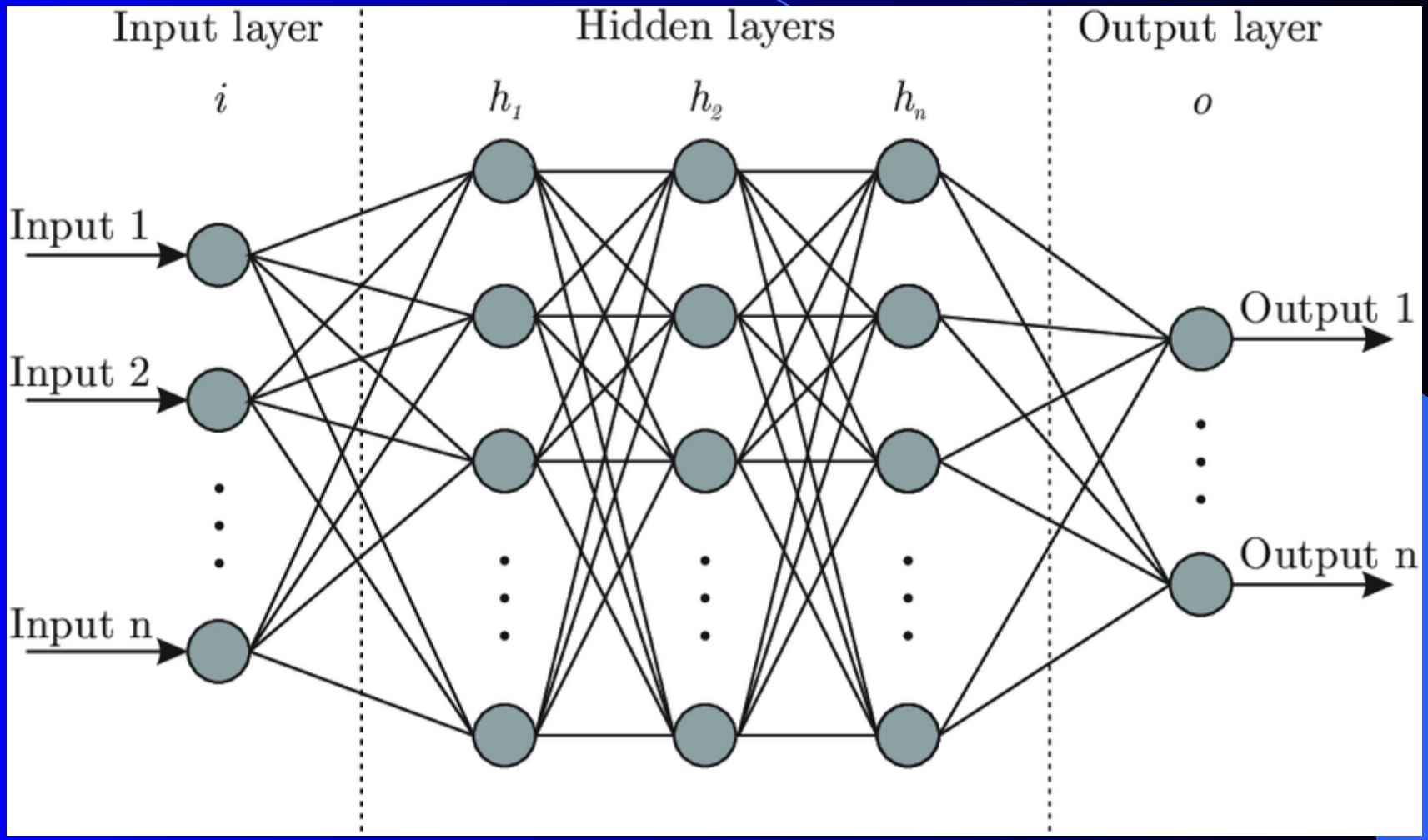
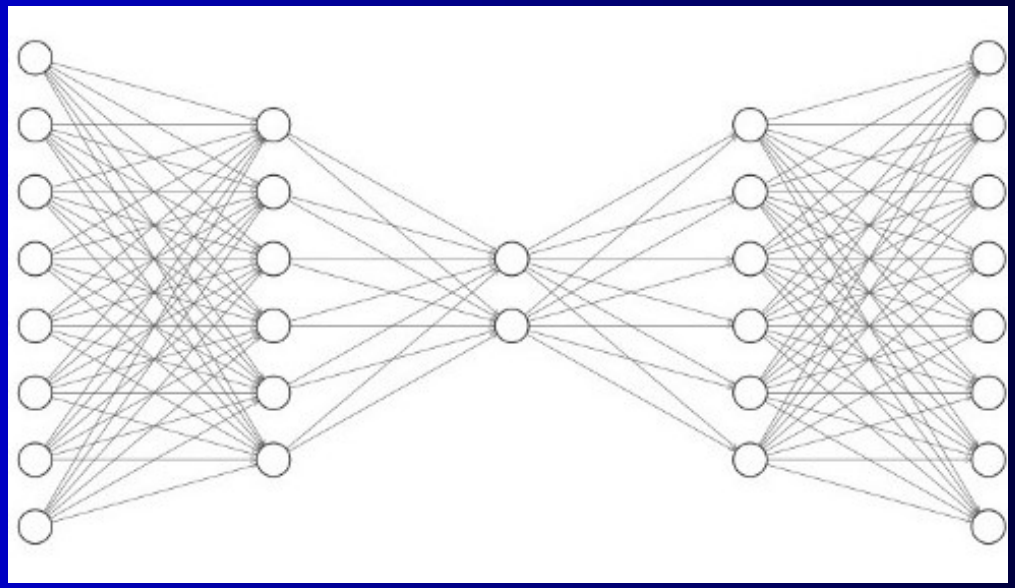
Multi-layer Perceptron with Backpropagation Learning

Multi-layer Perceptron (MLP)



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



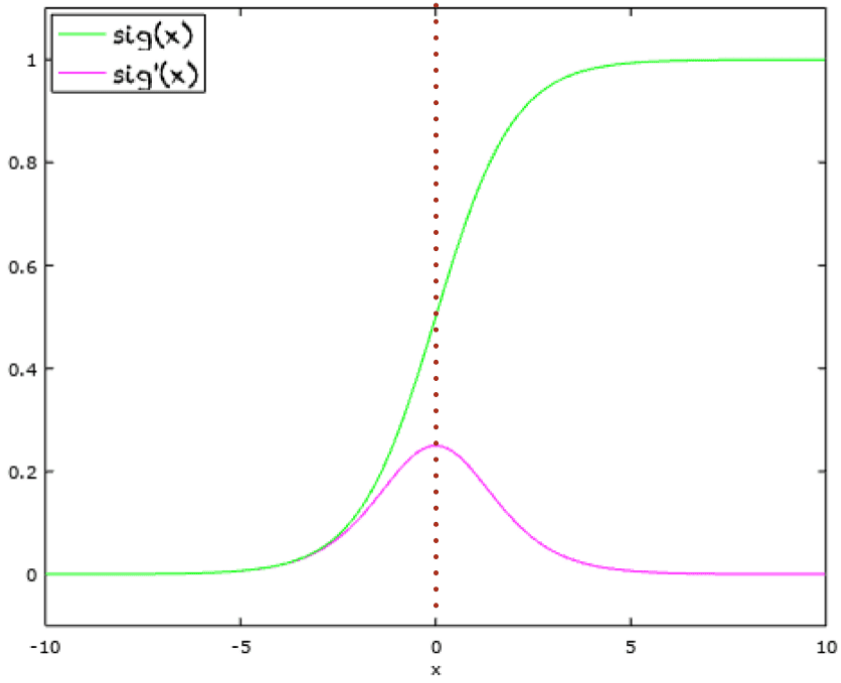
Number of Hidden Nodes

- How many needed is a function of how hard the task is
- Common to use one fully connected hidden layer. Initial number could be ~2n 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

NON-LINEAR ACTIVATIONS

Sigmoid Function

- Differentiable, most unstable in middle



Plot of σ(x) and its derivate σ'(x)

Domain: (−∞, +∞)

Range: (0, +1)

σ(0) = 0.5

Other properties

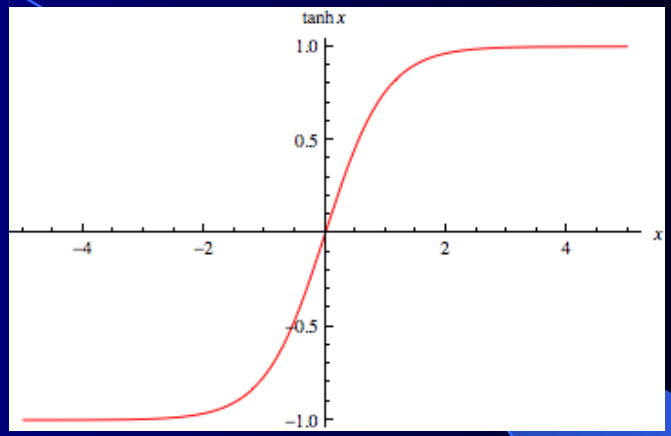
$$\sigma(x) = 1 - \sigma(-x)$$

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

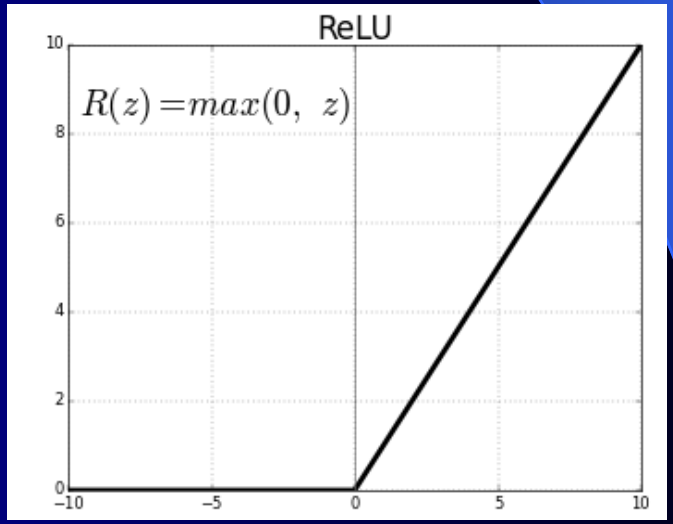
$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Other Activation Functions

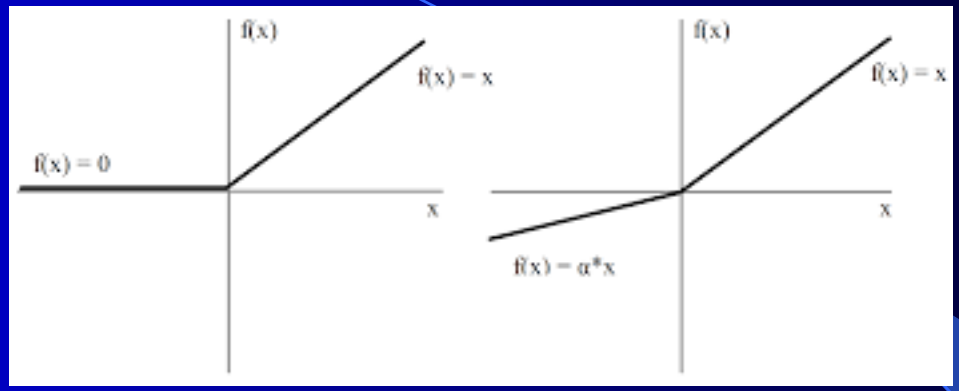
- Hyperbolic tangent (tanh)
  - Same shape as sigmoid but tanh(0)=0 (rather than 0.5)



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



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

Regression with MLP/BP

- For regression in MLPs we use the sum-squared error (L2) loss. 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

