A sense of belonging is important to our physical, mental, and spiritual well-being. Yet it is quite possible that at times each of us might feel that we don't fit in. In discouraging moments, we may feel that we will never measure up to the Lord's high standards or the expectations of others. We may unwittingly impose expectations on others—or even ourselves—that are not the Lord's expectations. We may communicate in subtle ways that the worth of a soul is based on certain achievements or callings, but these are not the measure of our standing in the Lord's eyes. "The Lord looketh on the heart." He cares about our desires and longings and what we are becoming.

-- Elder D. Todd Christofferson, October 2022

Alma 23:7

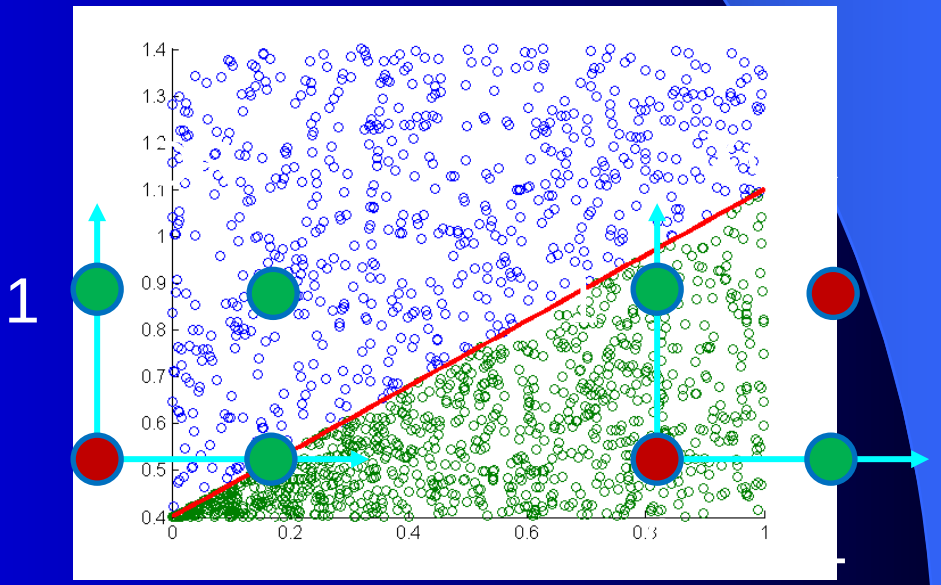
For they became a righteous people; they did lay down the weapons of their rebellion, that they did not fight against God any more, neither against any of their brethren.

MLPs with Backpropagation Learning

History: OR / XOR

- 1958: Perceptron
- 1969: Minsky & Papert: Perceptron can only classify linearly separable data
  - **This killed nearly all research on NNs for the next 15 years**
- XOR as popular example:

i1	i2	or	xor
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0



Towards a Solution

- Problem #1:
  - Perceptron implements discrete model of error (i.e., identifies the *existence* of error and adapts to it)
- Solution:
  - Allow nodes to have real-valued activations (amount of error = target output – computed output)
  - Design learning rule that adjusts weights based on error

    => Delta Rule $\Delta w_i = c(t - net)x_i$
  - Use the learning rule to implement a multi-layer algorithm

Recap (I)

- Replace threshold unit with linear unit, where:

$$o = net = \sum_{j=1}^{m} w_j x_j + w_{m+1}$$

bias

- Define error as:

$$\mathrm{E} = \frac{1}{2} \sum_{i} (t_i - o_i)^2$$

- Minimize E, giving gradient-descent rule:

$$\frac{\partial E}{\partial w_j} = -\sum_{i} (t_i - o_i) x_{ij}$$

Recap (II)

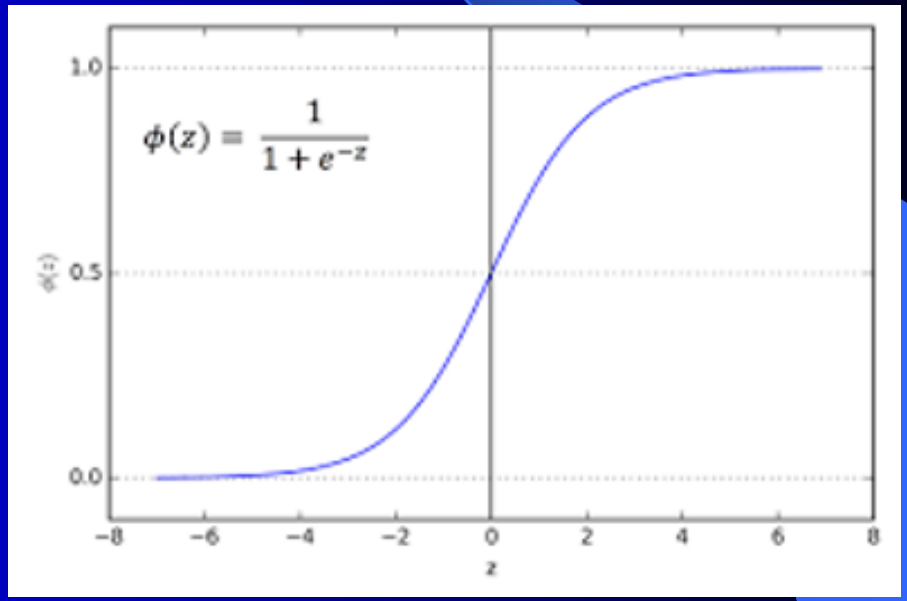
- Convergence is guaranteed if the problem is solvable
- BUT:
  - Still produces only linear functions
  - Even when used in a multi-layer context
- We need:
  - **Non-linear output function**
    - Must be differentiable for gradient computation

Non-linear Output

- Introduce non-linearity with sigmoid function:

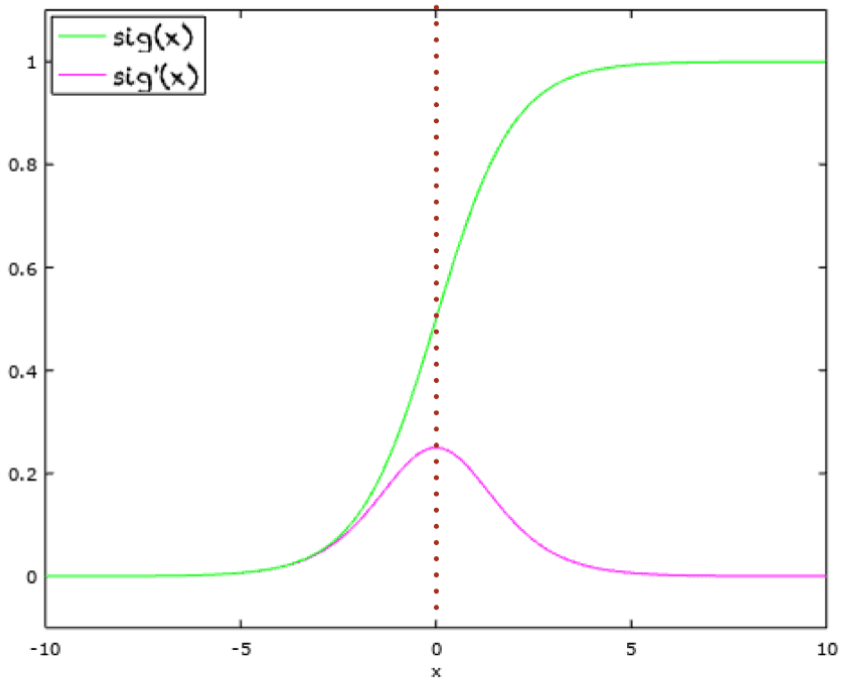
$$net = \sum_{j=1}^{m} w_j x_j + w_{m+1}$$

$$o = \frac{1}{1 + e^{-net}}$$



Sigmoid Function

- Differentiable, most unstable in middle



Plot of σ(x) and its derivate σ'(x)

Domain: $(-\infty, +\infty)$

Range: $(0, +1)$

$\sigma(0) = 0.5$

Other properties

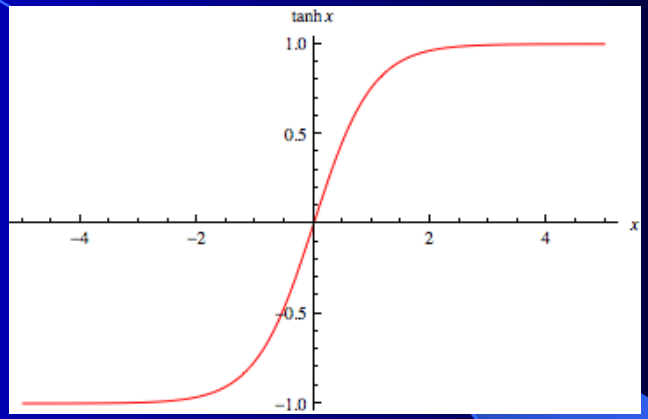
$$\sigma(x) = 1 - \sigma(-x)$$

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

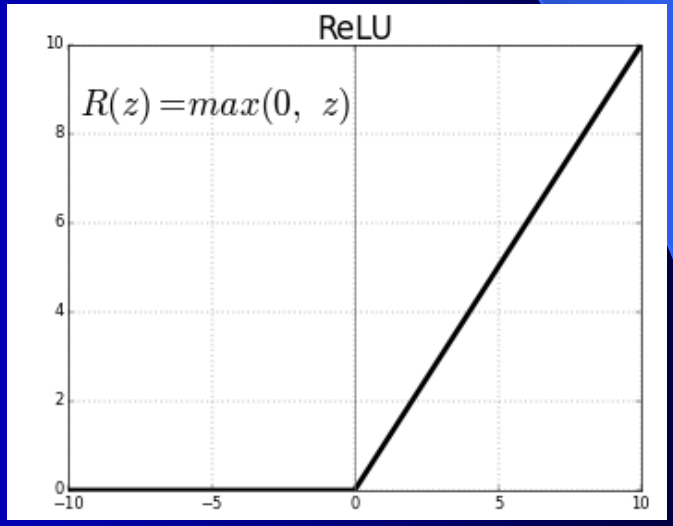
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Other Activation Functions

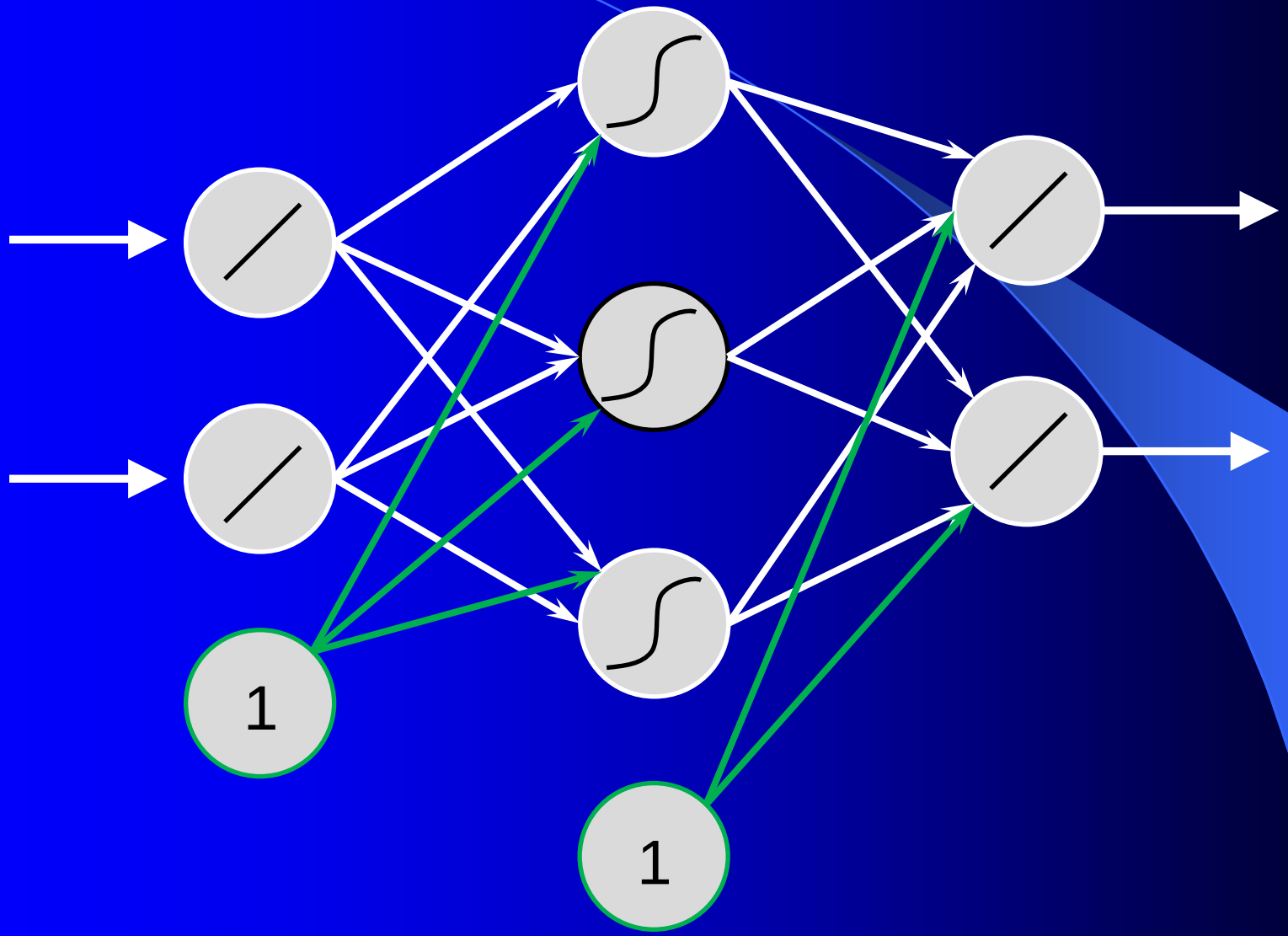
- Hyperbolic tangent (tanh)
  - Same shape as sigmoid but tanh(0)=0 (rather than 0.5)



- Rectified linear activation function (ReLU)
  - Popular in deep nets



Combine perceptrons in layers to create a more powerful model



Universal Function Approximation

Theorem

Given any bounded continuous function *f*, there exists a 1 hidden layer (i.e., 2 layers of weights) feed-forward neural network that can approximate *f* to any arbitrary degree of accuracy

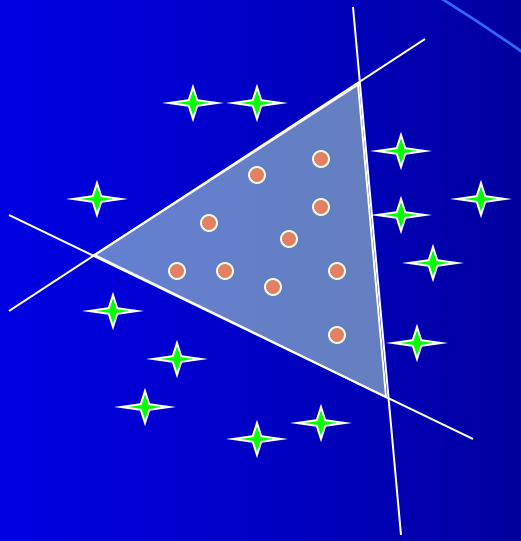
Theroy and Practice…

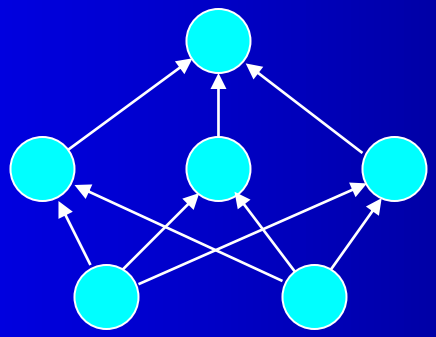
- In theory then, we don't need more than 1 hidden layer
- However, this is an EXISTENCE proof!

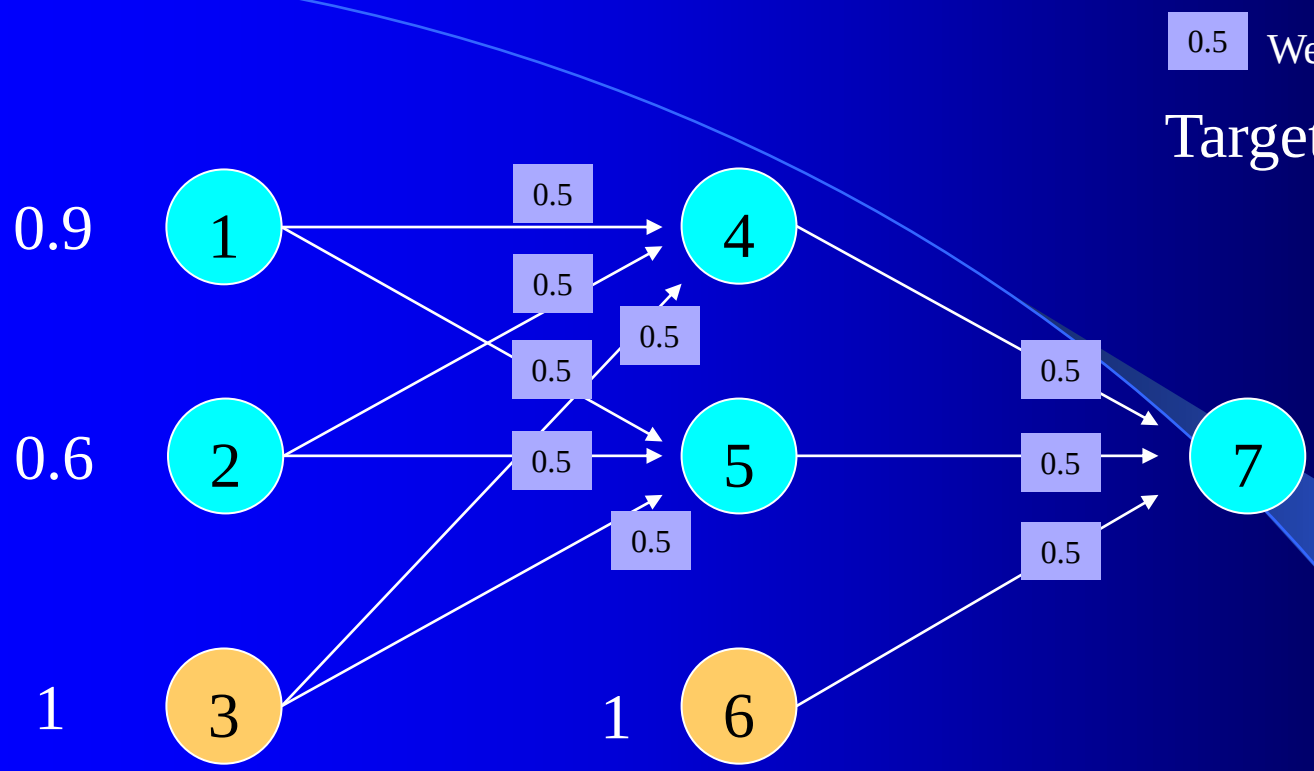
- In practice, and for a long time, these "simple" networks have been the most common
  - So, we describe them here in detail

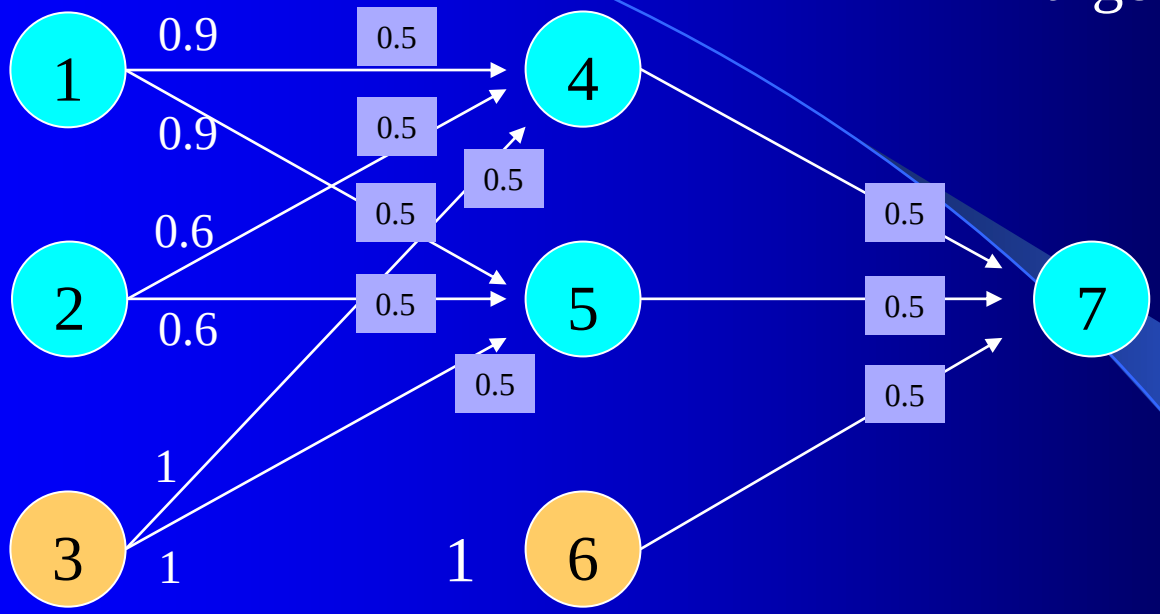
- Recently… deep(er) networks have been showing excellent results!
  - The basic building blocks/structure remain

Multi-Layer Generalization





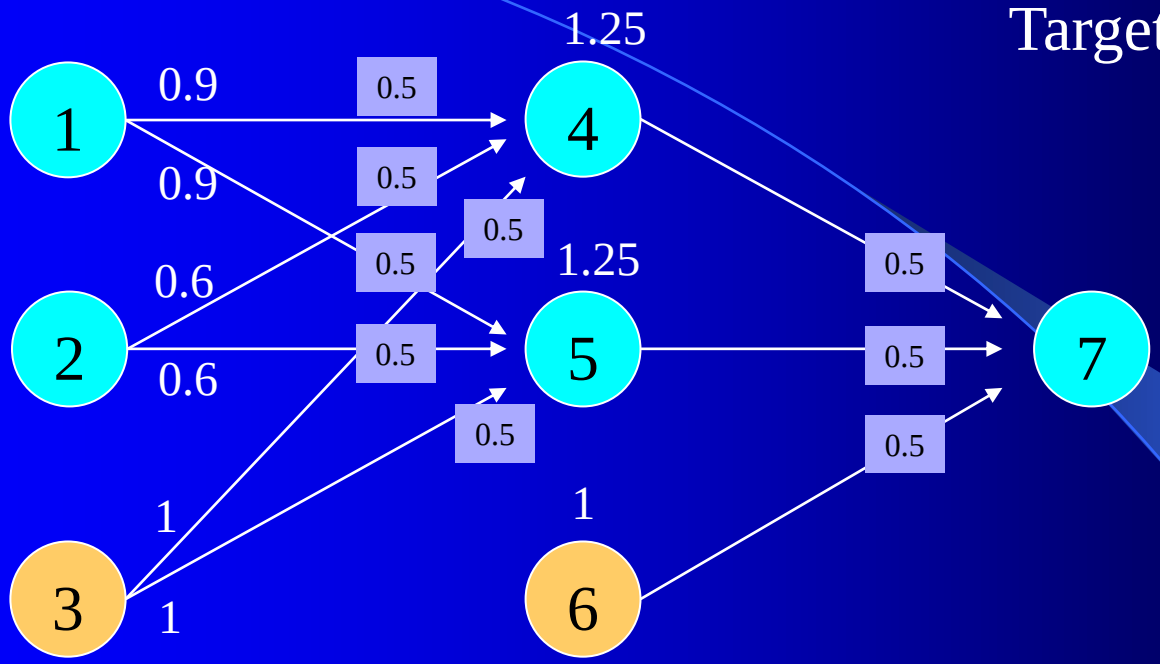

0.5 Weights
Target = 0
0.9
0.6
1
1
1
2
3
4
5
6
7
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5



$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

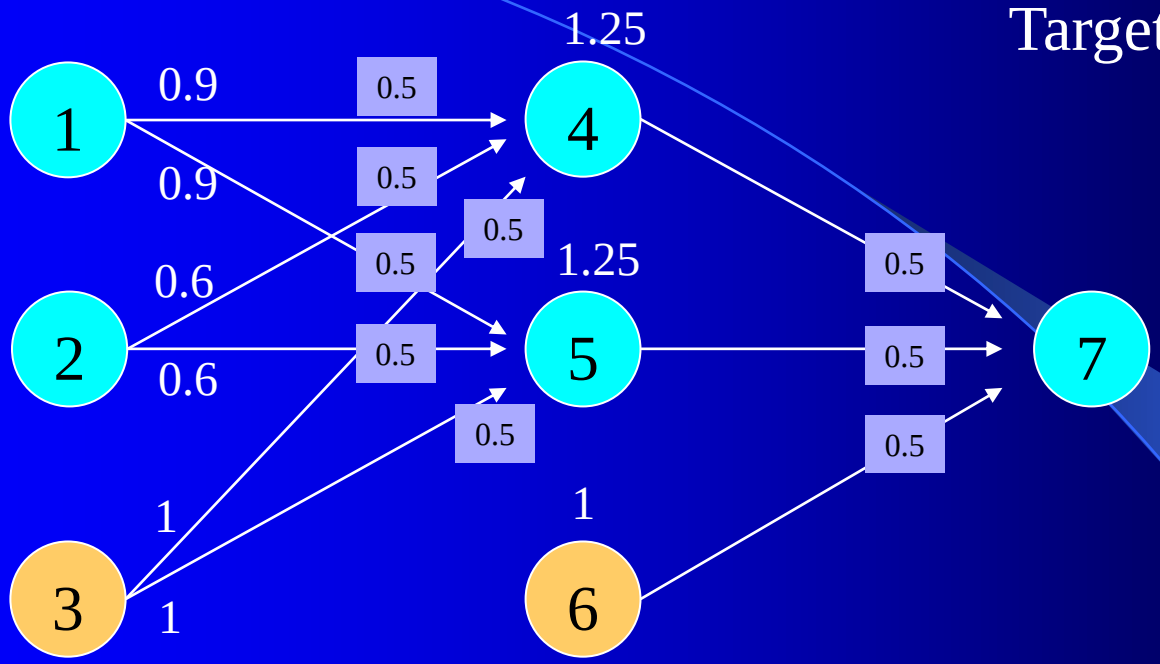
$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

