COMP47460

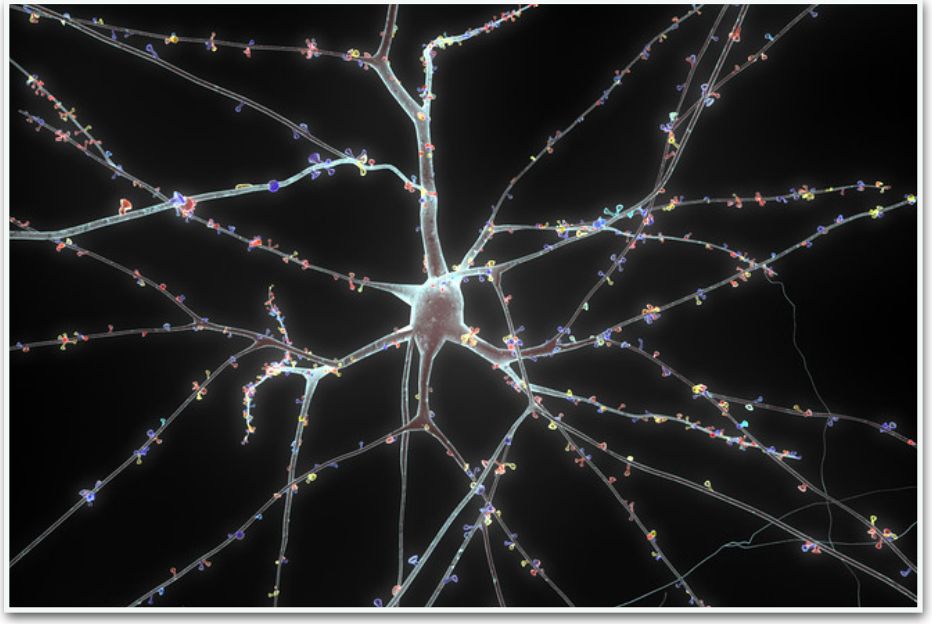
Neural Networks

Aonghus Lawlor
Derek Greene

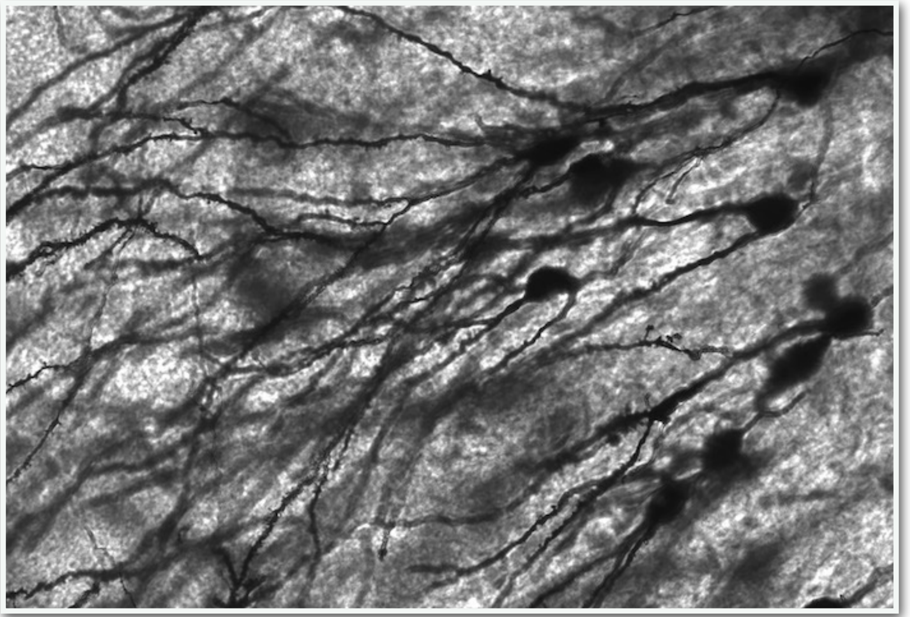
School of Computer Science
Autumn 2018

Motivation: Biological Inspiration

- Artificial Neural Networks (ANNs) are inspired by biological nervous systems, such as the brain, where large numbers of interconnected neurons work together to solve a problem.
- A neuron receives signals from other neurons through connections, called synapses.
- The combination of these signals, in excess of a certain activation level, will result in the neuron "firing" - i.e sending a signal on to other neurons connected to it.



wsj.com



wikipedia.org

Motivation: Decision Making

Q. "Will a customer wait for a restaurant table?" - "Yes" or "No"

- This decision might depend on a number of factors:

  x_1 : Is the restaurant full? (0 or 1)

  x_2 : Is the customer hungry? (0 or 1)

  x_3 : Is a suitable alternative restaurant nearby? (0 or 1)

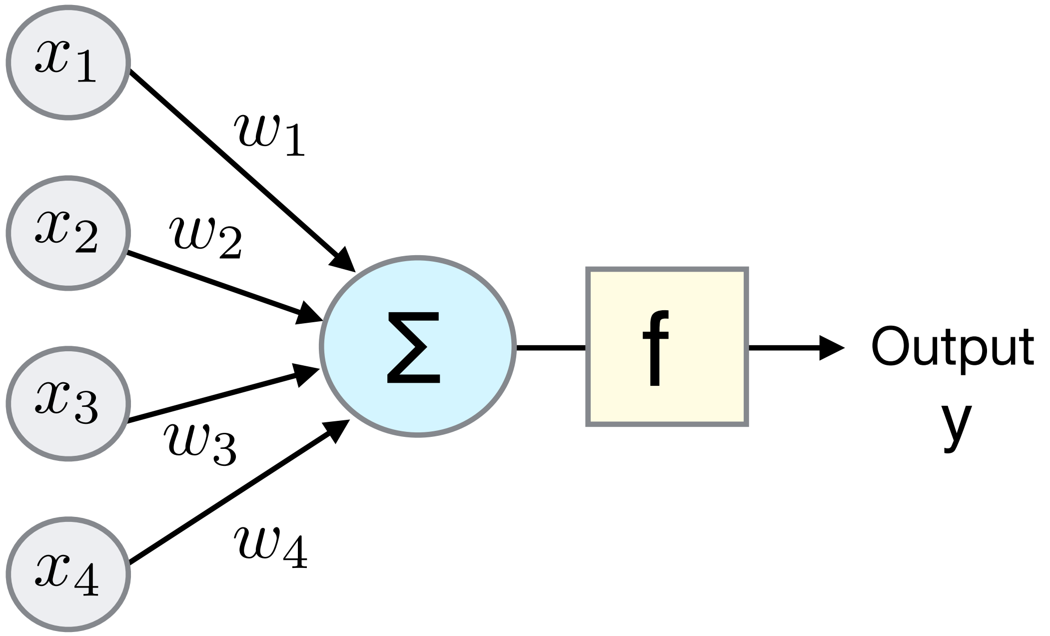
- We could determine weights w_i indicating how important each factor is in making the decision.
- For example, if x_2 is the most important factor, we might choose weights $w_1 = 0.2$, $w_2 = 0.6$, $w_3 = 0.2$
- If the weighted sum is greater than some predefined threshold, the customer might decide to move on to another restaurant ("No")

$$w_1x_1 + w_2x_2 + w_3x_3 \geq \text{threshold}$$

$$\text{e.g.}\ (0.2 \times x_1) + (0.6 \times x_2) + (0.2 \times x_3) \geq 0.8$$

Modelling Neurons

- An artificial neuron makes decisions by weighing up evidence. It takes many input signals $\{x_1, x_2, \ldots\}$ and produces a single output.
- The inputs each have weights $\{w_1, w_2, \ldots\}$. These are real numbers which indicate the importance of the inputs to the output.
- The output y is computed by applying some function f to the weighted sum of the input signals $\{x_1, x_2, \ldots\}$. This is often called the activation function.
- **Example:** Neuron with 4 inputs and 4 corresponding weights.



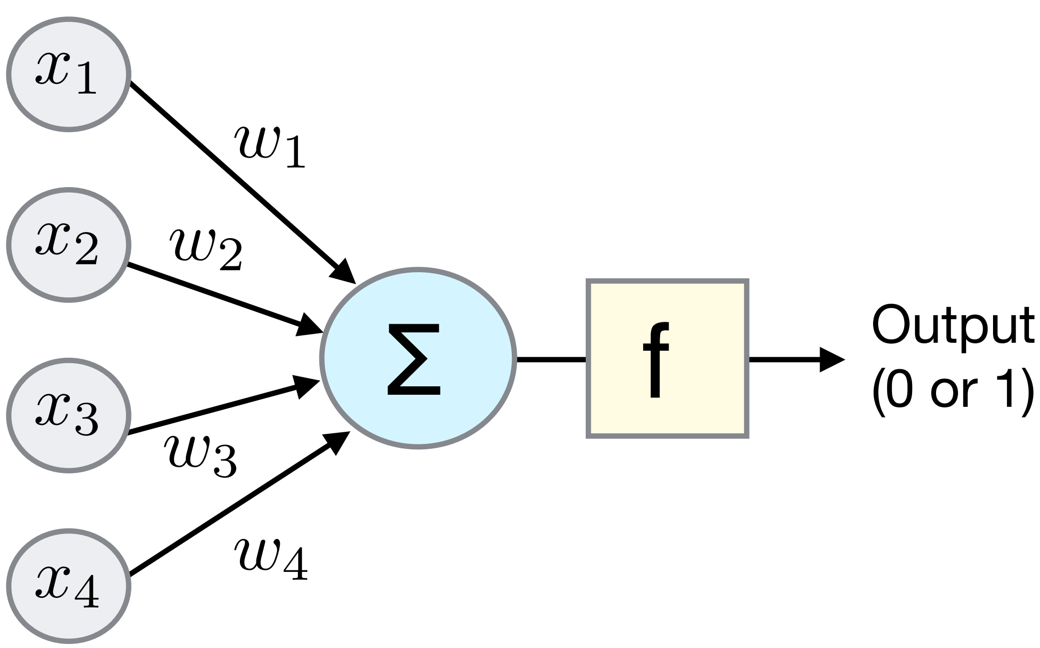
$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4)$$

f : activation function

Note: neurons are sometimes called "units" or "nodes".

Perceptrons

- A perceptron is an artificial neuron which takes in many input signals and produces a single binary output signal (0 or 1). It can be used to solve simple binary classification problems.
- To produce a binary output (i.e. a classification decision), we apply a threshold function to the weighted sum of inputs $\{x_1, x_2, \ldots\}$
- This function determines if the perceptron activates ("fires").
- **Example:** Perceptron with 4 inputs and 4 corresponding weights.

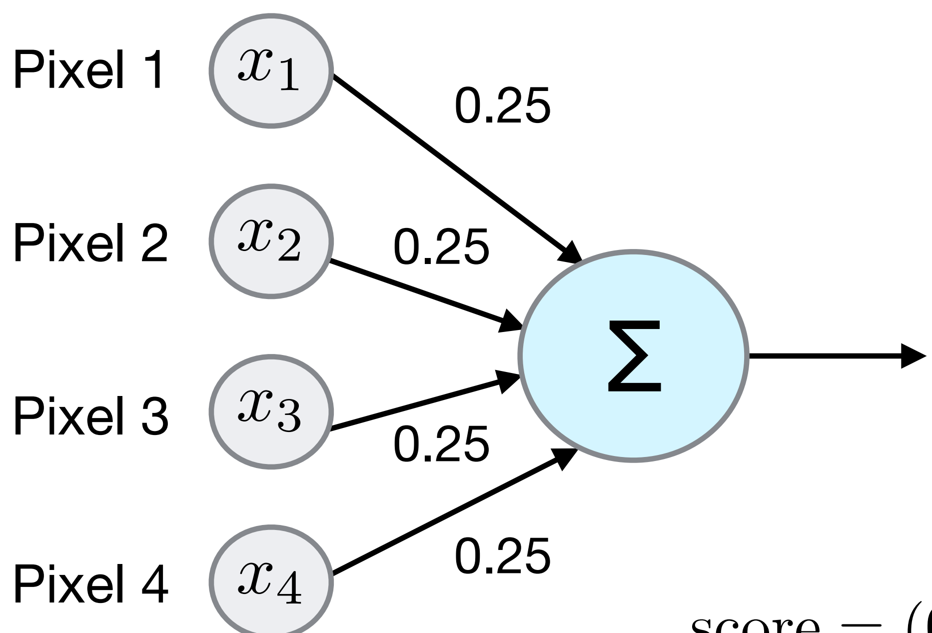


f : threshold function

$$\text{Output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j < \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j \geq \text{threshold} \end{cases}$$

Example: Perceptron

- Input is a 2x2 B&W image (i.e 4 pixels). Task is to classify the brightness of the image. All weights are equal (0.25) and using threshold value of 0.6. Output is "Dark" (0) or "Bright" (1).