Softmax Output Layer Activation

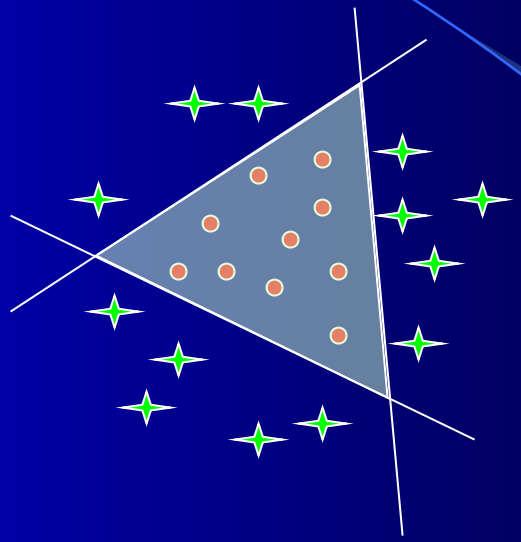
- For *classification* problems it is increasingly popular to use the *softmax* activation function, just at the output layer
- Softmax (softens) 1 of *n* targets to mimic a probability vector for the output nodes
- If there were 3 output nodes with net values 0, .5, and 1 then the outputs for each node would be .18, .31, .51 which 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

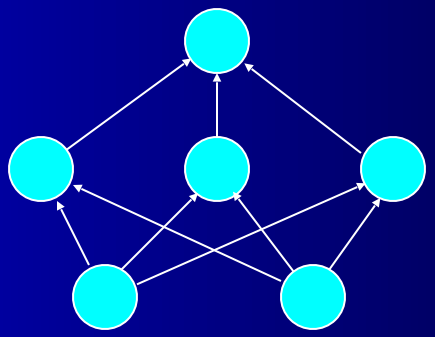
Cross-Entropy and Softmax

- For *classification* it is increasingly popular to use the cross-entropy loss function.
- Cross entropy measures the difference (in entropy) between two distributions.
- We must recalculate the gradient weight update equation when we use new activation/ loss functions.
- The hidden layers still update as usual and include $f'(net)$
- Sklearn always uses this approach for MLP classification

