

# Machine Learning



## Machine Learning

Lecture: Neural Networks

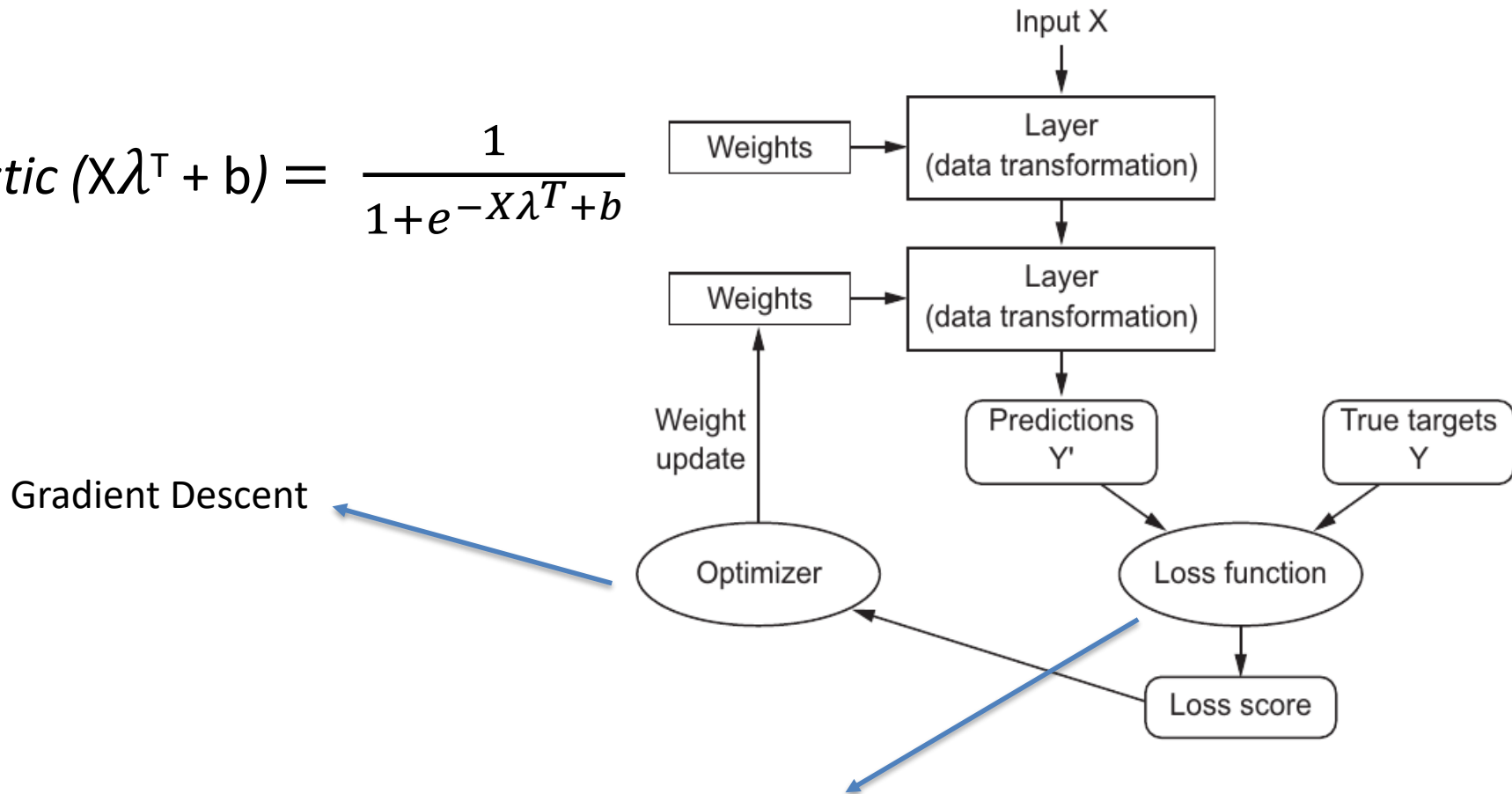
Ted Scully

# Contents

1. Introduction to Linear Regression
2. Applying Gradient Decent to Linear Regression
3. Multi-Variate Linear Regression
4. Review of Matrices and Broadcasting in Python
5. Logistic Regression
6. Vectorized Logistic Regression
7. Neural Networks

## Recap of Structure of Logistic Regression

$$\text{logistic}(X\lambda^T + b) = \frac{1}{1 + e^{-X\lambda^T + b}}$$



$$L(W) = -y \log(h(x^i)) - (1 - y) \log(1 - h(x^i))$$

$$C(W, b) = \frac{1}{m} \sum_{i=1}^m L(W)$$

# Vectorised Version of Logistic Regression

Let's assume we have a training set with  $m$  rows and  $n$  columns (features).

We create a matrix  $\mathbf{X}$  as follows, where each **column is a row from our training dataset**. For example, the first column contains all  $n$  features from the first row of our dataset.

Let's also assume that we have a vector  $W$  that contains all lambda values as shown.

Finally, let's assume we have a row vector  $Y$  that contains the actual regression value for each training instance  $i$ .

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

$$W = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix}$$

$$Y = [y^1, y^2, \dots y^m]$$

# Vectorised Version of Logistic Regression

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

$$db = \frac{1}{2m} \sum E$$

$$d\lambda = \frac{1}{2m} X E^T$$

$$W = W - (\text{alpha}) (d\lambda)$$

$$b = b - (\text{alpha}) (db)$$

$$A = W^T X + b$$

$$[\lambda_1, \lambda_2, \dots \lambda_n] \begin{bmatrix} x_1^1 & x_1^2 \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & x_n^2 \cdots & x_n^m \end{bmatrix} + b$$

# Vectorised Version of Logistic Regression

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

$$db = \frac{1}{2m} \sum E$$

$$d\lambda = \frac{1}{2m} X E^T$$

$$W = W - (\text{alpha}) (d\lambda)$$

$$b = b - (\text{alpha}) (db)$$

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

$$\frac{\partial}{\partial \lambda_2} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_2^i)$$

⋮

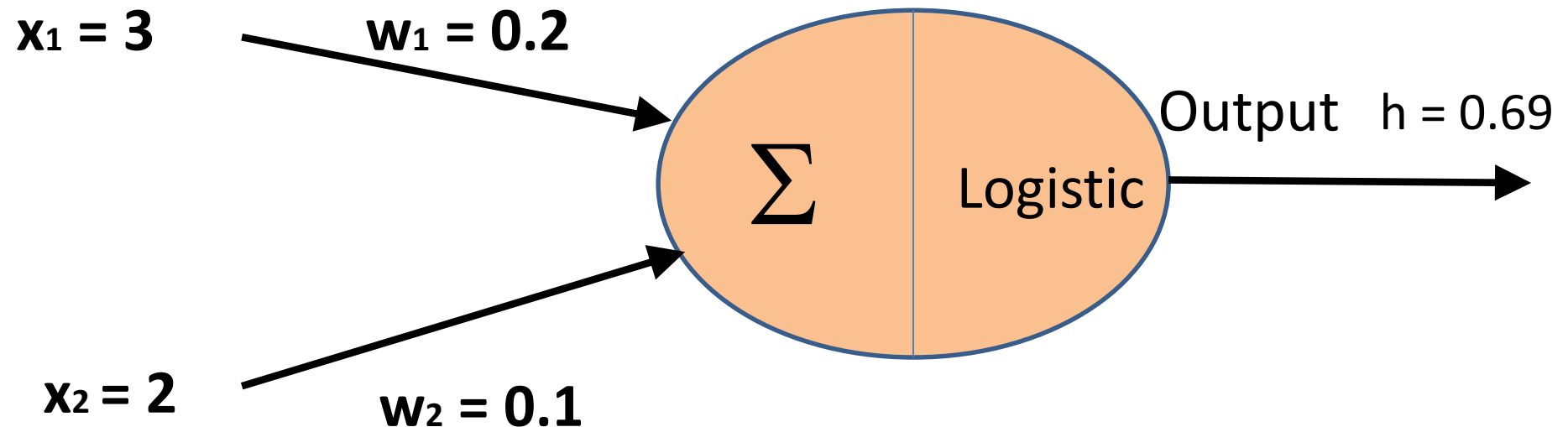
$$\frac{\partial}{\partial \lambda_n} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_n^i)$$

$$d\lambda = [d\lambda^1, d\lambda^2, \dots, d\lambda^n] = \frac{1}{2m} \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} e^1 \\ e^2 \\ \vdots \\ e^m \end{bmatrix}$$



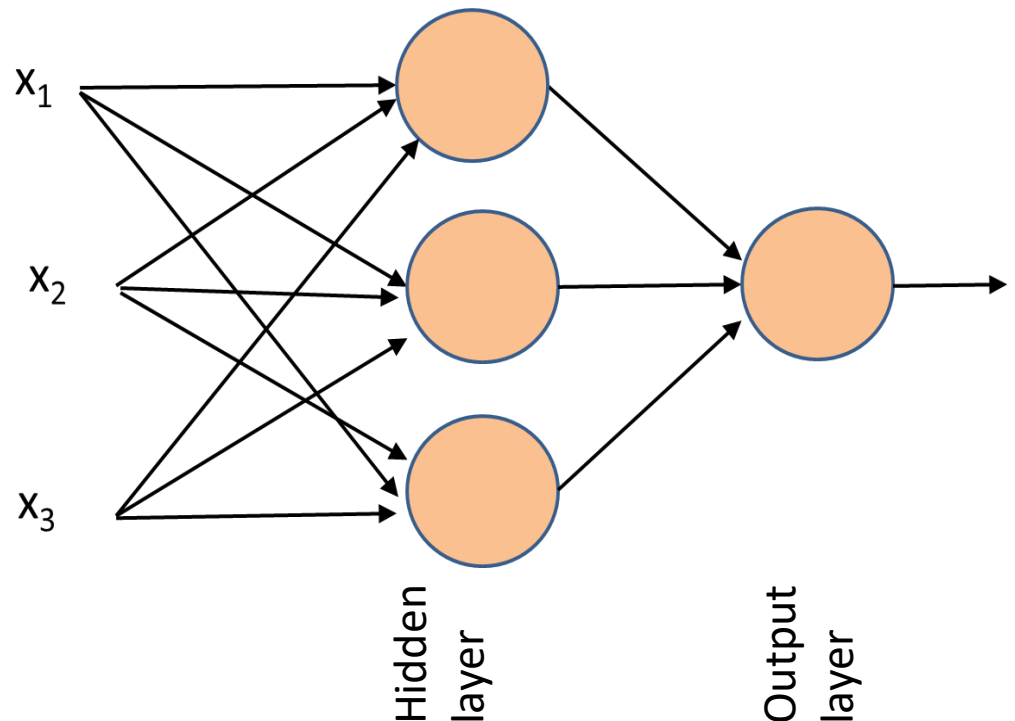
# Neural Networks – What is a Neuron?

- As we have seen we can view logistic regression as a single neuron. That is, you can view it as a composite building block of a neural network. When depicting neurons it is common not to represent the **bias** (reduce complexity of the image). However, it is still present.



# Artificial Neural Networks

- Artificial Neural Networks (ANNs) are composed of highly-interconnected neurons and are capable of altering the weights on incoming links based on training information.
- When specifying the depth of a deep neural network, we only consider the **layers that have learnable weights**
- Therefore, this example network just has two layers.
- Hidden layers are those layers with learnable weights that occur before the output layer.
- So what is the advantage of stacking all these neurons together?



# Tensor Playground



Epoch  
000,000

Learning rate

0.3

Activation

Sigmoid

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

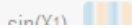
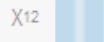


Batch size: 10



## FEATURES

Which properties do you want to feed in?



+ - 1 HIDDEN LAYER



1 neuron

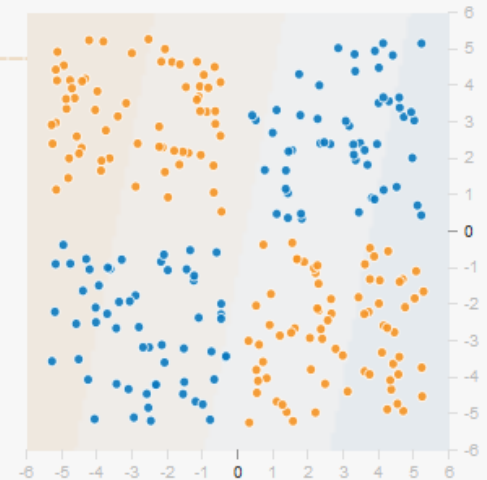


This is the output from one neuron. Hover to see it larger.