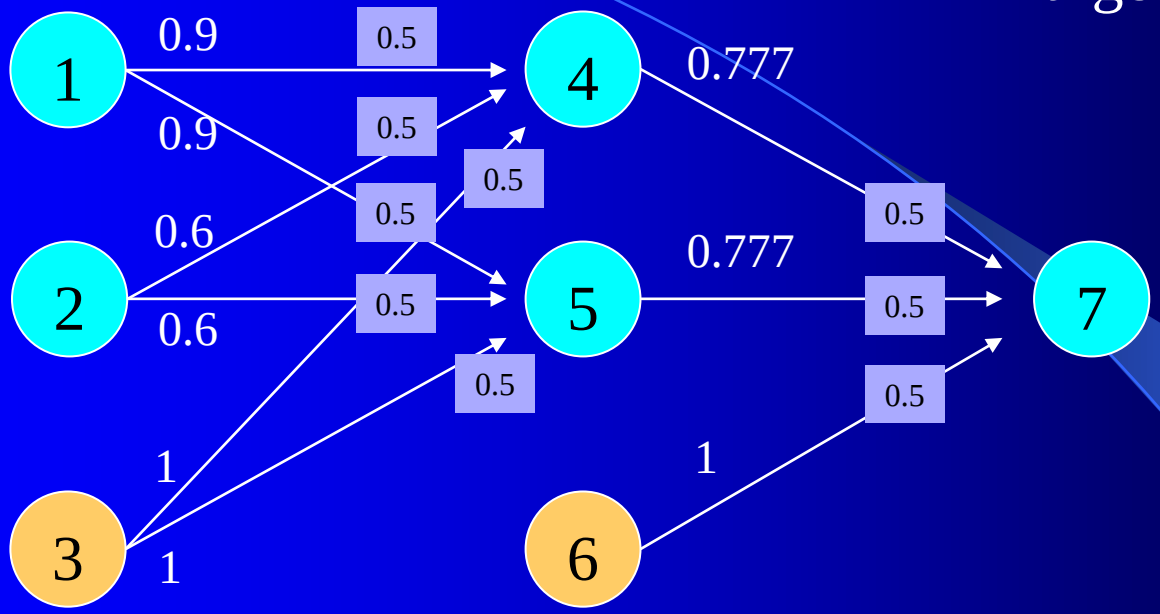
$z_i = \dfrac{1}{(1 + e^{-neti})}$

net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25

net_5 = 1.25

$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$



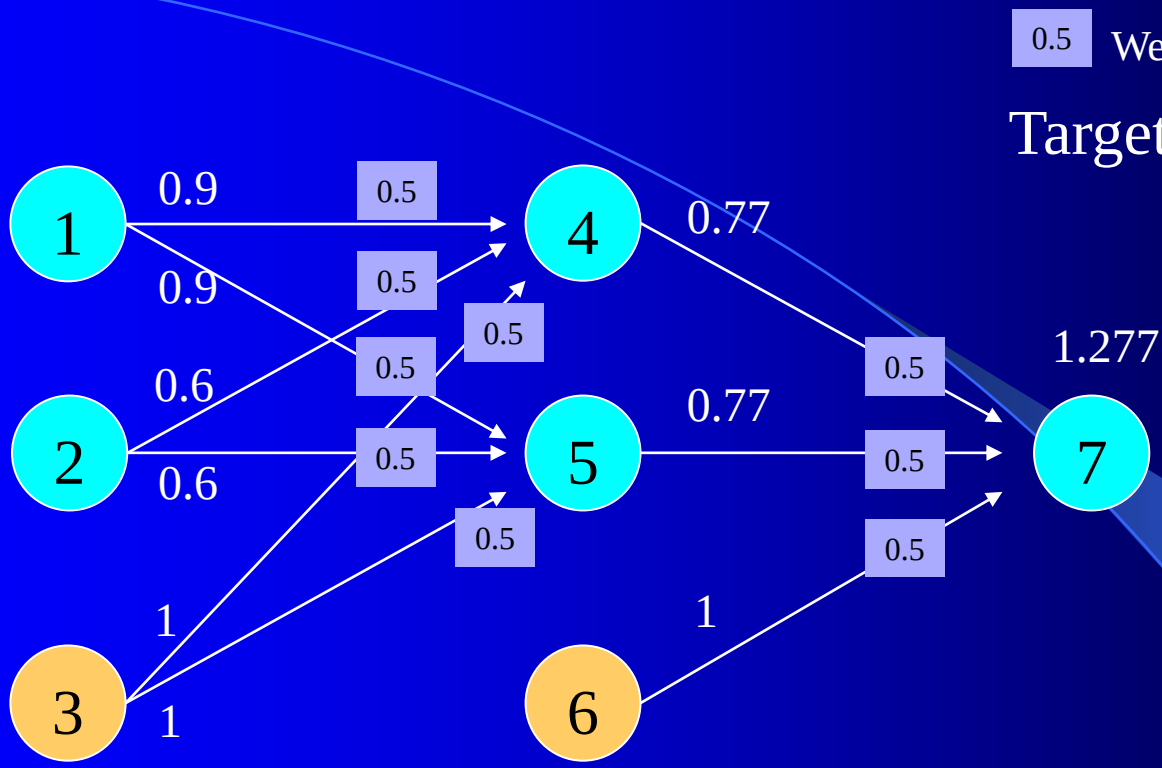
$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

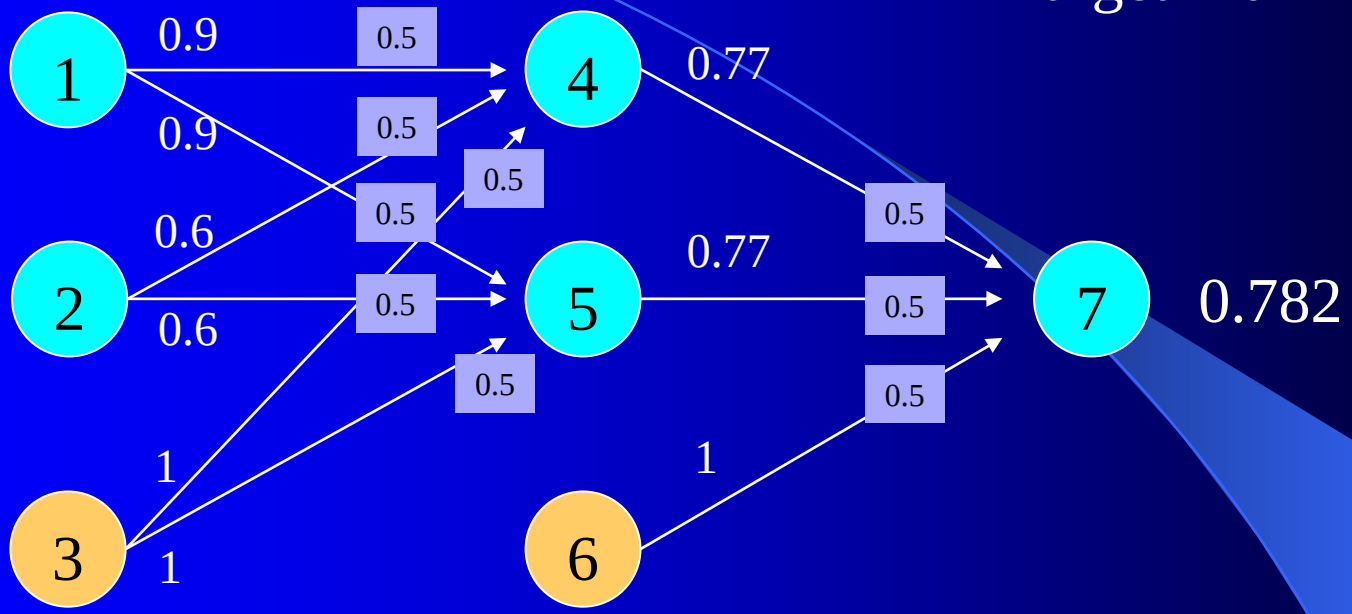
$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$



$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$
$net_5 = 1.25$
$z_4 = 1/(1 + e^{-1.25}) = .777$
$z_5 = .777$
$net_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$



$net_q = \sum x_i w_{ij}$

$$z_i = \frac{1}{(1 + e^{-neti})}$$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

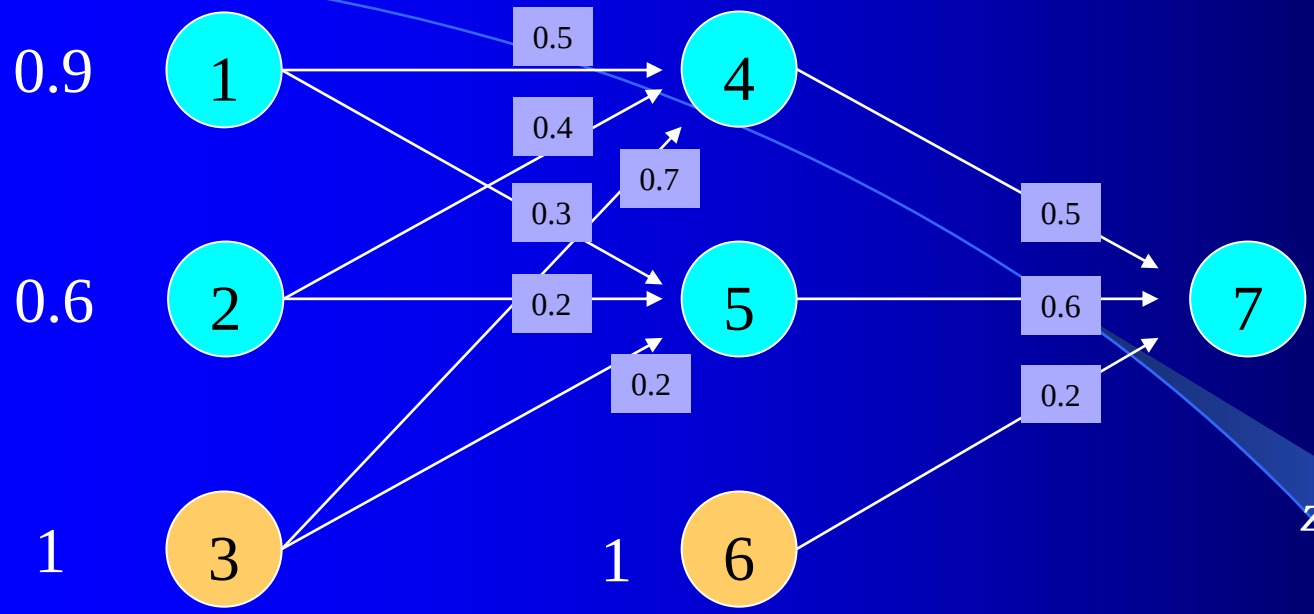
$net_5 = 1.25$

$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$

$net_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$

$z_7 = 1/(1 + e^{-1.277}) = .782$



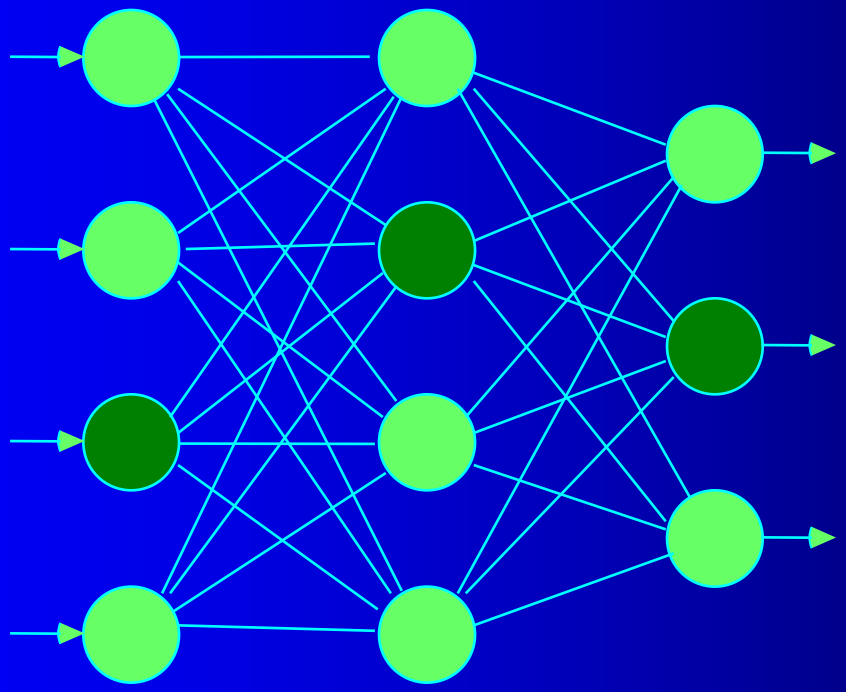
$$z_i = \frac{1}{(1 + e^{-neti})}$$

x1	x2	bias		weights for layer 1		layer 1 matrix multiply		output for layer 1	
0.9	0.6	1		0.5	0.3	1.39	0.59	0.80059224	0.64336515
0.7	0.4	1		0.4	0.2	1.21	0.49	0.77029895	0.62010643
1.2	0.6	1		0.7	0.2	1.54	0.68	0.82346473	0.6637387
-0.1	-0.2	1				0.57	0.13	0.63876318	0.53245431
	4 x 3			3 x 2		4 x 2			

output for layer 1		bias		weights for layer 2		layer 2 matrix multiply		output for layer 2
0.80059224	0.64336515	1		0.5		0.98631521		0.72835949
0.77029895	0.62010643	1		0.6		0.95721333		0.72256352
0.82346473	0.6637387	1		0.2		1.00997558		0.73301537
0.63876318	0.53245431	1				0.83885417		0.69822384

