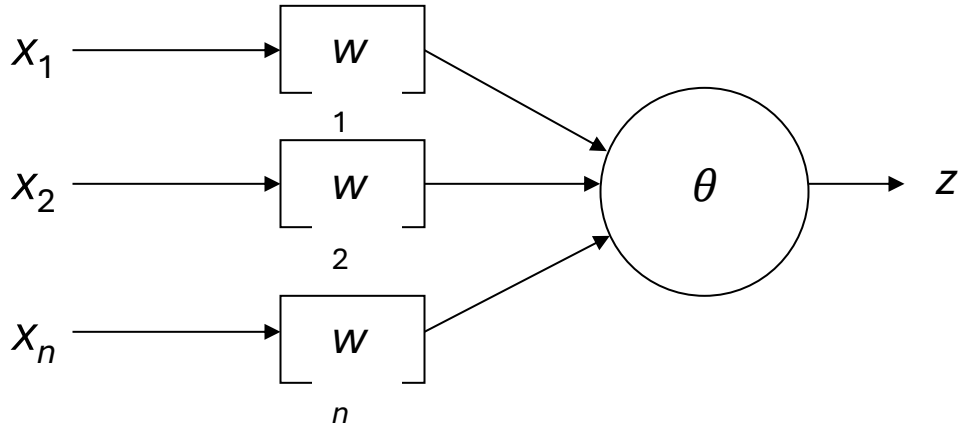
Backprop Math

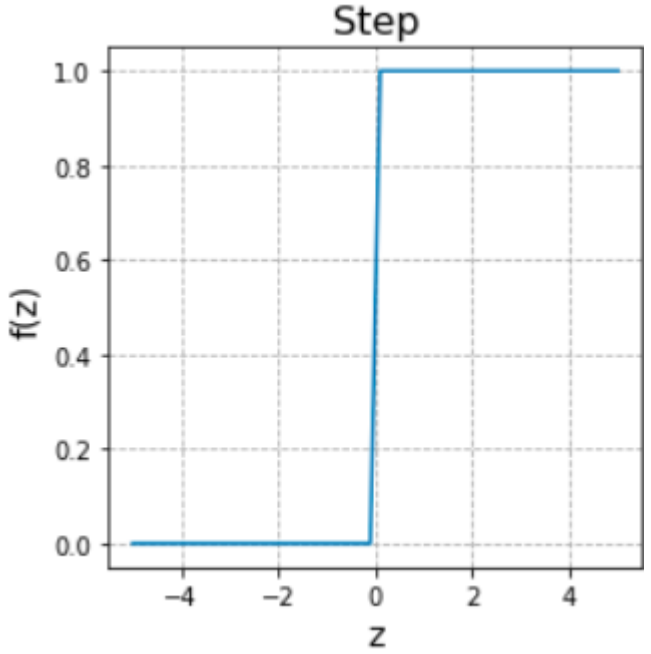
12 March 2026

Alex Lyman

Review

Perceptron Node – Threshold Logic Unit





- x_1, x_2, x_n Inputs
- w_1, w_2, w_n Weights
- θ Threshold Function
- z Output

$$z = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i w_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{n} x_i w_i < \theta \end{cases}$$

Perceptron Rule

$$\Delta w_i = c(t - z)\, x_i$$

- Where w_i is the weight from input i to the perceptron node,
  - c is the learning rate, (hyperparameter)
  - t is the target for the current instance,
  - z is the current output,
  - $(t\text{-}z)$ is the error
  - x_i is i^{th} input

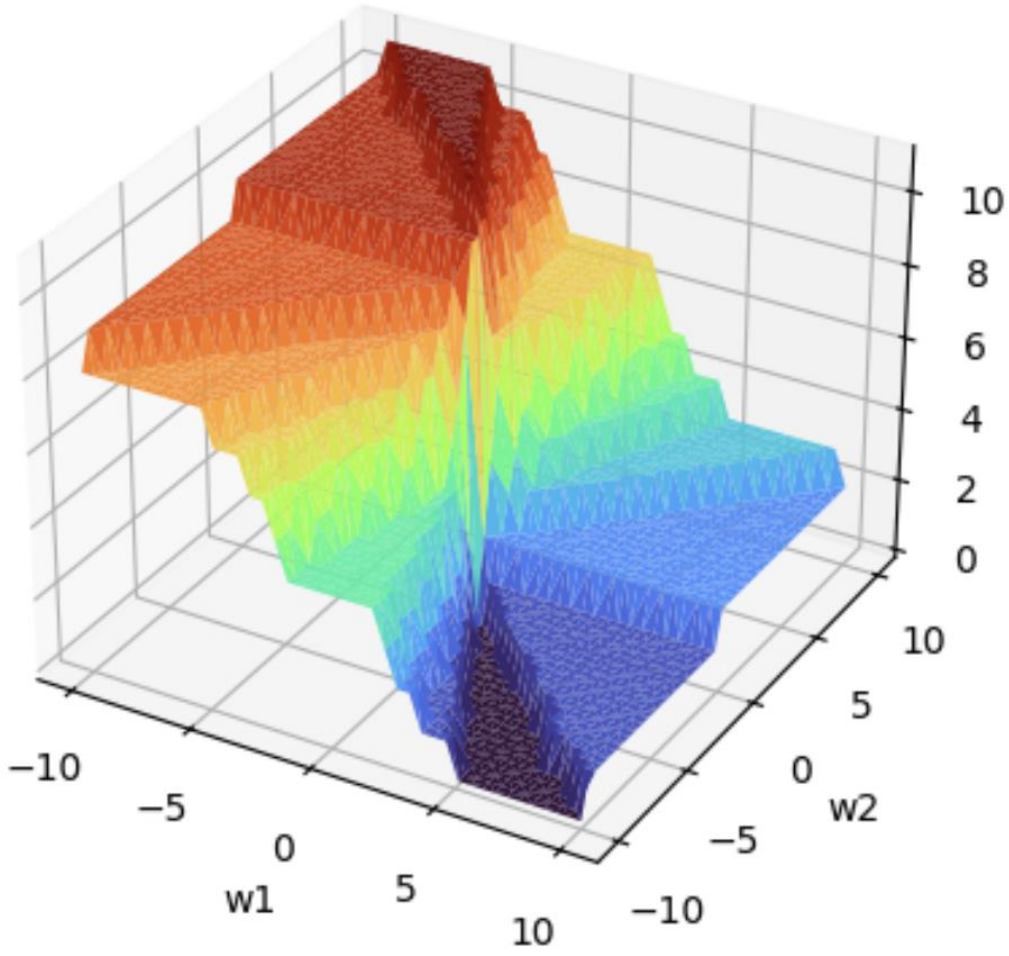
Delta Rule

$$\Delta w_i = c(t - net)\, x_i$$

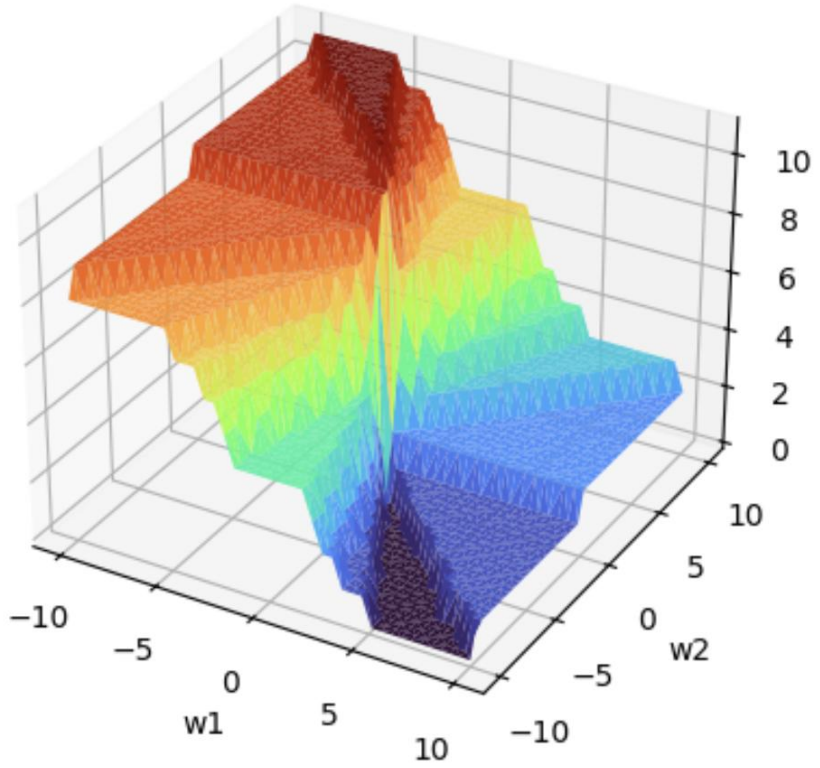
- Where w_i is the weight from input i to the perceptron node,
  - c is the learning rate, (hyperparameter)
  - t is the target for the current instance,
  - z is the net (BEFORE THE THRESHOLD FUNCTION),
  - *(t-net)* is the error
  - x_i is i^{th} input

Error Surface

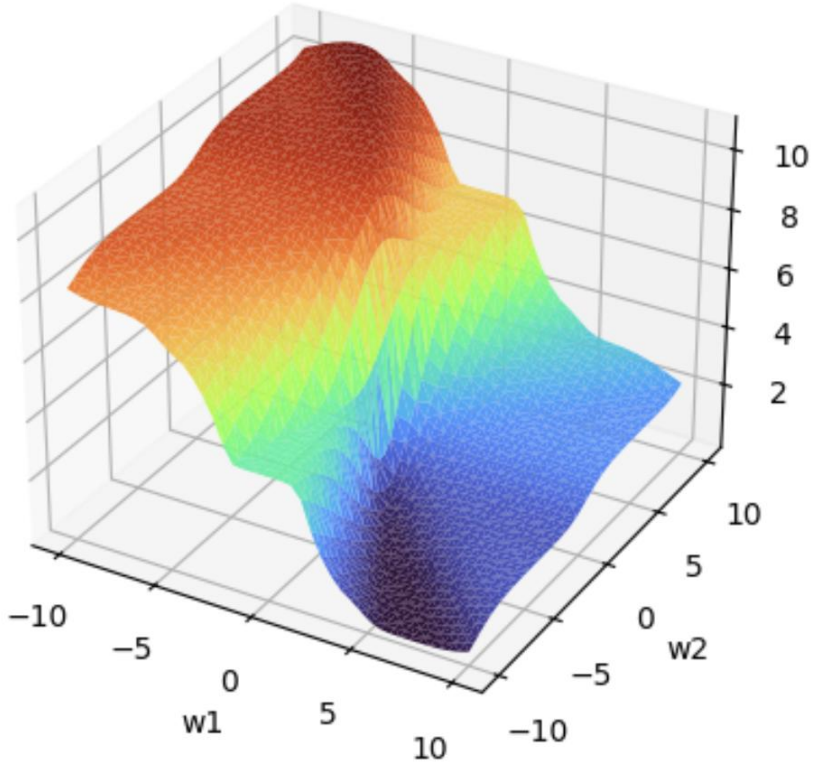
Total SSE: Sum Squared Error $\Sigma\,(t-z)^2$



Perceptron Rule + Step function



Delta Rule + Sigmoid(net)

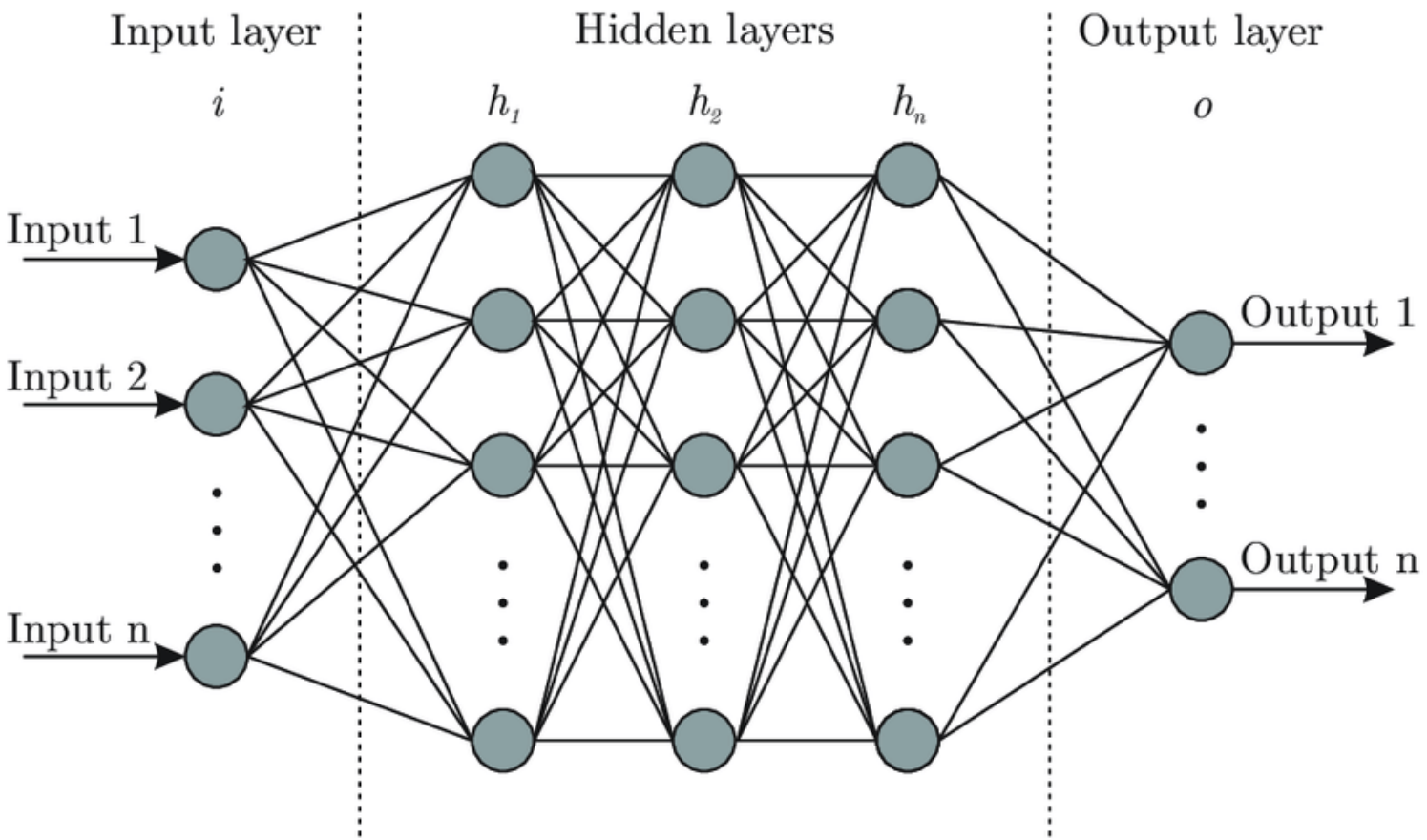


We want to use **Gradient Descent** to find the minimum **error** as a function of the **weights.** Smooth, differentiable function makes gradient descent work.