## OUTPUT

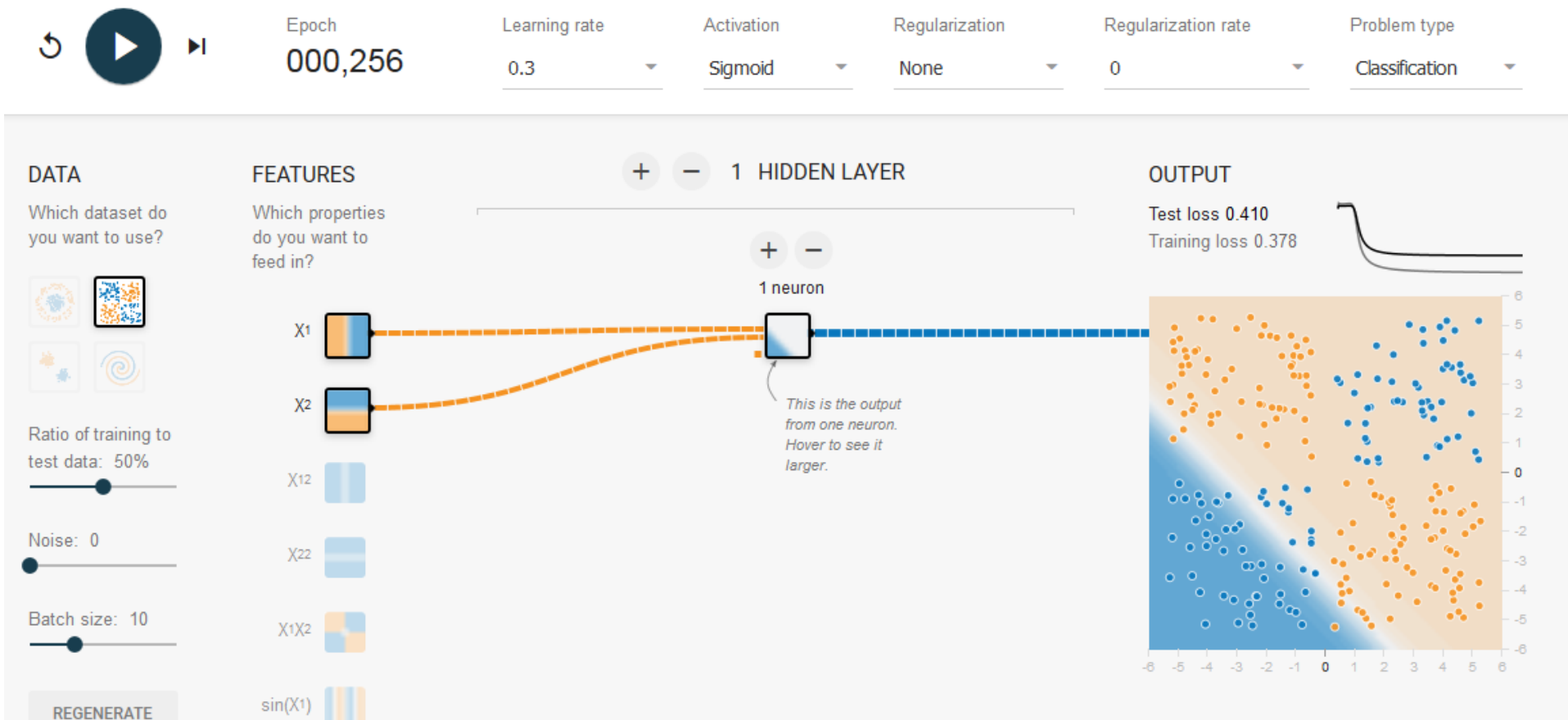
Test loss 0.500

Training loss 0.499



<https://playground.tensorflow.org>

- In the example below we train a Sigmoid classifier.





Epoch  
000,229

Learning rate

0.1

Activation

Sigmoid

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?



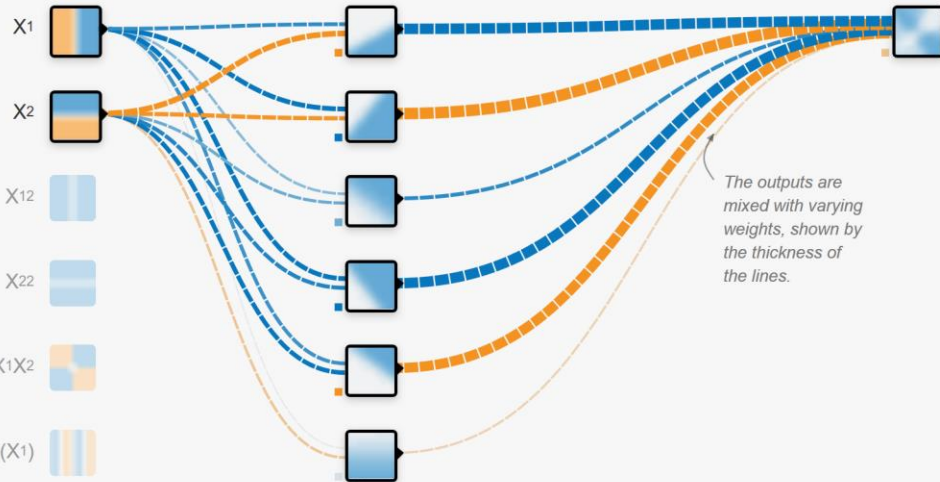
+ - 2 HIDDEN LAYERS

+ -

6 neurons

+ -

1 neuron



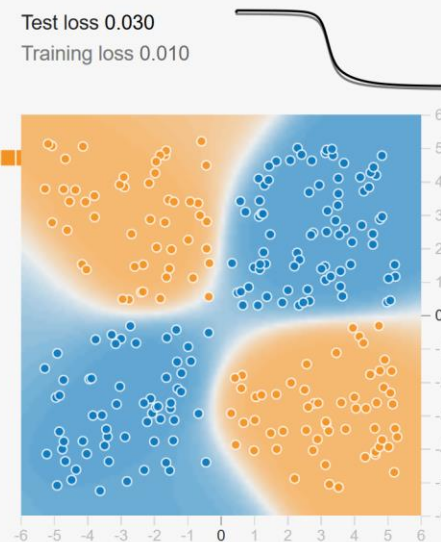
The outputs are mixed with varying weights, shown by the thickness of the lines.

This is the output from one neuron. Hover to see it larger.

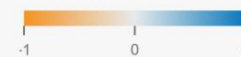
## OUTPUT

Test loss 0.030

Training loss 0.010

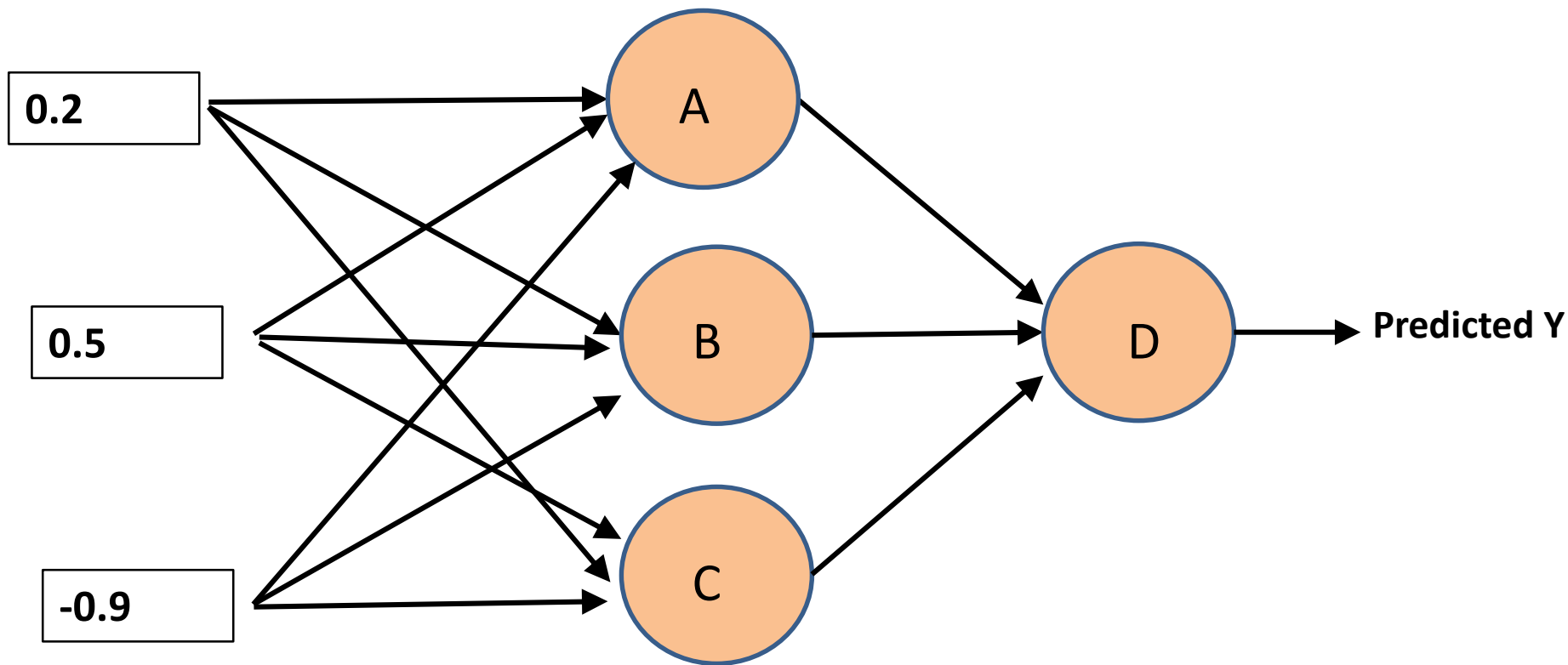


Colors shows data, neuron and weight values.



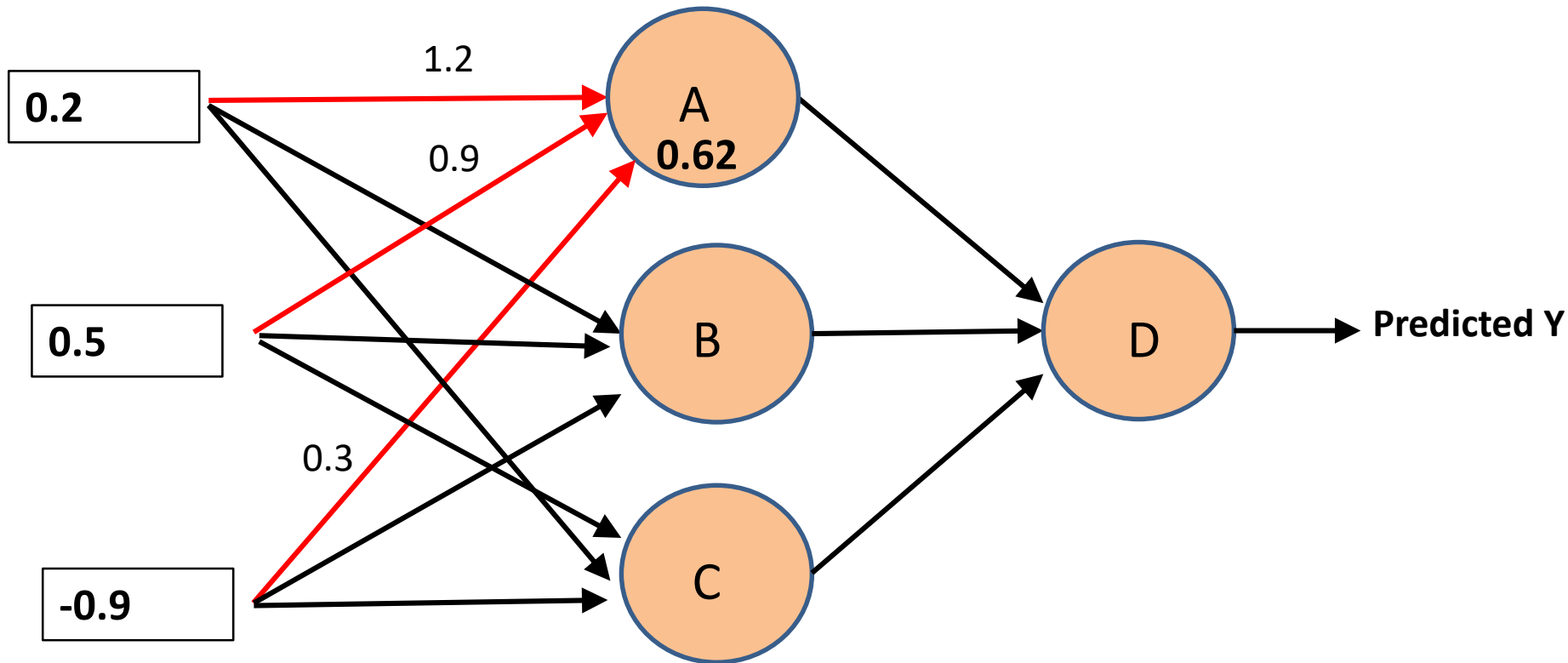
# Neural Networks

- The following is a simple illustration of what happens when we push a single feature feature  $\langle 0.2, 0.5, -0.9 \rangle$  through our neural network.
- For simplicity we will assume the **bias is just 0.1** for each of the neurons below.
- Remember each incoming connection has an associated **weight**.