NEURAL NETWORK PLAYGROUND

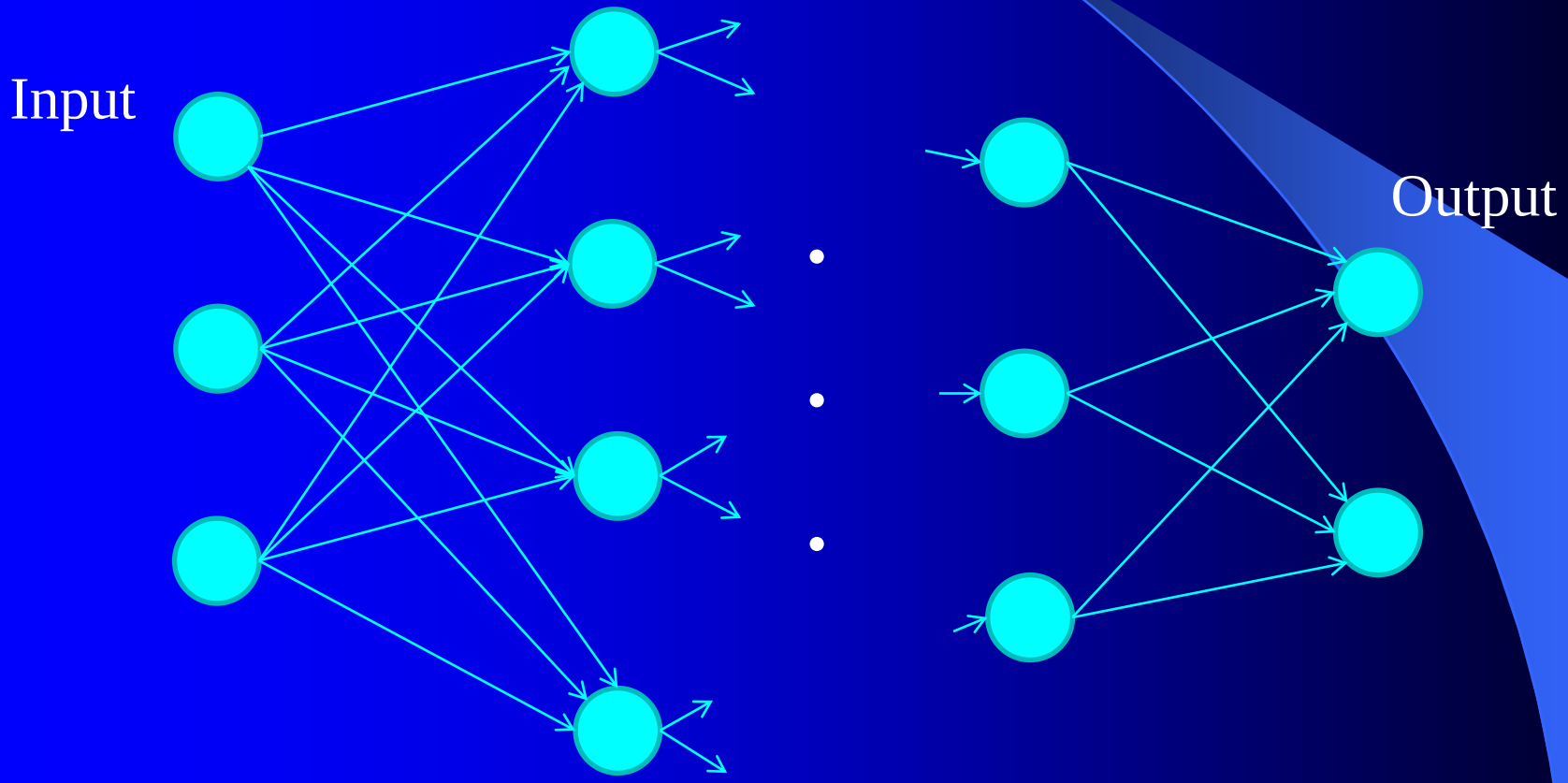
Responsibility Problem



Output 1
Wanted 0

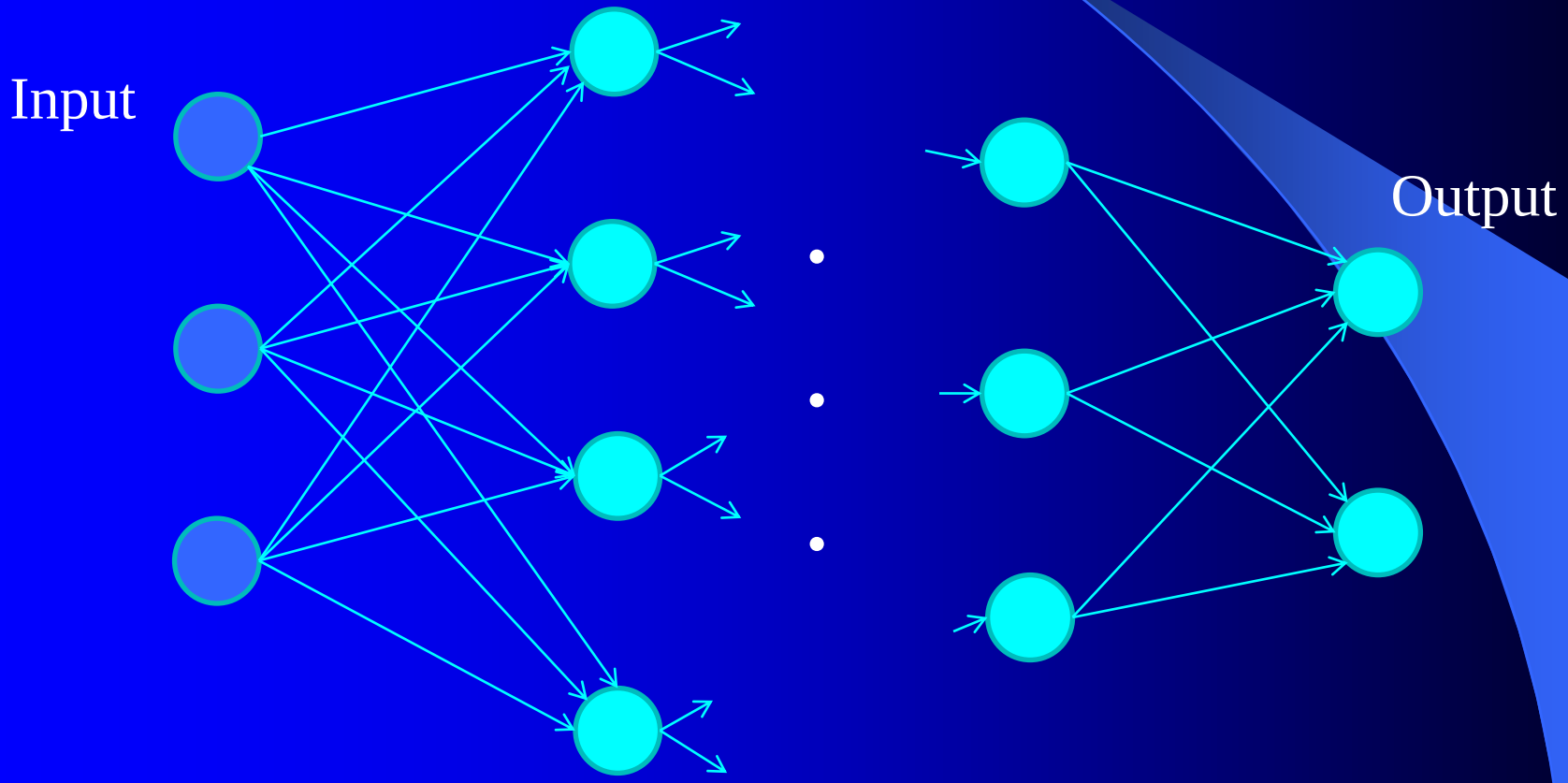
Back-Propagation

- Operates similarly to perceptron learning



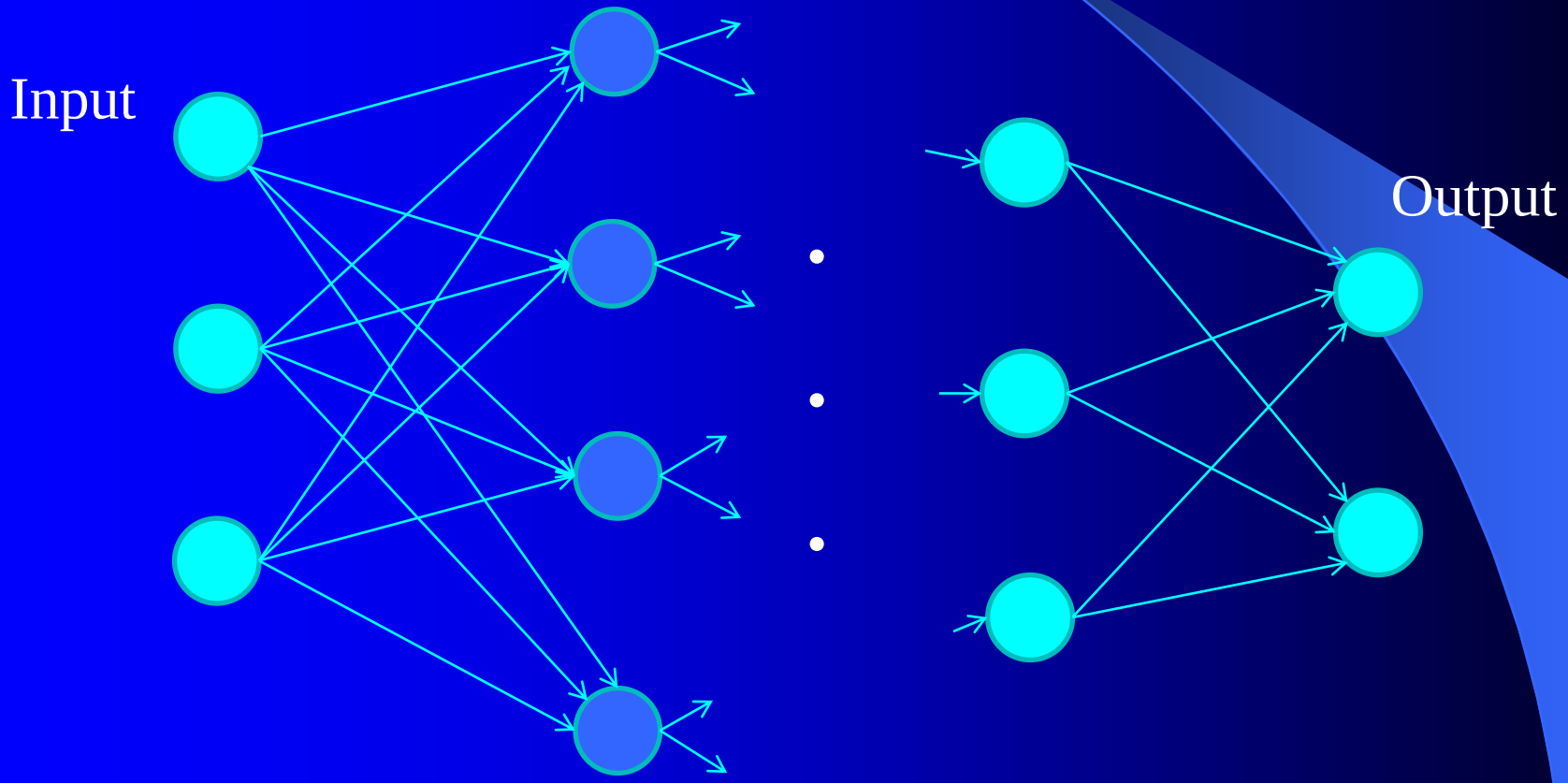
Back-Propagation

- Inputs are fed forward through the network



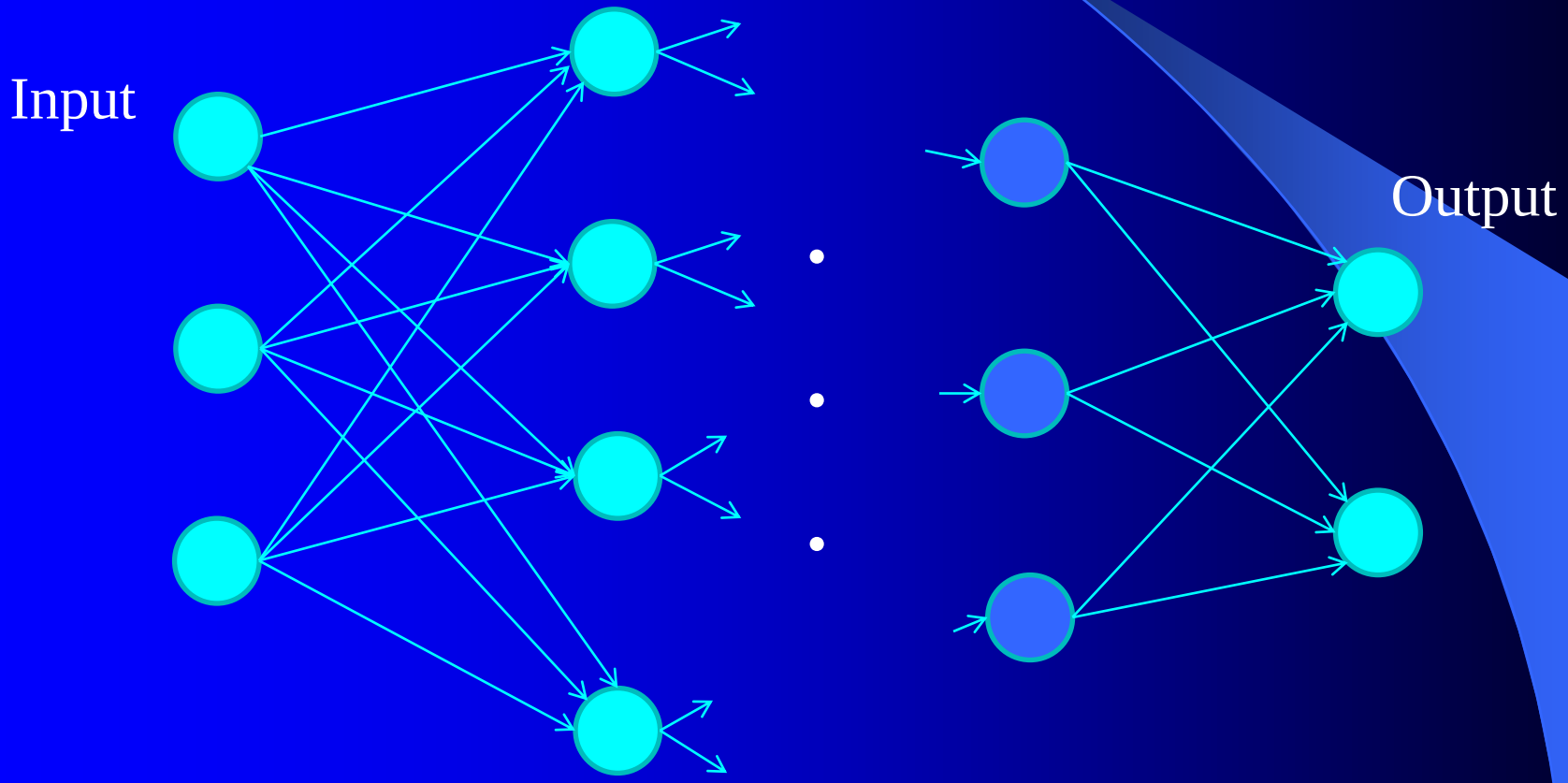
Back-Propagation

- Inputs are fed forward through the network



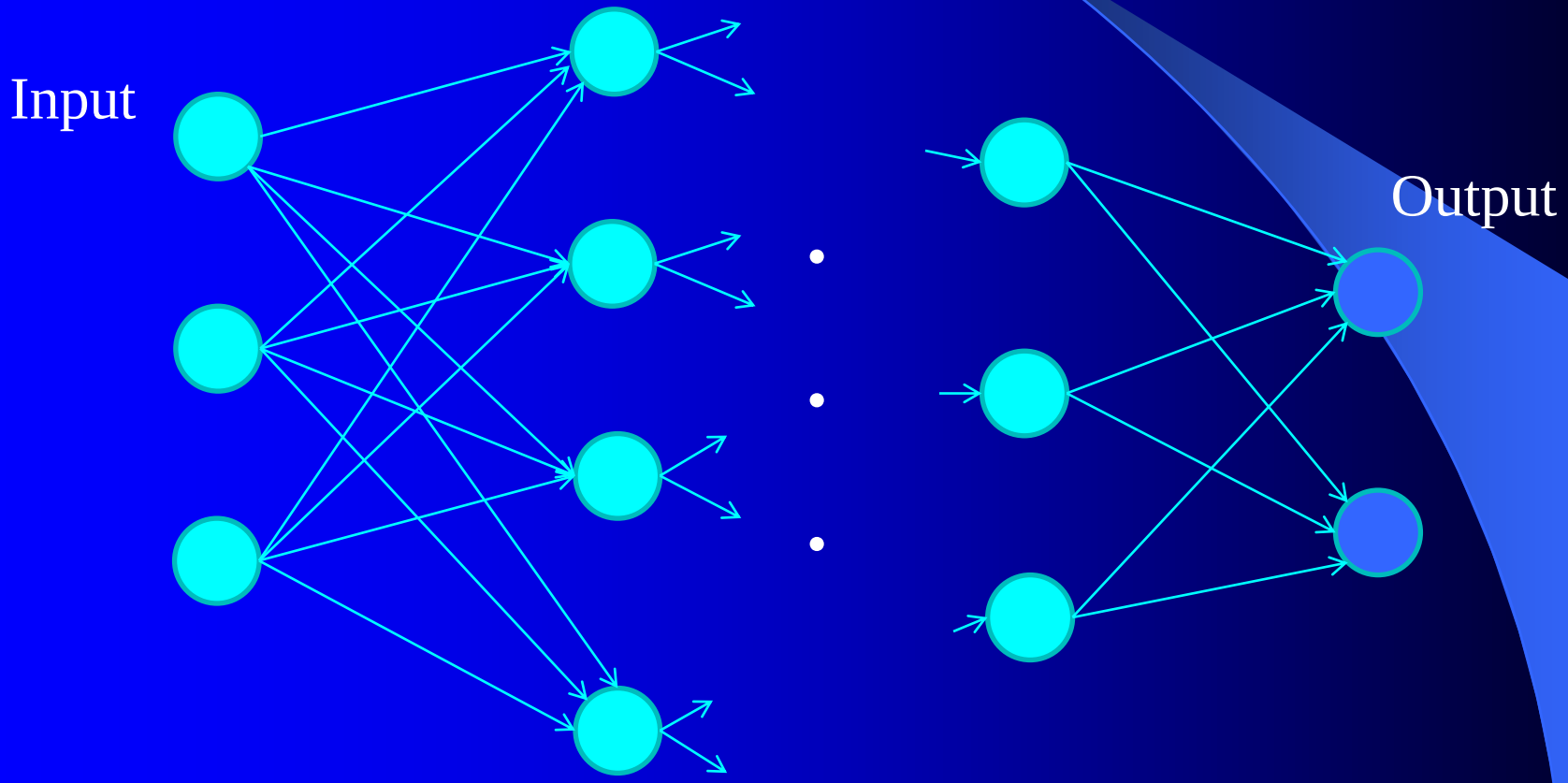
Back-Propagation

- Inputs are fed forward through the network



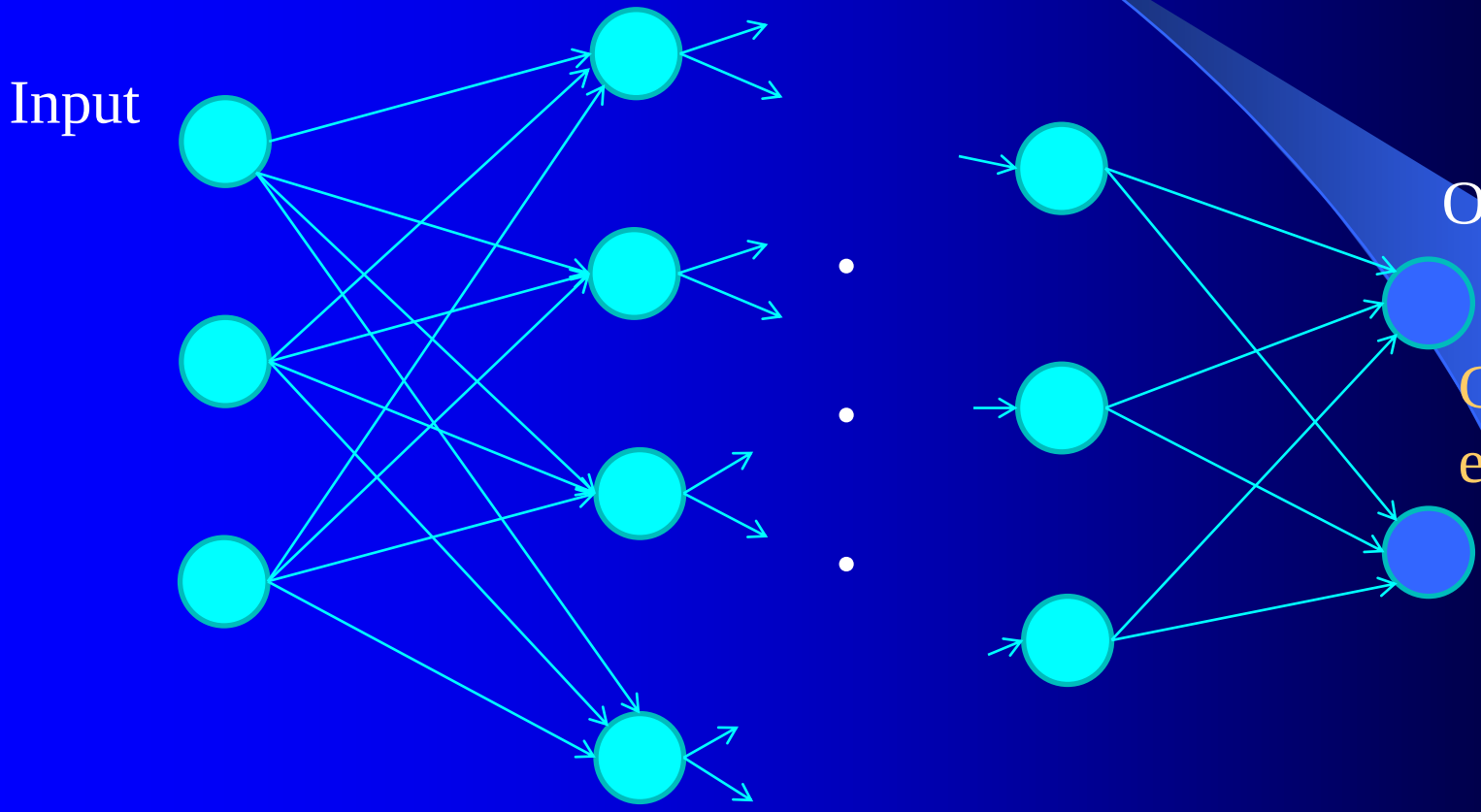
Back-Propagation

- Inputs are fed forward through the network



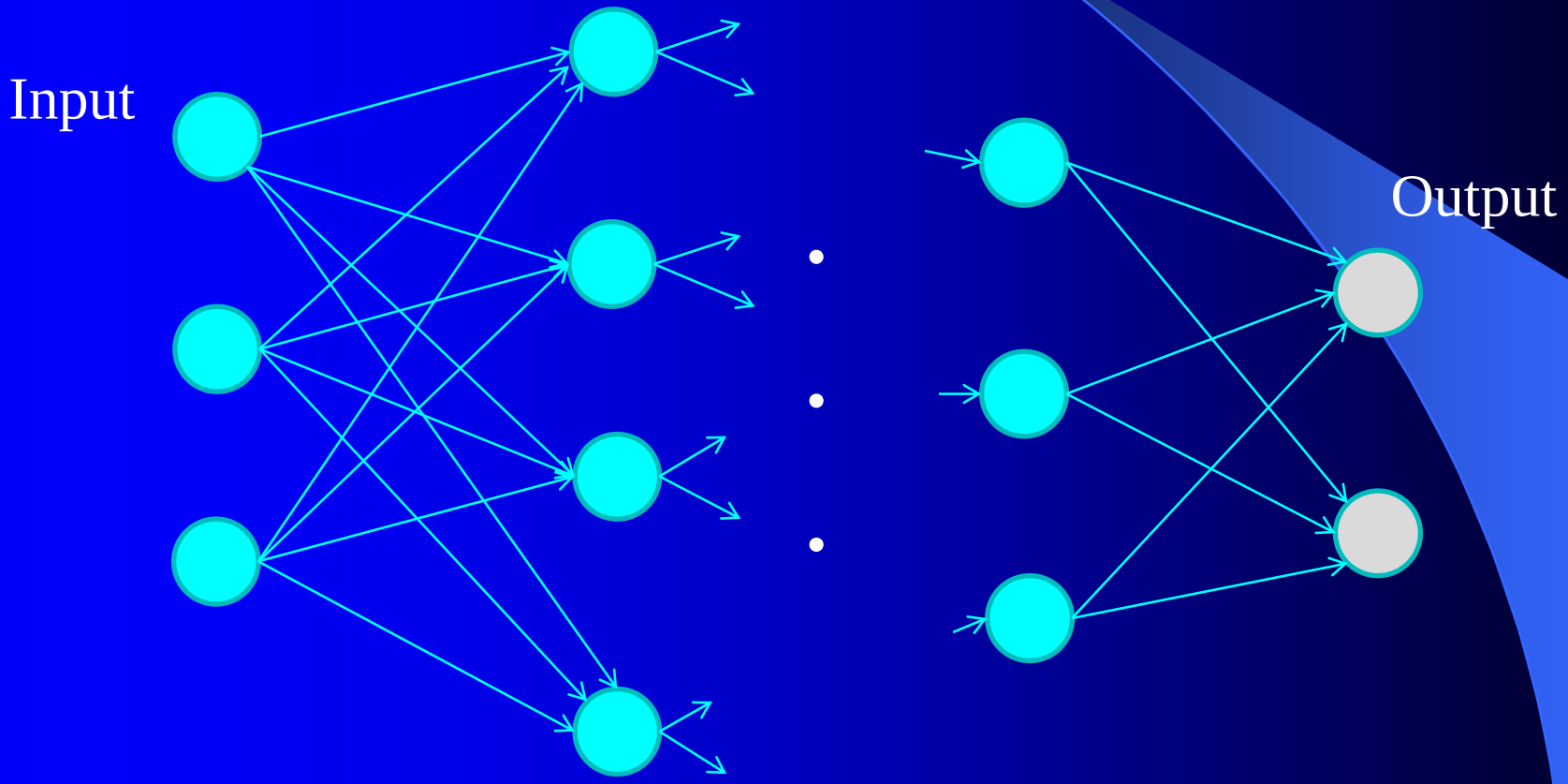
Back-Propagation

- Inputs are fed forward through the network



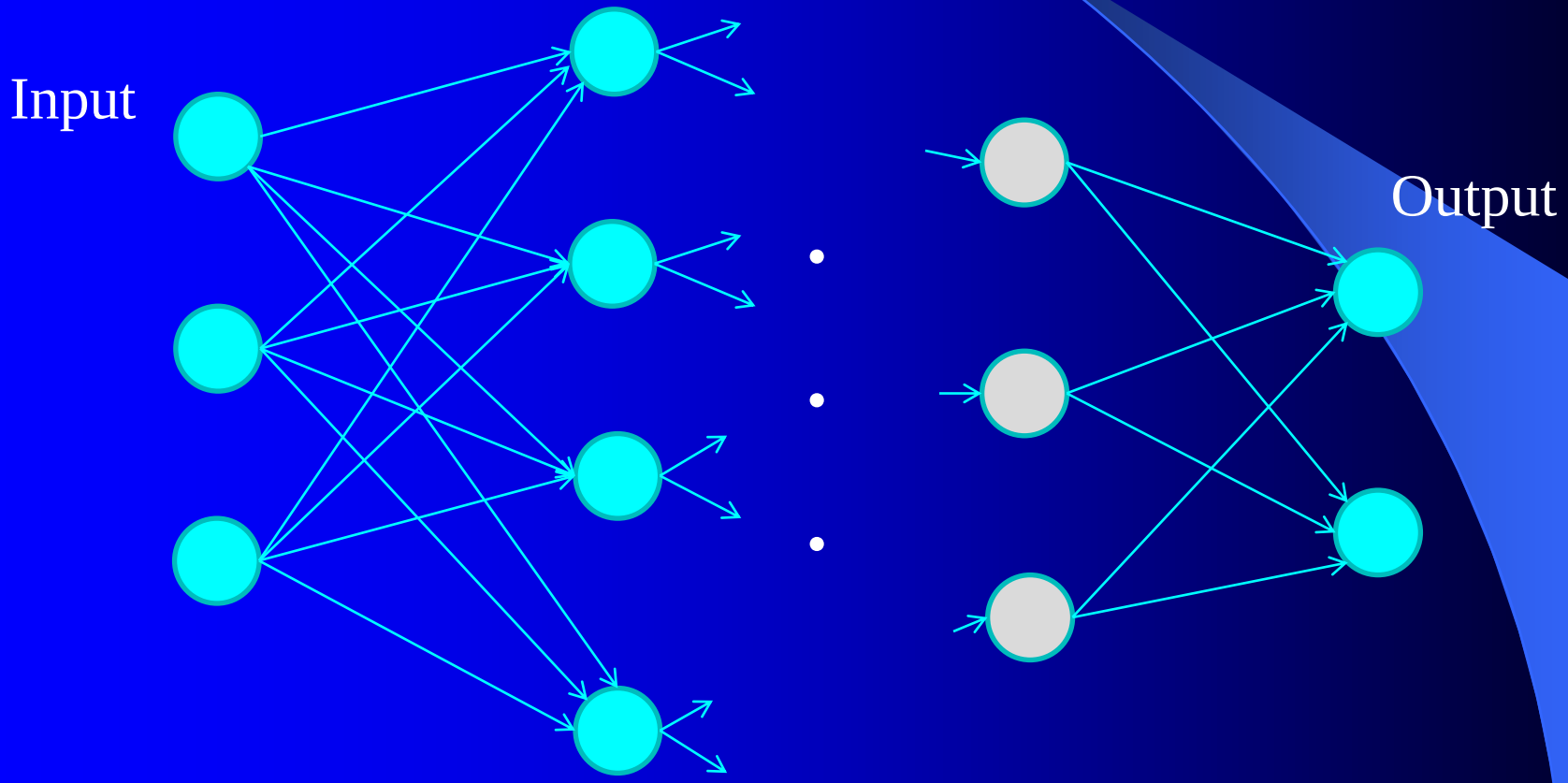
Back-Propagation

- Errors are propagated back



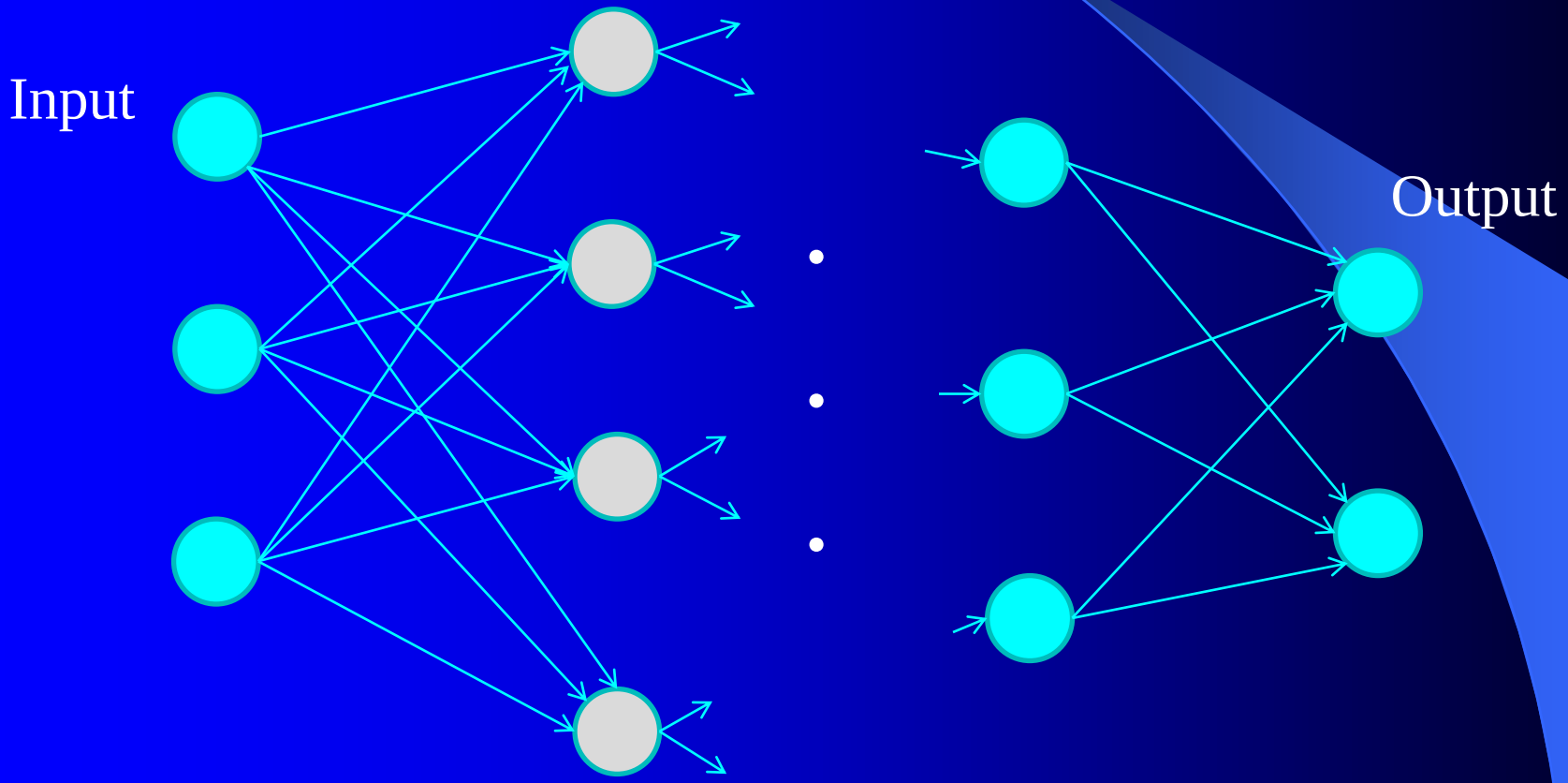
Back-Propagation

- Errors are propagated back



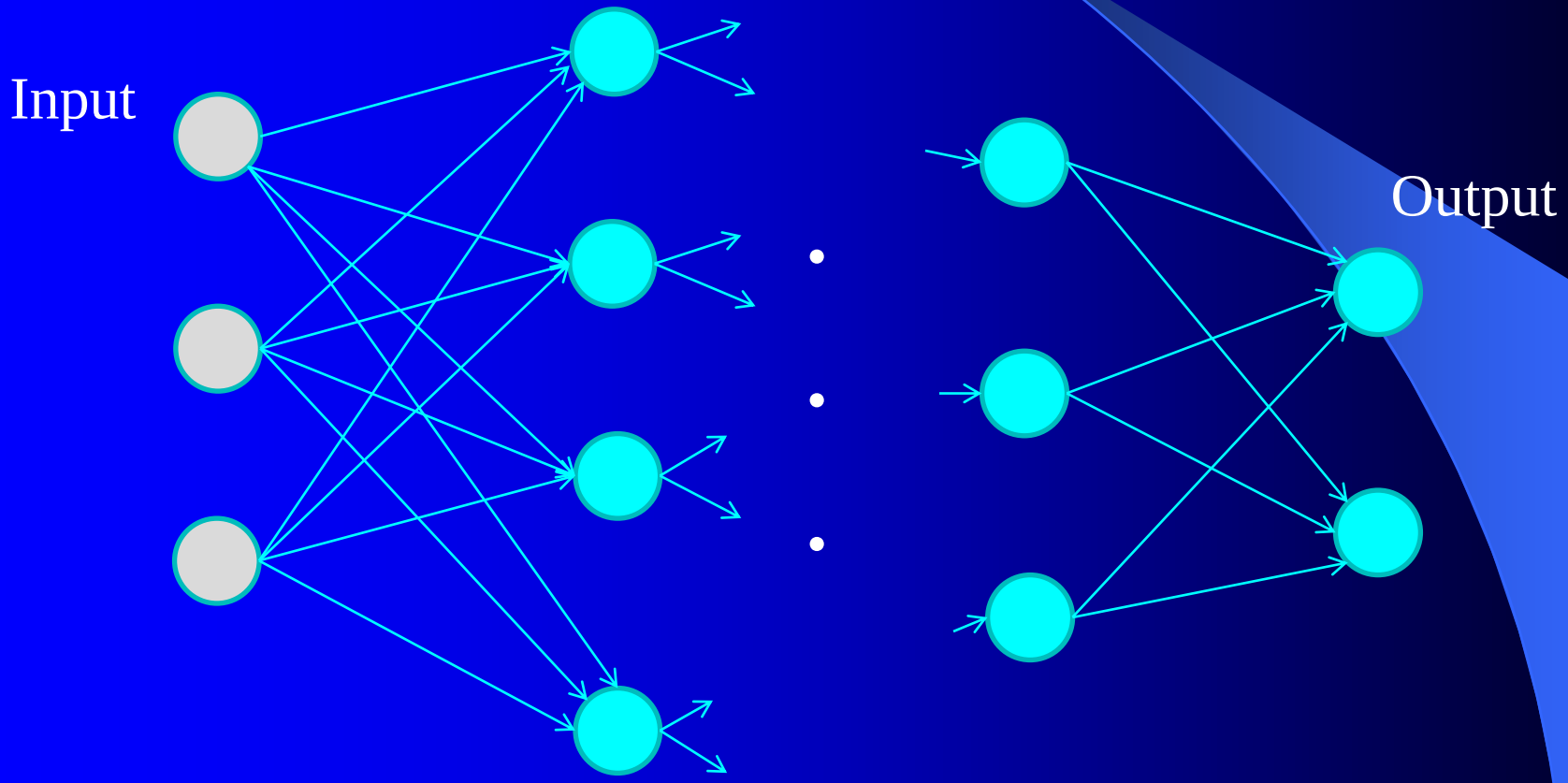
Back-Propagation

- Errors are propagated back



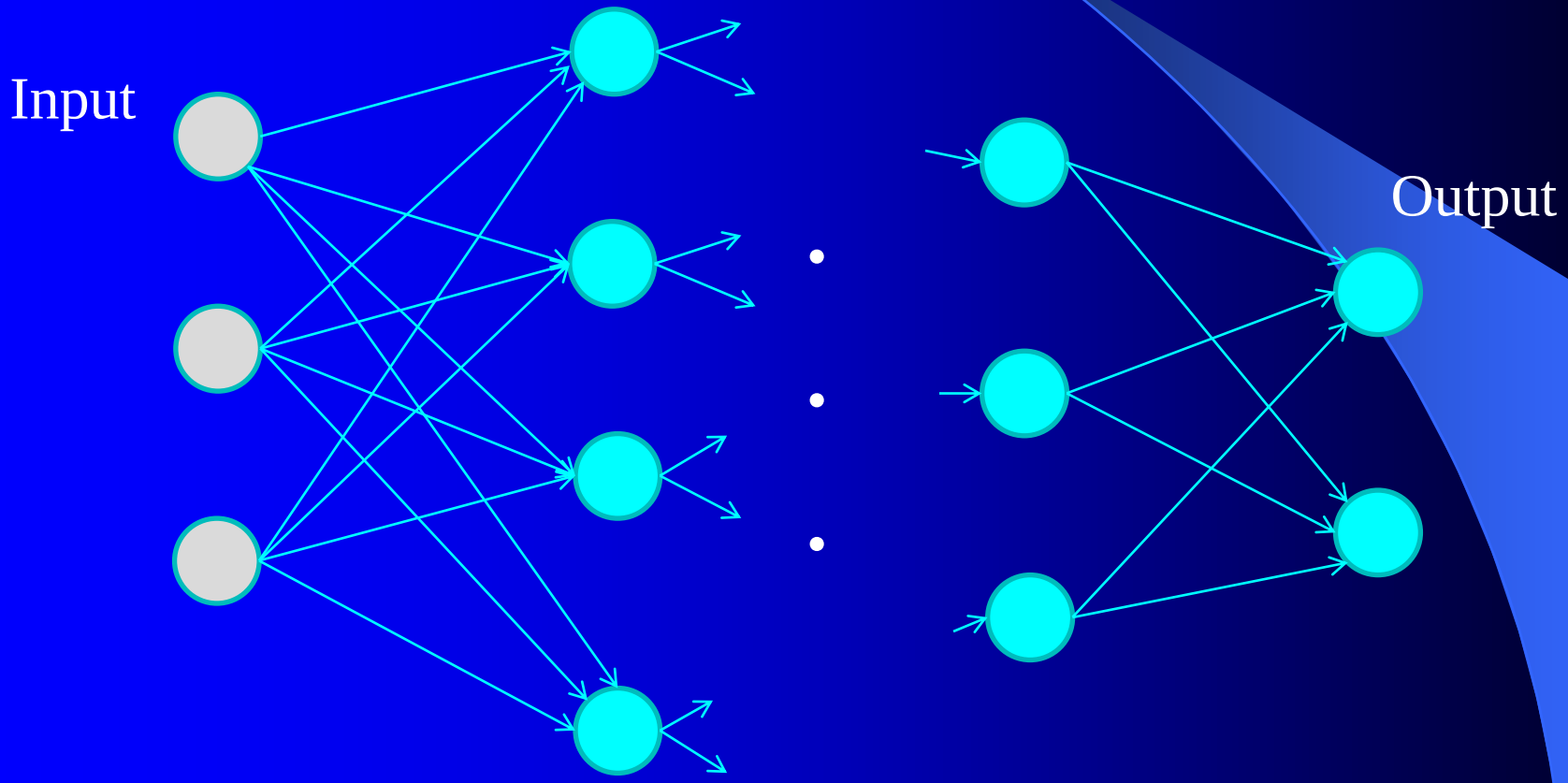
Back-Propagation

- Errors are propagated back



Back-Propagation

- Adjust weights based on errors



Backpropagation

- Seppo Linnainmaa (1970) – not specifically neural nets
- Neural specific – Werbos(1974, implementation - 1982),
- Rumelhart (1986) – backprop yields useful internal representations in hidden layers
- Explosion of neural net interest
- Able to train multi-layer perceptrons (and other topologies)
- Commonly uses differentiable sigmoid function which is the smooth (squashed) version of the threshold function
- Error is propagated back through earlier layers of the network
- Very fast efficient way to compute gradients!

