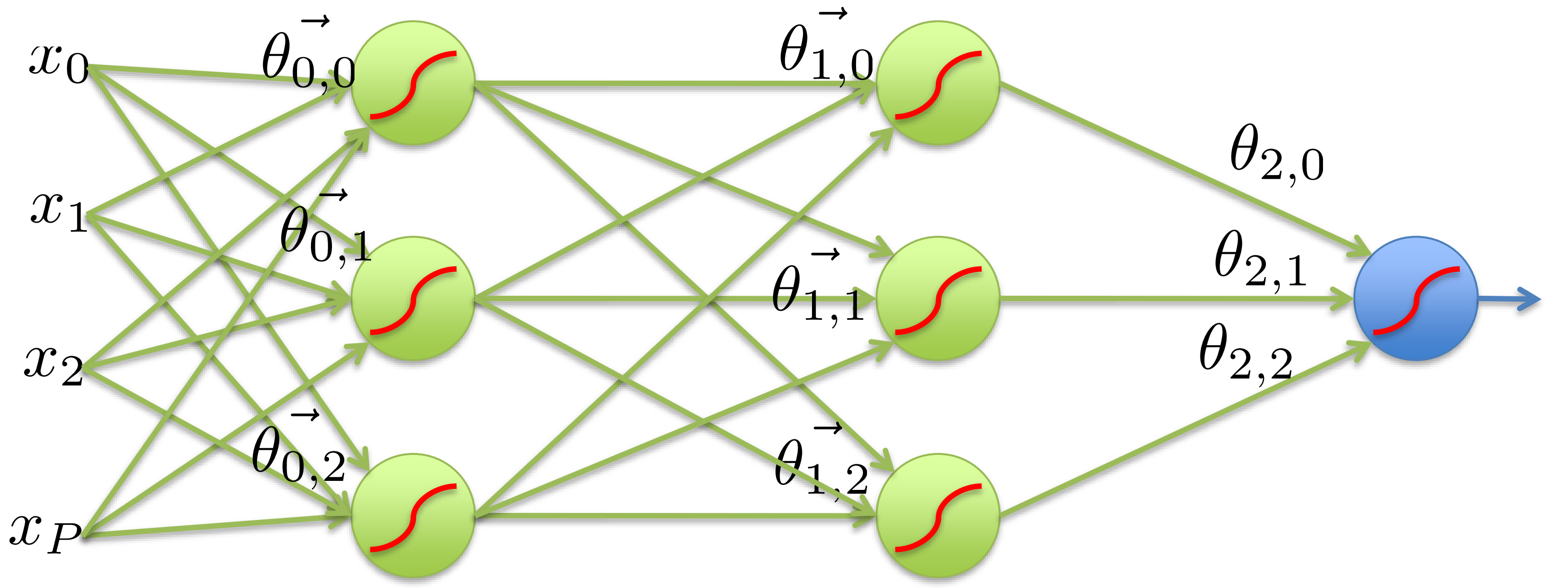
Feed-Forward Networks

- Predictions are fed forward through the network to classify



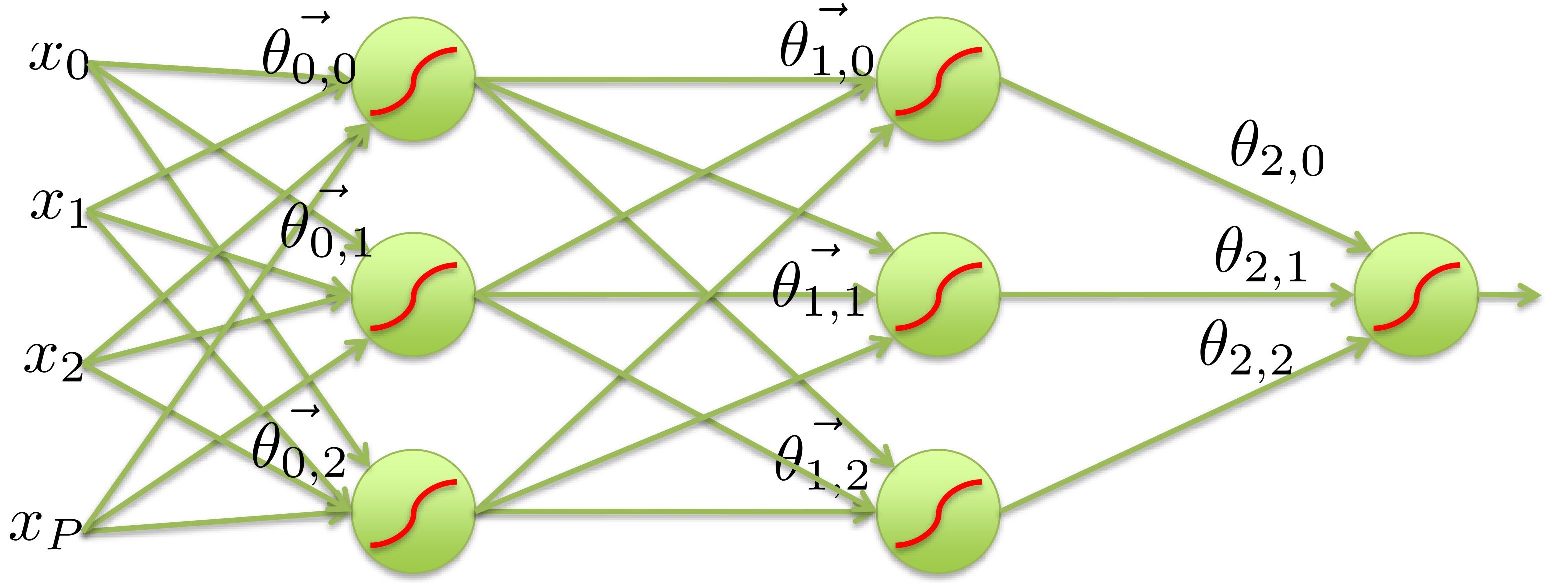
Feed-Forward Networks

- Predictions are fed forward through the network to classify



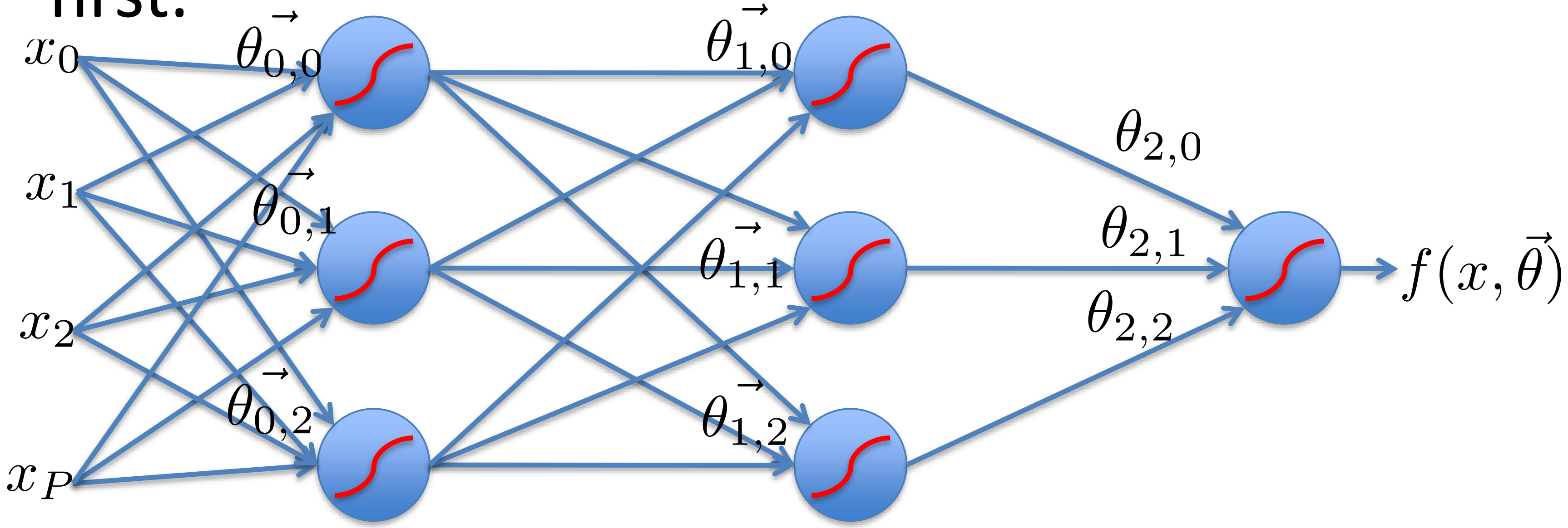
Feed-Forward Networks

- Predictions are fed forward through the network to classify