Questions?

Neural Nets

- MLP (Multi-Layer Perceptron)
- Fully Connected Network
- Feedforward Neural Network
- Artificial Neural Network (ANN)

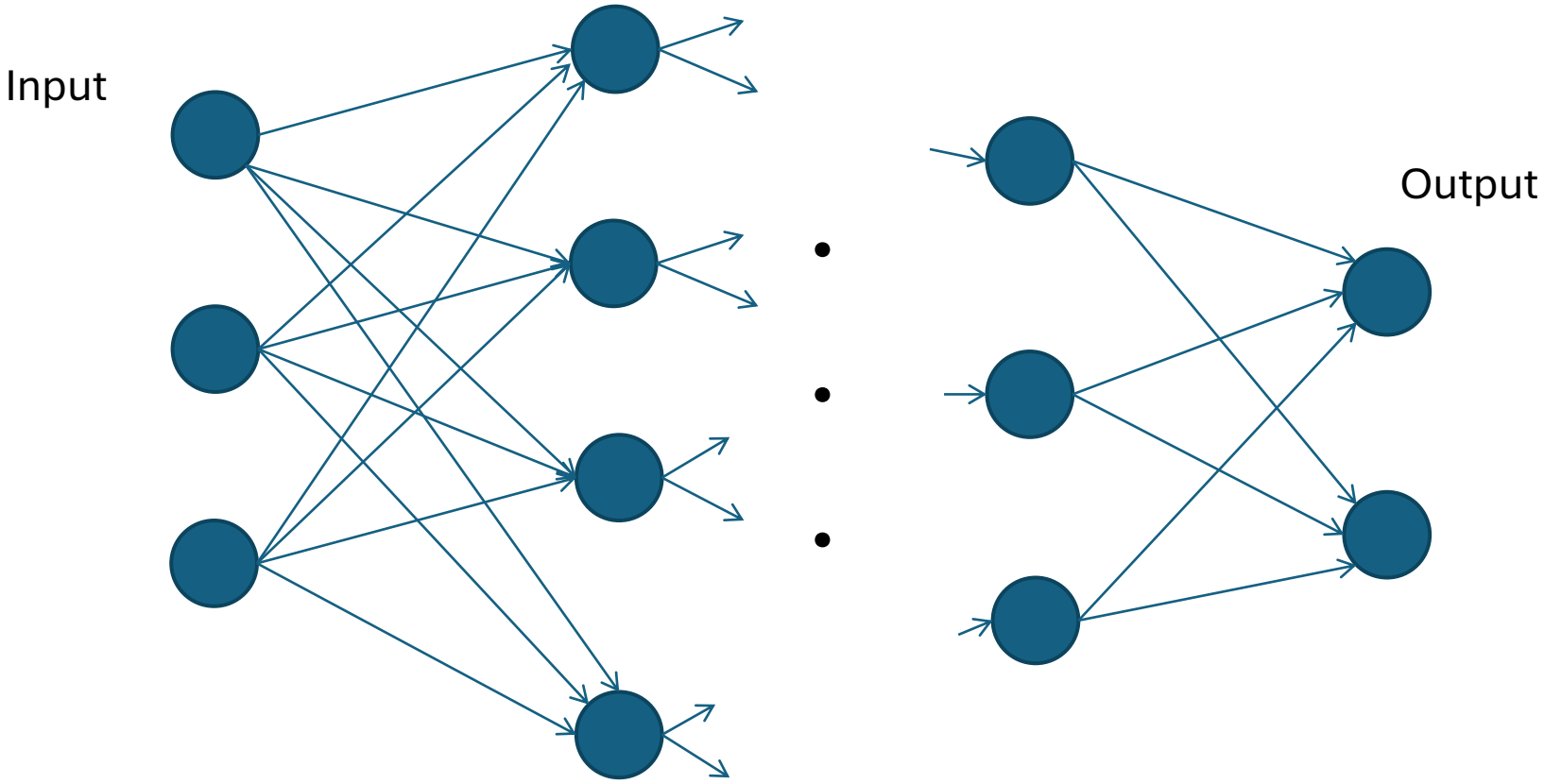


Break

Backpropagation

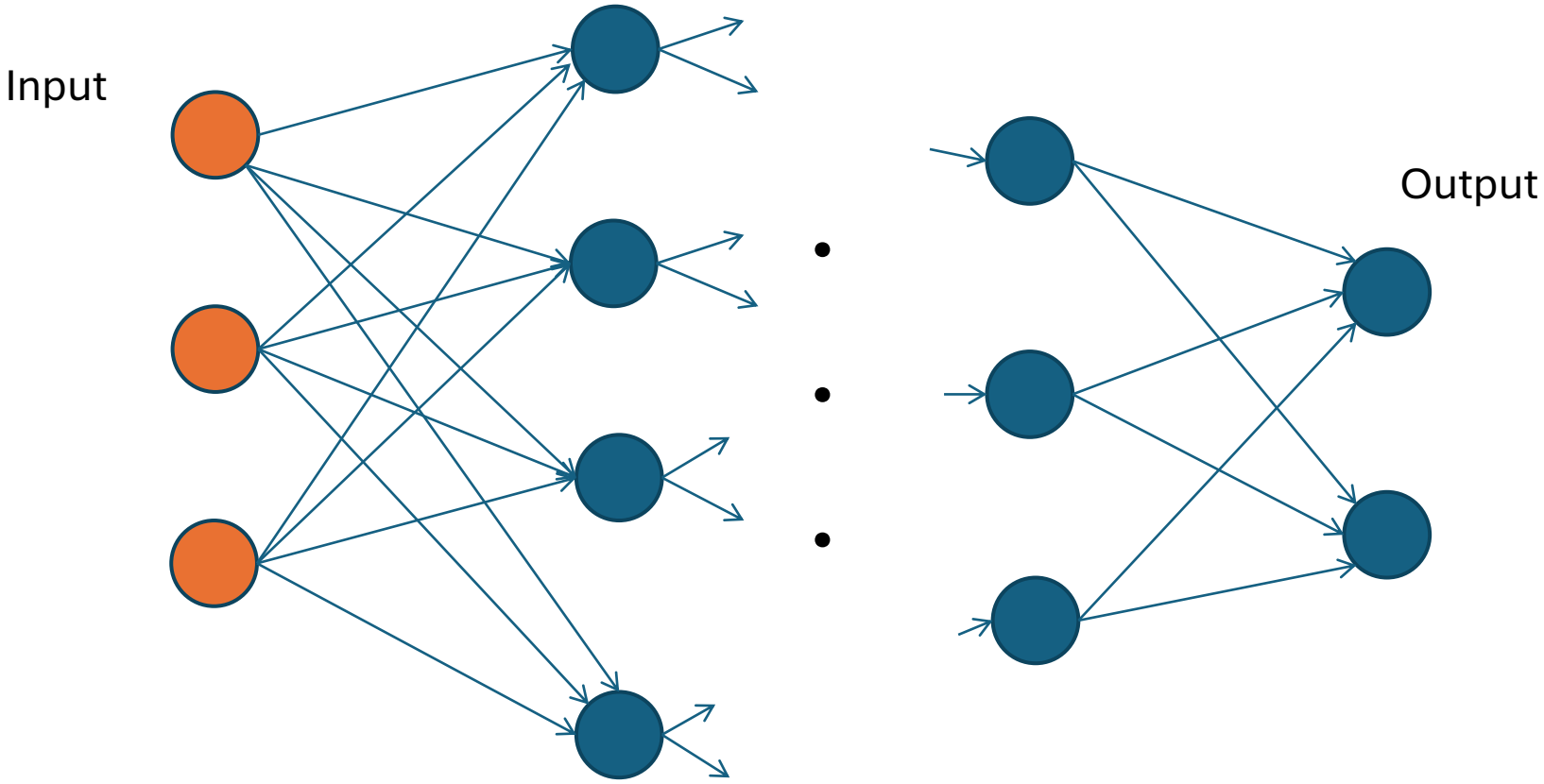
Backpropagation – How we train NNs

- Operates similarly to perceptron learning



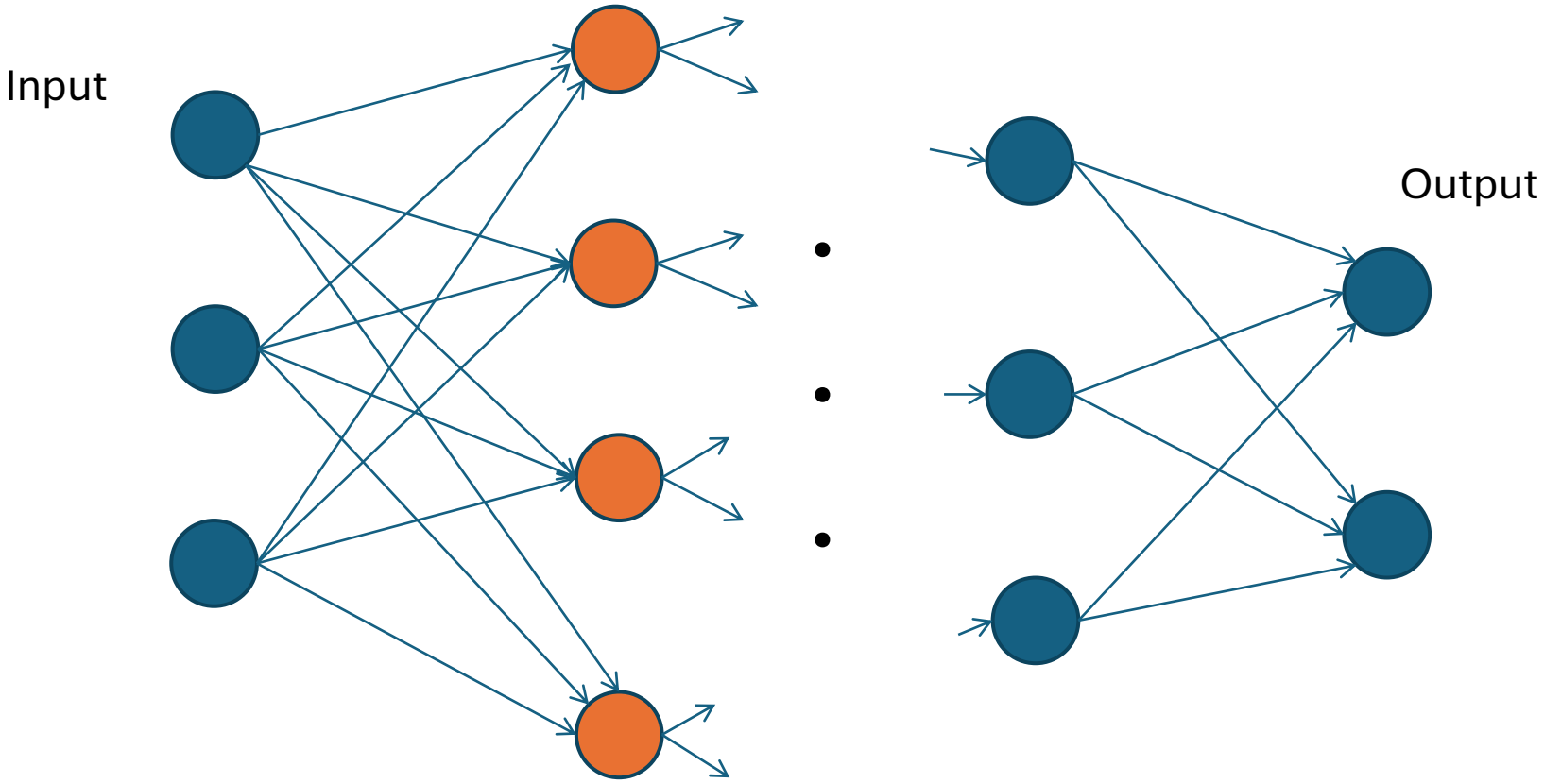
Backpropagation

- Inputs are fed forward through the network



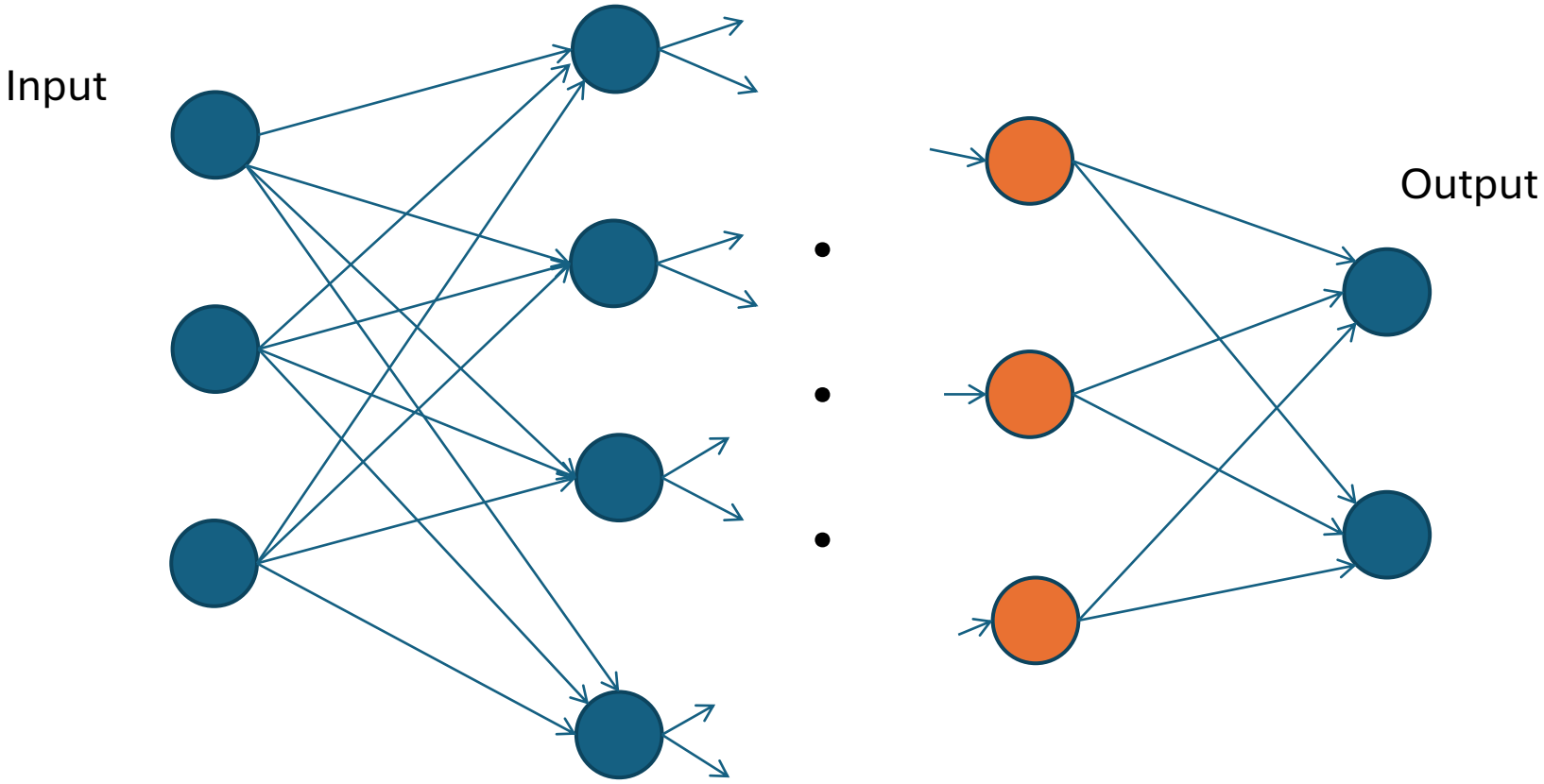
Backpropagation

- Inputs are fed forward through the network



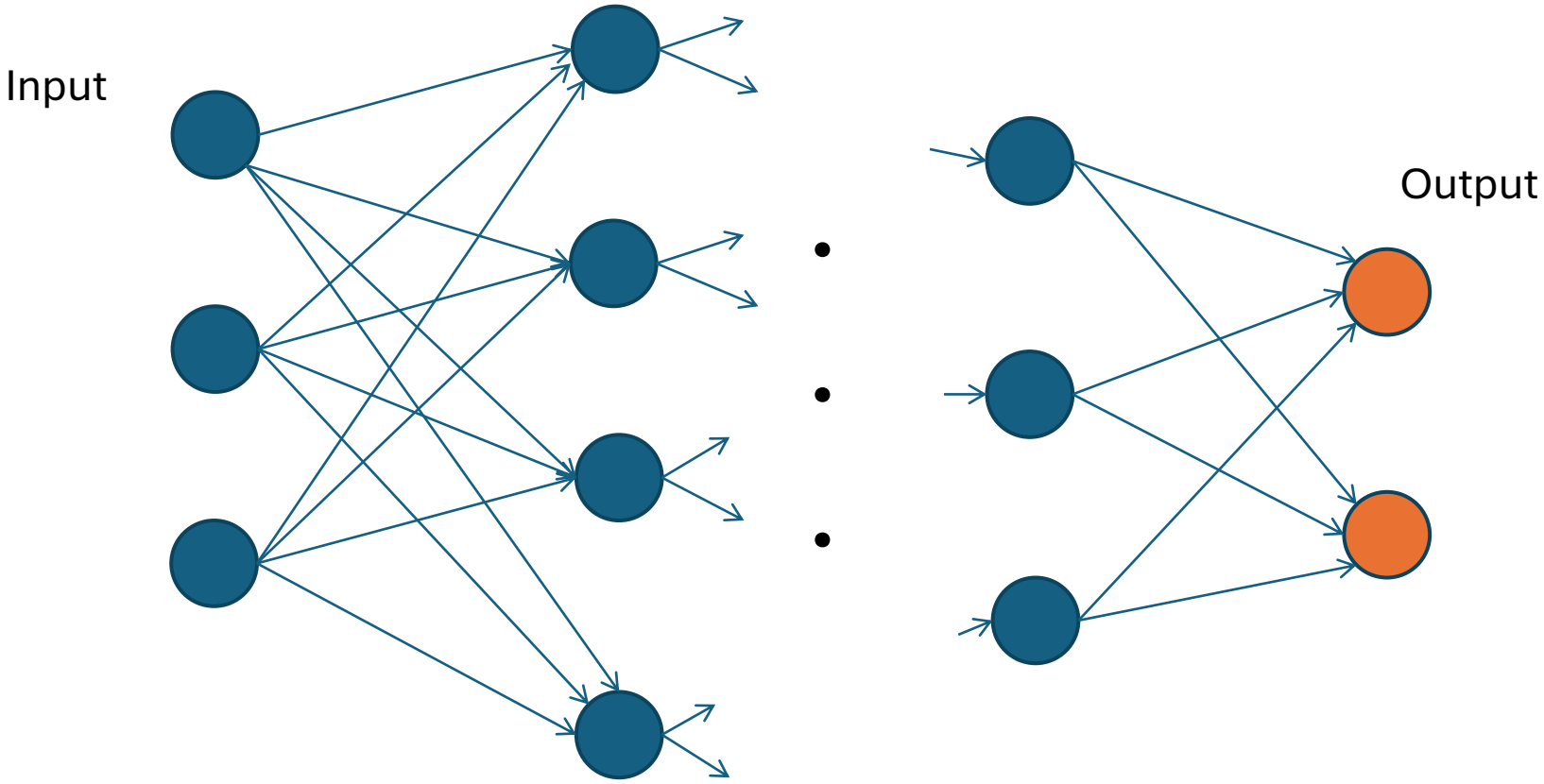
Backpropagation

- Inputs are fed forward through the network



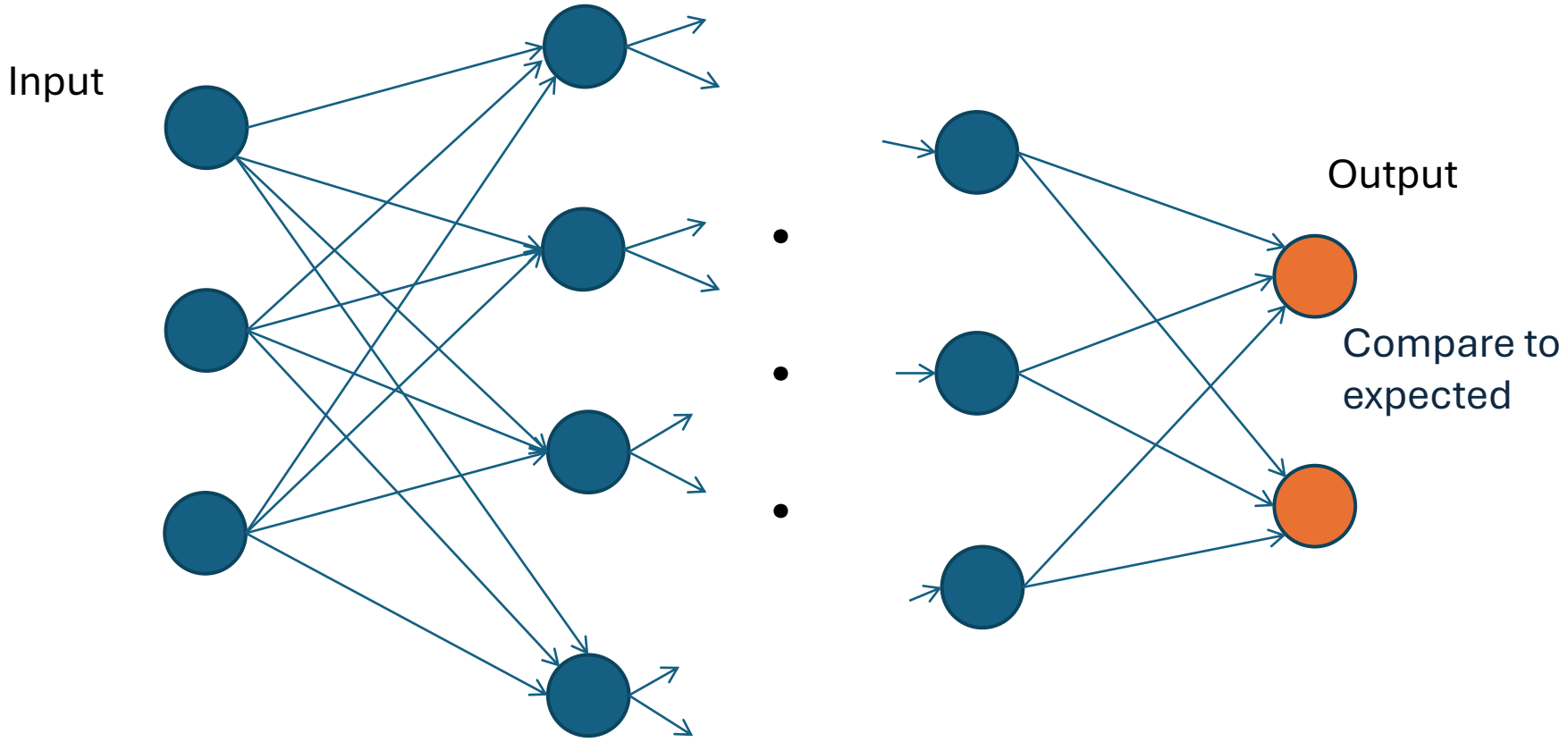
Backpropagation

- Inputs are fed forward through the network



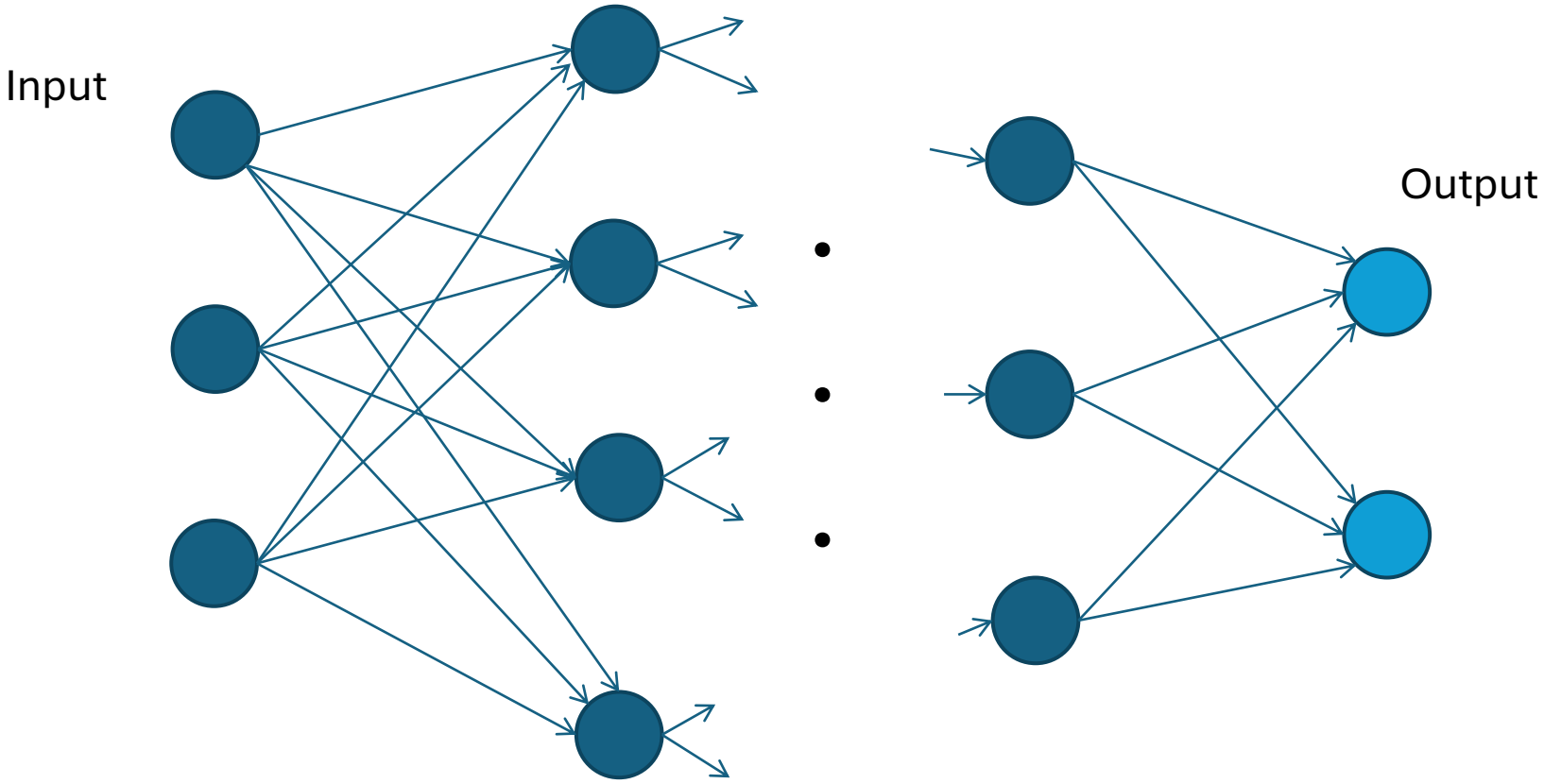
Backpropagation

- Inputs are fed forward through the network



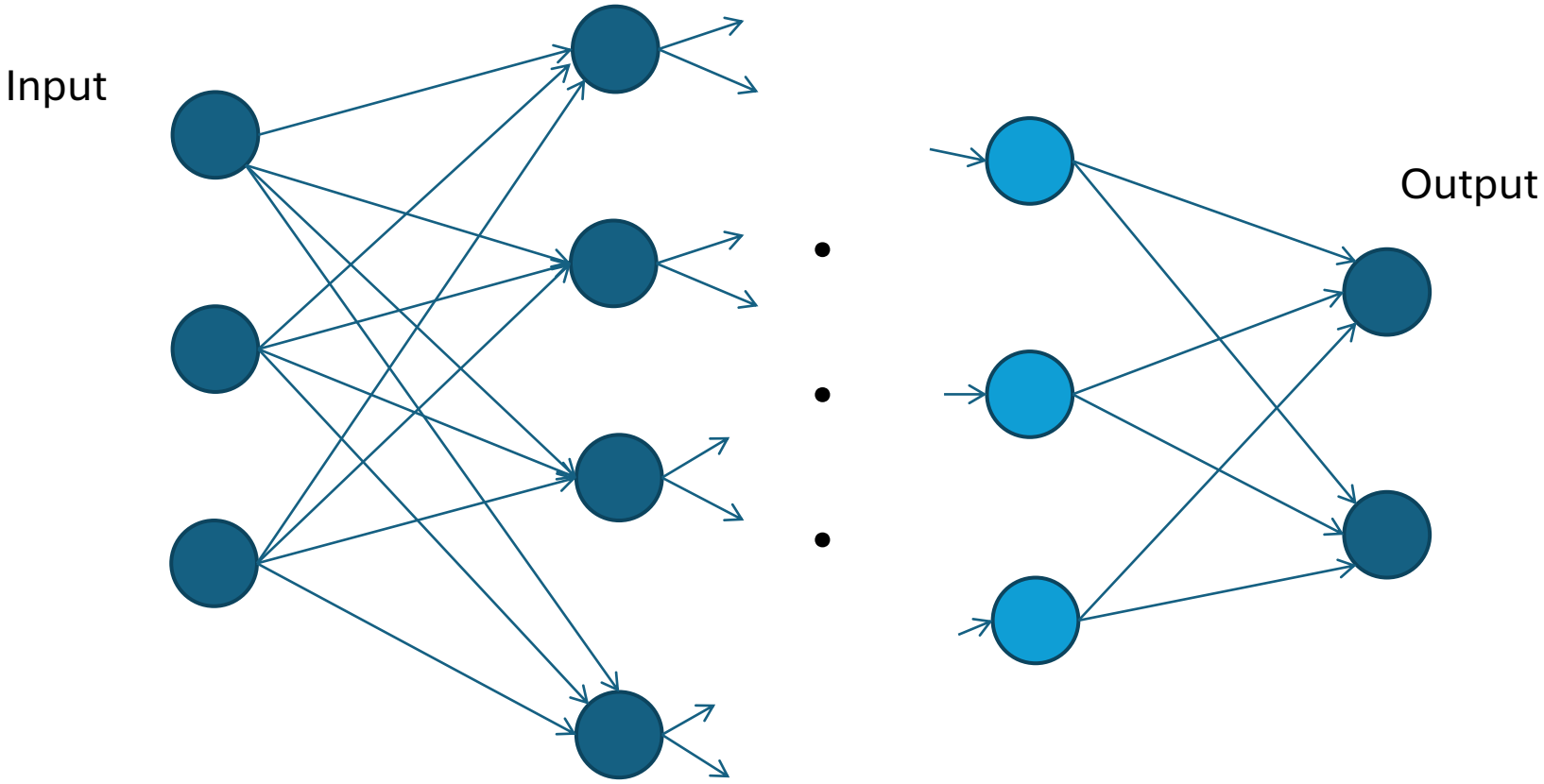
Backpropagation

- Errors are propagated back



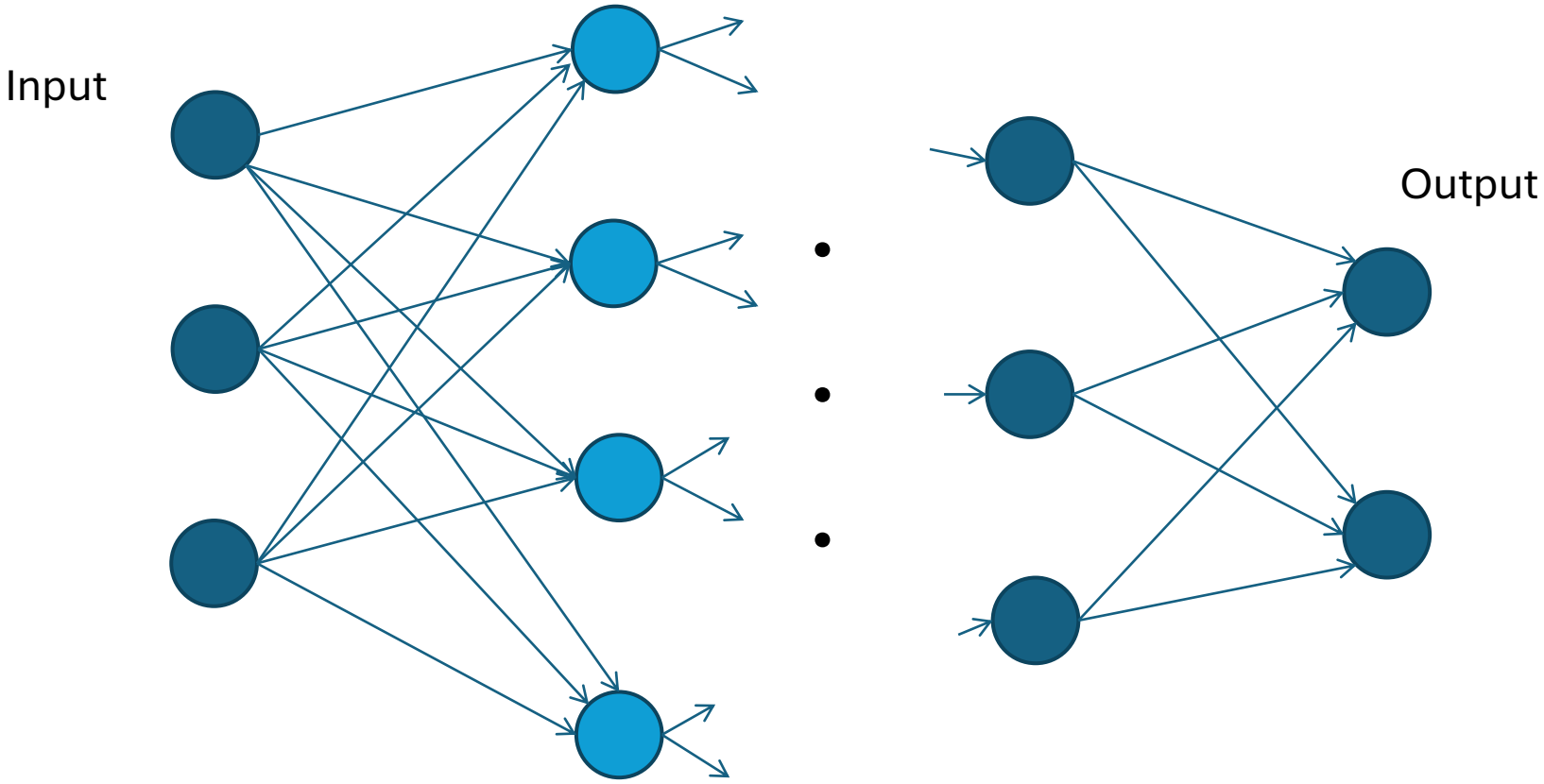
Backpropagation

- Errors are propagated back



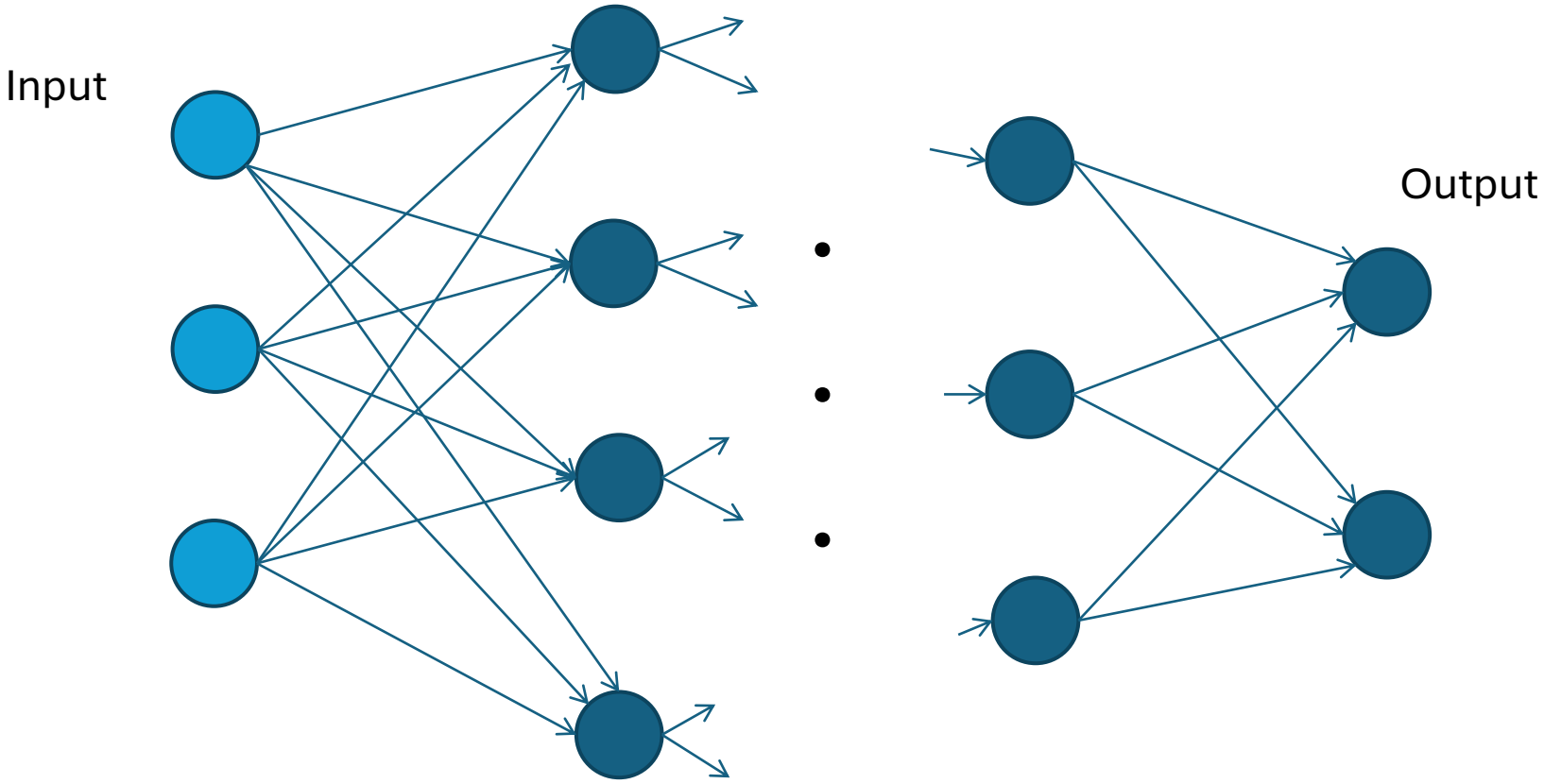
Backpropagation