t	z	CE
0	0	0
1	1	0
1	.9	.11
1	.5	.69
1	.1	2.30

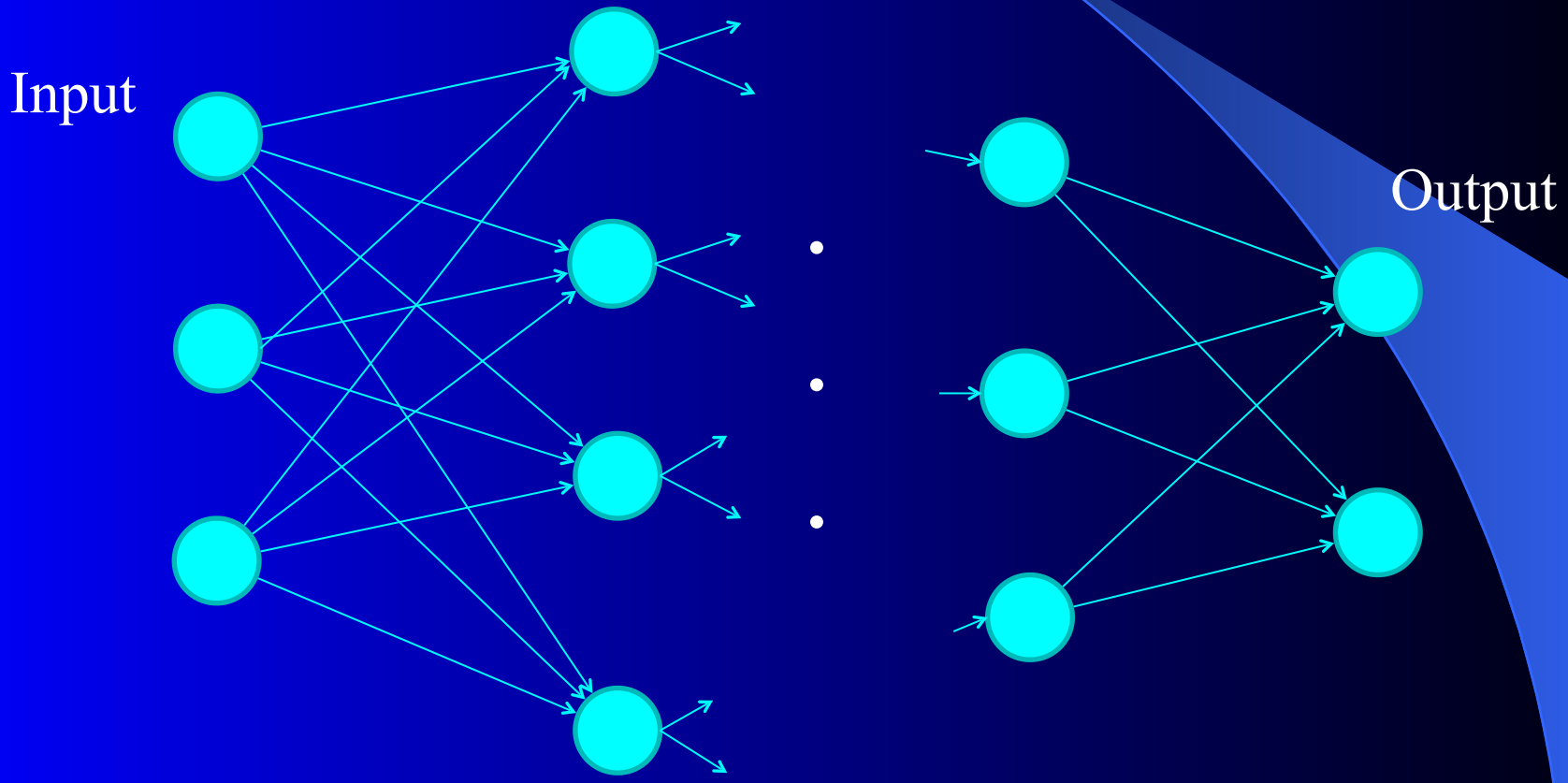
Multi-Layer Generalization





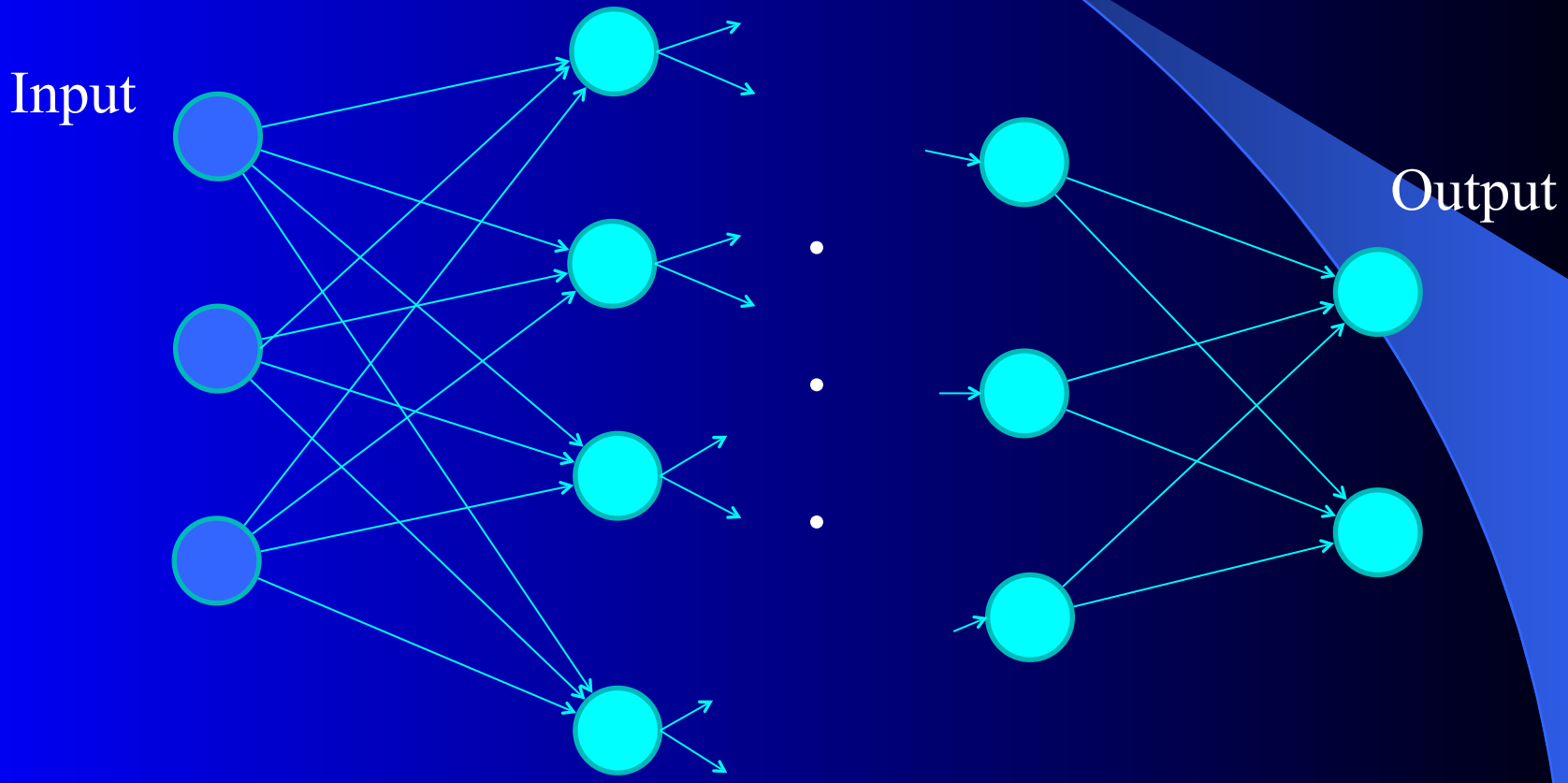
Back-Propagation

- Operates similarly to perceptron learning



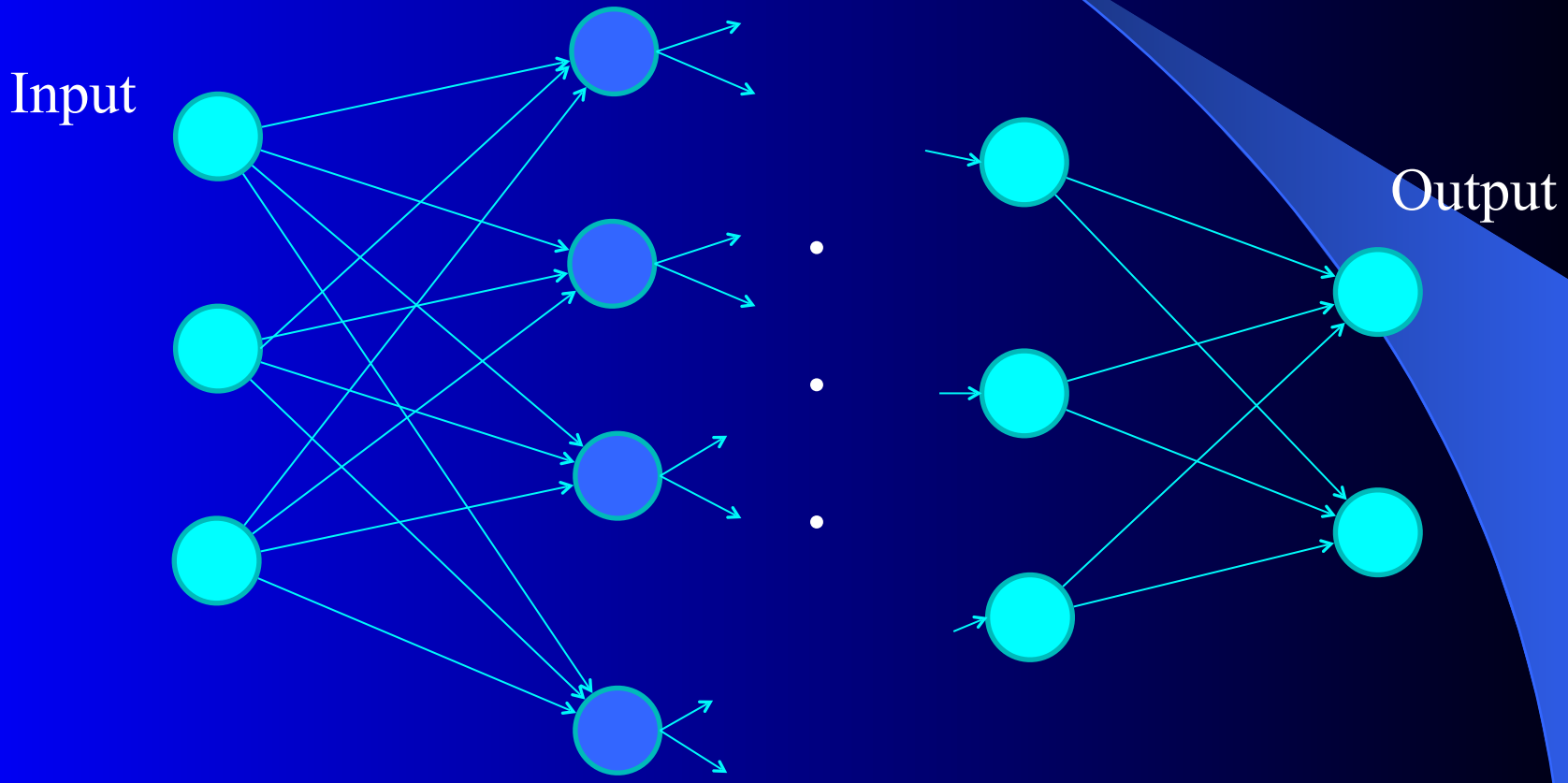
Back-Propagation

- Inputs are fed forward through the network



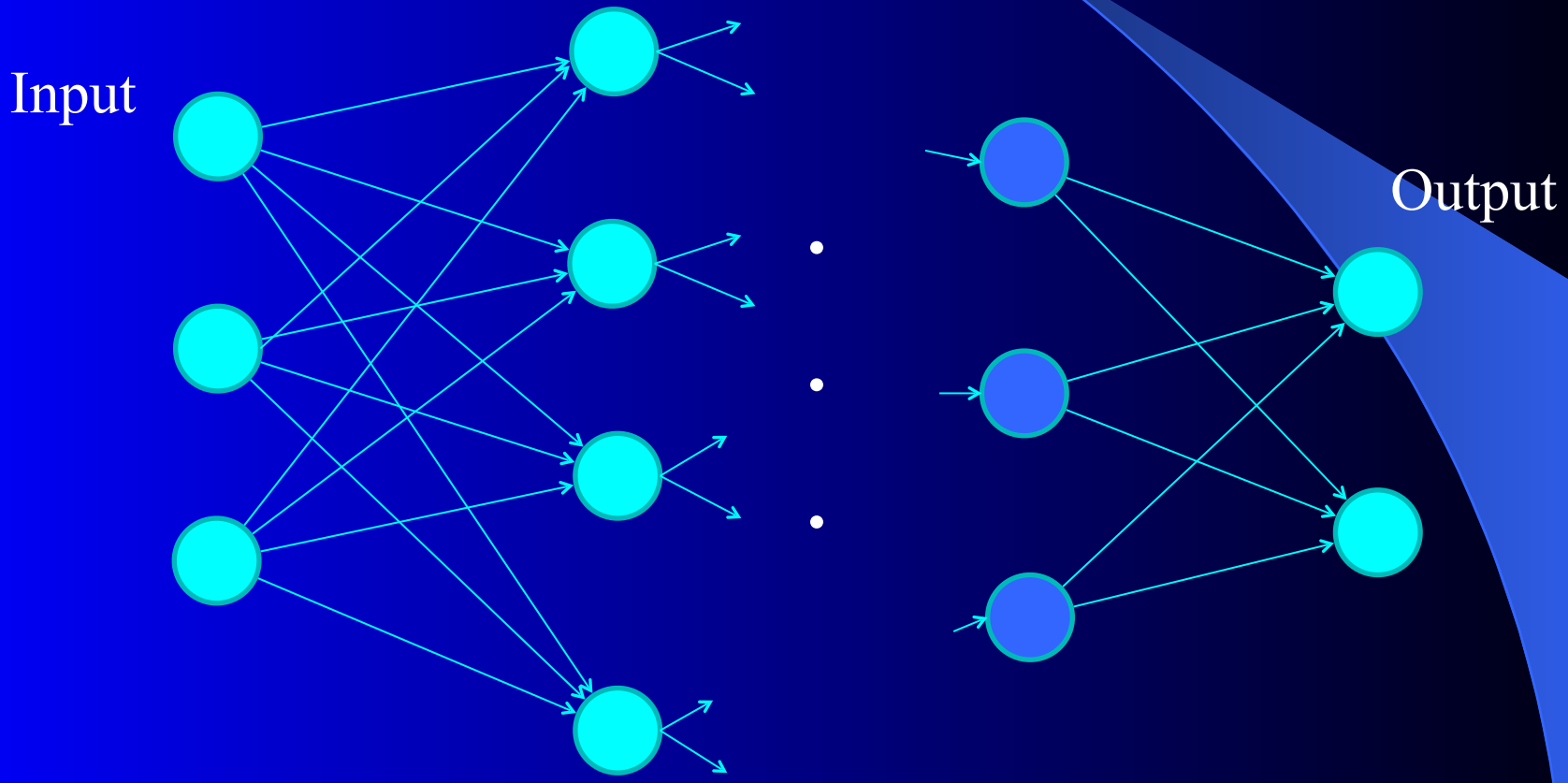
Back-Propagation

- Inputs are fed forward through the network



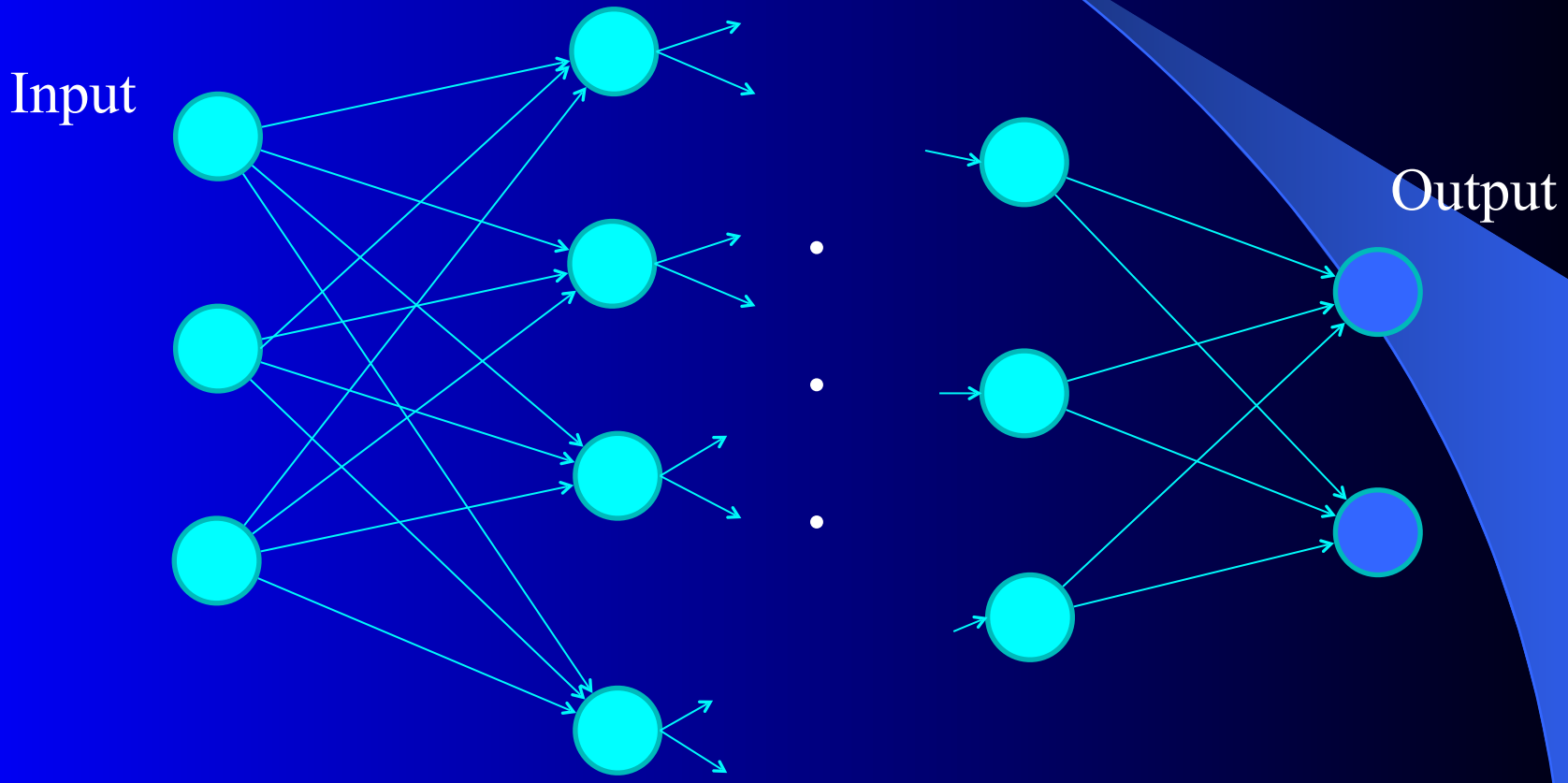
Back-Propagation

- Inputs are fed forward through the network



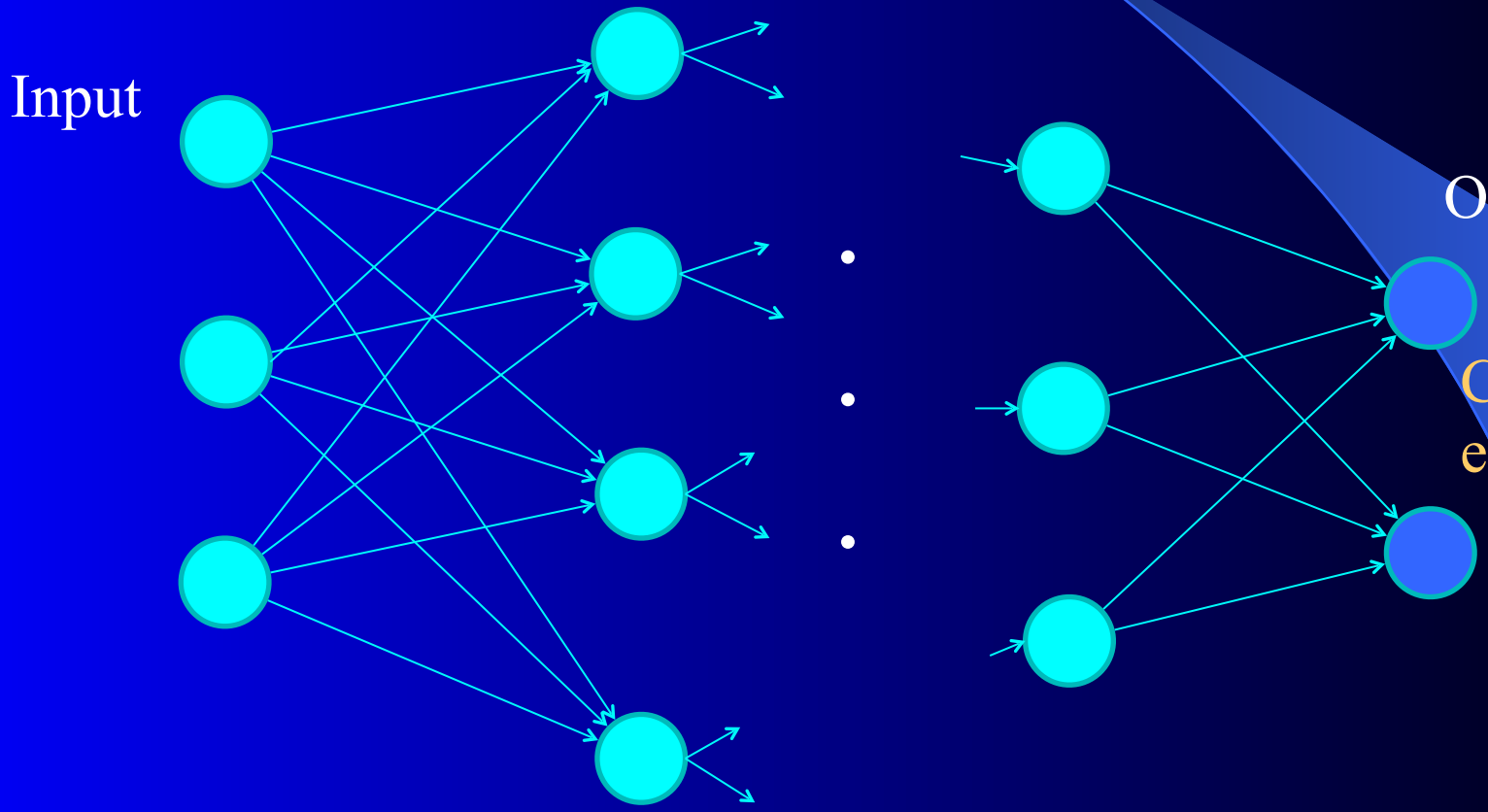
Back-Propagation