Apply threshold function

$$\text{Output} = \begin{cases} 0 \text{ (Dark)} & \text{if score} < 0.6 \\ 1 \text{ (Bright)} & \text{if score} \geq 0.6 \end{cases}$$

$$\text{score} = (0.25 \times x_1) + (0.25 \times x_2) + (0.25 \times x_3) + (0.25 \times x_4)$$

1.0	1.0
1.0	1.0

score = 1.0

$\geq 0.6 \implies$ Bright

1.0	0.5
0.5	1.0

score = 0.75

$\geq 0.6 \implies$ Bright

1.0	0.5
0.5	0.0

score = 0.5

$< 0.6 \implies$ Dark

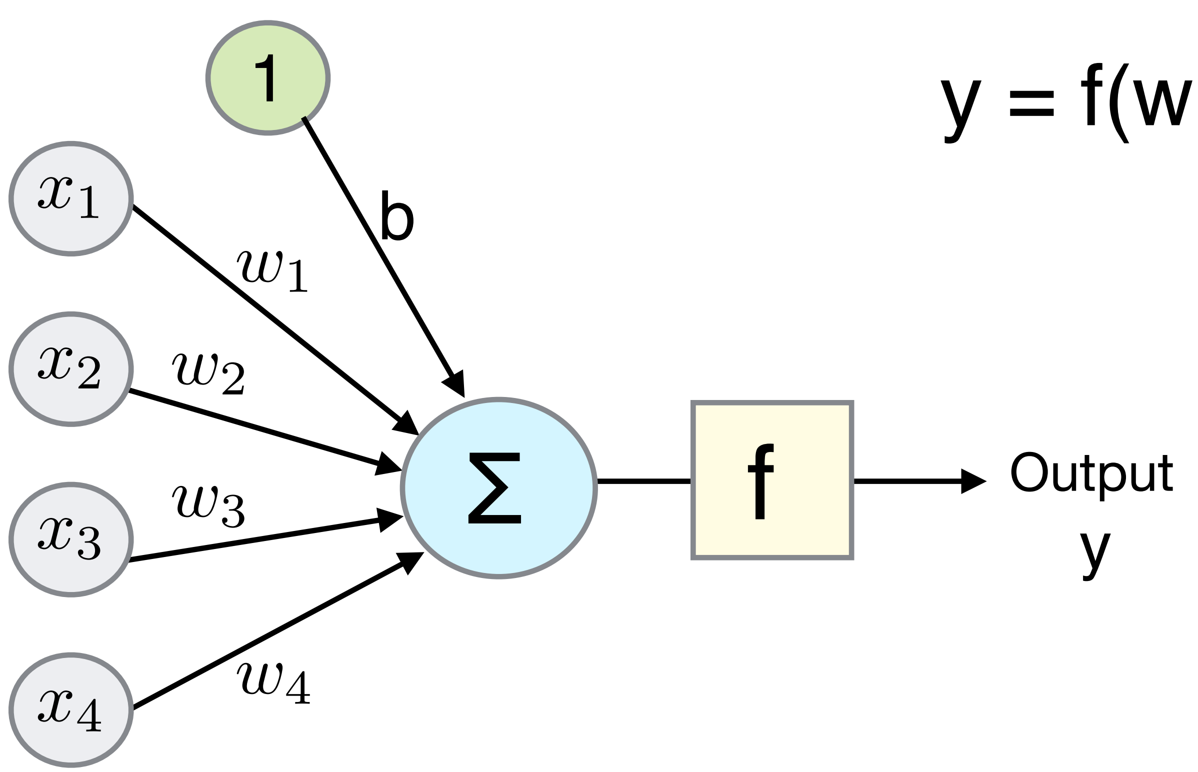
0.0	0.5
0.5	0.0

score = 0.375

$< 0.6 \implies$ Dark

Bias Term

- A bias term can be included in the network by adding an extra neuron with value $x_0 = 1$ to the inputs with weight b.
- Intuitively the bias represents how difficult it is for a particular neuron to send out a signal. It "shifts" the activation function.
- The bias term is treated like any other weight in the activation function. The main function of bias is to provide every neuron with a trainable constant value, in addition to the inputs.
- **Example:** Neuron with 4 inputs and a bias.



$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b)$$

f : activation function

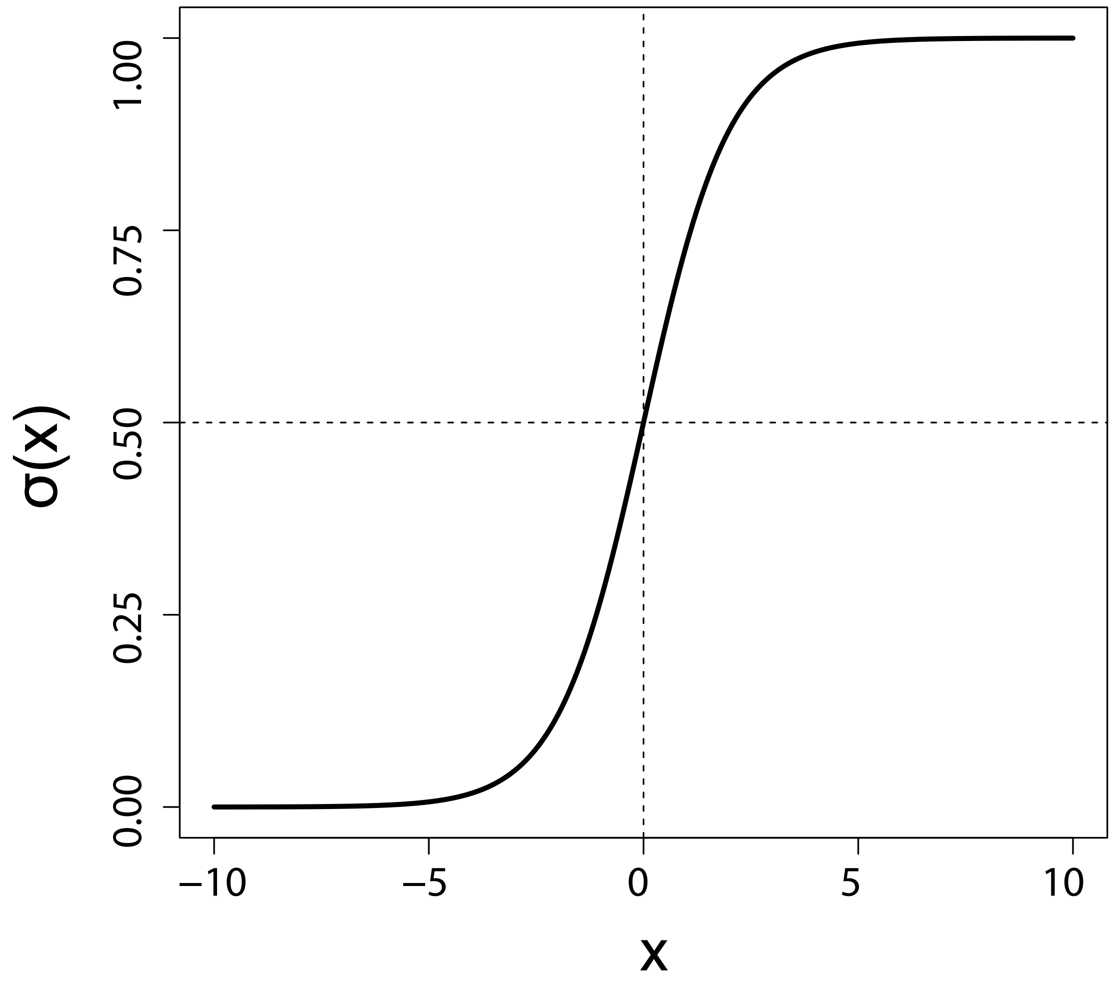
Now the weights and the bias impact on the output of the neuron.

Activation Functions

- An issue with a perceptron is that a small change in the input values can cause a large change in the output. This is because it has has only two possible states: 0 or 1.
- Instead we can use alternative activation functions which produce a continuous output.
- The sigmoid function (also called logistic function) is a common example, which follows a S-shape.

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

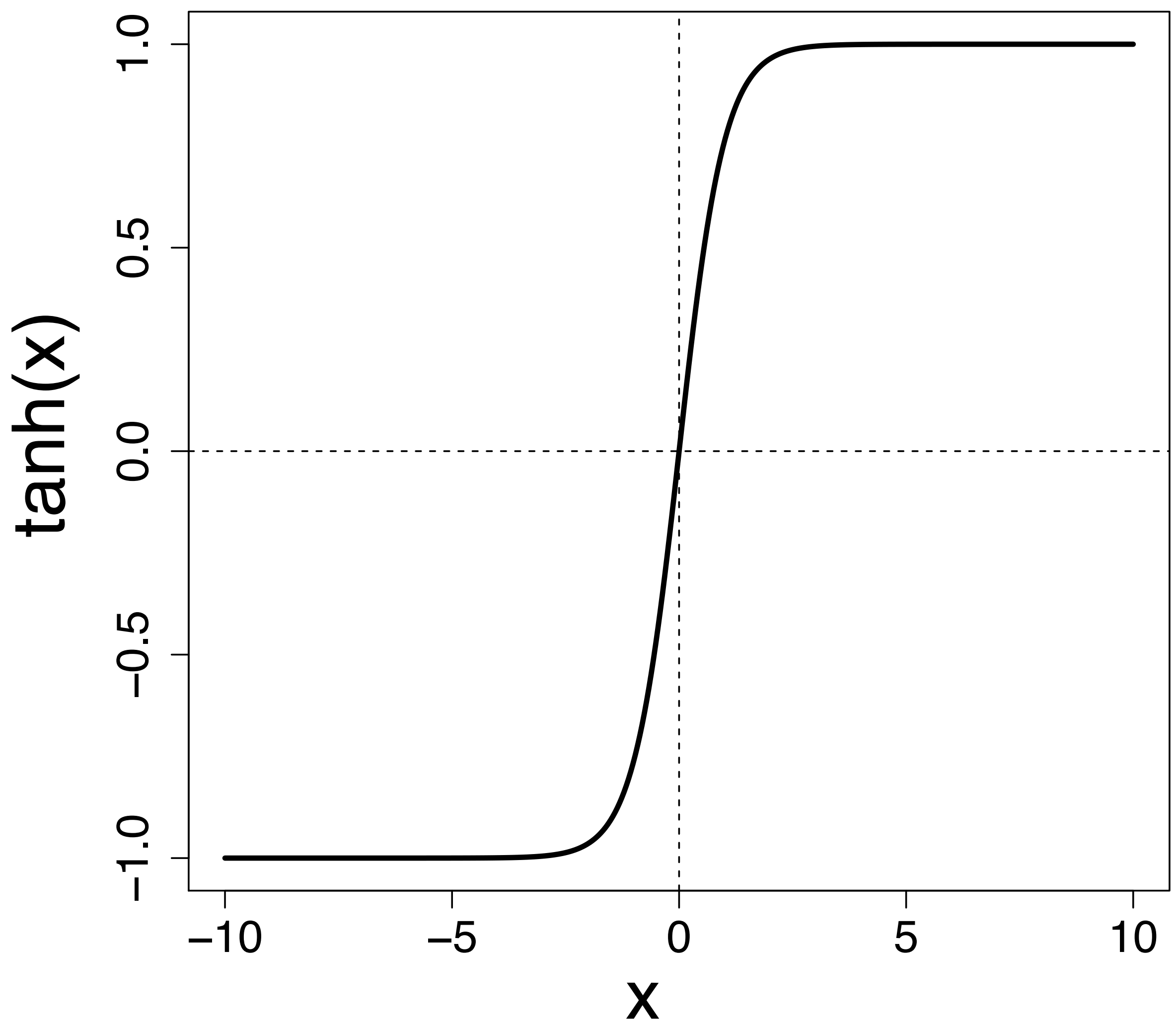
For a particularly positive or negative value of *x*, the output will be similar to a perceptron (0 or 1). For values closer to the boundary, the output will be near 0.5.