- Errors are propagated back



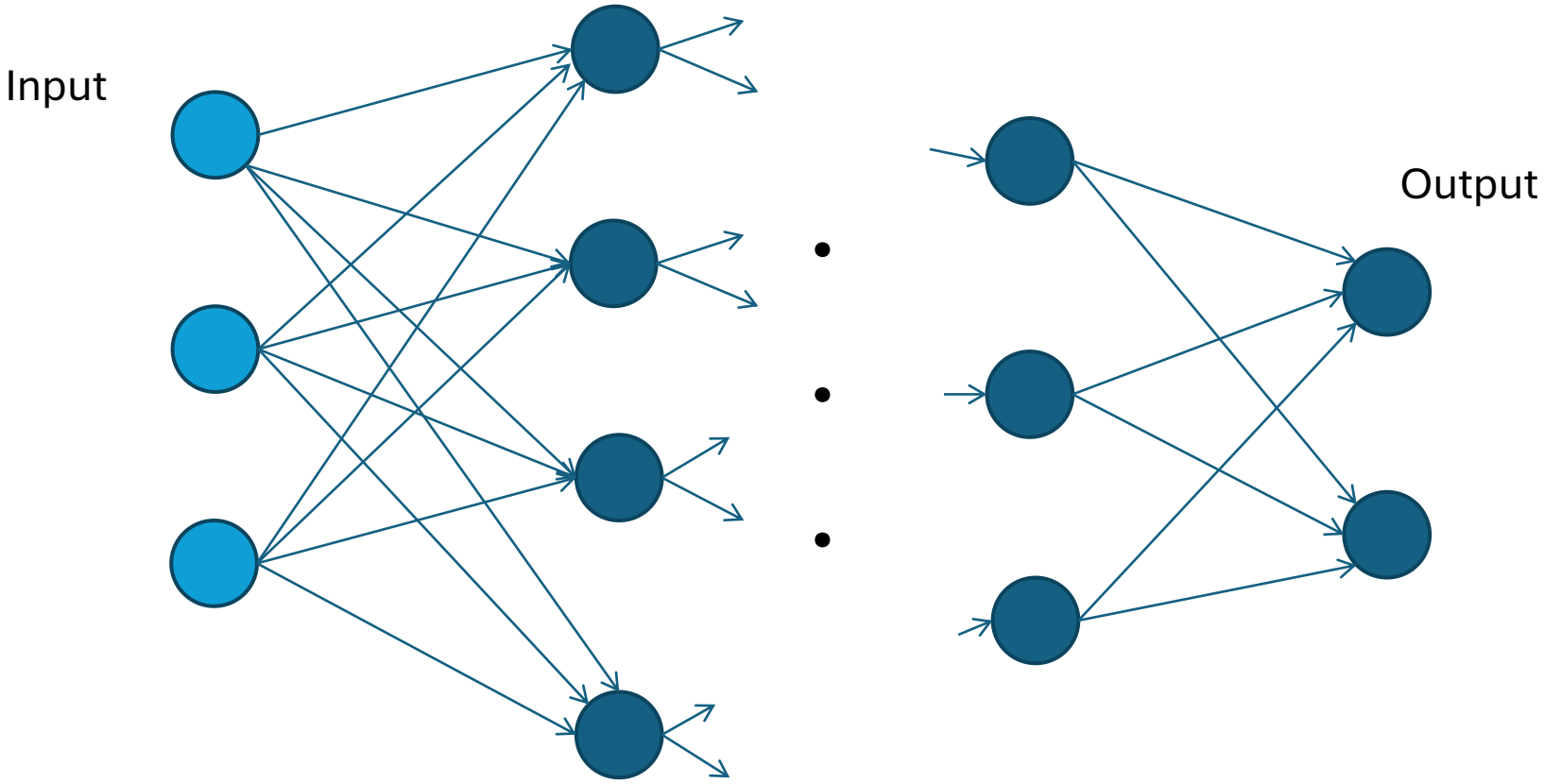
Backpropagation

- Errors are propagated back



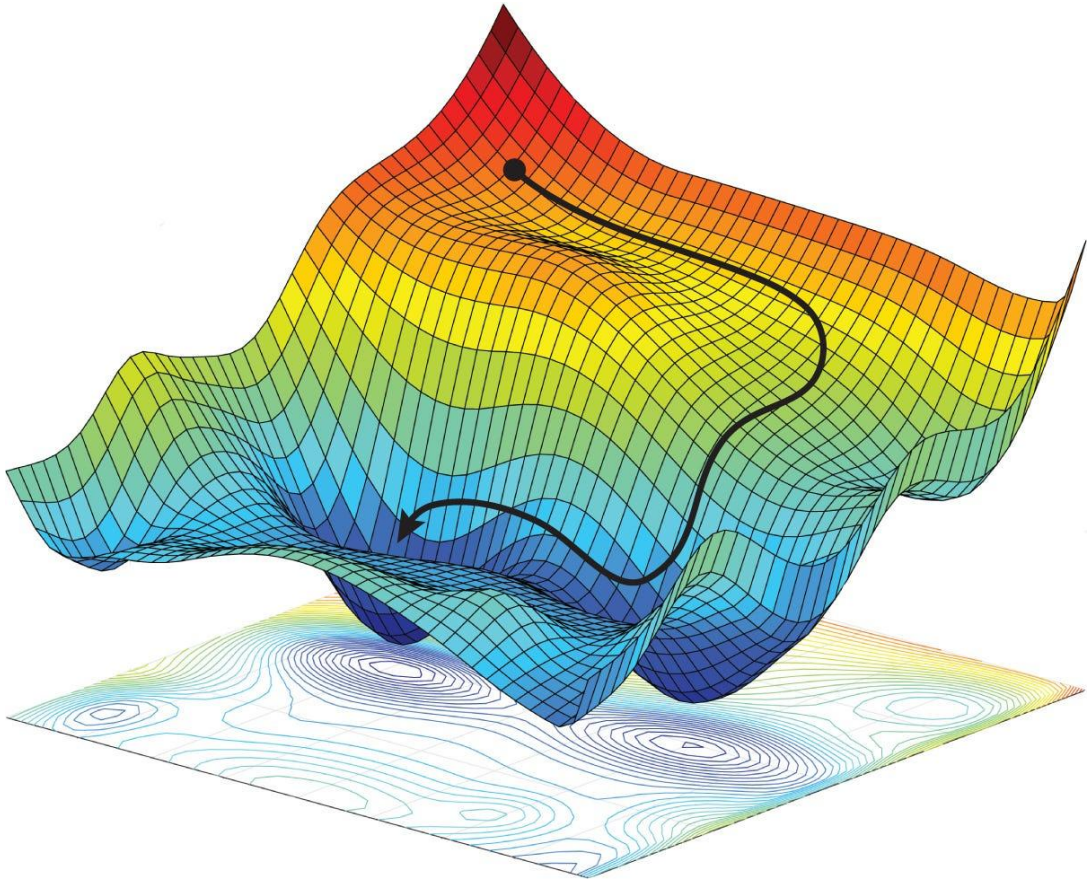
Backpropagation

- Adjust weights based on errors



Backpropagation Math

Backpropagation



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.

Derivative

91 languages

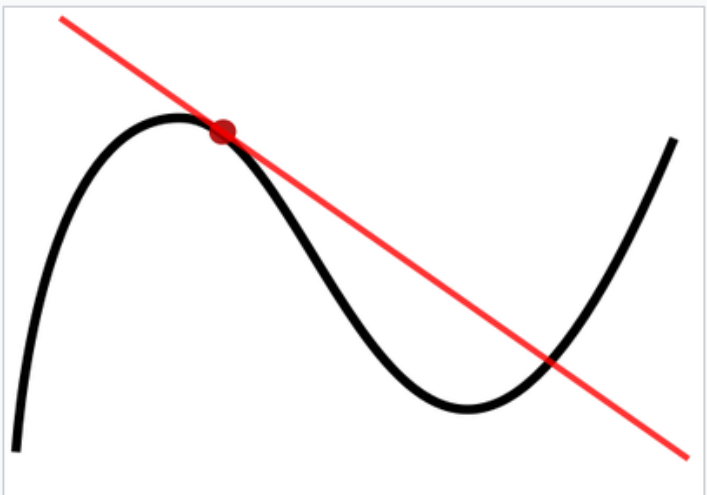
Article Talk Read View source View history Tools

From Wikipedia, the free encyclopedia

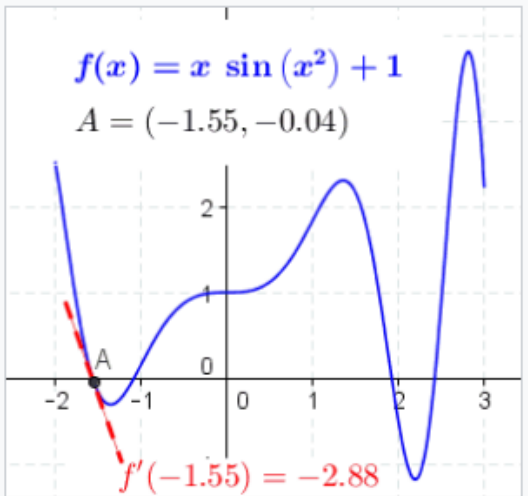
For the algebraic generalization, see Derivation (differential algebra). For other uses, see Derivative (disambiguation).

In mathematics, the **derivative** is a fundamental tool that quantifies the sensitivity to change of a function's output with respect to its input. The derivative of a function of a single variable at a chosen input value, when it exists, is the slope of the tangent line to the graph of the function at that point. The tangent line is the best linear approximation of the function near that input value. The derivative is often described as the **instantaneous rate of change**, the ratio of the instantaneous change in the dependent variable to that of the independent variable.