# Neural Networks – Forward Pass

- The following is a simple illustration of what happens when we push a single feature feature  $\langle 0.2, 0.5, -0.9 \rangle$  through our neural network.
- For simplicity we will assume the bias is just 0.1 for each of the neurons below.

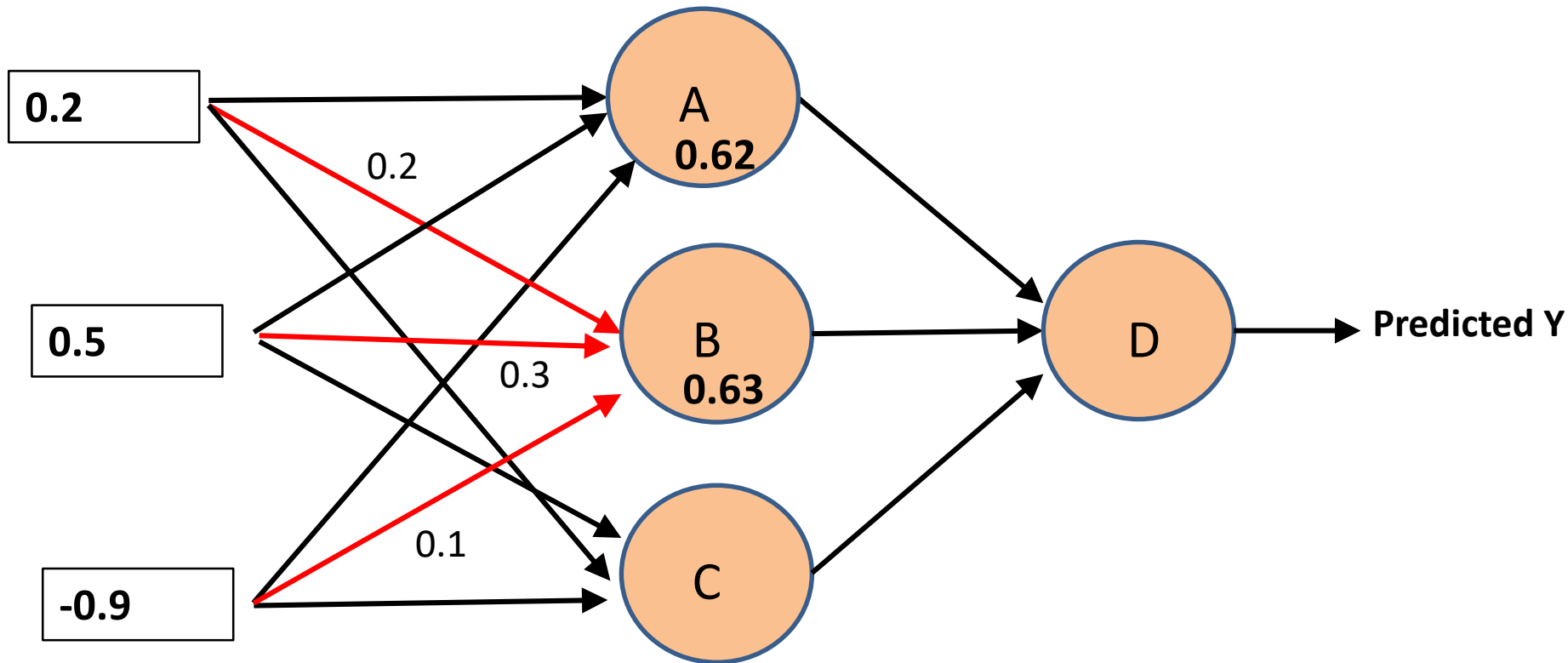


$$(0.2 * 1.2) + (0.5 * 0.9) + (-0.9 * 0.3) + 0.1 = 0.52$$

sigmoid (0.52) = 0.62

# Neural Networks – Forward Pass

- The following is a simple illustration of what happens when we push a single feature feature  $\langle 0.2, 0.5, -0.9 \rangle$  through our neural network.
- For simplicity we will assume the bias is just 0.1 for each of the neurons below.

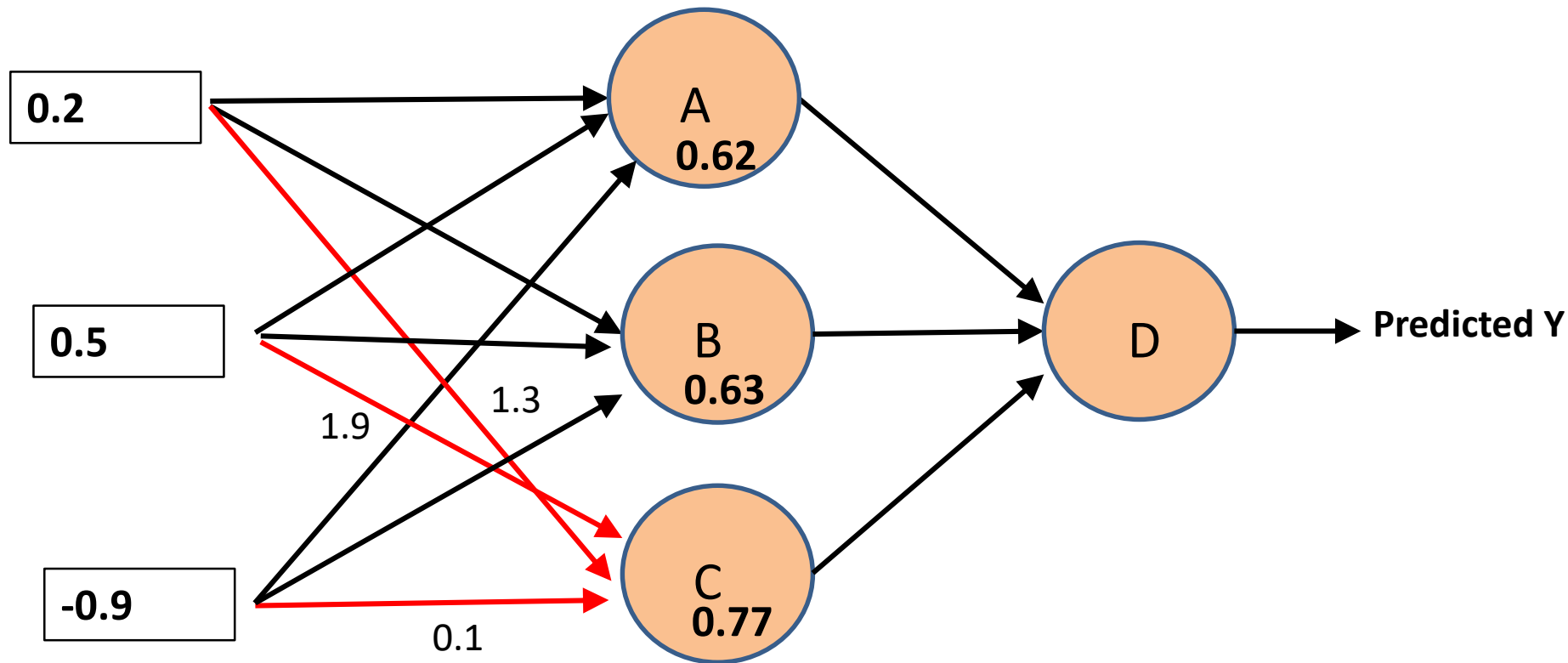


$$(0.2 \times 0.2) + (0.5 \times 0.3) + (-0.9 \times 0.1) + (0.1) = 0.55$$
$$\text{sigmoid}(0.55) = 0.63$$



# Neural Networks – Forward Pass

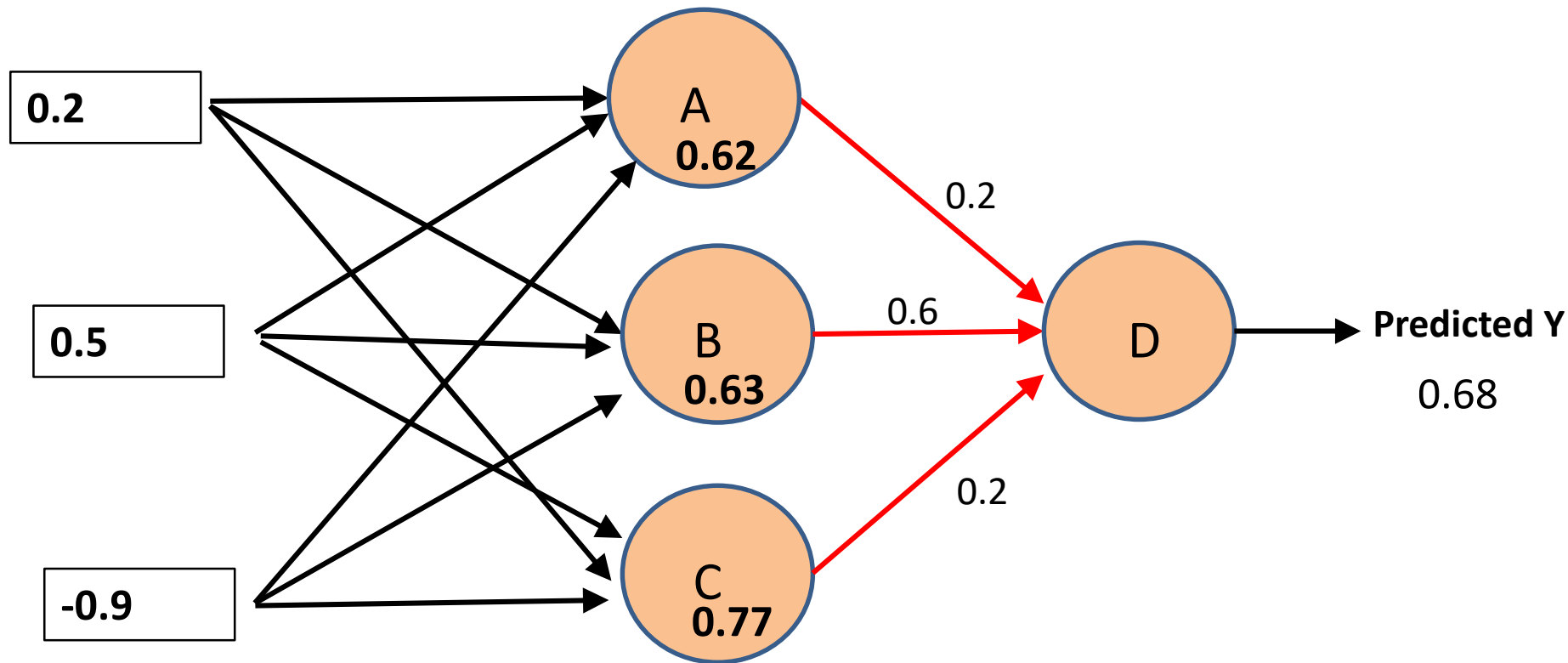
- The following is a simple illustration of what happens when we push a single feature feature  $\langle 0.2, 0.5, -0.9 \rangle$  through our neural network.
- For simplicity we will assume the bias is just 0.1 for each of the neurons below.



$$(0.2 * 1.3) + (0.5 * 1.9) + (-0.9 * 0.1) + 0.1 = 1.22$$
$$\text{sigmoid}(1.22) = 0.77$$

# Neural Networks – Forward Pass

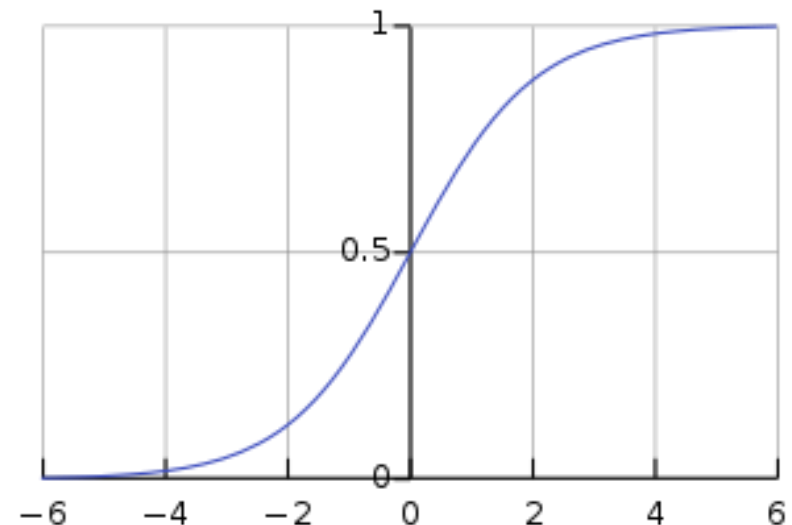
- The following is a simple illustration of what happens when we push a single feature feature  $\langle 0.2, 0.5, -0.9 \rangle$  through our neural network.