Back-Propagation Training

- Weights might be updated after each pass or after multiple passes
- Need a comprehensive training set
- Network cannot be too large for the training set
- No guarantees the network will learn
- Network design and learning strategies impact the speed and effectiveness of learning

Backpropagation Learning Algorithm

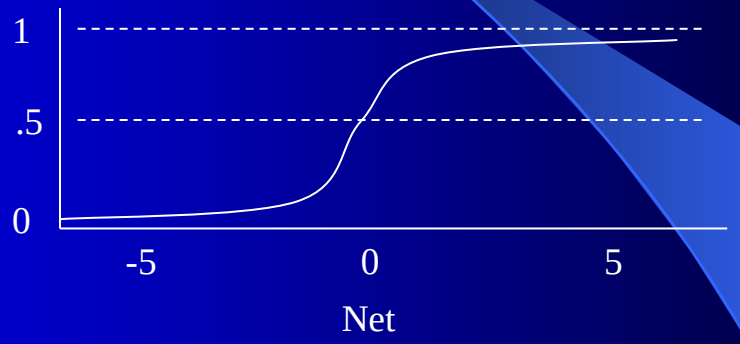
- Until Convergence (low error or other stopping criteria) do
  - Present a training pattern
  - Calculate the error of the output nodes (based on $T - Z$)
  - Calculate the error of the hidden nodes (based on the error of the output nodes which is propagated back to the hidden nodes)
  - Continue propagating error back until the input layer is reached
  - Then update all weights based on the standard delta rule with the appropriate error function δ

$$\Delta w_{ij} = C\, \delta_j\, Z_i$$

Activation Function and its Derivative

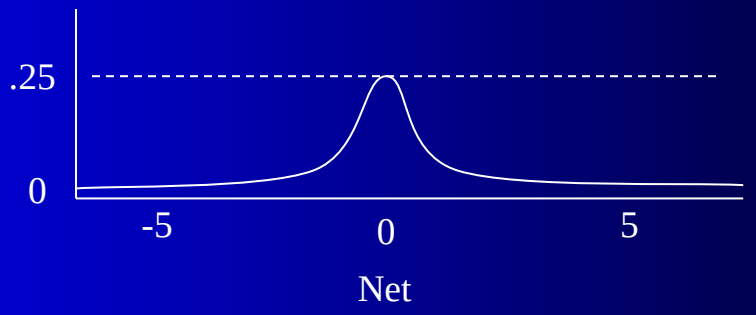
- Node activation function *f(net)* is commonly the sigmoid

$$Z_j = f(net_j) = \frac{1}{1 + e^{-net_j}}$$



- Derivative of activation function is a critical part of the algorithm

$$f'(net_j) = Z_j(1 - Z_j)$$

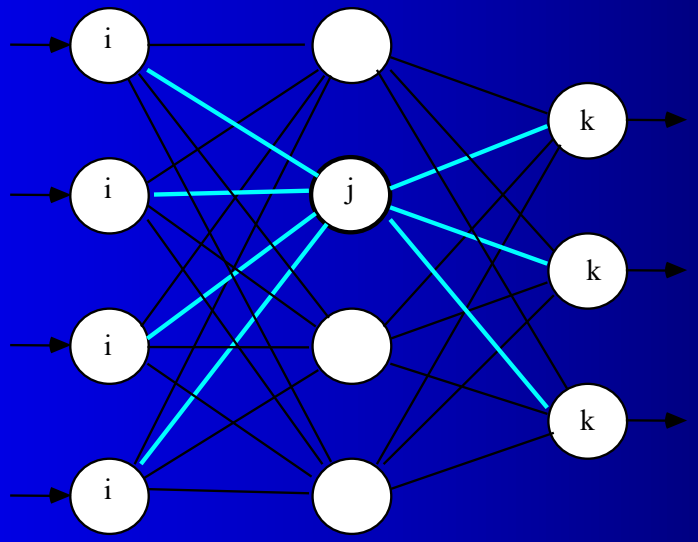


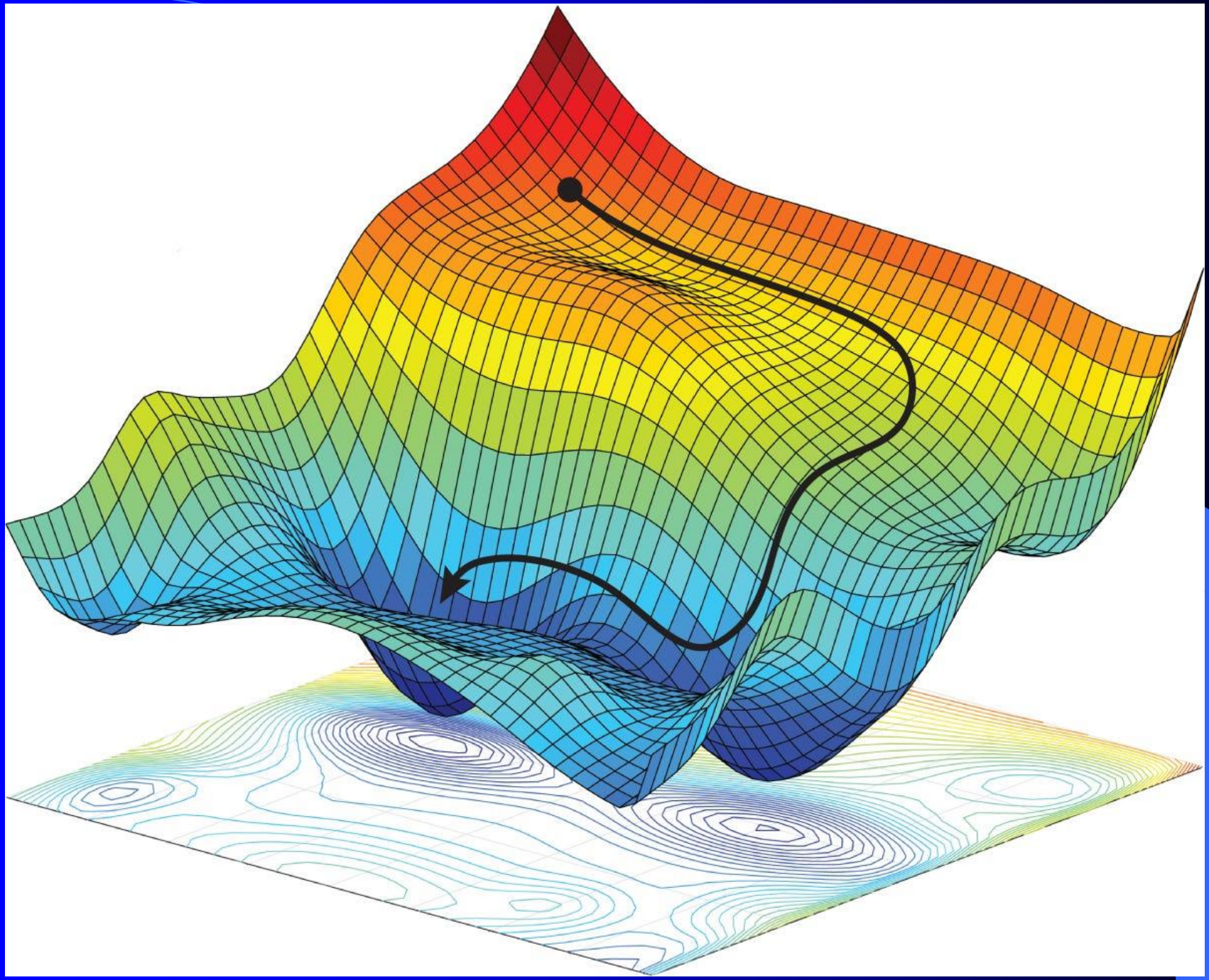
Backpropagation Learning Equations

$$\Delta w_{ij} = C \delta_j Z_i$$

$$\delta_j = (T_j - Z_j) f'(net_j) \qquad \text{[Output Node]}$$

$$\delta_j = \sum_k (\delta_k w_{jk}) f'(net_j) \qquad \text{[Hidden Node]}$$





Network Error and Gradient

E_d is the error on training example d, summed over all output units:

$$E_d = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Let net_q represent the activation function of unit q prior to its passing through the sigmoid:

$$net_q = \sum_r w_{rq} o_r$$

where the o_r's are the outputs of the units feeding directly into unit q.

Using the chain rule, for a given weight w_{pq} between q and one of its input units p, we have:

$$\begin{aligned}
\frac{\partial E_d}{\partial w_{pq}} &= \frac{\partial E_d}{\partial net_q} \frac{\partial net_q}{\partial w_{pq}} \\
&= \frac{\partial E_d}{\partial net_q} \frac{\partial}{\partial w_{pq}} \sum_r w_{rq} o_r \\
&= \frac{\partial E_d}{\partial net_q} \sum_r \frac{\partial}{\partial w_{pq}} [w_{rq} o_r] \\
&= \frac{\partial E_d}{\partial net_q} o_p
\end{aligned}$$

There are now two cases to consider:

1. q is an output unit.
2. q is an hidden unit.

Gradient search computes the gradient at our current spot on the Error space. We then adjust our weights to follow that gradient down to the minimum point.

To calculate the gradient, compute the partial derivative of the Error function with respect to the weights.

Output Node Error

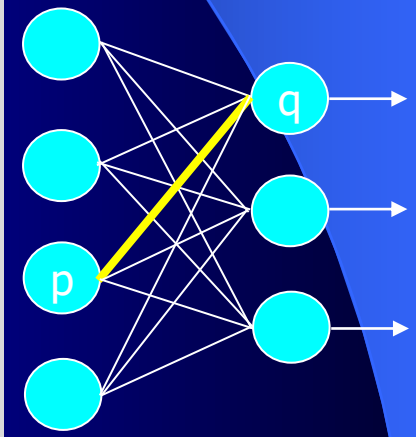
$$\frac{\partial E_d}{\partial net_q} = \{\frac{\partial E_d}{\partial o_q}\}\{\frac{\partial o_q}{\partial net_q}\}$$
$$= \{\frac{\partial E}{\partial o_q}[\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2]\}\{\frac{\partial}{\partial net_q}\sigma(net_q)\}$$
$$= \{\frac{1}{2}\sum_{k \in outputs}[\frac{\partial}{\partial o_q}(t_k - o_k)^2]\}\{\sigma(net_q)(1 - \sigma(net_q))\}$$
$$= \{\frac{1}{2}\sum_{k \in outputs}[2(t_k - o_k)\frac{\partial}{\partial o_q}(t_k - o_k)]\}\{o_q(1 - o_q)\}$$
$$= \{\sum_{k \in outputs}[(t_k - o_k)\frac{\partial}{\partial o_q}(t_k - o_k)]\}\{o_q(1 - o_q)\}$$
$$= \{-(t_q - o_q)\}\{o_q(1 - o_q)\}$$
$$= -(t_q - o_q)o_q(1 - o_q)$$

And it follows that:

$$\Delta w_{pq} = c(t_q - o_q)o_q(1 - o_q)o_p$$

Now we calculate the partial derivative of E w.r.t. net_q

$\sigma(net_q)$ is the sigmoid function on the net value of node q which is the output of node $q = o_q$



Hidden Node Error

$$\frac{\partial E_d}{\partial net_q} = \sum_{k \in follows(q)} \{\frac{\partial E_d}{\partial net_k}\}\{\frac{\partial net_k}{\partial net_q}\}$$

$$= \sum_{k \in follows(q)} \{\frac{\partial E_d}{\partial net_k}\}\{\frac{\partial net_k}{\partial o_q}\frac{\partial o_q}{\partial net_q}\}$$

$$= \sum_{k \in follows(q)} \{\frac{\partial E_d}{\partial net_k}\}\{\frac{\partial}{\partial o_q}\sum_r w_{rk} o_r \frac{\partial o_q}{\partial net_q}\}$$

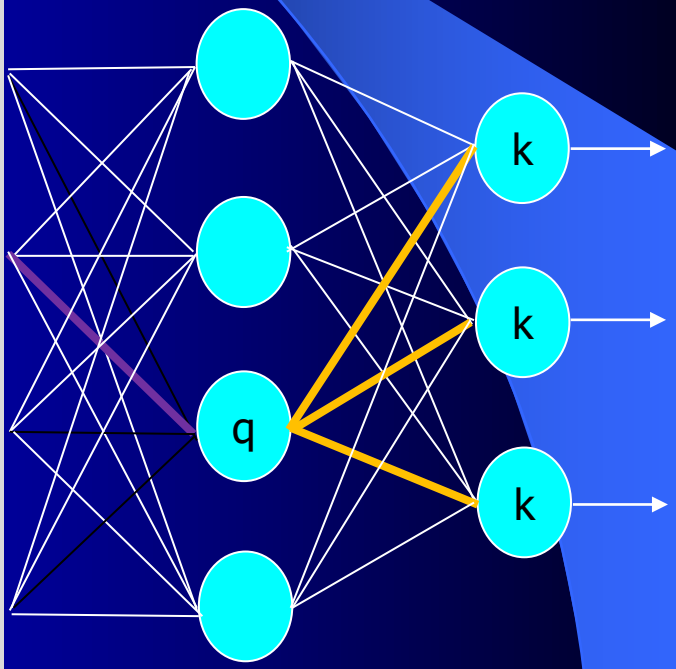
$$= \sum_{k \in follows(q)} \{\frac{\partial E_d}{\partial net_k}\}\{w_{qk}\frac{\partial o_q}{\partial net_q}\}$$

$$= \sum_{k \in follows(q)} \{\frac{\partial E_d}{\partial net_k}\}\{w_{qk} o_q(1-o_q)\}$$

$$= [\sum_{k \in follows(q)} w_{qk}\frac{\partial E_d}{\partial net_k}] o_q(1-o_q)$$

Now, let $E_k = -\frac{\partial E_d}{\partial net_k}$. It follows that:

$$\Delta w_{pq} = c\sum_k w_{qk} E_k o_q(1-o_q) o_p$$



Note: $E_k = \delta_k$

Error Computation

Let p and q be two units connected to each other in a feedforward neural network, such that q follows p. Let δ_q denote the error at q.

If q is in the **output** layer,

$$\delta_q = (t_q - o_q)o_q(1 - o_q)$$

If q is in a **hidden** layer,

$$\delta_q = (\sum_{k \in follows(q)} w_{qk}\delta_k)o_q(1 - o_q)$$

It follows that:

$$\Delta w_{pq} = c\delta_q o_p$$

Note that $o_q(1 - o_q)$ is the derivative – dampens where "certain", enhances where "uncertain"

Also note that o_p is the output of node p – we often refer to it as z_p

The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      Forwards phase:

      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_{j} a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \tag{4.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{4.11}$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

Section 4.2.1
In your textbook

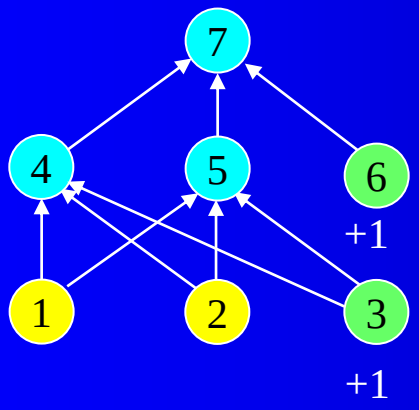
A Few Differences:
Textbook uses $(y_k - t_k)$
slides use $(T_k - Z_k)$

Textbook uses
$w_{ij} = w_{ij}$ - change in w
Slides use
$w_{ij} = w_{ij}$ + change in w

Follow slides for HW

Backpropagation Learning Example

Assume the following 2-2-1 MLP has all weights initialized to .5. Assume a learning rate of 1. Show the updated weights after training on the pattern .9 .6 -> 0. Show all net values, activations, outputs, and errors. Nodes 1 and 2 (input nodes) and 3 and 6 (bias inputs) are just placeholder nodes and do not pass their values through a sigmoid.



$$Z_j = f(net_j) = \frac{1}{1 + e^{-net_j}}$$

$$f'(net_j) = Z_j(1 - Z_j)$$

$$\Delta w_{ij} = C d_j Z_i$$

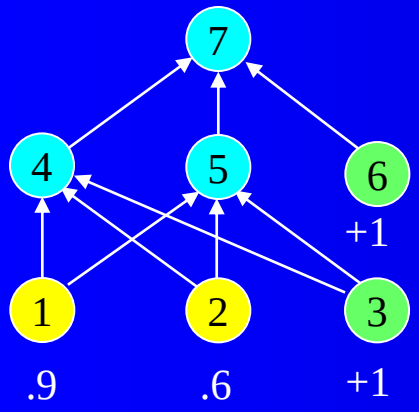
$$d_j = (T_j - Z_j) f'(net_j) \qquad \text{[Output Node]}$$

$$d_j = \sum_k (d_k w_{jk}) f'(net_j) \qquad \text{[Hidden Node]}$$

Backpropagation Learning Example

$$Z_j = f(net_j) = \frac{1}{1 + e^{-net_j}}$$

$$f'(net_j) = Z_j(1 - Z_j)$$



$$Dw_{ij} = Cd_jZ_i$$

$$d_j = (T_j - Z_j)f'(net_j) \quad \text{[Output Node]}$$

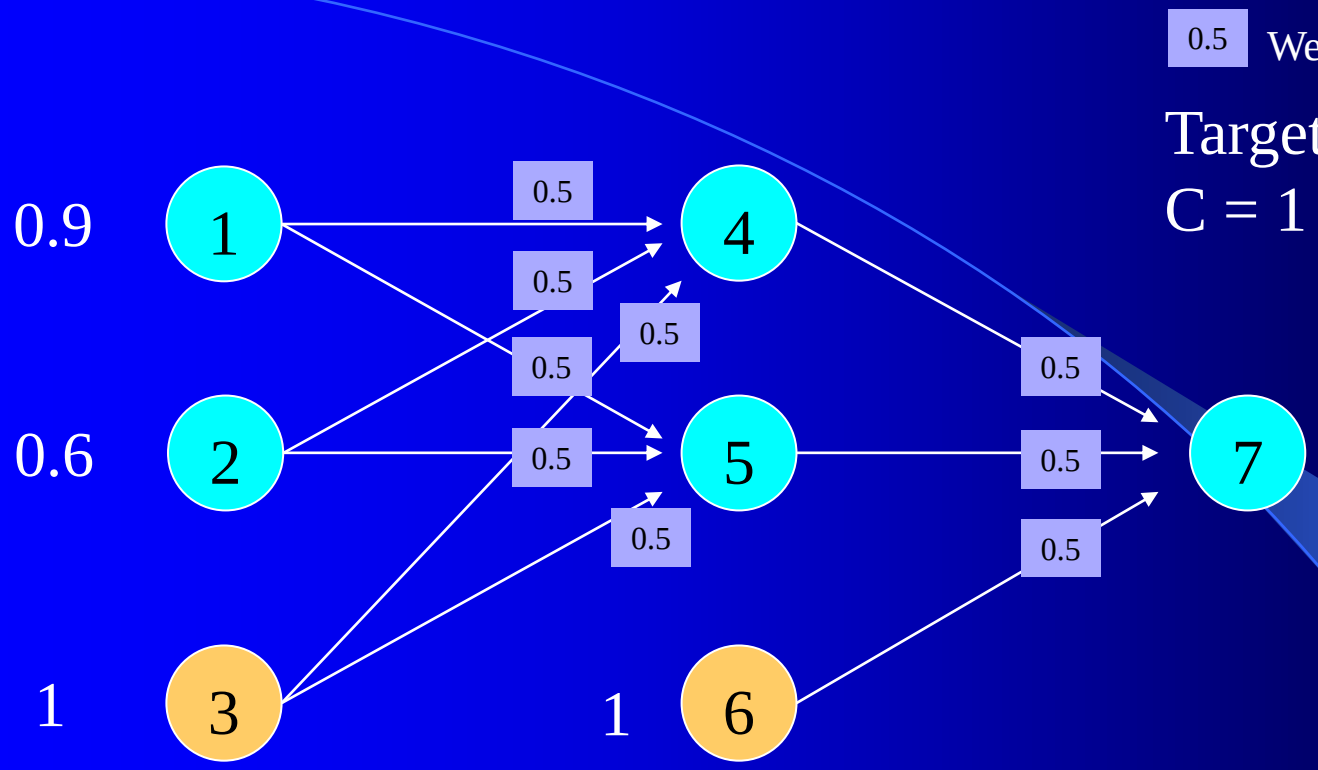
$$d_j = \sum_k (d_k w_{jk}) f'(net_j) \quad \text{[Hidden Node]}$$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$
$net_5 = 1.25$
$z_4 = 1/(1 + e^{-1.25}) = .777$
$z_5 = .777$
$net_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$
$z_7 = 1/(1 + e^{-1.277}) = .782$

$\delta_7 = (0 - .782) * .782 * (1 - .782) = -.133$
$\delta_4 = (-.133 * .5) * .777 * (1 - .777) = -.0115$
$\delta_5 = -.0115$

$w_{47} = .5 + (1 * -.133 * .777) = .3964$
$w_{57} = .3964$
$w_{67} = .5 + (1 * -.133 * 1) = .3667$

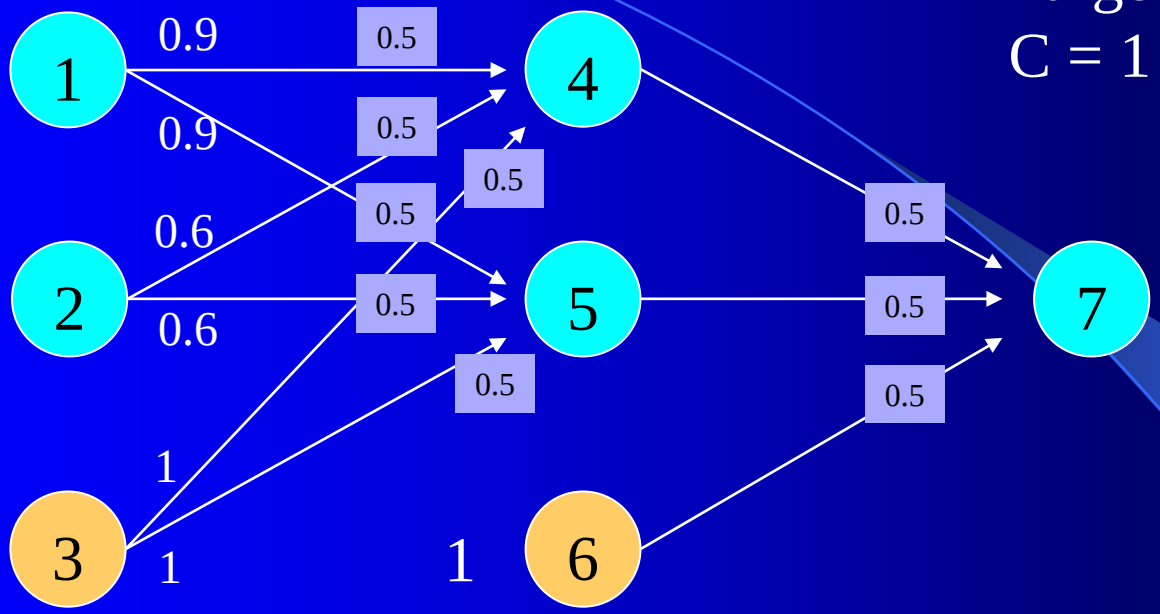
$w_{14} = .5 + (1 * -. 0115 * .9) = . 4896$
$w_{15} = .4896$
$w_{24} = .5 + (1 * -. 0115 * .6) = . 4931$
$w_{25} = .4931$
$w_{34} = .5 + (1 * -. 0115 * 1) = .4885$
$w_{35} = .4885$


0.5 Weights
Target = 0
C = 1 # Learning Rate
0.9
0.6
1
1
1
2
3
4
5
6
7
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5

0.5 Weights

Target = 0

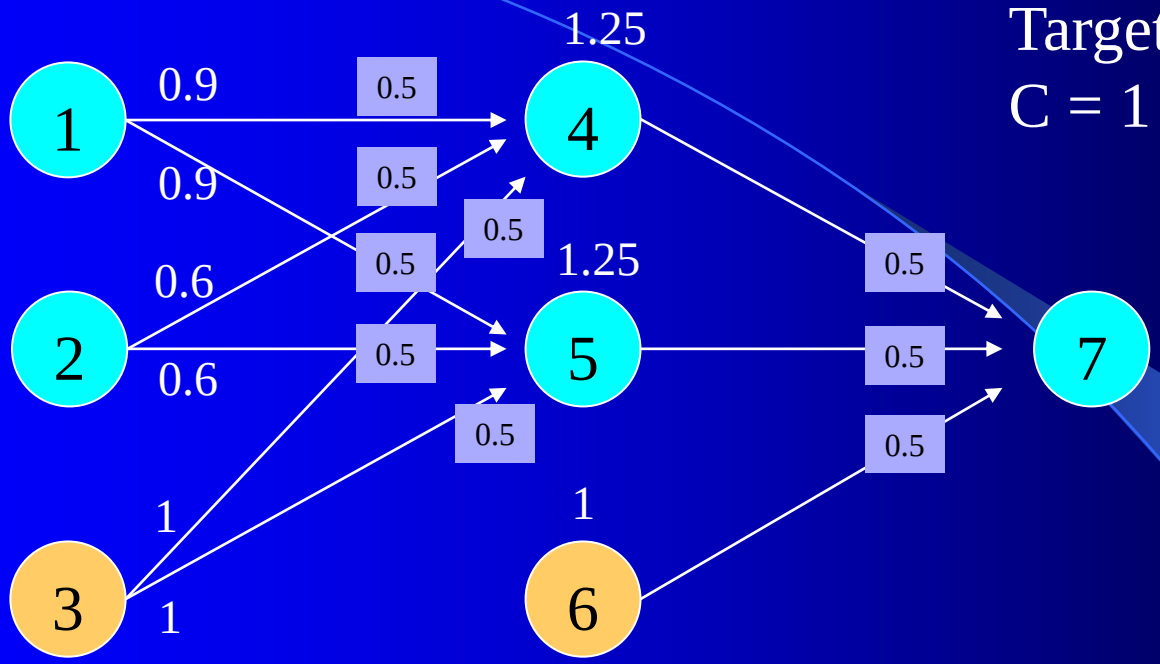
C = 1 # Learning Rate



$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

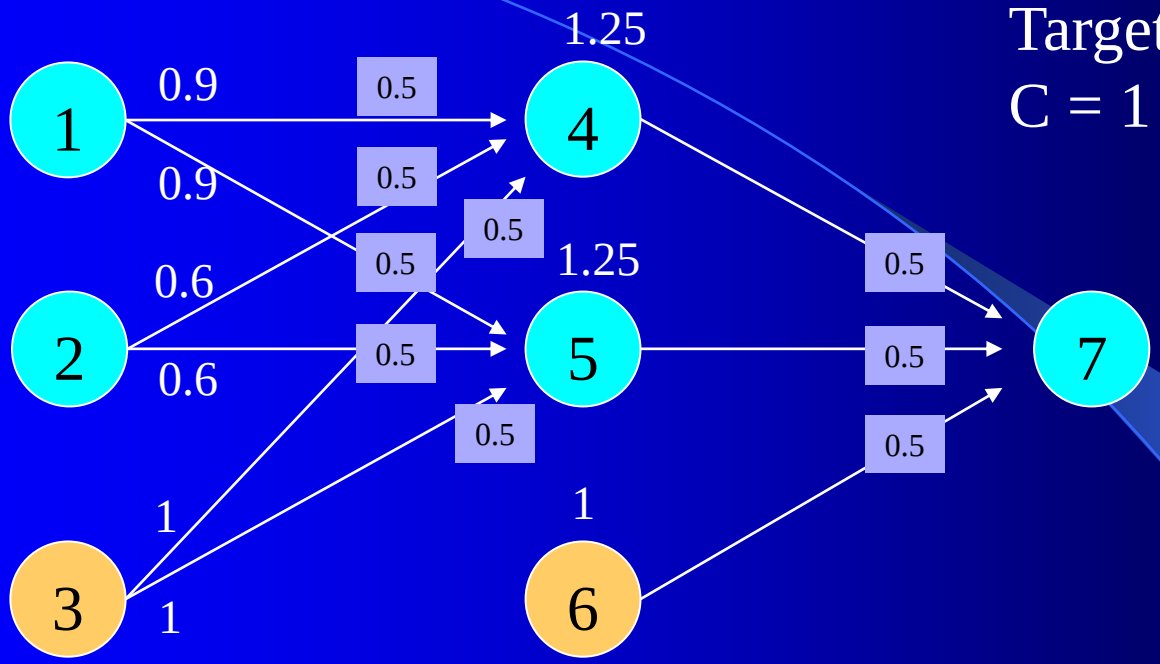
$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

