

Machine Learning



Machine Learning

Lecture: Neural Networks

Ted Scully

Recap - 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).

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

Recap - 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).

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

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

Recap - Vectorized Forward Pass for ANNs

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

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

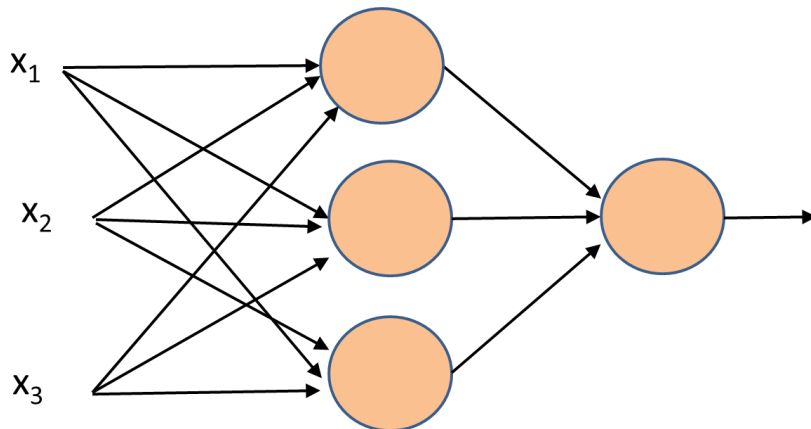
$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

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

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 2nd 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.

Recap - Vectorized Forward Pass for ANNs

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

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

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

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

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

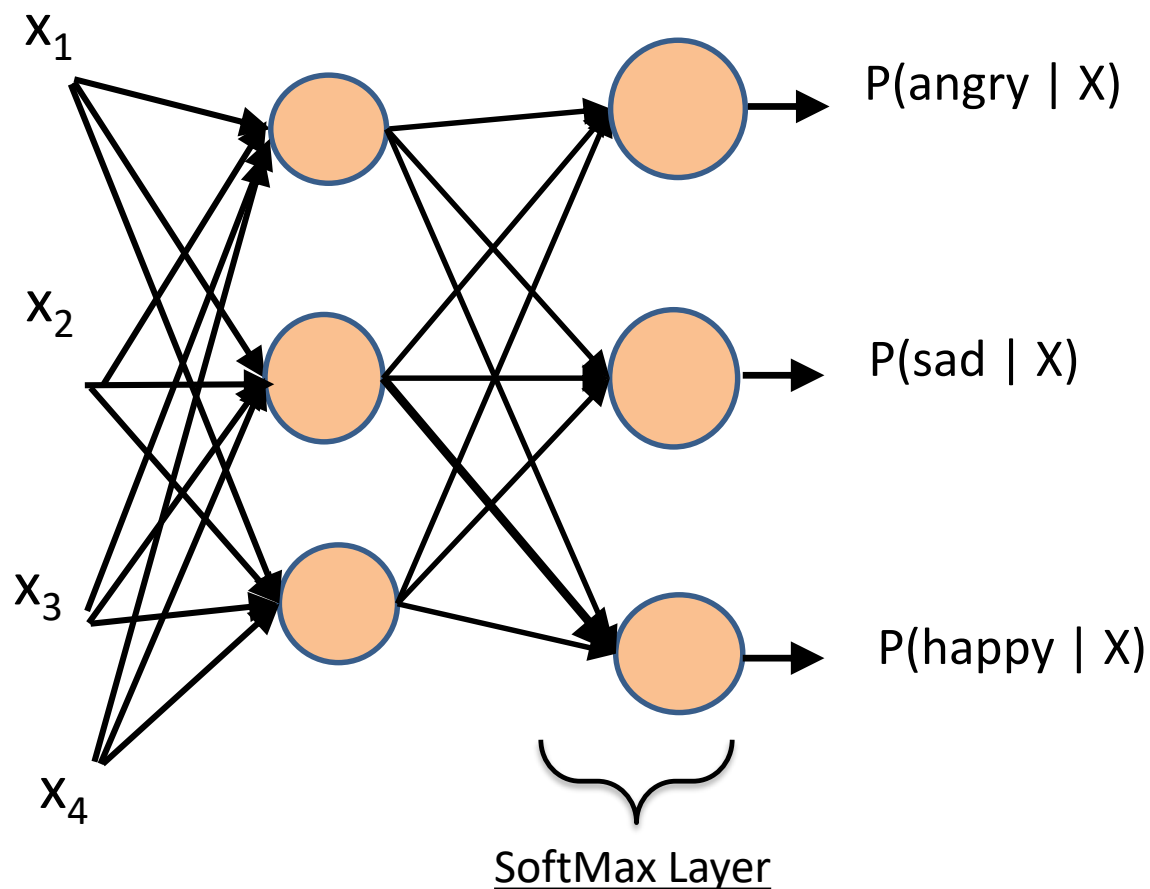
$$\text{act}\left(\begin{bmatrix} a_1^1 & \cdots & a_1^m \end{bmatrix} + \begin{bmatrix} b_1^{[2]} \end{bmatrix} \right) = \begin{bmatrix} h_1^1 & \cdots & h_1^m \end{bmatrix}$$

Building Neural Networks - Softmax Activation Layer

- ▶ All the examples we have looked at so far were concerned with binary classification and as such we were using the Sigmoid activation function in the output layer.
- ▶ However, the Softmax function is a **generalization of the logistic function** that allows us to perform multi-class classification.
- ▶ Let's assume that we want to perform classification for three different classes. Such as classify an image of a person as sad, happy or angry.
- ▶ The softmax function is typically used in the final layer of a neural network-based classifier. **It is a way of forcing the outputs of a neural network to sum to 1, which represents a probability distribution across multiple classes.**

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

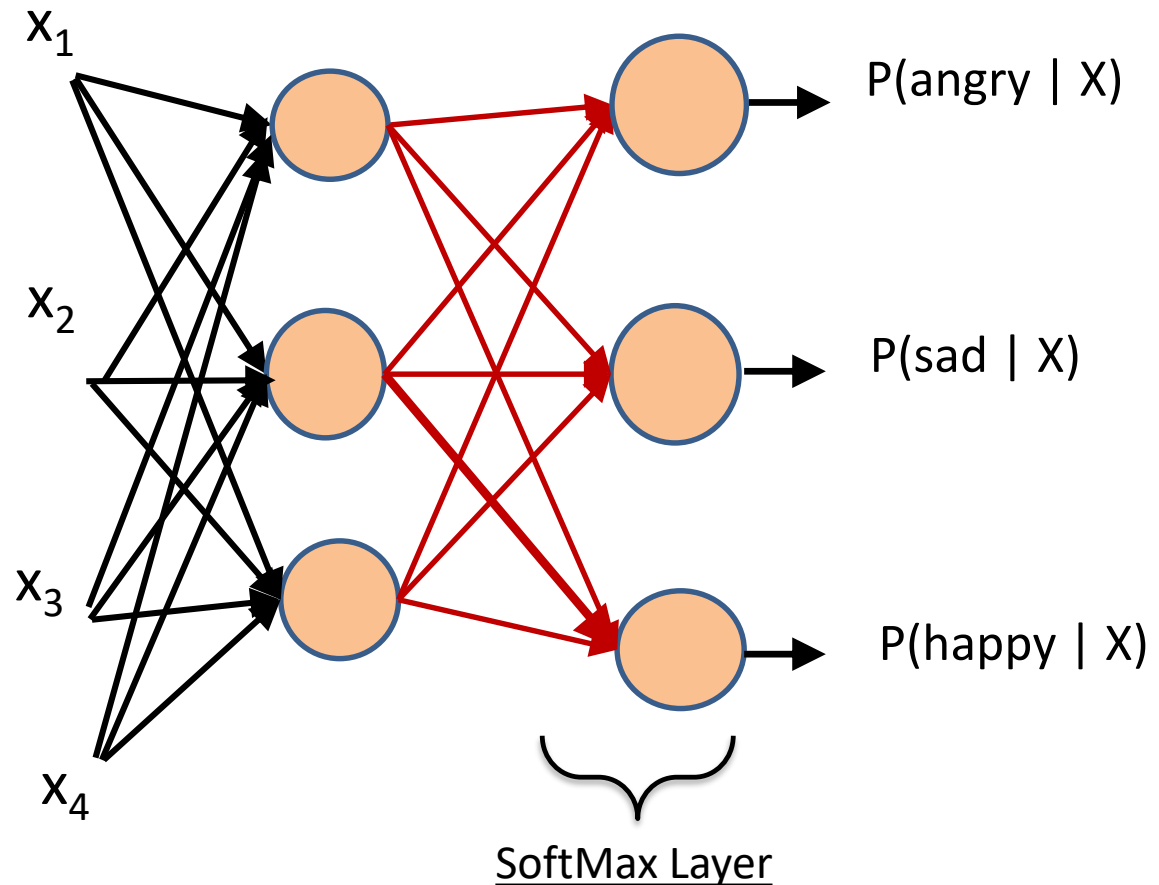
Notice our neural network here has 3 output units (there may be multiple hidden units but in this example we just include one for illustration).
If we feed our model a training example (an image of a person) we want each neuron to output the **probability of this image belonging to a specific class**. We can achieve this using a SoftMax layer.



Let's assume we have pushed a single training example through the first layer. We will now focus on what happens after this stage. As per normal we multiply the weights of the second layer ($W^{[2]}$) by the incoming values ($H^{[1]}$) and add the bias ($b^{[2]}$).

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

$A^{[2]}$ is the pre-activation value of each neuron in the Softmax layer.
The next step is to apply the activation function.



The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t , we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t .

The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t , we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t .

Assume $A^{[2]}$ is the following 3 element vector $A^{[2]} = [6, -2, 3]$.

1. Calculate element-wise exponentiation .

$$t = [e^6, e^{-2}, e^3]$$

$$t = [403.4, 0.135, 20.0]$$

The activation in a softmax layer can be described in two steps.

1. We calculate a new vector t , we calculate e to the power of the pre-activation outputs.

$$t = e^{A^{[2]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[2]} = \frac{e^{A^{[2]}}}{\sum t} = \frac{t}{\sum t}$$

Notice that the output of the Softmax is just t normalized by dividing by the sum of t .

Assume $A^{[2]}$ is the following 3 element vector $A^{[2]} = [6, -2, 3]$.