$$(0.62 \times 0.2) + (0.63 \times 0.6) + (0.77 \times 0.2) + 0.1 = 0.756$$
$$\text{sigmoid}(0.756) = 0.68$$

# Activation Functions for ANNs

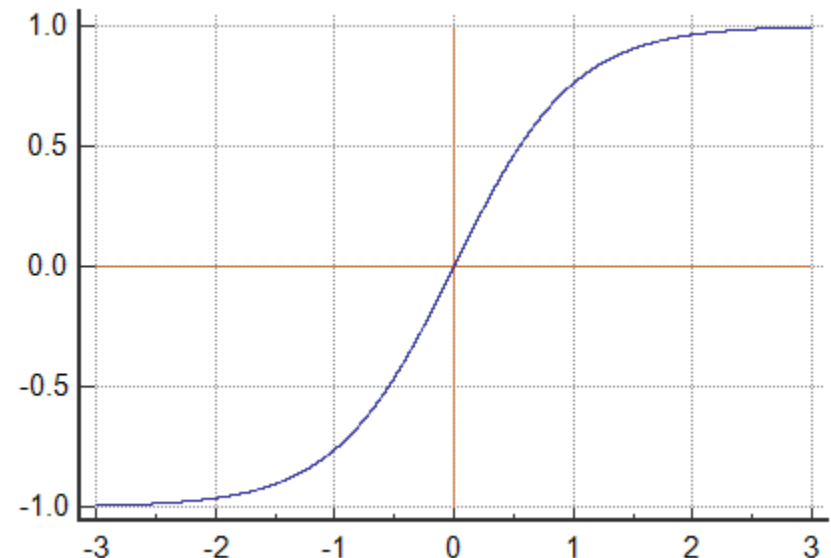
- ▶ So far we have only used a single activation function (the Sigmoid activation function).
- ▶ However, when using neural networks there are a whole range of activation functions that can be used for both the hidden layers and the output layer.
- ▶ This is an ongoing active area of research in ML.
- ▶ There are a host of alternatives to the sigmoid function:
  - ▶ Tanh function
  - ▶ ReLu function
  - ▶ Leaky ReLu function
  - ▶ Elu function
  - ▶ Selu function ... etc.



# Activation Functions - Tanh

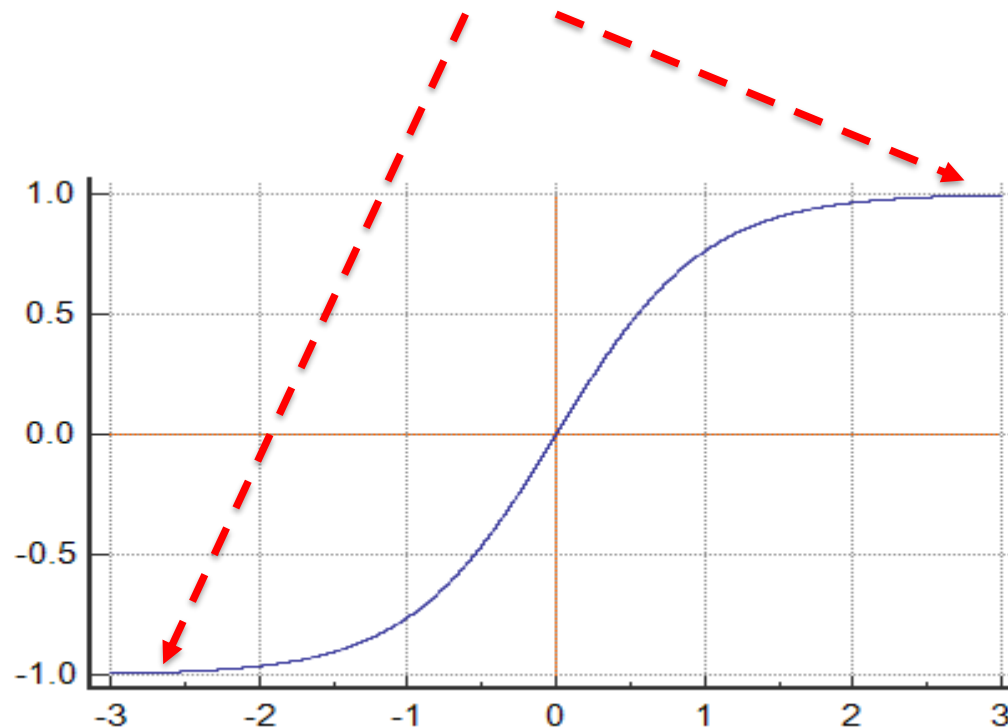
- ▶ Unlike the Sigmoid function the Tanh function goes between -1 and 1.
- ▶ Technically it is a **scaled version** of the Sigmoid curve so that it crosses the y axis at 0.
- ▶ In practice the Tanh function is more commonly used over the Sigmoid function.
- ▶ The only exception is when you want to perform binary classification (you want the output of your ANN to be either 0 or 1). In such cases it makes sense for your output layer to be a sigmoid activation function.

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



# Drawbacks of Tanh and Sigmoid

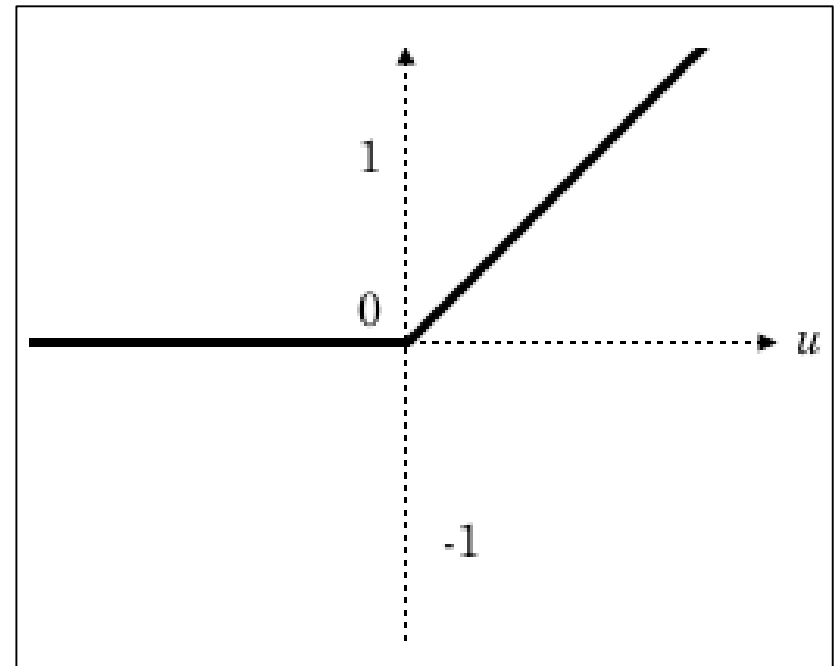
- ▶ One of the disadvantages of both the Tanh and Sigmoid function is that **if the value being inputted** is a very **large negative** or **positive** number then we end up with a gradient that is close to 0, which can slow down convergence of gradient descent very significantly.
- ▶ In deep learning slow convergence is a very significant problem.

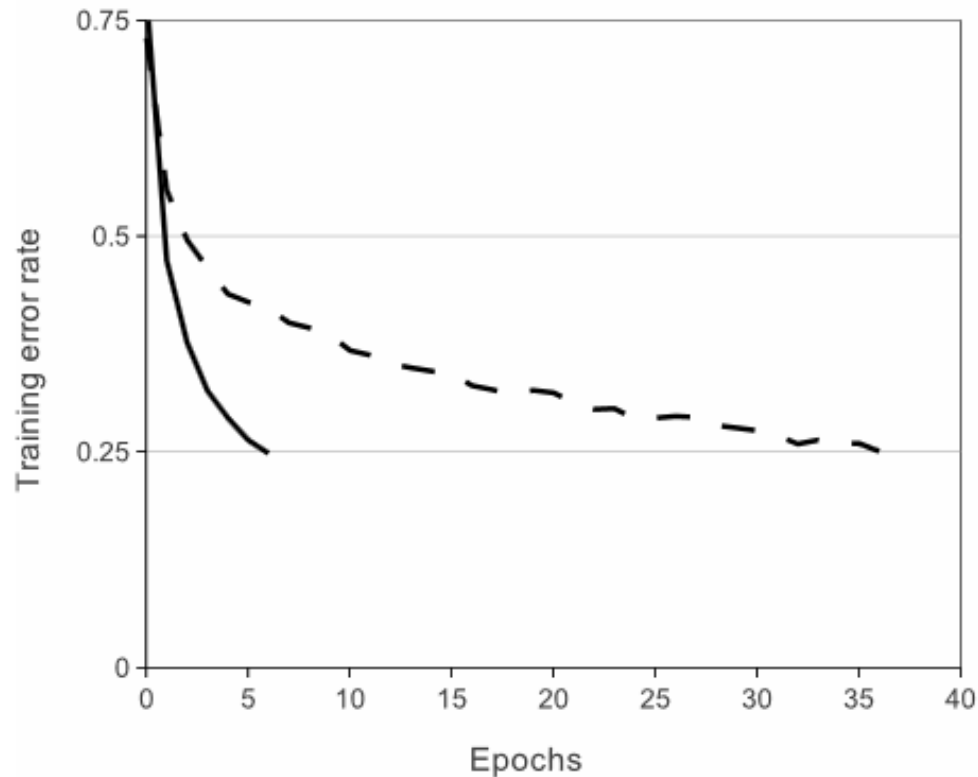


# The Rectified Linear Unit (ReLU) Function

- ▶ A very popular alternative to the sigmoid and tanh activation function is the Rectified Linear Unit (ReLU).
- ▶ It is a very simple function. The derivative of any positive number is 1, while the derivative is 0 for any negative number.
- ▶ The ReLU function is the most popular activation function used in deep neural networks and is also becoming increasingly popular in shallow neural networks.

$$\text{ReLU}(x) = \max(0, x)$$



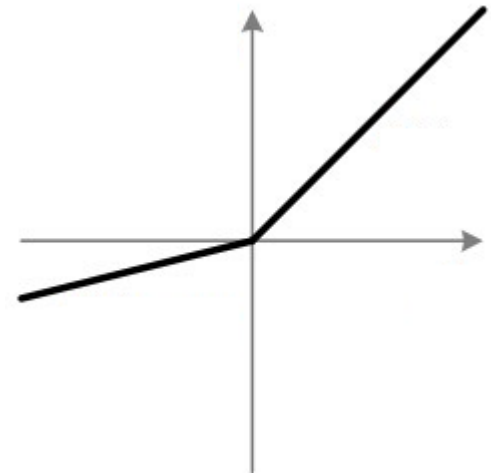


A four-layer convolutional neural network with ReLUs(solid line)reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons(dashed line).

[ImageNet Classification with Deep Convolutional Neural Networks - Alex Krizhevsky](#)

# The Leaky Rectified Linear Unit (Leaky ReLu) Function

- ▶ One disadvantage of the ReLu function is that the **derivative is 0 when the input is negative**. In practice this still works fine but there is a variant of the ReLu called the Leaky ReLu function where there is a small derivative for negative values of  $x$ .
- ▶ The main advantage of the ReLU and the leaky ReLU is that typically the ANN will converge much faster and it suffers less from the issue of very low gradients that we outlined with the sigmoid and tanh function.