[1] The process of finding a derivative is called **differentiation**.

There are multiple different notations for differentiation. *Leibniz*



The graph of a function, drawn in black, and a tangent line to that graph, drawn in red. The slope of the tangent line is equal to the derivative of the function at the marked point.

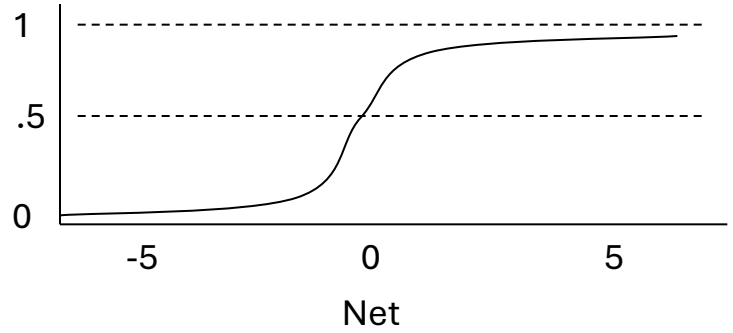


The derivative at different points of a differentiable function. In this case, the derivative is equal to $\sin(x^2) + 2x^2 \cos(x^2)$.

Activation Function and its Derivative

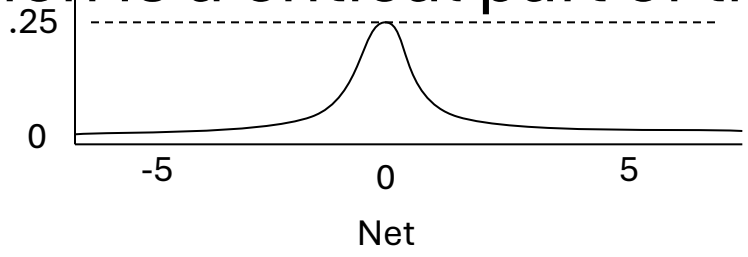
- Node activation function *f(net)* is commonly the sigmoid

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



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

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

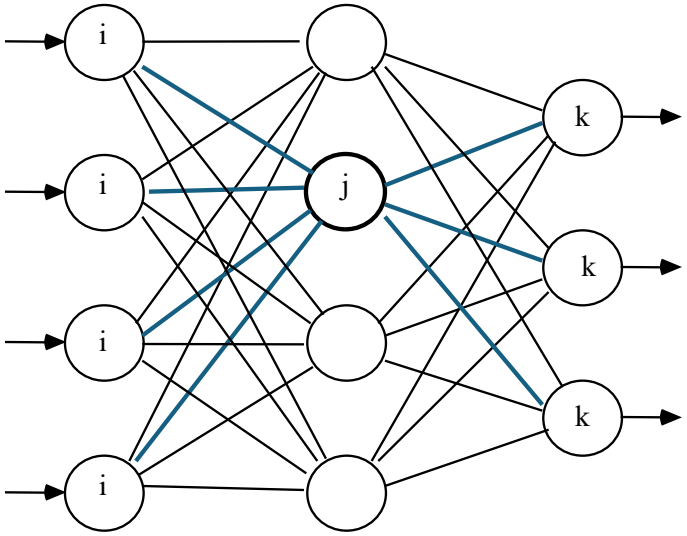


Backpropagation Learning Equations

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

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

