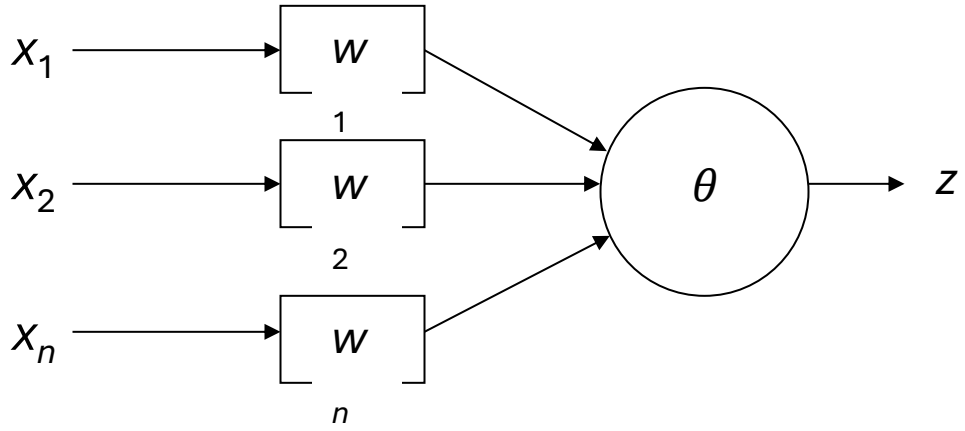
Neural Nets 1

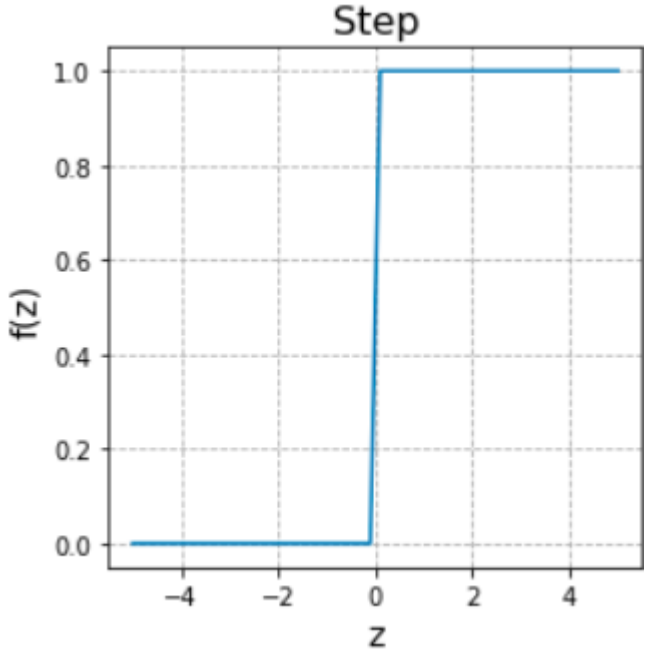
5 March 2026

Alex Lyman

Perceptron 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

Perceptron

- Perceptron will converge to a boundary on any linearly separable function.
- What if not linearly separable?

Upgrading the Perceptron

Perceptron Problems

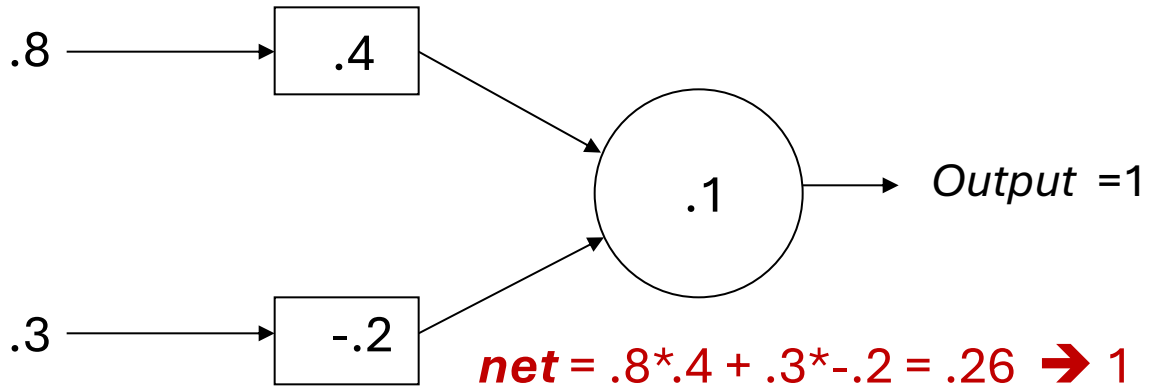
- Problem:
  - Perceptron implements discrete model of error (i.e., identifies the *existence* of error and adapts to it)
- Solution:
  - Allow nodes to have real-valued activations

    (amount of error = target output – computed output)
  - Design learning rule that adjusts weights based on error

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

    Perceptron Rule $\Delta w_i = c(t - output)x_i$

First Training Instance



x_1	x_2	target
.8	.3	1
.4	.1	0

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

Delta Rule

- Delta rule uses (target - net) **before** the net value goes through the threshold in the learning rule to decide weight update

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

- Weights are updated **even when the output would be correct**

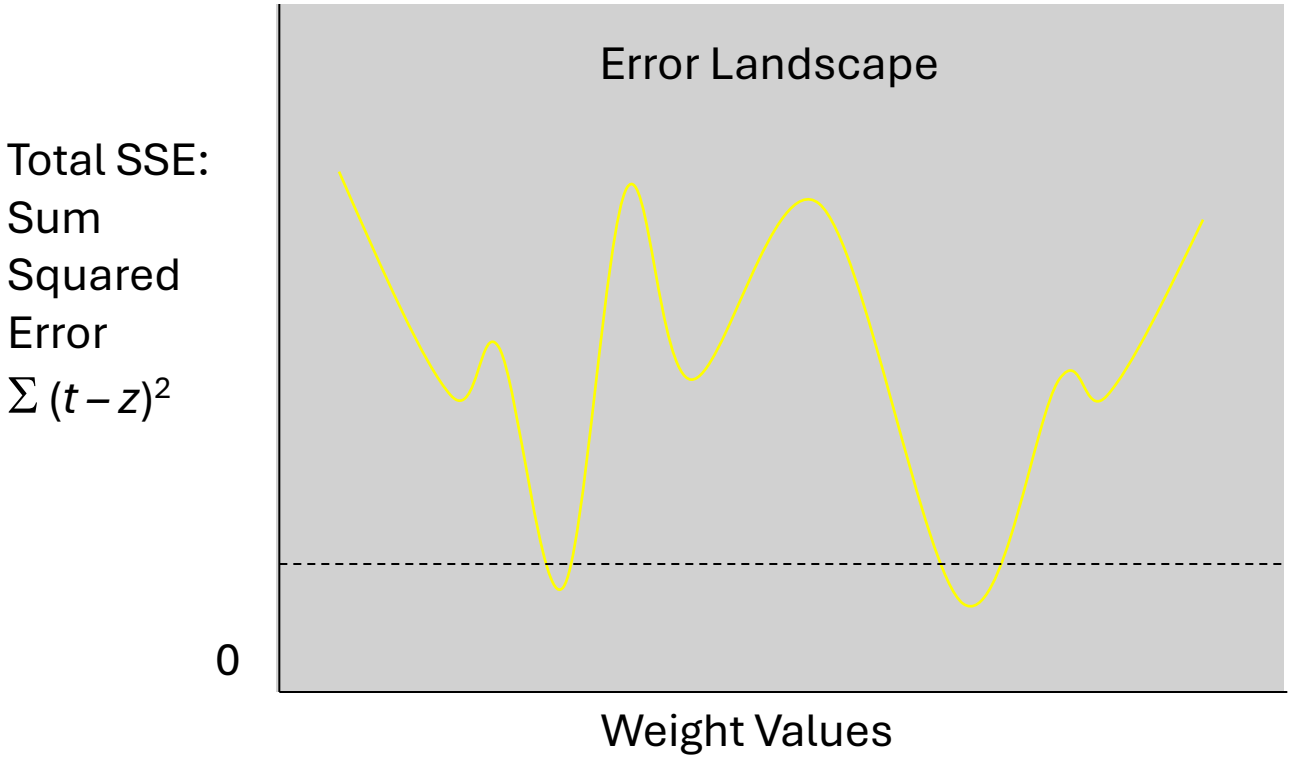
Perceptron rule vs Delta rule

- Perceptron rule (target - thresholded output) guaranteed to converge to a separating hyperplane if the problem is linearly separable. Otherwise may not converge – could get in a cycle
- Single layer Delta rule guaranteed to have only one global minimum. Thus, it will converge to the best SSE solution whether the problem is linearly separable or not.
  - Could have a higher misclassification rate than with the perceptron rule and a less intuitive decision surface – we will discuss this later with regression where Delta rules is more appropriate
- Stopping Criteria – For these models we stop when no longer making progress
  - When you have gone a few epochs with no significant improvement/change between epochs (including oscillations)