- Inputs are fed forward through the network



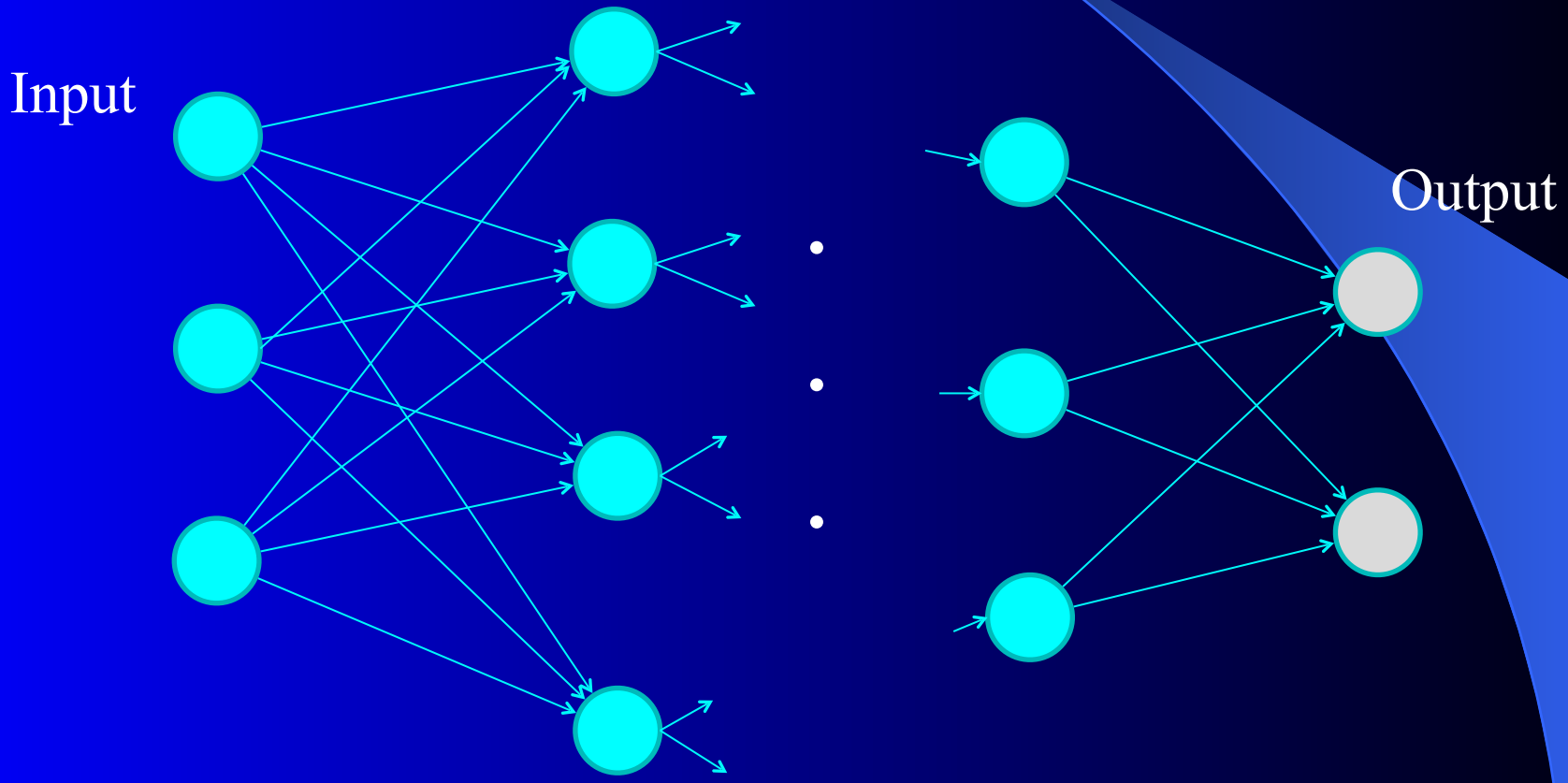
Back-Propagation

- Inputs are fed forward through the network



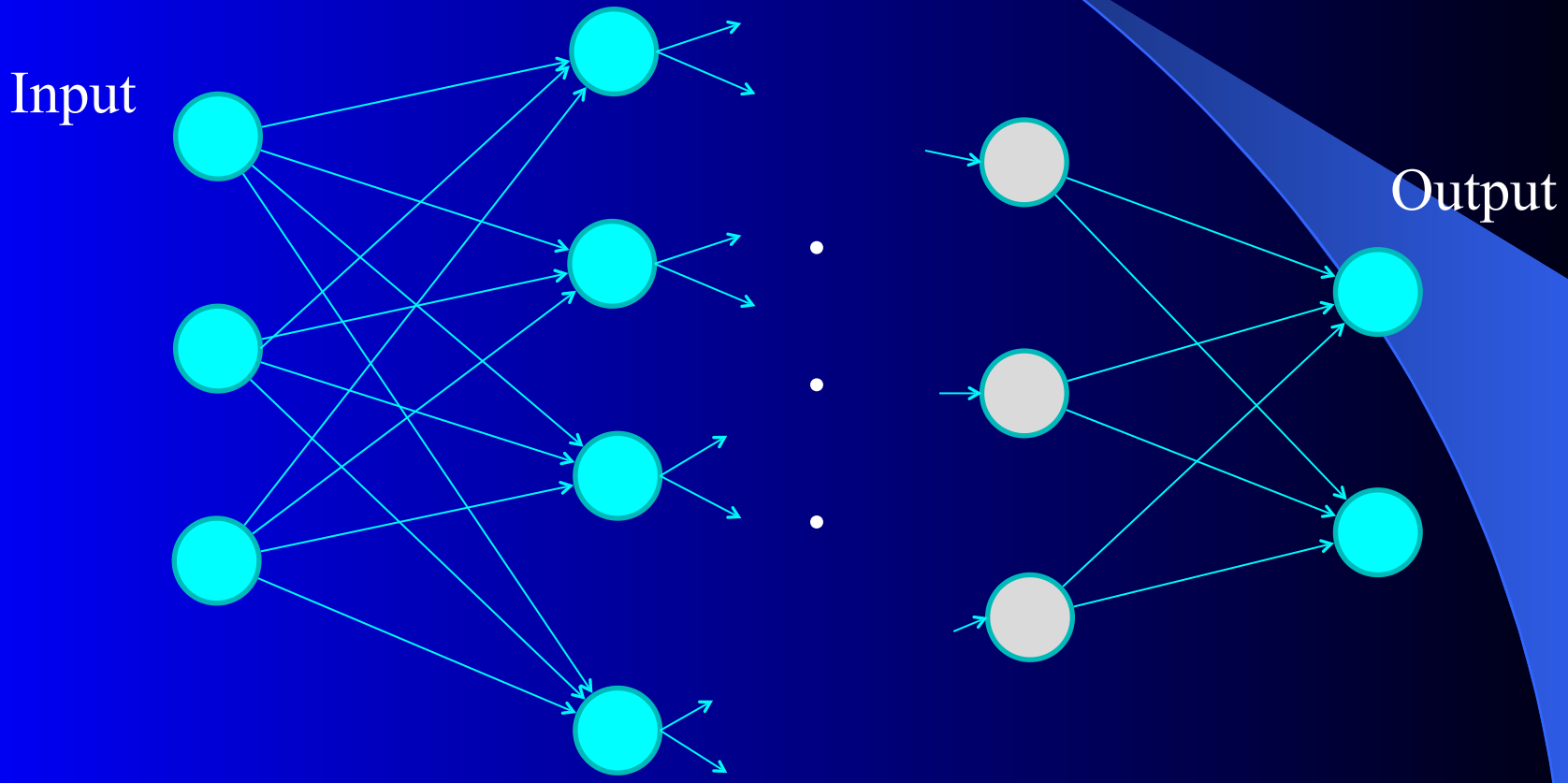
Back-Propagation

- Errors are propagated back



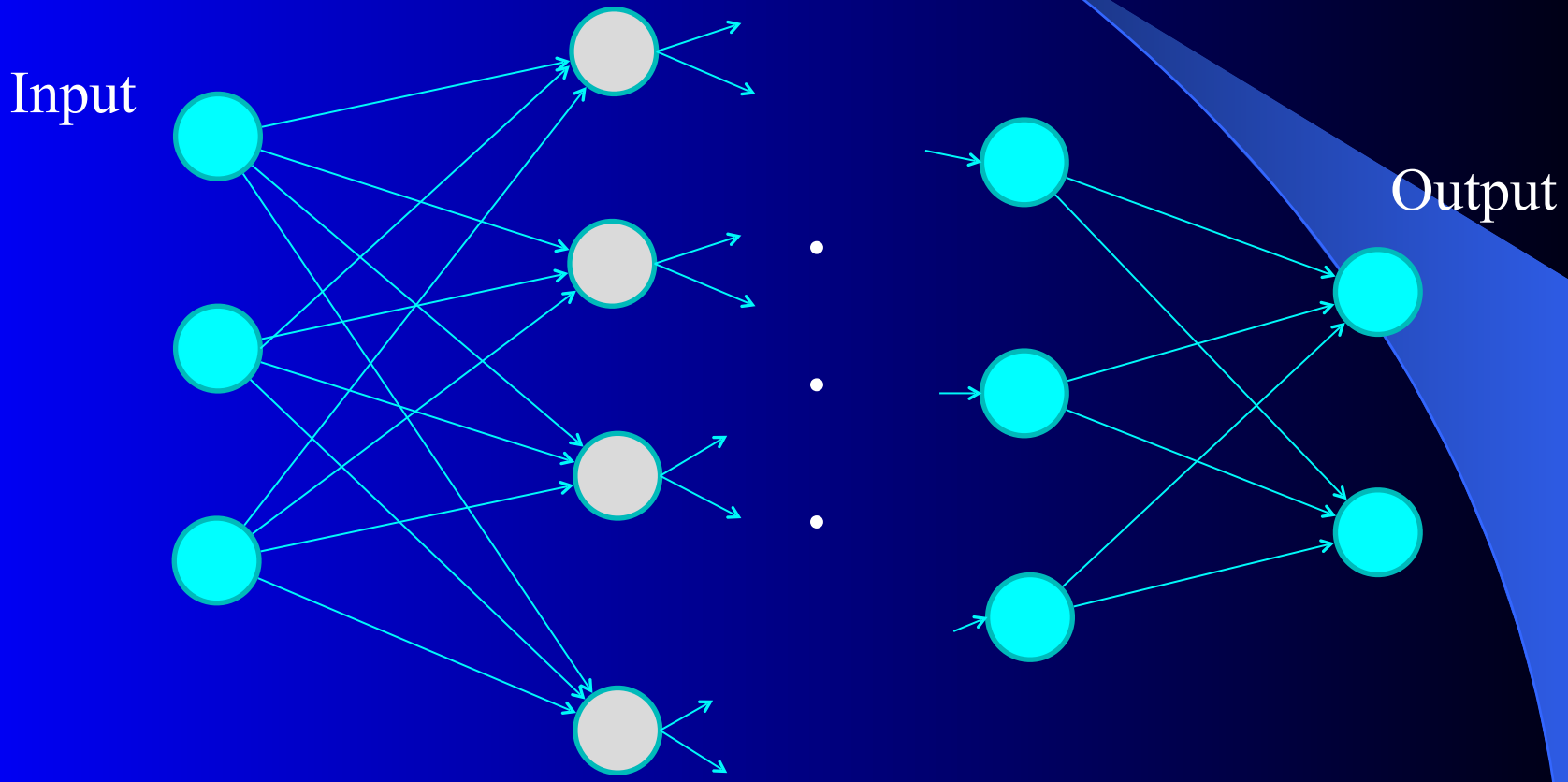
Back-Propagation

- Errors are propagated back



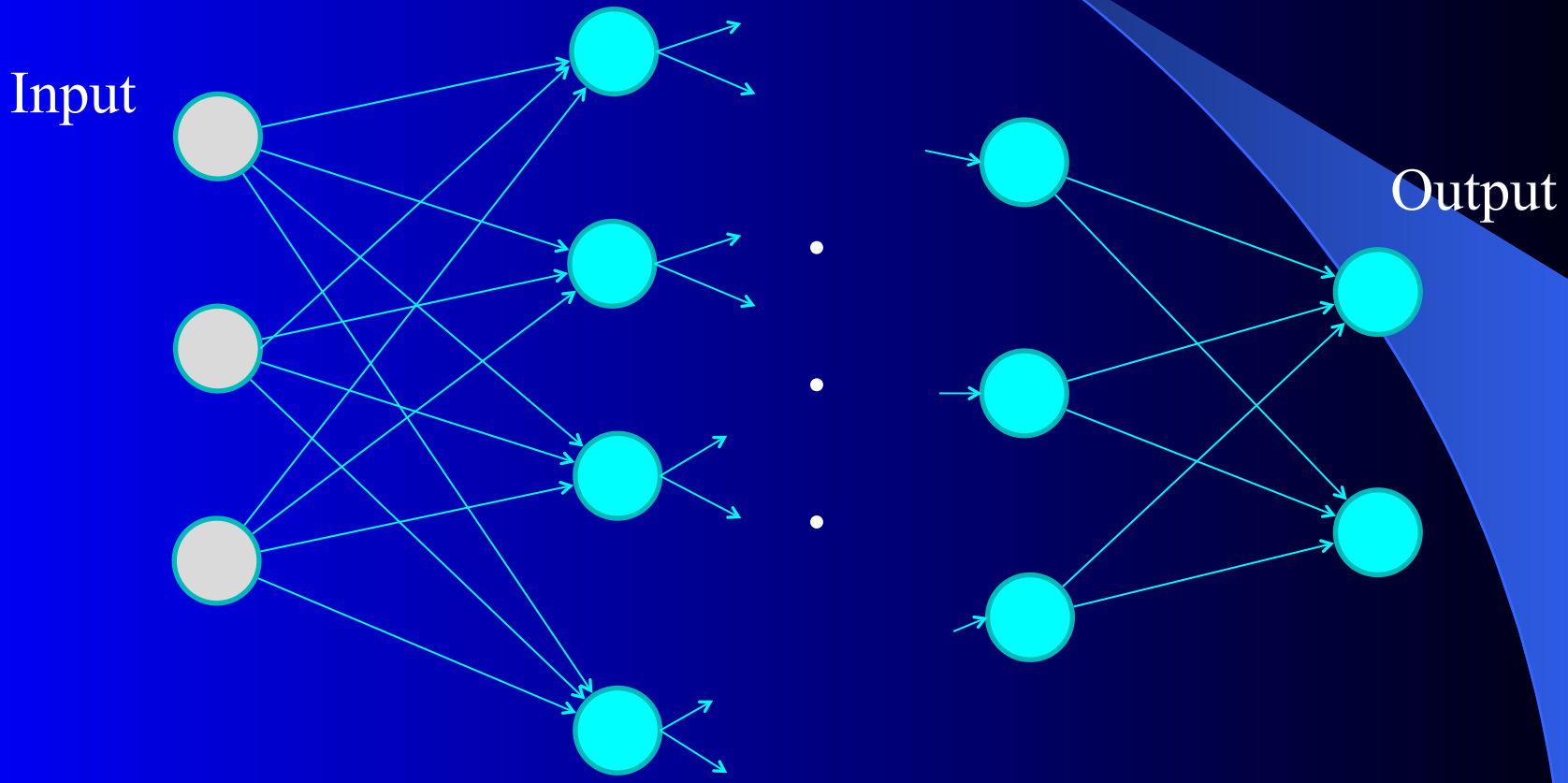
Back-Propagation

- Errors are propagated back



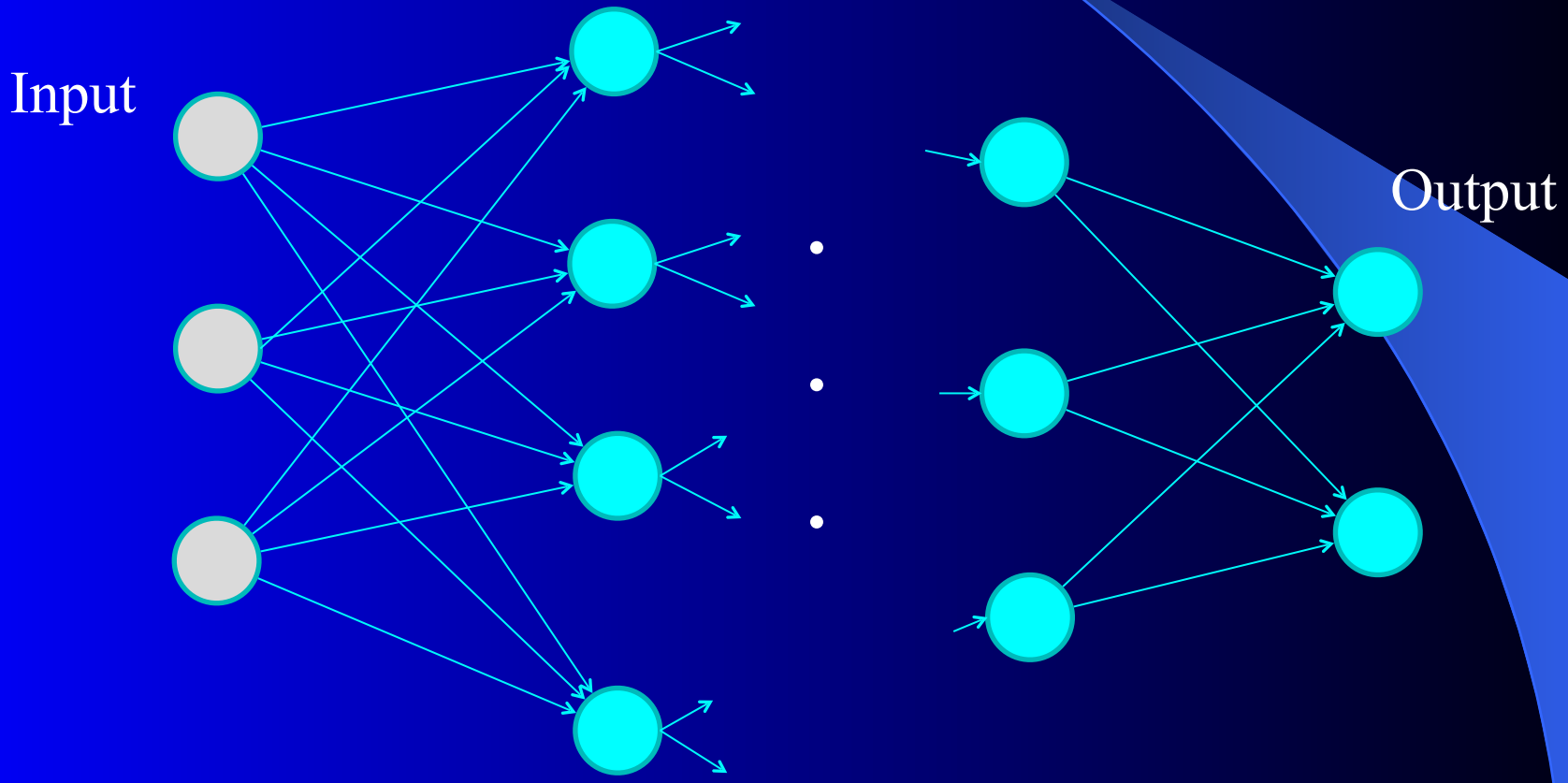
Back-Propagation

- Errors are propagated back



Back-Propagation

- Adjust weights based on errors



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