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



C – Learning Rate
Z – Input
δ (lowercase delta) – Blame – how much is this node responsible for the final error.

(Tj−Zj) – Target minus Output (we've seen this before.
$f'(net_j)$ – Derivative of activation function Output node is just delta rule!

$\sum_k (\delta_k w_{jk})$ – For each node you're connected to, you ask 'how much blame did you contribute (δ_k) and how strong was our connection (w_{jk}).

Do you remember Calculus?

Differentiation Rules	
Constant Rule	$\frac{d}{dx}[c] = 0$
Power Rule	$\frac{d}{dx}x^n = nx^{n-1}$
Product Rule	$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$
Quotient Rule	$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Chain Rule	$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$

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

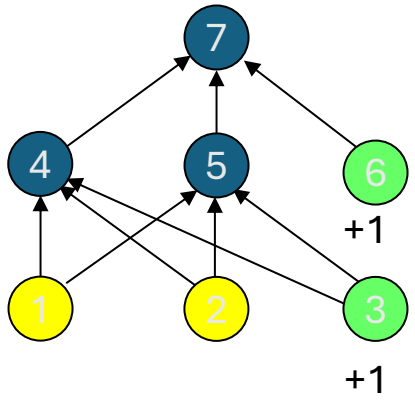
A Few Differences:
Textbook uses $(y_k - t_k)$
slides use $(T_k - Z_k)$

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

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.