Activation Functions

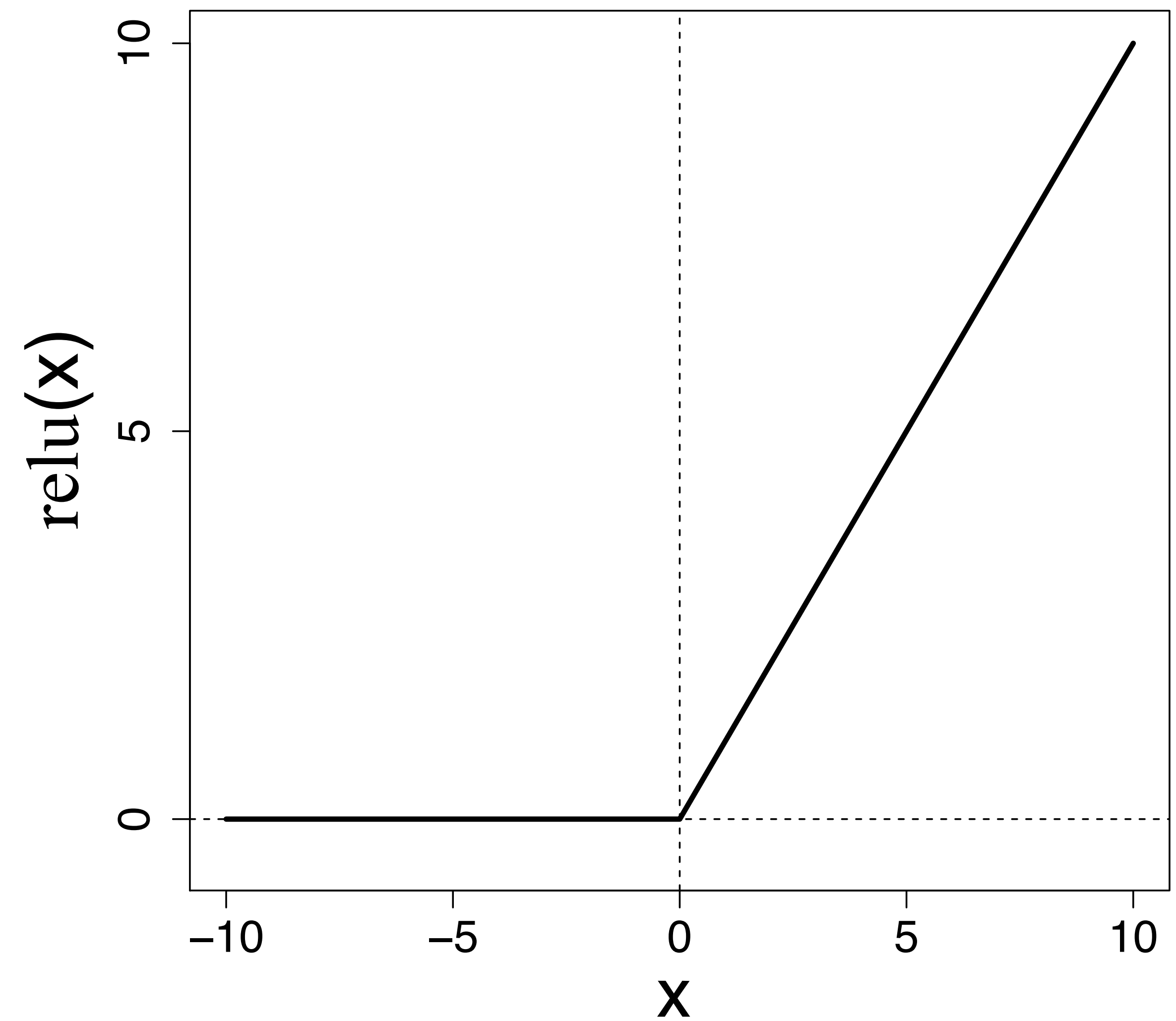
- Many other activation functions have been proposed...

Hyperbolic Tangent



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

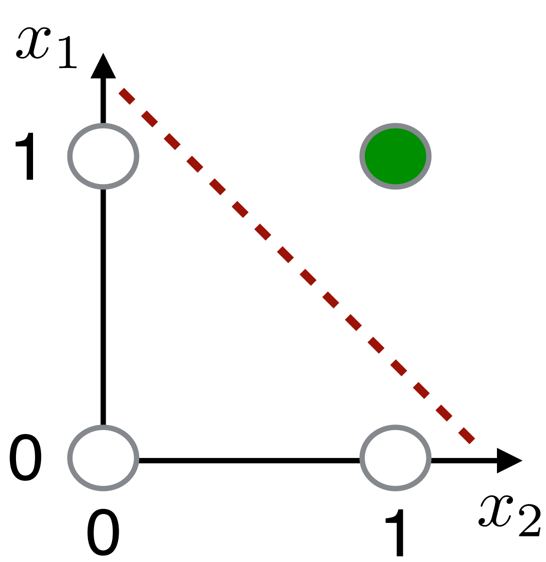
Rectifier



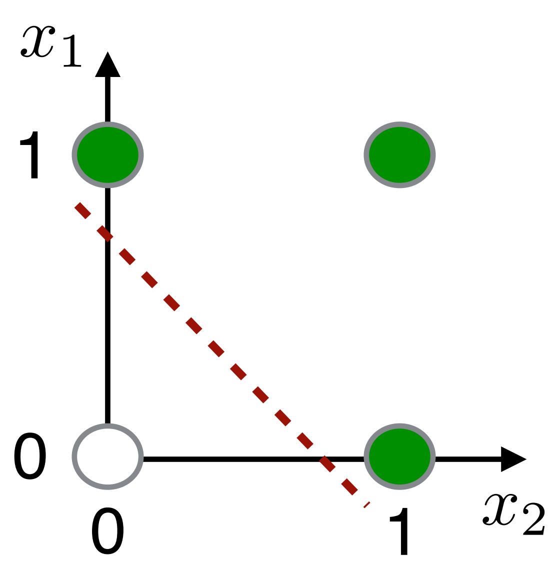
$$\mathrm{relu}(x) = \max(0, x)$$

Perceptron Limitations

- A single perceptron can only handle linearly separable problems i.e. if there is a line that can divide the fires and the non-fires.
- **Example:** Boolean AND and OR functions are linearly separable.



x_1 AND x_2



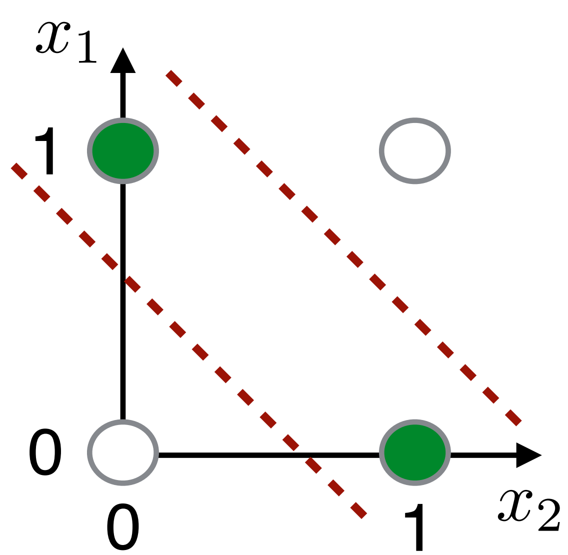
x_1 OR x_2

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Perceptron Limitations

- A single perceptron can only handle linearly separable problems i.e. if there is a line that can divide the fires and the non-fires.
- **Example:** Boolean AND and OR functions are linearly separable. But the XOR ("Exclusive OR") function is not linearly separable.



x_1 XOR x_2

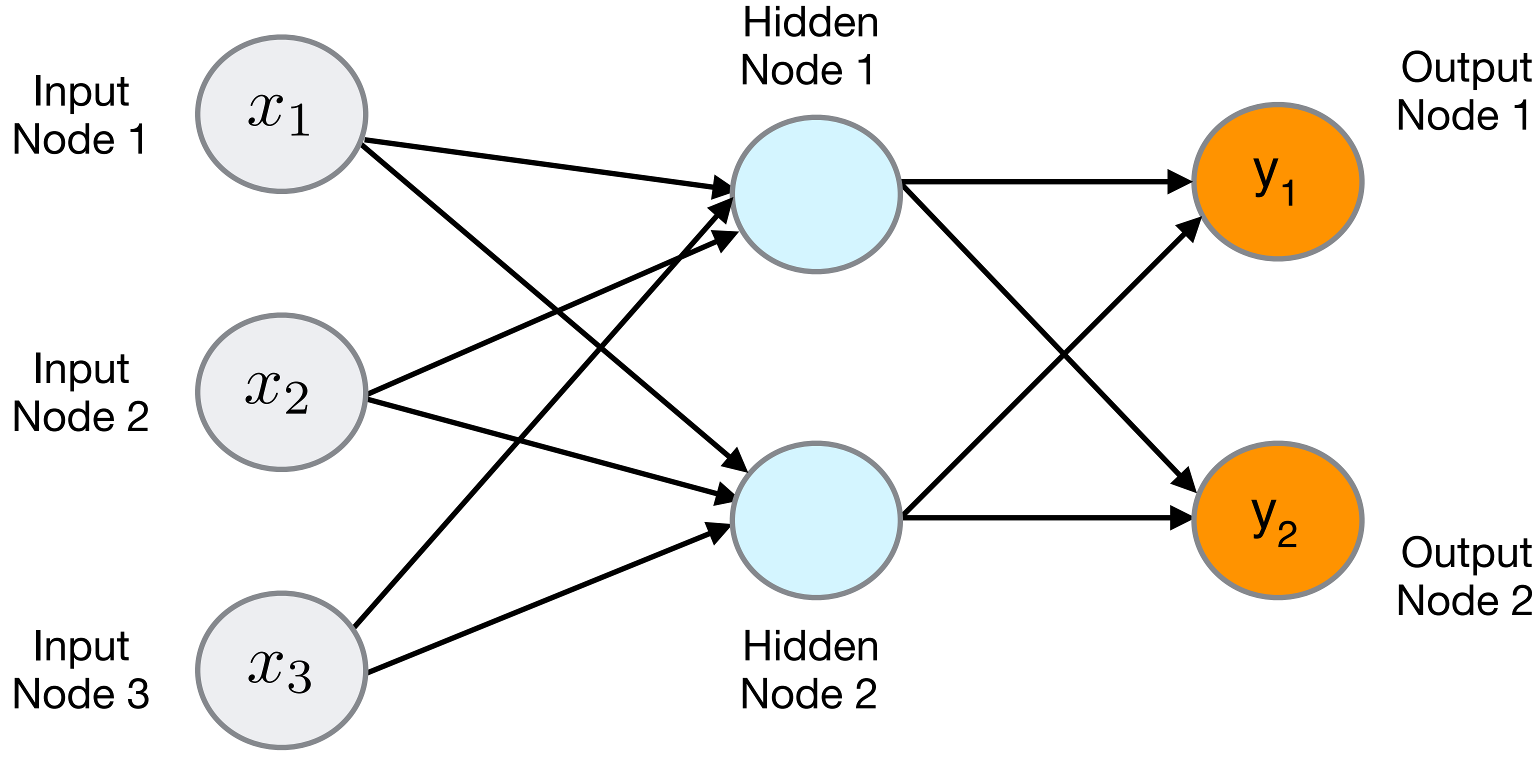
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