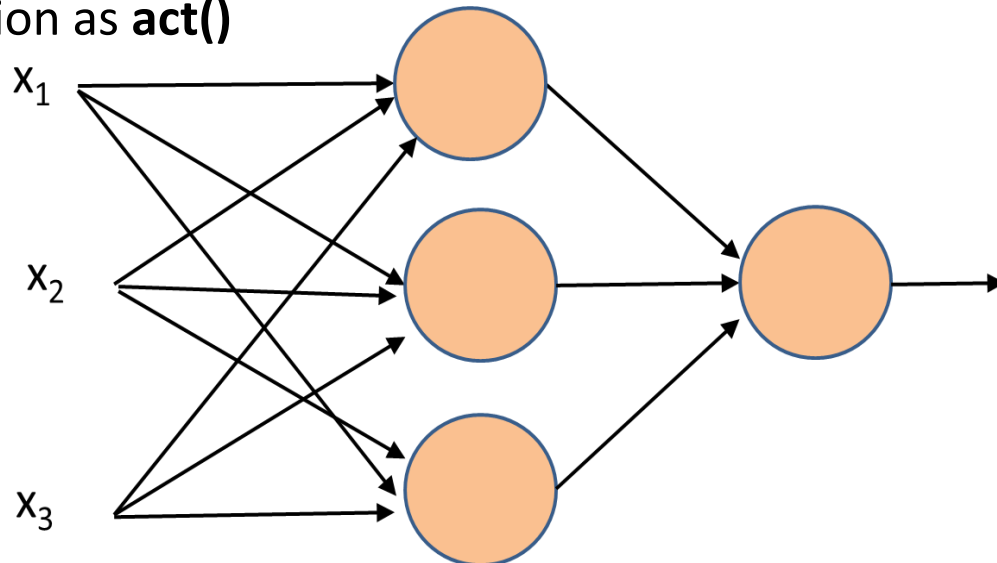


$$\text{Leakly ReLu}(x) = \max(f \cdot x, x) \text{ where } f \text{ small such as } 0.01$$

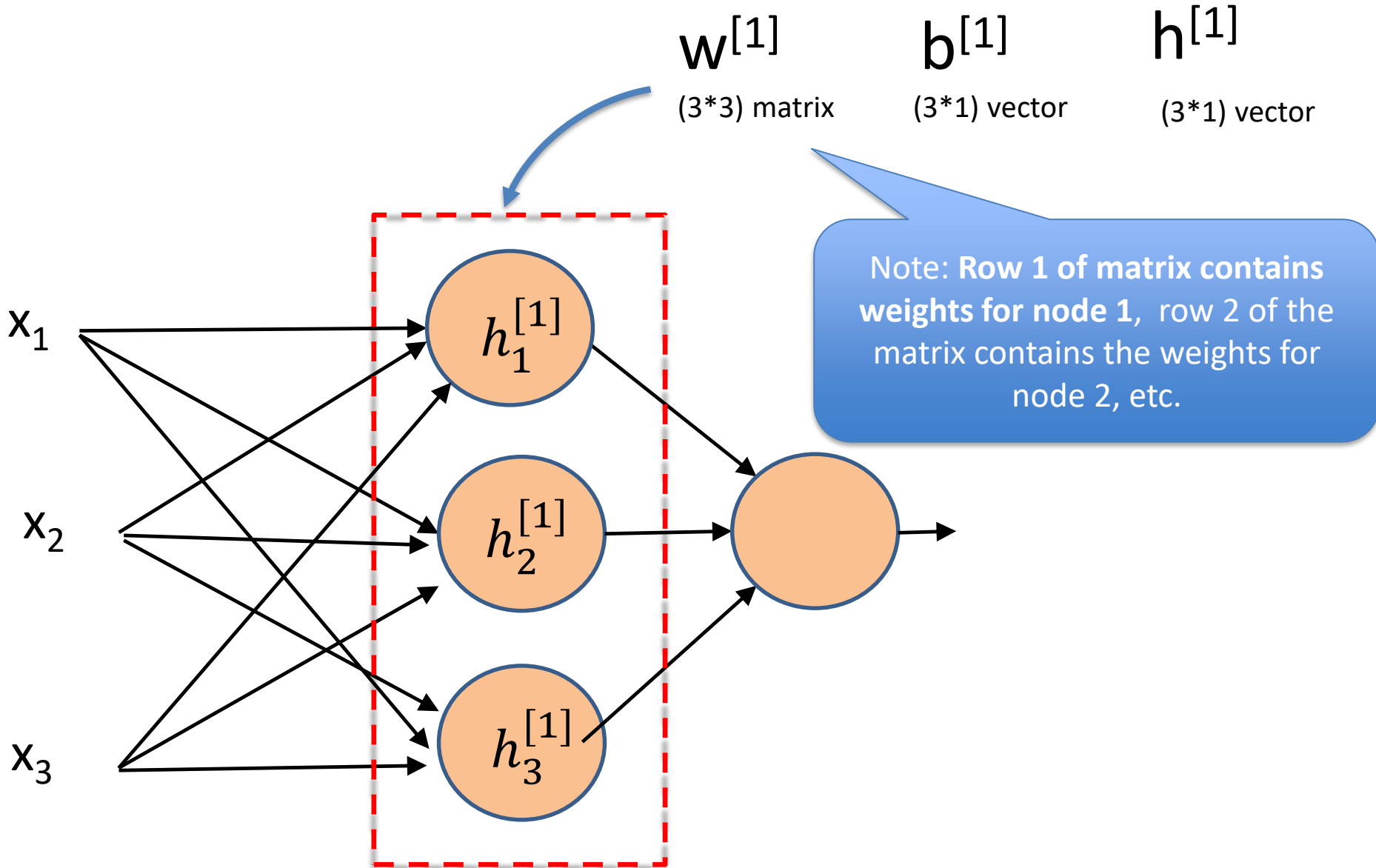


# Notation for ANNs

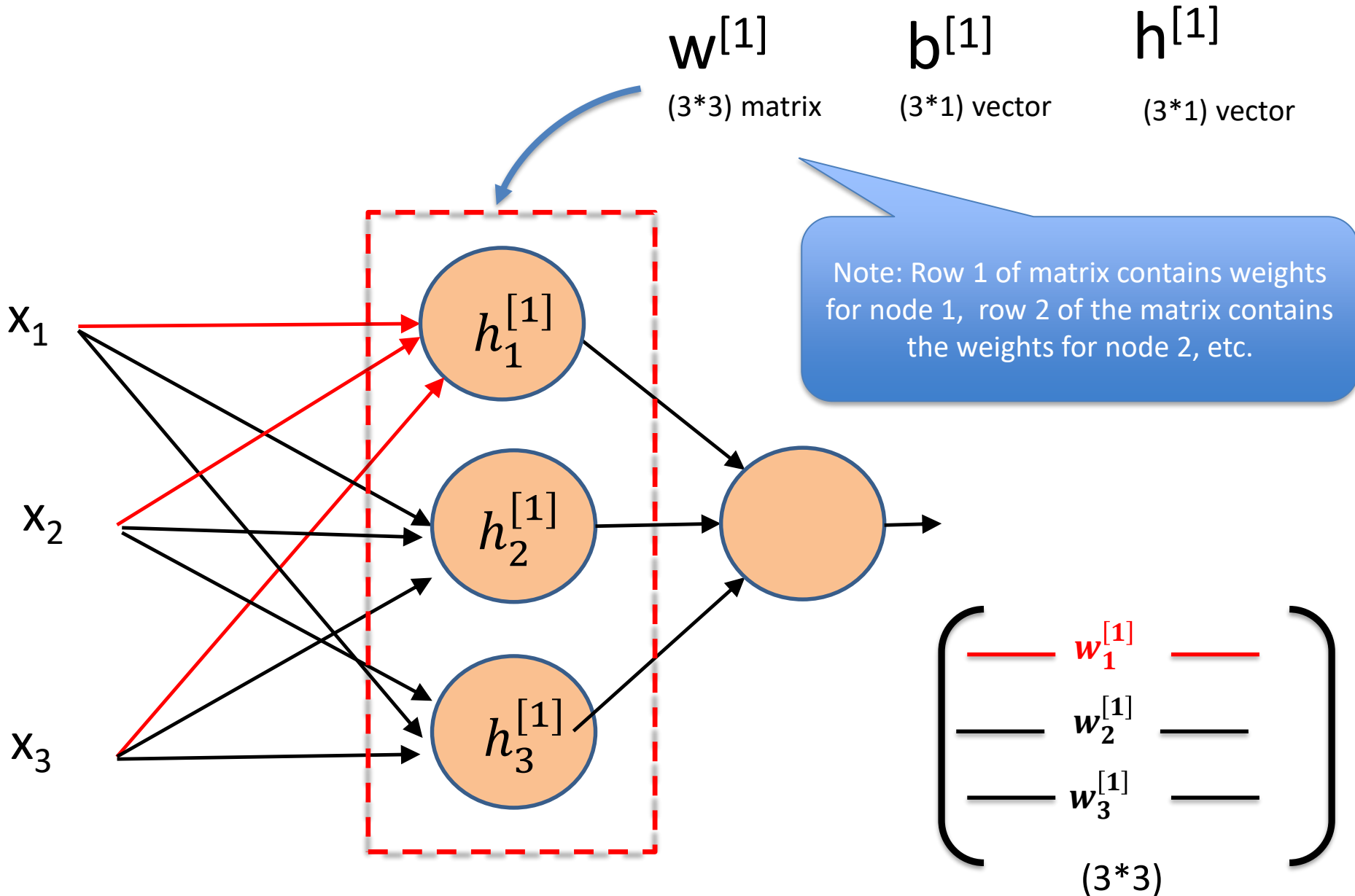
- In neural networks we have different layers with each layer made up of a number of nodes.
- We introduce the following notation:
  - $\mathbf{W}^{[i]}$  refers to all weights in layer  $i$  of the neural network (each row represent all weights incoming for one neuron)
  - $w_j^{[i]}$  refers to weights for neuron  $j$  for the weight matrix for layer  $i$
  - $\mathbf{b}^{[i]}$  refers to all bias values for layer  $i$  of the network ( $b_j^{[i]}$ ).
  - $\mathbf{h}^{[i]}$  will refer to the outputs of layer  $i$  of the network ( $h_j^{[i]}$ ).
  - Also rather than using the Sigmoid function directly we will now refer to the activation function as **act()**



# Notation for ANNs



# Notation for ANNs



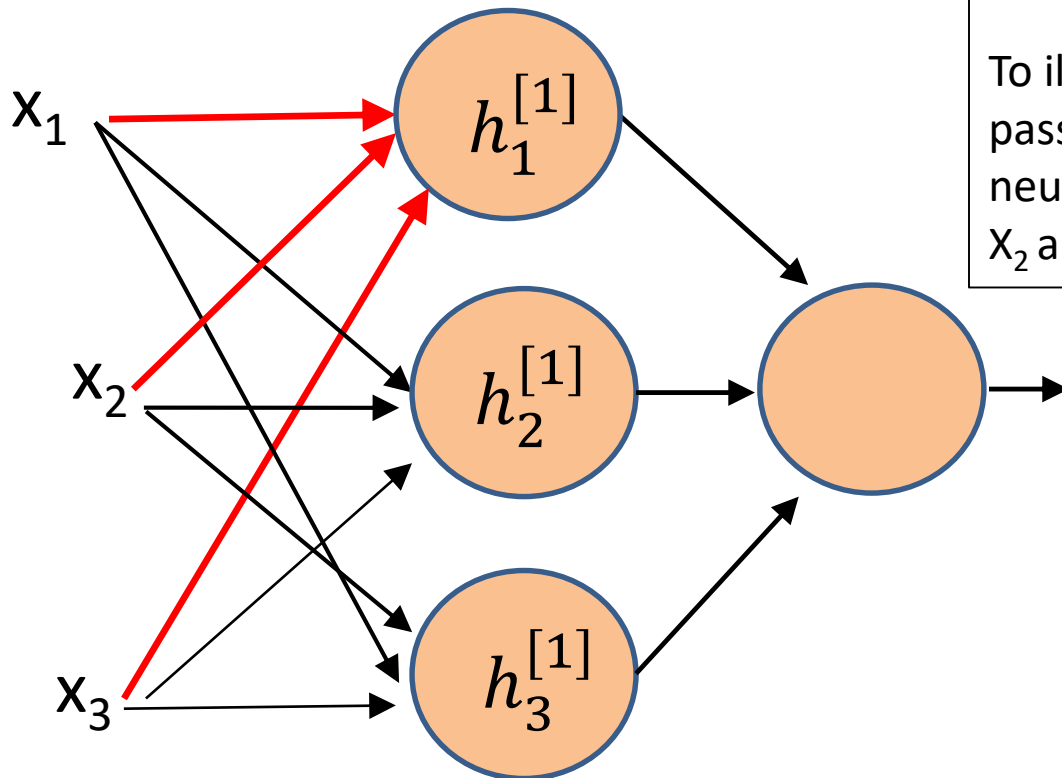
# Calculations for a Forward Pass – Single Example with One Neuron

$$a_1^{[1]} = w_1^{[1]}x + b_1^{[1]}$$
$$h_1^{[1]} = \text{act}(a_1^{[1]})$$

Note that  $w_1^{[1]}$  is a row vector containing the weights for our first node.

The value of  $x$  is a **column vector** containing the three input value  $X_1$ ,  $X_2$  and  $X_3$ .

To illustrate the operation of a forward pass for an ANN we start with just a single neurons and a single training example  $X_1$ ,  $X_2$  and  $X_3$ .