Batch Update

- With On-line (stochastic) update we update weights after every instance
- 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 usually be at the average gradient
- Proper approach? - Conference experience/Parallel experience
  - Most (including us) 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

Speed up variations of SGD

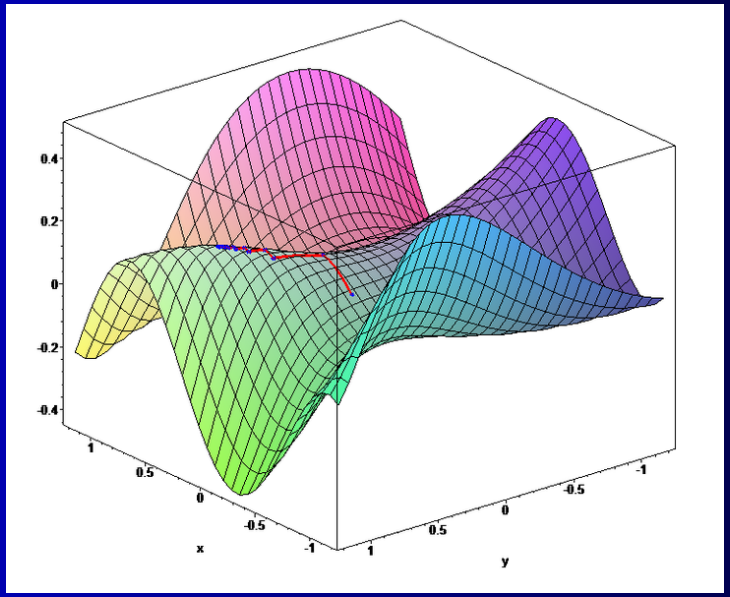
- Use mini-batch rather than single instance for better gradient estimate – *Sometimes* helpful if SGD variation more sensitive to bad gradient, and also for some parallel (GPU) implementations.