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

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

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

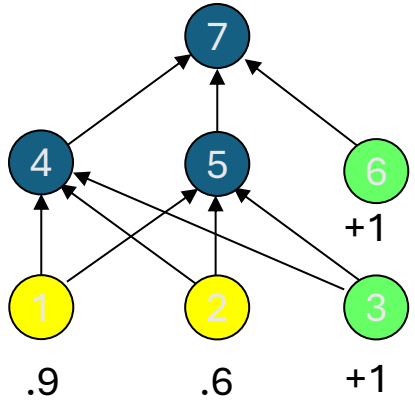
$$\delta_j = (T_j - Z_j) f'(net_j) \qquad \text{[Output Node]}$$

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

Backpropagation Learning Example

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

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



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

δ_7 = (0 - .782) * .782 * (1 - .782) = -.133
δ_4 = (-.133 * .5) * .777 * (1 - .777) = -.0115
δ_5 = -.0115

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

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

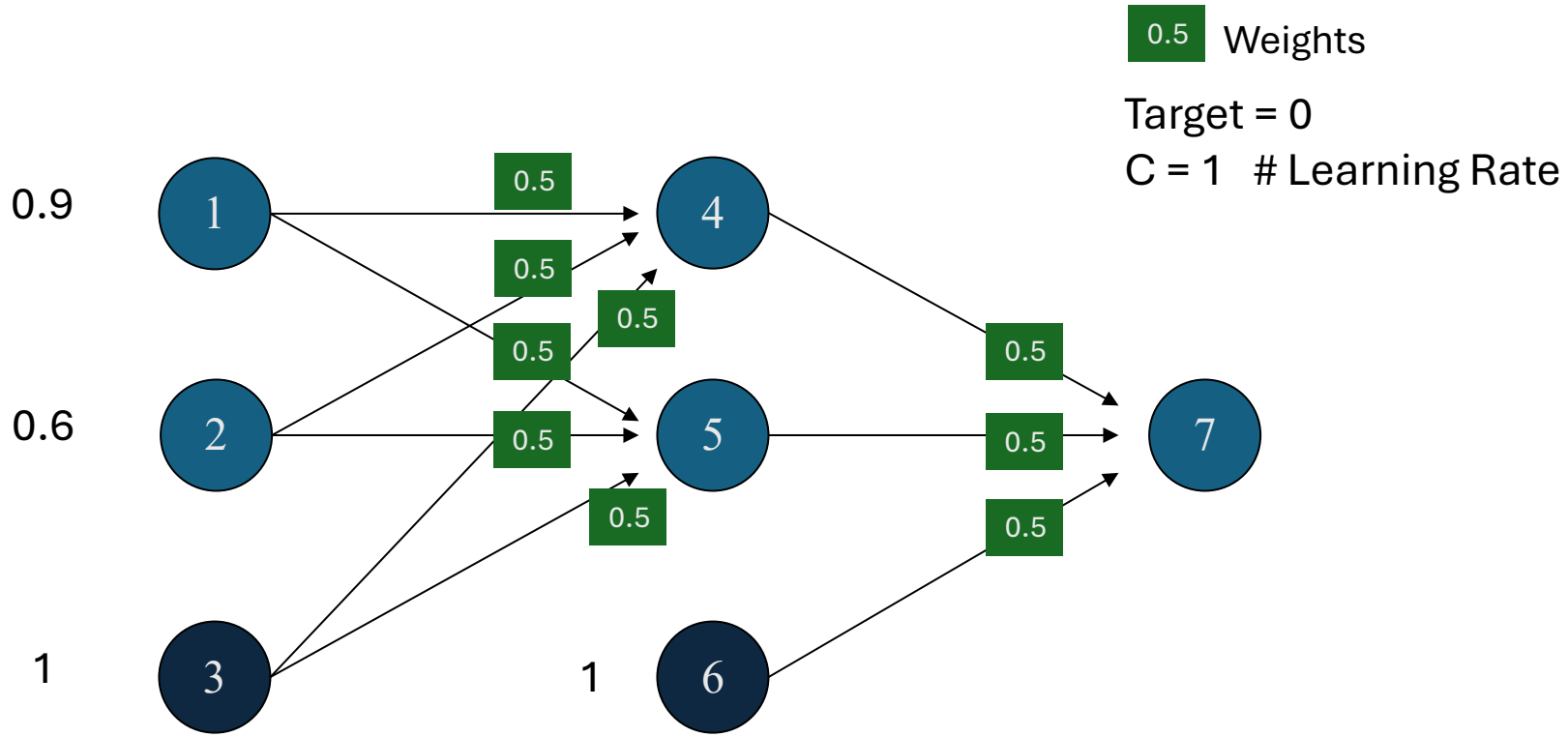
$$\delta_j = \sum_k (\delta_k w_{jk}) f'(net_j) \quad \text{[Hidden Node]}$$

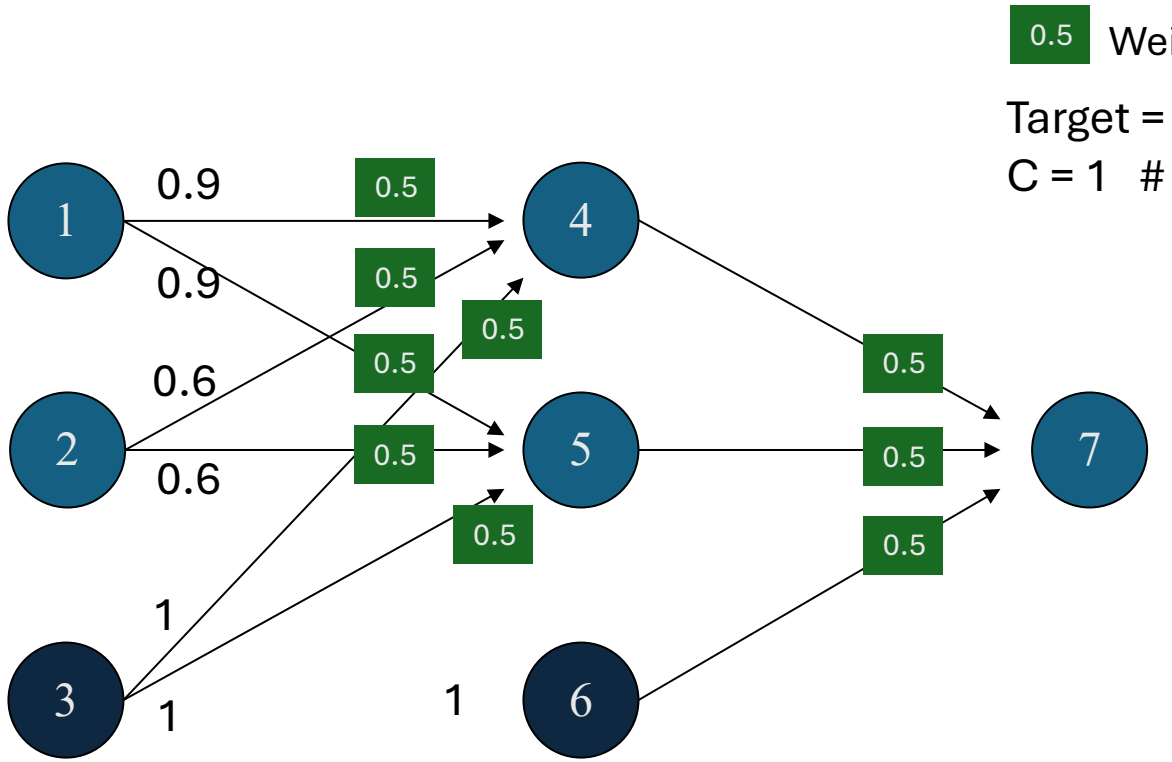
w_{47} = .5 + (1 * -.133 * .777) = .3964
w_{57} = .3964
w_{67} = .5 + (1 * -.133 * 1) = .3667

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

One More Time

Forward Pass Example
(no learning happening here)

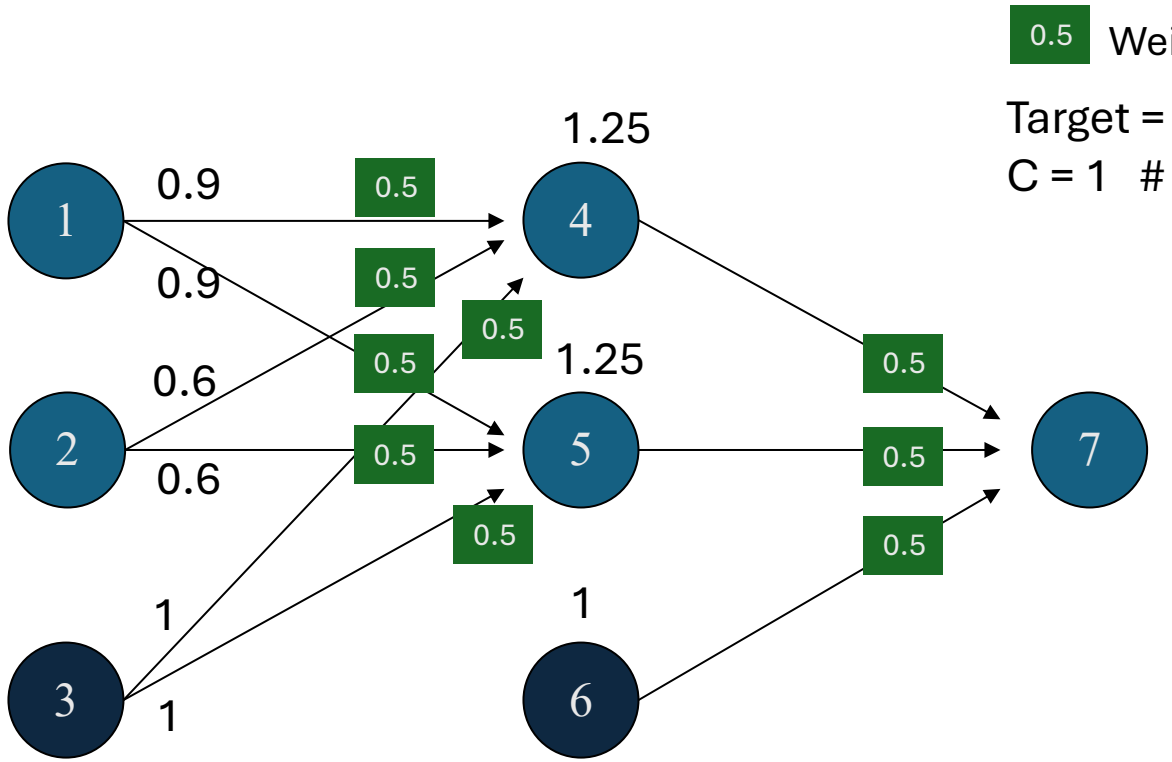

0.5 Weights
Target = 0
C = 1 # Learning Rate
0.9
0.6
1
1
1
2
3
4
5
6
7
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5



$net_q = \sum x_i w_{ij}$

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

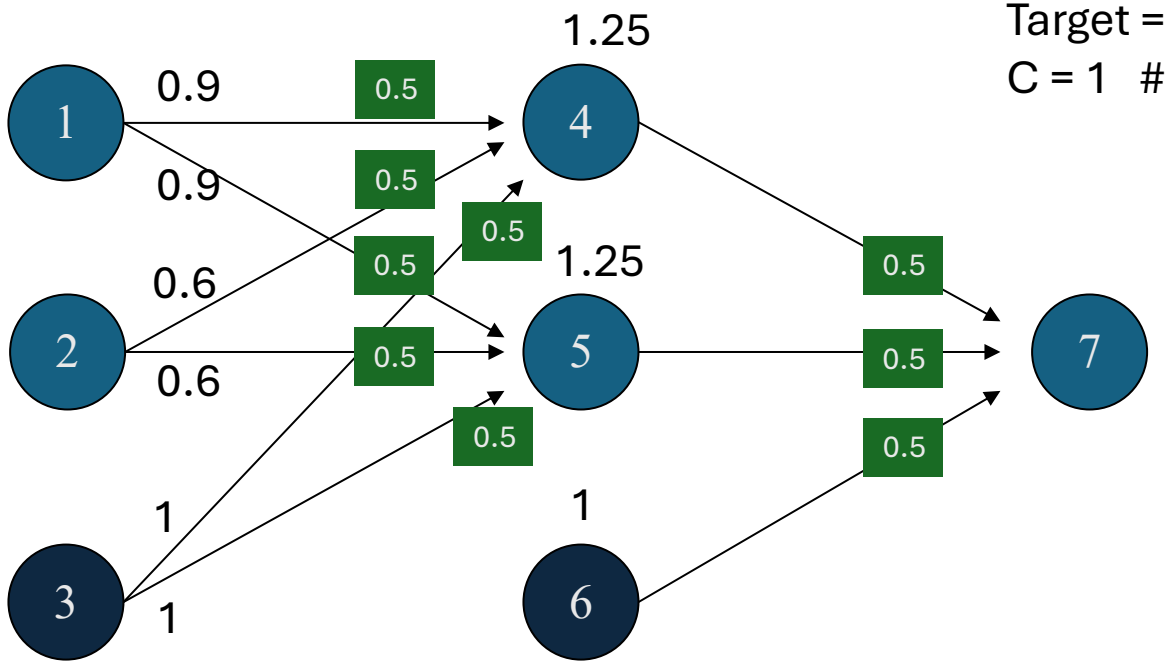
$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