Error Backpropagation

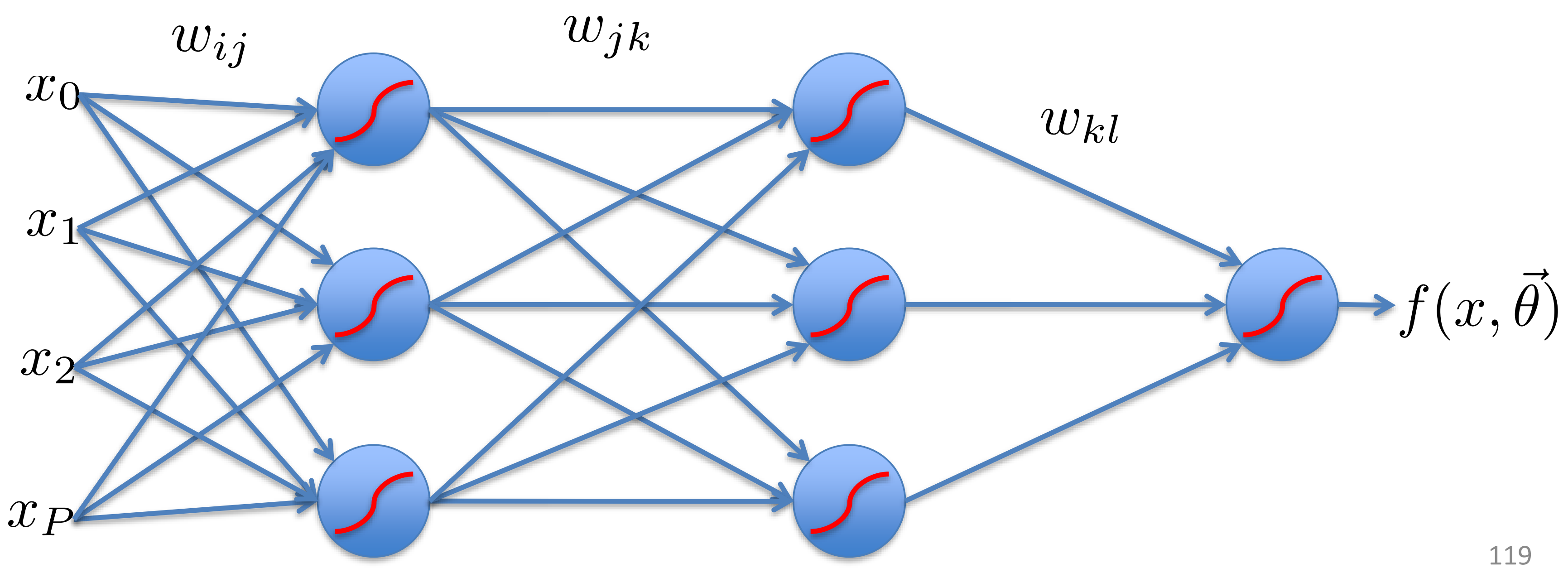
- We will do gradient descent on the whole network.
- Training will proceed from the last layer to the first.



Error Backpropagation

- Introduce variables over the neural network

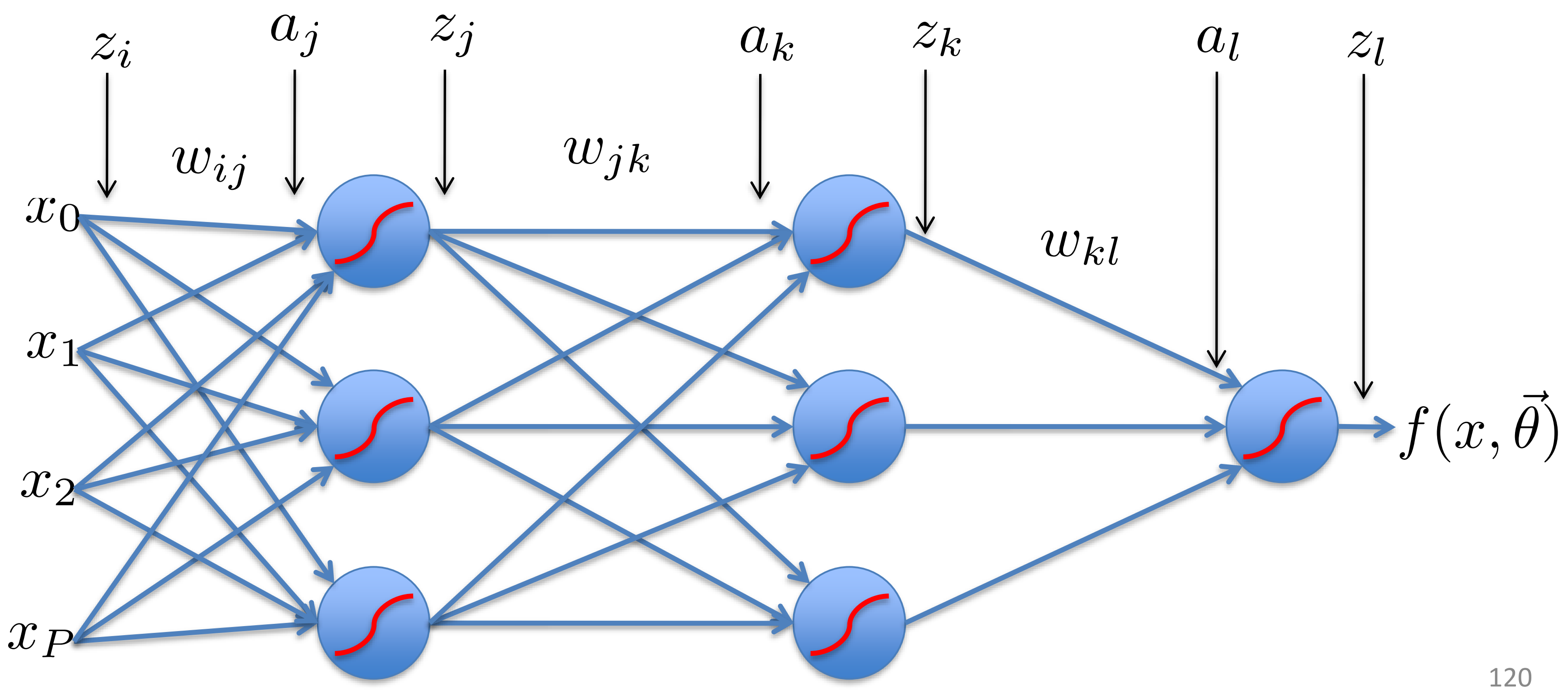
$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