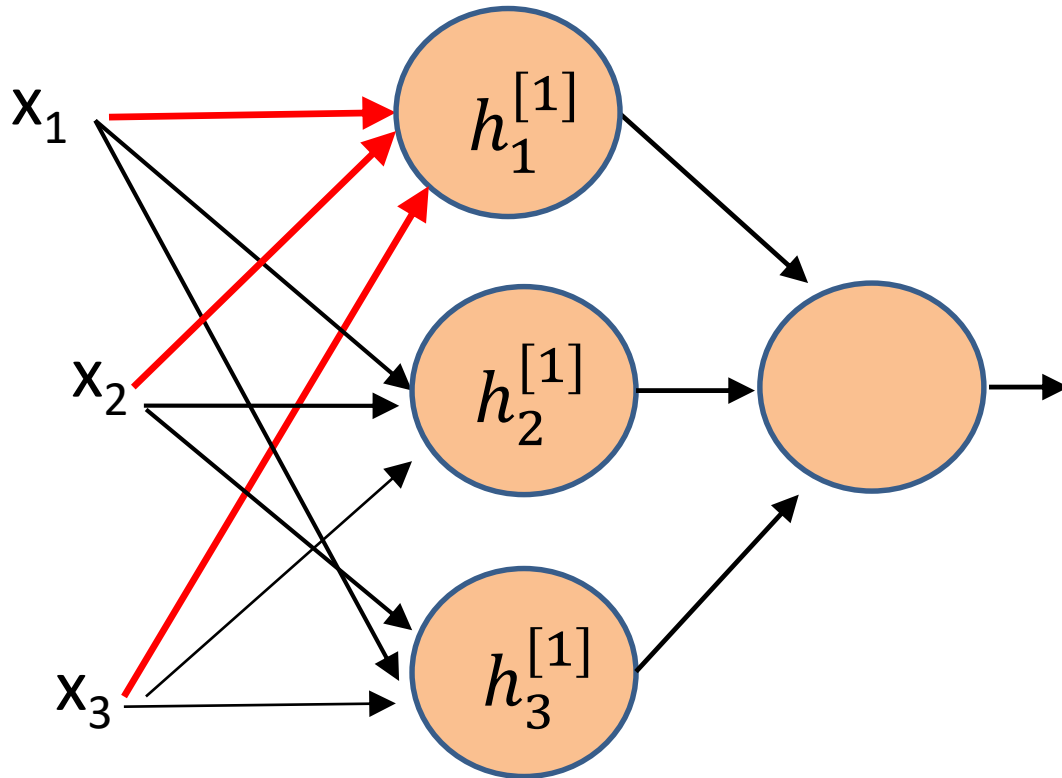


# Calculations for a Forward Pass – Single Example with One Neuron

$$\begin{aligned} a_1^{[1]} &= w_1^{[1]}x + b_1^{[1]} \\ h_1^{[1]} &= \text{act}(a_1^{[1]}) \end{aligned}$$

$$\left( \text{--- } \mathbf{w}_1^{[1]} \text{ ---} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$(1 \times 3)$   $(3 \times 1)$

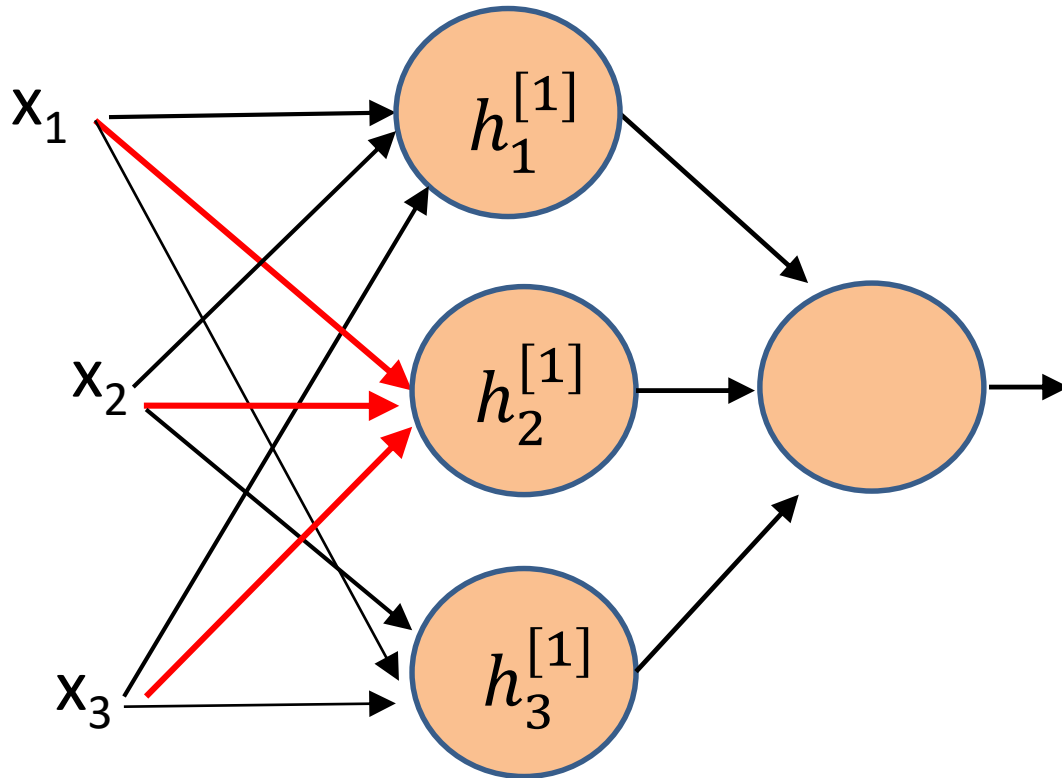


# Calculations for a Forward Pass

$$\begin{aligned} a_2^{[1]} &= w_2^{[1]}x + b_2^{[1]} \\ h_2^{[1]} &= \text{act}(a_2^{[1]}) \end{aligned}$$

$$\left( \text{--- } \mathbf{w}_2^{[1]} \text{ ---} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$(1 \times 3)$   $(3 \times 1)$

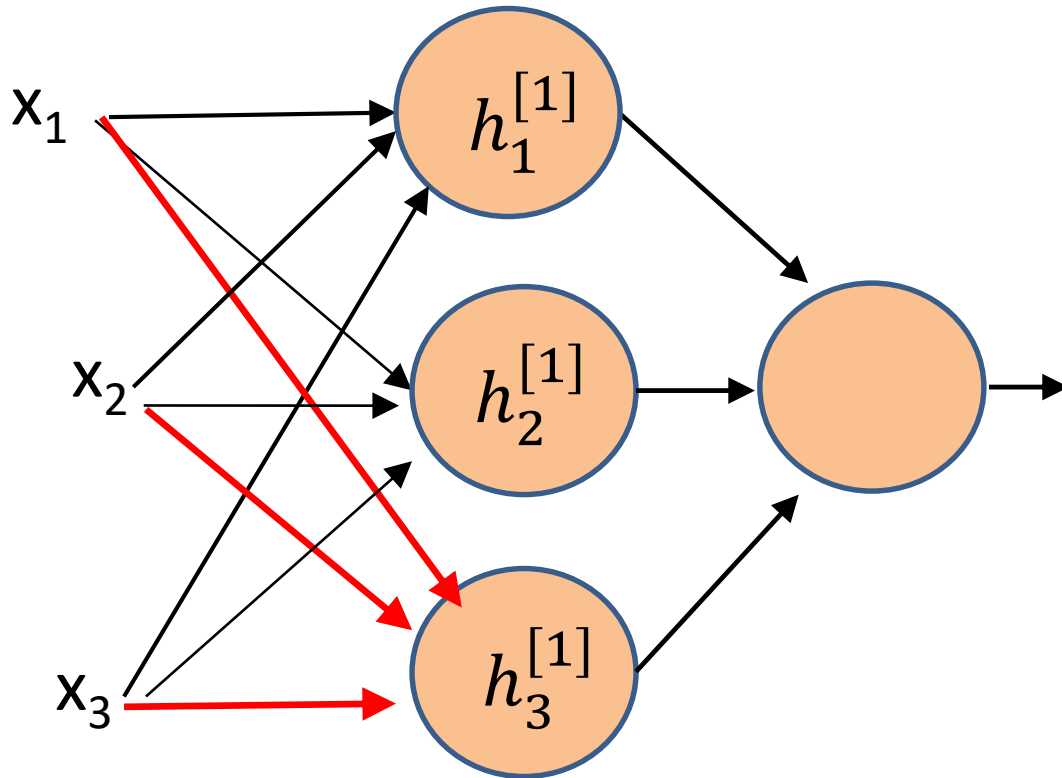


# Calculations for a Forward Pass

$$\begin{aligned} a_3^{[1]} &= w_3^{[1]}x + b_3^{[1]} \\ h_3^{[1]} &= \text{act}(a_3^{[1]}) \end{aligned}$$

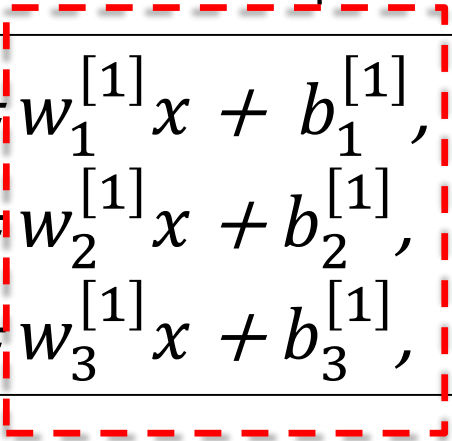
$$\left( \text{--- } \mathbf{w}_3^{[1]} \text{ ---} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$(1 \times 3)$   $(3 \times 1)$



# Calculations for a Forward Pass

While we could implement this using a **for loop**, it is much more efficient to use a vectorized implementation and perform it as one matrix multiplication.


$$\begin{array}{ll} a_1^{[1]} = w_1^{[1]}x + b_1^{[1]}, & h_1^{[1]} = \text{act}(a_1^{[1]}) \\ a_2^{[1]} = w_2^{[1]}x + b_2^{[1]}, & h_2^{[1]} = \text{act}(a_2^{[1]}) \\ a_3^{[1]} = w_3^{[1]}x + b_3^{[1]}, & h_3^{[1]} = \text{act}(a_3^{[1]}) \end{array}$$



# Calculations for a Forward Pass

While we could implement this using a **for loop**, it is much more efficient to use a vectorized implementation and perform it as one matrix multiplication.

$$\begin{array}{ll} a_1^{[1]} = w_1^{[1]}x + b_1^{[1]}, & h_1^{[1]} = \text{act}(a_1^{[1]}) \\ a_2^{[1]} = w_2^{[1]}x + b_2^{[1]}, & h_2^{[1]} = \text{act}(a_2^{[1]}) \\ a_3^{[1]} = w_3^{[1]}x + b_3^{[1]}, & h_3^{[1]} = \text{act}(a_3^{[1]}) \end{array}$$

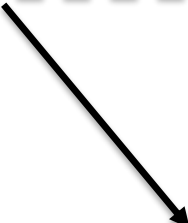
$$\begin{pmatrix} \text{---} w_1^{[1]} \text{---} \\ \text{---} w_2^{[1]} \text{---} \\ \text{---} w_3^{[1]} \text{---} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{pmatrix} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix}$$

$(3 \times 3) \quad (3 \times 1)$

# Calculations for a Forward Pass

While we could implement this using a **for loop**, it is much more efficient to use a vectorized implementation.