1. Calculate element-wise exponentiation .

$$t = [e^6, e^{-2}, e^3]$$

$$t = [403.4, 0.135, 20.0]$$

2. We calculate a new vector t , we calculate e to the power of the pre-activation outputs.

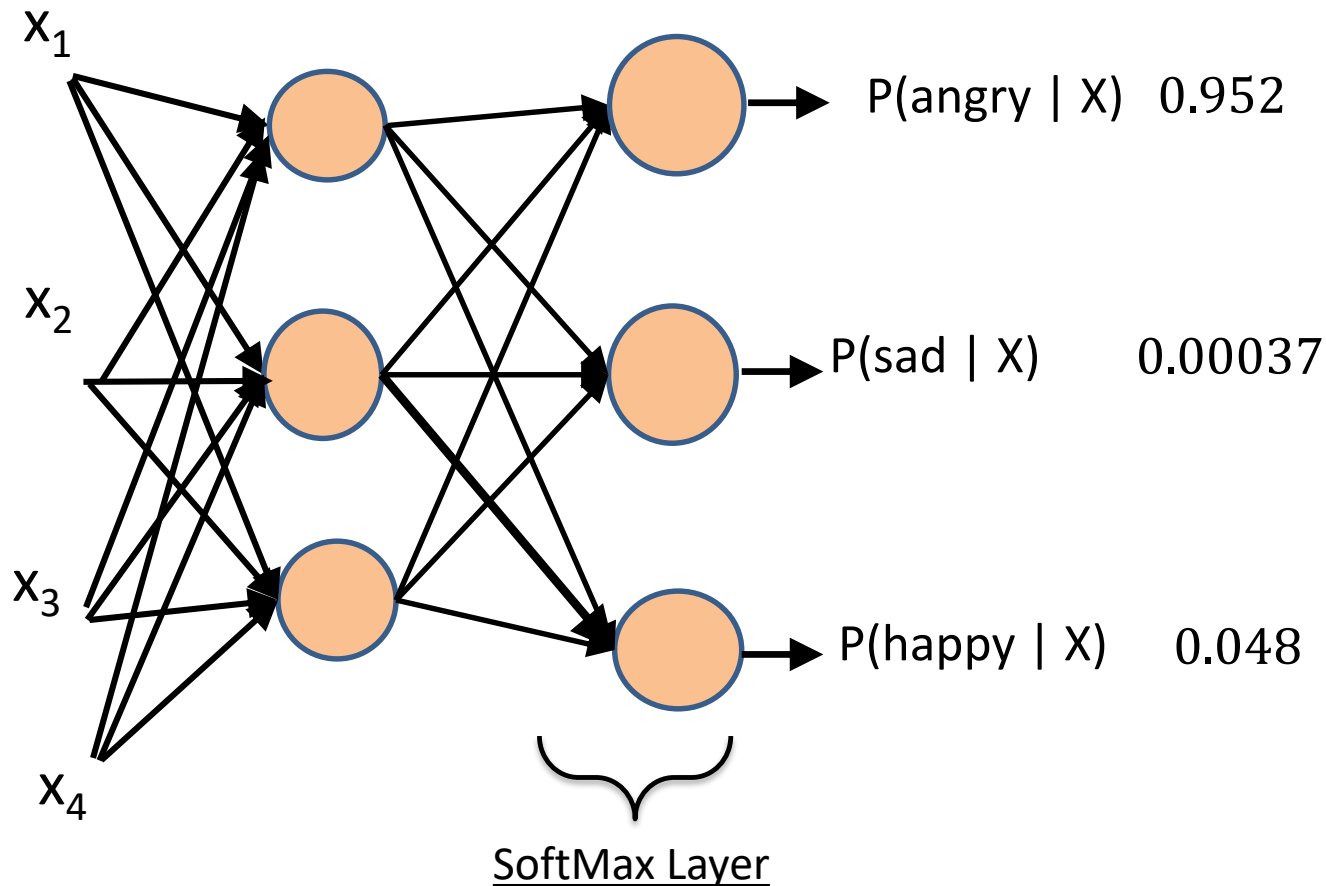
$$\sum t = 423.5$$

$$H^{[2]} = \frac{[403.4, 0.135, 20.0]}{423.5}$$

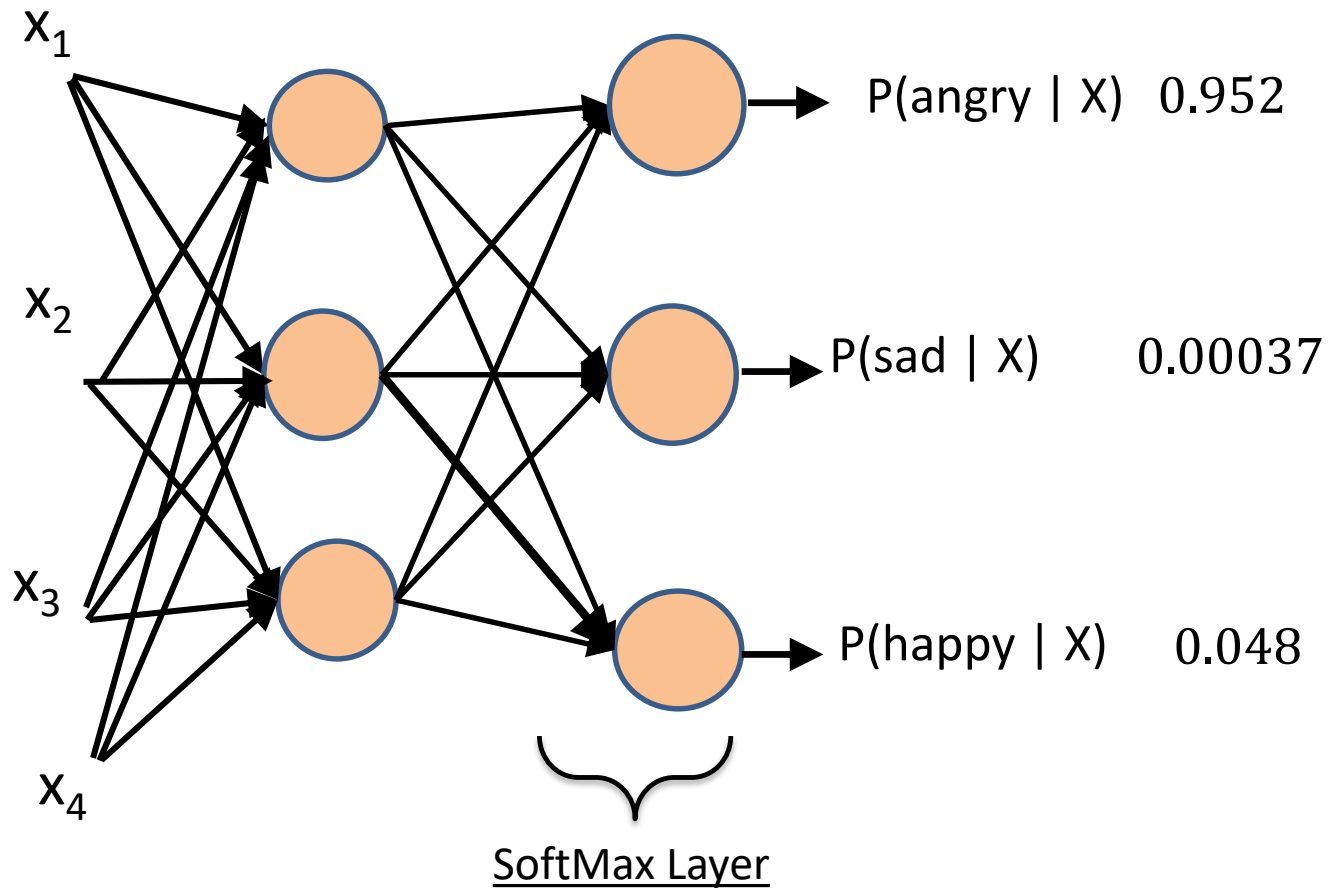
$$= [0.952, 0.00037, 0.048]$$

Going back to the visualization our output from the Softmax layer is shown below. Notice that our outputs all sum to 1.

It should be noted that the true target classification values are often **one hot encoded** when dealing with Softmax. For this training instance the true class label could be represented as a the following vector of values **<1, 0, 0>**



Our next step is to use our **loss function** to determine how good or bad this prediction is.



Notation for Loss Function for Softmax

- ▶ Over the next few slides we will describe the loss function we will use for Softmax. Firstly we will use the following notation.
- ▶ The number of classes in the classification problem is denoted as c .
- ▶ The total number of training instances is denoted as m .
- ▶ \mathbf{p}^i is a vector containing the probability outputs from the softmax layer for the i^{th} training vector.
- ▶ The j^{th} element of this vector is p_j^i .
- ▶ For example, assume in the previous slides that we pushed the first training vector through the Softmax layer. Therefore, the probabilities outputted from the Softmax layer \mathbf{p}^1 is the vector $\langle 0.952, 0.00037, 0.048 \rangle$.
- ▶ p_1^1 would refer to 0.952, p_2^1 would refer to 0.00037 and p_3^1 would refer to 0.048

Notation for Loss Function for Softmax

- ▶ Over the next few slides we will describe the loss function that we will use for Softmax. Firstly we will use the following notation.
- ▶ \mathbf{y}^i is the true class label for the i^{th} training instance. Each true class label is represented as a one hot encoded vector. Therefore, \mathbf{y}^i is a one-hot encoded vector with c elements. Remember c is the number of classes in this problem. We can refer to the j^{th} element of \mathbf{y}^i as y_j^i
- ▶ Therefore, if the true class label for the first training example is $\langle 1, 0, 0 \rangle$ then:
 - ▶ $\mathbf{y}^1 = \langle 1, 0, 0 \rangle$
 - ▶ $y_1^1 = 1$, $y_2^1 = 0$ and $y_3^1 = 0$
- ▶ Clearly because \mathbf{y}^i is one hot encoded then only one value within the vector can have a value of 1.

Notation for Loss Function for Softmax

- ▶ The loss function used for the Softmax function is a variant of the cross entropy loss function.
- ▶ The loss for any single training instance i can be denoted as

$$L(p^i, y^i) = - \sum_{j=1}^c y_j^i \log(p_j^i)$$

- ▶ Clearly we know that only one value in y^i can have the value 1.
- ▶ For example let's assume that the correct class for training instance 1 is class 5.
- ▶ Then all we above is calculate the loss (for the first training instance) is to get the negative log of the outputted softmax probability for the fifth class (because this is the true class and ultimately we want to determine how close our predicted probability is for the true class).

Loss Function for Softmax

$$L(p^i, y^i) = -\sum_{j=1}^c y_j^i \log(p_j^i)$$

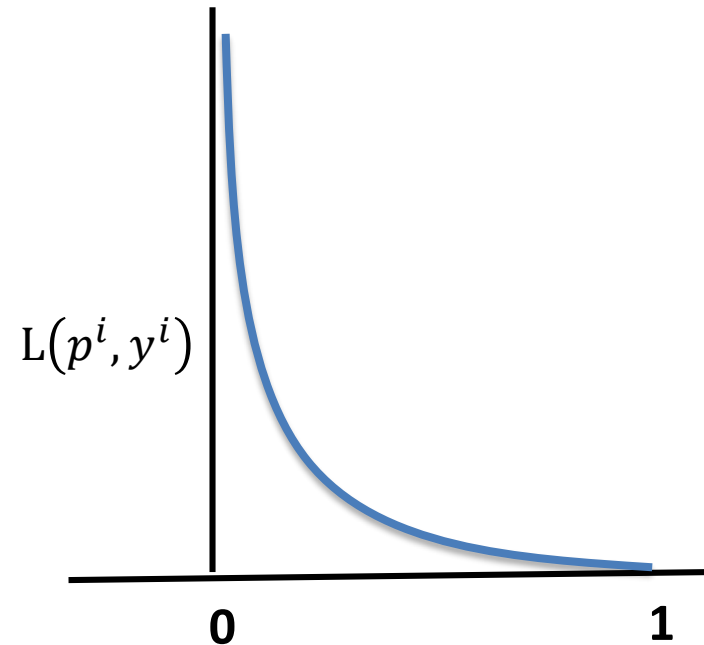
- ▶ Let's work through the example where the outputted probabilities from Softmax for the first training example are $\langle 0.952, 0.00037, 0.048 \rangle$ and the true class labels for the training instance is $\langle \mathbf{1}, \mathbf{0}, \mathbf{0} \rangle$.

$$p^1 = \langle 0.952, 0.00037, 0.048 \rangle$$

$$y^1 = \langle \mathbf{1}, \mathbf{0}, \mathbf{0} \rangle$$

- ▶ $L(p^1, y^1) = -1(\log(0.952)) - 0(\log(0.00037)) - 0(\log(0.048))$
 $= -0.021$

[a low loss, the true class is 1, and softmax outputs a high probability for class 1]



Loss Function for Softmax

$$L(p^i, y^i) = -\sum_{j=1}^c y_j^i \log(p_j^i)$$

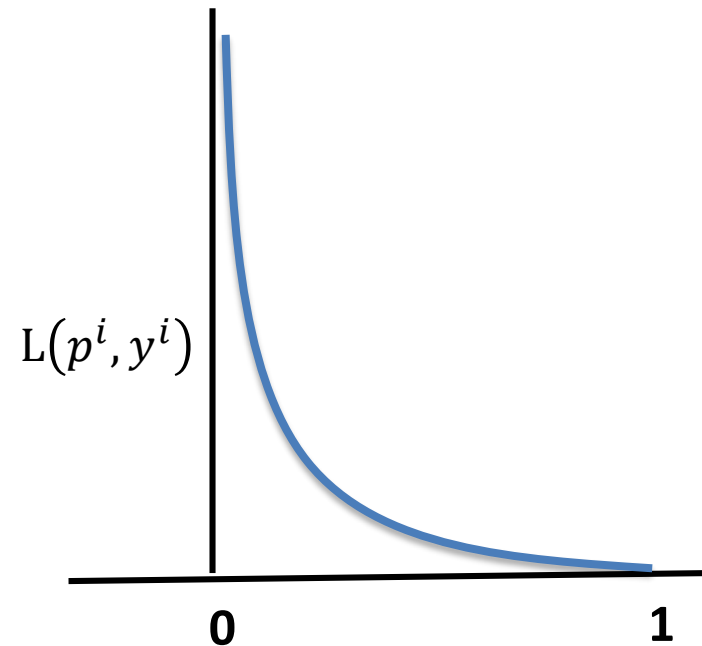
- ▶ Let's look at what happens if the encoded class label was changed. Therefore again, the outputted probabilities from Softmax for the first training example are $\langle 0.952, 0.00037, 0.048 \rangle$ but this time the true class labels for the training instance is $\langle \mathbf{0}, \mathbf{0}, \mathbf{1} \rangle$.

$$p^1 = \langle 0.952, 0.00037, 0.048 \rangle$$

$$y^i = \langle \mathbf{0}, \mathbf{0}, \mathbf{1} \rangle$$

- ▶ $L(p^1, y^1) = -\mathbf{0}(\log(0.952)) - \mathbf{0}(\log(0.00037)) - \mathbf{1}(\log(0.048))$
 $= 1.31$

[a high loss, the true class is 1, and softmax outputs a low probability for class 1]



Average Total Loss for Softmax

- ▶ While the function below provides the loss for a specific training instance we need to calculate the individual loss for all training instances and then calculate the average (we will refer to this as the cost denoted C)

$$L(p^i, y^i) = - \sum_{j=1}^c y_j^i \log(p_j^i)$$

- ▶ Once we know the individual loss for each training instance we can calculate the average loss across all training instances (C) as shown below (as usual m below refers to the total number of training instances in the training set).