- Introduce variables over the neural network
  - Distinguish the input and output of each node

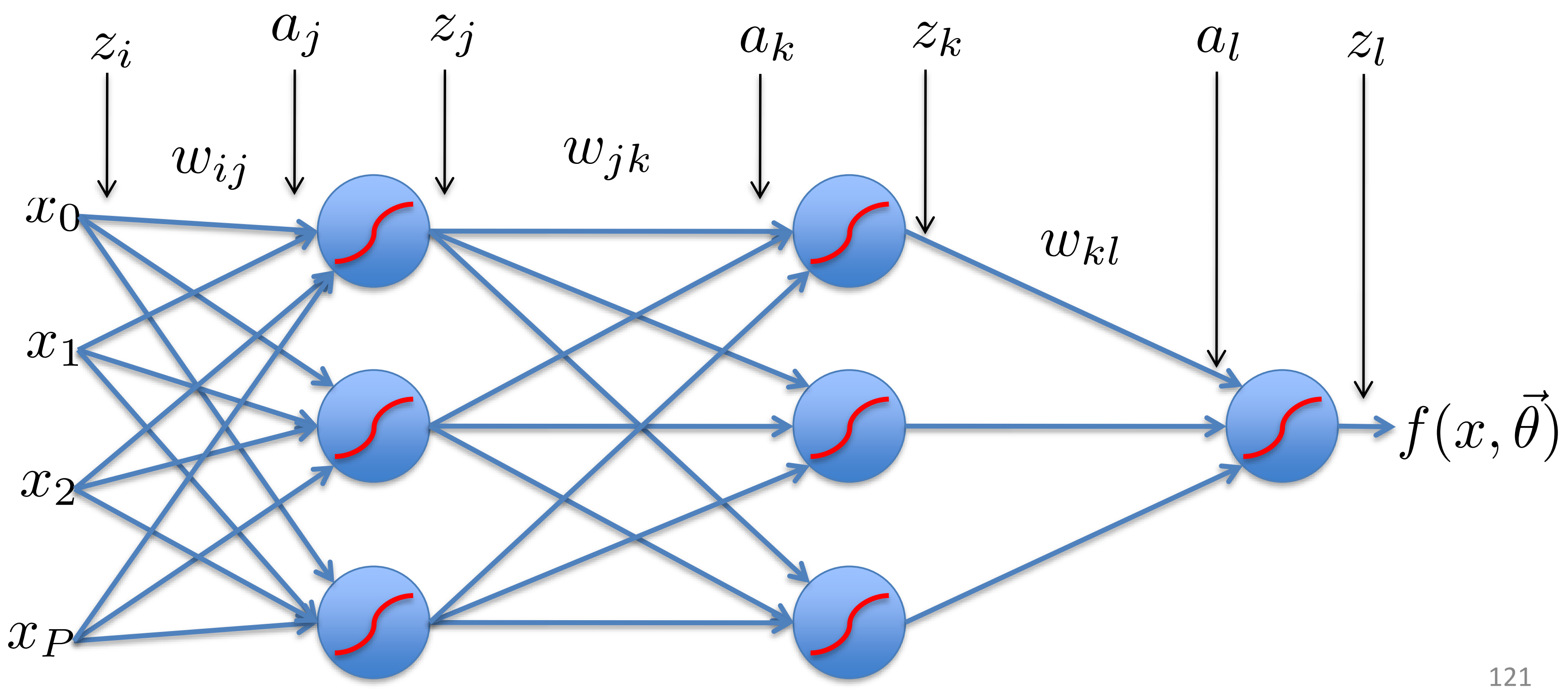


Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



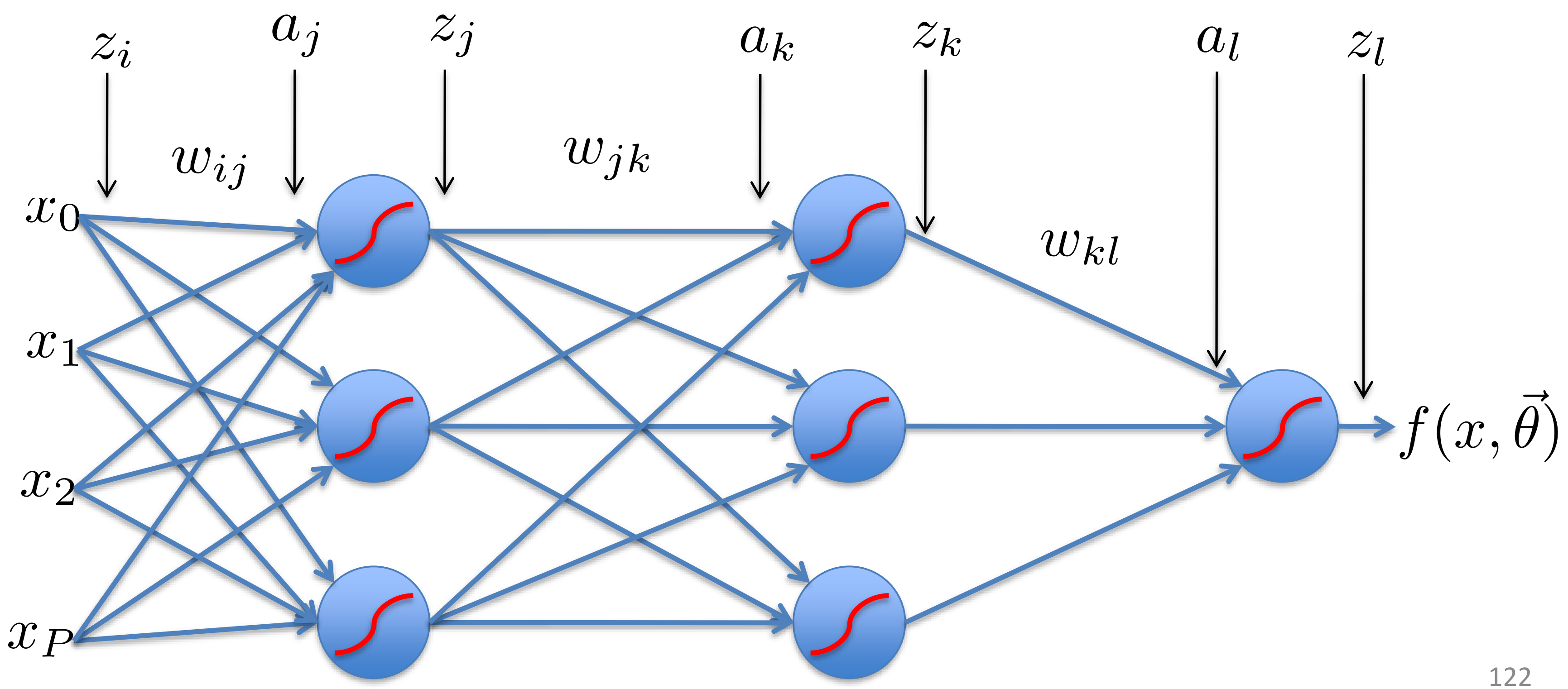
Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Training: Take the gradient of the last component and iterate backwards