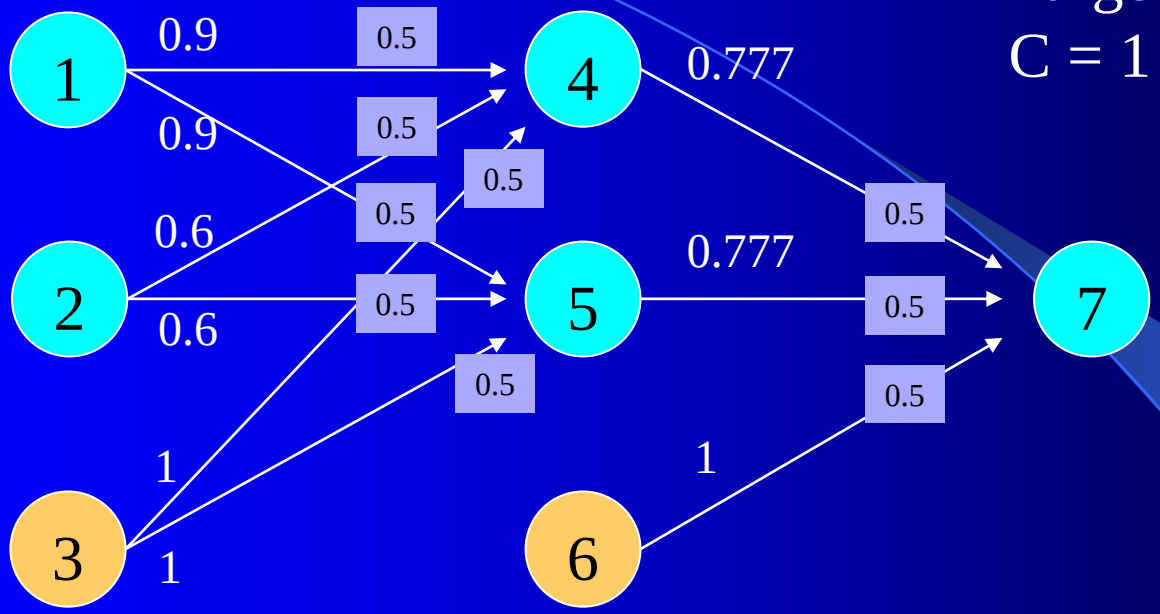
$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$

$$z_i = \frac{1}{(1 + e^{-neti})}$$



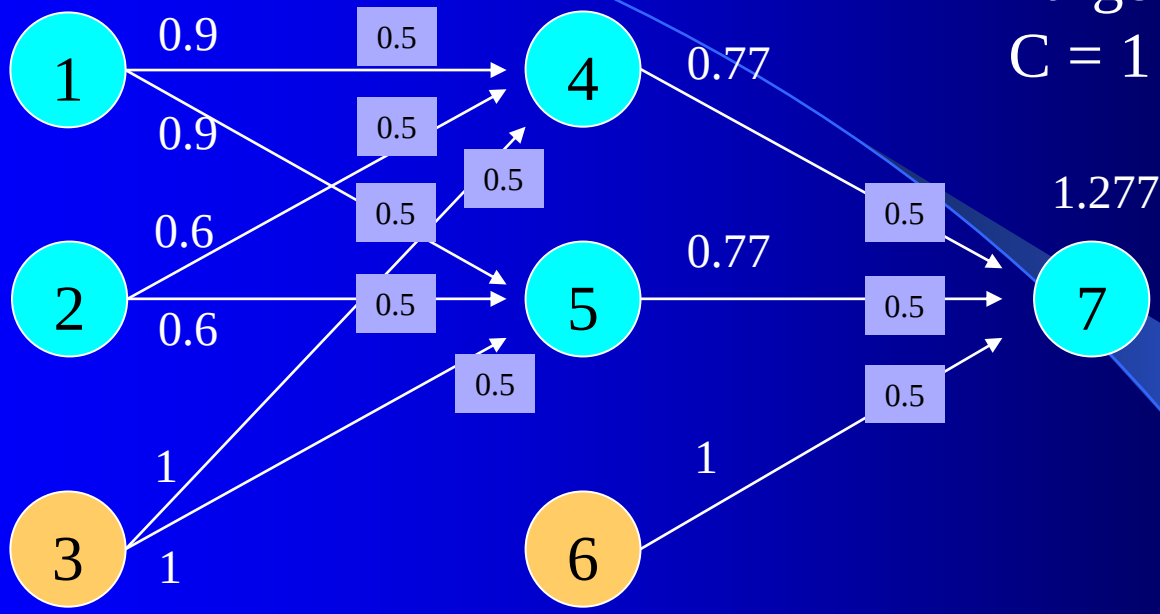
$net_q = \sum x_i w_{ij}$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$



$net_q = \sum x_i w_{ij}$

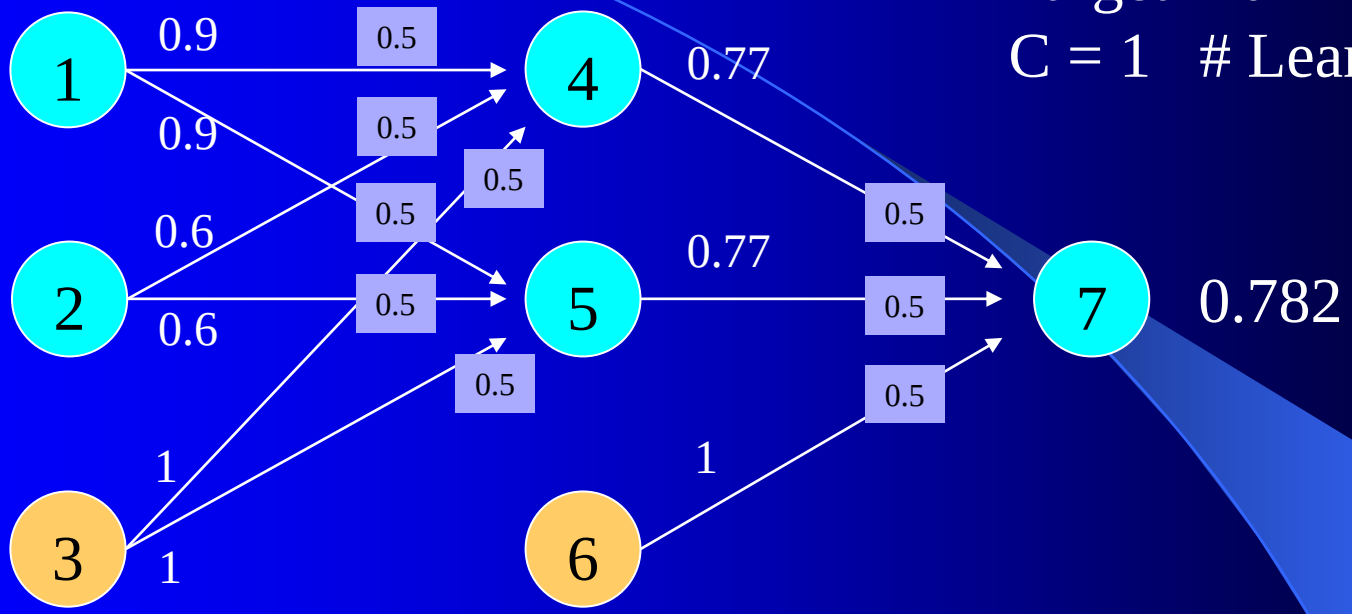
$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$

$net_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$



$net_q = \sum x_i w_{ij}$

$$z_i = \frac{1}{(1 + e^{-neti})}$$

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

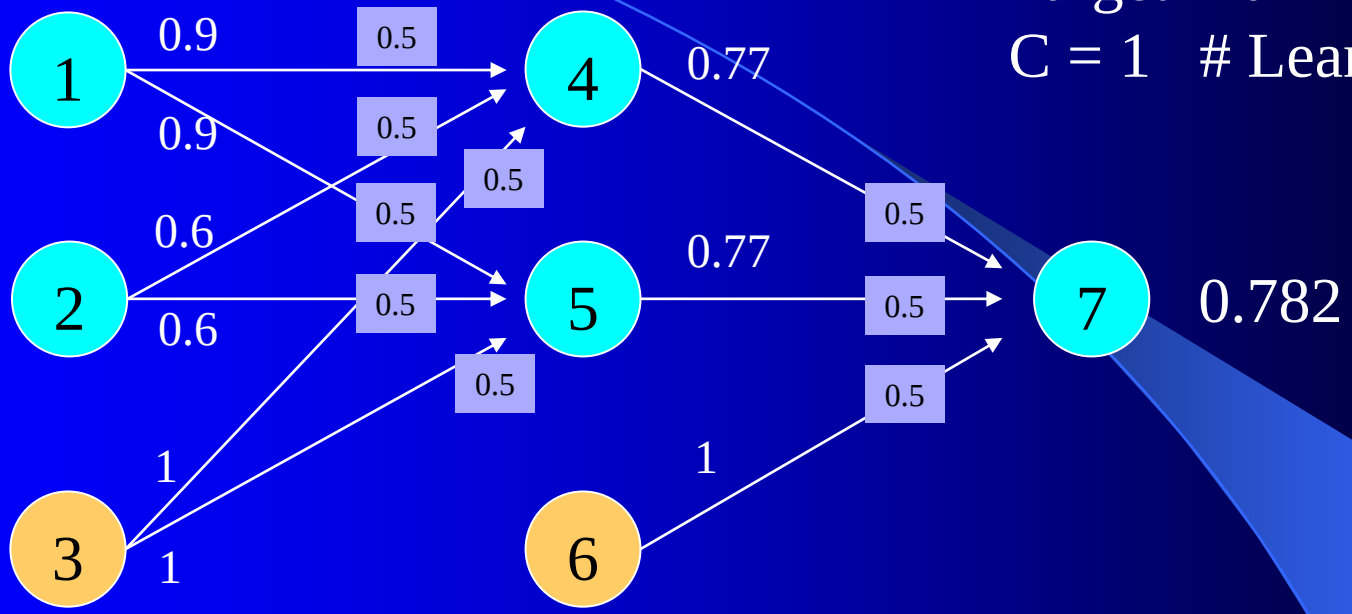
$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$

$net_7 = .777 * .5 + .777 * .5 + 1 * .5 = 1.277$

$z_7 = 1/(1 + e^{-1.277}) = .782$

Calculate δ_j



$$Dw_{ij} = C d_j Z_i$$

$$d_j = (T_j - Z_j) f'(net_j) \quad \text{[Output Node]}$$

$$d_j = \sum_k (d_k w_{jk}) f'(net_j) \quad \text{[Hidden Node]}$$

$$f'(net_j) = Z_j (1 - Z_j)$$

Output node – only one

$\delta_7 = (T_7 - Z_7) * Z_7 * (1 - Z_7)$

$\delta_7 = (0 - .782) * .782 * (1 - .782) = -.133$

Hidden nodes – only have node 7 following

$\delta_j = \Sigma(\delta_j w_{jk}) * Z_j * (1 - Z_j)$

$\delta_4 = (-.133 * .5) * .777 * (1 - .777) = -.0115$

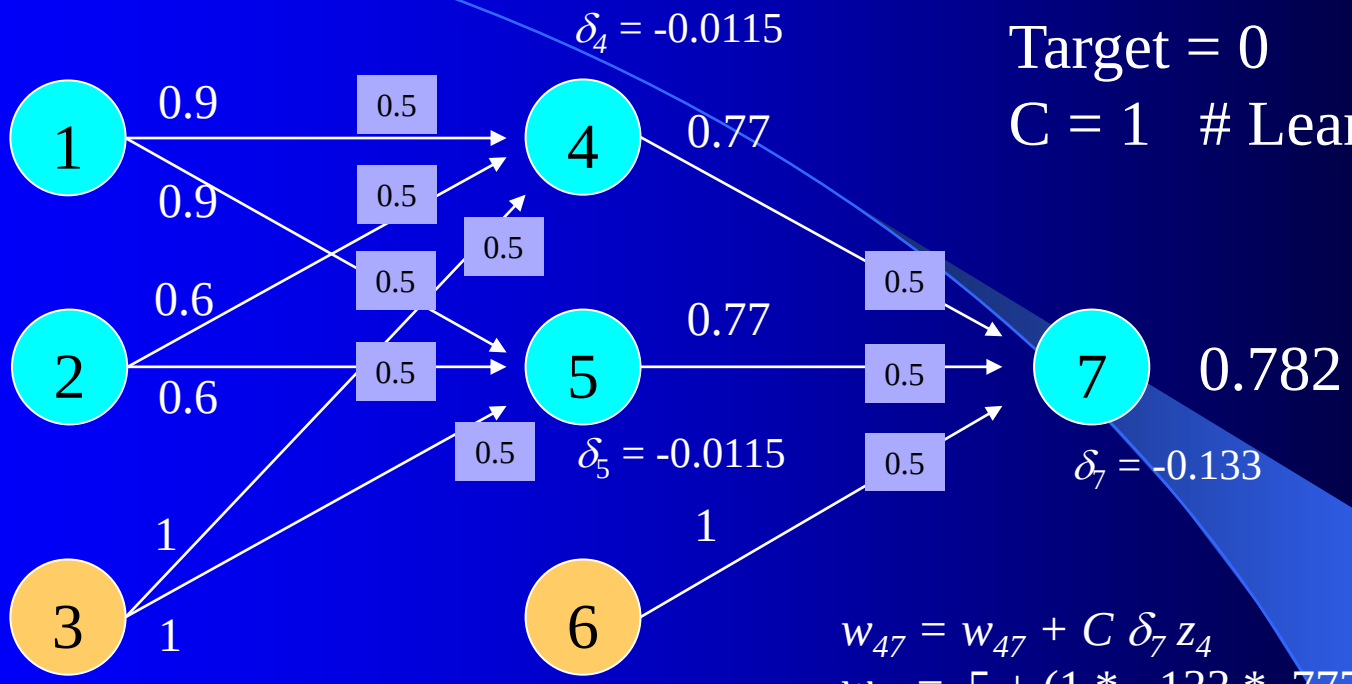
$\delta_5 = -.0115$

Calculate new weights

0.5 Weights

Target = 0

C = 1 # Learning Rate



$$w_{ij} = w_{ij} + C\,\delta_j\,z_i$$

$w_{47} = w_{47} + C\,\delta_7\,z_4$

w_{47} = .5 + (1 * -.133 * .777) = .3964

w_{57} = .3964

w_{67} = .5 + (1 * -.133 * 1) = .3667

w_{14} = .5 + (1 * -. 0115 * .9) = . 4896

w_{15} = .4896

w_{24} = .5 + (1 * -. 0115 * .6) = . 4931

w_{25} = .4931

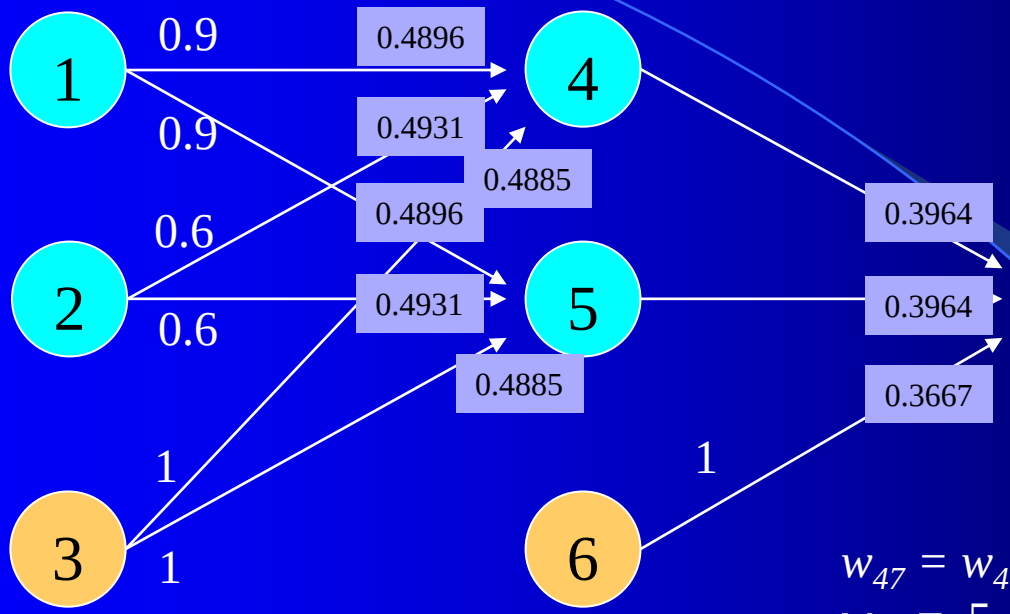
w_{34} = .5 + (1 * -. 0115 * 1) = .4885

w_{35} = .4885

0.5 Weights

Target = 0

C = 1 # Learning Rate



0.782

$$w_{ij} = w_{ij} + C\,\delta_j\,z_i$$

$w_{47} = w_{47} + C\,\delta_7\,z_4$

$w_{47} = .5 + (1 * -.133 * .777) = .3964$

$w_{57} = .3964$

$w_{67} = .5 + (1 * -.133 * 1) = .3667$

$w_{14} = .5 + (1 * -.0115 * .9) = .4896$

$w_{15} = .4896$

$w_{24} = .5 + (1 * -.0115 * .6) = .4931$

$w_{25} = .4931$

$w_{34} = .5 + (1 * -.0115 * 1) = .4885$

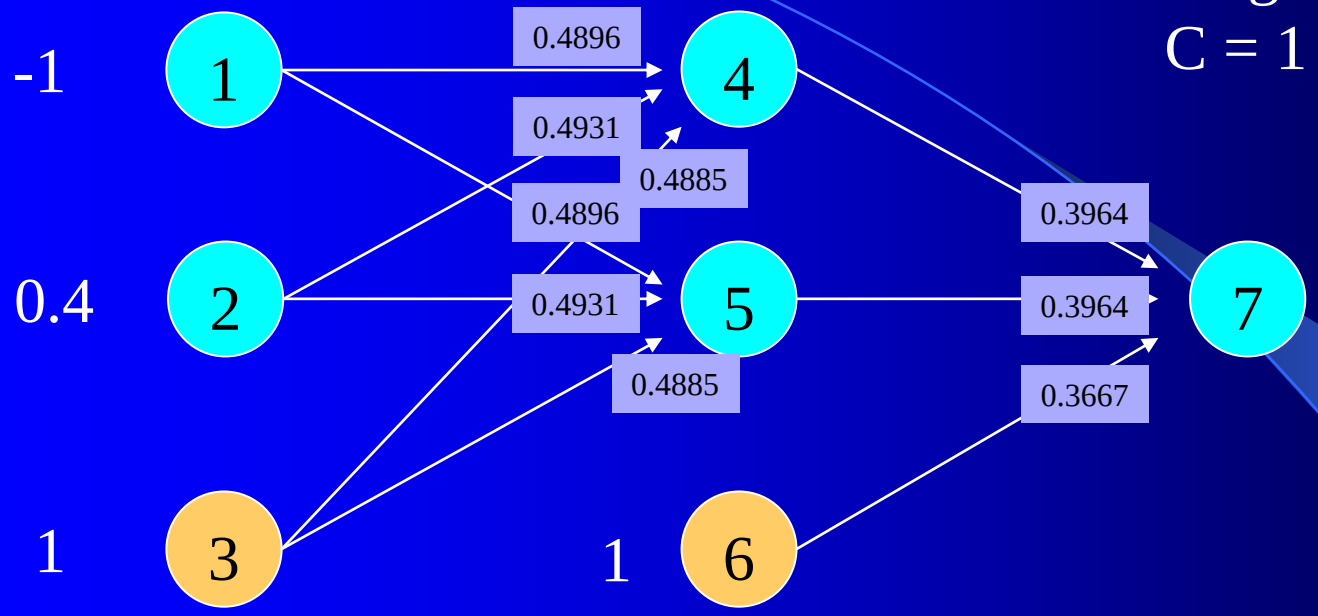
$w_{35} = .4885$

Homework

0.5 Weights

Target = 0.2

C = 1 # Learning Rate



Evaluate the network using the new weights on the new input

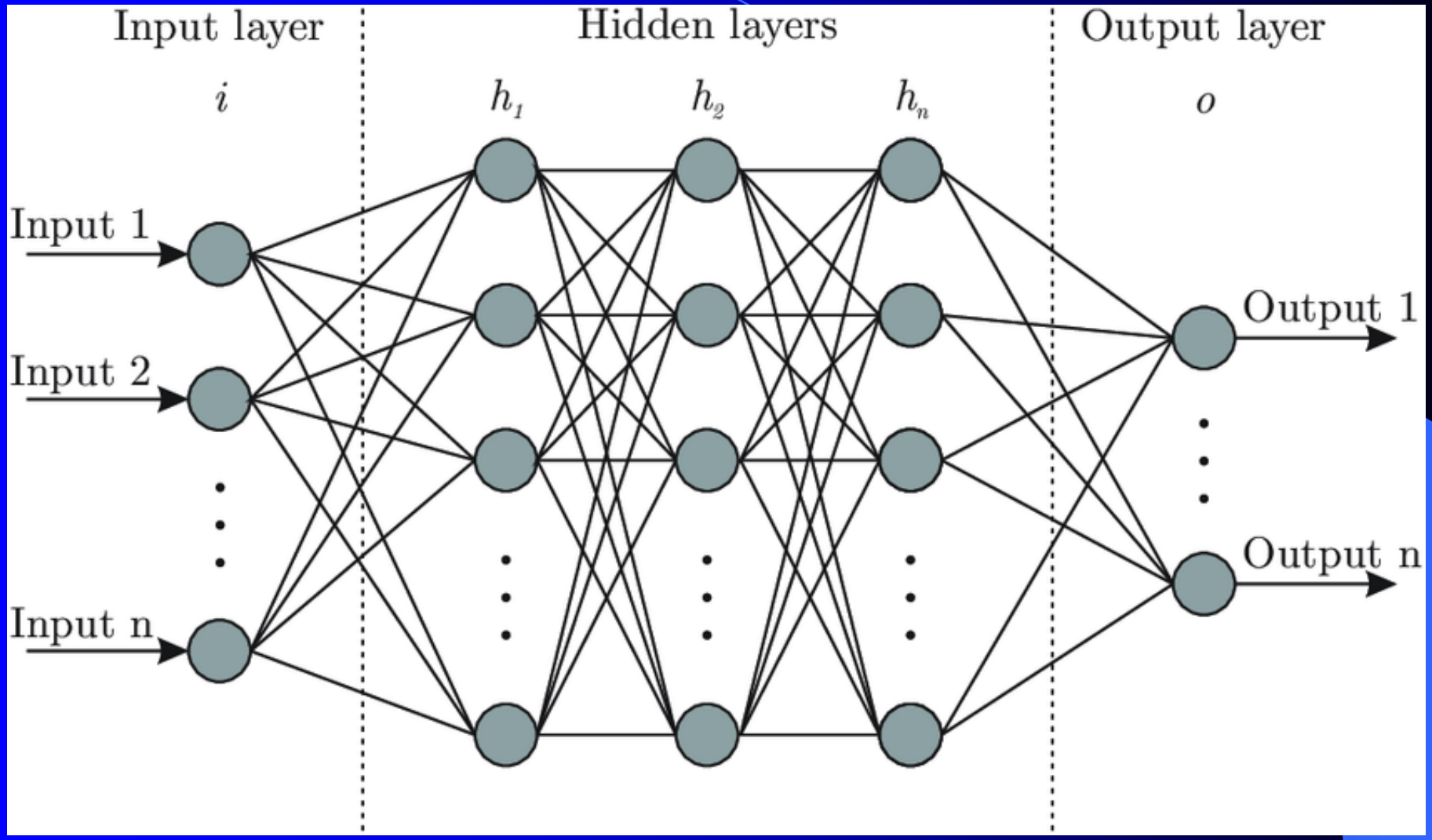
Using those values, back propagate the error and calculate new weights for the network

Backprop Homework

1. For your homework, update the weights for a second pattern -1 .4 -> .2. Continue using the updated weights shown on the previous slide. Show your work like we did on the previous slide.
2. Then go to the link below: Neural Network Playground using the *tensorflow* tool and play around with the BP simulation. Try different training sets, layers, inputs, etc. and get a feel for what the nodes are doing. You do not have to hand anything in for this part.

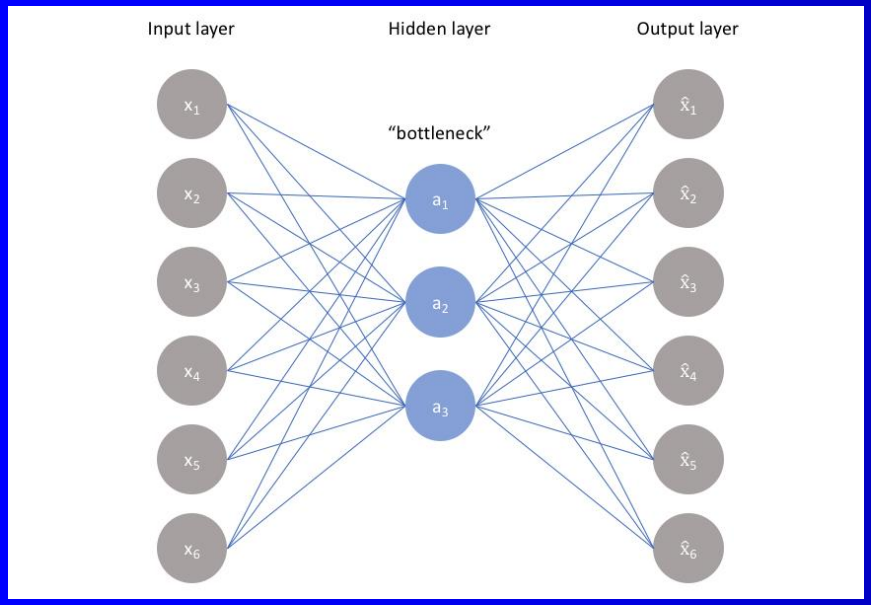
- http://playground.tensorflow.org/

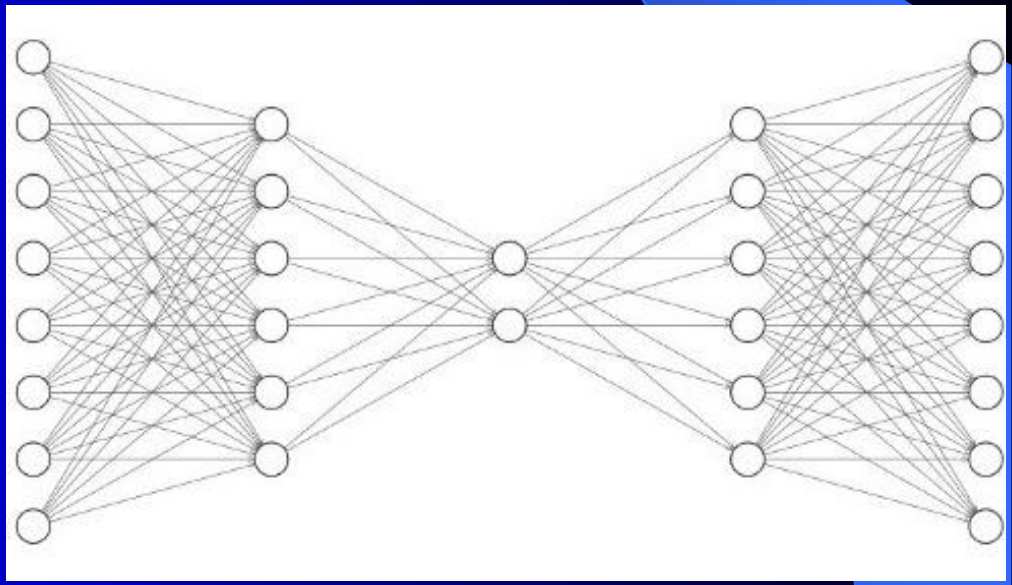
Multi-layer Perceptron (MLP) Topology

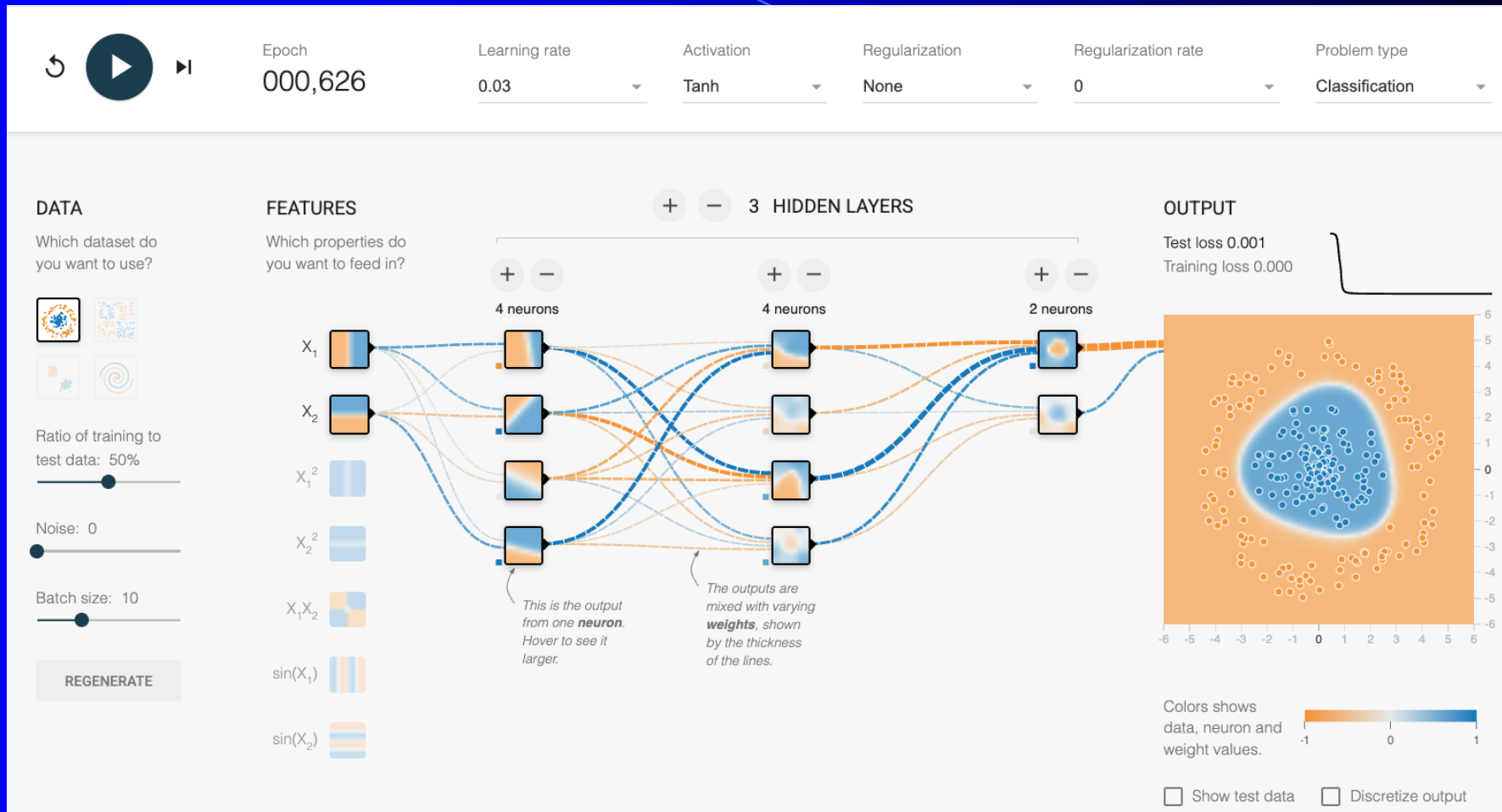


What are the Hidden Nodes Doing?