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

Dangers of Overfitting with Neural Networks

- ▶ As we previously mentioned neural networks can easily overfit on your training data. They are powerful machine learning models that have in some cases millions of learnable parameters.
- ▶ After epoch 8 the model depicted on the right begins to overfit. We can see the validation loss going back up and the training loss continue to fall.



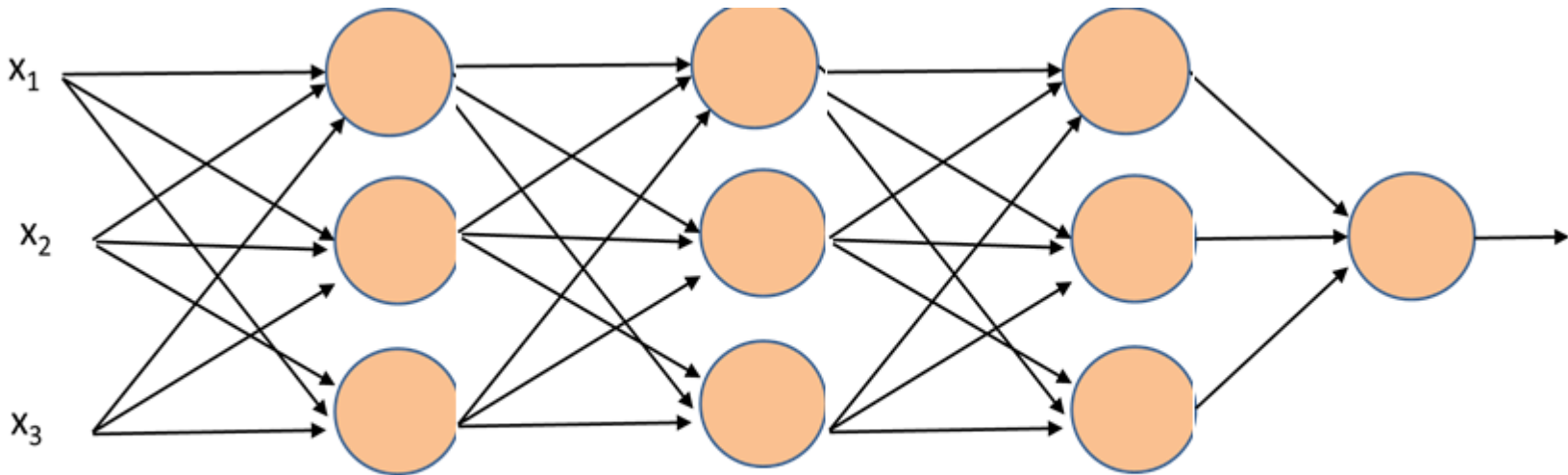
Dangers of Overfitting with Neural Networks

- ▶ There are a number of techniques that can be used to mitigate overfitting when building a model.
- ▶ Two very commonly used techniques are:
 - ▶ Regularization
 - ▶ Dropout



Regularization

- ▶ In neural networks there is a relationship between the **magnitude of the weights and the complexity of the model**. For example, if we let neural network execute for a large number of iterations you will find that some of the weights become quite large and it starts to overfit on the training data.
- ▶ The idea behind regularization is that we try to encourage the model to reduce the magnitude of the weights.



Regularization

- ▶ To understand regularization we focus on it's application to an MLR.
- ▶ Remember in Multivariate Linear Regression we are trying to minimize the following cost function, where L is the cross entropy error function.

$$C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m ((h(\mathbf{x}^i) - y^i))^2$$

- ▶ How might we modify a model to reduce the magnitude of it's weights?
- ▶ Well one approach is that we could add an additional component to our loss function related to the weights.
- ▶ So gradient descent will now not only try to reduce the difference between the predicated and actual value but also the magnitude of the weights.

Regularization

- ▶ To add regularization to an MLR we add the following to our cost function (below is referred to as L2 regularization). This variant of an MLR is referred to a Ridge regression.

- ▶
$$C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m ((h(\mathbf{x}^i) - y^i))^2 + \delta \sum_{j=1}^n \lambda_j^2$$

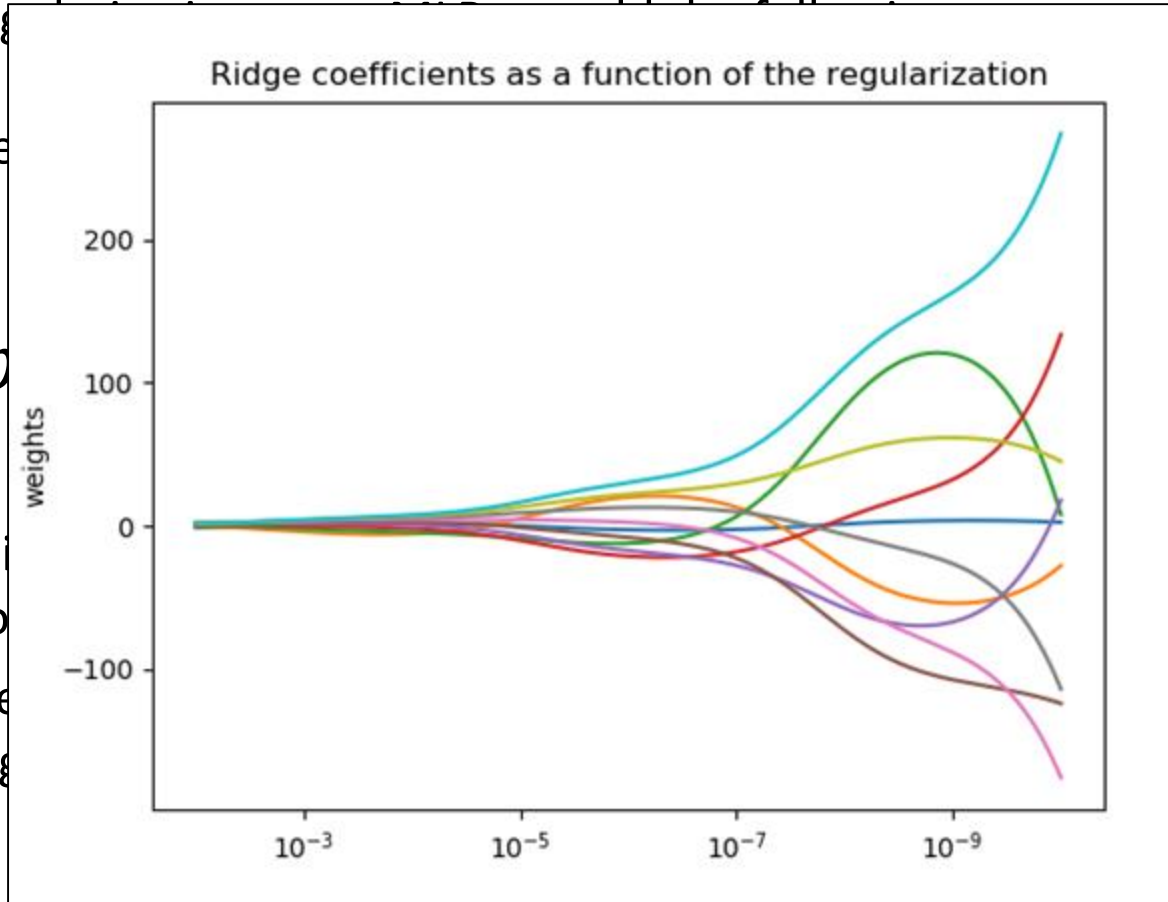
- ▶ Where δ is a constant called the regularization rate and controls the amount of shrinkage in the weights.
- ▶ Remember n is the number of features/coefficients, while m is the number of training examples.
- ▶ Penalizes the model for having very large weights.
- ▶ The **larger the value of δ** then the **greater** the amount of **shrinkage in the weights**.

Regularization

- ▶ To add regularization to the cost function (below is the cost function for a Ridge regression)

▶ $C(\lambda, b)$

- ▶ Where δ is the amount of regularization
- ▶ Remember that the cost function is a function of training data
- ▶ Penalizes large weights
- ▶ The larger the value of δ then the greater the amount of shrinkage in the weights.



the cost function
is referred to as the

λ_j^2

the
the number

Regularization

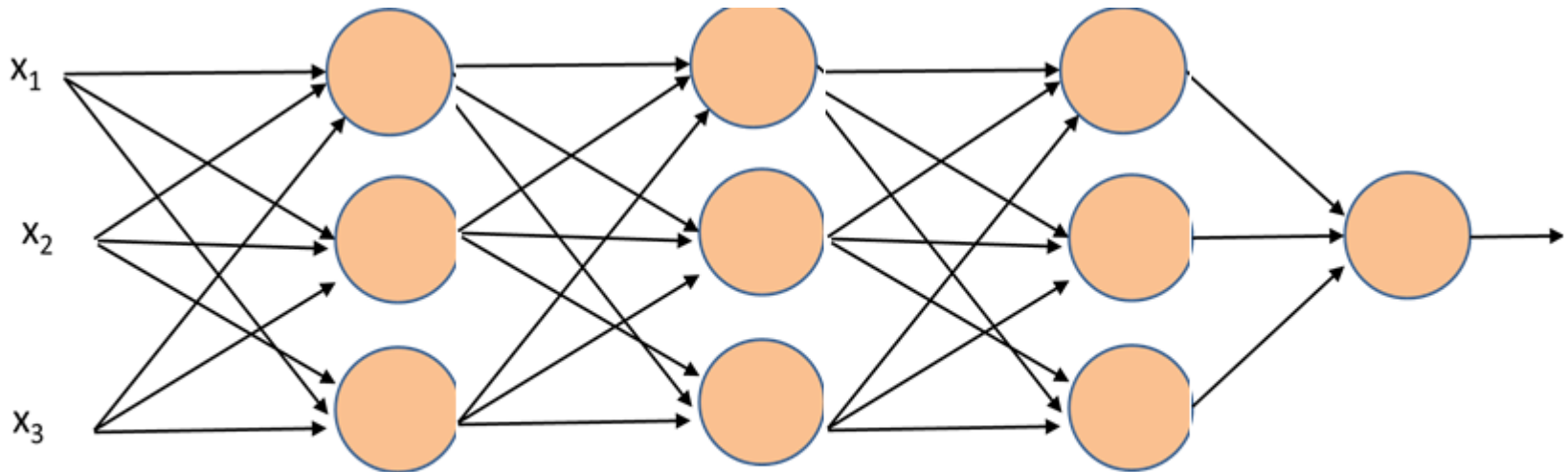
- ▶ Another commonly used variant is referred to as L1 regularization. This is often referred to as Lasso Regression when used with an MLR.