(Minsky & Papert, 1969)

Q. How can we solve non-linearly separable problems like this?

Multilayer Networks

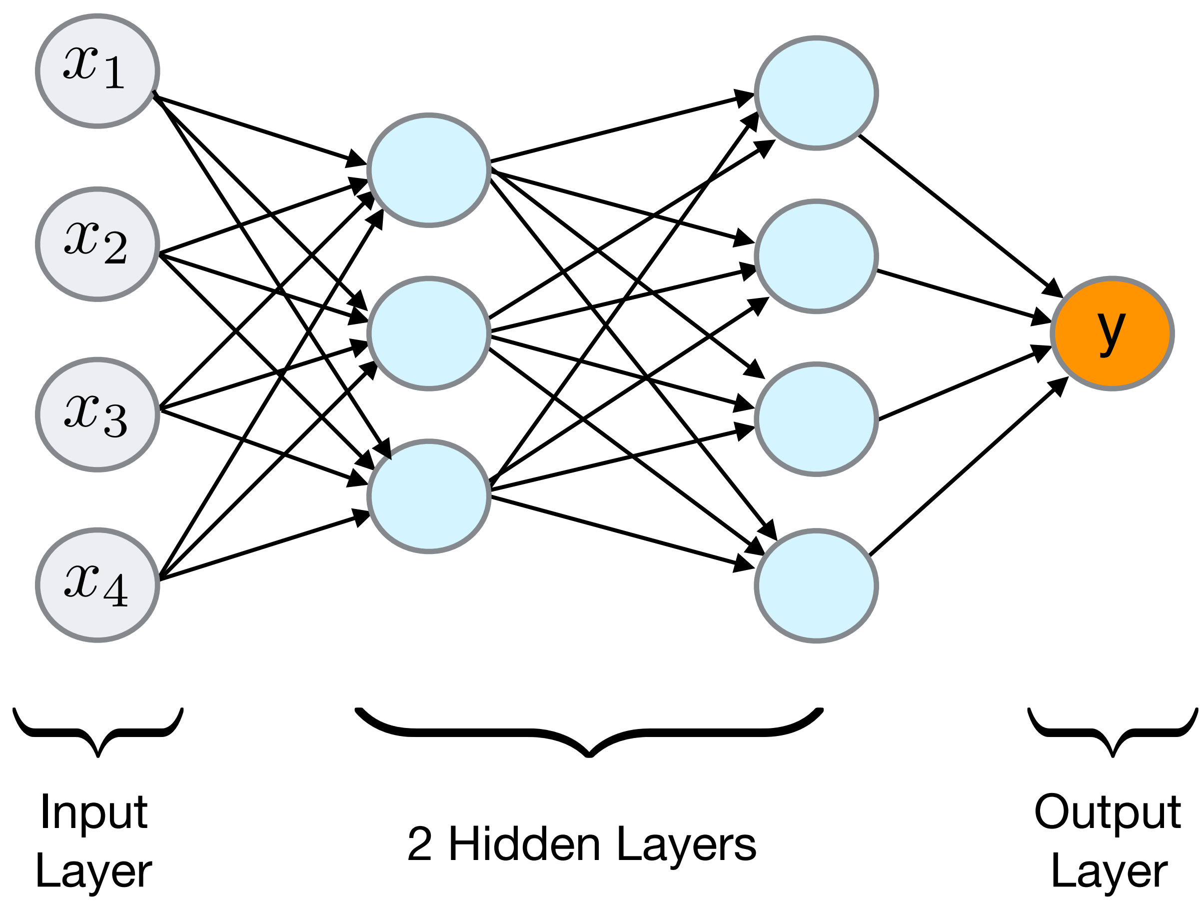
- Depending on the problem, complex decisions may require long chains of computational stages.
- By building a slightly more complicated neural network with an intermediary layer, we can solve non-linear problems that cannot be solved using only a single layer of inputs and outputs.
- The inputs and outputs are typically represented as separate nodes. The network can have multiple outputs as well as inputs. The nodes in between are "hidden".



Multilayer Networks

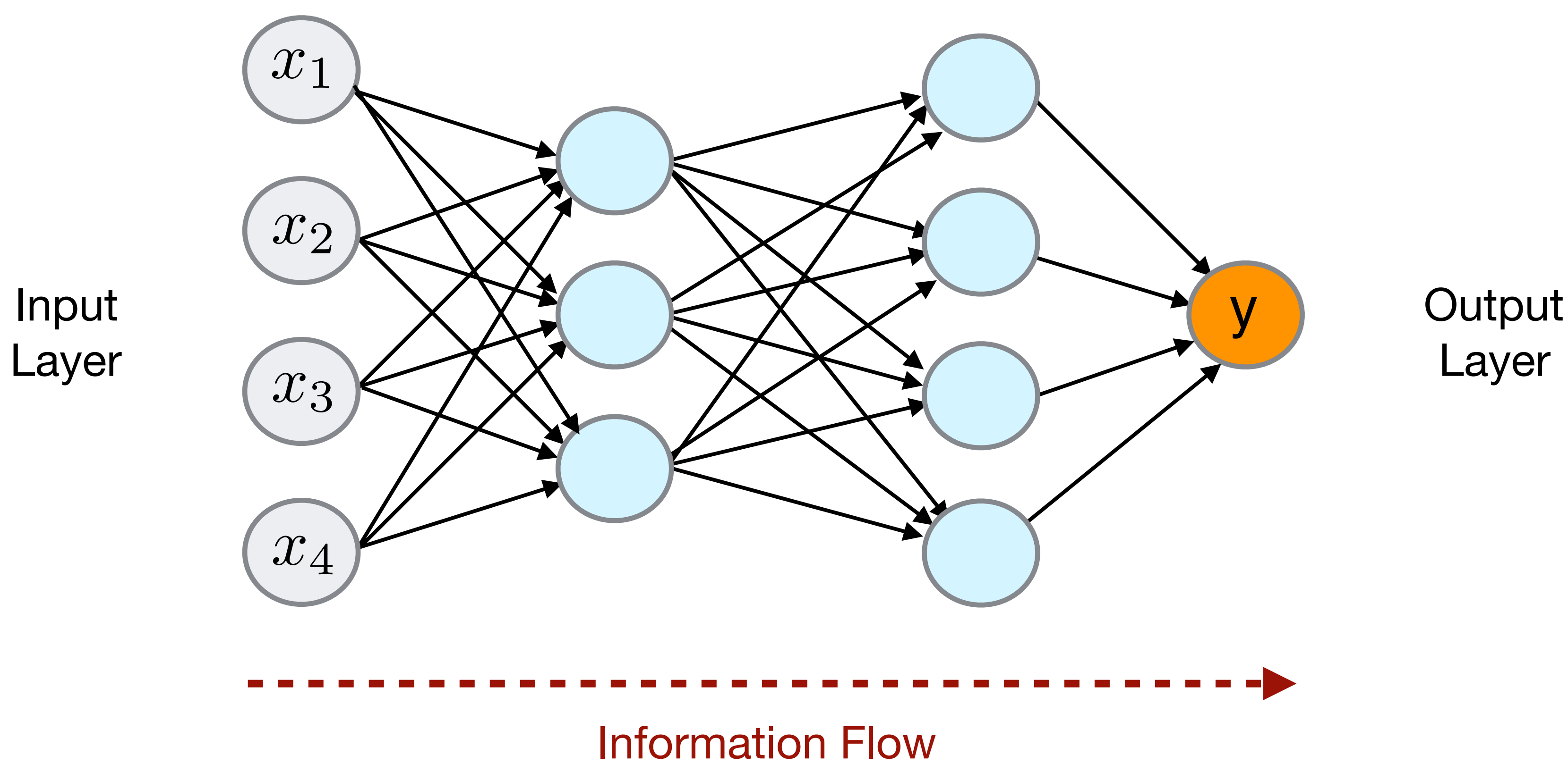
- We can build many different network configurations involving multiple layers of nodes (neurons):
  - 1st layer makes a decision based on the inputs.
  - 2nd layer makes a decision based on the decision from 1st layer.
  - 3rd layer can make an even more complex decision etc...

- The layers of nodes stacked between the inputs and outputs are called the hidden layers.
- Note nodes can feed into multiple new nodes in the next layer.



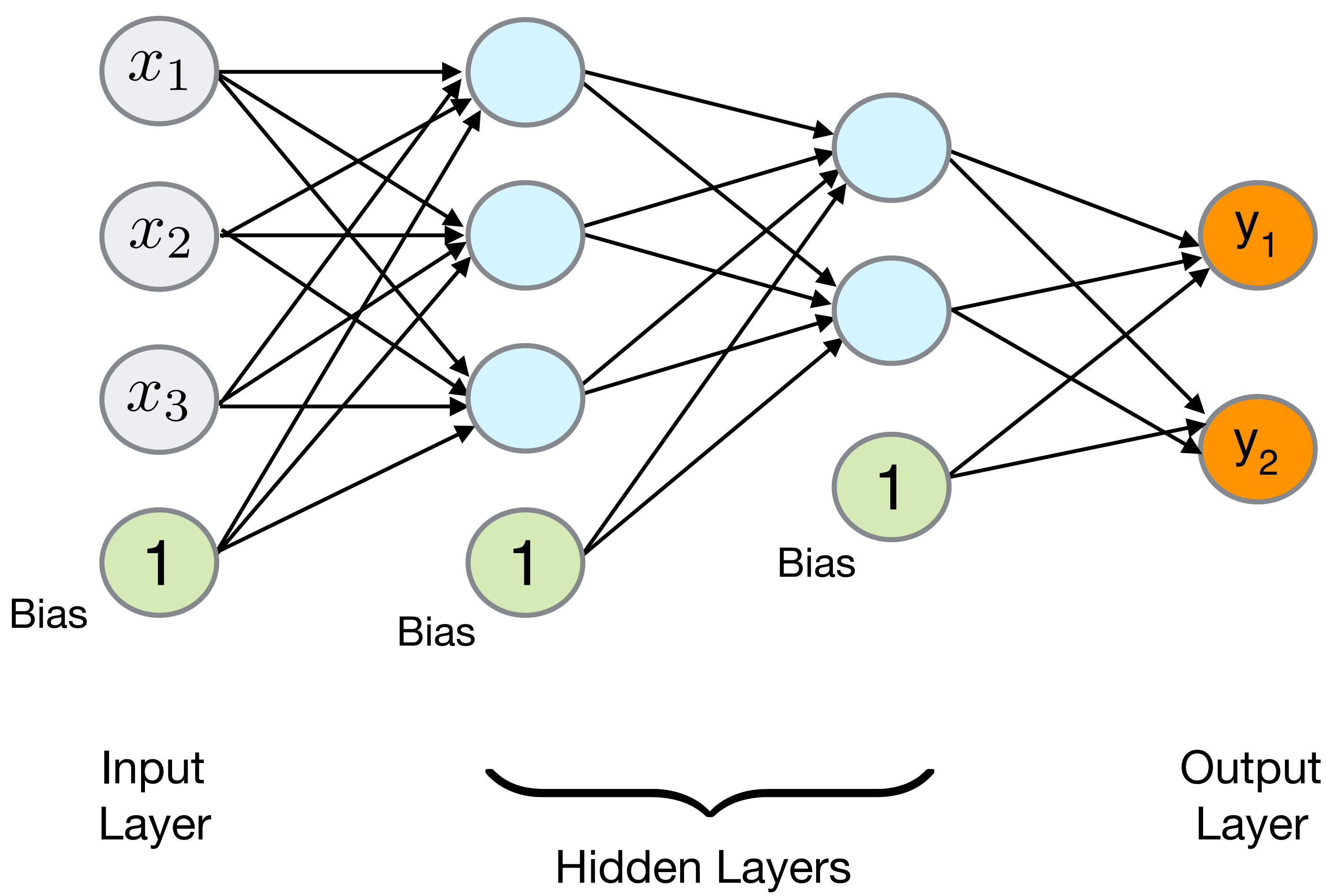
Feedforward Networks

- Since information only flows in one direction, this kind of network is often referred to as a feed-forward network.
- Each layer consists of nodes which receive their input from the previous layer directly above, and send their outputs to the next layer directly below. There are no connections within a layer.
- To compute the output of the network, we can successively compute all the activations in Layer 1, Layer 2, etc.