Error

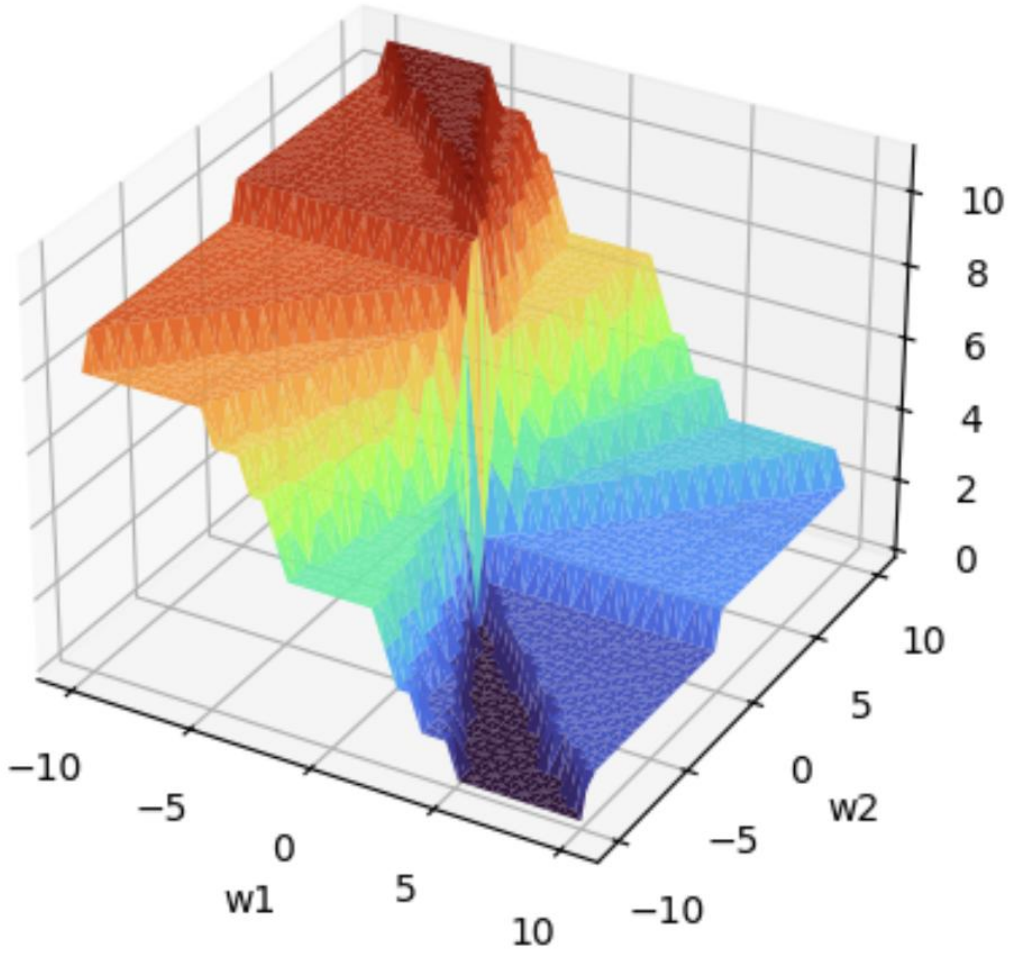
Error Surface

- Error is a function of the weights
  - Using SSE
  - $\mathrm{E} = \Sigma(t_i - z_i)^2 = \Sigma(t_i - \Sigma x_j w_{ij})^2$
- *If we could search this space*, we could find the minimum error!



Error Surface

- Single perceptron w/perceptron rule.
- If we could search this space (all possible combinations of values for all weights), we could find the spot that minimizes error.
- That would be awesome.
- Too computationally expensive
- We minimize error (non-exhaustively search error space) using **gradient descent.**



Gradient Descent

Gradient Descent

- Mapping the whole error surface is too computationally expensive.
- It is probably doable for a small perceptron, but what about a deep neural net with thousands, millions, billions of weights?
- We can’t do that.
- Gradient descent lets us search the space without mapping every possible combination of weights.

Gradient Descent Intuition

- Imagine you are on a snowy mountain at night.
- The lodge is at the bottom of the mountain, and you need to get there.
- But you don’t have a flashlight and you can’t see.
- How do you know which way to go?
- Take a step in one direction. If you are going up, turn around.
- If you are going down, keep going down.
- You can feel with your feet to know where to go.
- Repeat the process until you get to the bottom of the mountain (lodge).

Gradient Descent Intuition

- We can do that same thing on an error surface. (Error as a function of the values of our model's weights)
- We can search, one step at a time, to try and find the bottom (area with lowest error)
- Then, we know which weights would minimize error.

Gradient Descent Learning: Minimize (Maximize) the Objective Function

- Gradient descent algorithm
  - Find a starting location – set of weights
  - Loop
    - Calculate output values
    - Use the *gradient* to adjust your weight values
- Adjusting the weights
  - **Derivative of the error function** w.r.t the weights – slope or gradient
  - Derivative is slope or rate of change at a given point.

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

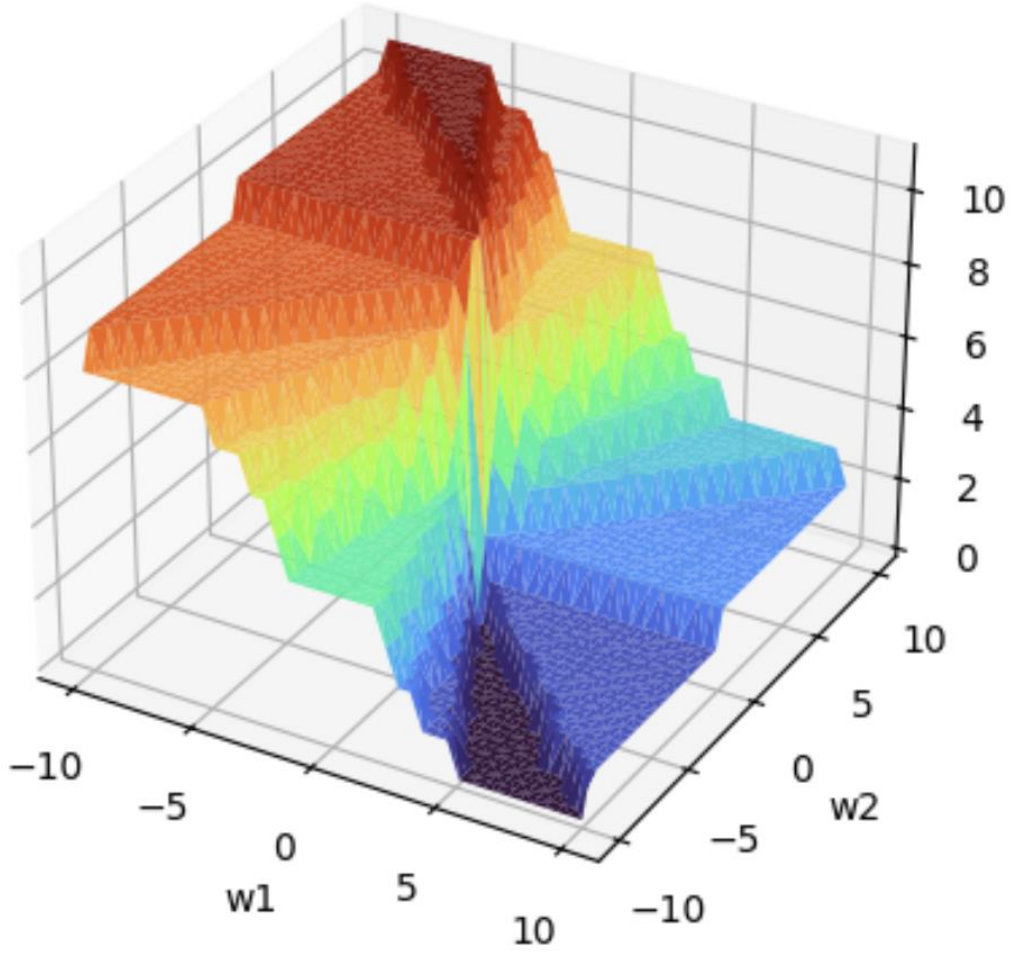
Deriving a Gradient Descent Learning Algorithm

- Goal is to decrease overall error (or other loss function) each time a weight is changed
- Sum Squared error one possible loss function $E = \Sigma (t - z)^2$
  - Actually use $E = \frac{1}{2}\Sigma (t - z)^2$ to make the derivative nice and clean
- Other reasons to use SSE
  - All errors are positive
  - Amplifies the effect of larger errors
  - Transforms the error surface – smooth and differentiable

- Partial derivative of the error function w.r.t the weights gives us a weight update function

Gradient Descent – Error Surface

- Is the classic perceptron error surface differentiable?
- That means we can't use gradient descent.
- Think of snowy mountain at night example – most of your steps are flat! (no information)
- How can we fix this?
- By using a differentiable activation function.

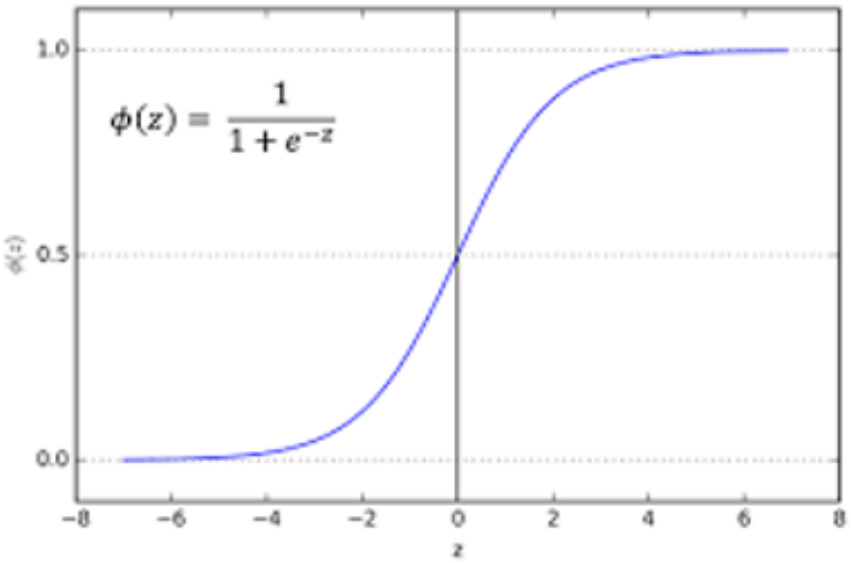


Non-Linear Output

- Introduce non-linearity with sigmoid function:

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

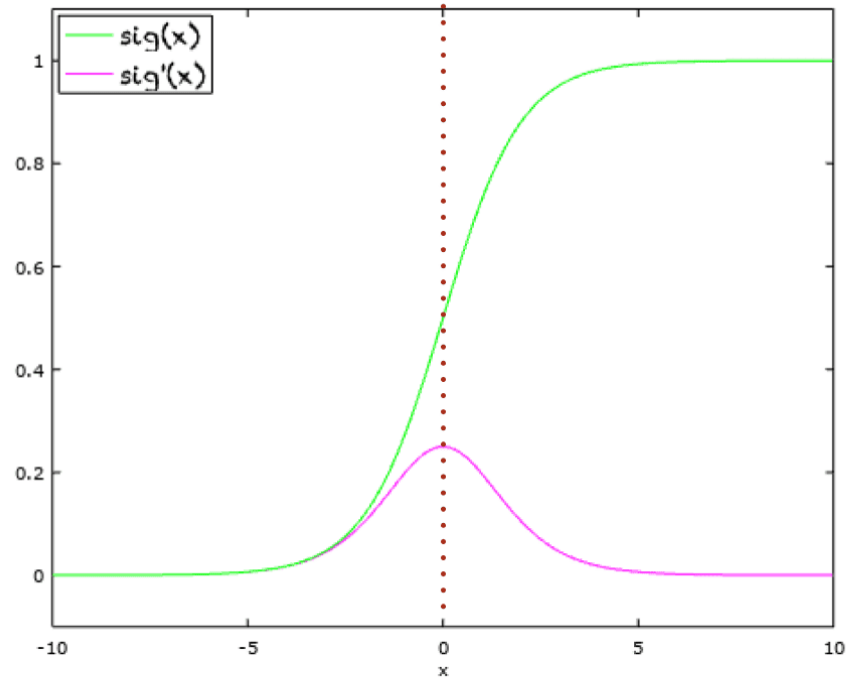
$$o = \frac{1}{1 + e^{-net}}$$



- Smooth and differentiable

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

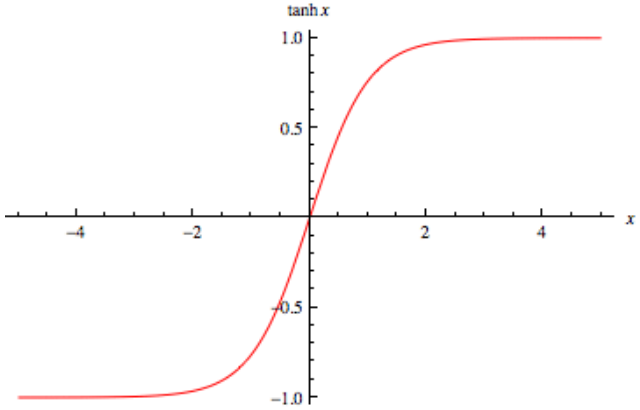
$$\sigma(x) = 1 - \sigma(-x)$$

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

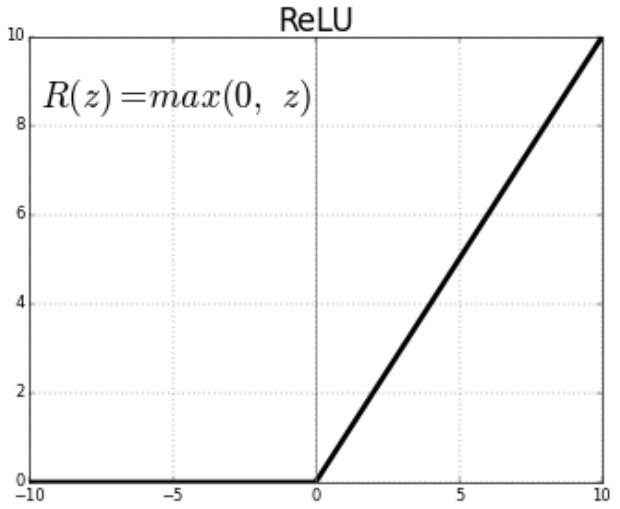
$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Other Activation Functions

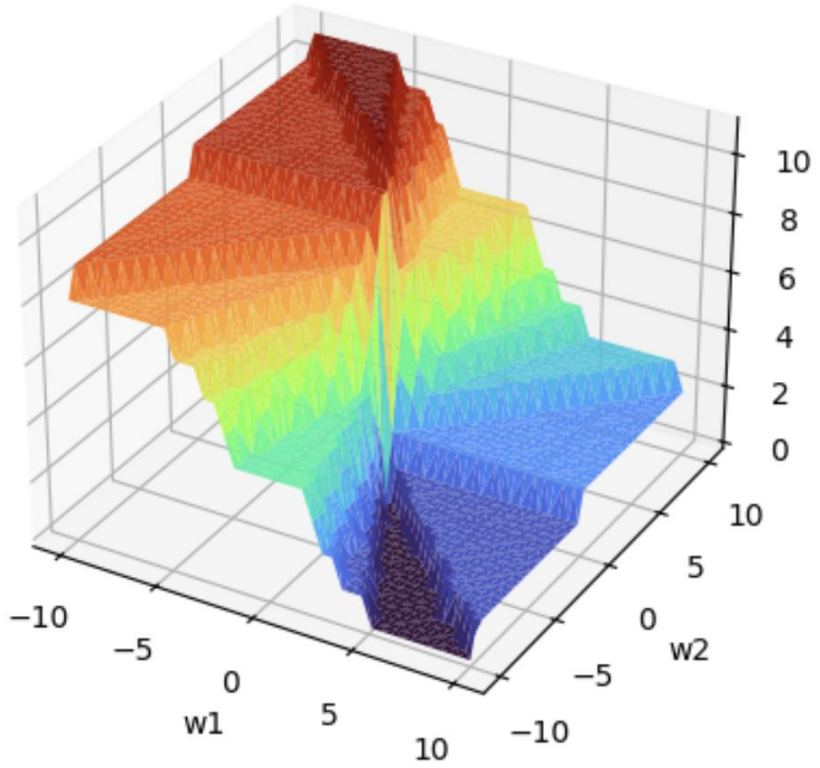
- Hyperbolic tangent (tanh)
  - Same shape as sigmoid but tanh(0)=0 (rather than 0.5)



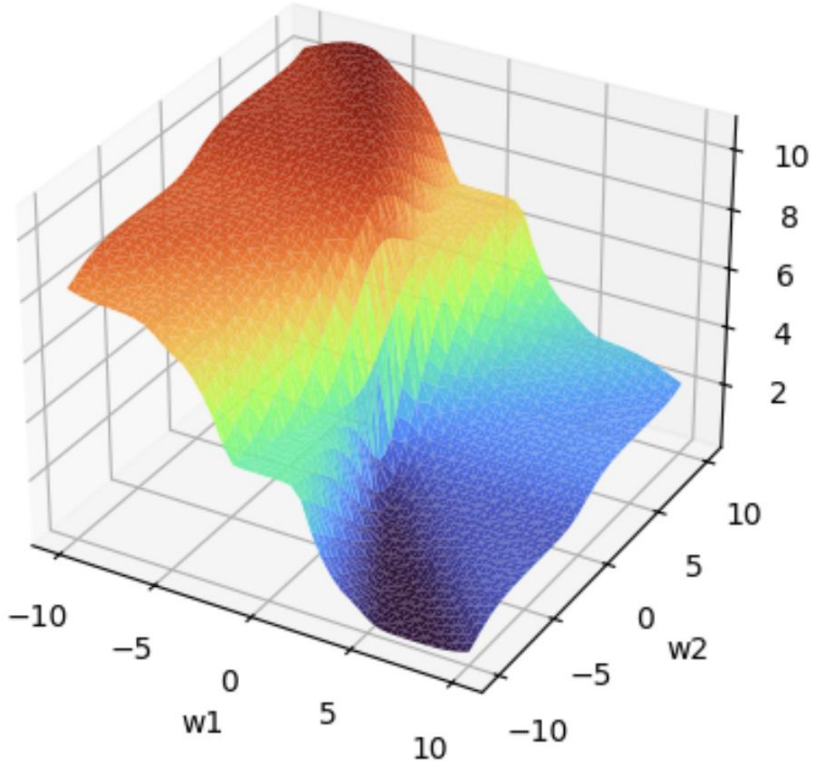
- Rectified linear activation function (ReLU)
  - Popular in deep nets



Perceptron Rule + Step function



Delta Rule + Sigmoid(net)



Changing to the Delta Rule and using Sigmoid(net) for output changes the decision surface to smooth and differentiable.
Which of these would be easier to feel around in on a dark night?

Gradient Descent Intuition – Learning Rate

- Imagine you are back on that snowy mountain, trying to get to the bottom of the canyon.
- In this example, you also have extremely strong legs.
- How big of a step should you take?
  - If you take too small of a step, you'll have to take millions of steps to get to the bottom, that would take too long.
  - If you take too big of a step, you might leap over the lodge altogether to the other side of the canyon.
- This is what the learning rate hyperparameter controls.

Derivation (for enrichment purposes only)

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D} 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \frac{1}{2}\sum_{d\in D} 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}.\vec{x})$$

Chain rule

$$\frac{\partial E}{\partial w_i} = \sum_{d\in D}(t_d - o_d)(-x_{id})$$

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)x_{id}$$

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:

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

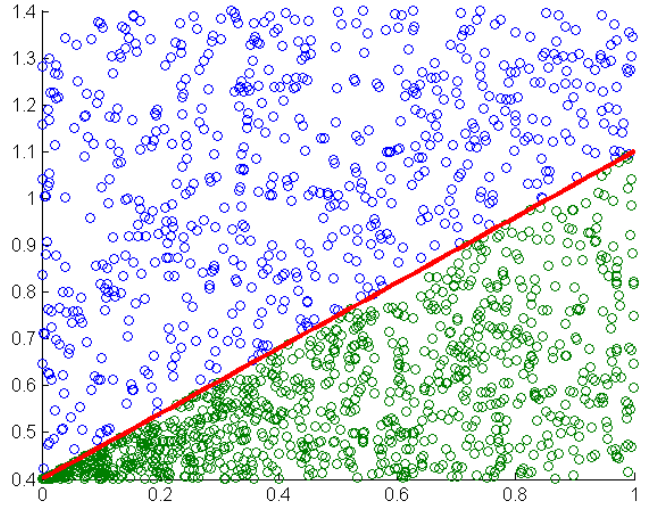
Recap (II)

- 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
    - Something like Sigmoid, ReLU, or TANH (or softmax)