- ▶
$$C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m ((h(\mathbf{x}^i) - y^i))^2 + \delta \sum_{j=1}^n \lambda_j$$

- ▶ L1 just adds up each weight and opposed to the squared value of each weight.
- ▶ L1 in the form of an MLR (Lasso) is often used a mechanism of feature selection as it reduce the number of features on which the given solution is dependent.

Regularization

- ▶ The more we **increase our regularization parameter** the more aggressively our algorithm will attempt reduce the weights.
- ▶ This can have the effect of almost negating the impact of some of the hidden units (or dramatically reducing their impact).
- ▶ As such you end up with a simpler model and simpler models are less prone to overfitting.
- ▶ However, if set your regularization parameter to be low then the weights will not decrease as aggressively, which can cause higher variance as a result.

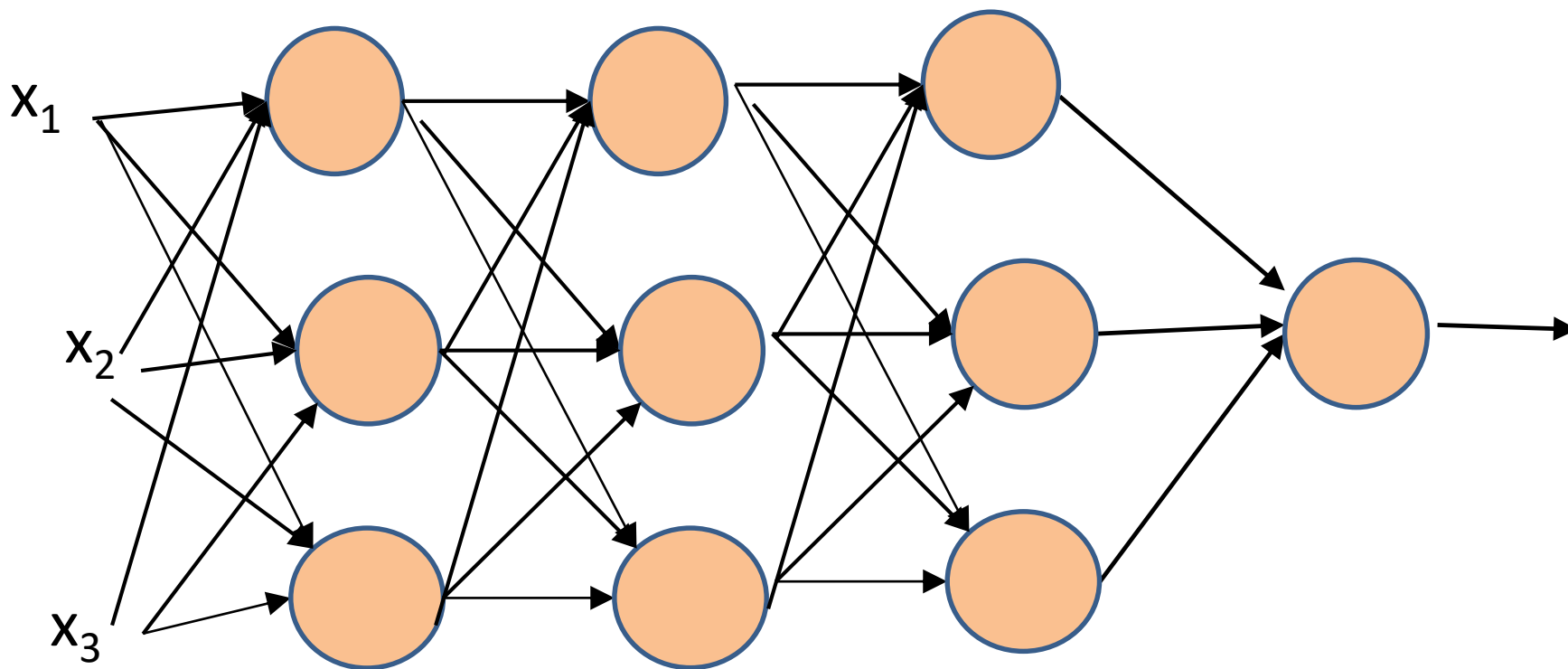


Dropout Regularization

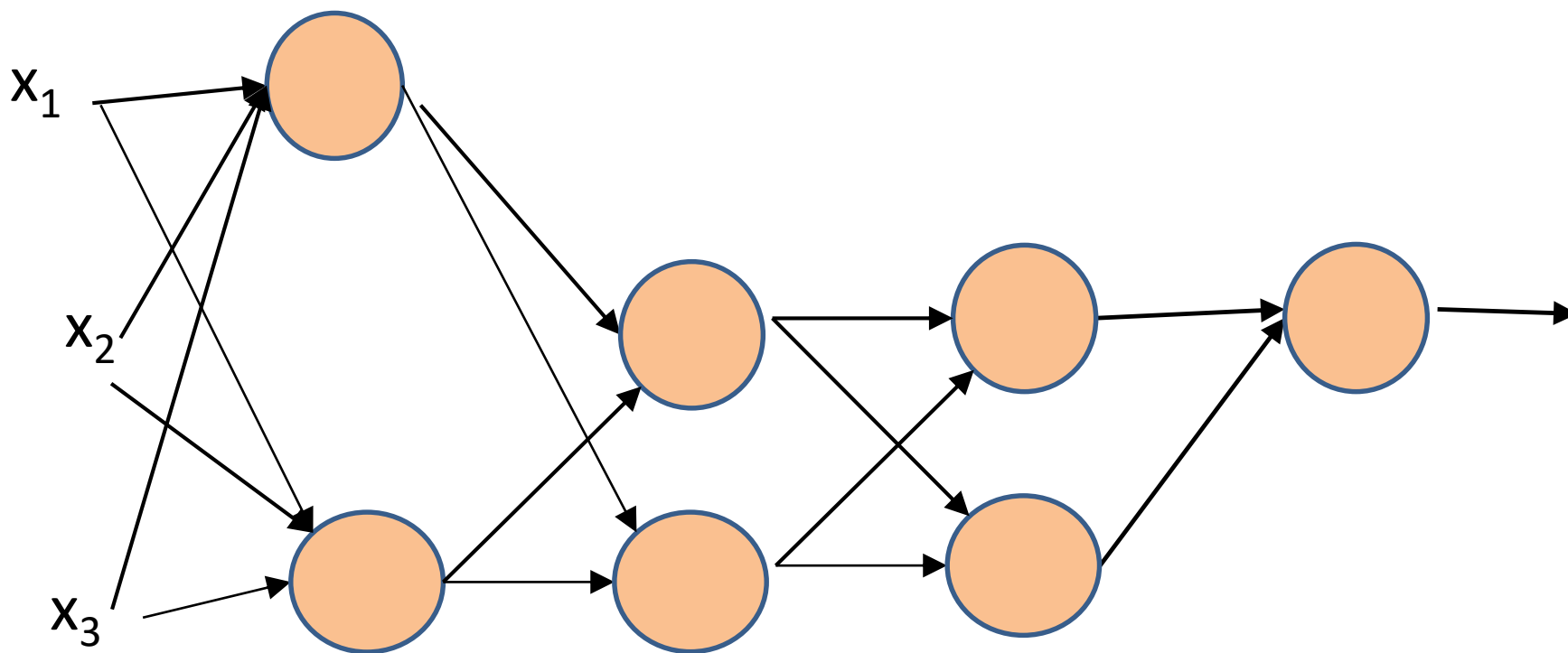
- ▶ Drop-out regularization associates a specific drop-out probability with each layer of a neural network.
- ▶ When training the network our algorithm picks the next training example. It then steps through each layer in the network.
 - ▶ In each layer it generates a random probability for each node in that layer.
 - ▶ If the random number generated for a node is lower than the drop-out probability then the node is removed from the neural network.
 - ▶ The remaining nodes and links are then trained (forward pass and backward pass just for that specific training example).

[Improving neural networks by preventing co-adaptation of feature detectors \(2012\)](#)

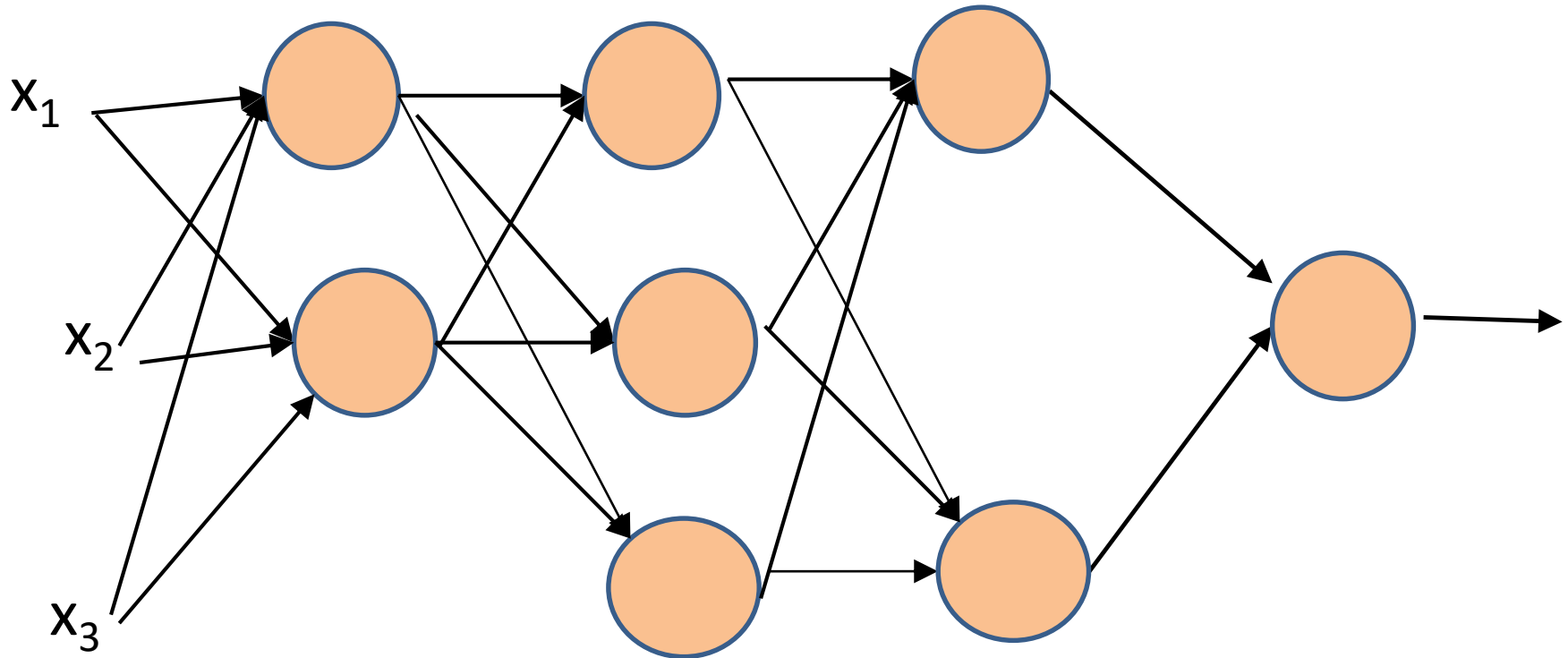
In this example, let's assume the drop probability is 0.3.



For the **first training example**, we may end up with the following network



For **the second training example**, we may end up with the following network



Sometime dropout is referred to as a **dropout layer**. However, as you can see this is a little misleading. It is easier to think of dropout as a **filter** applied to the outputs of an existing layer. We will see this in more detail over the next few slides.

Vectorized Forward Pass for ANNs

$$\begin{aligned} A^{[1]} &= w^{[1]}X + b^{[1]} \\ H^{[1]} &= \text{act}(A^{[1]}) \end{aligned}$$

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).

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

Vectorized Forward Pass for ANNs

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

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


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).