- Adaptive learning rate approaches (and other speed-ups) are often used for deep learning since there are so many training updates
  - 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) (1st and 2nd moments) – updates based on a moving average of both - Popular

Local Minima and Neural Networks

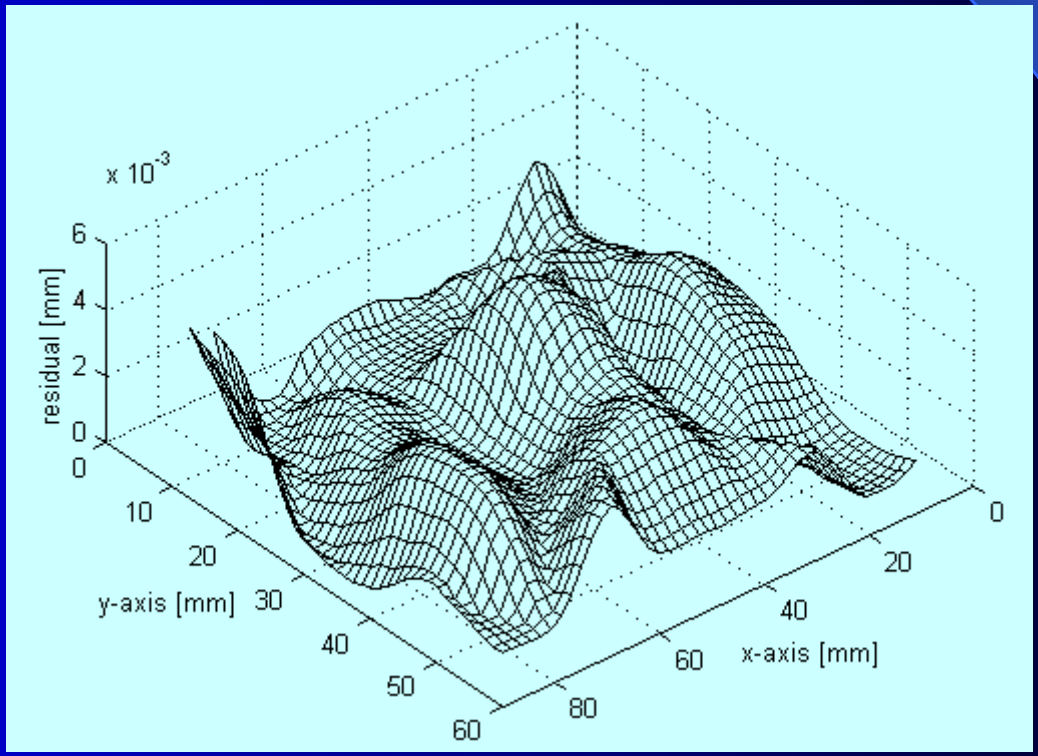
- Neural Network can get stuck in local minima for small networks, but for most large networks (many weights), local minima rarely occur in practice

- 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

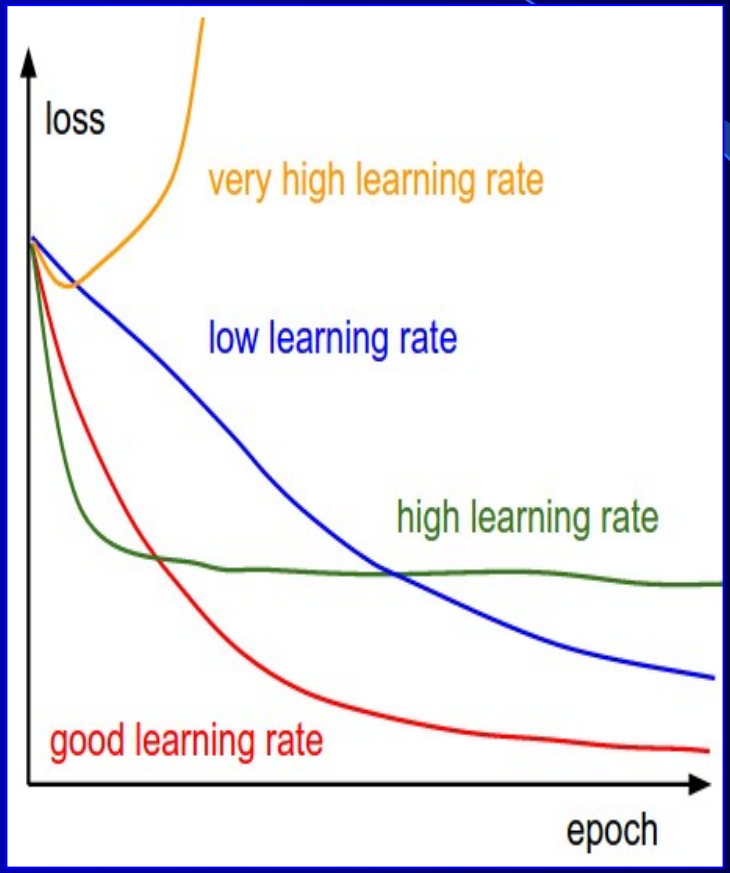


Learning Rate

- 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.



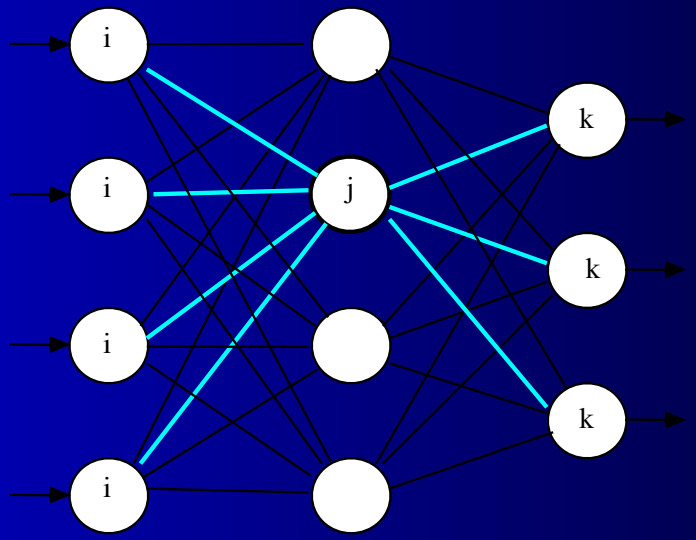
Learning Rate



Backpropagation Learning Algorithm

- Until Convergence (low error or other stopping criteria) do
  - Present a training instance
  - 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
- No bias δ:
  - Note that you never have to calculate δ for a bias node since no layers before the bias node have nodes connected to it.
  - You update the bias node weight based only on the nodes that follow the bias, i.e., are connected to the bias after it.

Backpropagation Learning Equations



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:

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

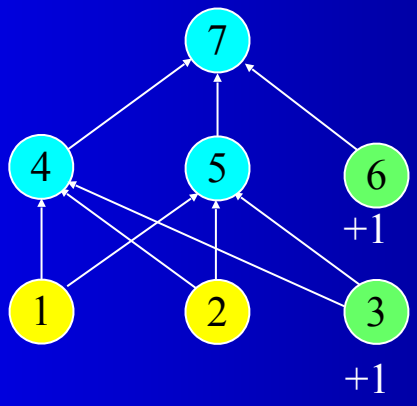
Note that $o_q\left(1 - o_q\right)$ is the derivative – dampens where "certain", enhances where "uncertain"

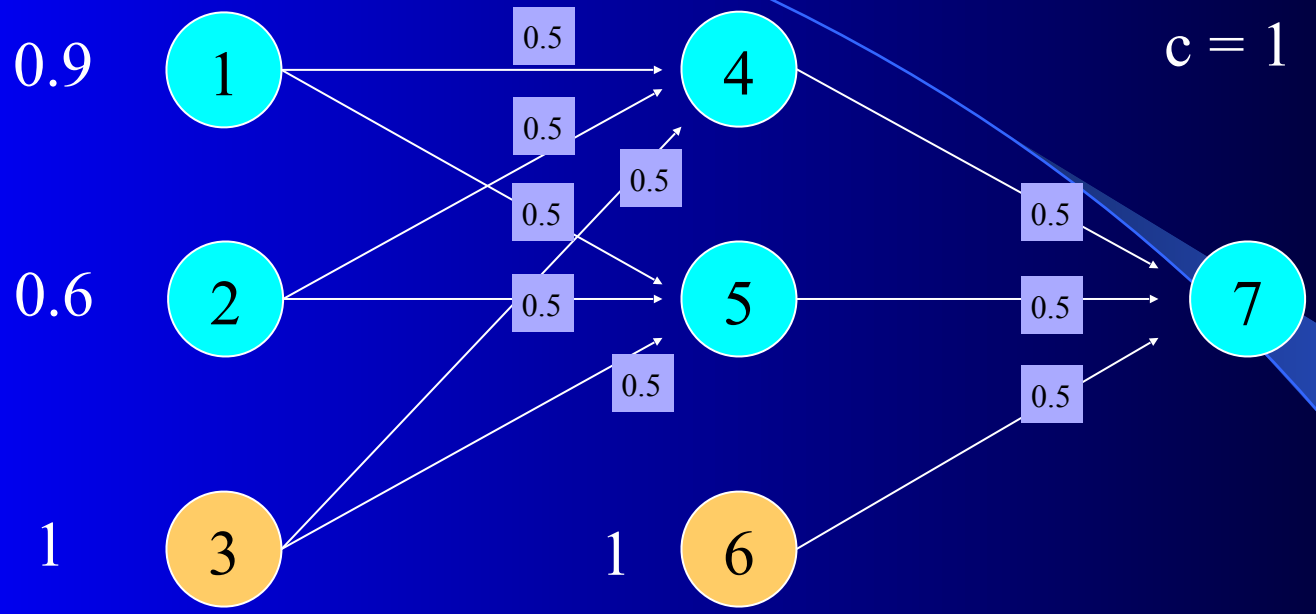
Also note that o_p is the output of node p – we often refer to it as z_p

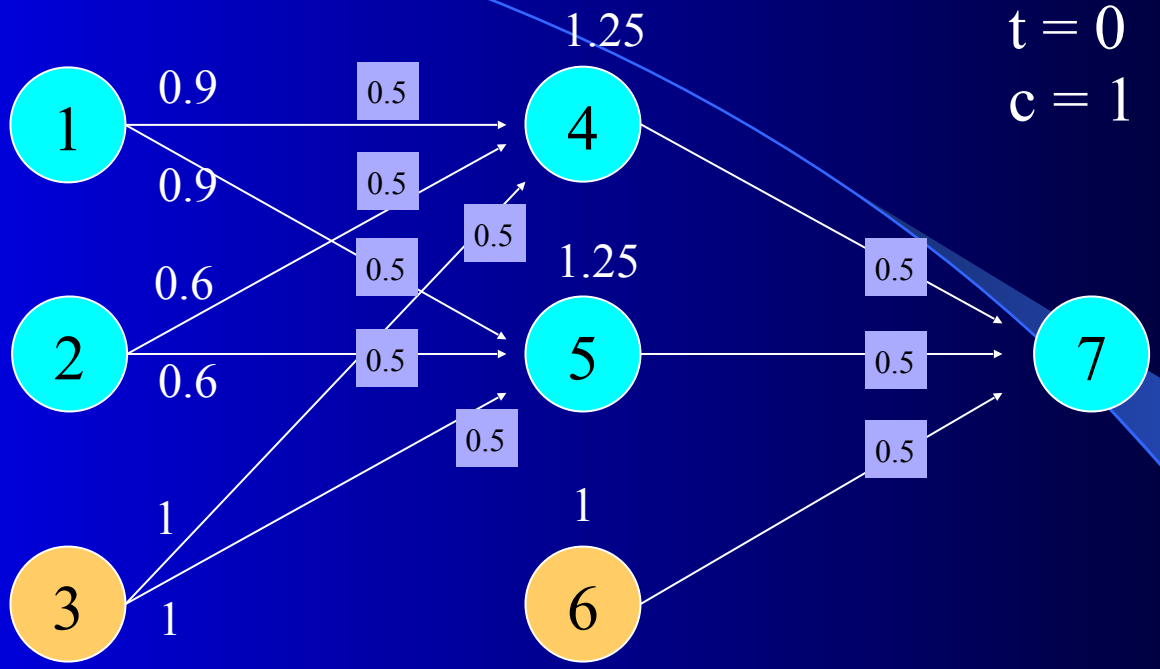
Backpropagation Learning Example

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 data .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 an activation.