- Higher order features vs 1st order features (perceptron/Us)
  - The real power of machine learning (exponential # of variations)
- Hidden nodes discover new *higher order* features which are fed into subsequent layers
- Compression






Epoch
000,626
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
3 HIDDEN LAYERS
4 neurons
4 neurons
2 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
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.
OUTPUT
Test loss 0.001
Training loss 0.000
Colors shows data, neuron and weight values.
Show test data
Discretize output

MLP for Classification & Regression

- The output of an MLP is a continuous value
- How we use the output determines what we are doing

- Classification uses the output to determine a class
  - We must then interpret the output values to classes

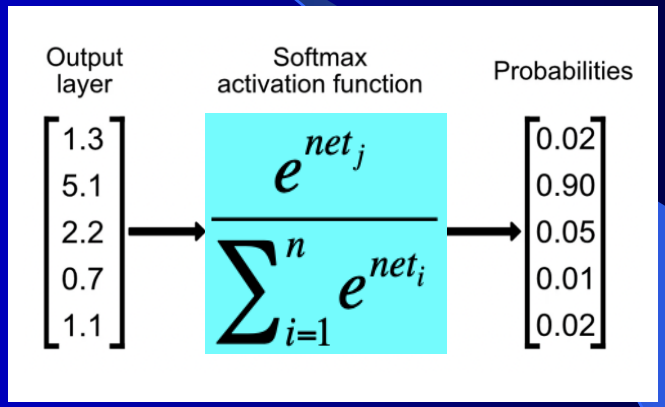
- Regression is trying to produce the output value that is close to the value we want

- The way we calculate error should correlate with our task

Softmax Output Layer Activation

- For *classification* problems we generally use the *softmax* activation function, just at the output layer
- Softmax (softens) 1 of *n* targets to mimic a probability vector for the output nodes

$$f(net_j) = \frac{e^{net_j}}{\sum_{i=1}^{n} e^{net_i}}$$



- If there were 3 output nodes with net values 0, .5, and 1 then the outputs for each node would be
  - 1/5.37, 1.65/5.37, 2.72/5.37 = .18, .31, .51
  - sums to 1 and can be considered as probability estimates
- All the hidden nodes still do a standard activation function such as logistic, hyperbolic tangent, or ReLU
- Sklearn automatically uses softmax at the output layer for MLP classification, and you choose the activation function for hidden nodes

Regression with MLP/BP

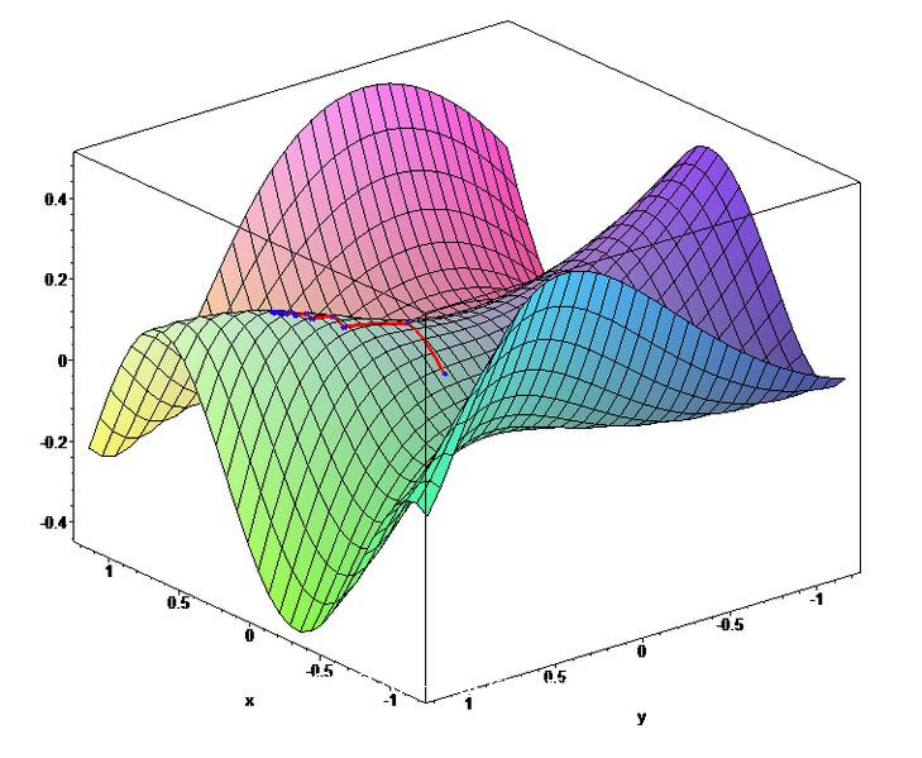
- For regression in MLPs we use the sum-squared error (L2) loss function which seeks the maximum likelihood hypothesis under the assumption that the training data can be modeled by normally distributed noise added to the target function value. More natural for regression than for classification.
- Output nodes use a linear activation (i.e. identity function which just passes the *net* value through). This naturally supports unconstrained regression.
  - Don't typically normalize output
- The output error is still $(t - z)\, f'(net)$, but since $f'(net)$ is 1 for the linear activation, the output error is just $(target - output)$
- Hidden nodes still use a non-linear activation function (such as logistic) with the standard $f'(net)$
- This is how sklearn always does MLP regression

Local Minima

- Most algorithms which have difficulties with simple tasks get much worse with more complex tasks
- Good news with MLPs
- Many dimensions make for many descent options
- Local minima more common with simple/toy problems, rare with larger problems and larger nets
- Even if there are occasional minima problems, could simply train multiple times and pick the best
- Some algorithms add noise to the updates to escape minima

Local Minima and Neural Networks

- Neural Network can get stuck in local minima for small networks, but for most large networks (many weights), local minima rarely occur in practice
- This is because with so many dimensions of weights it is unlikely that we are in a minima in every dimension simultaneously – almost always a way down



Creating a Neural Network

- Look at example code

BP Lab

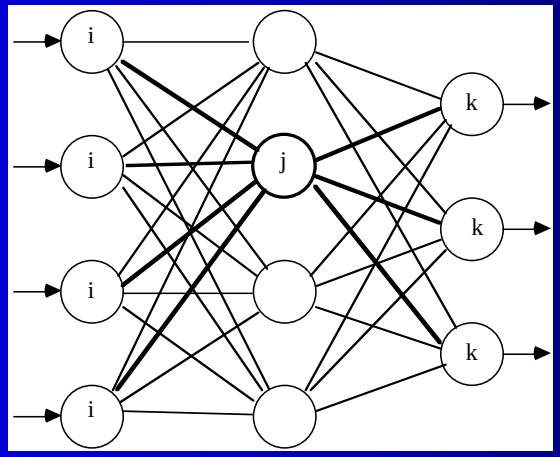
- Go over Lab together

MLPClassifier Hyperparameters

- Number of layers and layer sizes
- Activation Function
- Regularization
- Momentum
- Solver
- Weight initialization

Number of Hidden Nodes

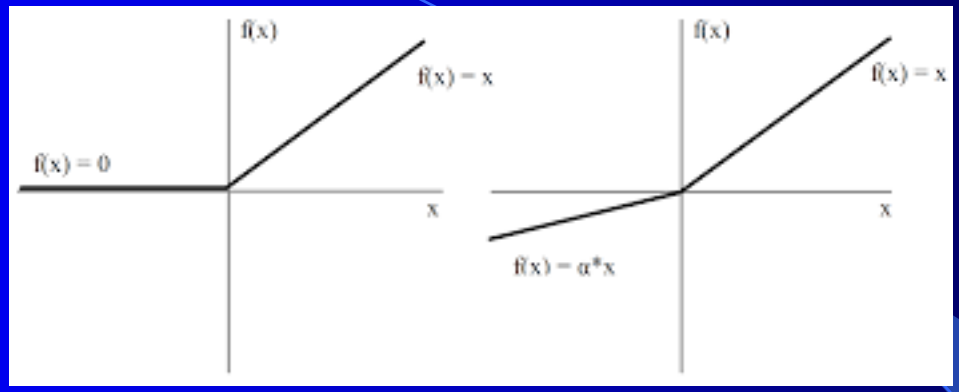
- How many needed is a function of how hard the task is
- Common to use one fully connected hidden layer. Initial number could be ~2*n* hidden nodes where *n* is the number of inputs.
- In practice we train with a small number of hidden nodes, then keep doubling, etc. until no more significant improvement on test sets
  - Too few will underfit
  - Too many nodes can make learning slower and could overfit
    - Having somewhat too many hidden nodes is preferable if using reasonable regularization; avoids underfit and should ignore unneeded nodes
- Each output and hidden node should have its own bias weight



Activation Function

- MLPClassifier has 4 options
  - identity
  - logistic (sigmoid)
  - tanh (hyperbolic tangent)
  - relu (rectified linear unit)
- logistic (sigmoid) was the choice in the 1990s, tanh became the choice in the early 2000s
  - tanh can perform better than sigmoid
  - Both can suffer from vanishing gradient
- relu is most common now
  - Allows for discontinuities
- New ones are being used more – leaky relu

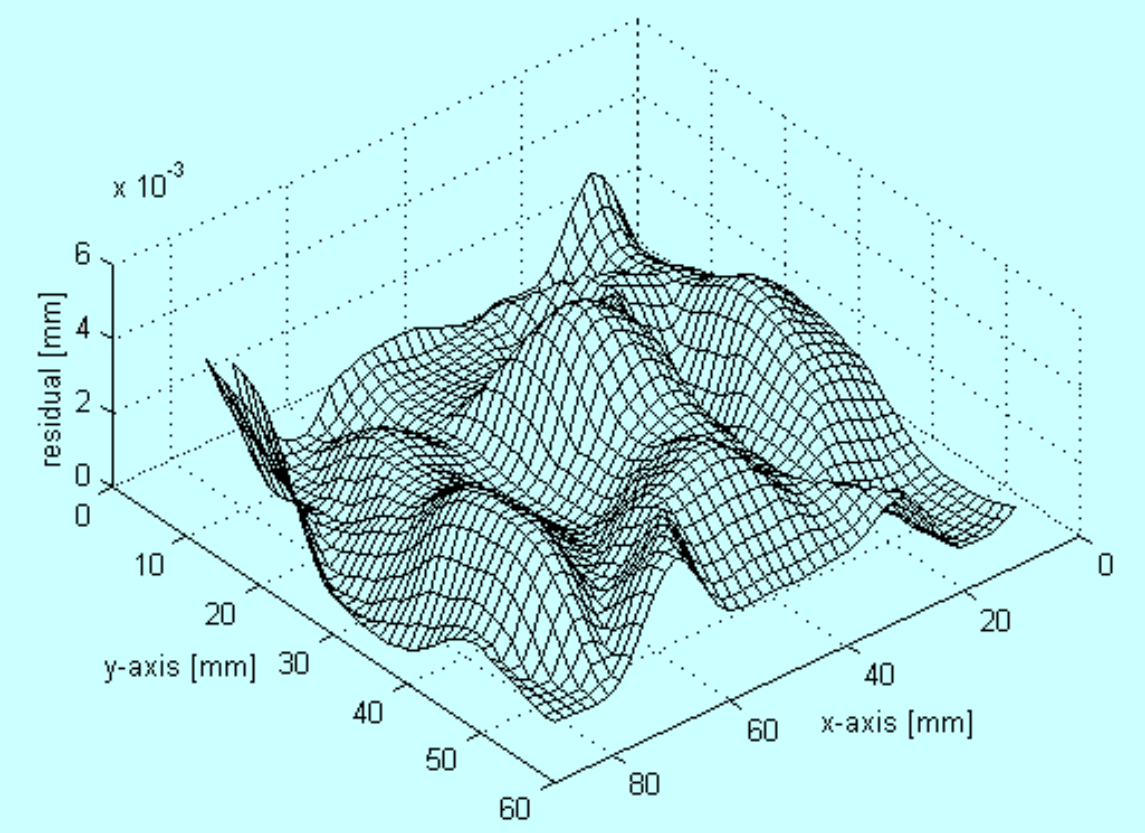
Rectified Linear Units



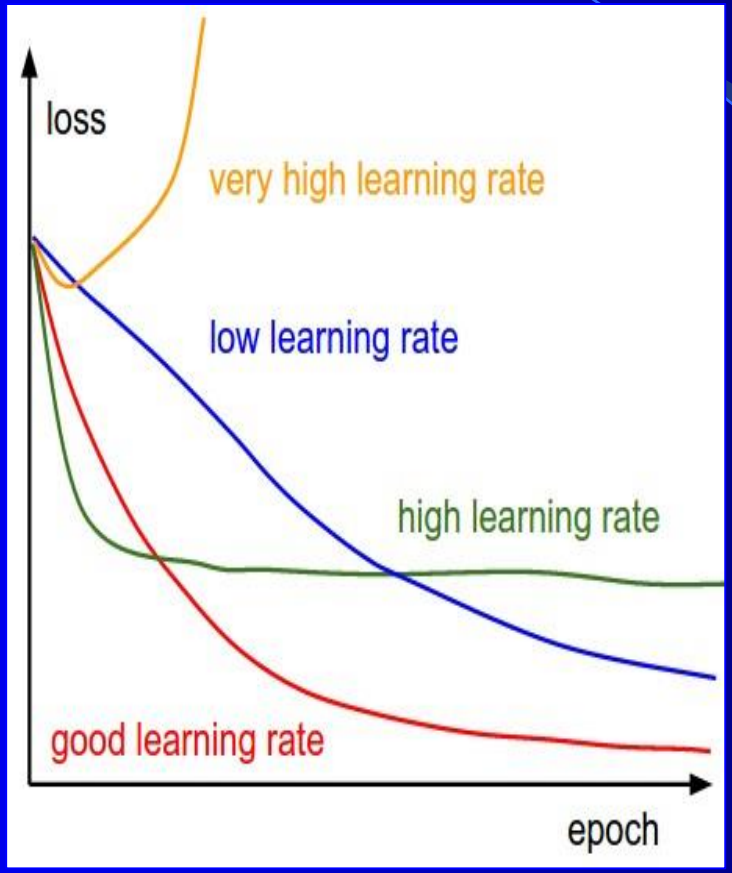
- BP can work with any differentiable non-linear activation function (e.g. sine)
- *ReLU* is common these days especially with deep learning: $f(x) = \text{Max}(0,x)$
  - More efficient computation: Only comparison, addition and multiplication
  - $f'(net)$ is 0 or constant, just fold into learning rate
- Leaky ReLU $f(x) = x$ if $x > 0$, else ax, where $0 \leq a <= 1$, so for net < 0 the derivate is not 0 and can do some learning (does not "die").
  - Lots of other variations
- Sparse activation: For example, in a randomly initialized networks, only about 50% of hidden units are activated (having a non-zero output)
- Not differentiable but we just "cheat" and include the discontinuity point with either side of the linear part of the ReLU function – piecewise linear

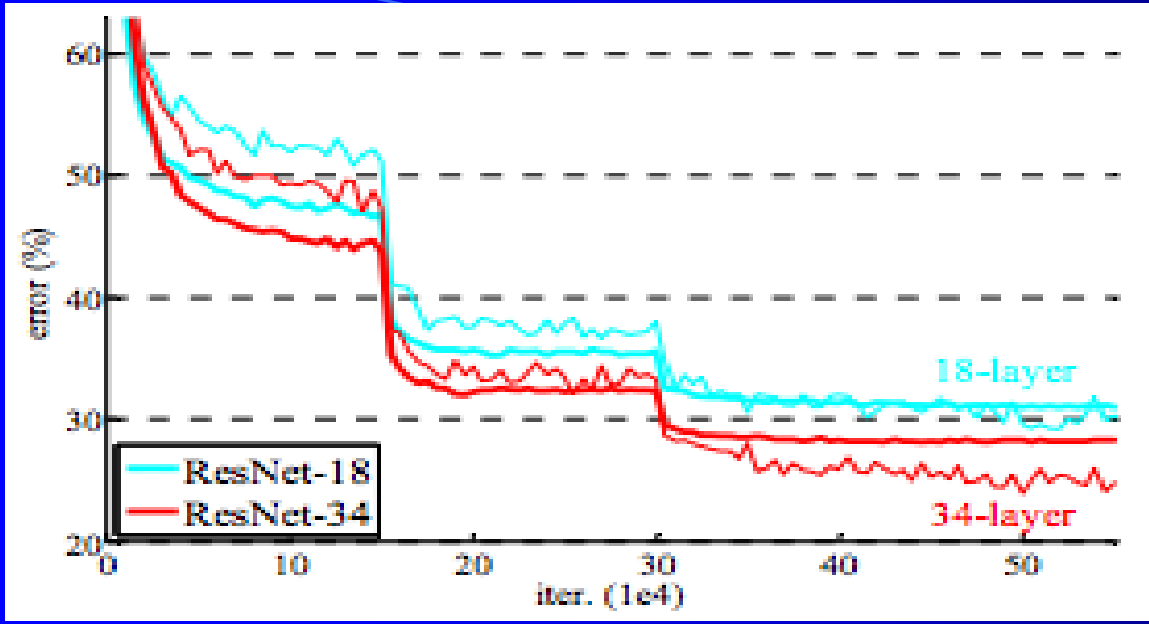
Learning Rate

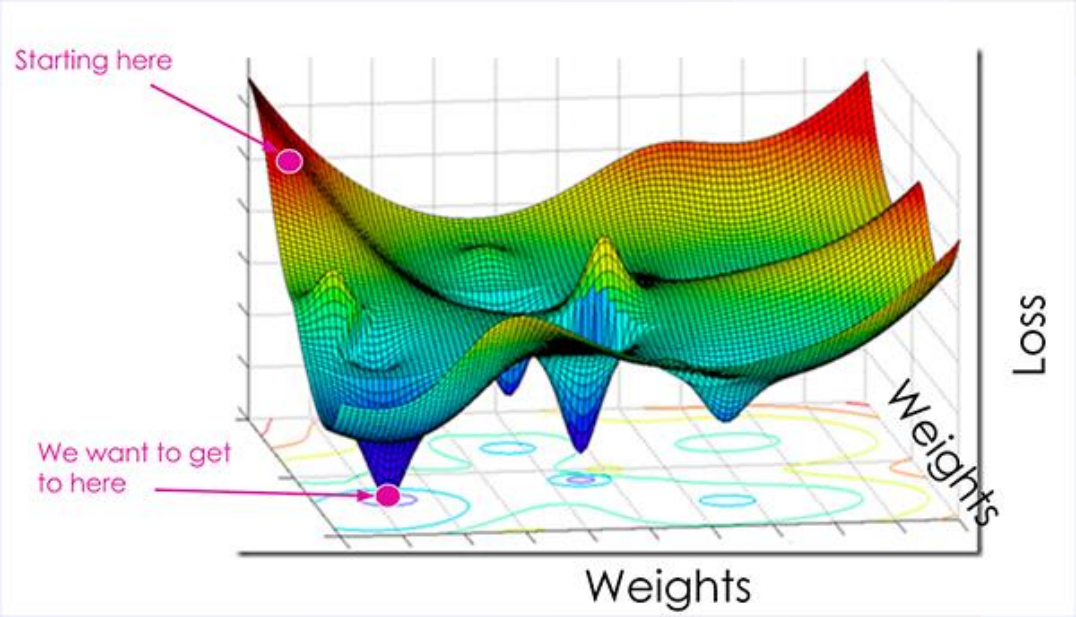
- Learning Rate - Relatively small (.01 - .5 common), if too large BP will not converge or be less accurate, if too small it is just slower with no accuracy improvement as it gets even smaller
- Gradient – computed only where you are, avoid too big of jumps



Learning Rate




60
50
40
30
20
error (%)
0
10
20
30
40
50
iter. (1e4)
ResNet-18
ResNet-34
18-layer
34-layer


Starting here
We want to get
to here
Loss
Weights
Weights

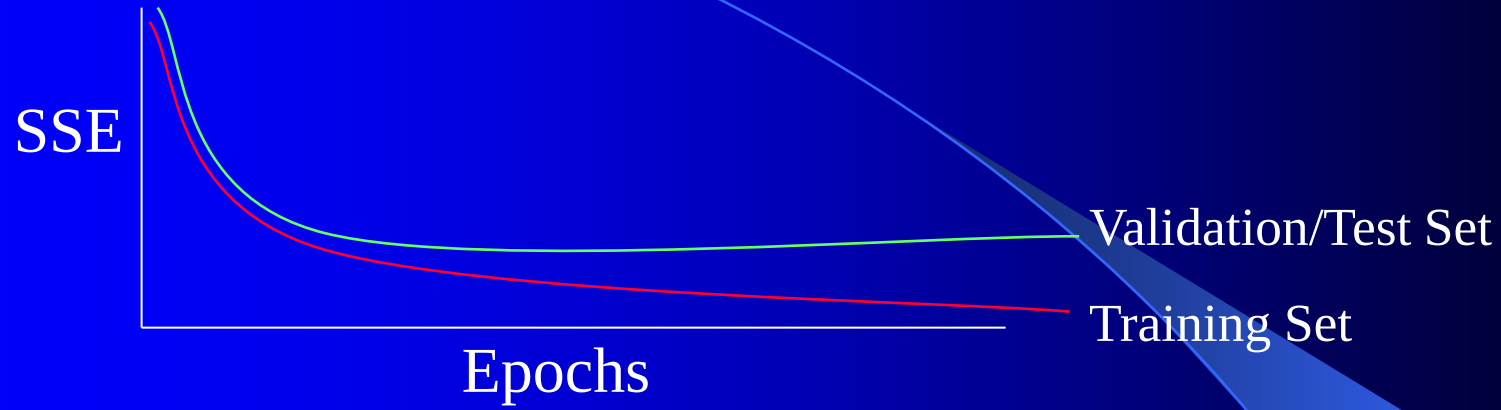
Inductive Bias & Weight Initialization

- When weights are large, saturates the node and/or net
- Node Saturation - Avoid early, but all right later
  - With small weights all nodes have low confidence at first (linear range)
  - When saturated (confident), the output changes only slightly as the net changes. An incorrect output node can still have low error.
  - Start with weights close to 0. Once nodes saturate, hopefully have learned correctly, others still looking for their niche.
  - Saturated error even when wrong? – Multiple training set loss drops
  - Don't start with equal weights (can get stuck), random small Gaussian/uniform with 0 mean
- Inductive Bias
  - Start with simple net (small weights, initially linear changes)
  - Gradually build a more complex until accurate enough without getting too complex

Regularization in MLP

- Remember: Regularization refers to a method to ovoid overfitting
- Methods:
  - Data
    - Get more data
    - Get better data – more representative
  - Early stopping
  - Model methods – keep the model simple
  - Loss regularization – L1 and L2
  - Dropout

Stopping Criteria and Overfit Avoidance



- More Training Data (vs. overtraining - One epoch in the limit)
- Validation Set - save weights which do best job so far on the validation set. Keep training for enough epochs to be sure that no more improvement will occur (e.g. once you have trained *m* epochs with no further improvement, stop and use the best weights so far, or retrain with all data).
  - Note: If using *N*-way CV with a validation set, do *n* runs with 1 of *n* data partitions as a validation set. Save the number of training updates for each run. To get a final model you can train on all the data and stop after the average number of training updates.