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

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

Breakdown of matrix $H^{[L]}$

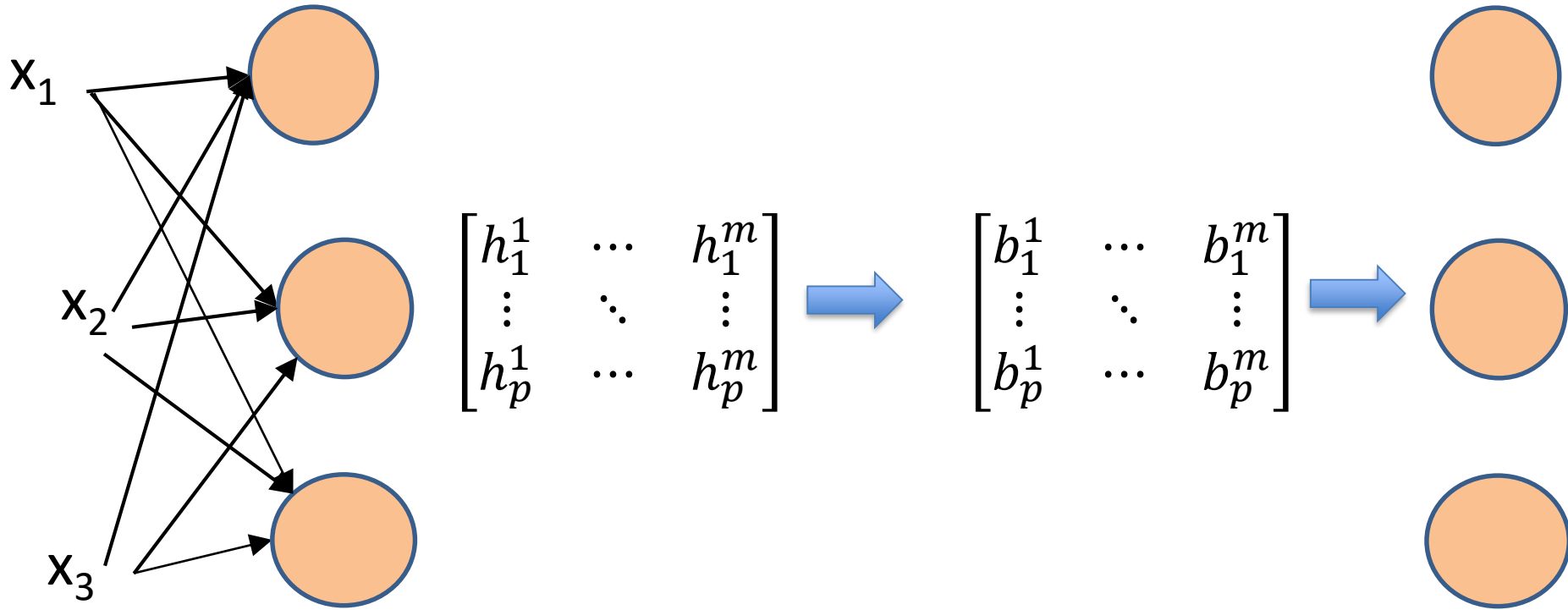
Each columns corresponds to the output for a single instance from each neuron.
We have p neurons.


$$\begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix}$$



Each row corresponds to a neuron and the output of that neuron for all training example. For each neurons we will have m output values.

So going back to our example network this is the matrix that will be outputted. This output matrix is multiplied by a binary filter which has the impact of reduce the output from specific neurons for specific training examples to 0



Dropout Regularization

- ▶ The following pseudocode illustrates the basic concept of drop-out regularization for a neural network
- ▶ Remember the matrix $H^{[L]}$ is a matrix output for layer L that contains a row for each neuron and a column for each training example. For example, the first row contains the output from node 1 in layer L for each training example.

```
probThreshold = 1- dropOutProb
```

```
For each layer (L) in your neural network
```

```
neuronsSize =  $H^{[L]}$ .shape[0]
```

```
trainingSize =  $H^{[L]}$ .shape[1]
```

```
dropMatrix = np.random.rand(neuronsSize, trainingSize) < probThreshold
```

```
 $H^{[L]} = H^{[L]} * \text{dropMatrix}$ 
```

```
 $H^{[L]} = H^{[L]} / \text{probThreshold}$ 
```

```
 $A^{[L+1]} = W^{[L+1]} . H^{[L]} + b^{[L+1]}$ 
```

```
...
```

This line generates a 2D Boolean array with the same dimensions as H. Therefore, consider the first row (which corresponds to node 1). Assuming probThreshold is 0.75 then approx. $\frac{1}{4}$ of the elements in row 1 will be false.

```
probThreshold = 1- dropOutProb
```

For each layer (L) in your neural network

```
neuronsSize = H[L].shape[0]
```

```
trainingSize = H[L].shape[1]
```

```
dropMatrix = np.random.rand(neuronsSize, trainingSize) < probThreshold
```

```
H[L] = H[L] * dropMatrix
```

```
H[L] = H[L] / probThreshold
```

```
A[L+1] = W[L+1] . H[L] + b[L+1]
```

```
...
```

The second line multiplies the outputs of layer L for every training example by the dropout matrix. This has the effect of making some of the outputs zero. Those that are multiplied by False.

probThreshold = 1- dropOutProb

For each layer (L) in your neural network

neuronsSize = $H^{[L]}$.shape[0]

trainingSize = $H^{[L]}$.shape[1]

dropMatrix = np.random.rand(neuronsSize, trainingSize) < probThreshold

$H^{[L]} = H^{[L]} * \text{dropMatrix}$

$H^{[L]} = H^{[L]} / \text{probThreshold}$

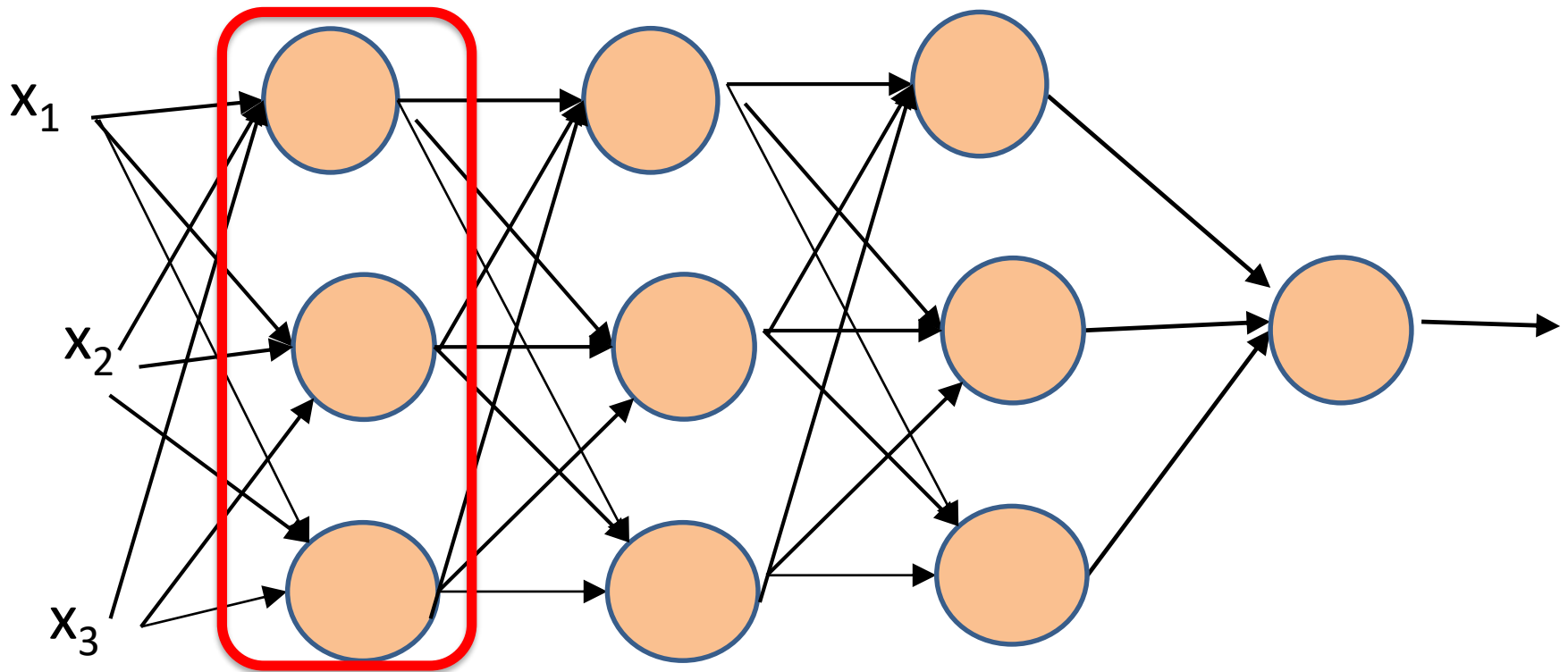
$A^{[L+1]} = W^{[L+1]} . H^{[L]} + b^{[L+1]}$

...

This line performs **scaling** (specifically we scale up the values to compensate for those values that have been removed). If we have performed dropout on layer 3 and we begin processing for layer 4 then we will have $A^{[4]} = W^{[4]} . H^{[3]} + b^{[4]}$. Therefore, the expected value of $A^{[4]}$ will be significantly reduced. In order to compensate for this after we perform the dropout we scale the result upward to compensate for the loss.

Dropout Example

- Let's return to our previous network where probability of dropout is 0.3. Let's consider applying dropout to layer 1.
- In this example, we push four training instances through our network.



Dropout Example

- The first step is to **randomly generate the Boolean array** that will dictate which neurons are removed from consideration for each training example.
- The notation dM below is short for dropout matrix
- We can see that for the first training example, we are going to remove neuron 3. For the second training example, we will be removing neuron 1. All neurons are retained for the third training example and for the fourth example both neuron 2 and 3 are removed.

$$H^{[1]} = \begin{bmatrix} 0.1 & 0.2 & 0.4 & 0.5 \\ 0.2 & 0.05 & 0.1 & 0.4 \\ 0.25 & 0.1 & 0.6 & 0.1 \end{bmatrix} * \text{dM} = \begin{bmatrix} T & F & T & T \\ T & T & T & F \\ F & T & T & F \end{bmatrix}$$

$$H^{[1]} = \begin{bmatrix} 0.1 & \mathbf{0} & 0.4 & 0.5 \\ 0.2 & 0.05 & 0.1 & \mathbf{0} \\ \mathbf{0} & 0.1 & 0.6 & \mathbf{0} \end{bmatrix}$$