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

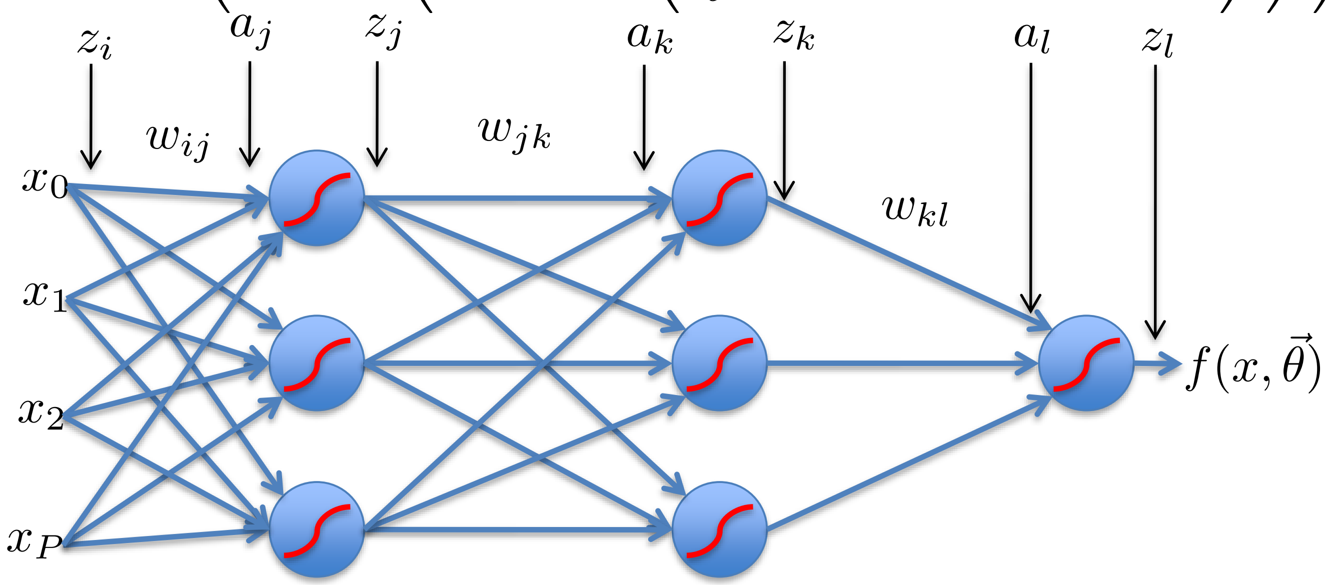
$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Error Backpropagation

$$\begin{aligned} R(\theta) &= \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n)) \\ &= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}(y_n - f(x_n))^2 \\ &= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl} g\left(\sum_j w_{jk} g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2 \end{aligned}$$

Empirical Risk Function



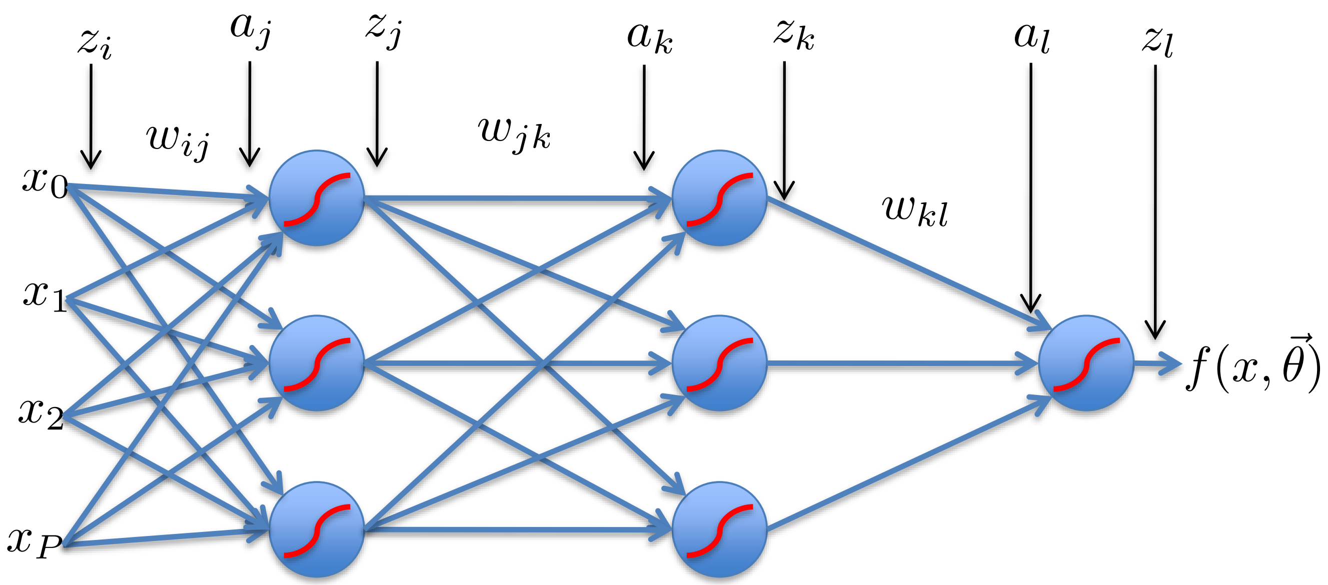
Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

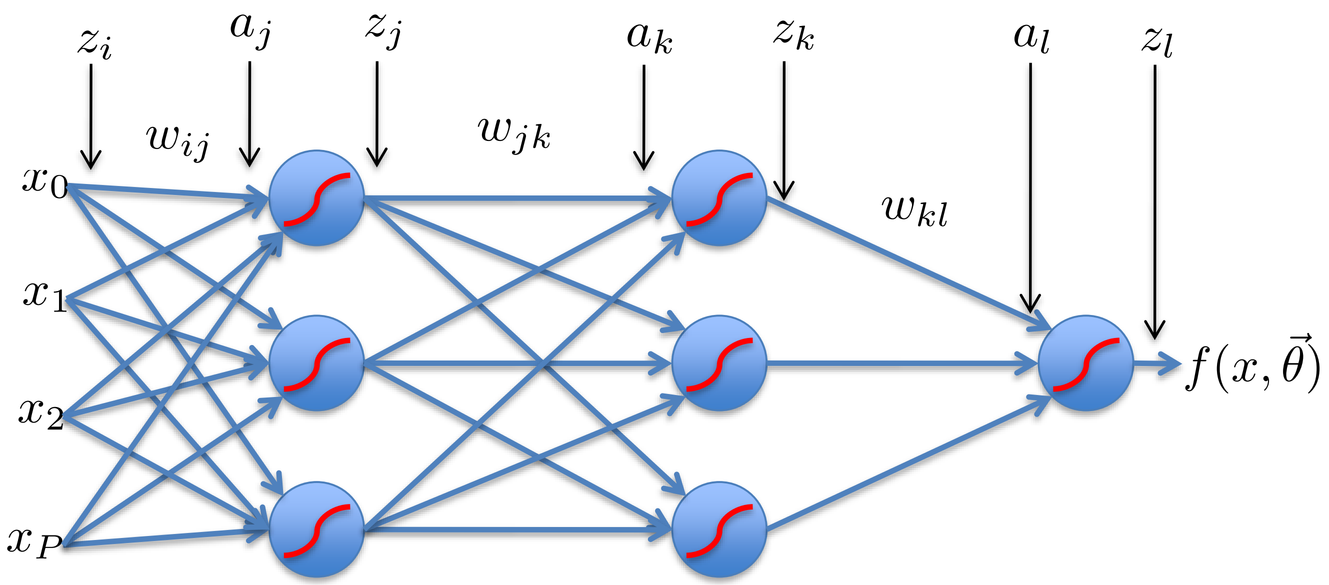
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}\left(y_n - g(a_{l,n})\right)^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$