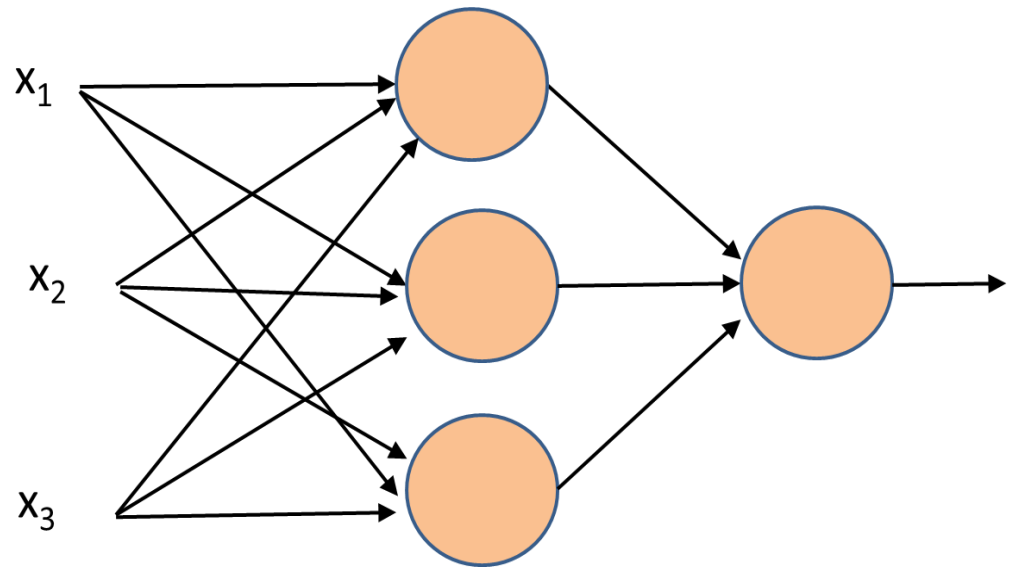
$$\begin{array}{ll} a_1^{[1]} = w_1^{[1]}x + b_1^{[1]}, & h_1^{[1]} = \text{act}(a_1^{[1]}) \\ a_2^{[1]} = w_2^{[1]}x + b_2^{[1]}, & h_2^{[1]} = \text{act}(a_2^{[1]}) \\ a_3^{[1]} = w_3^{[1]}x + b_3^{[1]}, & h_3^{[1]} = \text{act}(a_3^{[1]}) \end{array}$$


$$\text{act}\left(\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}\right)$$

# Calculations for a Forward Pass

- So we can view the operation for the first layer as shown on the right.
- Remember this is just the operation for a **single training example  $x$** .
- The vector  $\mathbf{h}^{[1]}$  is a **column vector**.

$$\mathbf{a}^{[1]} = \mathbf{w}^{[1]}x + \mathbf{b}^{[1]}$$
$$\mathbf{h}^{[1]} = \text{act}(\mathbf{a}^{[1]})$$



# Calculations for a Forward Pass

- The calculations for the second layer are shown on the right hand side.
- So the entire calculations for a forward pass in our neural network is computed as follows:

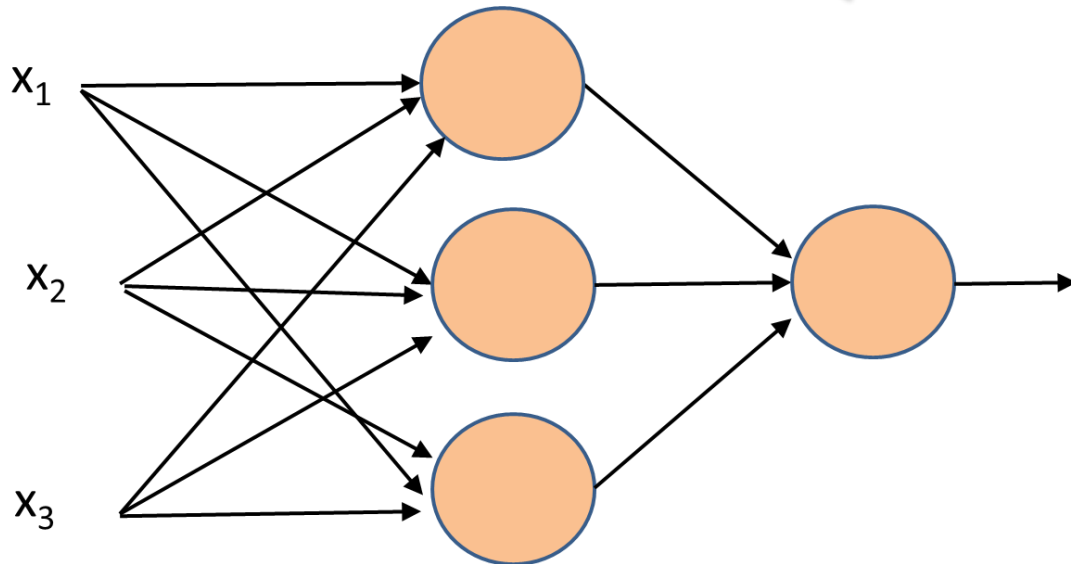
$$a^{[1]} = w^{[1]}x + b^{[1]}$$

$$h^{[1]} = \text{act}(a^{[1]})$$

$$a^{[2]} = w^{[2]} h^{[1]} + b^{[2]}$$

$$h^{[2]} = \text{act}(a^{[2]})$$

$$a^{[2]} = w^{[2]} h^{[1]} + b^{[2]}$$
$$h^{[2]} = \text{act}(a^{[2]})$$



# Forward Pass for Multiple Examples

- ▶ The previous slides depicted how to perform a forward pass in a neural network for a single training example.
- ▶ We now consider how to perform a forward pass for **multiple training examples** using a vectorised approach.
- ▶ We could just implement this using a for loop but again ensuring efficiency is very important. In the notation  $\mathbf{h}^{[2]}(i)$  refers to the prediction (outputted by layer 2 for training example  $i$ )

```
for i in range (1, m+1):
```

$$\mathbf{a}^{[1]}(i) = \mathbf{w}^{[1]}x^i + \mathbf{b}^{[1]}$$

$$\mathbf{h}^{[1]}(i) = \text{act}(\mathbf{a}^{[1]}(i))$$

$$\mathbf{a}^{[2]}(i) = \mathbf{w}^{[2]} \mathbf{h}^{[1]}(i) + \mathbf{b}^{[2]}$$

$$\mathbf{h}^{[2]}(i) = \text{act}(\mathbf{a}^{[2]}(i))$$

# Vectorized Forward Pass for Multiple Examples

- ▶ Number of training examples is  $m$
  - ▶ The number of features is  $n$
  - ▶ And  $p$  is the number of nodes in the current layer.
- 
- ▶ Remember from Logistic regression that we organize our training examples as a matrix with  $m$  columns (one for each training example) and  $n$  rows (one for each feature). First training example is the first column and so on.

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

( $n$ ,  $m$ ) matrix

# Vectorized Forward Pass for Multiple Examples

- ▶ Number of training examples is **m**
- ▶ The number of features is **n**
- ▶ The number of nodes in the current layer is **p**
- ▶ Also remember our matrix  $w^{[1]}$  contains all the weights that will be applied to each node in layer 1 of our neural network.
- ▶ The first row of this matrix contains all weights for node 1 (as node 1 is connected to each of the n features then it will contain n entries).
- ▶ The second row of the matrix contain n values that correspond to the n weights applied to the second node. The third rows contains the n weights applied to the third node and so on.
- ▶ There are p neurons, therefore, p rows.

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

(n, m) matrix

$$w = \begin{bmatrix} w_1^{[1](1)} & \cdots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \cdots & w_p^{[1](n)} \end{bmatrix}$$

(p, n) matrix

# Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$

$$H^{[1]} = \text{act}(A^{[1]})$$

$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

$$H^{[2]} = \text{act}(A^{[2]})$$

The first operation we perform in the vectorised code is to multiply the **weight matrix** by the training **data matrix**.

This is a  $(p \times n)$  matrix multiplied by a  $(n \times m)$  matrix, which gives us back a  $(p \times m)$  matrix. Notice rather than just multiplying a single example by the weights associated with each node (as we did previously) we are now multiplying all examples by the weights (vectorising the entire operation).

$$\begin{bmatrix} w_1^{[1](1)} & \cdots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \cdots & w_p^{[1](n)} \end{bmatrix} \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix} = \begin{bmatrix} t_1^1 & \cdots & t_1^m \\ t_2^1 & \cdots & t_2^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \cdots & t_p^m \end{bmatrix}$$



So the first column only relates to the **first training vector (training instance)**. It contains p values. For each of the p nodes in our network this is the result of multiplying the first training example by the input weights for that node.

The second column relates to the second training example and so on.

$$\begin{bmatrix} w_1^{[1](1)} & \dots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \dots & w_p^{[1](n)} \end{bmatrix} \begin{bmatrix} x_1^1 & \dots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \dots & x_n^m \end{bmatrix} = \begin{bmatrix} t_1^1 & \dots & t_1^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \dots & t_p^m \end{bmatrix}$$

The first row relates to the **first neuron**. It contains the output of the first neuron for every single training example.

The second row relates to the second neuron and there are p neurons altogether.

$$\begin{bmatrix} w_1^{[1](1)} & \dots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \dots & w_p^{[1](n)} \end{bmatrix} \begin{bmatrix} x_1^1 & \dots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \dots & x_n^m \end{bmatrix} = \begin{bmatrix} t_1^1 & \dots & t_1^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \dots & t_p^m \end{bmatrix}$$

# Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$

$$H^{[1]} = \text{act}(A^{[1]})$$

$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

$$H^{[2]} = \text{act}(A^{[2]})$$

As normal we then add the bias. Finally, we obtain the output for each node in layer 1 for each training example, a  $(p * m)$  matrix (p is the number of nodes and m is the number of training examples).

$$\begin{bmatrix} t_1^1 & \cdots & t_1^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \cdots & t_p^m \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_p^{[1]} \end{bmatrix} = \begin{bmatrix} a_1^1 & \cdots & a_1^m \\ \vdots & \ddots & \vdots \\ a_p^1 & \cdots & a_p^m \end{bmatrix}$$

$$\text{act}\left(\begin{bmatrix} a_1^1 & \cdots & a_1^m \\ \vdots & \ddots & \vdots \\ a_p^1 & \cdots & a_p^m \end{bmatrix}\right) = \begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix}$$

# Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$

$$H^{[1]} = \text{act}(A^{[1]})$$

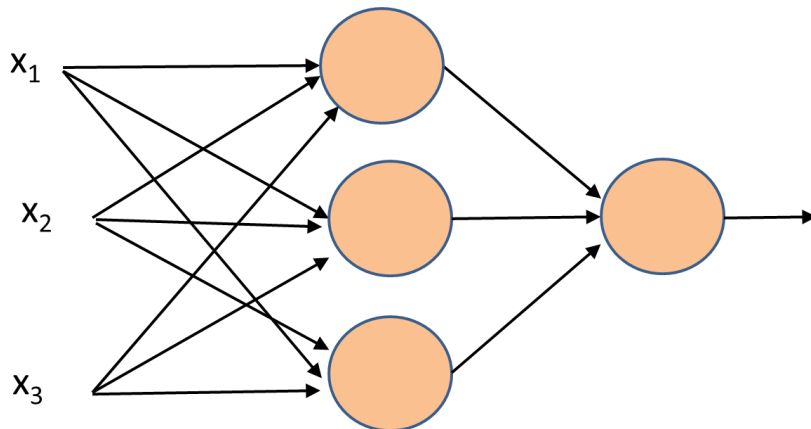
$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

$$H^{[2]} = \text{act}(A^{[2]})$$

Now let's focus on the forward pass operations for the next layer of neurons.

We take the matrix output of the first layer and multiply it by the weights for the second layer.

$$\begin{bmatrix} w_1^{[2](1)} & \dots & w_1^{[2](p)} \end{bmatrix} \begin{bmatrix} h_1^1 & \dots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \dots & h_p^m \end{bmatrix}$$



Notice there are  $p$  weights in our weights matrix for the 2<sup>nd</sup> layer of the network. Each neuron in the first layer (of which there are  $p$ ) is connected to the single neuron in the second layer.

# Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$

$$H^{[1]} = \text{act}(A^{[1]})$$

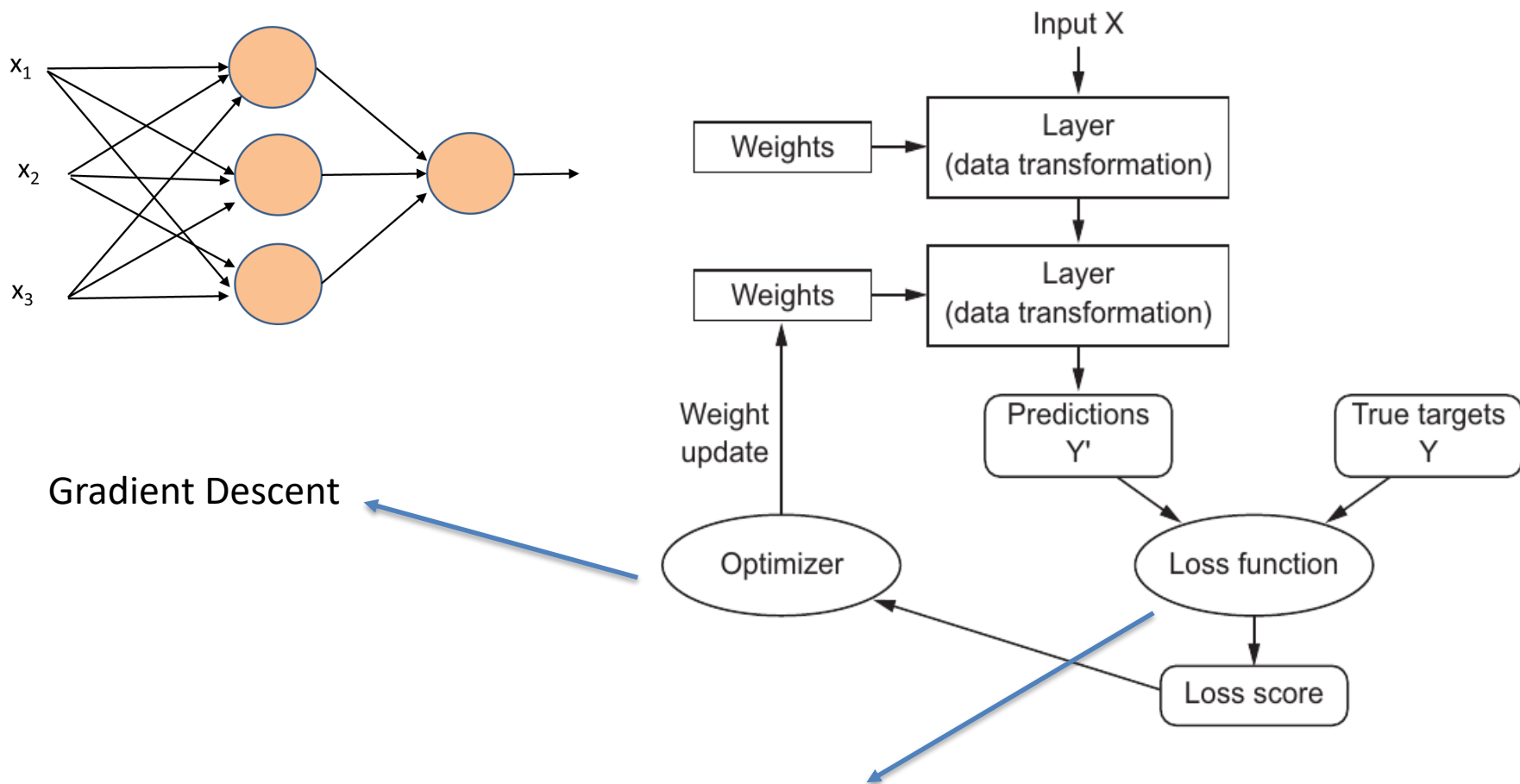
$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