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

$net_5 = 1.25$



$net_q = \sum x_i w_{ij}$

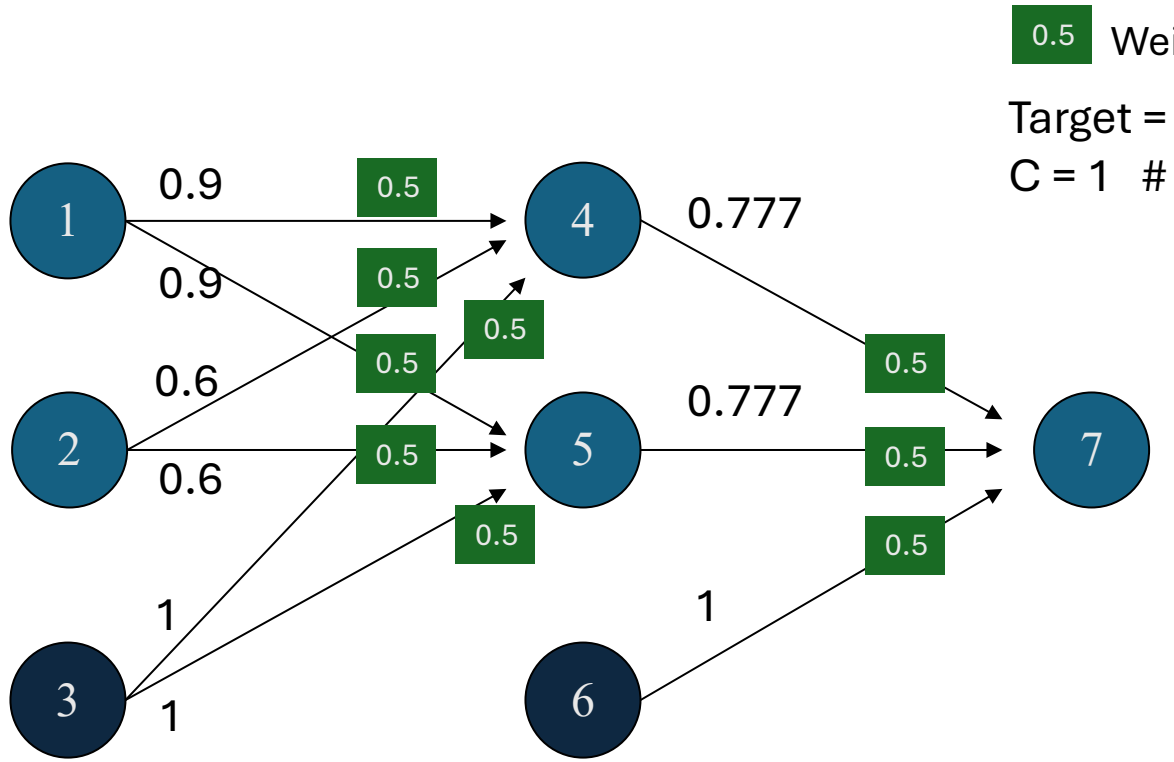
$z_i = \frac{1}{(1 + e^{-neti})}$

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



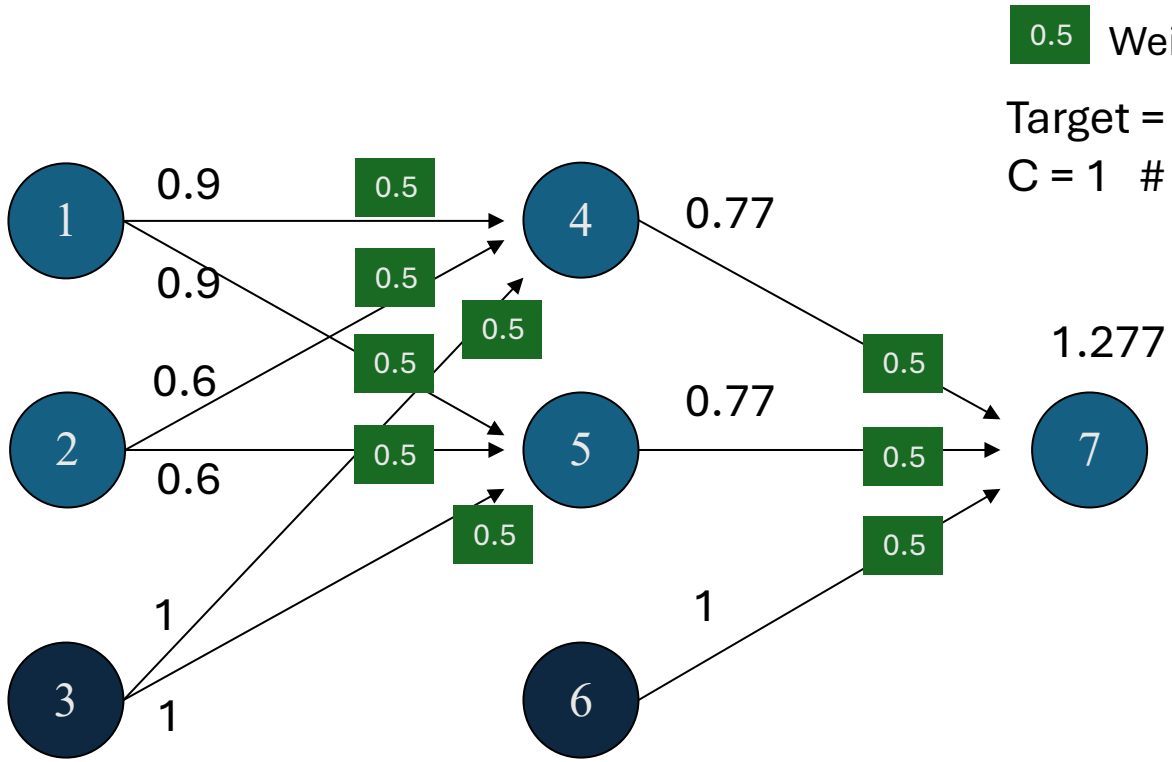
$net_q = \sum x_i w_{ij}$

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

$net_5 = 1.25$

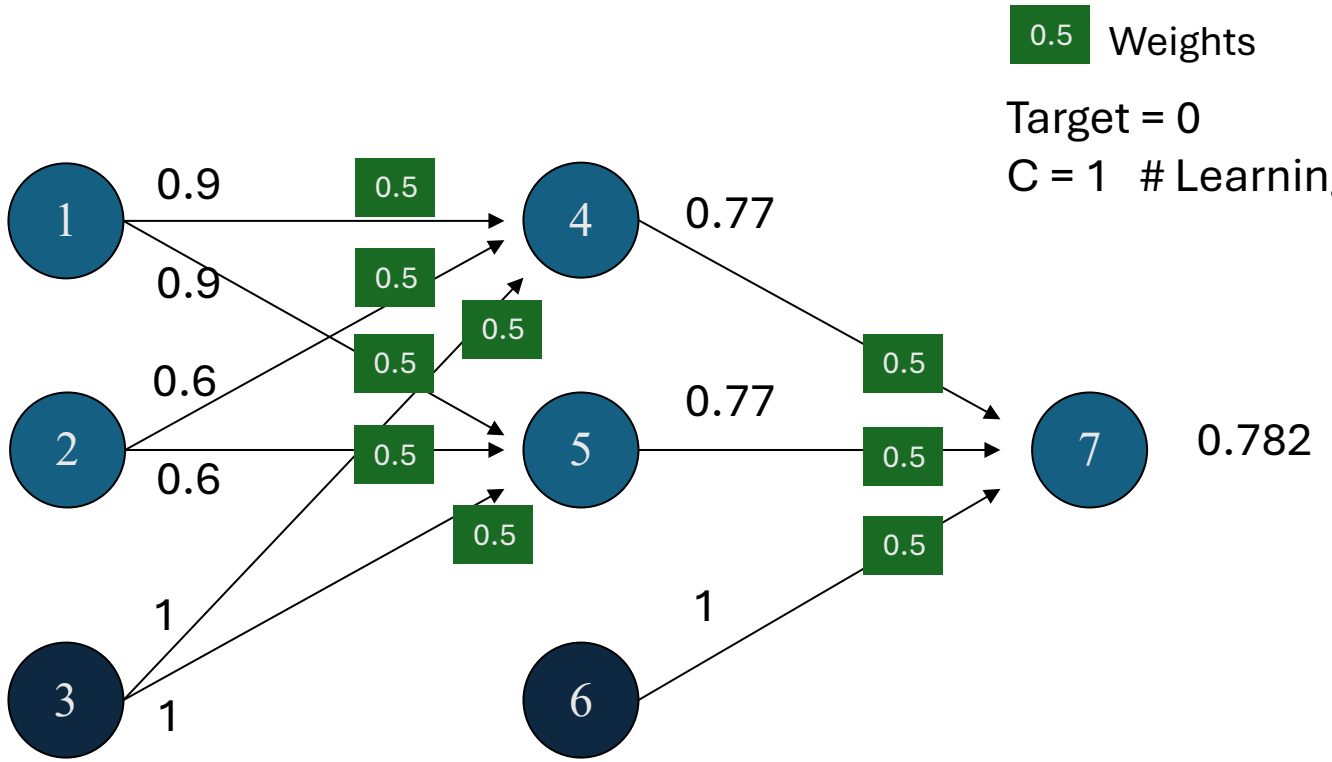
$z_4 = 1/(1 + e^{-1.25}) = .777$

$z_5 = .777$



$net_q = \sum x_i w_{ij}$

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



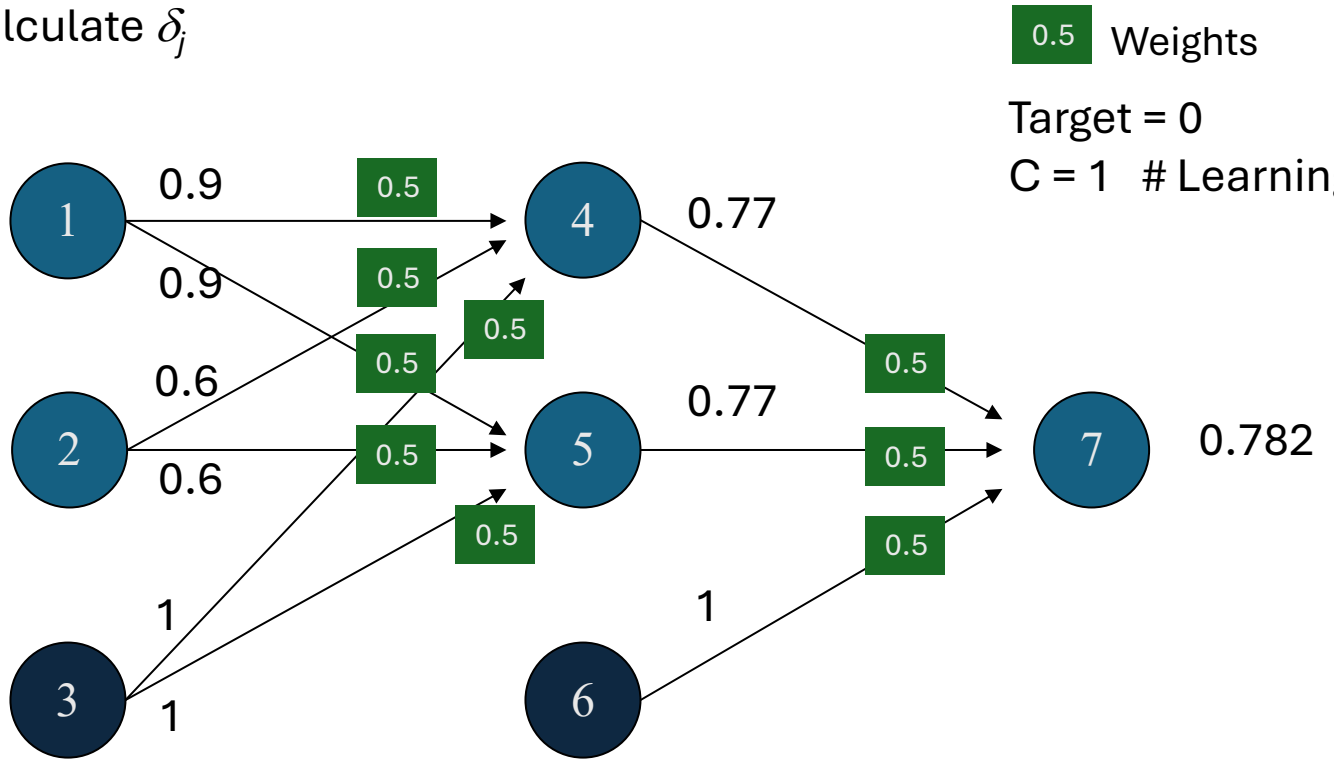
$net_q = \sum x_i w_{ij}$

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

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

Backward Pass Example
(Where the Magic Happens)

Calculate δ_j



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

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

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

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

Output node – only one

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

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

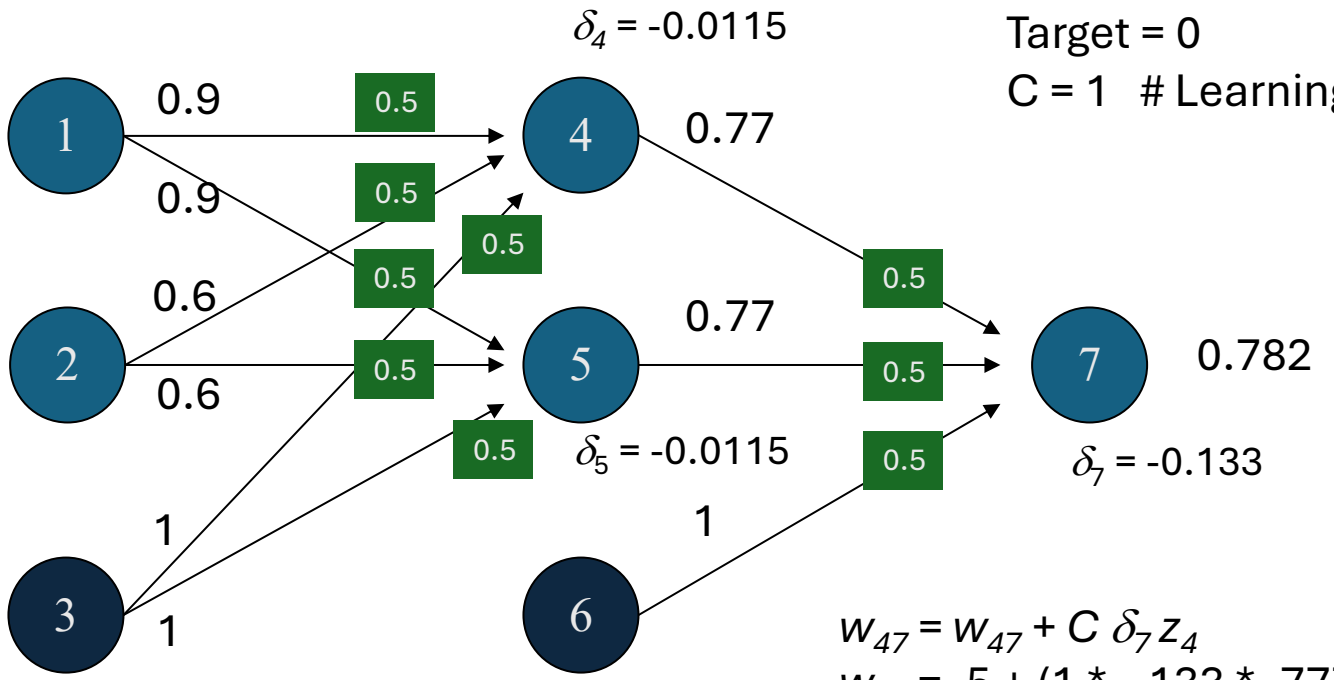
Hidden nodes – only have node 7 following