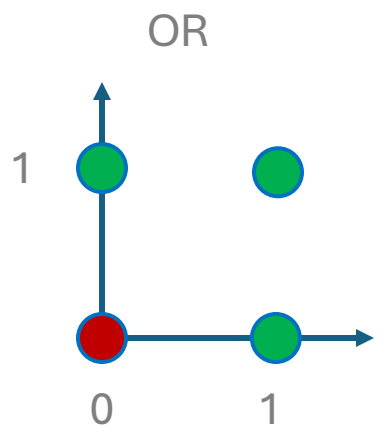
From Perceptrons to Neural Nets

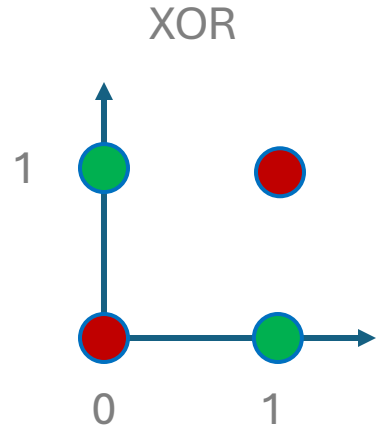
History: OR / XOR

- 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

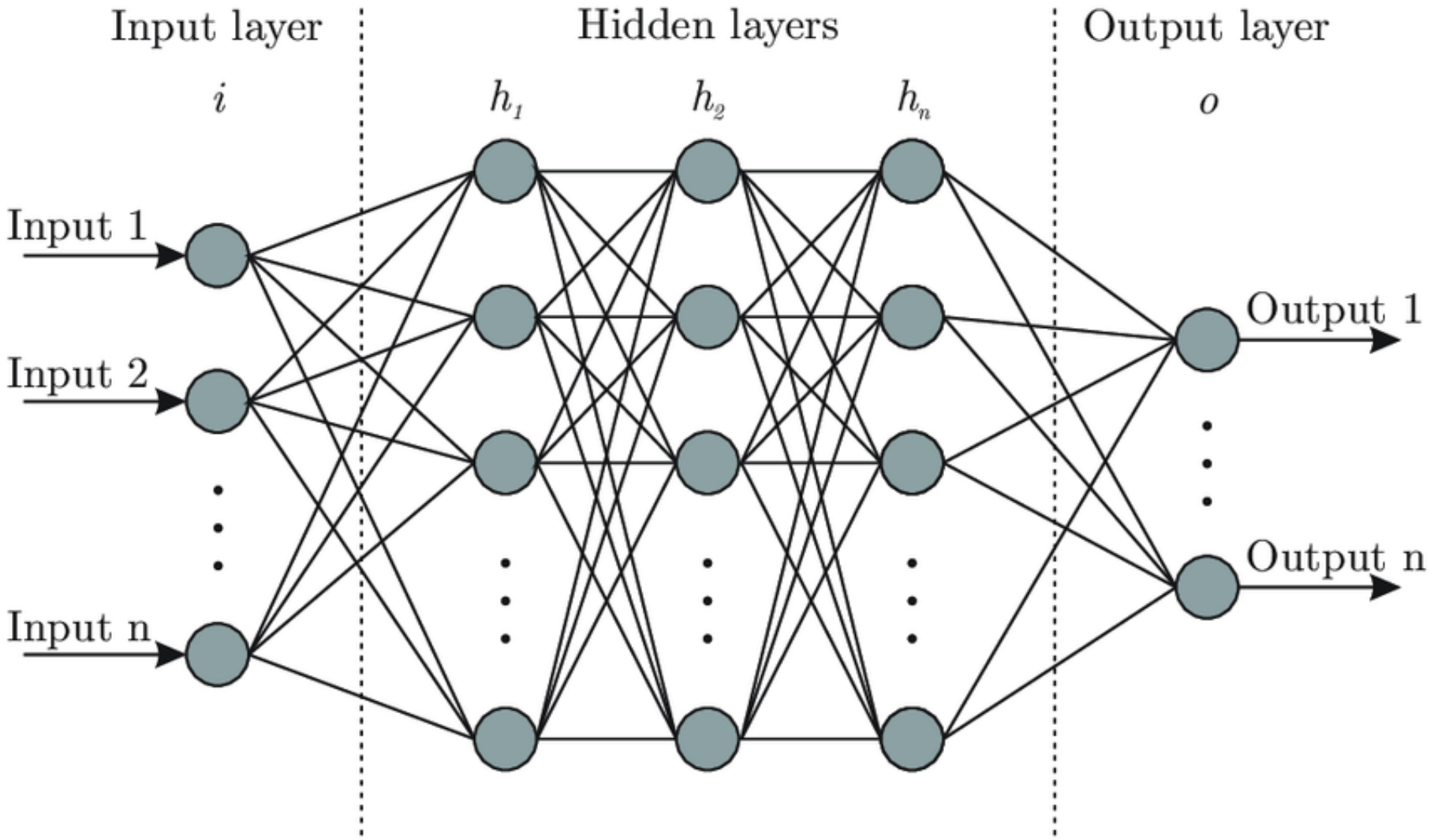




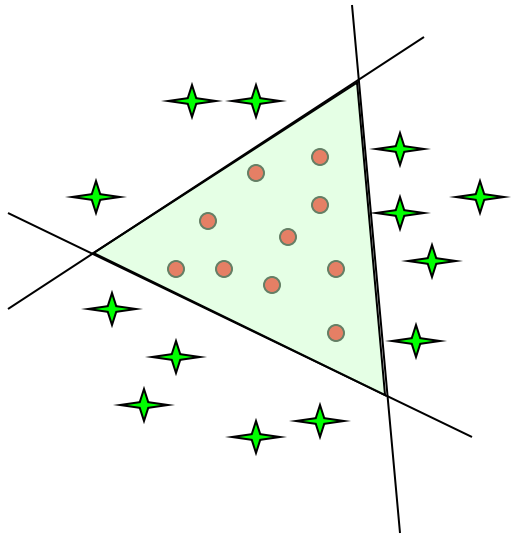
Perceptrons to Neural Nets

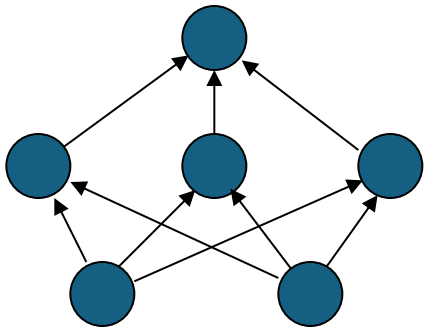
- What if we stacked a bunch of perceptrons together in a net?
- We could wire the output of each neuron to the input of the next layer of neurons.
- Then we could approximate more complex functions.
- If we introduced nonlinearity (with nonlinear activation functions), then we could approximate any weird nonlinear function.
- MLP (Multi-Layer Perceptron was born)

Multi-layer Perceptron (MLP) Topology



Multi-Layer Generalization





Universal Function Approximation Theorem

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

Theory and Practice…

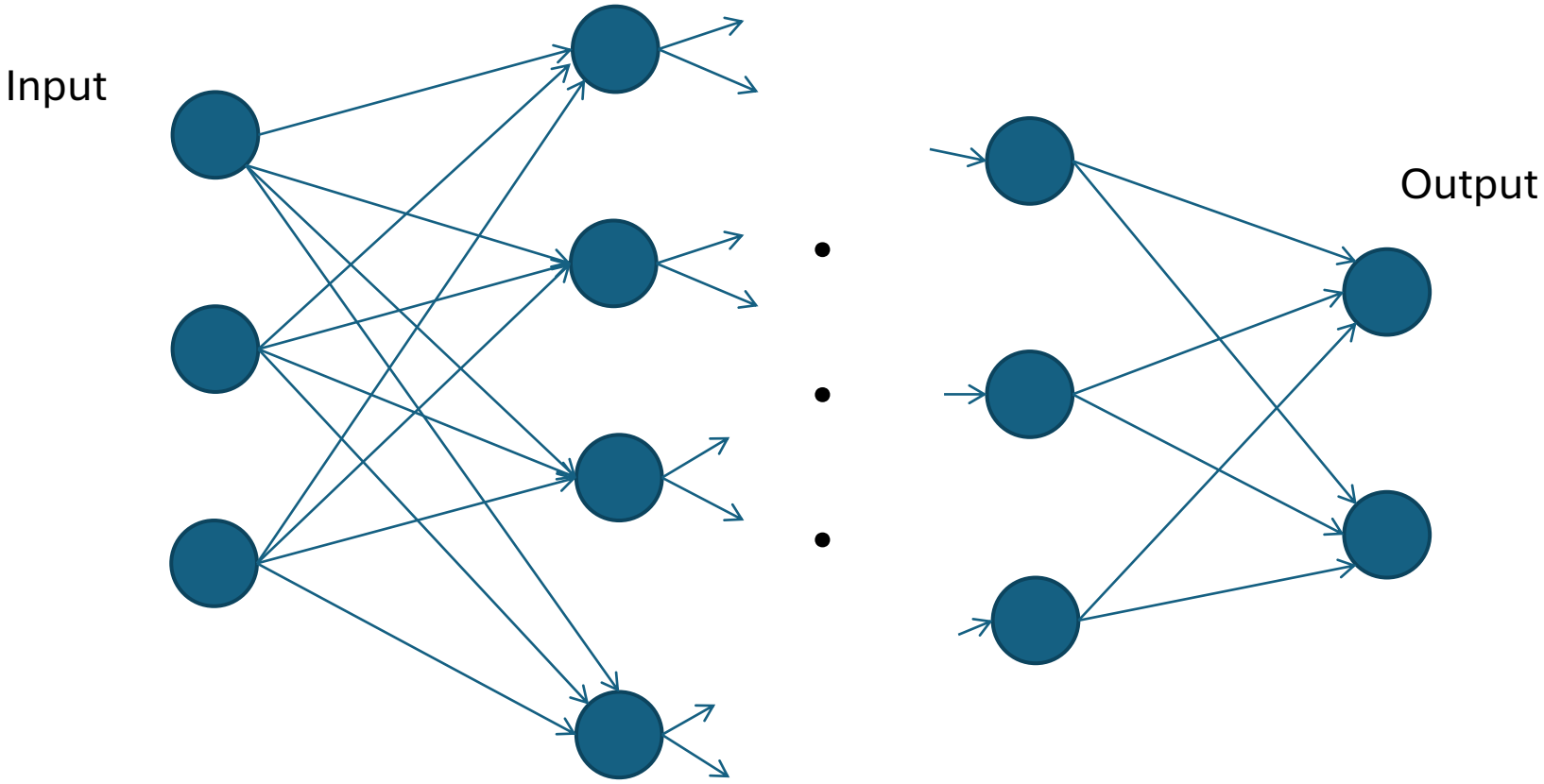
- In theory then, we don’t need more than 1 hidden layer
  - However, this is an existence proof.