Validation Set

- Often you will use a validation set (separate from the training or test set) for stopping criteria, etc.
- In these cases you should take the validation set out of the training set
- For example, you might use the random test set method to randomly break the original data set into 80% training set and 20% test set. Independent and subsequent to the above split you would take *n*% (10-20%) of the training set to be a validation set for that particular training run.
- You will usually shuffle the weight training part of your training set for each epoch, but you use the same unchanged validation set throughout the entire training
  - Never use an instance in the VS which has been used to train weights
  - Sklearn does all this for you by just setting the early_stopping = True

Backpropagation Regularization

- How to avoid overfit – Keep the model simple
  - Keep decision surfaces smooth
  - Smaller overall weight values lead to simpler models with less overfit
- Early stopping with validation set is a common approach to avoid overfitting (since weights don't have time to get too big)
- Could make complexity an explicit part of the loss function
  - Then we don’t need early stopping (though sometimes one is better than the other and we can even do both simultaneously)
- Regularization approach: Model (h) selection
  - Minimize $F(h) = Error(h) + \lambda \cdot Complexity(h)$
  - Tradeoff accuracy vs complexity
- Two common approaches
  - Lasso (L1 regularization)
  - Ridge (L2 regularization)

L1 (Lasso) Regularization

- Standard BP update is based on the derivative of the loss function with respect to the weights. We can add a model complexity term directly to the loss function such as:
  - $L(\boldsymbol{w}) = \text{Error}(\boldsymbol{w}) + \lambda\Sigma|w_i|$
  - λ is a hyperparameter which controls how much we value model simplicity vs training set accuracy
  - Gradient of $L(\boldsymbol{w})$: Gradient of $\text{Error}(\boldsymbol{w}) + \lambda$
  - To make it gradient descent we negate the Gradient: $(-\nabla\text{error}(\boldsymbol{w}) - \lambda)$
    - This is also called weight decay
    - Gradient of Error is just equations we have used if $\text{Error}(\boldsymbol{w})$ is TSS, but may differ for other error functions
- Common values for lambda are 0, .001, .01, .03, etc.
- Weights that really should be significant stay large enough, but weights just being nudged by a few data instances go to 0

L2 (Ridge) Regularization

- $L(\boldsymbol{w}) = \text{Error}(\boldsymbol{w}) + \lambda \Sigma w_i^2$
- -Gradient of $L(\boldsymbol{w})$: -Gradient of $\text{Error}(\boldsymbol{w}) - 2\lambda w_i$
- Regularization portion of weight update is scaled by weight value (fold 2 into λ)
  - Decreases change when weight small (<0), otherwise increases
  - λ is % of weight change, .03 means 3% of the weight is decayed each time
- L1 vs L2 Regularization
  - L1 drives many weights all the way to 0 (Sparse representation and feature reduction)
  - L1 more robust to large weights (outliers), while L2 makes larger decay with large weights
  - L1 leads to simpler models, but L2 often more accurate with more complex problems which require a bit more complexity

Momentum

- Simple speed-up modification (type of adaptive learning rate)

$$\Delta w_{ij}(t) = C\delta_j z_i + \alpha \Delta w_{ij}(t\text{-}1)$$

- Save $\Delta w_{ij}(t)$ for each weight to be used as next $\Delta w_{ij}(t\text{-}1)$
- A large momentum (e.g. 0.9) will mean that the update is strongly influenced by the previous update, whereas a modest momentum (0.2) will mean very little influence.
- Weight update maintains momentum in the direction it has been going
  - Significant speed-up, common value $\alpha \approx .9$
  - Effectively increases learning rate in areas where the gradient is consistently the same sign. (Which is a common approach in adaptive learning rate methods which we will mention later).
  - Can overshoot
- These types of terms make the algorithm less pure in terms of gradient descent.

Adaptive Learning Rate Approaches

- Momentum is a type of adaptive learning rate mechanism

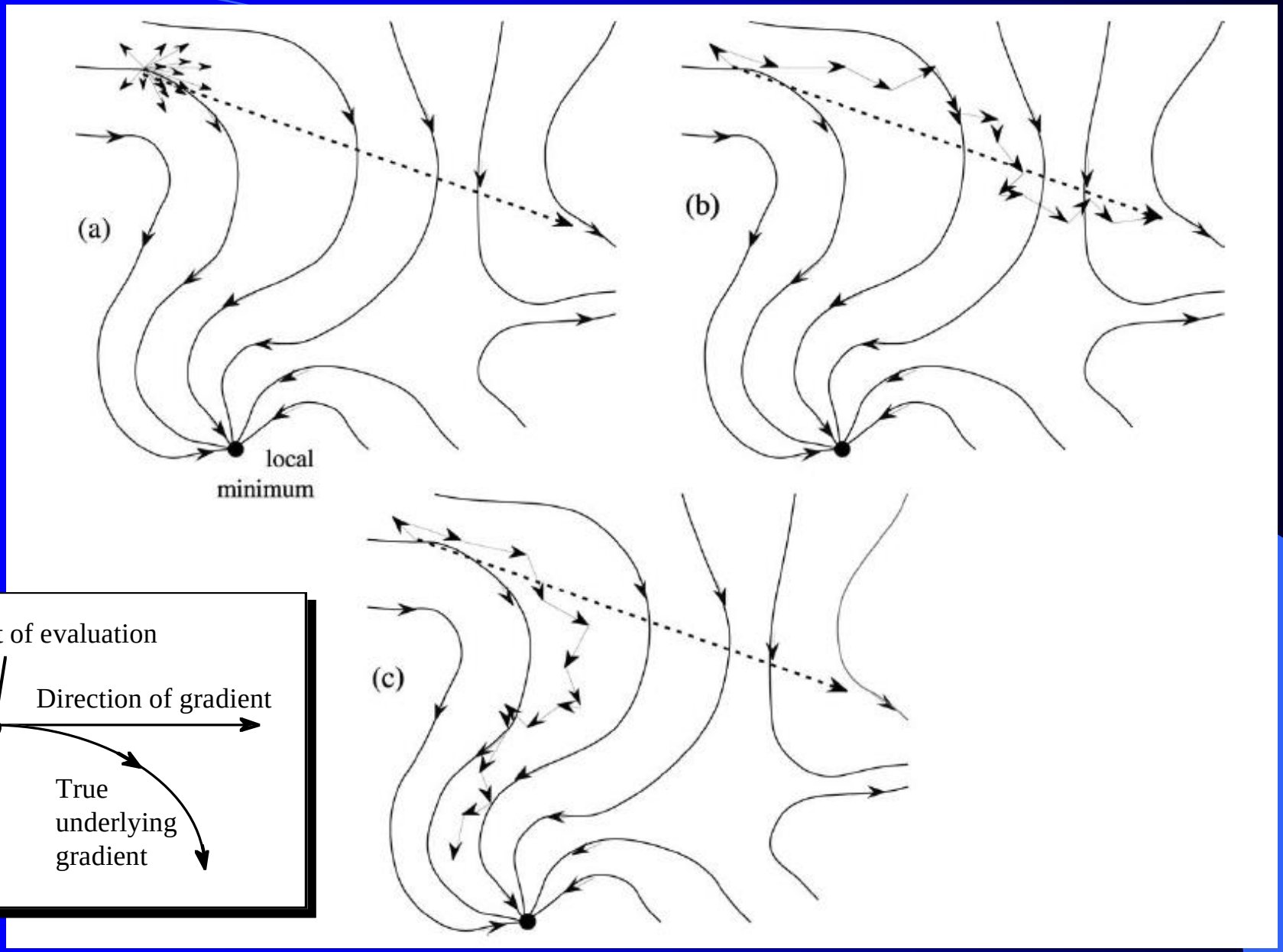
$$\Delta w_{ij}(t) = C\delta_j z_i + \alpha \Delta w_{ij}(t\text{-}1)$$

- Adaptive Learning rate methods
  - Start LR small
  - As long as weight change is in the same direction, increase a bit (e.g. scalar multiply > 1, etc.)
  - If weight change changes directions (i.e. sign change) reset LR to small, could also backtrack for that step, or …

- Use mini-batch rather than single instance for better gradient estimate – *Sometimes* helpful if SGD variation more sensitive to bad gradient, and also for some parallel (GPU) implementations.

Batch Update

- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each batch
- Batch update gives a better estimated direction of the gradient for the data set, while on-line could do some weight updates in directions quite different from the average gradient of the entire data set
  - Based on noisy instances and also just that specific instances will not usually be at the average gradient
- Proper approach? -
  - Most assumed batch more appropriate, but batch/on-line a non-critical decision with similar results
- We show that batch is less efficient
  - Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. **16**, no. 10, pp. 1429-1452, 2003


(a)
local
minimum
(b)
(c)
Point of evaluation
Direction of gradient
True
underlying
gradient

Speed up variations of SGD

- Standard Momentum
  - Note that these approaches already do an averaging of gradient, also making mini-batch less critical
- Nesterov Momentum – Calculate point you would go to if using normal momentum. Then, compute gradient at that point. Do normal update using *that* gradient and momentum.
- Rprop – Resilient BP, if gradient sign inverts, decrease it's individual LR, else increase it – common goal is faster in the flats, variants that backtrack a step, etc.
- Adagrad – Scale LRs inversely proportional to sqrt(sum(historical values))
- RMSprop – Adagrad but uses exponentially weighted moving average, older updates basically forgotten
- Adam (Adaptive moments) –Momentum terms on both gradient and squared gradient (uncentered variance) (1^{st} and 2^{nd} moments) – updates based on a moving average of both - Popular

Weight Initialization

- Not available by default in MLPClassifier
- Typically small random values within a range defined by layer sizes

- Can override, but… advanced

```
# new class
class MLPClassifierOverride(MLPClassifier):
# Overriding _init_coef method
def _init_coef(self, fan_in, fan_out):
    if self.activation == 'logistic':
        init_bound = np.sqrt(2. / (fan_in + fan_out))
    elif self.activation in ('identity', 'tanh', 'relu'):
        init_bound = np.sqrt(6. / (fan_in + fan_out))
    else:
        raise ValueError("Unknown activation function %s" %
                         self.activation)
    coef_init = ### place your initial values for coef_init here

    intercept_init = ### place your initial values for intercept_init here

    return coef_init, intercept_init
```

Hyperparameter Selection

- LR (e.g. .1)
- Momentum – (.5 … .99)
- Connectivity: fully connected between layers
- Number of hidden nodes: Problem dependent
  - Less hidden nodes NOT a great approach because may underfit
- Number of layers: 1 (common) or 2 hidden layers which are usually sufficient for good results, attenuation makes learning very slow – modern deep learning approaches show significant improvement using many layers and many hidden nodes
- Manual CV can be used to set hyperparameters: trial and error runs
  - Often sequential: find one hyperparameter value with others held constant, freeze it, find next hyperparameter, etc.
- Hyperparameters could be learned by the learning algorithm in which case you must take care to not overfit the training data – always use a cross-validation technique to measure hyperparameters

- Weight decay
  - Similar to L1/L2 regularization but with different parameter
- Adjusted Error functions
  - Use a different loss function that may be more appropriate for your data
- Dropout
  - Randomly drop out nodes during training
  - Forces the network to rely on other nodes
  - Mostly used in deep learning

Loss and Multiclass Classification

- Loss in binary classification is straightforward
  - t - z
- Loss in multiclass classification
  - We need a single number that represents a distance
  - But we have probability distributions
- How do we calculate a distance?
- Information Theory

t	*z*
0	0.2
1	1
1	.9
1	.5
1	.1

Entropy

- Information theory metric that quantifies the number of bits required to transmit or encode an event.
  - Quantifies the *surprise* of an event
  - High probability events are less surprising – less bits needed
  - Low probability events are more surprising – more bits needed
- Calculated as
  - h(x) = - log(P(x)) - negative log because P(x) < 1
- For a set of instances
  - $H(X) = -\sum P(x)\log(P(x))$
- Cross Entropy builds on this idea and calculates the difference in entropy between two probability distributions

Cross-Entropy and Softmax

- Cross-entropy measures the difference (in entropy) between two distributions.
  - $H(P,Q) = -\sum P(x)\log(Q(x))$
  - P(x) becomes t_i
  - Log(P(x)) becomes $\log(z_i)$ or $\ln(z_i)$
- Assumes that the outputs are probability distributions

$$Loss_{CrossEntopy} = -\sum_{i=1}^{n} t_i \ln(z_i)$$

t	*z*	CE
0	0.01	0
1	1	0
1	.9	.11
1	.5	.69
1	.1	2.30
		3.1

- Use the cross entropy as the loss for output nodes
  - Replaces (t – z) and there is no *f ' (net)*
- The hidden layers still update as usual and include *f* '(*net*) for their activation function
- Sklearn always uses this approach for MLP classification

Debugging ML algorithms

- Debugging ML algorithms can be difficult
  - Unsure beforehand about what the results should be, differ for different tasks, data splits, initial random weights, hyperparameters, etc.
  - Adaptive algorithm can learn to compensate somewhat for bugs
  - Bugs in accuracy evaluation code common – false hopes!
- **Do a small example by hand (e.g. your homework) and make sure your algorithm gets the exact same results (and accuracy)
- Compare results with our supplied debug and LS examples
- Compare results (not code, etc.) with classmates
- Compare results with a published version of the algorithm (e.g. sklearn), won't be exact because of different training/test splits, etc.
  - Use Frederick Zarndt's thesis (or other publications) to get a ballpark feel of how well you should expect to do on different data sets. http://axon.cs.byu.edu/papers/Zarndt.thesis95.pdf

MLP/Backpropagation Summary

- Excellent Empirical results
- Scaling – The pleasant surprise
  - Local minima may be rare as problem and network complexity increase
- Most common neural network approach
  - Many other different styles of neural networks (RBF, Hopfield, etc.)
- Hyper-parameters usually handled by trial and error or search
- Many variants
  - Adaptive Parameters, Ontogenic (growing and pruning) learning algorithms
  - Many different learning algorithm approaches
  - Recurrent networks
  - Deep networks!
  - An active research area

APPENDIX

Localist vs. Distributed Representations

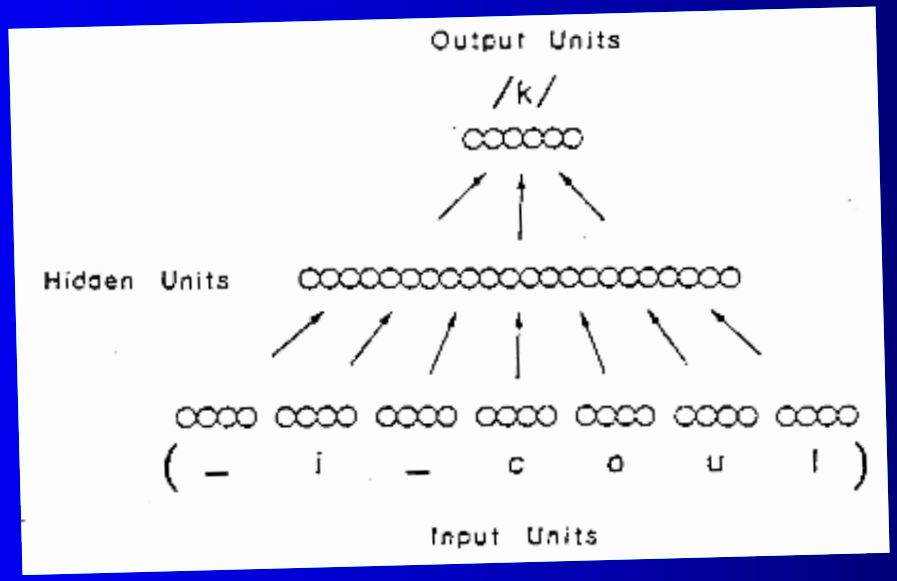
- Is Memory Localist ("grandmother cell") or distributed
- Output Nodes
  - One node for each class (classification) – "one-hot"
  - One or more graded nodes (classification or regression)
  - Distributed representation
- Input Nodes
  - Normalize real and ordered inputs
  - Nominal Inputs - Same options as above for output nodes
- Hidden nodes - Can potentially extract rules if localist representations are discovered. Difficult to pinpoint and interpret distributed representations.

Learning Variations

- Different activation functions - need only be differentiable
- Different objective functions
  - Cross-Entropy
  - Classification Based Learning
- Higher Order Algorithms - 2nd derivatives (Hessian Matrix)
  - Quickprop
  - Conjugate Gradient
  - Newton Methods
- Constructive Networks
  - Cascade Correlation
  - DMP (Dynamic Multi-layer Perceptrons)

Application Example - NetTalk

- One of first application attempts
- Train a neural network to read English aloud
- Input Layer - Localist representation of letters and punctuation
- Output layer - Distributed representation of phonemes
- 120 hidden units: 98% correct pronunciation
  - Note steady progression from simple to more complex sounds



Higher order "shortcut"

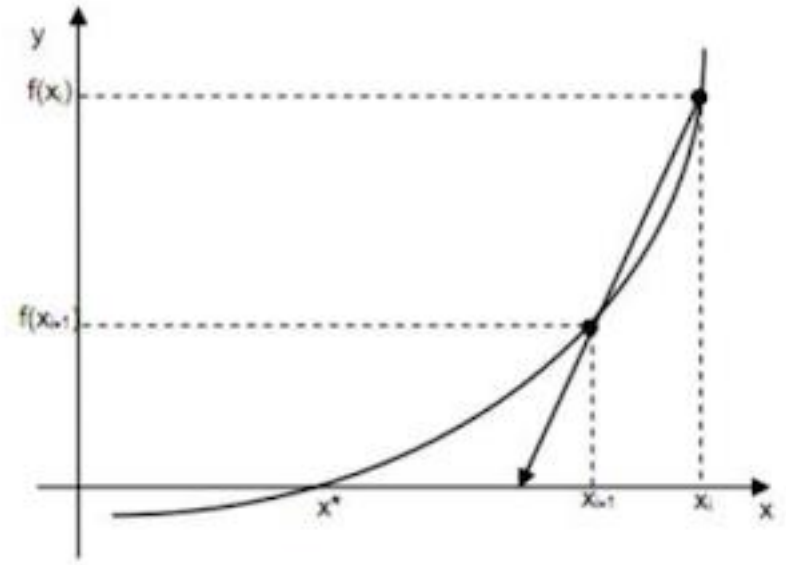
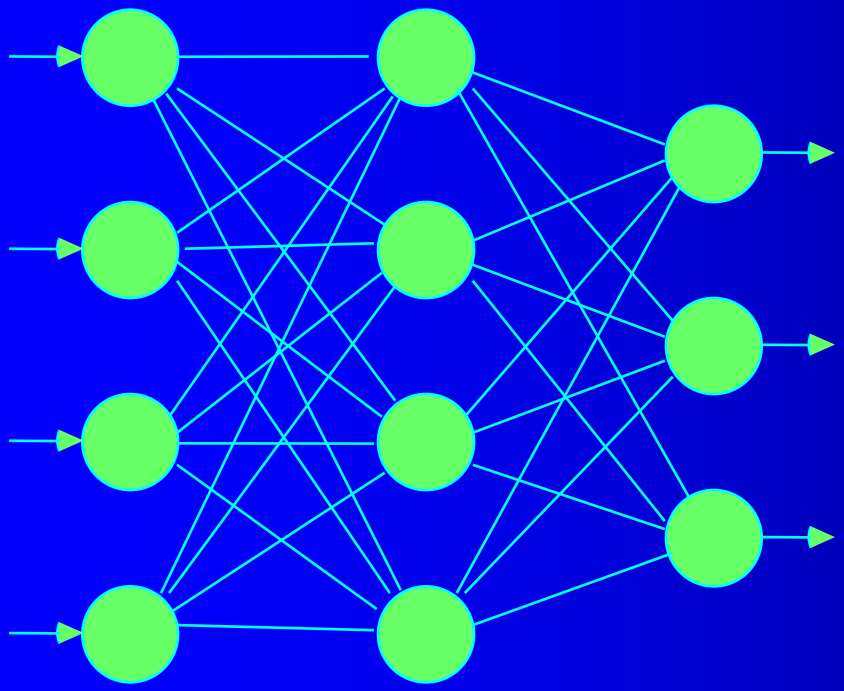


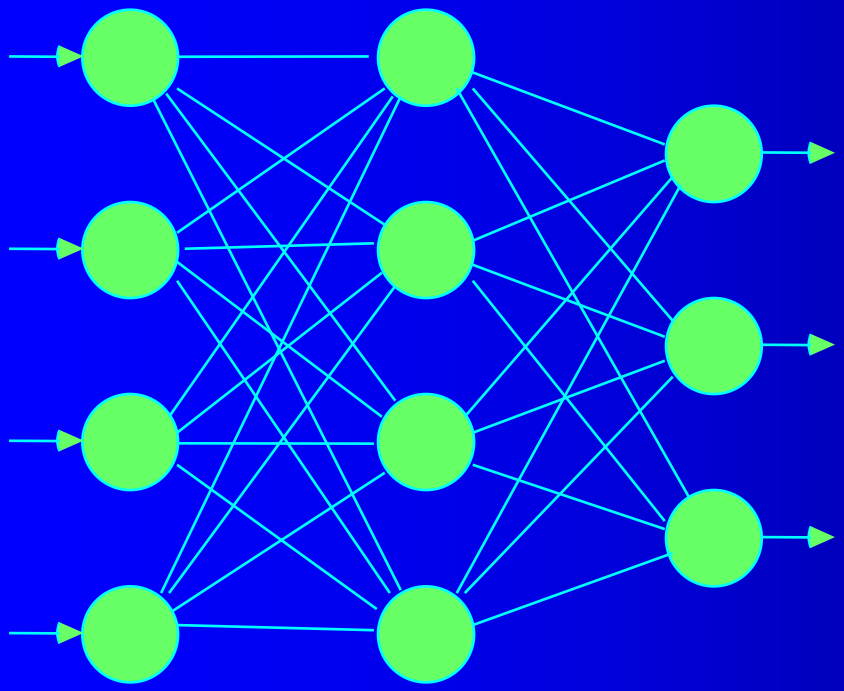
Fig. 6 from 'Metodos numéricos' book

Classification Based (CB) Learning



Target	Actual	BP Error	CB Error
1	.6	.4*$f'(net)$	0
0	.4	-.4*$f'(net)$	0
0	.3	-.3*$f'(net)$	0

Classification Based Errors



Target	Actual	BP Error	CB Error
1	.6	$.4*f'(net)$	.1
0	.7	$-.7*f'(net)$	-.1
0	.3	$-.3*f'(net)$	0

Results

- Standard BP: **97.8%**

Sample Output:

```
 0:    a    1.00
 1:    A    0.56
 2:   Ww    0.05
 3:  OoO    0.01
 4:    8    0.01
 5:    b    0.00
 6:    D    0.00
 7:    B    0.00
 8:    3    0.00
 9:   Vv    0.00
10:    T    0.00
11:   Cc    0.00
12:   Xx    0.00
13:   Yy    0.00
14:    E    0.00
15:    F    0.00
16:    4    0.00
17:    5    0.00
18:    7    0.00
19:   G6    0.00
20:   Jj    0.00
21:    Q    0.00
22:   Ss    0.00
23:   Zz    0.00
24:    2    0.00
25:    d    0.00
26:    e    0.00
27:    f    0.00
28:   g9    0.00
29:    q    0.00
30:    t    0.00
31:    N    0.00
32:    R    0.00
33:    H    0.00
34:   Mm    0.00
35:    H    0.00
36:  Iil    0.00
37:   Zz    0.00
38:   Pp    0.00
39:    1    0.00
```

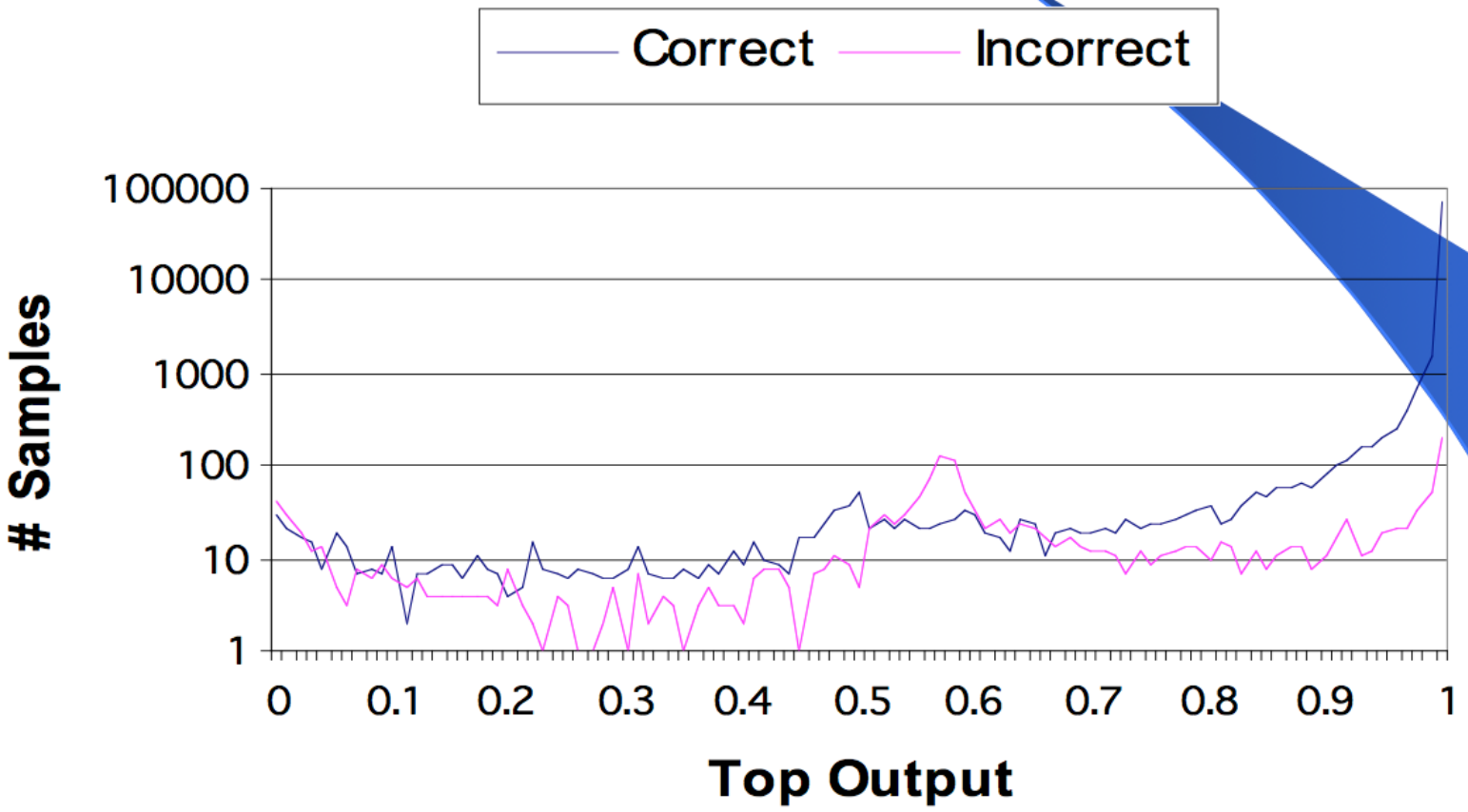
Results

- Classification Based Training:
  99.1%

```
 0:     A   0.71
 1:     a   0.53
 2:     N   0.44
 3:     R   0.44
 4:    Ss   0.42
 5:     H   0.39
 6:    Kk   0.38
 7:    Mm   0.36
 8:     B   0.35
 9:    Ww   0.34
10:     6   0.33
11:     3   0.32
12:     8   0.31
13:     n   0.31
14:     h   0.30
15:    Xx   0.29
16:   Iil   0.28
17:     5   0.28
18:     9   0.28
19:     t   0.27
20:     g   0.24
21:     G   0.23
22:     J   0.22
23:     E   0.22
24:    Uu   0.21
25:    Zz   0.20
26:     4   0.19
27:     d   0.18
28:   OoO   0.18
29:     L   0.17
30:     2   0.16
31:     b   0.14
32:     f   0.14
33:     e   0.11
34:     Q   0.10
35:    Cc   0.09
36:    Yy   0.09
37:     F   0.08
38:     D   0.07
39:     7   0.06
40:     r   0.06
41:    Pp   0.05
42:     j   0.05
43:     q   0.05
44:     T   0.04
45:    Vv   0.02
46:     1   0.01
```

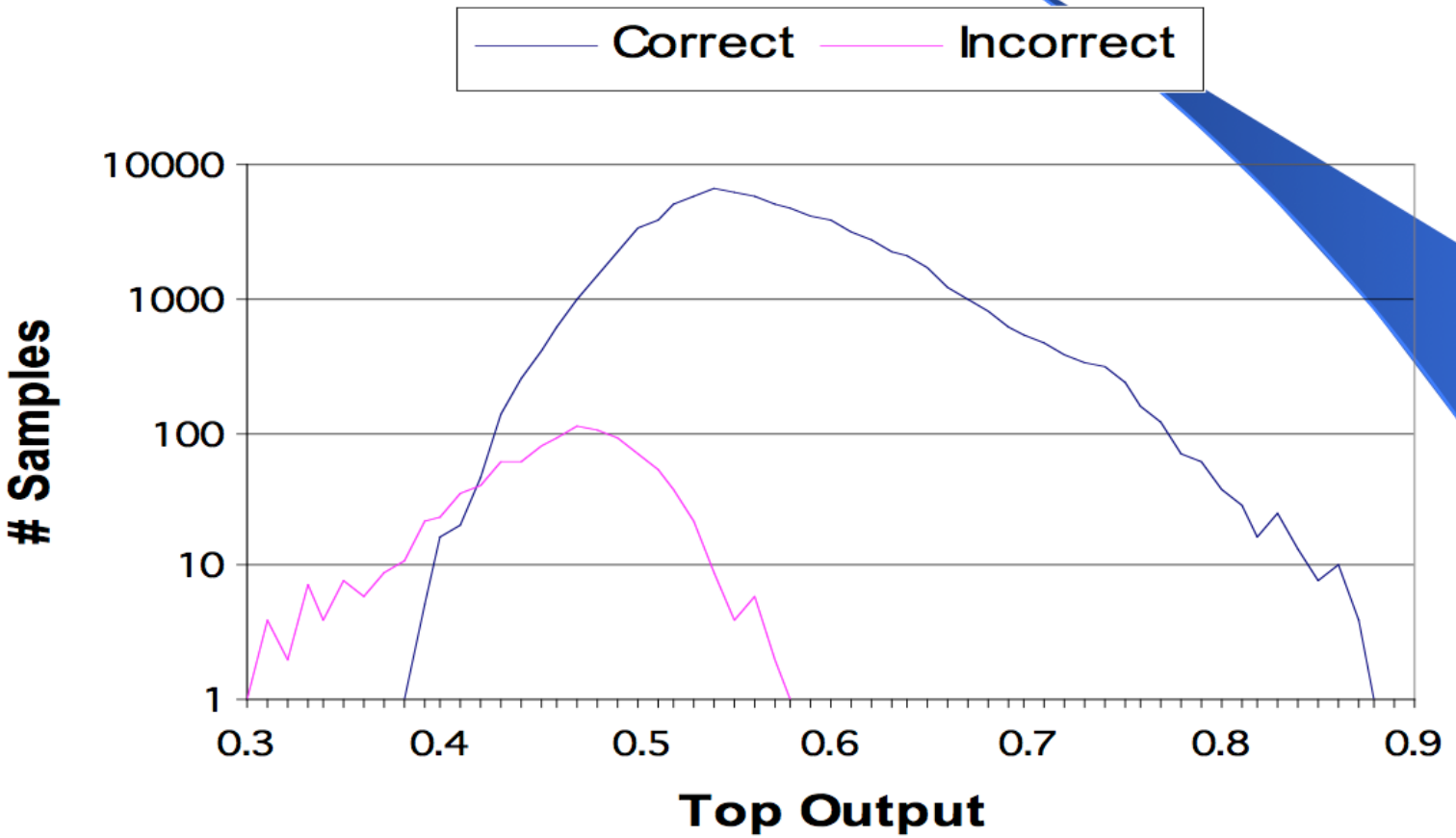
Sample Output:

Analysis



Network outputs on test set after standard backpropagation training.