Error Backpropagation

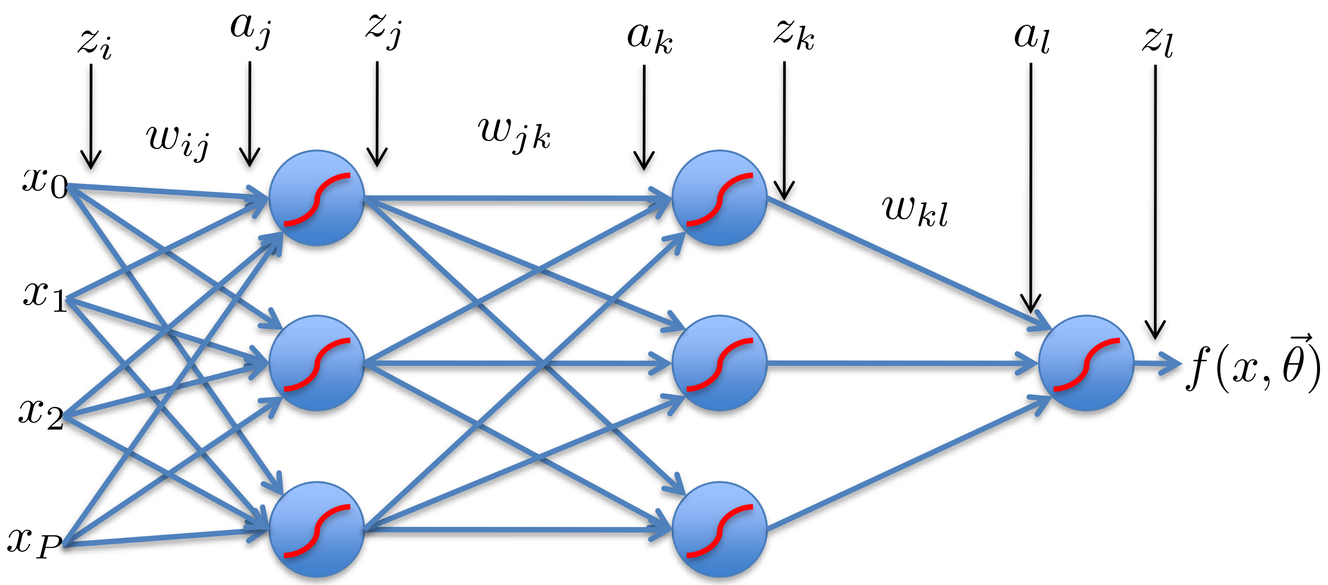
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$



Error Backpropagation

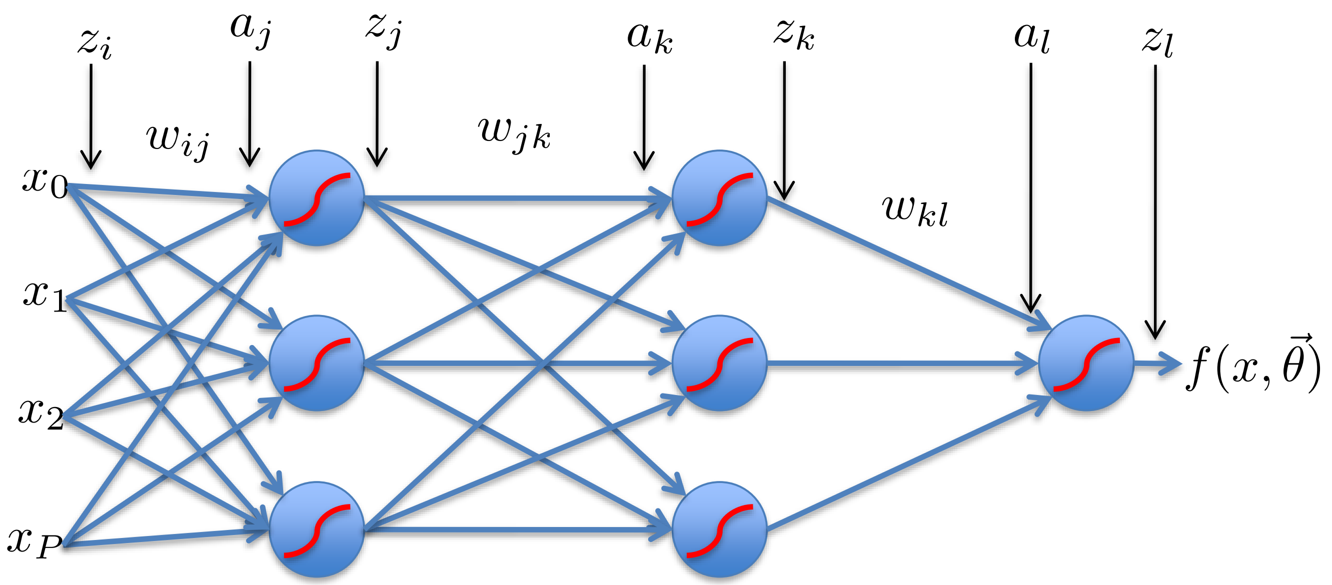
Optimize last layer weights w$_{kl}$

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

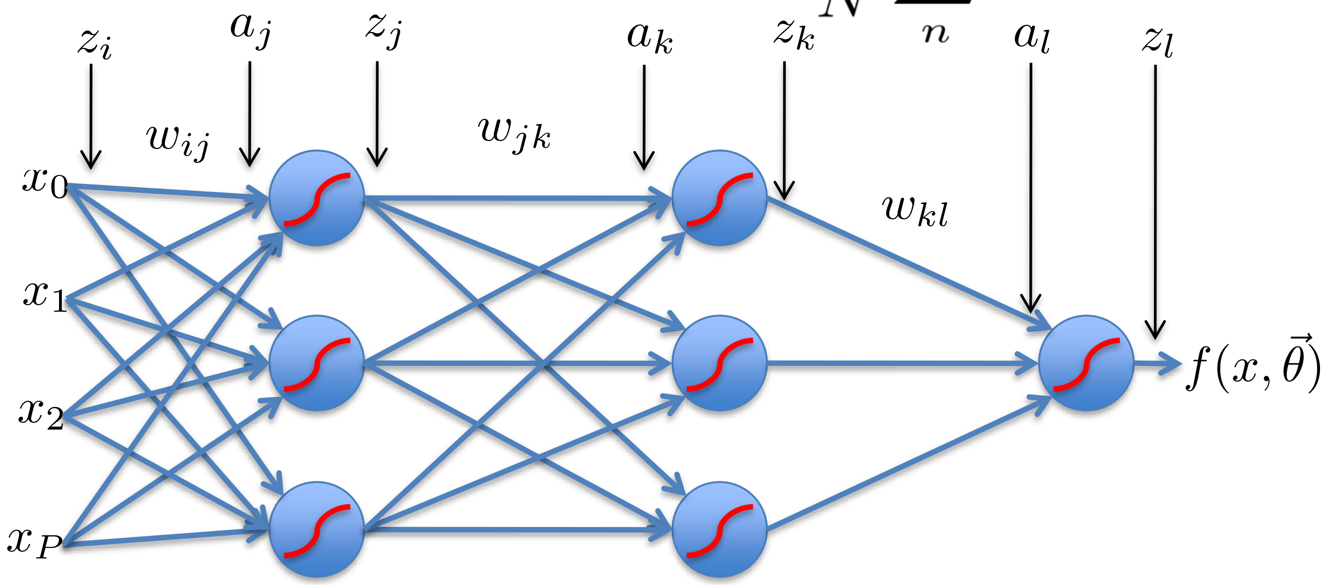
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \boxed{\left[-(y_n - z_{l,n})g'(a_{l,n})\right]} z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