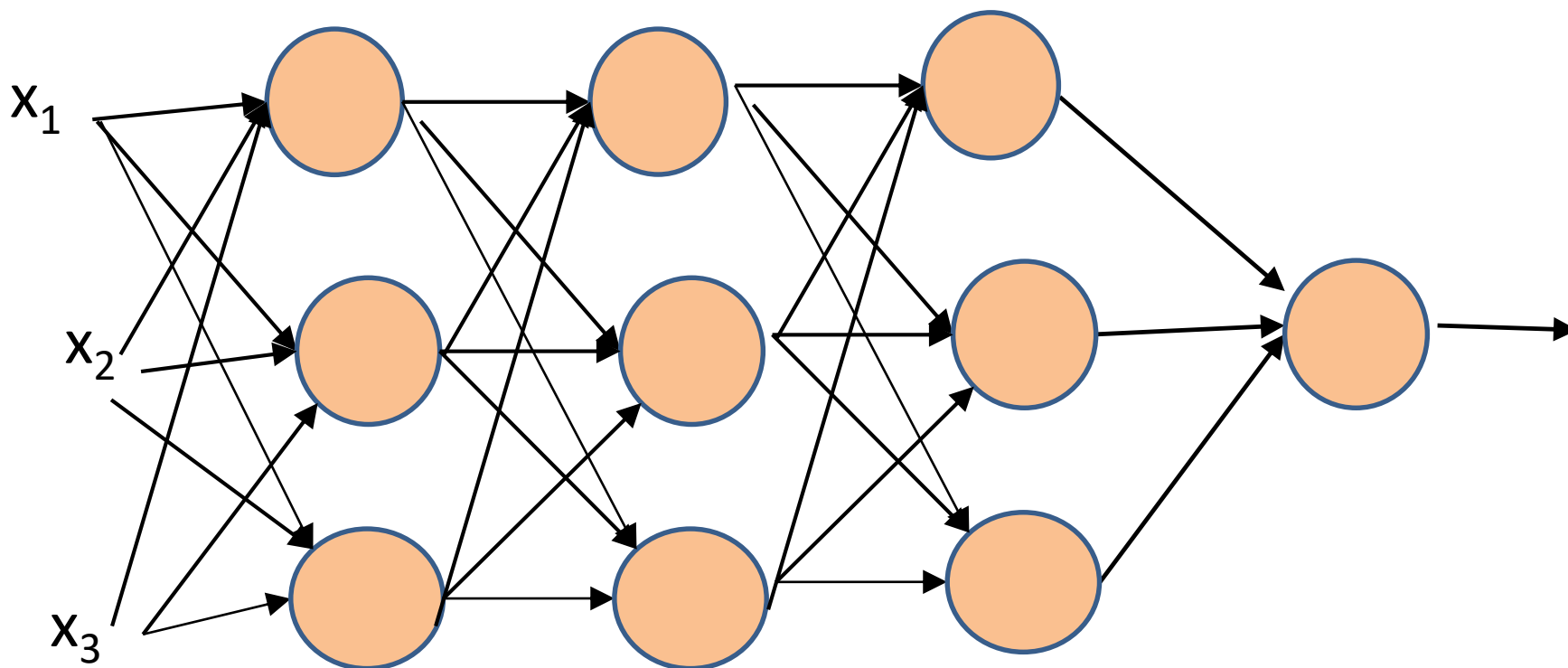
Dropout Example

- Finally we need to rescale by dividing by the 1- dropout probability (we referred to this as the probability threshold in our code).

$$H^{[1]} = \begin{bmatrix} 0.1 & 0.2 & 0.4 & 0.5 \\ 0.2 & 0.05 & 0.1 & 0.4 \\ 0.25 & 0.1 & 0.6 & 0.1 \end{bmatrix} * \text{dM} = \begin{bmatrix} T & \mathbf{F} & T & T \\ T & T & T & \mathbf{F} \\ \mathbf{F} & T & T & \mathbf{F} \end{bmatrix}$$

$$H^{[1]} = \begin{bmatrix} 0.1 & \mathbf{0} & 0.4 & 0.5 \\ 0.2 & 0.05 & 0.1 & \mathbf{0} \\ \mathbf{0} & 0.1 & 0.6 & \mathbf{0} \end{bmatrix}$$

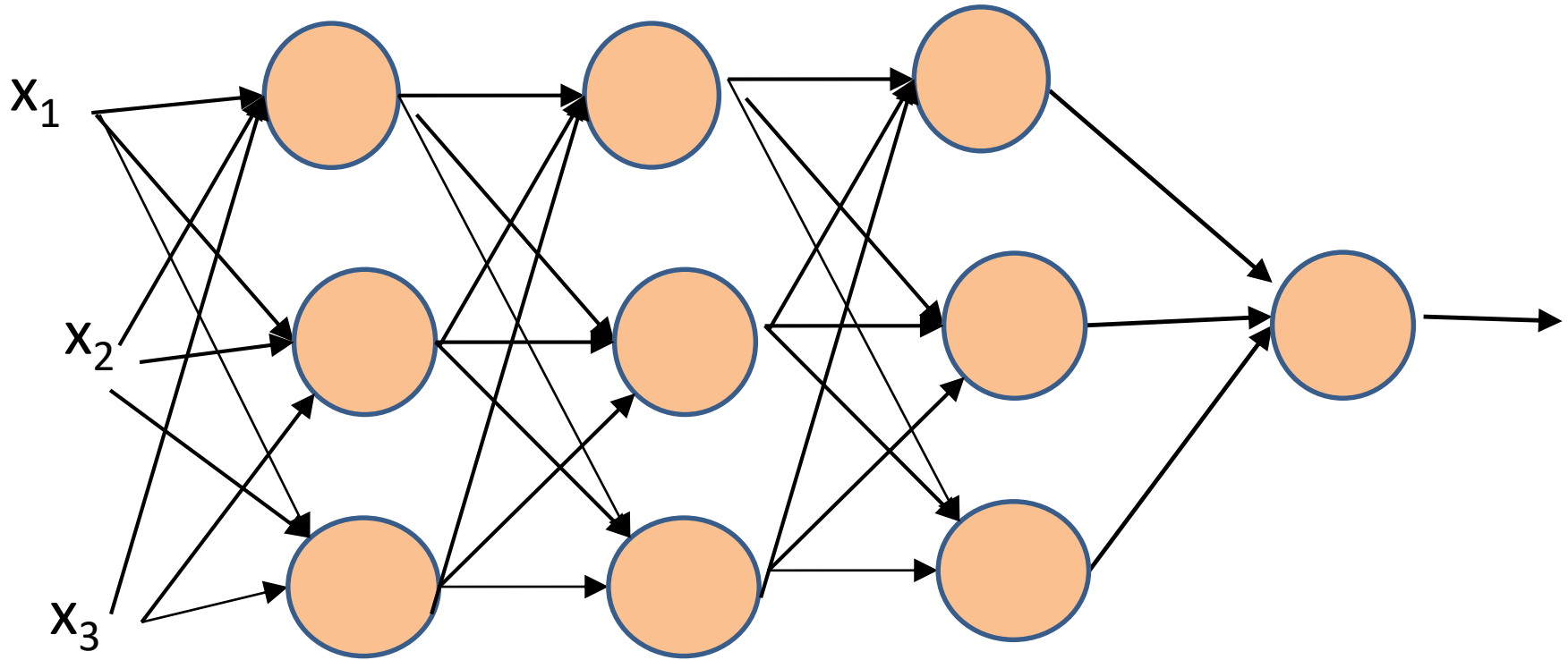
$$H^{[1]} = \begin{bmatrix} 0.1/0.7 & \mathbf{0}/0.7 & 0.4/0.7 & 0.5/0.7 \\ 0.2/0.7 & 0.05/0.7 & 0.1/0.7 & \mathbf{0}/0.7 \\ \mathbf{0}/0.7 & 0.1/0.7 & 0.6/0.7 & \mathbf{0}/0.7 \end{bmatrix} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$



$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

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

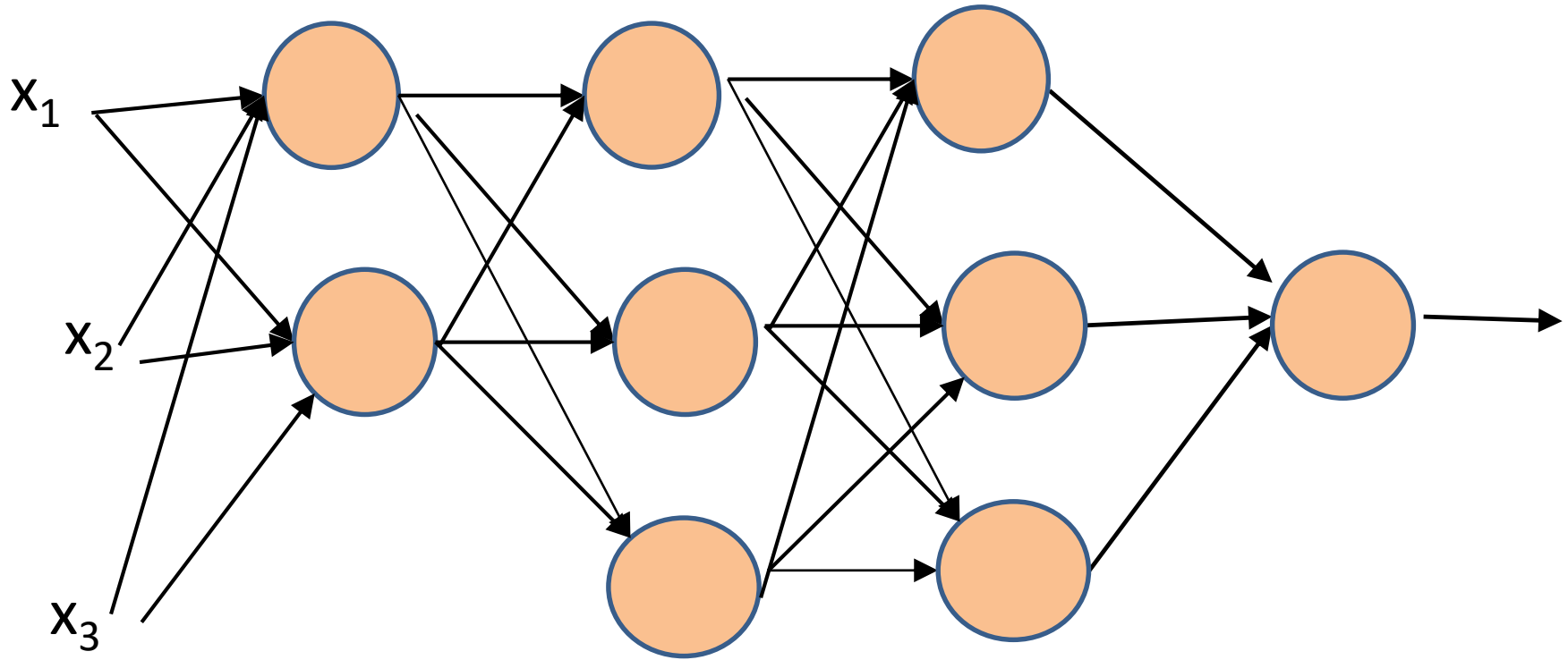
$$H^{[2]} = \text{logistic}(A^{[2]})$$



↓

$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

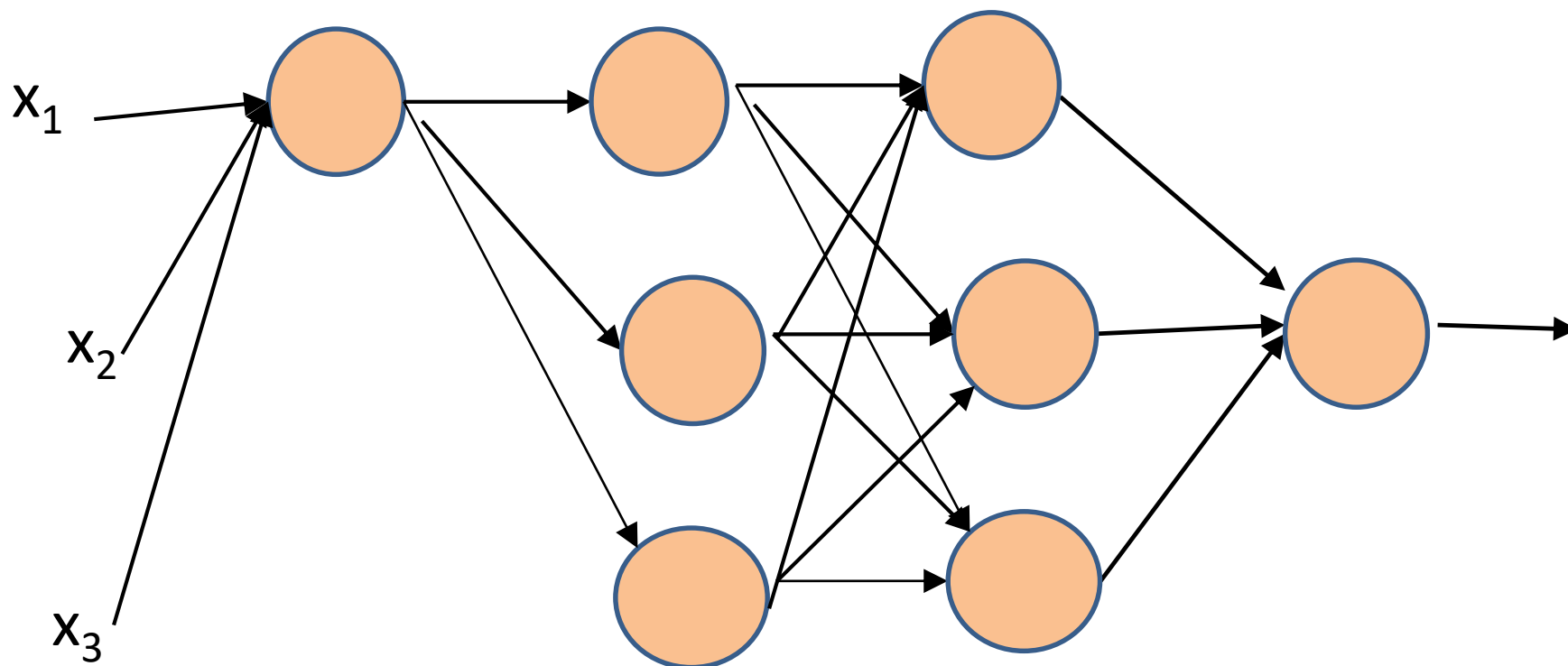
Let's look at example 1. Notice it will have no output from the third neuron.