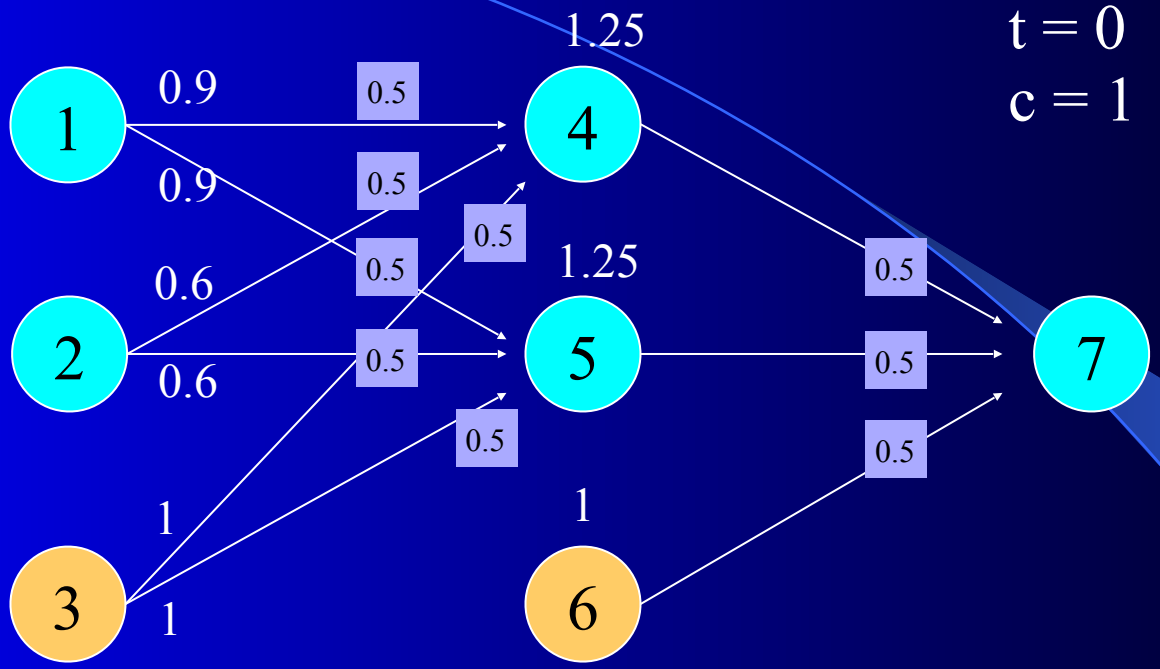

0.5 Weights
t = 0
c = 1 # Learning Rate
0.9
0.6
1
1
1
2
3
4
5
6
7
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5



$$net_q = \sum_{i=1}^{3} x_i w_{ij}$$

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

$net_5 = 1.25$



$$net_q = \sum_{i=1}^{3} x_i w_{ij}$$

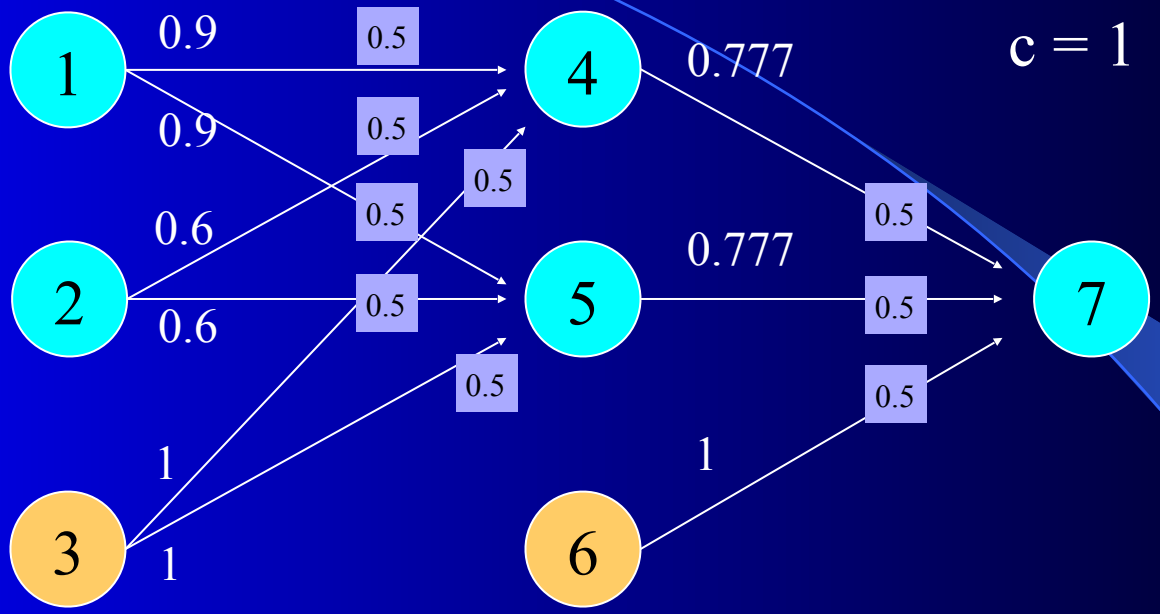
$$z_i = \frac{1}{1 + e^{-net_i}}$$

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

$net_5 = 1.25$

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

$z_5 = .777$



$$net_q = \sum_{i=1}^{3} x_i w_{ij}$$

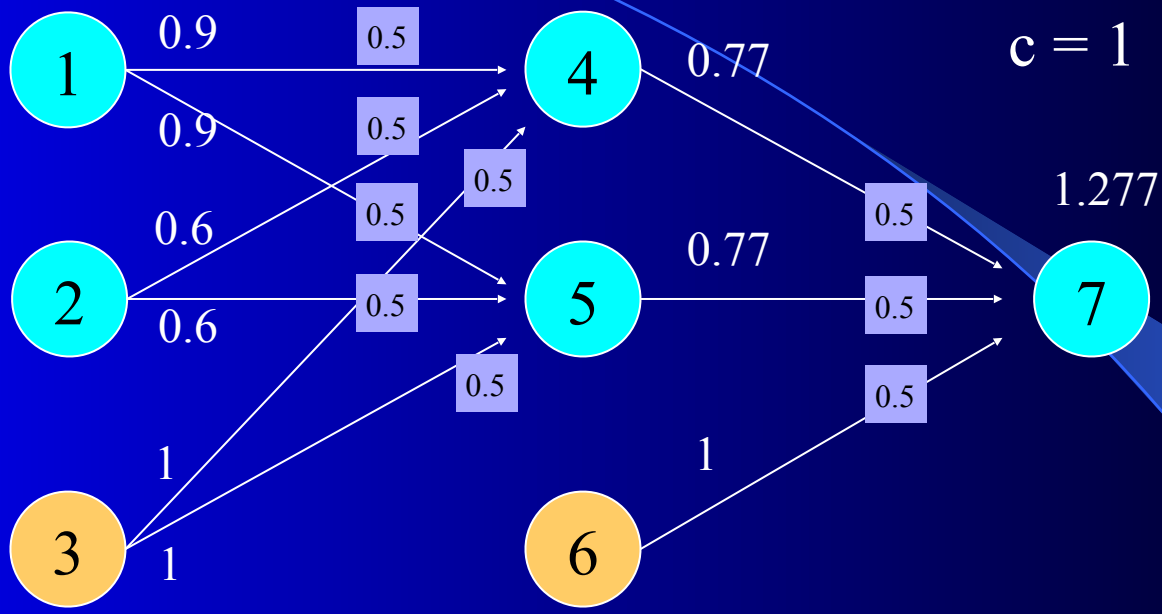
$$z_i = \frac{1}{1 + e^{-net_i}}$$

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

$net_5 = 1.25$

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

$z_5 = .777$



$$net_q = \sum_{i=1}^{3} x_i w_{ij}$$

$$z_i = \frac{1}{1 + e^{-net_i}}$$

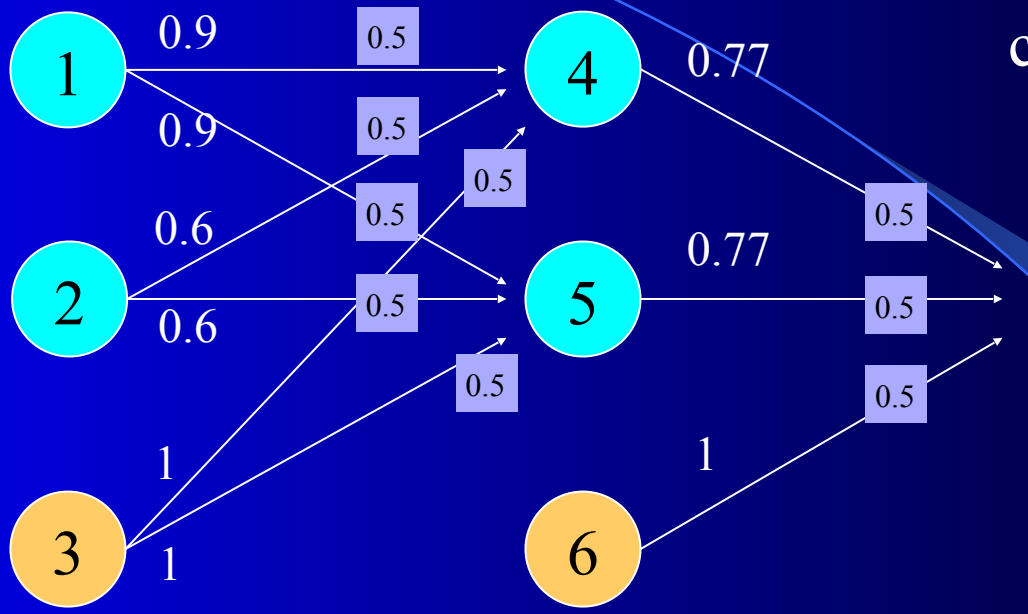
$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$