- In practice, it is difficult to learn the correct weights for a task!
  - Lots of work has gone into figuring out the best way to make a neural net.

Backpropagation

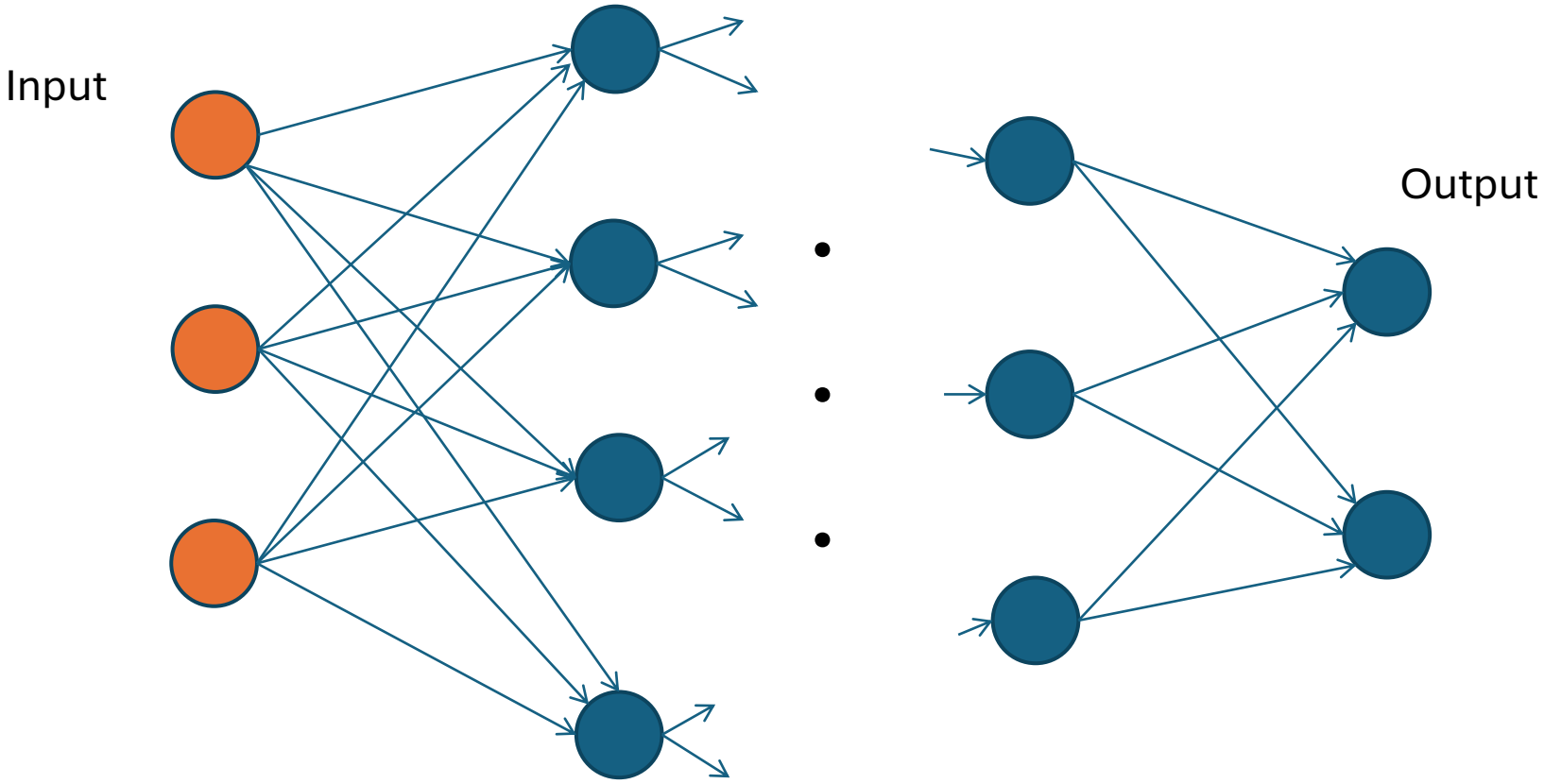
Backpropagation

- Operates similarly to perceptron learning



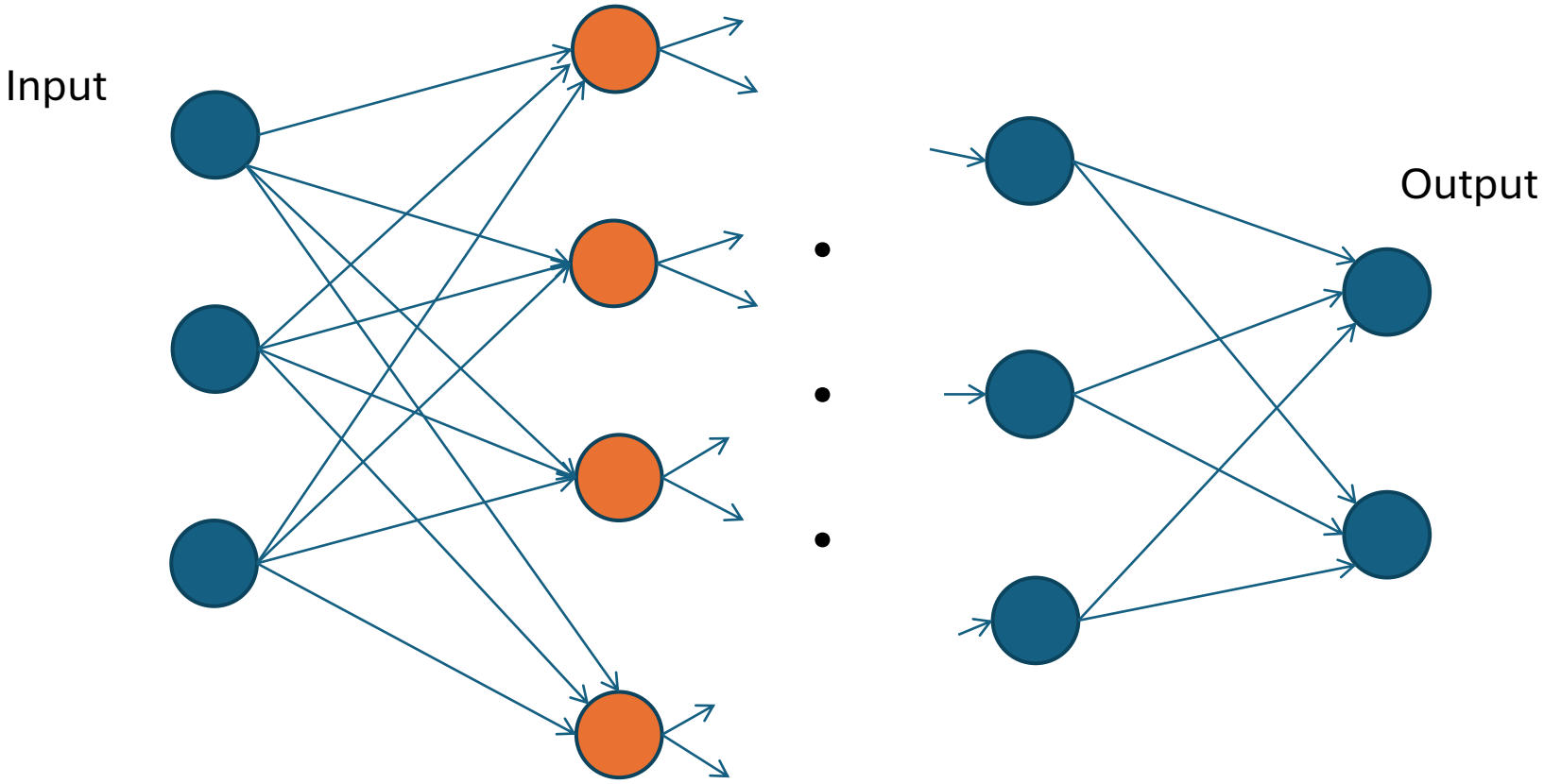
Backpropagation

- Inputs are fed forward through the network



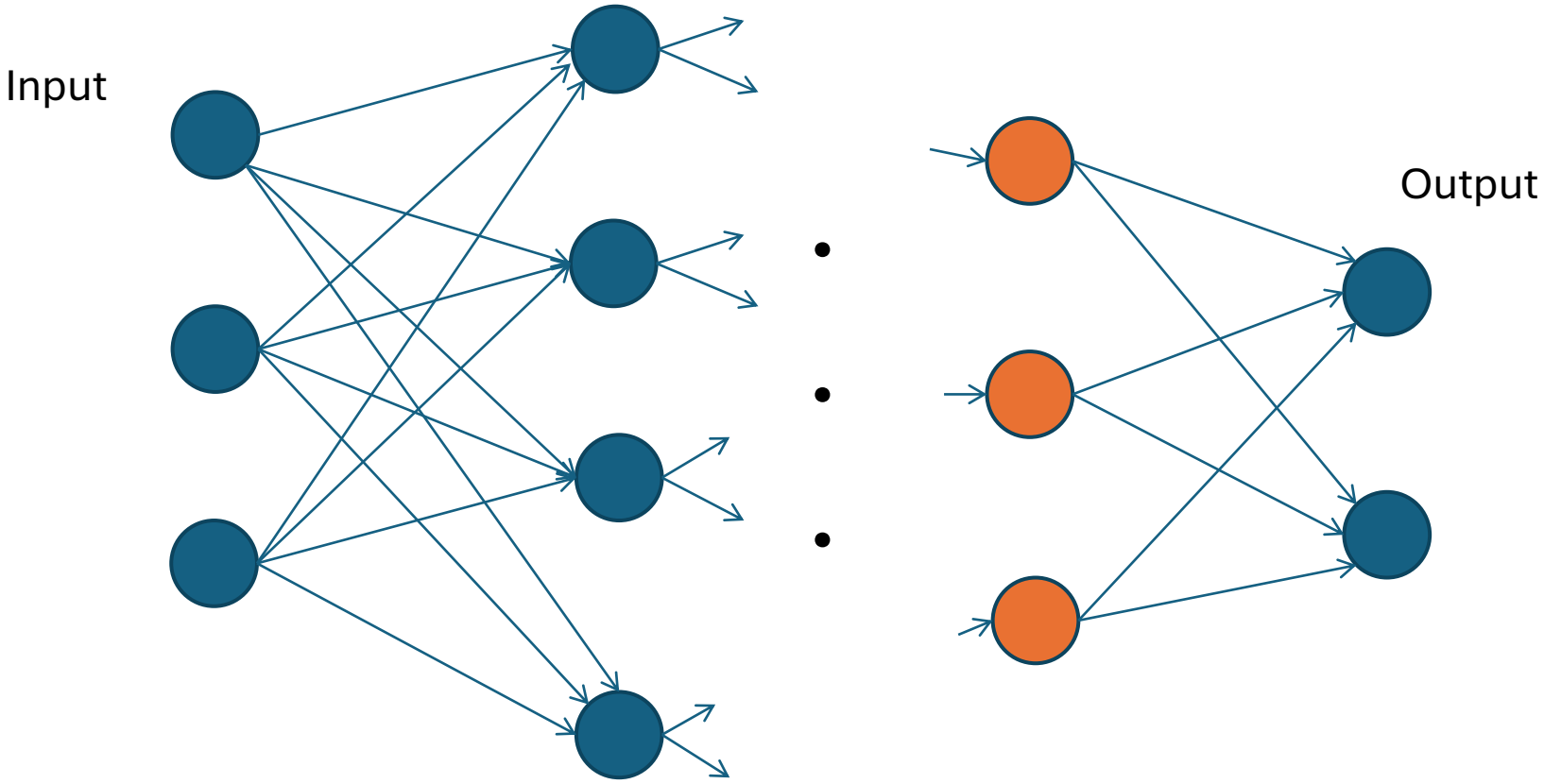