↓

$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

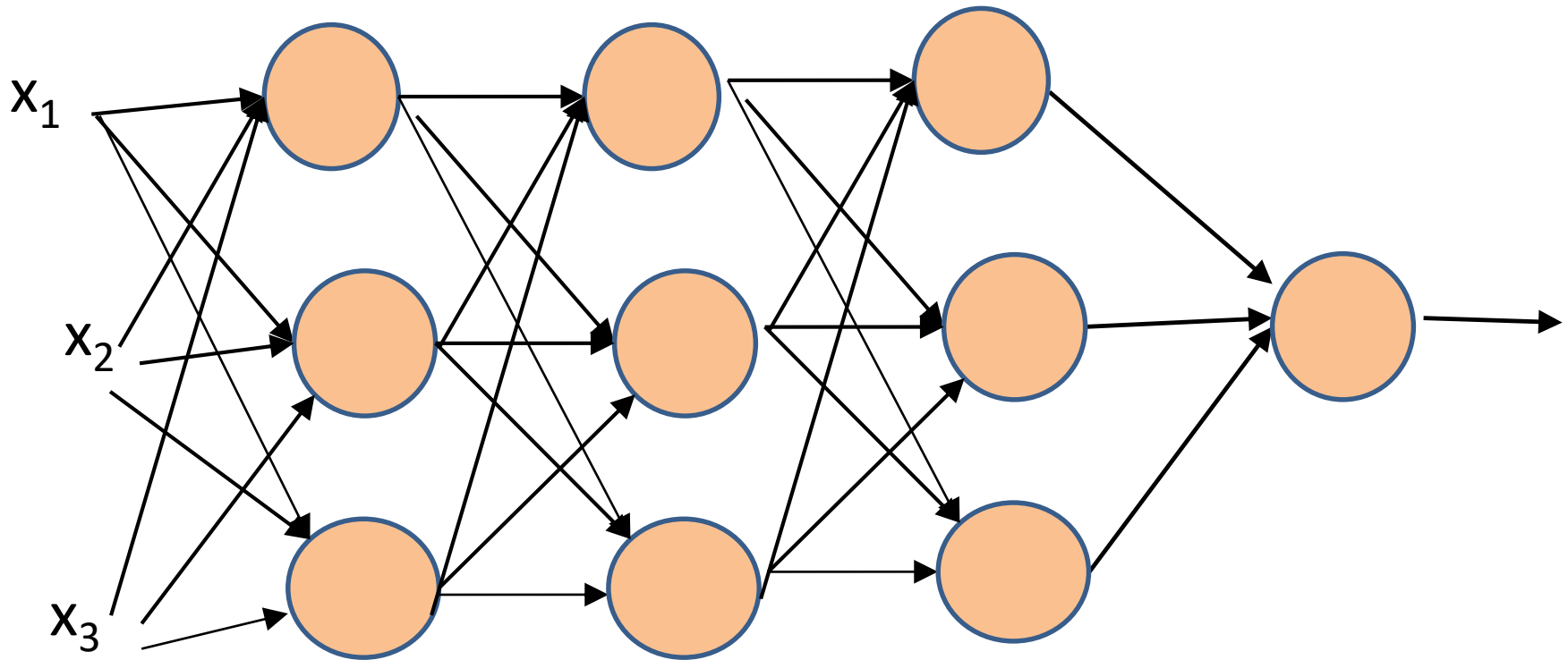
Let's look at training example 1.
Notice it will have no output
from the third neuron.



$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

Let's look at training example 4.
Notice it will have no output
from the second or third
neuron.

Once we have rescaled our data we can continue as normal and push the training examples through the second later. Once we have calculated the output of the second layer $H^{[2]}$ then we can repeat the same process again and apply dropout again.



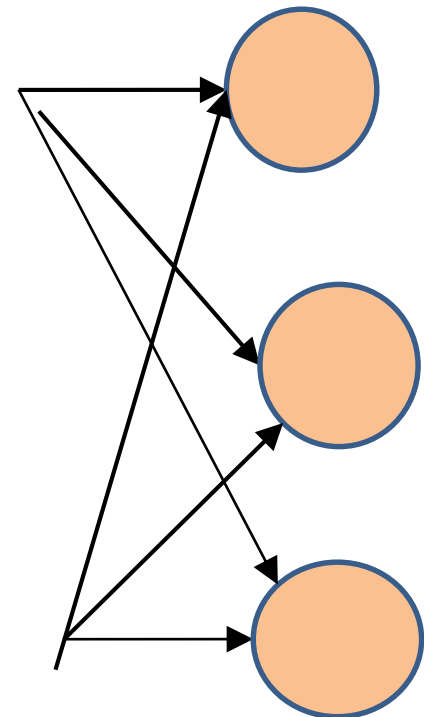
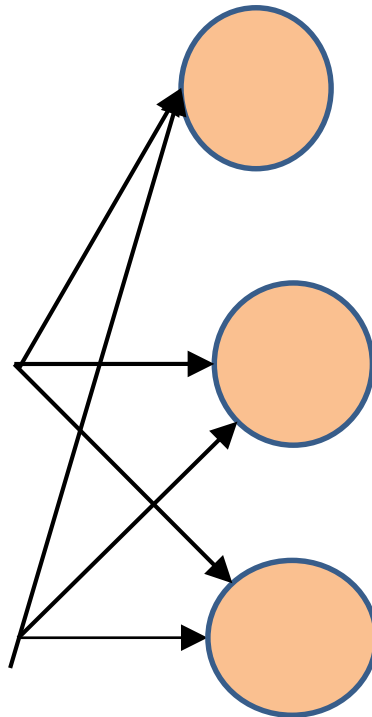
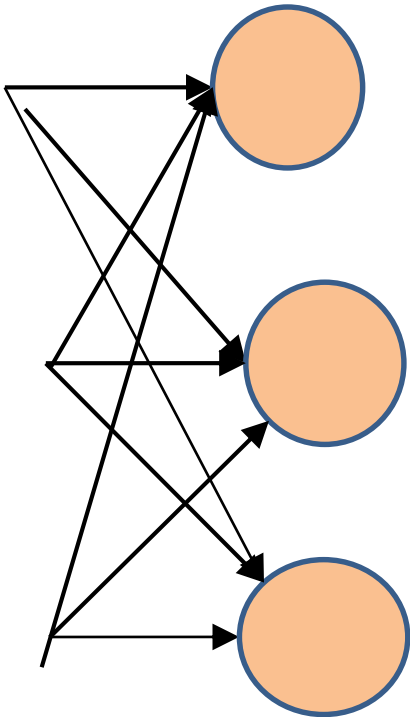
$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

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

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

Observations

- ▶ One of the effects of drop-out is that it helps distribute the weights for each layer.
- ▶ Neurons can't depend too much on one incoming connection because it may not always be there (it may be dropped in training) and this causes them to more evenly distribute the weights amongst the incoming connections. In other words it aims to avoid a scenario where a few incoming weight get very high.
- ▶ The consequence of this is that the squared sum of the weights will be lower if we perform drop-out. Similar to the impact of L2 regularization.



Rationale

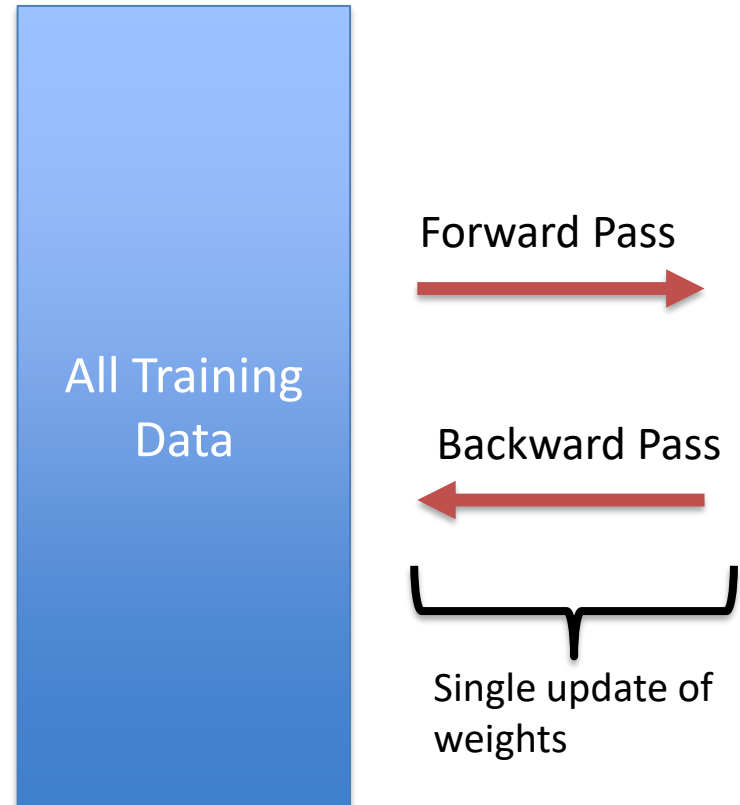
- ▶ Of course the consequence of using dropout is that we now have yet another **hyper-parameters** to estimate for our neural network.
 - ▶ The appropriate range of values is an open question and can vary depending on the type of network you are training. For example, Hinton's original paper used a dropout prob of 0.5 for standard deep densely connect neurons, while more [recent work](#) has shown that a dropout in the range 0.1 to 0.2 is better for convolutional neural networks.
 - ▶ You will notice that in the previous example we were using a single dropout probability. It is worth mentioning that the drop-out probability doesn't necessary have to be the same for every layer.
- ▶ Has been used extensively in **computer vision problem**.
- ▶ Another important consideration with dropout is that our **cost function may not always be decreasing** with every iteration.