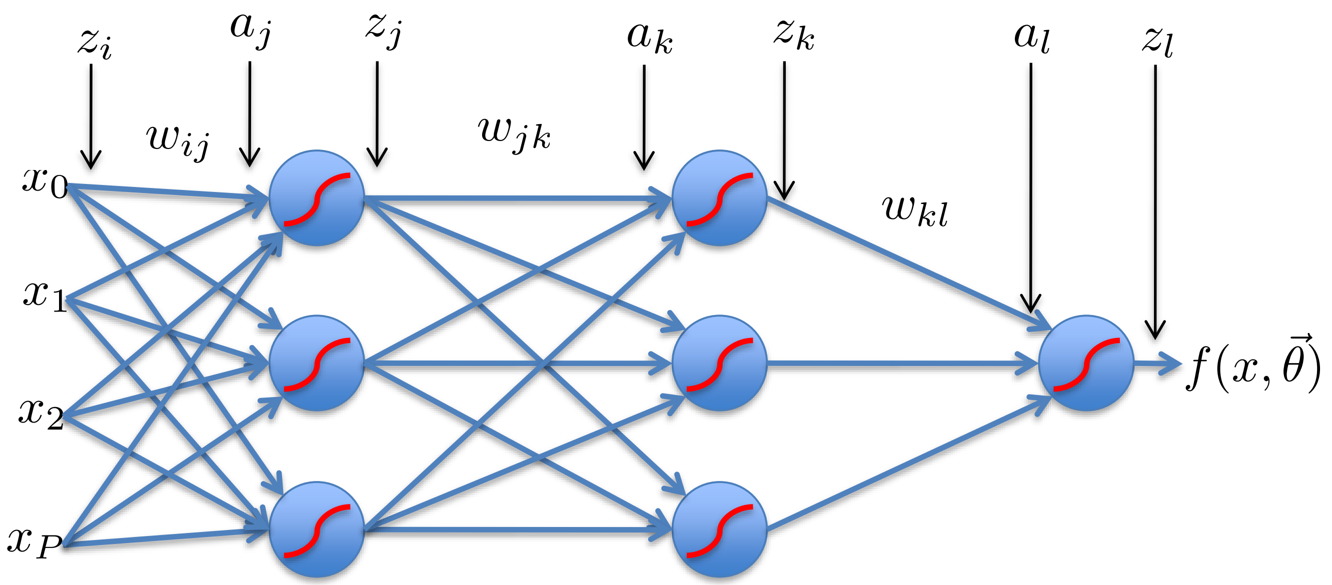


Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$



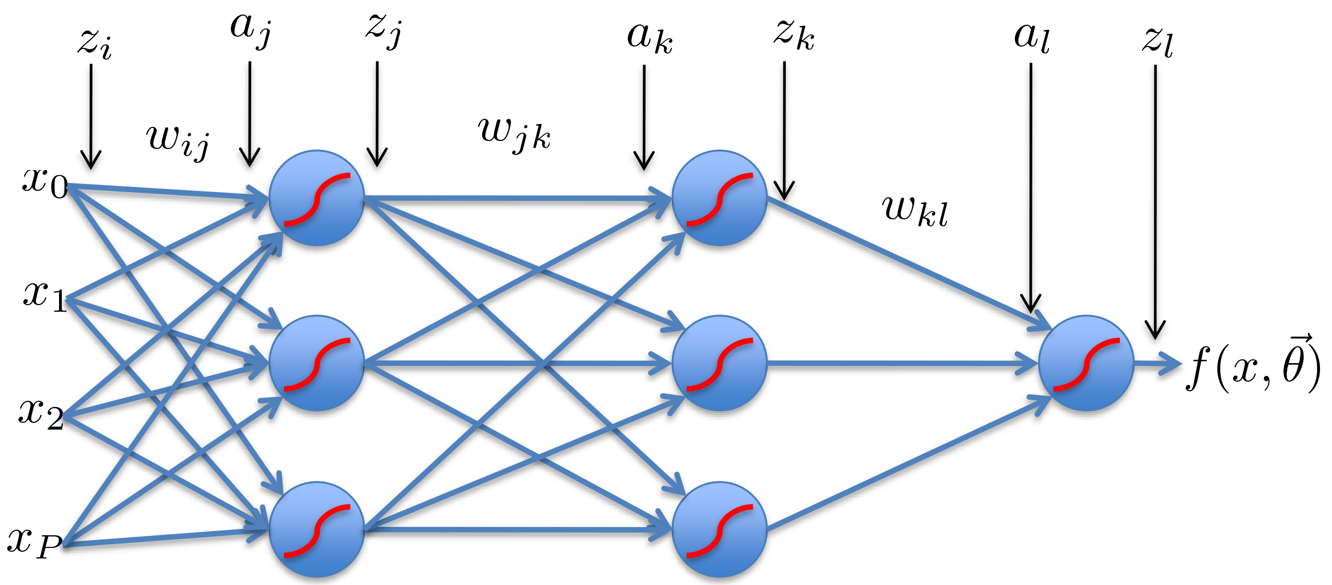
Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule



Error Backpropagation

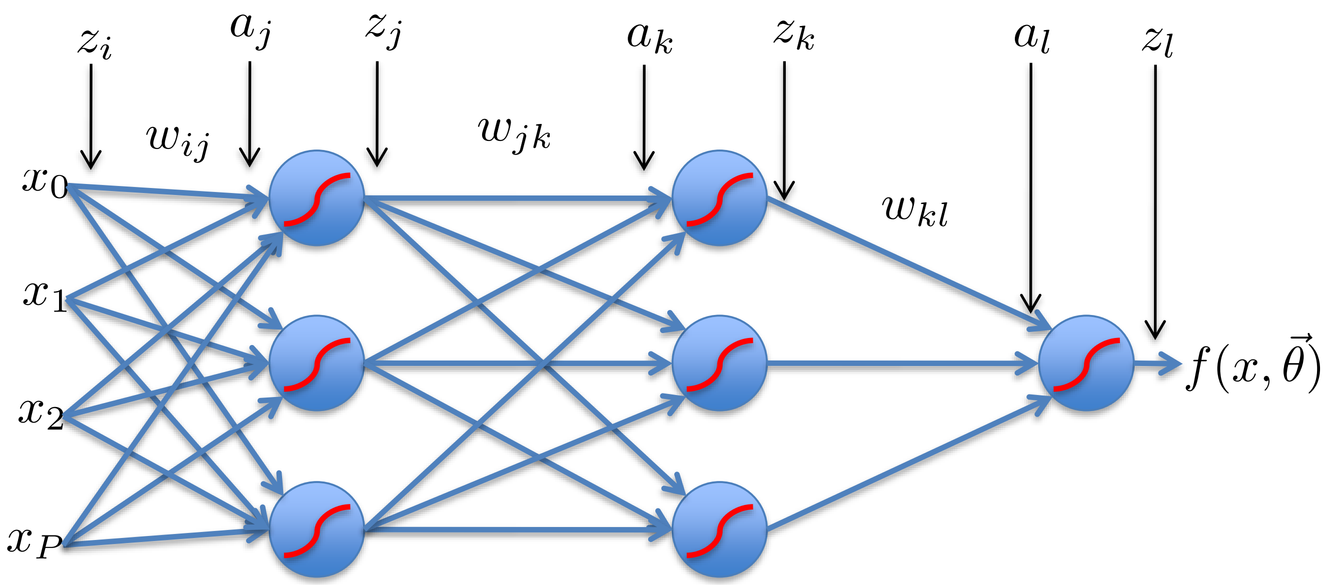
Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] [z_{j,n}]$$