$net_5 = 1.25$

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

$z_5 = .777$

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



$$net_q = \sum_{i=1}^{3} x_i w_{ij}$$

$$z_i = \frac{1}{1 + e^{-net_i}}$$

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

$net_5 = 1.25$

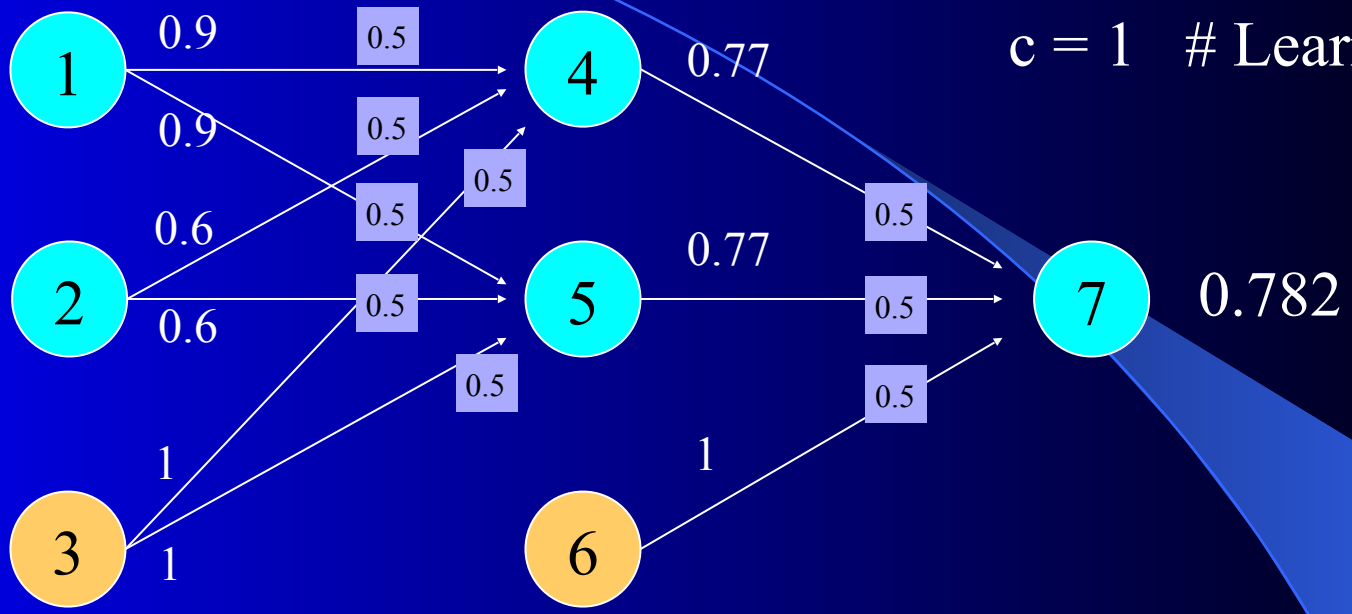
$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$

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

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

Calculate δ_j



Output node – only one

$$\delta_j = (t_j - z_j) \cdot z_j \cdot (1 - z_j)$$

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

Hidden nodes – only have node 7 following

$$\delta_i = \left(\sum_{k \in follows(i)} \delta_k w_{ik} \right) \cdot z_i \cdot (1 - z_i)$$

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

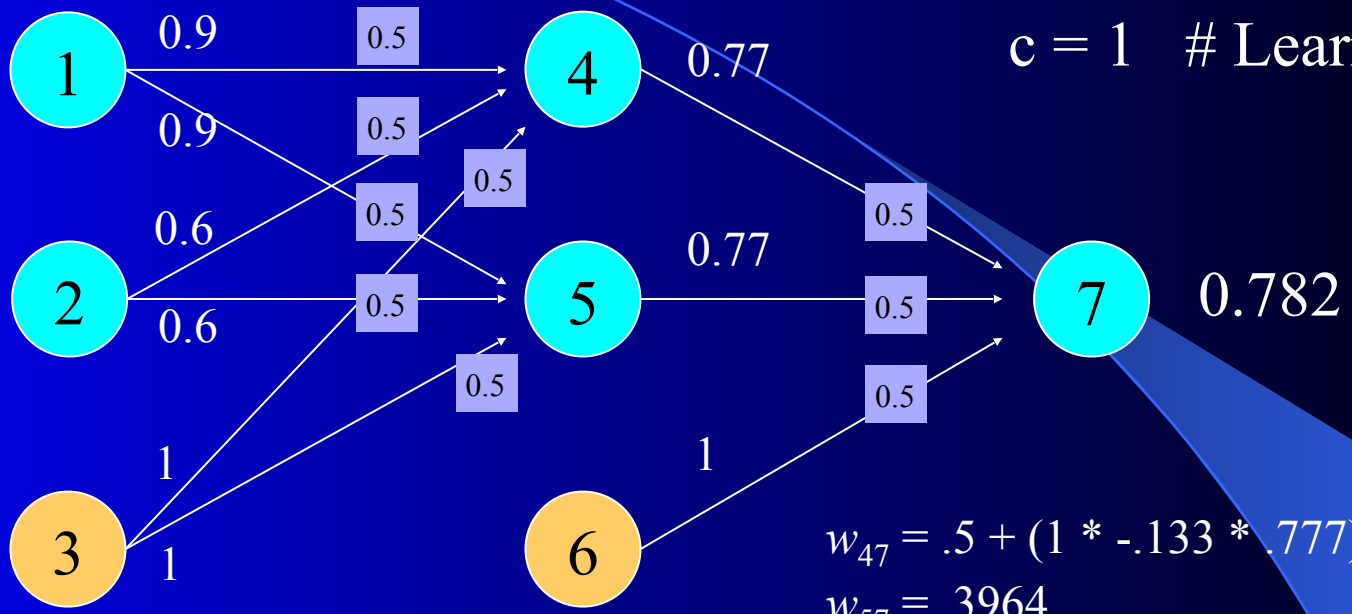
$\delta_5 = -.0115$

Calculate new weights

0.5 Weights

t = 0

c = 1 # Learning Rate



$$w_{ij} = w_{ij} + c \cdot \delta_j \cdot z_i$$

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

$w_{57} = .3964$

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

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

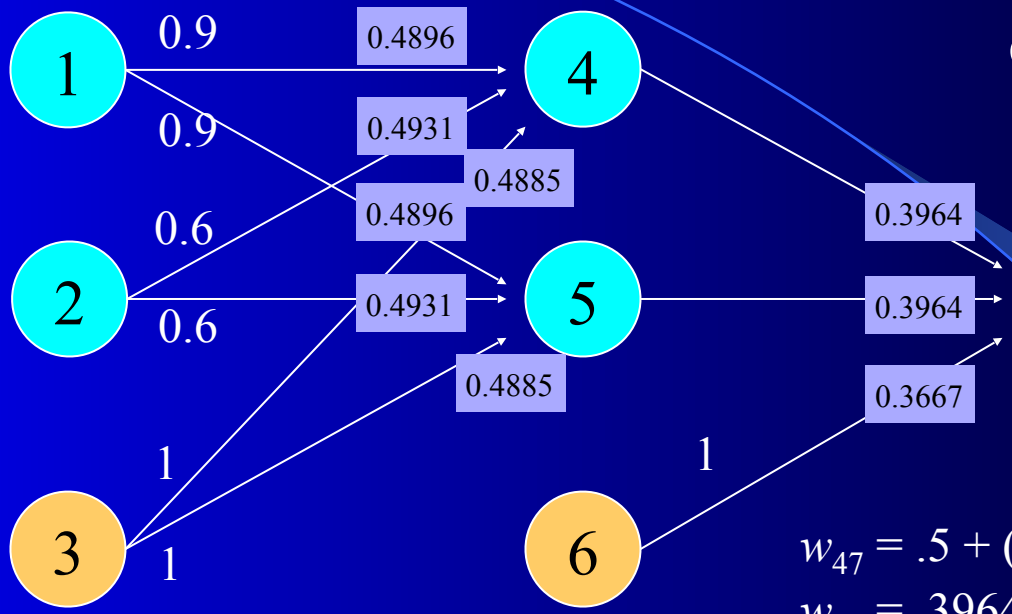
$w_{15} = .4896$

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

$w_{25} = .4931$

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

$w_{35} = .4885$



0.5 Weights

t = 0

c = 1 # Learning Rate

$$w_{ij} = w_{ij} + c \cdot \delta_j \cdot z_i$$

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

$w_{57} = .3964$

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

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

$w_{15} = .4896$

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

$w_{25} = .4931$

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

$w_{35} = .4885$

Backpropagation Learning Example

$net_4 = .9 * .5 + .6 * .5 + 1 * .5 = 1.25$
$net_5 = 1.25$

$z_4 = 1/(1 + e^{-1.25}) = .777$
$z_5 = .777$

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

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

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

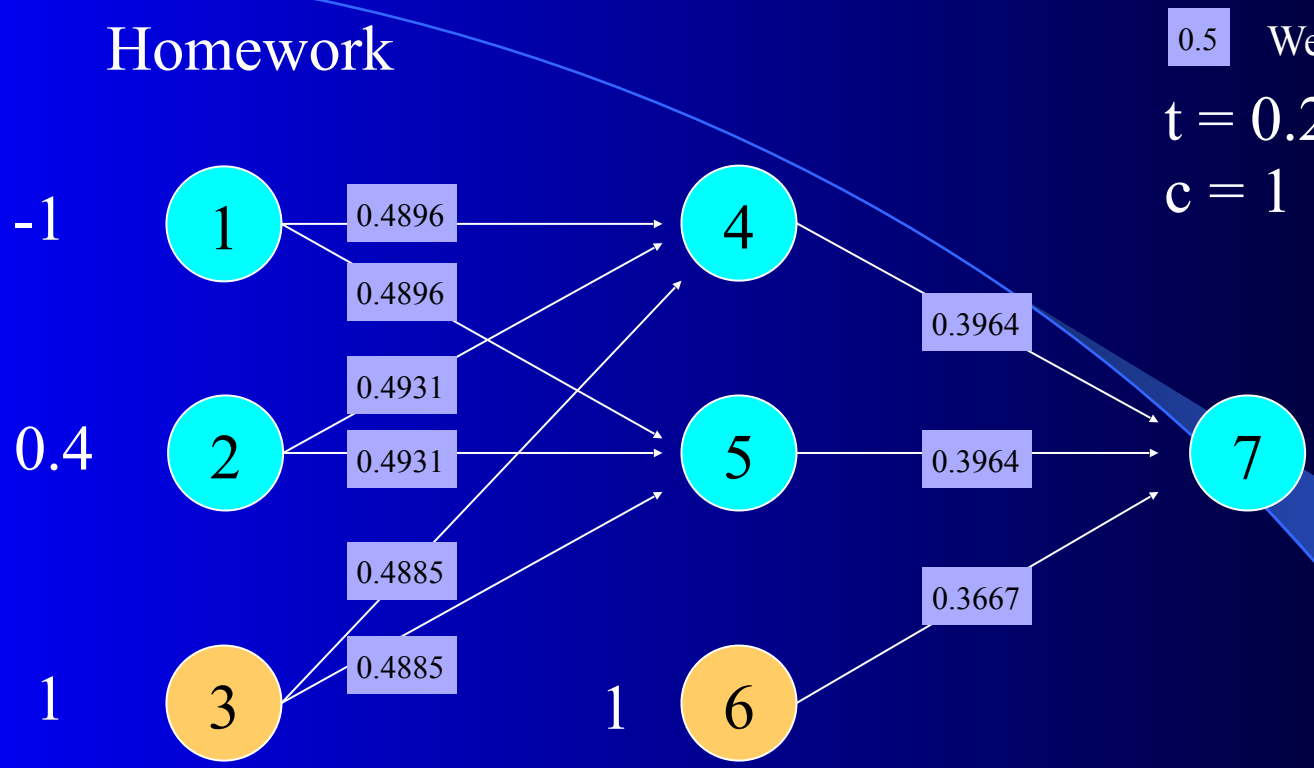
$w_{47} = .5 + (1 * -.133 * .777) = .3964$
$w_{57} = .3964$
$w_{67} = .5 + (1 * -.133 * 1) = .3667$

$w_{14} = .5 + (1 * -. 0115 * .9) = . 4896$
$w_{15} = .4896$

$w_{24} = .5 + (1 * -. 0115 * .6) = . 4931$
$w_{25} = .4931$

$w_{34} = .5 + (1 * -. 0115 * 1) = .4885$
$w_{35} = .4885$

Homework



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

- For your homework, update the weights for a second instance -1 .4 ==> .2. Continue using the updated weights shown on the previous slide. Show your work like we did on the previous slide.

- Then go to the link below: Neural Network Playground 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/

MLP/Backpropagation Summary

- Excellent empirical results
- Scaling – The pleasant surprise
  - Local minima very rare as problem and network complexity increase
- Hyper-parameters usually handled by trial and error
- Many variants
  - Adaptive Parameters, ontogenic (growing and pruning) learning algorithms
  - Many different architectures
    - Recurrent networks
    - Deep networks
  - An active research area