$$H^{[2]} = \text{act}(A^{[2]})$$

$$\begin{bmatrix} w_1^{[2](1)} & \cdots & w_1^{[2](p)} \end{bmatrix} \begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix} = [a_1^1 \cdots a_1^m]$$

$$\text{act}\left( [a_1^1 \cdots a_1^m] + [b_1^{[2]}] \right) = [h_1^1 \cdots h_1^m]$$

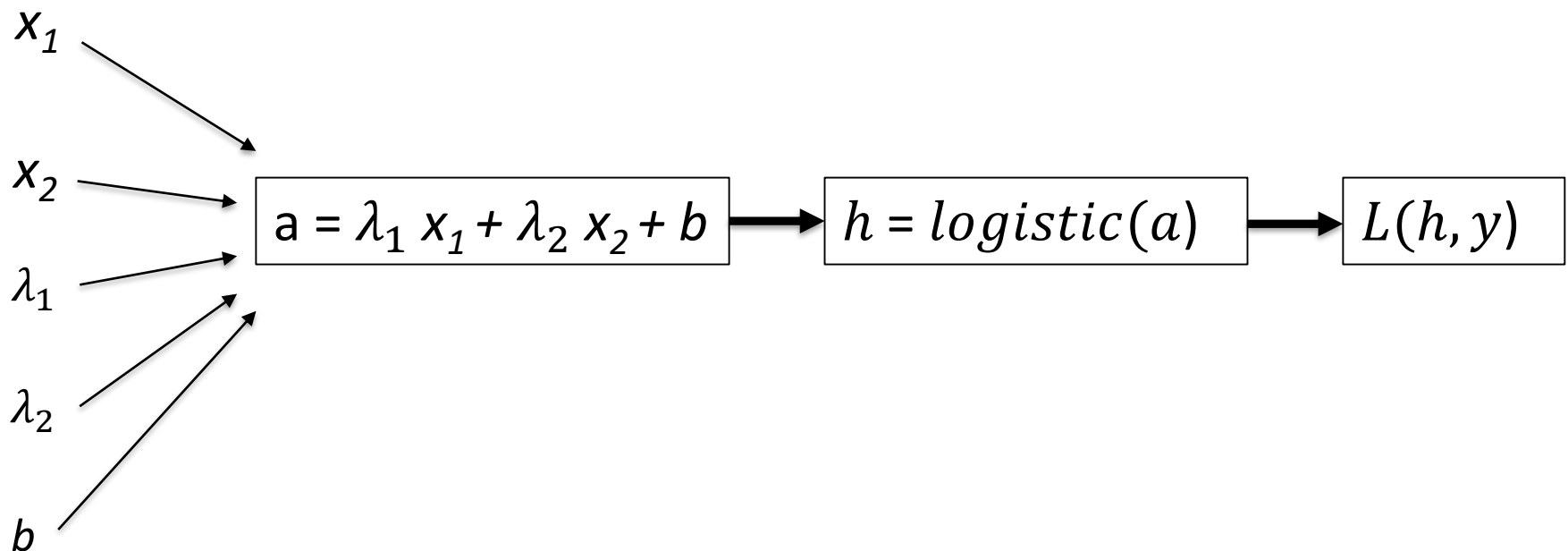
Here we assume that the model is build as a binary classifier (if this is the case we can use the cross entropy error, just as with logistic regression. )

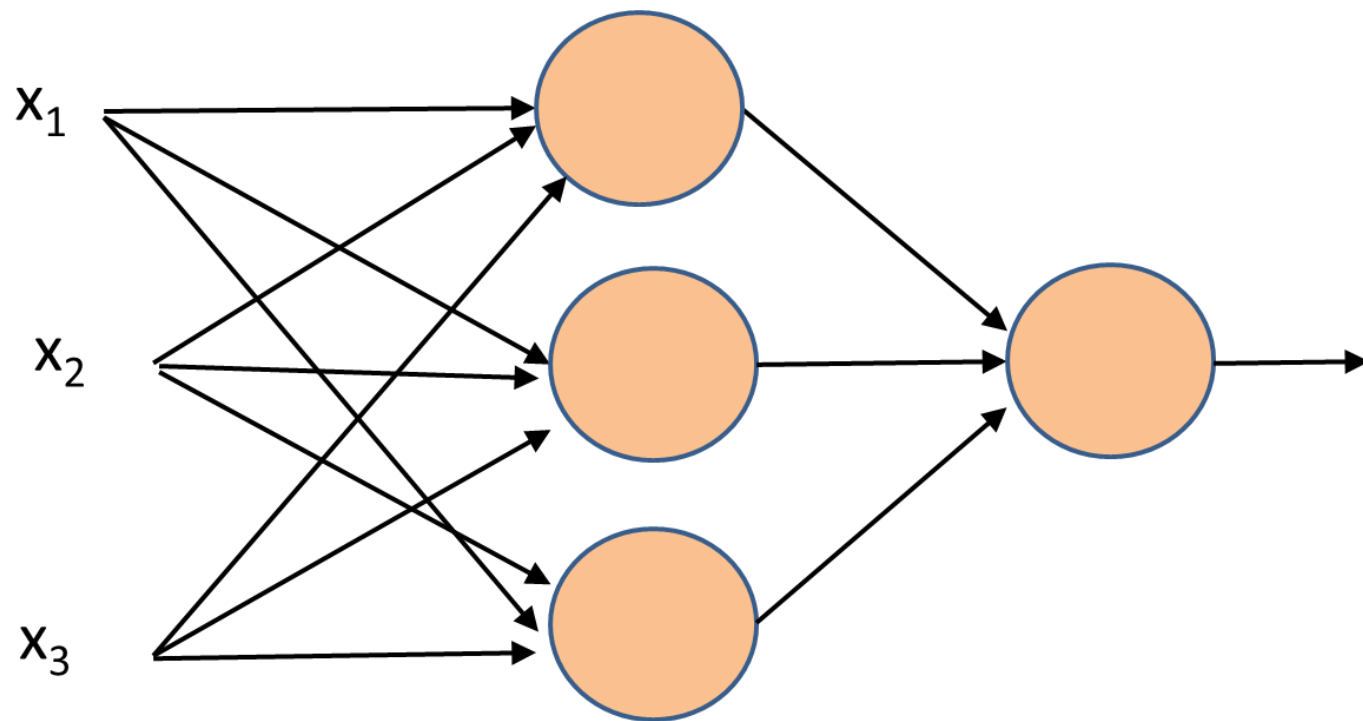


$$\mathcal{L}(W) = -y \log(h(x^i)) - (1 - y) \log(1 - h(x^i))$$

$$\mathcal{C}(W, b) = \frac{1}{m} \sum_{i=1}^m L(W)$$

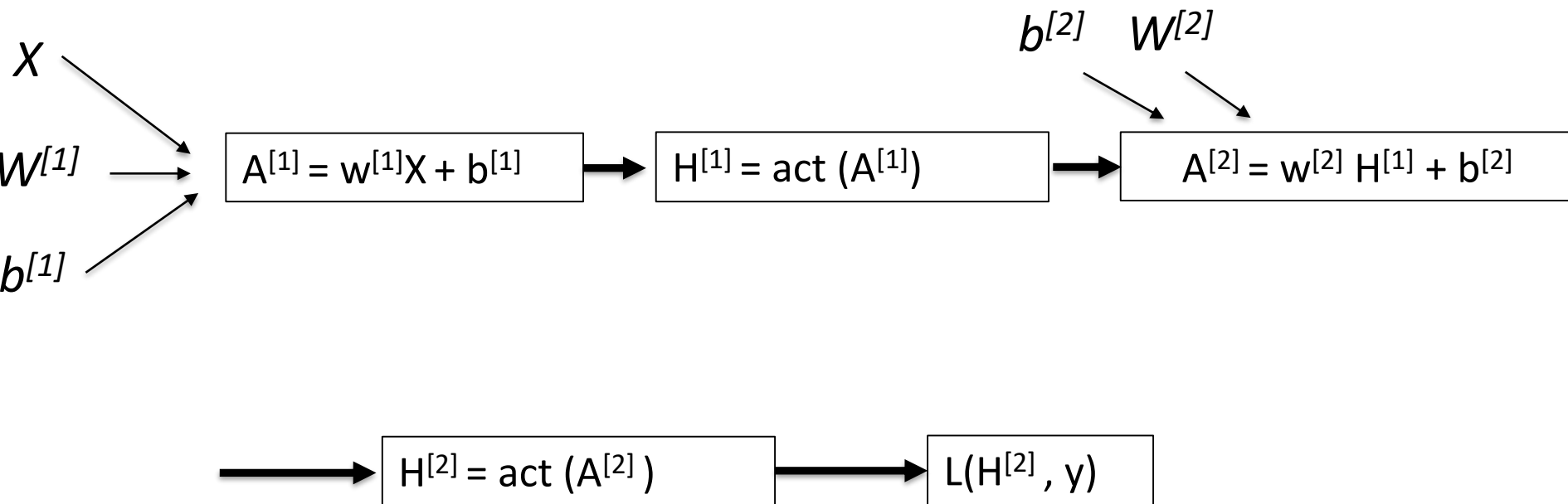
In the example below we deal with a dataset that has just **two features**. Remember the objective is that we want to modify our parameters  $\lambda_1$ ,  $\lambda_2$ ,  $b$  in order to reduce our loss function. In order to do that we need to move backward through the graph in order to calculate the derivatives.







We can view our neural network as a computation graph. As we did with logistic regression we begin to work backward through the graph in order to determine each of the derivatives we need in order to obtain the new bias and weights.



# Gradient Descent For A Neural Network (Back Propagation)

- ▶ Repeat
  - ▶ Compute the predictions  $H^{[2]}$  (Contains one prediction for each training example).
  - ▶ Calculate derivatives ( $dW^{[2]}$  ,  $db^{[2]}$  ,  $dW^{[1]}$  ,  $db^{[1]}$  )

$$W^{[1]} = W^{[1]} - \textit{alpha} * dW^{[1]}$$

$$b^{[1]} = b^{[1]} - \textit{alpha} * db^{[1]}$$

$$W^{[2]} = W^{[2]} - \textit{alpha} * dW^{[2]}$$

$$b^{[2]} = b^{[2]} - \textit{alpha} * db^{[2]}$$

# Vectorized Back Propagation

Once we have determined the output of the neural network given the current parameters (bias and weight) values we must now work backwards (in the same manner as we did with logistic regression to get the derivatives for each of the parameters). These derivatives are then used to update the parameter values in the usual way. Note the code shown below uses a logistic activation unit.

$$dA^{[2]} = H^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m}(dA^{[2]} H^{[1]T})$$

$$db^{[2]} = \frac{1}{m}(\text{np.sum}(dA^{[2]}, \text{axis} = 1))$$

$$dA^{[1]} = W^{[2]T} dA^{[2]} * (1 - (H^{[1]})^2)$$

$$dW^{[1]} = \frac{1}{m}(dA^{[1]} X^T)$$

$$db^{[1]} = \frac{1}{m}\text{np.sum}(dA^{[1]}, \text{axis} = 1)$$