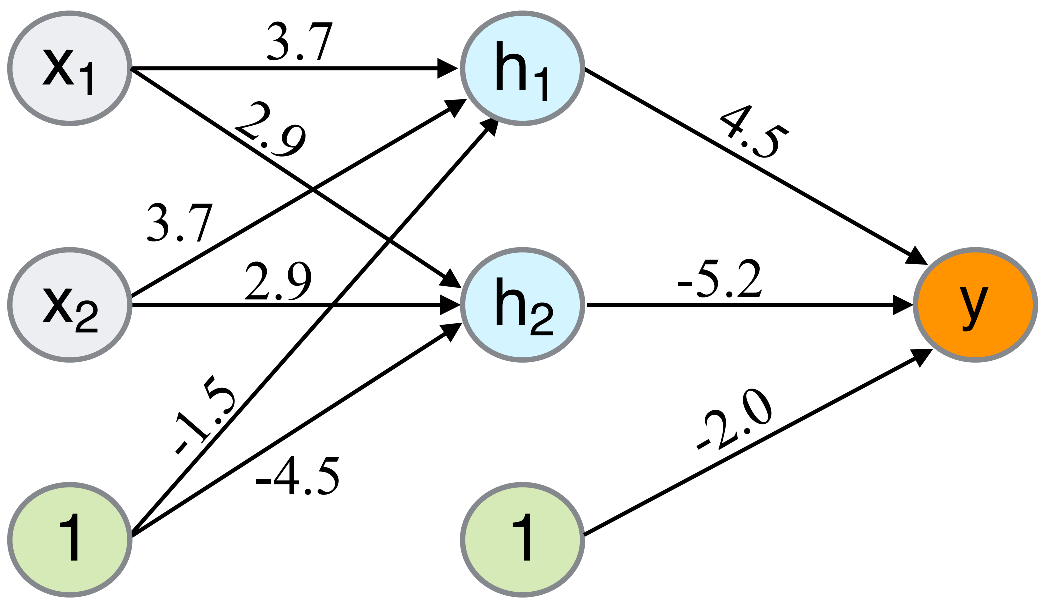
Bias in Multilayer Networks

- In multi-layer networks we can add a bias term, with a constant value 1, as an extra node in each pre-output layer. In this case, the bias terms can all have different weights.
- These bias nodes are not connected to any nodes in the previous layer, so are not influenced by the outputs from previous layers.



Example: Multilayer Network

- **Configuration:** One input layer with 2 inputs. One hidden layer, one output. Using a sigmoid activation function.

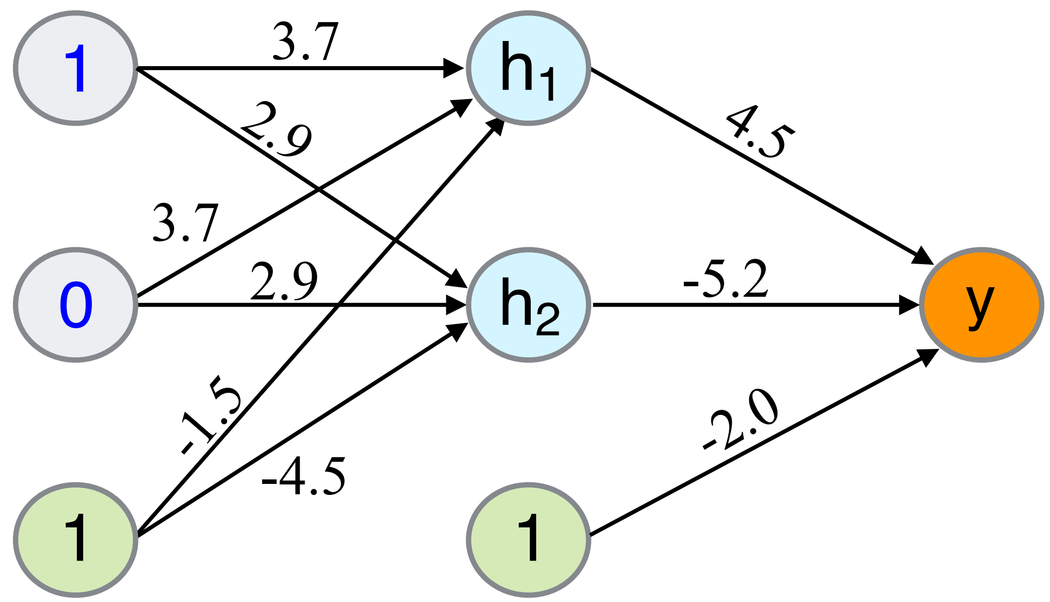


Example: Multilayer Network

- **Configuration:** One input layer with 2 inputs. One hidden layer, one output. Using a sigmoid activation function.

- Consider the case of the inputs:

  $x_1 = 1, x_2 = 0$



- Compute values based on the inputs:

$$h_1 = \sigma((1 \times 3.7) + (0 \times 3.7) + (1 \times -1.5) = \sigma(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$h_2 = \sigma((1 \times 2.9) + (0 \times 2.9) + (1 \times -4.5) = \sigma(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

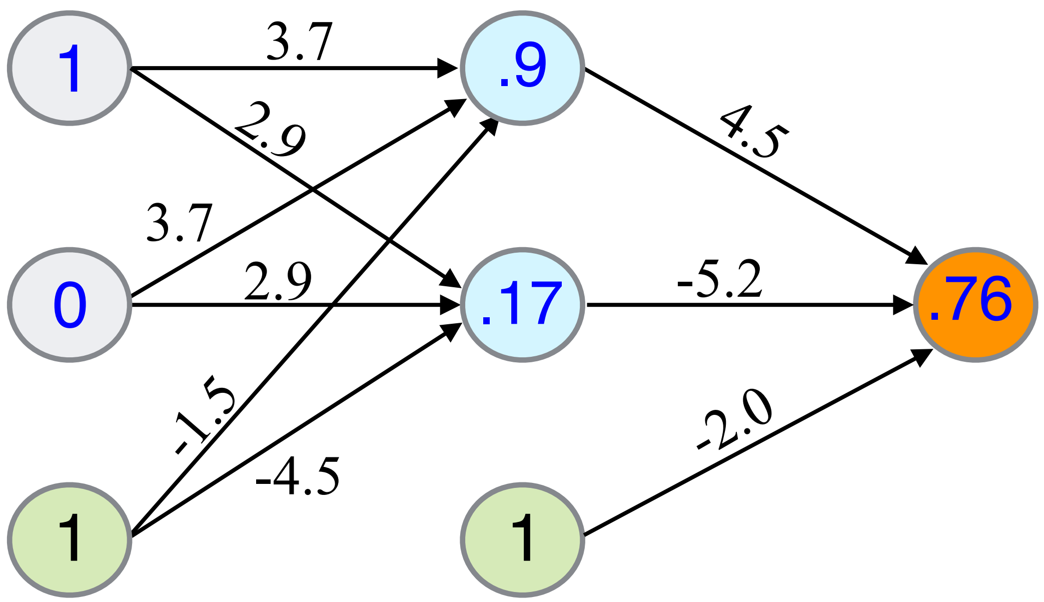
Example: Multilayer Network

- We now use these values as inputs to the hidden layer, in order to compute the output value of the network.
- Original inputs:

  $x_1 = 1, x_2 = 0$

- Inputs to hidden layer:

  $h_1 = 0.9, h_2 = 0.17$



- Compute output value:

$$y = \sigma((0.9 \times 4.5) + (0.17 \times -5.2) + (1 \times -2.0) = \sigma(1.17)$$

$$= \frac{1}{1 + e^{-1.17}} = 0.76$$

Example: Multilayer Network

- We can compute other network outputs for different pairs of binary inputs in the same way:

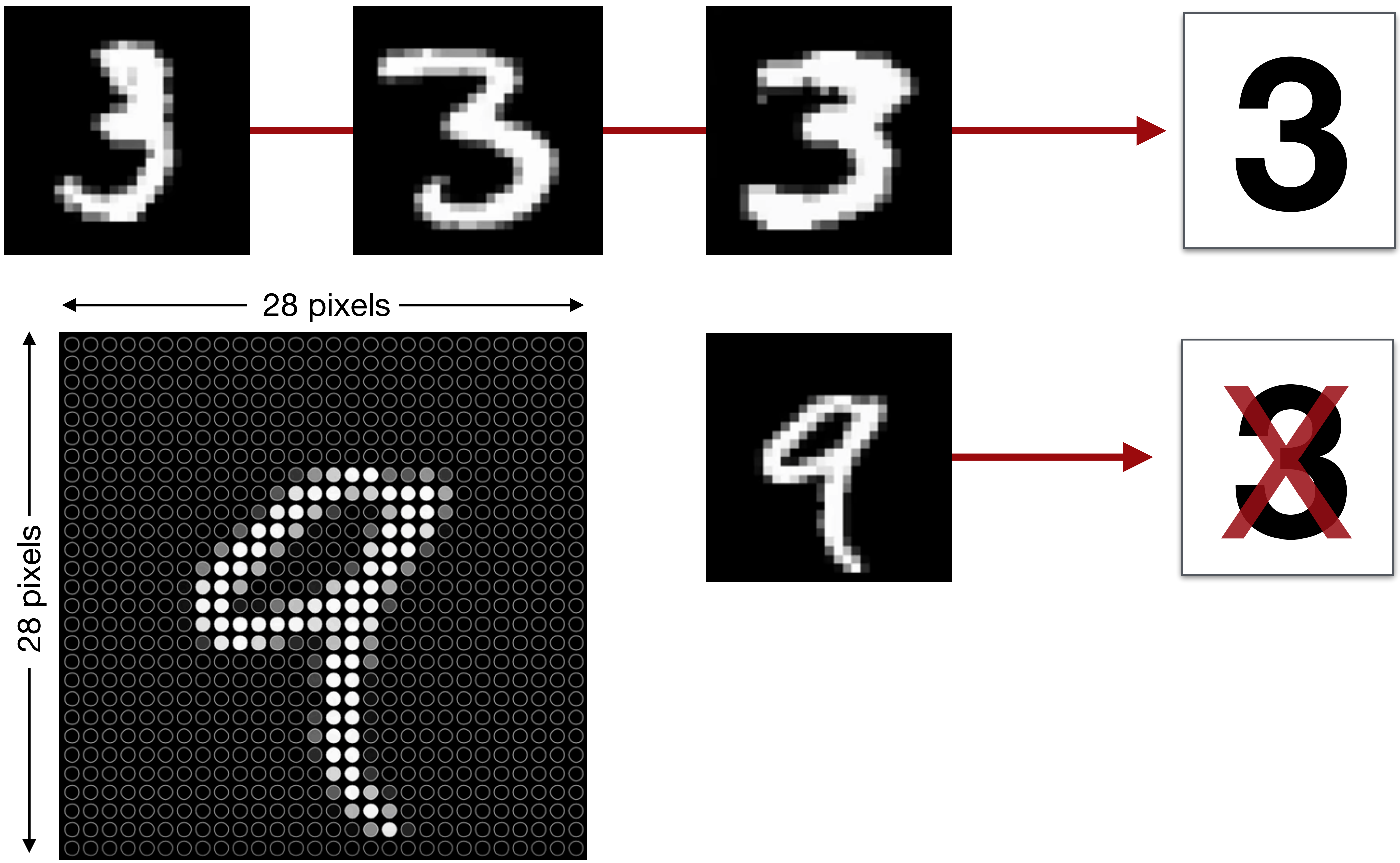
Input x_1	Input x_2	Hidden h_1	Hidden h_2	Output	Approx.
0	0	0.18	0.01	0.23	$\Longrightarrow$ 0
0	1	0.90	0.17	0.76	$\Longrightarrow$ 1
1	0	0.90	0.17	0.76	$\Longrightarrow$ 1
1	1	1.00	0.79	0.17	$\Longrightarrow$ 0

- Notice that this network roughly implements the XOR function.
- The hidden node h_1 implements the OR function.
- The hidden node h_2 implements the AND function.