Backpropagation

- Inputs are fed forward through the network



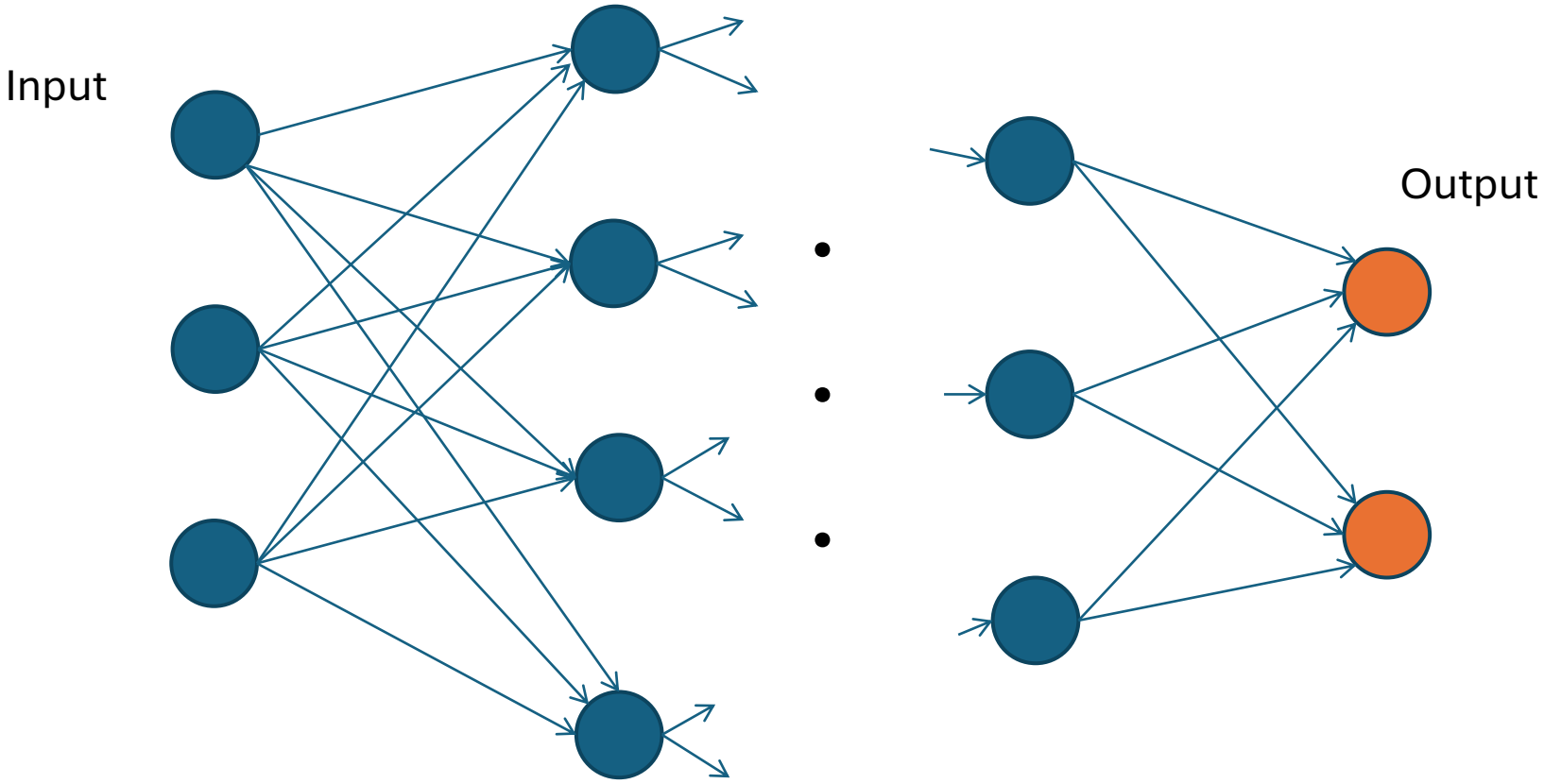
Backpropagation

- Inputs are fed forward through the network



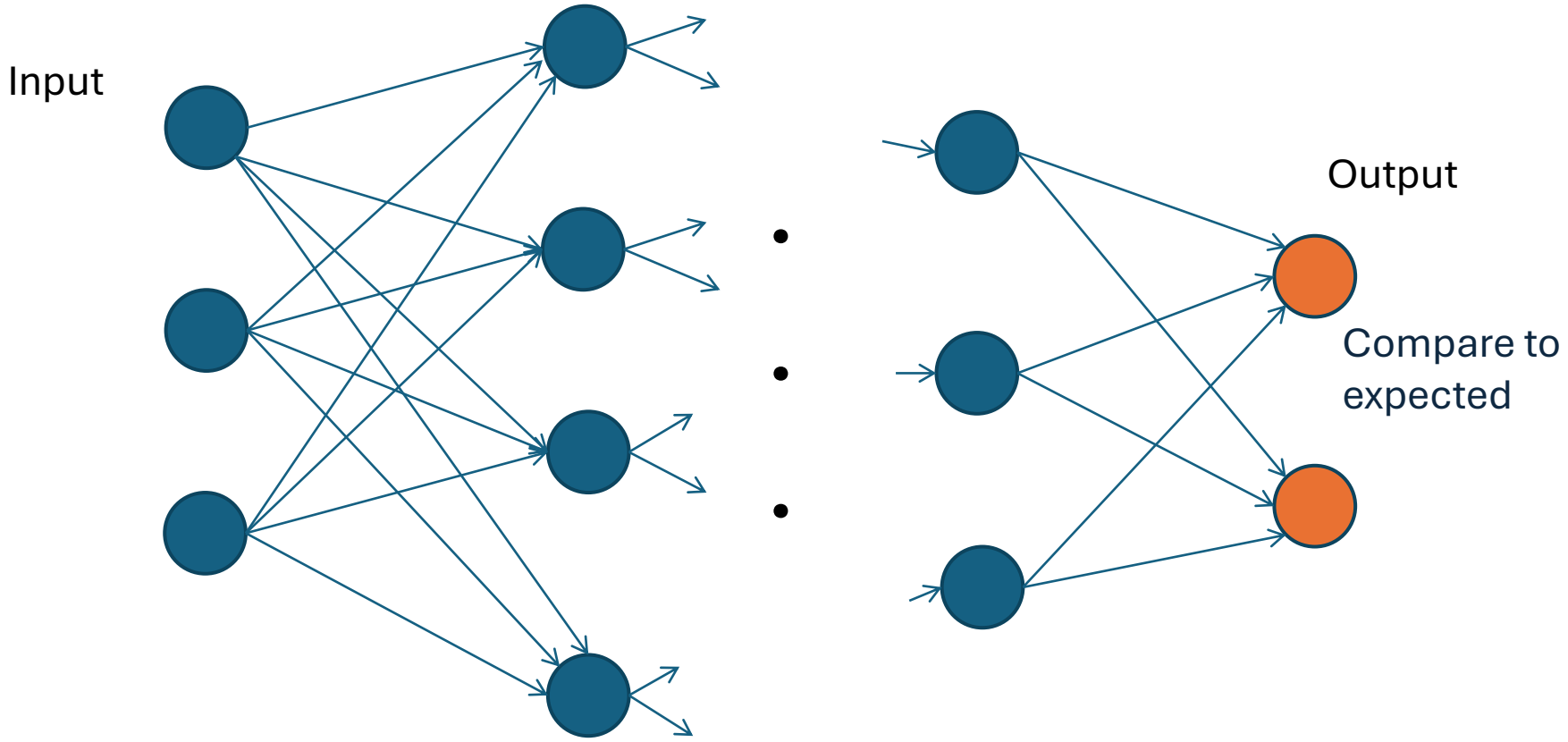
Backpropagation

- Inputs are fed forward through the network



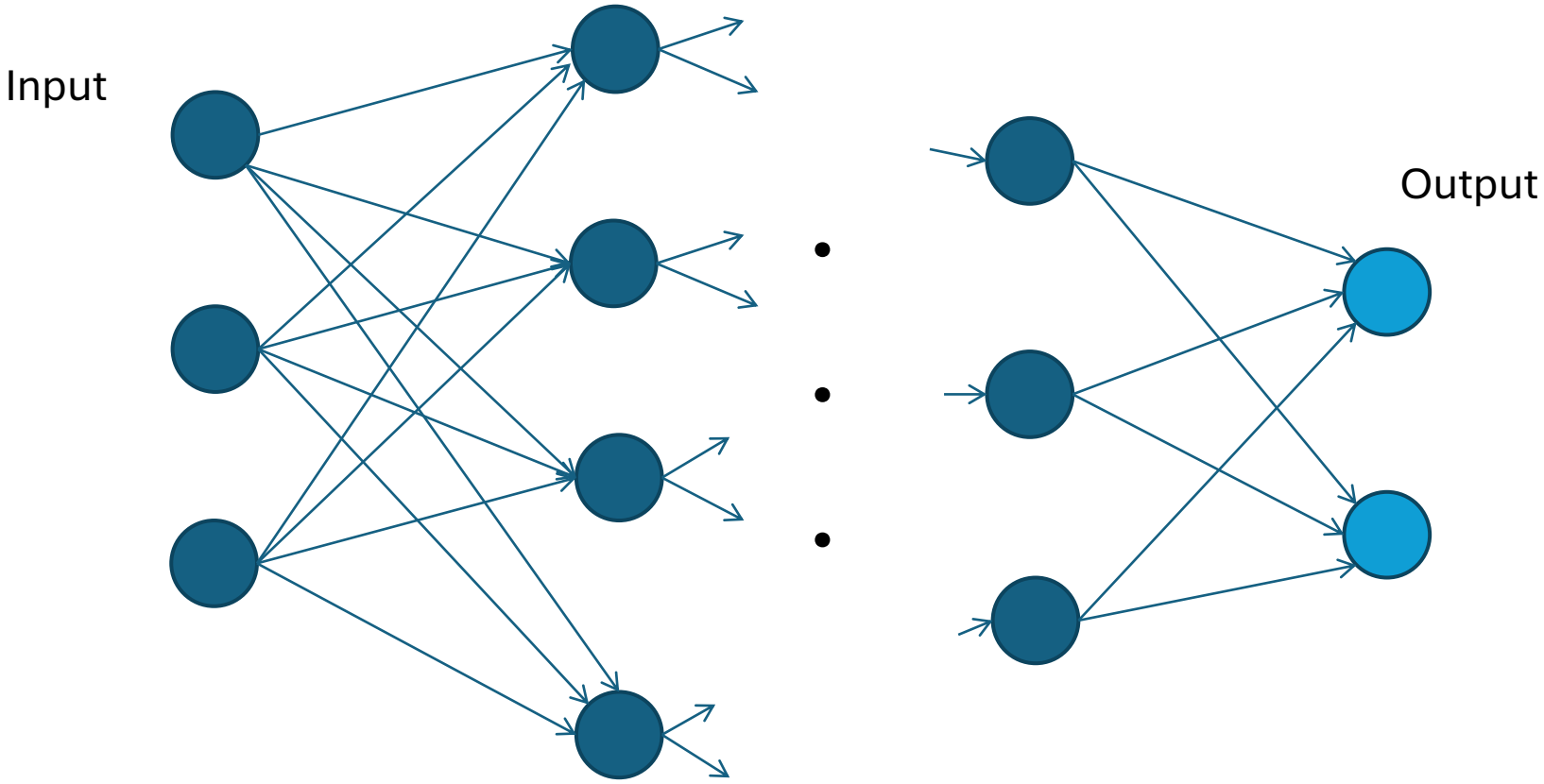
Backpropagation

- Inputs are fed forward through the network



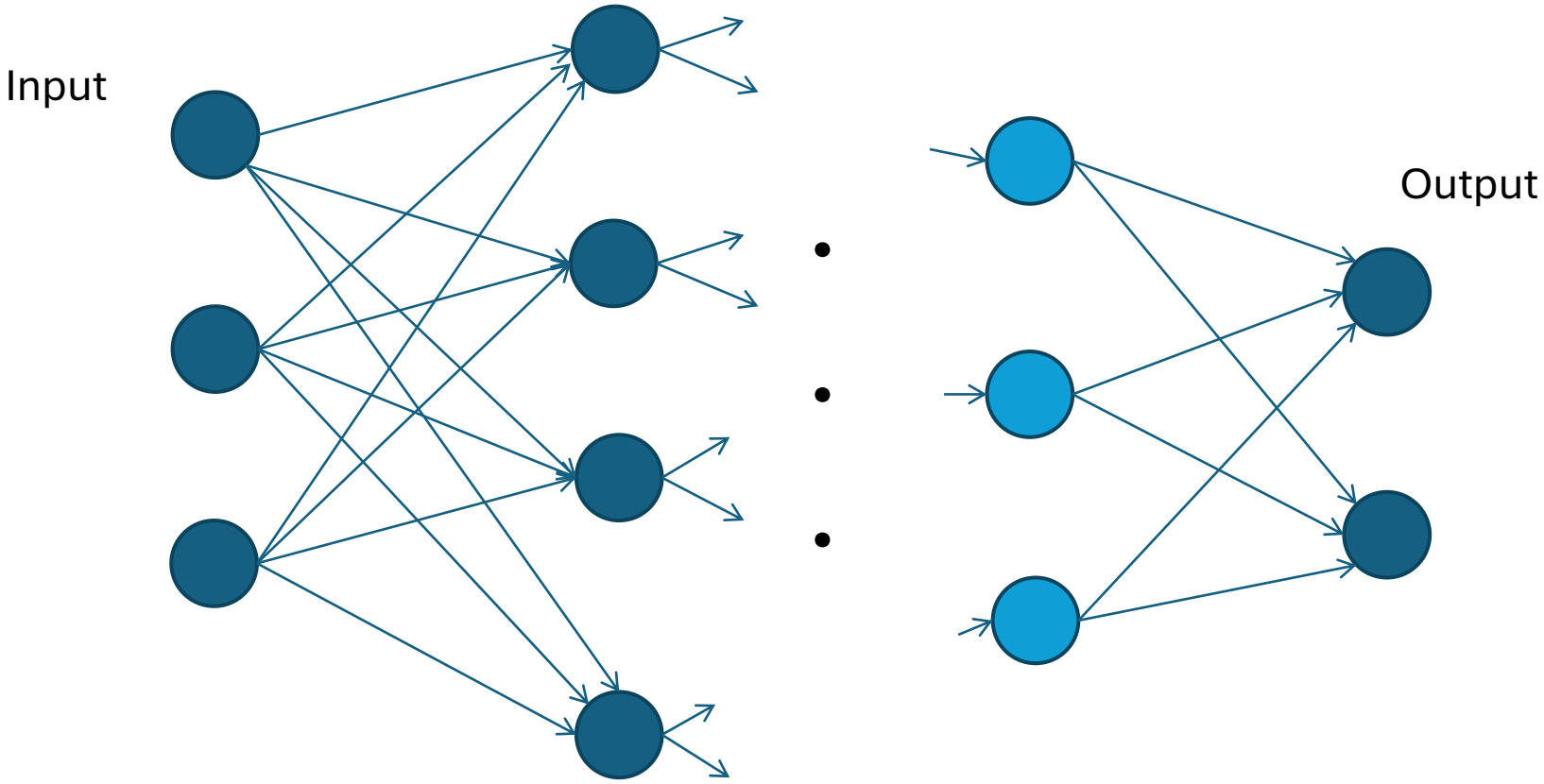