Analysis



Network outputs on test set after CB training.

Classification Based Models

- CB1: Only backpropagates error on misclassified training patterns
- CB2: Adds a confidence margin, μ, that is increased globally as training progresses
- CB3: Learns a confidence C_i for each training pattern i as training progresses
  - Patterns often misclassified have low confidence
  - Patterns consistently classified correctly gain confidence
  - Best overall results and robustness

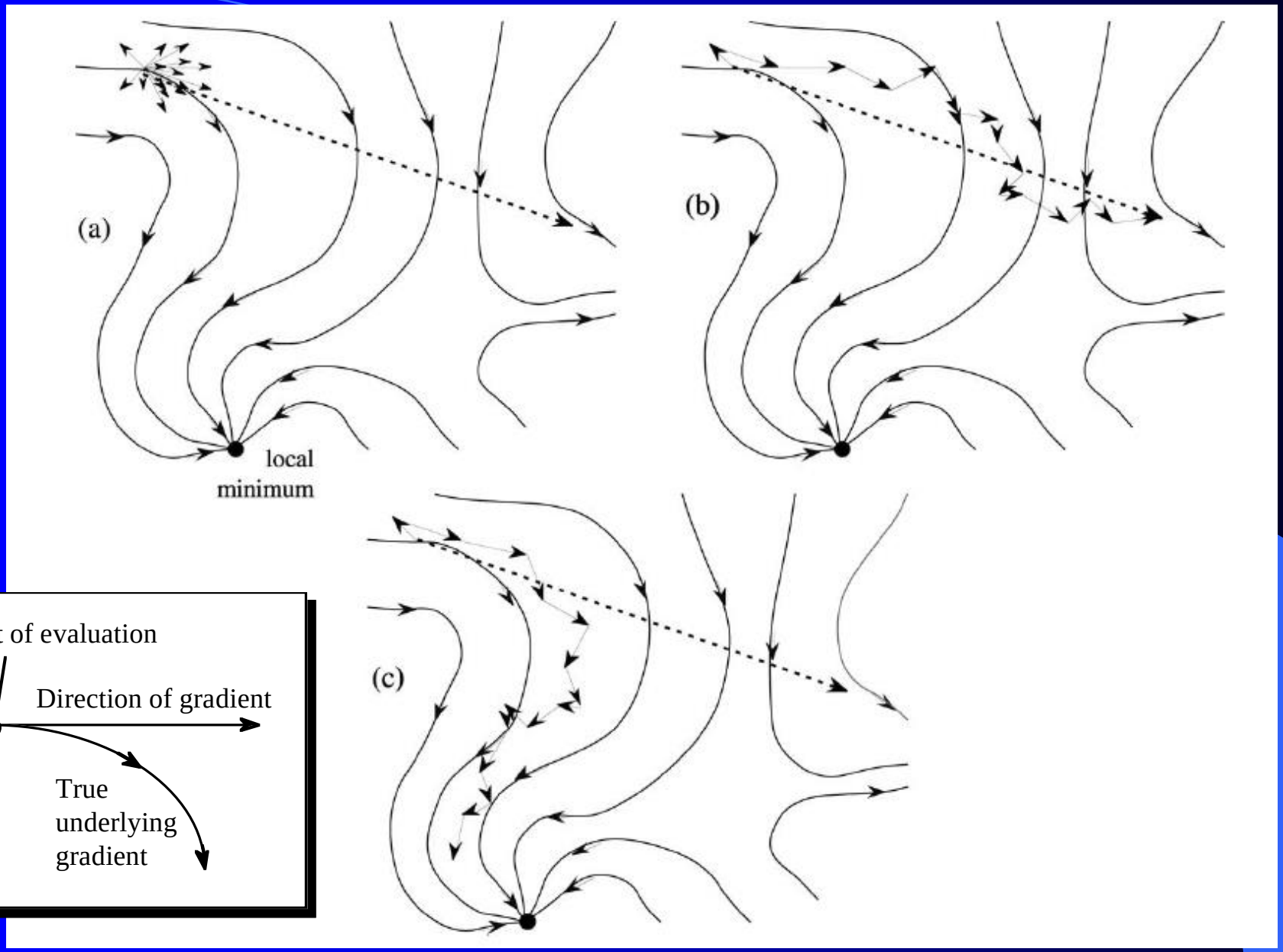
Batch Update

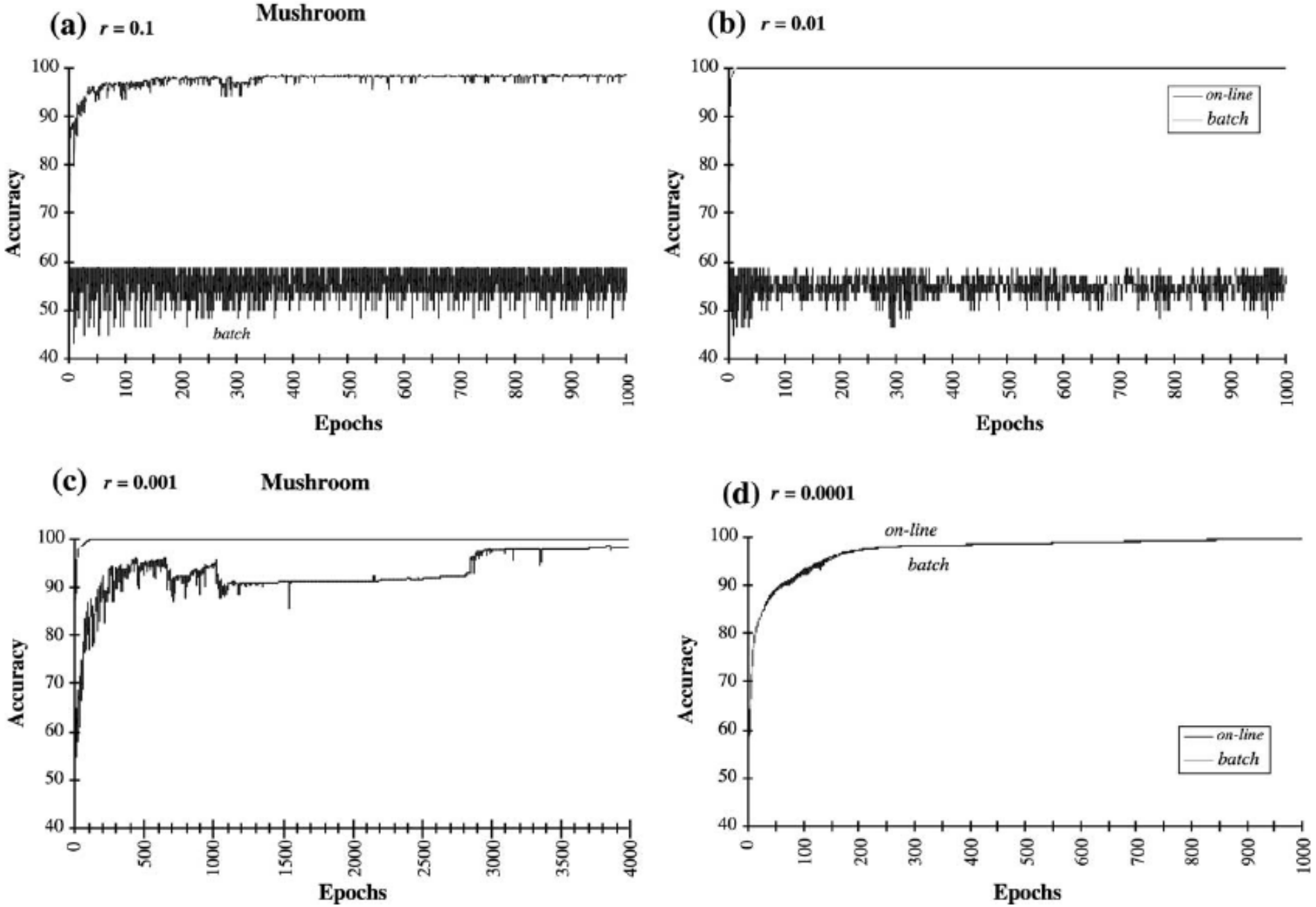
- With On-line (stochastic) update we update weights after every pattern
- With Batch update we accumulate the changes for each weight, but do not update them until the end of each epoch
- Batch update gives a correct direction of the gradient for the entire data set, while on-line could do some weight updates in directions quite different from the average gradient of the entire data set
  - Based on noisy instances and also just that specific instances will not represent the average gradient
- Proper approach? - Conference experience
  - Most (including us) assumed batch more appropriate, but batch/on-line a non-critical decision with similar results
- We tried to speed up learning through "batch parallelism"

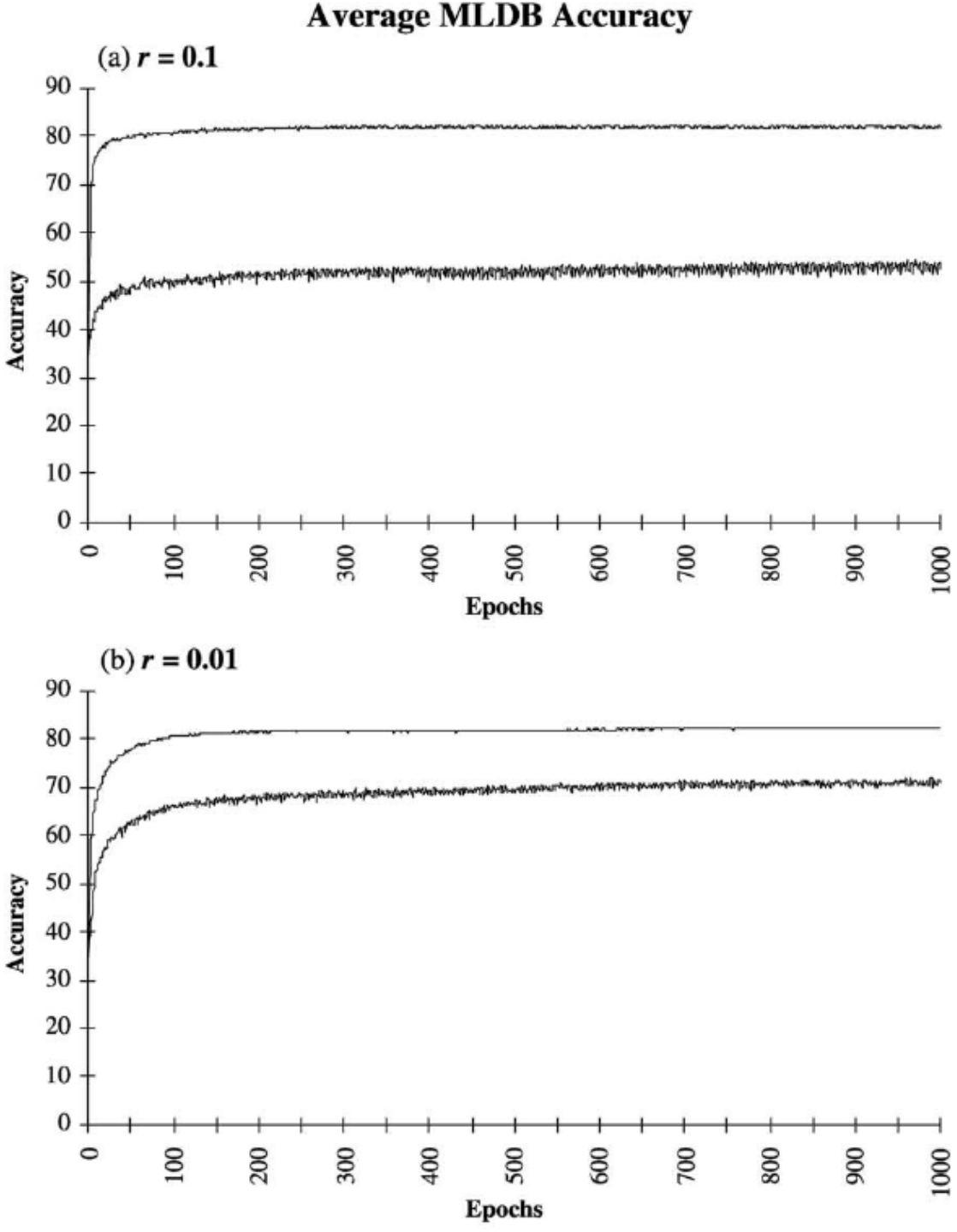
On-Line vs. Batch

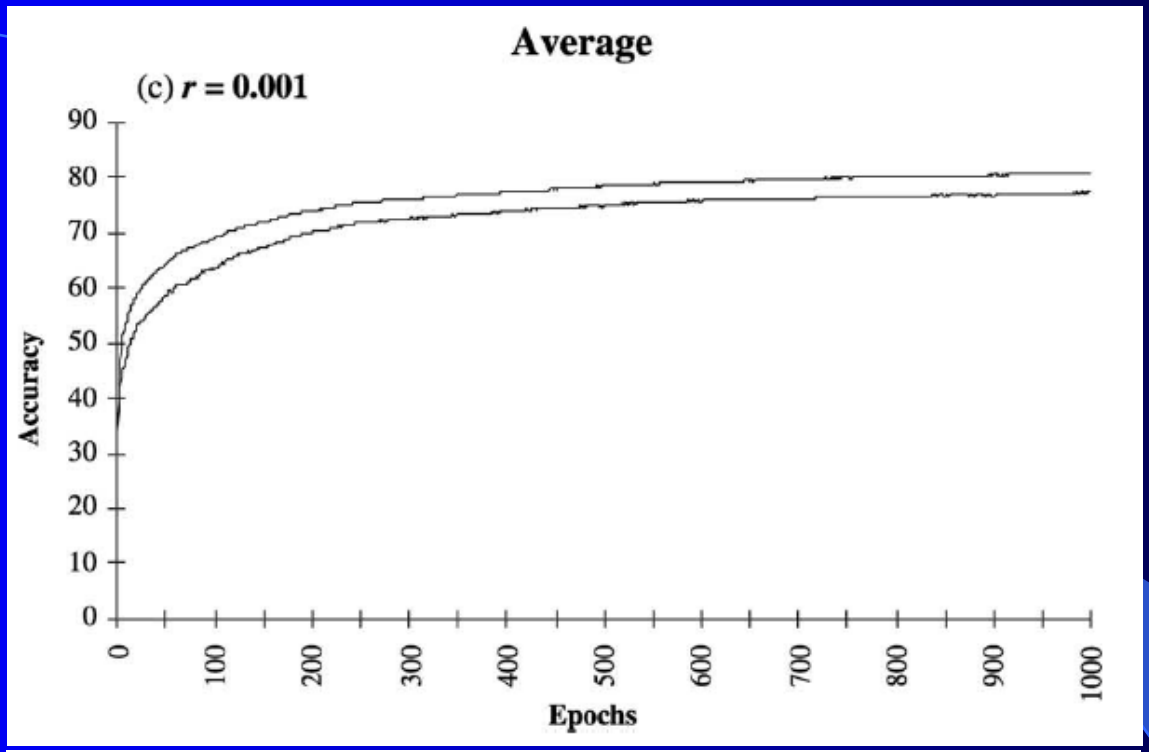
Wilson, D. R. and Martinez, T. R., The General Inefficiency of Batch Training for Gradient Descent Learning, *Neural Networks*, vol. **16**, no. 10, pp. 1429-1452, 2003

- Many people still not aware of this issue – Changing
- Misconception regarding "Fairness" in testing batch vs. on-line with the same learning rate
  - BP already sensitive to LR - why? Both approaches need to make a *small* step in the calculated gradient direction – (about the same magnitude)
  - With batch need a "smaller" LR since weight changes accumulate (alternatively divide by $|TS|$)
  - To be "fair", on-line should have a comparable LR??
  - Initially tested on relatively small data sets
- On-line update approximately follows the curve of the gradient as the epoch progresses
- With appropriate learning rate batch gives correct result, just less efficient, since you have to compute the entire training set for each small weight update, while on-line will have done $|TS|$ updates


(a)
local
minimum
(b)
(c)
Point of evaluation
Direction of gradient
True
underlying
gradient


(a) r = 0.1
Mushroom
Accuracy
Epochs
batch
(b) r = 0.01
on-line
batch
(c) r = 0.001
Mushroom
(d) r = 0.0001
on-line
batch


Average MLDB Accuracy
(a) r = 0.1
Accuracy
Epochs
(b) r = 0.01
Accuracy
Epochs


Average
(c) r = 0.001
Accuracy
Epochs
0
10
20
30
40
50
60
70
80
90
0
100
200
300
400
500
600
700
800
900
1000

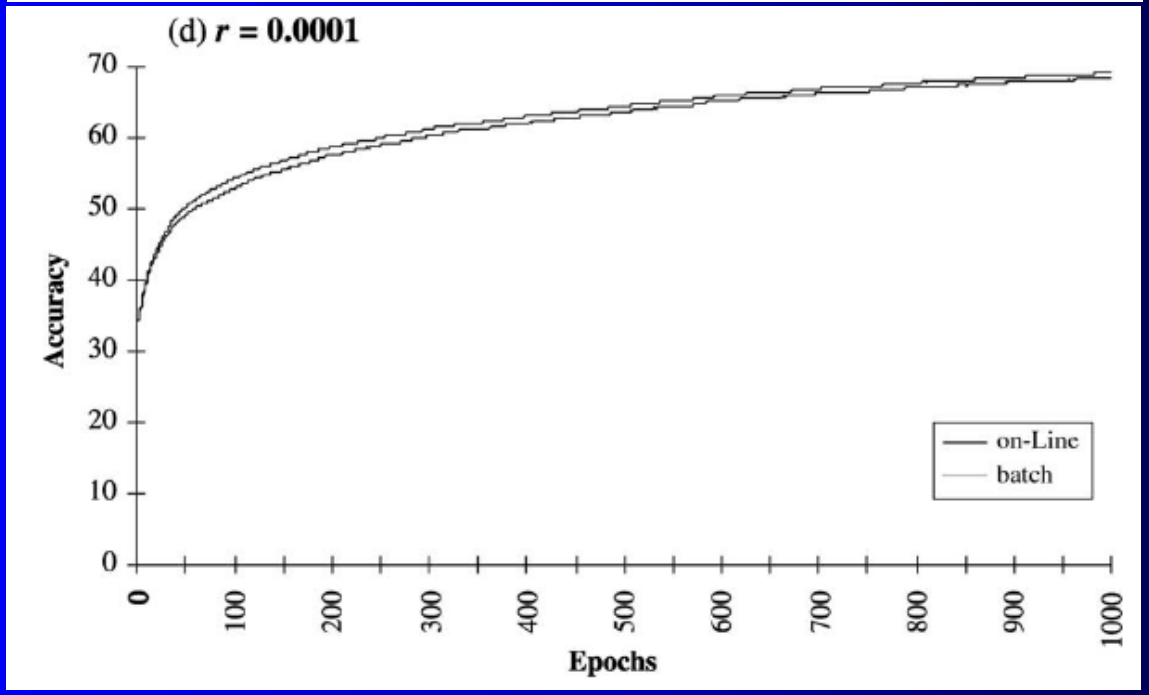

(d) r = 0.0001
Accuracy
Epochs
on-Line
batch
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700
800
900
1000

Table 1
Summary of MLDB experiments. The maximum average generalization accuracy for on-line and batch training for each task are shown, along with the 'best' learning rate and the number of epochs required for convergence with the learning rate shown. The ratio of required epochs for batch to on-line training is shown as the 'speedup' of on-line training

Dataset	Output classes	Training set size	Max accuracy			Best LR		Epochs needed		On-line's speedup
			On-line	Batch	Diff.	On-line	Batch	On-line	Batch	
Bridges	7	42	82.14	82.86	−0.72	0.1	0.1	29	156	5.4
Hepatitis	2	48	93.75	93.75	0.00	0.01	0.01	85	87	1.0
Zoo	7	54	94.44	94.44	0.00	0.1	0.1	14	41	2.9
Iris	3	90	100.00	100.00	0.00	0.01	0.01	596	602	1.0
Wine	3	107	100.00	100.00	0.00	0.1	0.01	68	662	9.7
Flag	8	116	55.13	56.67	−1.54	0.1	0.01	9	115	12.8
Sonar	2	125	81.47	81.95	−0.48	0.01	0.01	103	576	5.6
Glass	7	128	72.32	73.02	−0.70	0.1	0.01	921	6255	6.8
Voting	2	139	95.74	95.74	0.00	0.1	0.01	6	49	8.2
Heart	2	162	80.37	79.07	1.30	0.1	0.01	825	29	0.0[a]
Heart (Cleaveland)	5	178	85.33	84.33	1.00	0.01	0.01	80	500	6.3
Liver (Bupa)	2	207	72.9	72.32	0.58	0.1	0.01	435	3344	7.7
Ionosphere	2	211	93.72	93.57	0.15	0.1	0.01	500	978	2.0
Image segmentation	7	252	97.86	97.5	0.36	0.1	0.01	305	2945	9.7
Vowel	11	317	90.76	89.53	1.23	0.1	0.01	999	9434	9.4
CRX	2	392	89.77	89.85	−0.08	0.01	0.001	26	569	21.9
Breast cancer (WI)	2	410	96.62	96.47	0.15	0.01	0.01	38	54	1.4
Australian	2	414	88.41	88.41	0.00	0.001	0.0001	134	1257	9.4
Pima Indians diabetes	2	461	76.41	76.73	−0.32	0.1	0.01	58	645	11.1
Vehicle	4	508	85.27	83.43	1.84	0.1	0.01	1182	9853	8.3
LED Creator	10	600	74.35	74.05	0.30	0.1	0.01	29	179	6.2
Sat	7	2661	92.02	91.65	0.37	0.01	0.001	3415	14049	4.1
Mushroom	2	3386	100.00	100.00	0.00	0.01	0.0001	20	1820	91.0
Shuttle	7	5552	99.69	97.44 +	2.25	0.1	0.0001	7033	9966	100.0[b]
LED creator +17	10	6000	73.86	73.79	0.07	0.01	0.001	6	474	79
Letter recognition	26	12000	83.53	73.25 +	10.28	0.001	0.0001	4968	9915	100.0[b]
Average	6	1329	86.76	86.15	0.62	0.061	0.014	842	2867	20.03
Median	4	232	89.09	88.97	0.04	0.1	0.01	94	624	7.95

[a] On the *Heart* task, on-line achieved 78.52% accuracy in 26 epochs with a learning rate of 0.01.
[b] Batch training had not finished after 10,000 epochs for the *Shuttle* and *Letter-Recognition* tasks, but appeared to be progressing about 100 times slower than on-line.

Semi-Batch on Digits

Learning Rate	Batch Size	Max Word Accuracy	Training Epochs
0.1	1	96.49%	21
0.1	10	96.13%	41
0.1	100	95.39%	43
0.1	1000	84.13%+	4747+
0.01	1	96.49%	27
0.01	10	96.49%	27
0.01	100	95.76%	46
0.01	1000	95.20%	1612
0.01	20,000	23.25%+	4865+
0.001	1	96.49%	402
0.001	100	96.68%	468
0.001	1000	96.13%	405
0.001	20,000	90.77%	1966
0.0001	1	96.68%	4589
0.0001	100	96.49%	5340
0.0001	1000	96.49%	5520
0.0001	20,000	96.31%	8343

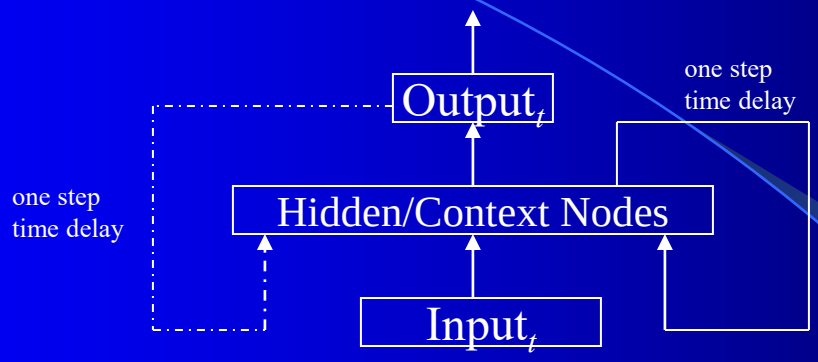
On-Line vs. Batch Issues

- Some say just use on-line LR but divide by *n* (training set size) to get the same feasible LR for both (non-accumulated), but on-line still does *n* times as many updates per epoch as batch and is thus much faster
- True Gradient - We just have the gradient of the training set anyways which is an approximation to the true gradient and true minima
- Momentum and true gradient - same issue with other enhancements such as adaptive LR, etc.
- Training sets are getting larger - makes discrepancy worse since we would do batch update relatively less often
- Large training sets great for learning and avoiding overfit - best case scenario is huge/infinite set where never have to repeat - just 1 partial epoch and just finish when learning stabilizes – batch in this case?
- Mini-batches can be useful for algorithms which are sensitive to a bad gradient direction, and when GPU parallelism gets it for free

Multiple Outputs

- Typical to have multiple output nodes, even with just one output feature (e.g. Iris data set)
- Would if there are multiple "independent output features"
  - Could train independent networks
  - Also common to have them share hidden layer
    - May find shared features
    - Transfer Learning
  - Could have shared and separate subsequent hidden layers, etc.
- Structured Outputs
- Multiple Output Dependency? (MOD)
  - New research area

Recurrent Networks



- Some problems happen over time - Speech recognition, stock forecasting, target tracking, etc.
- Recurrent networks can store state (memory) which lets them learn to output based on both current and past inputs
- Learning algorithms are more complex but are becoming increasingly better at solving more complex problems (LSTM - more with deep)
- Alternatively, for some problems we can use a larger "snapshot" of features over time with standard backpropagation learning and execution (e.g. NetTalk)