➡ By chaining these, we can solve a more complex problem.

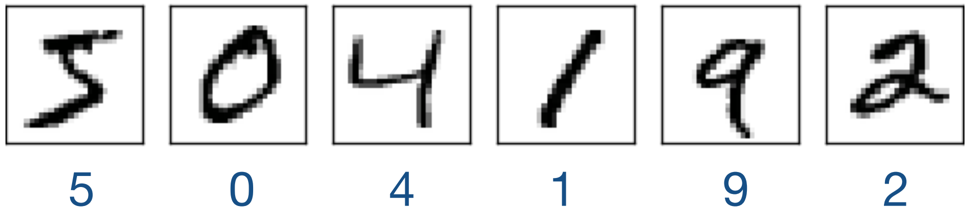
Example: Handwritten Digits

- **Task:** How can we learn to automatically recognise hand written digits, based on their pixel representations?



Example: Handwritten Digits

- **Goal:** Classify handwritten digit images into classes (0,1,…9)
- Input: Training set of many 28x28 pixel images labelled with correct digit.

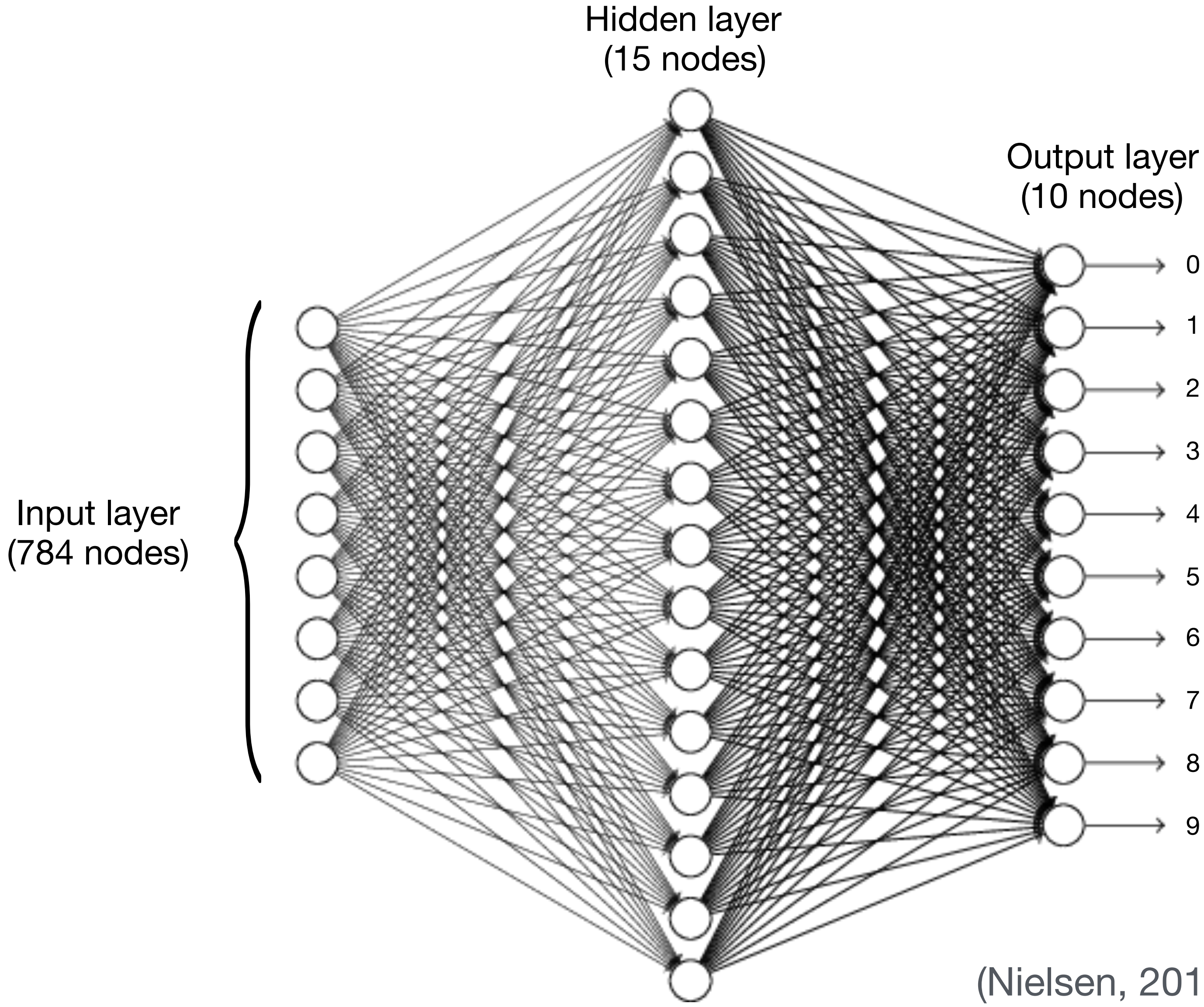


- Neural Network:

  Input layer:
  28x28 pixels=784 neurons

  Hidden layer:
  15 nodes

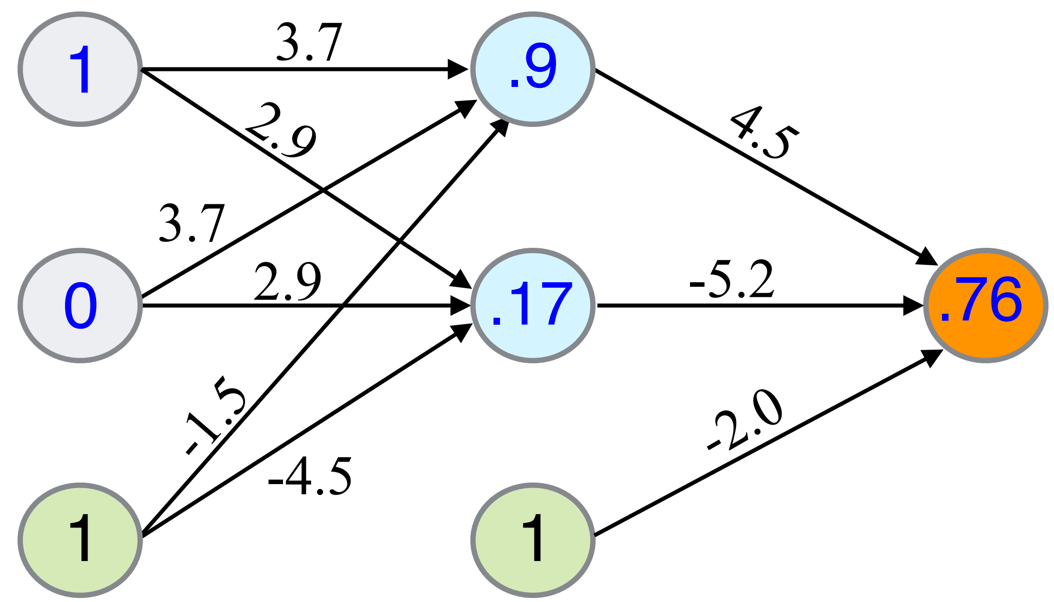
  Output layer:
  One output node per digit.
  To determine which class to assign for an input, we look at which of the output nodes has the largest value.



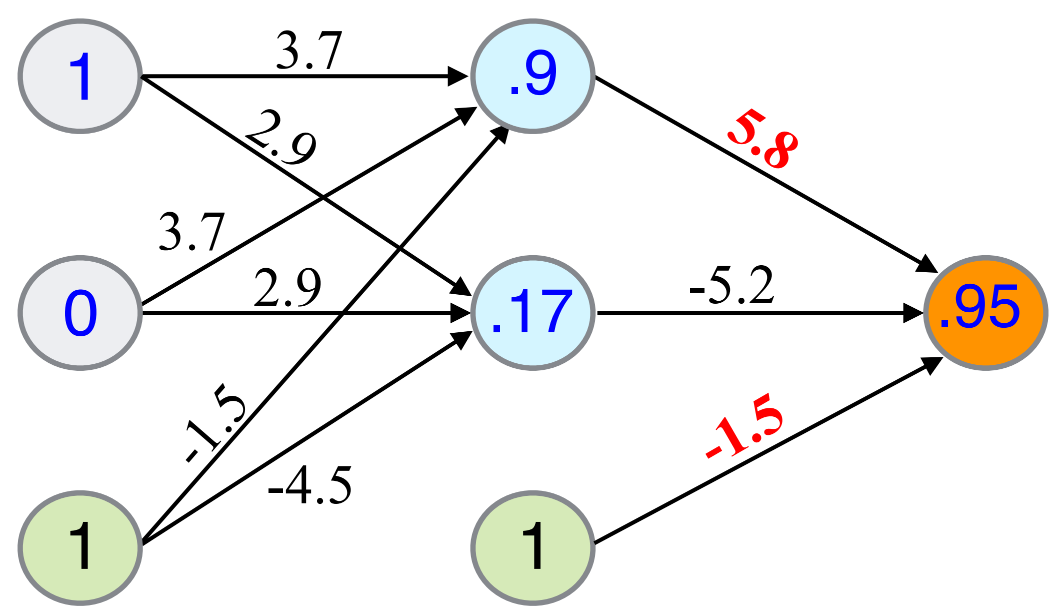
(Nielsen, 2015)

Error in Neural Networks

- Recall our previous network implementing the XOR function.
- Ideally the network's output for the inputs $x_1 = 1$, $x_2 = 0$ would have been 1.0, not 0.76



- We could adjust the weights in the network to reduce this error i.e the difference between the output values computed by the model and the correct values.
- Need to do this in a way that generalises to different inputs.