Backpropagation

- Errors are propagated back



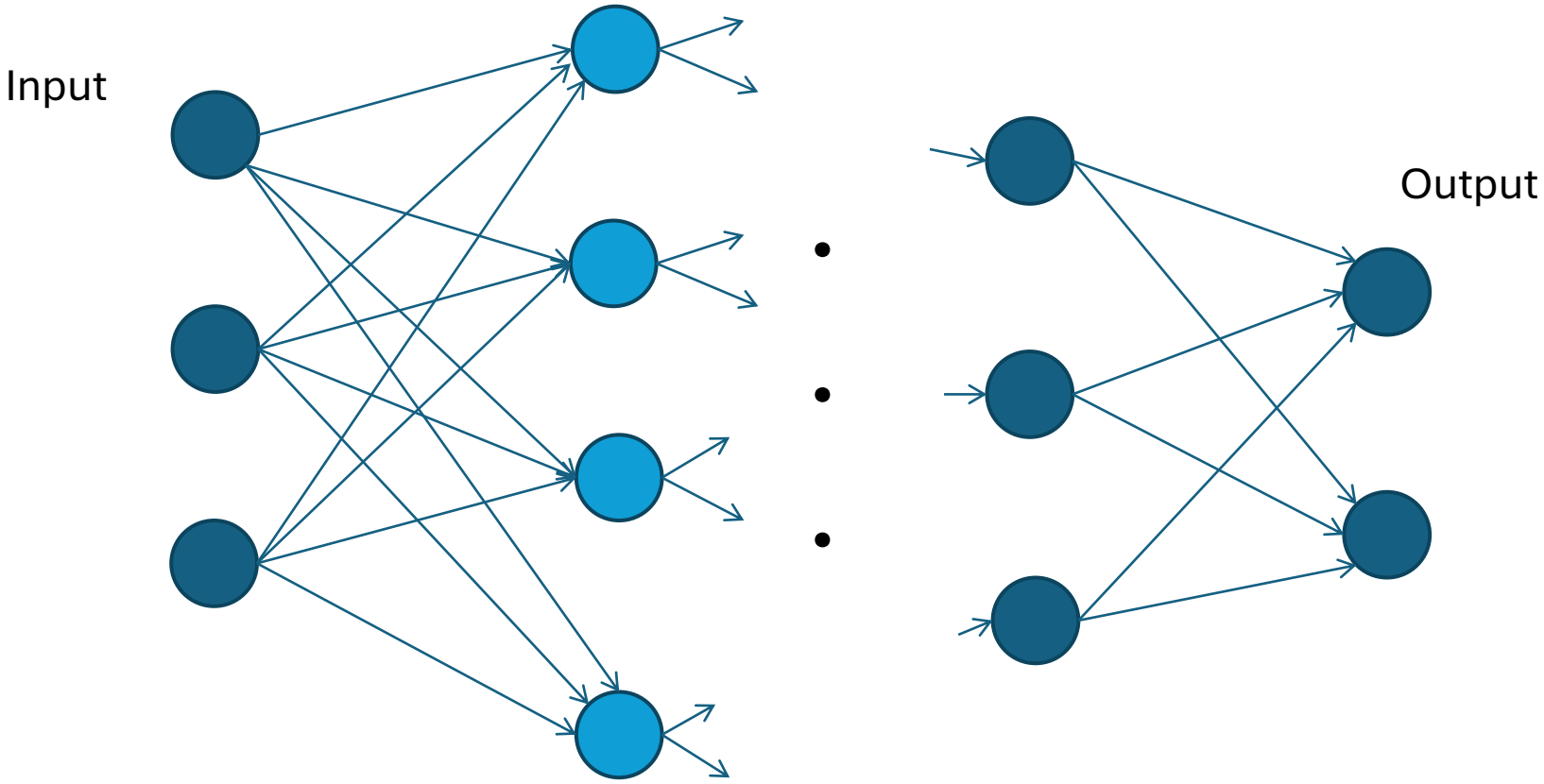
Backpropagation

- Errors are propagated back



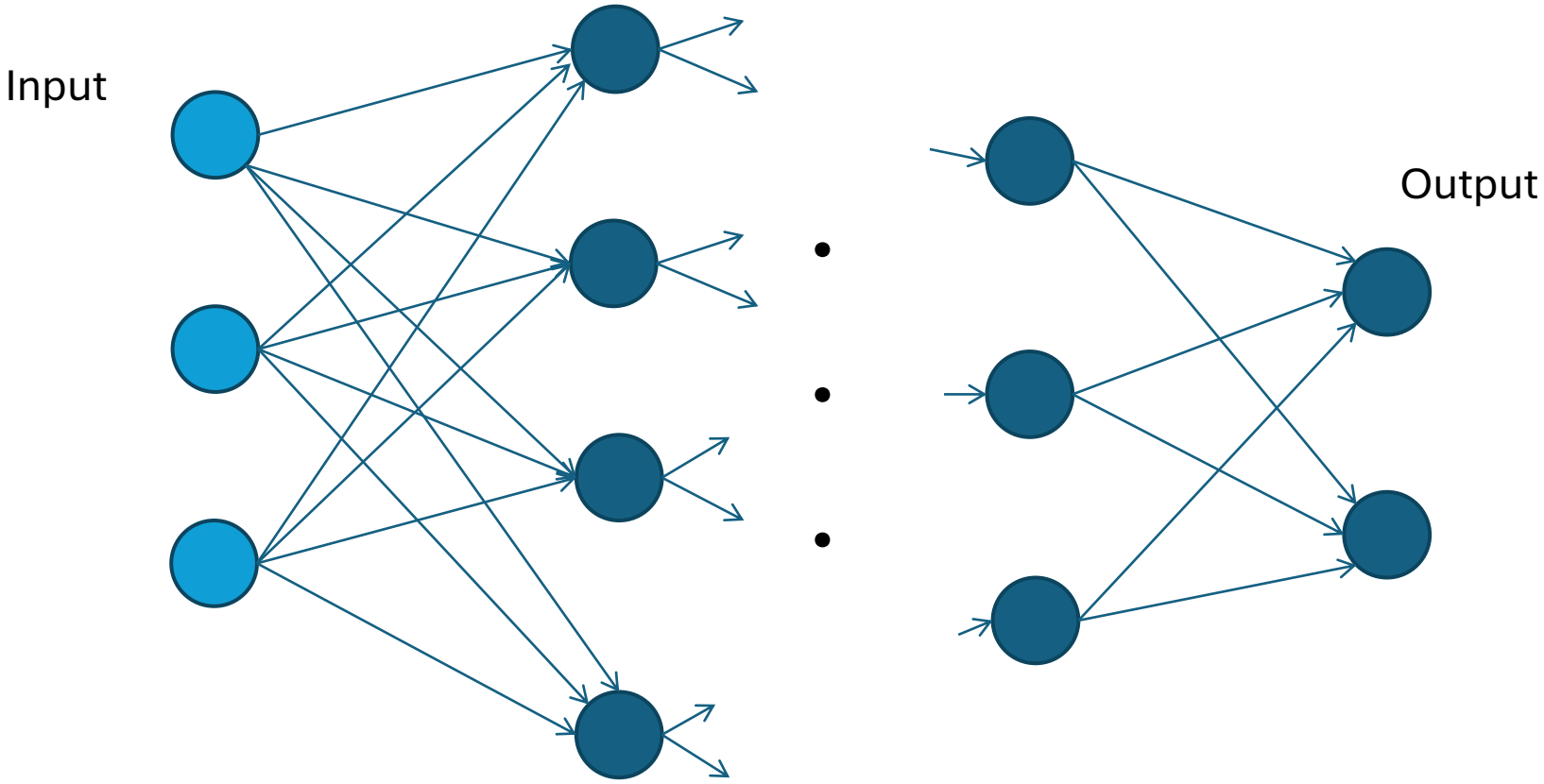
Backpropagation

- Errors are propagated back



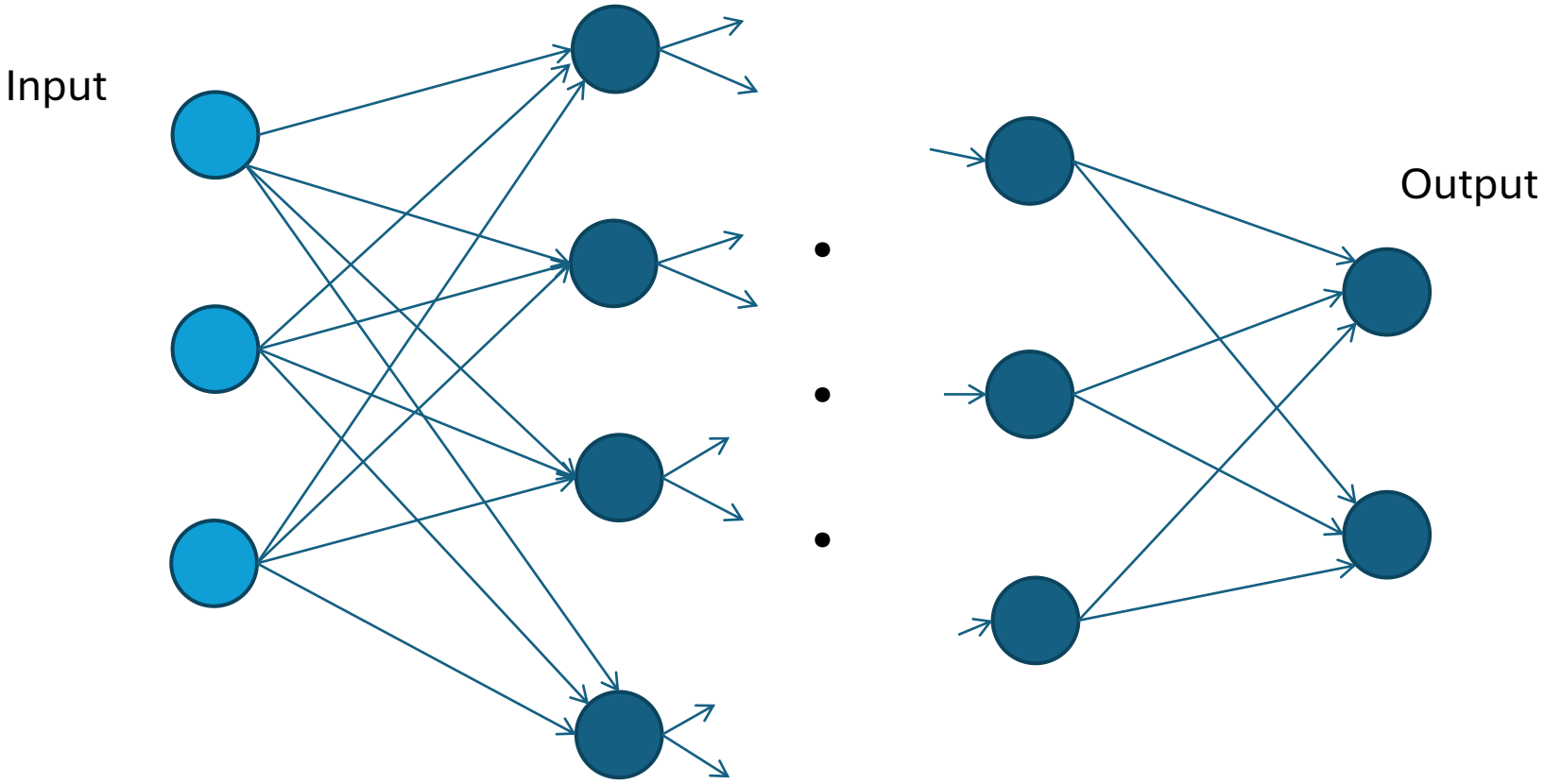
Backpropagation

- Errors are propagated back



Backpropagation

- Adjust weights based on errors



Backpropagation History (Enrichment)

- 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!

Backpropagation 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

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

Quiz Time!

Resources

- 3 Blue, 1 Brown Neual Nets, Gradient Descent, Backprop
  - https://www.youtube.com/watch?v=aircAruvnKk
  - https://www.youtube.com/watch?v=IHZwWFHWa-w
  - https://www.youtube.com/watch?v=Ilg3gGewQ5U

Code Time!