Error Backpropagation

Optimize last hidden weights w_{jk}

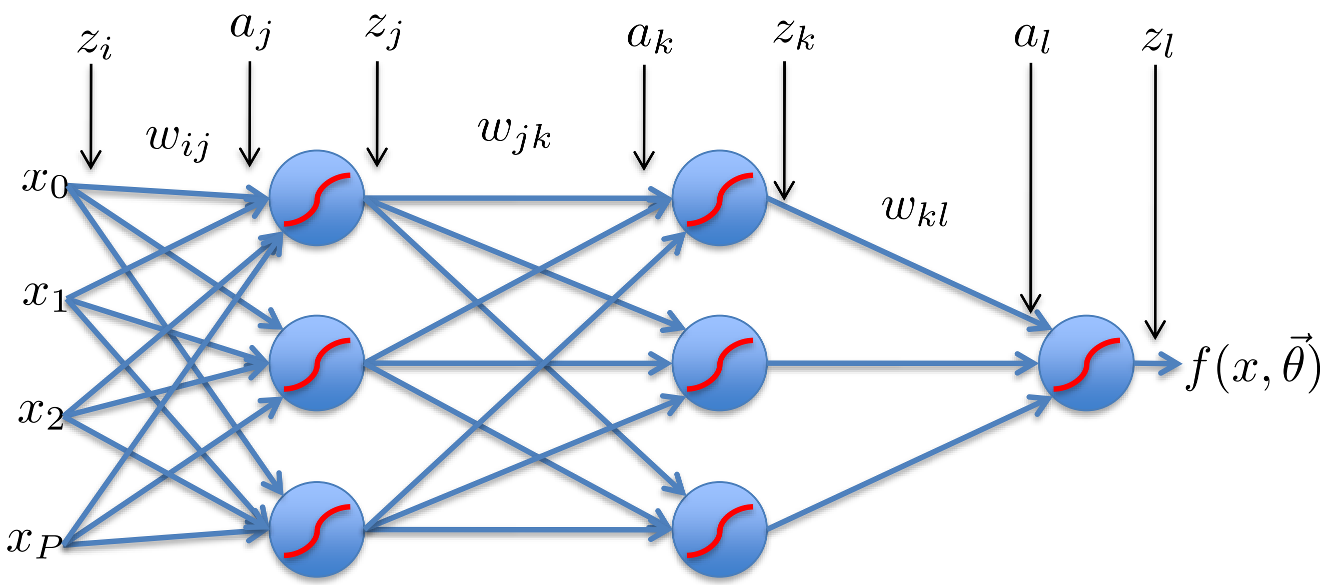
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right][z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$



Error Backpropagation

Optimize last hidden weights w_{jk}

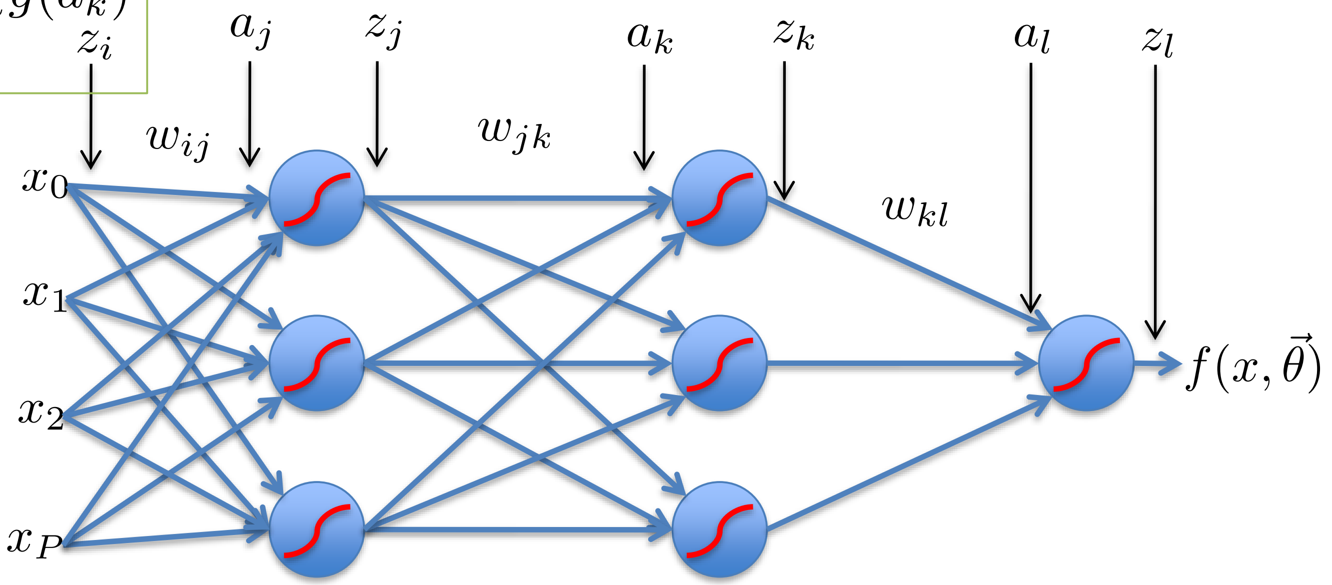
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\boxed{\sum_l \delta_l w_{kl} g'(a_{k,n})}\right][z_{j,n}] = \frac{1}{N}\sum_n [\delta_{k,n}][z_{j,n}]$$