Cost Functions

- Once the architecture of the network has been chosen, its parameters (the weights w and biases b) need to be learned from the training data.
- Cost function: a function $C(w, b)$ is used in a neural network to quantify the inconsistency between predicted values and the corresponding correct values (also known as a loss function).
- The choice of cost function used in a network depends on the task being performed - e.g. regression, binary classification etc.
- Training a neural network involves applying an optimisation procedure to find weights and biases to minimise the cost function.
- The training algorithm has done a good job if it finds weights and biases so that the cost is low - i.e. $C(w, b) \approx 0$

Cost Functions

- A common cost function for regression is mean squared error. It is the average of the square of the difference between each predicted value $\hat{y}$ and the true value y:

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

w : All the weights in the network

b : All the biases in the network

n : Number of examples in the training set

- For binary classification, instead cross entropy (also called log loss) is often used. It measures the average uncertainty of the probabilities of the model p_i , compared to the true labels y :

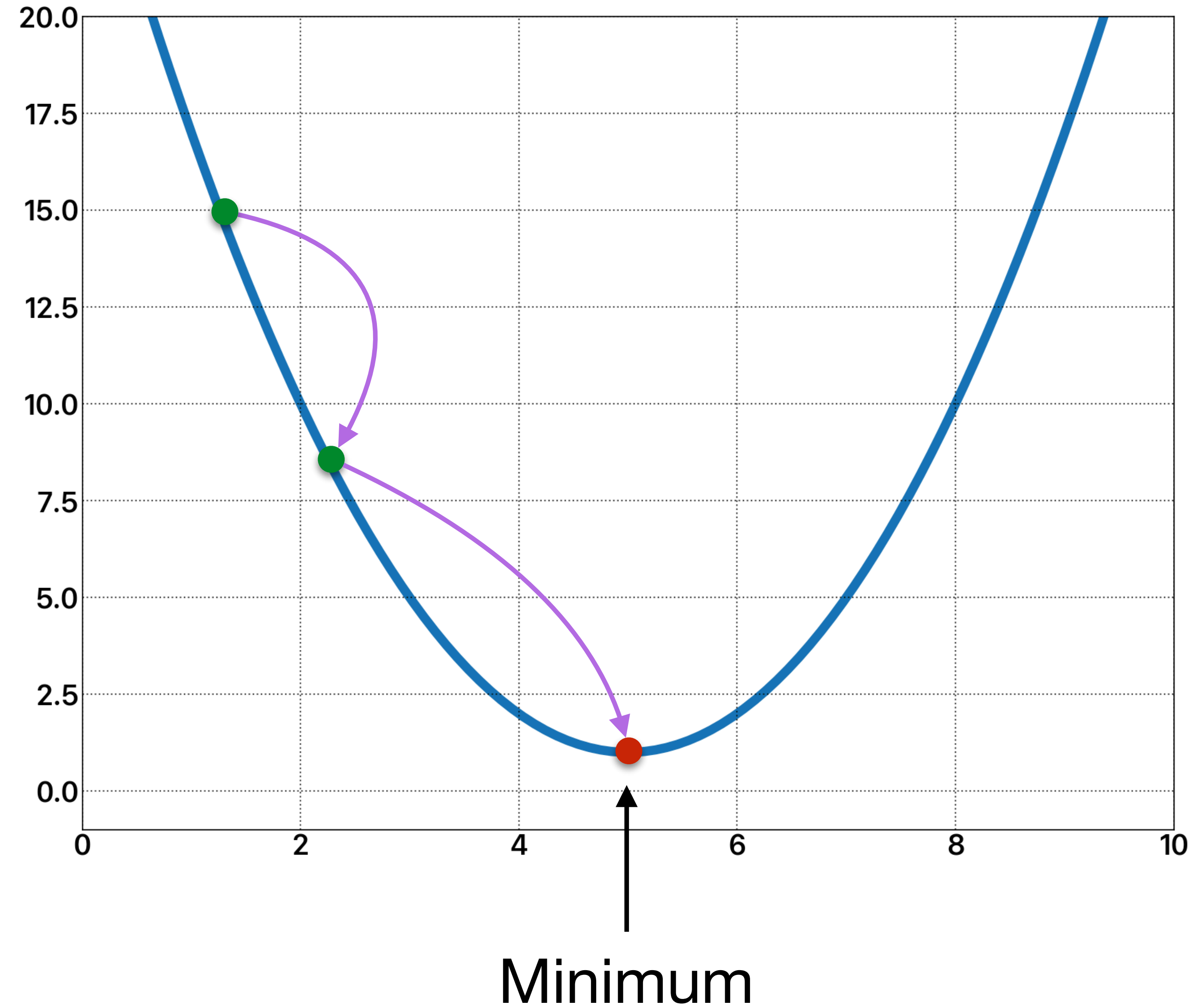
$$C(w,b) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log p_i + (1-y_i)\log(1-p_i)]$$

p_i : Predicted probability score from the model.

Q. How do we actually change the weights and biases in our neural network to minimise the value of a cost function $C(w,b)$?

Reminder: Gradient Descent

- Gradient descent is an algorithm that makes small steps along a function to find a local minimum.
- We start at some point and find the gradient (slope).
- We take a step in the opposite direction to the gradient (i.e. downhill). The size of the step is controlled by an adjustable parameter.
- This algorithm gets us closer and closer to the local minimum.

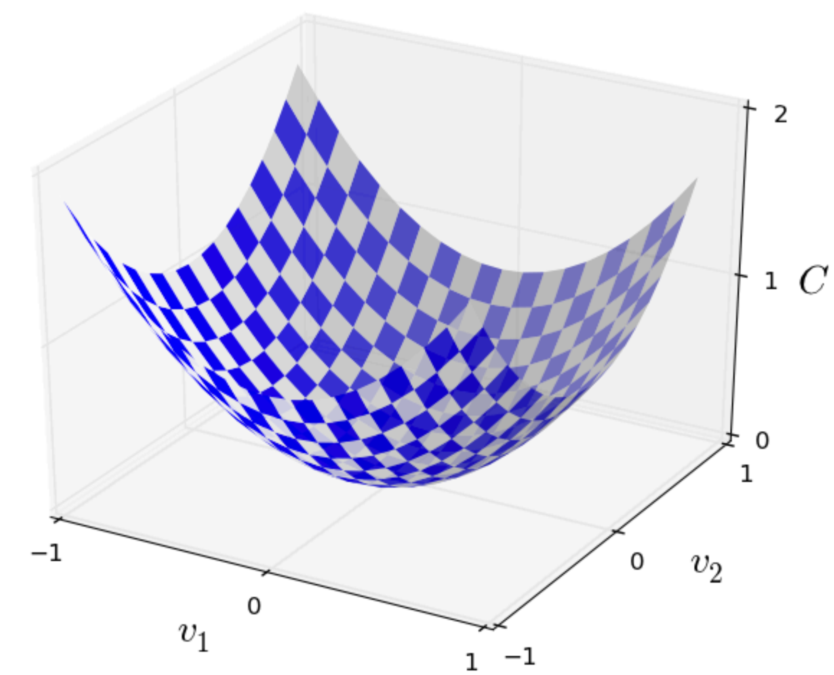


In a 3D space, it would be like rolling a ball down a hill to find the lowest point.

Training via Gradient Descent

- Neural networks generally apply the gradient descent algorithm to try to adjust all the weight and bias variables to find the minimum cost.
- This involves calculating the derivative of the cost function C.
- For the example with 2 variables, we can make a change Δv_1 and Δv_2 where the overall change in C is given by:

Example: 2 variables



(Nielsen, 2015)

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Partial derivatives

Δv_1 : Change to variable v_1

Δv_2 : Change to variable v_2

- We choose Δv_1 and Δv_2 so that the overall change ΔC is negative i.e. we want to get closer to the minimum.

Training via Gradient Descent

- We can define the gradient of C as the vector ∇C containing all variable partial derivatives:

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

- We can also create a vector containing all the changes for the variables:

$$\Delta v = (\Delta v_1, \Delta v_2)^T$$

- We can rewrite the overall change in the cost function C as:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \longrightarrow \Delta C \approx \nabla C \cdot \Delta v$$

- We can now choose values for the gradient Δv to make the change ΔC always negative by applying a learning rate η:

$$\Delta v = -\eta \nabla C$$

η : A small positive value

- This gives us an update rule for minimising the cost C:

$$v \rightarrow v' = v - \eta \nabla C$$

Stochastic Gradient Descent

- In a neural network, rather than just two variables (v_1, v_2), we have many variables (the weights w and biases b), but the core optimisation strategy remains the same:

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

Update rule for weights

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

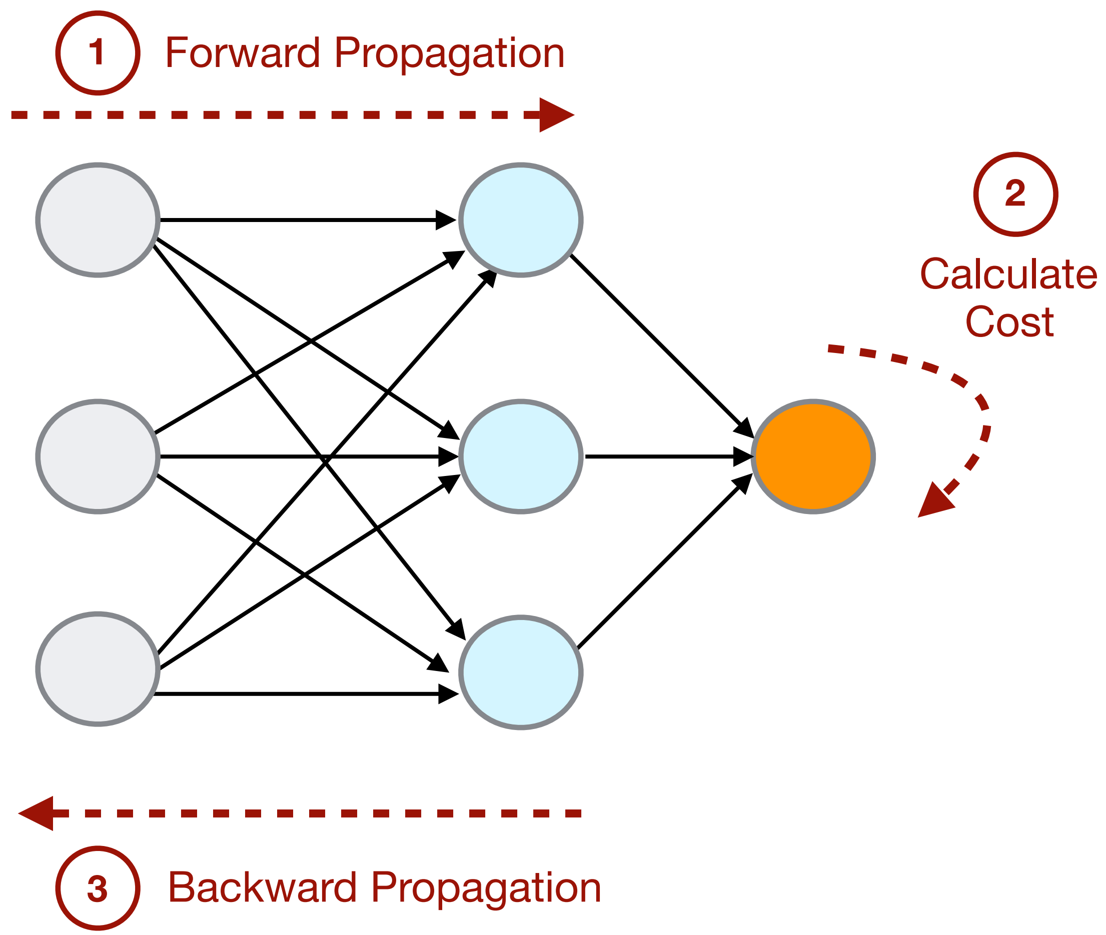
Update rule for biases

η : Learning rate

- When learning on large training sets, repeatedly computing the gradients can be a very slow process.
- In practice, stochastic gradient descent is often applied to speed up learning, which involves estimating the gradient ∇C for only a small batch of randomly selected training examples.
- This can provide a good estimate, while significantly reducing the time required to train a network.