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

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

$\delta_5 = -.0115$

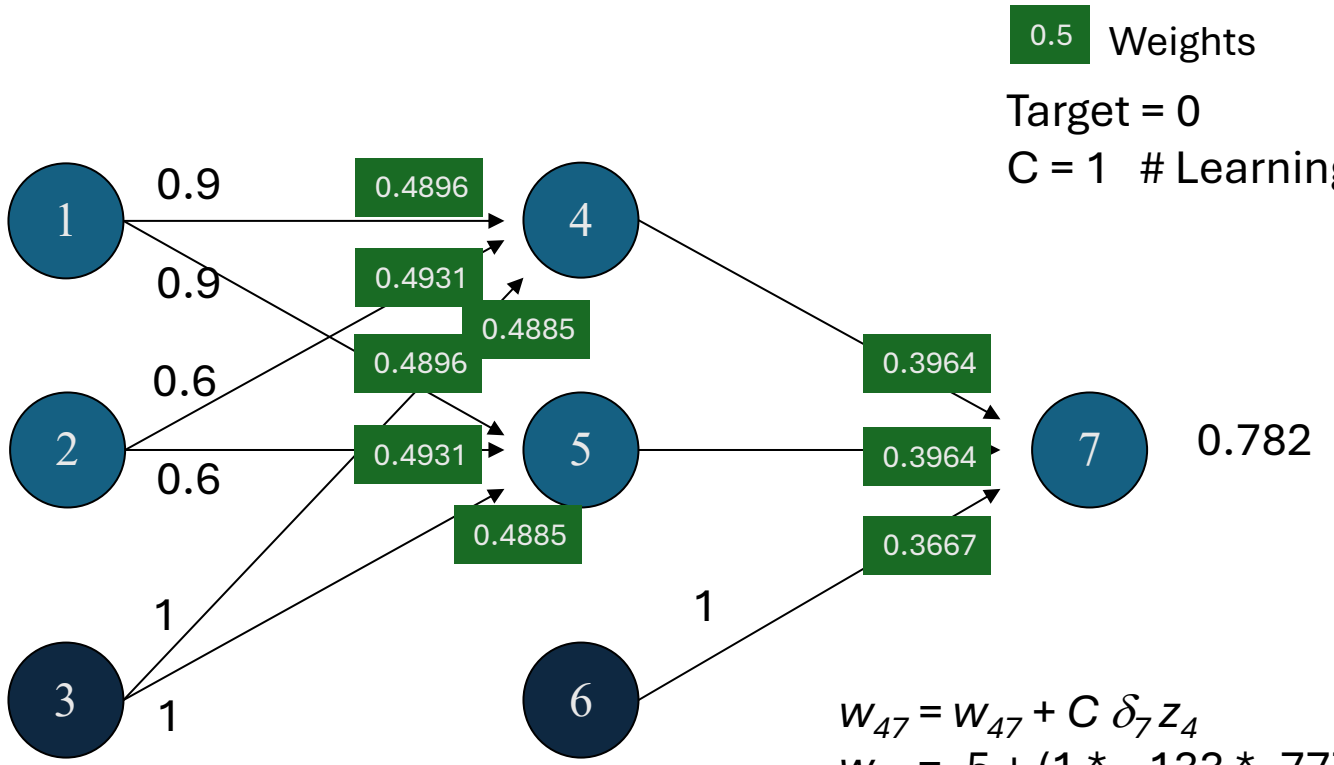
Calculate new weights



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

$w_{47} = w_{47} + C\ \delta_7 z_4$
$w_{47} = .5 + (1 * -.133 * .777) = .3964$
$w_{57} = .3964$
$w_{67} = .5 + (1 * -.133 * 1) = .3667$

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



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

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

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

$w_{57} = .3964$

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

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

$w_{15} = .4896$

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

$w_{25} = .4931$

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

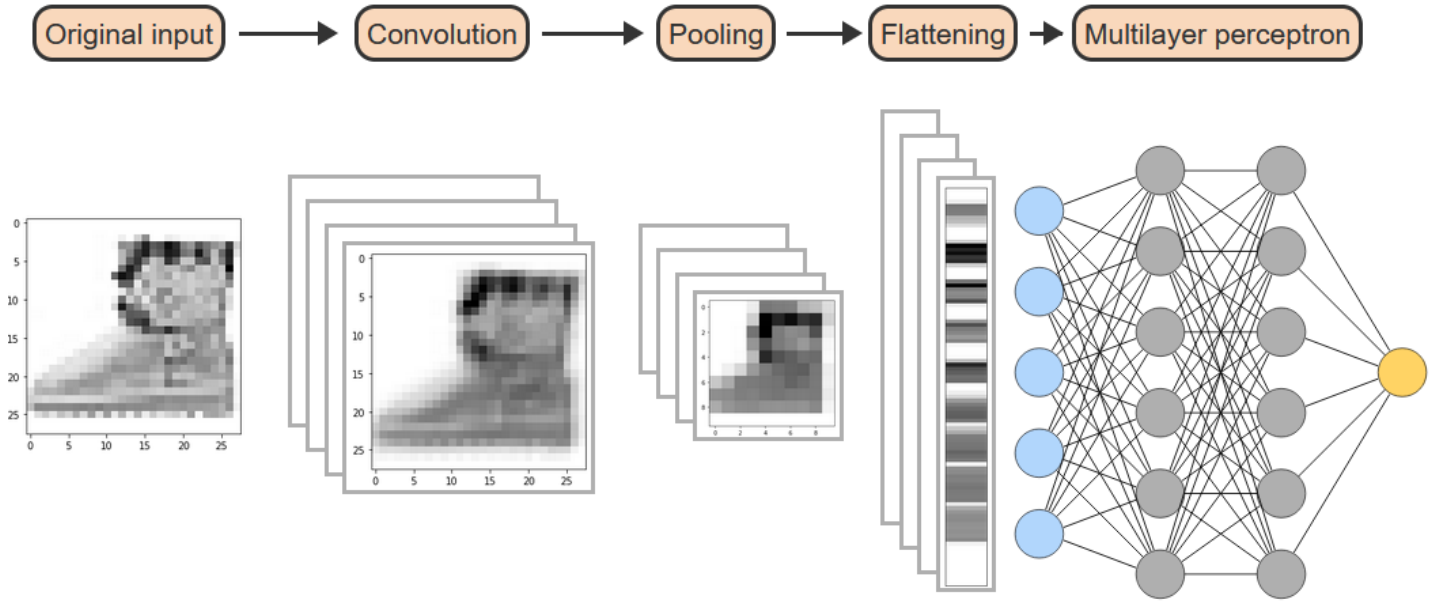
$w_{35} = .4885$

Quiz Time!

Beyond Fully Connected MLPs (Didn’t get to this last class)

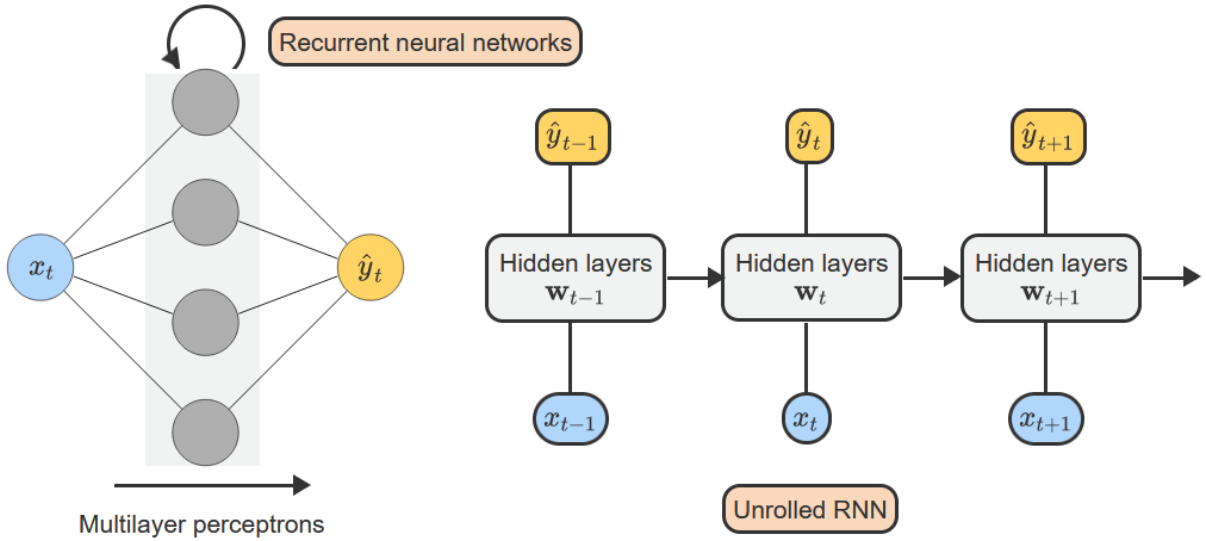
Neural Networks Beyond Fully Connected Layers - CNN

- Transforms and projections can help neural network learn structure.
- Convolutional Neural Network
- Used for image classification



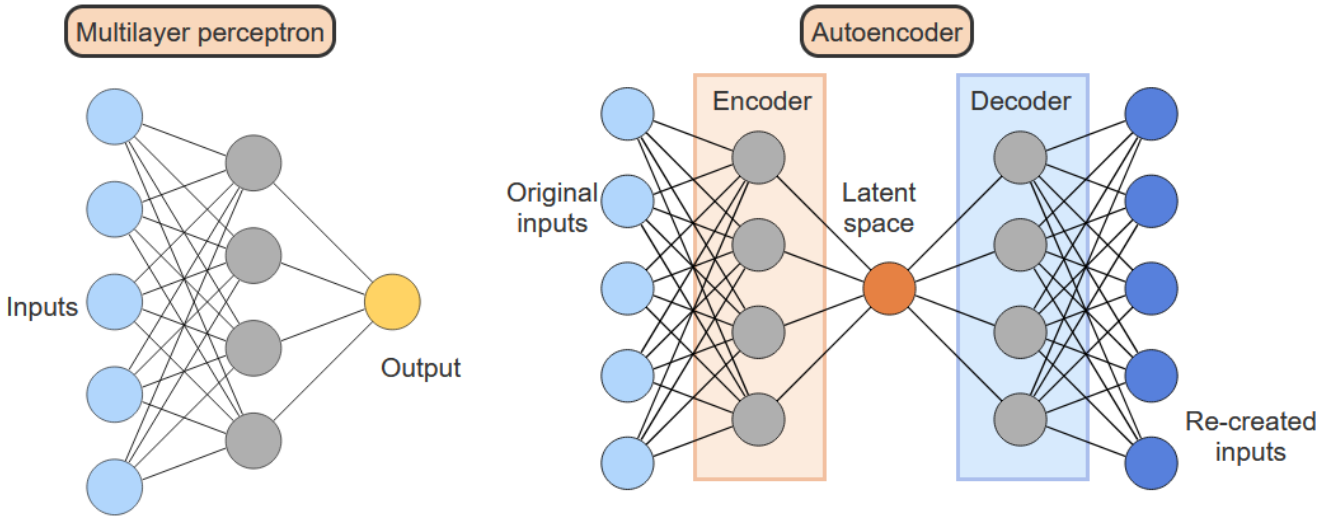
Neural Networks Beyond Fully Connected Layers - RNN

- Pass in last output in addition to current input
- Great for sequential data
- This is how we used to do text processing
- Recurrent Neural Network



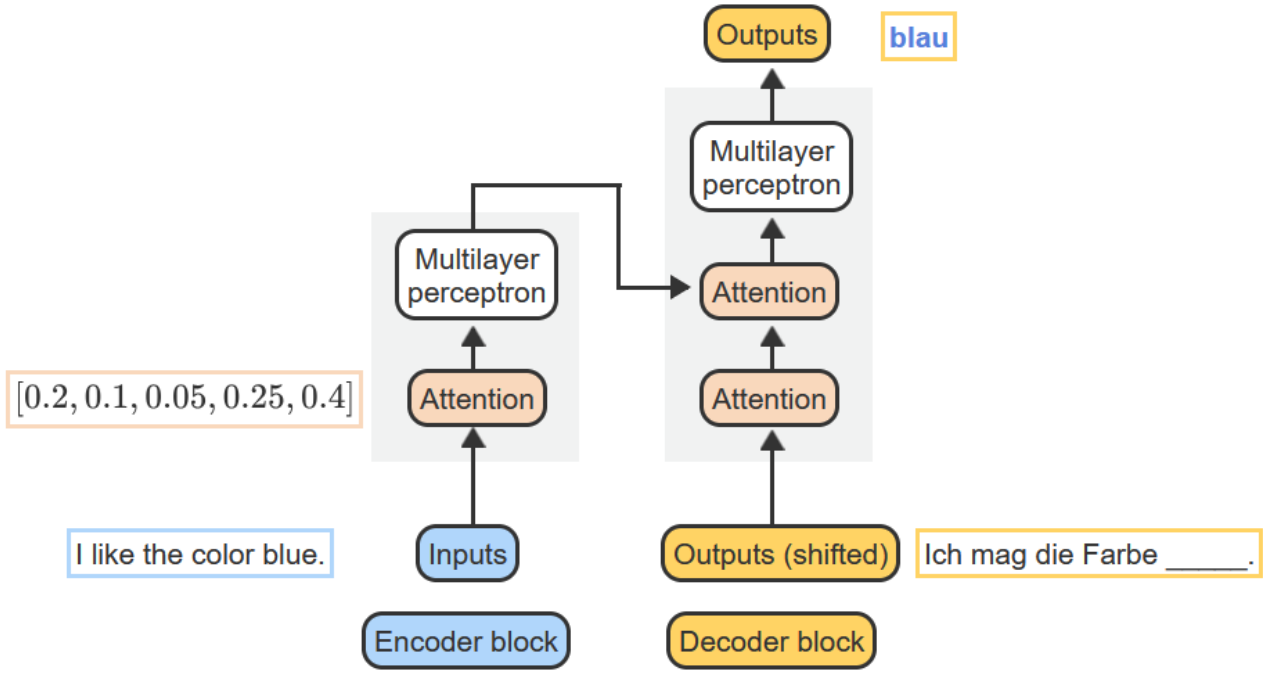
Neural Networks Beyond Fully Connected Layers - Autoencoders

- Use MLP to learn a latent space
- Then another MLP to reconstruct from the latents
- Form of compression



Neural Networks Beyond Fully Connected Layers - Transformers

- House MLPs between attention layers
- Attention learns relationship between inputs
- Modern LLMs are transformers



Neural Networks Beyond Fully Connected Layers - GANs

- Has multiple networks
- One to generate things, one to guess whether generated or real
- Play them against each other
- Tricky to train
- Out of fashion now
- How we used to do image generation