$$\boxed{a_l = \sum_k w_{kl} g(a_k)}$$



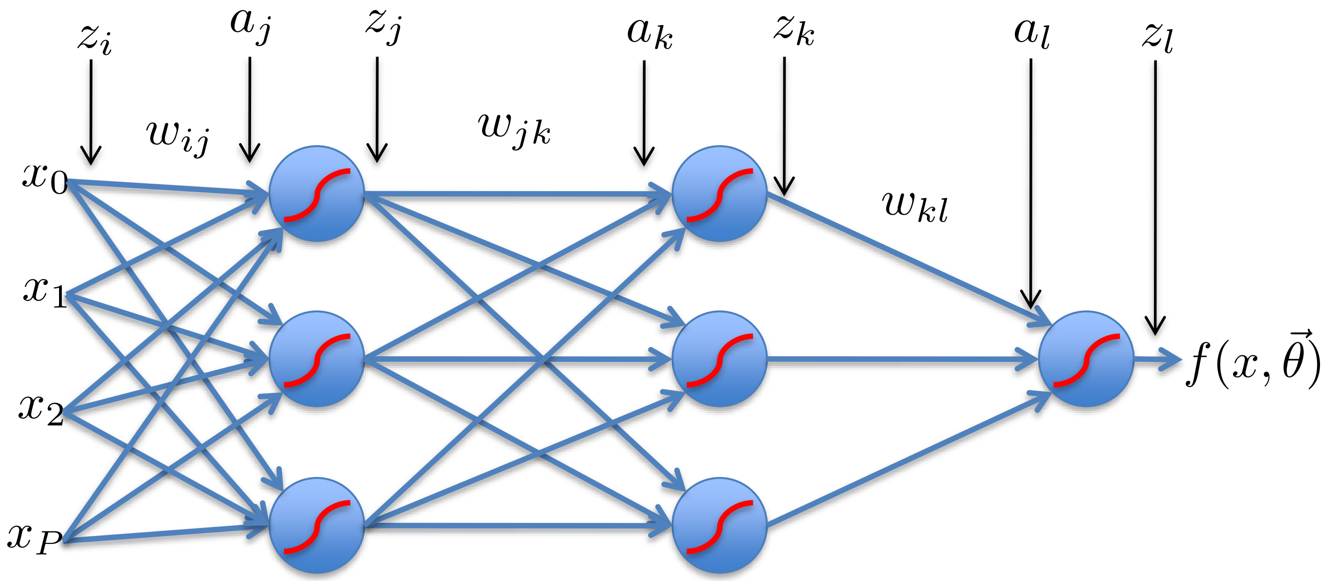
Error Backpropagation

Repeat for all previous layers

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

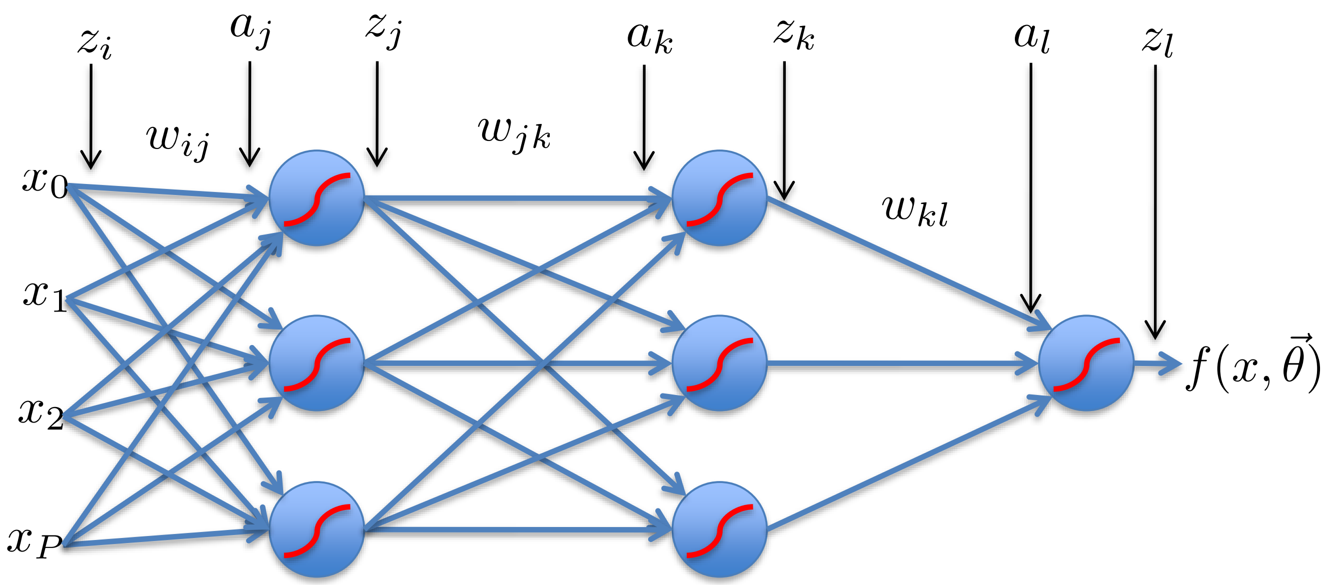
$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Error Backpropagation

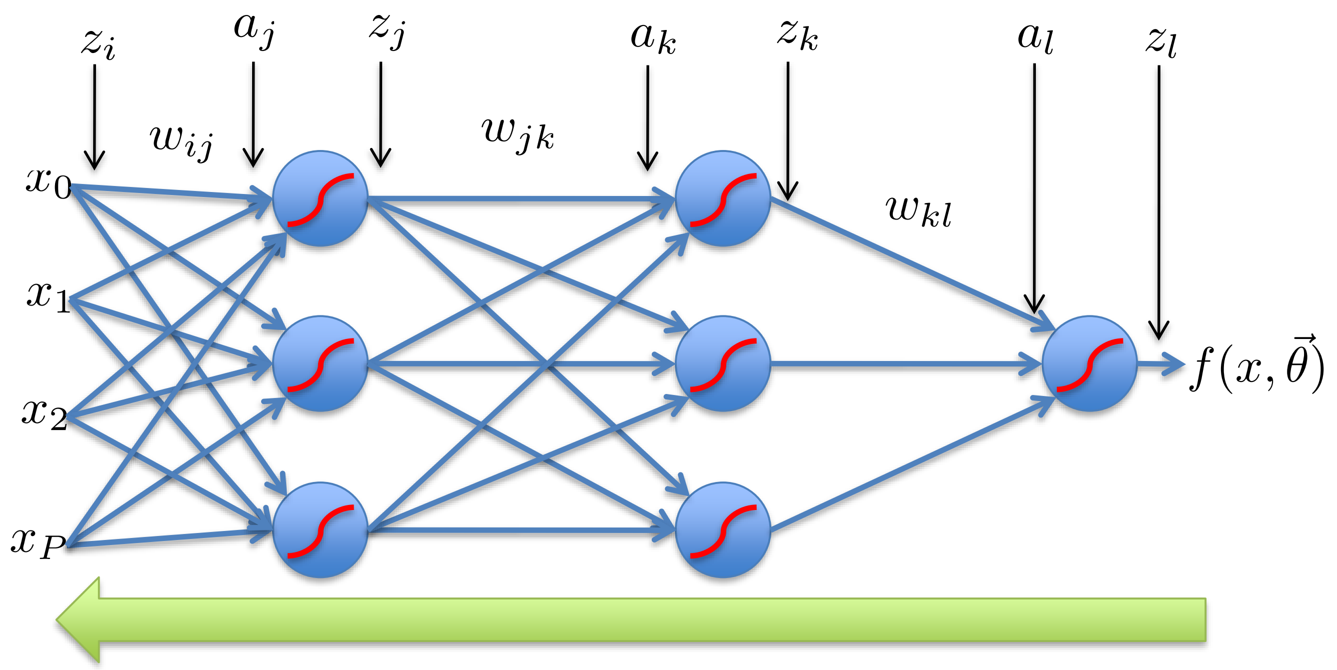
Now that we have well defined gradients for each parameter, update using Gradient Descent

$$\begin{aligned} w_{ij}^{t+1} &= w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}} \\ w_{jk}^{t+1} &= w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}} \\ w_{kl}^{t+1} &= w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}} \end{aligned}$$



Error Back-propagation

- Error backprop unravels the multivariate chain rule and solves the gradient for each partial component separately.
- The target values for each layer come from the next layer.
- This feeds the errors back along the network.



z_i
a_j
z_j
a_k
z_k
a_l
z_l
w_{ij}
w_{jk}
w_{kl}
x_0
x_1
x_2
x_P
$f(x, \vec{\theta})$