Backpropagation

- To train the network, we start with random initial guesses for the weights and biases in the model. The optimisation algorithm then repeats a cycle of propagations and weight updates.
- Forward propagation: Feed training examples through the network layers, and calculate the resulting outputs. Use the cost function to measure the difference between the outputs and the correct answers.
- Backpropagation: Starting at the output layer, propagate errors back through the network, which allows us to calculate the gradient of the cost function.
- Weight update: The gradient is fed to the optimisation method, which in turn uses it to update the weights, in an attempt to minimise the cost function.

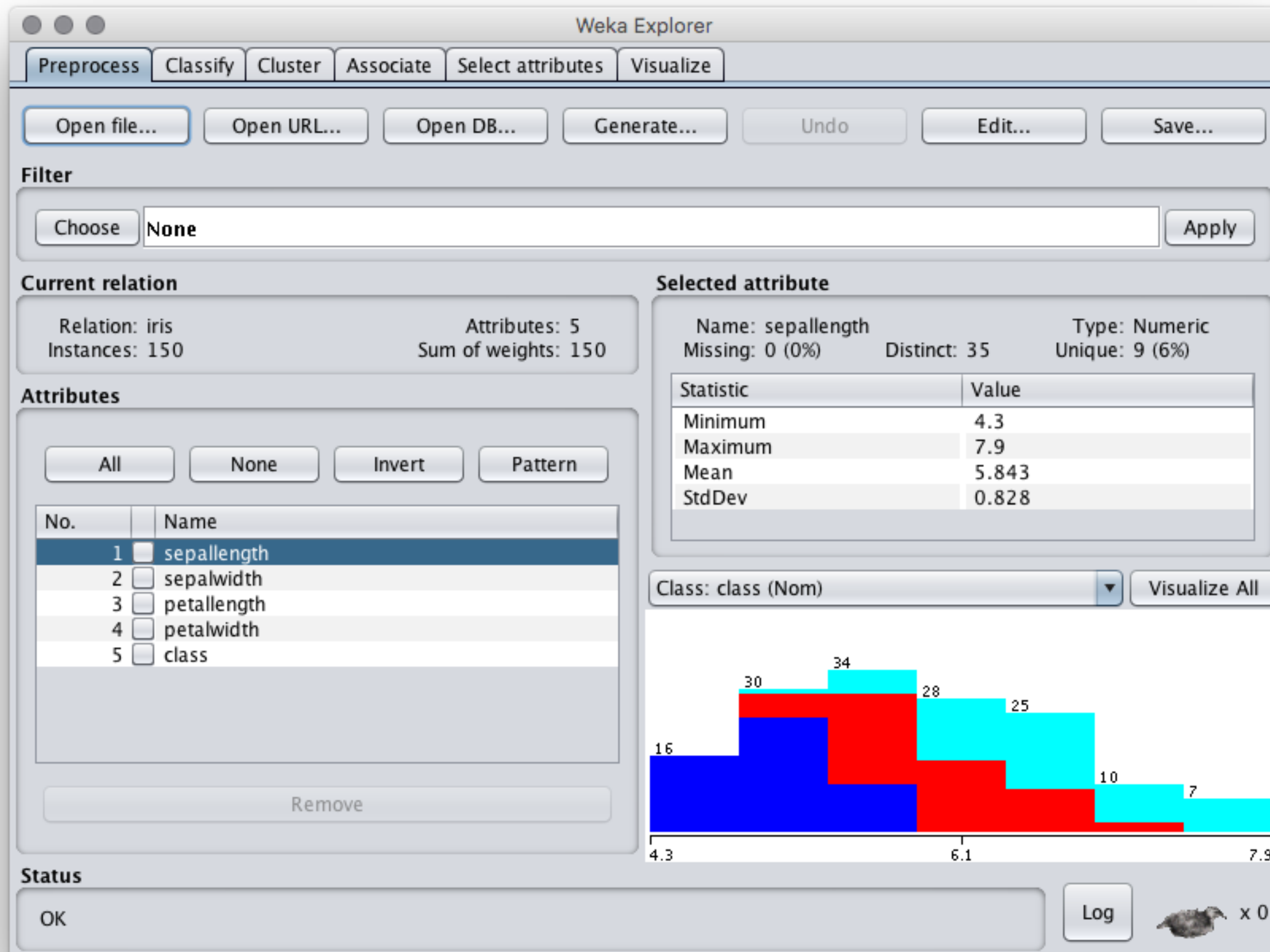


Training Neural Networks

- Applying gradient descent in the context of training neural networks involves the following steps:
  - Choose initial weights (typically small random values)
  - Until convergence repeat:
    1. Set all gradients to 0
    2. For each training example
       a. Predict the output from the model
       b. Calculate the resulting cost
       c. Update the gradients for each weight and bias term
    3. Update the weights and biases using weight update rule

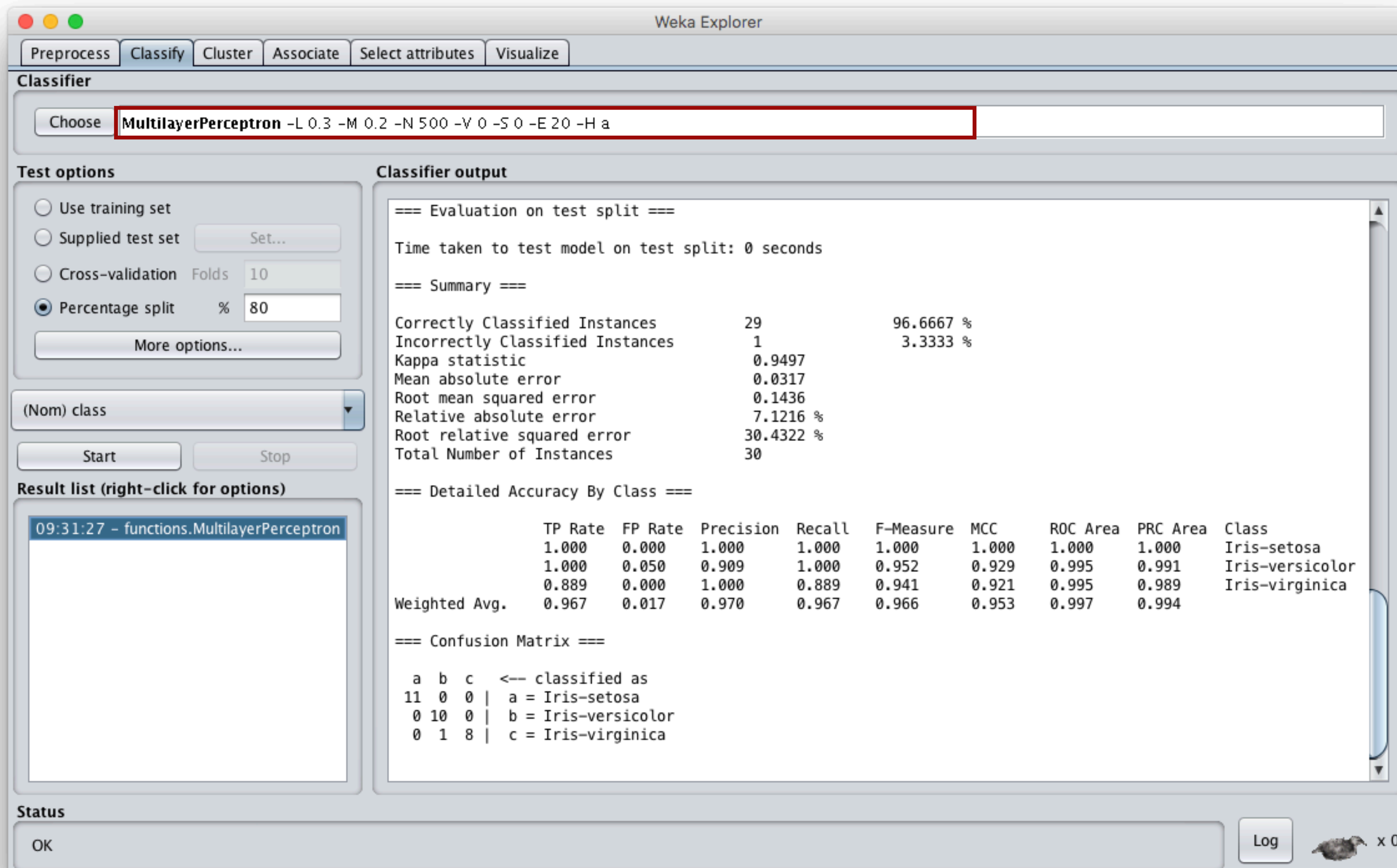
Neural Networks in Weka

1. Launch the WEKA application and click on the *Explorer* button
2. *Open File* - iris.arff



Neural Networks in Weka

3. In *Classify* tab, click *Choose* and *Functions*→ *MultilayerPerceptron*
4. Select *Test options: Percentage split* and set it to 80%
5. Click *Start*



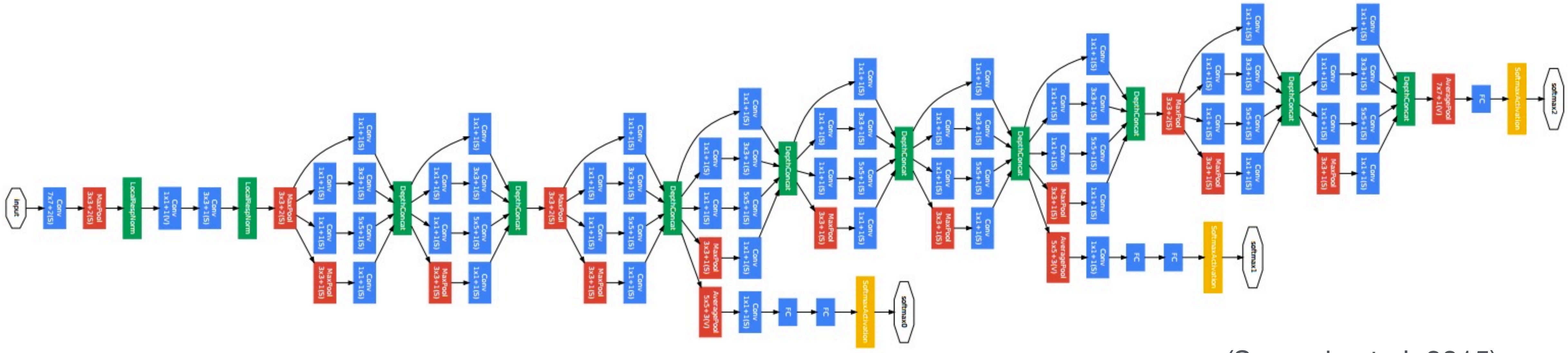
What are Neural Networks Good For?

- **Advantages**
  - Can learn and model non-linear and complex relationships.
  - Work well when training data is noisy or inaccurate.
  - Fast performance once a network is trained.
- **Disadvantages**
  - Often require a large number of training examples.
  - Training time can be very long.
  - Network is like a “black box”. A human cannot look inside and easily understand the model or interpret the outputs.

➡ Recent work has sought to address these issues...

Deep Learning

- Recent developments in neural networks have led to a step change in the performance of machine learning models.
- Difference: Deeper networks, more complex architectures, huge performance improvements, many past issues have been solved.
- Gradient descent is still the core algorithm behind deep learning.



(Szegedy et al, 2015)

For more details, see COMP47650 (Deep Learning) and COMP47590 (Advanced Machine Learning) in Semester 2

References

- Nielsen, M. A. (2015). Neural networks and deep learning. Determination Press. http://neuralnetworksanddeeplearning.com
- Hassoun, M. H. (1995). Fundamentals of artificial neural networks. MIT press.
- Kröse, B., van der Smagt, P. (1993). An introduction to neural networks.
- Koehn, P. (2017). Machine Translation: Introduction to Neural Networks.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.
- Szegedy, C. et al. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition 2015.