Optimization

- ▶ So far we have considered a vectorised solution for neural networks where we push **all of our training data** through the network at one time.
- ▶ Also we have only considered updating the weights using our standard gradient descent update rule.
- ▶ Over the next few slides we will look at widely used variants around both the **forward pass process** and the **update of weights** in the backward pass.
- ▶ **Mini-batch gradient descent.**
- ▶ Learning Rate Decay
- ▶ Adaptive Learning Rates

Batch v's Mini-Batch Gradient Descent

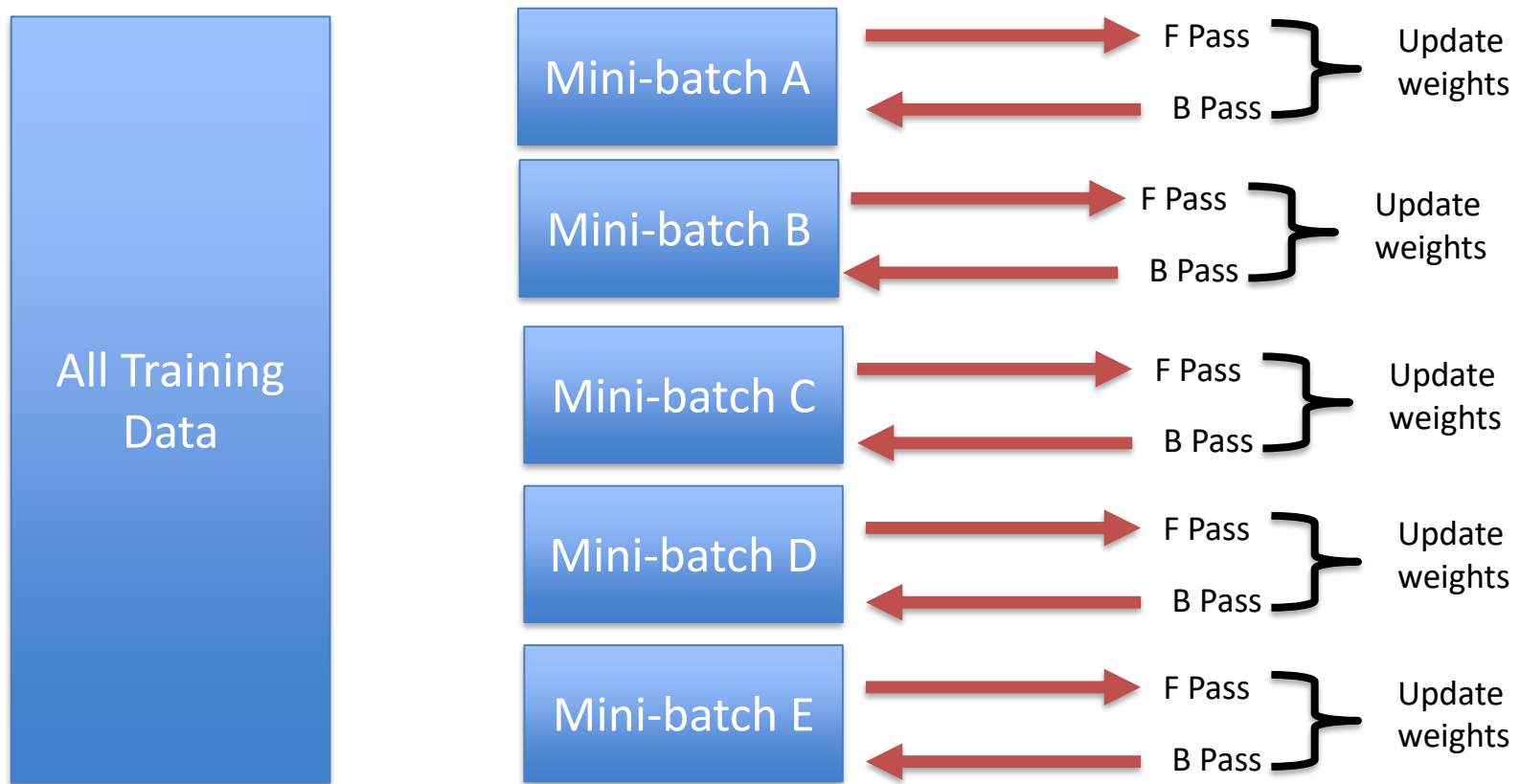
- ▶ Consider a situation where you have a very large number of train examples (**4,000,000**).
- ▶ The issue that arises in such scenarios is that we must push all 4M data points through our graph in order to determine a single update for our weights.
- ▶ This can take a very long time and can mean that training the model takes a very very long time. Given that the process of gradient descent machine learning is so iterative and we try **many different model configurations** this represents a serious problem.
- ▶ This approach is often referred to as **batch processing** (we take the entire batch (training set) for each update of our weights)



Batch v's Mini-Batch Gradient Descent

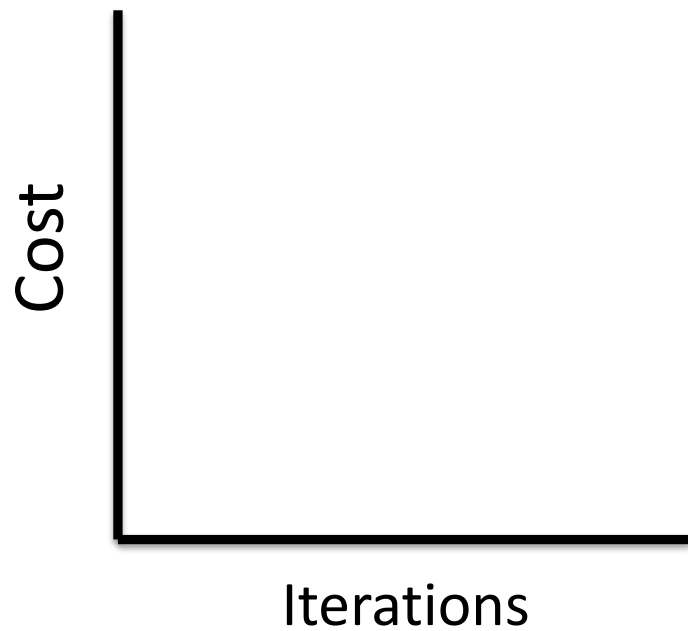
- ▶ An alternative approach that significantly alleviates the problem outlined in the previous slide is to break-up the training data into exclusive partitions, referred to as **mini-batches**.
- ▶ For example, we might break our **4M** training examples in **8000 mini-batches** with **500 training instances** in each mini-batch.
- ▶ Subsequently, we take the first mini-batch and complete a single forward and backward pass and use it to update the weights once. We then take the next mini-batch and complete a single forward and backward pass and use it to update the weights once. We continue this process until we have processed all 8000 mini-batches.
- ▶ At this stage we have completed **one epoch** but the weights have been updated 8000 times (as opposed to just once with batch processing). All training examples have been pushed through the model once. We may then have many epochs before we reach a convergence.

Batch v's Mini-Batch Gradient Descent

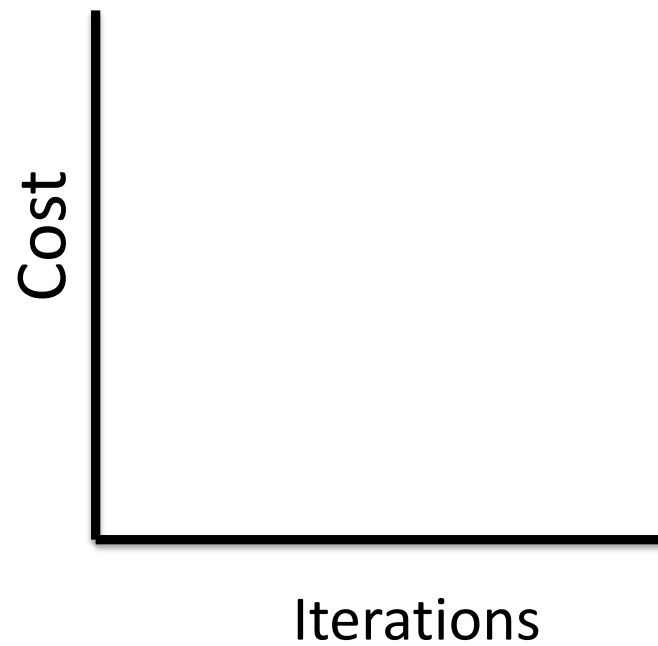


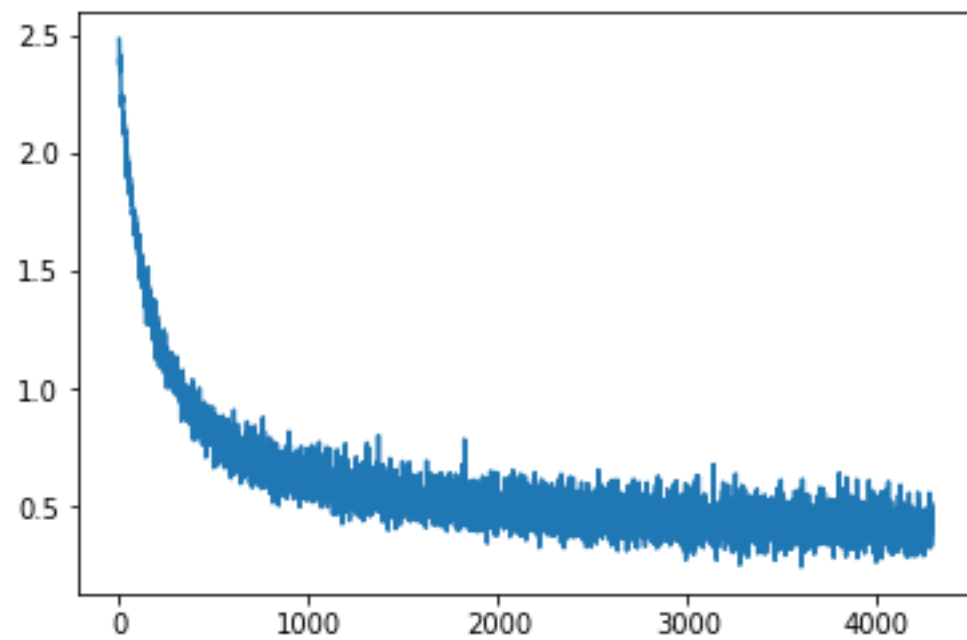
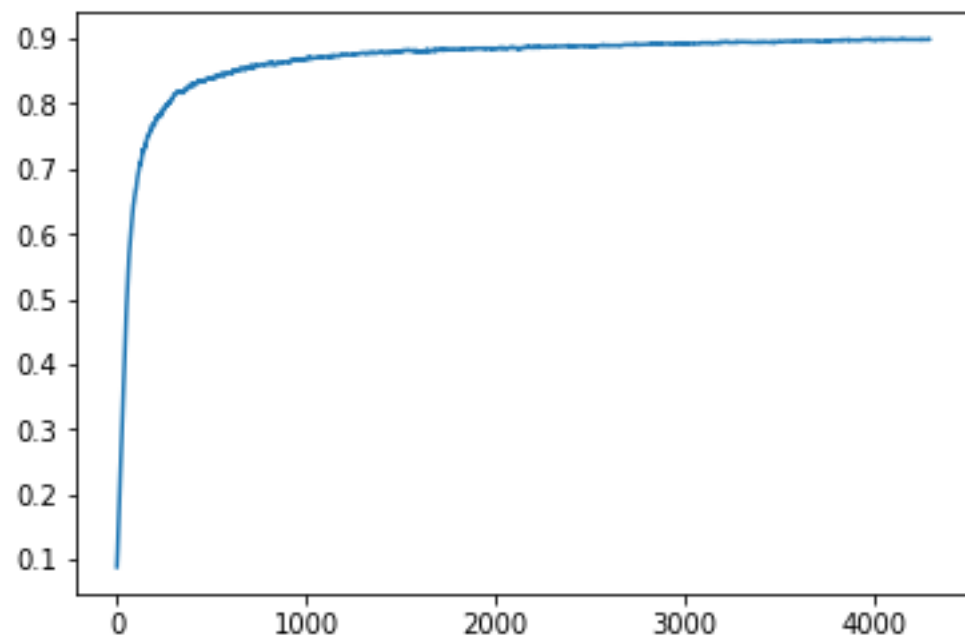
- ▶ The main difference is that when using mini-batch, after one epoch the weights will have been updated mini-batch number of times ... in our example the weights would have been updated 8000 times.
- ▶ However, after a single epoch with batch gradient descent the weight have only been updated once.

Batch GD



Mini-Batch GD





Selecting a size for your Mini-Batch

- ▶ If your batch size is the same as your training set size then you are using **batch GD**.
 - ▶ You would typically only use batch GD when you have a small training set (typically $m < 4000$)
 - ▶ We have already outlined the problem with this option. Too slow for large training sets.

Size of Mini-Batch

- ▶ If the **batch size** = 1 we are using what is called stochastic GD.
 - ▶ The problem with this option is it can be very noisy. Each step will on average take you closer to the minimum but some will also step in the opposite direction.
 - ▶ An additional consequence of the noisy learning process is that stochastic gradient descent can find it difficult to converge on an true minimum for a specific problem and may end up circling the value (but will obtain a good approximation).
 - ▶ The other problem with stochastic GS is that you lose the speed advantage provided by vectorization because you are just processing one training example at a time.

Size of Mini-Batch

- ▶ If ***batch size*** > 1 and $< m$ then we are using what is called mini-batch gradient descent.
 - ▶ Typically sizes for mini-batch is 32, 64, 128, 256, 512, 1024.
 - ▶ The advantage of this technique is that we can achieve a relatively high frequency of weight updates while still retain the benefit of vectorised speed up advantages.
 - ▶ The behaviour of the loss function is still noisy although not as noisy as pure stochastic GD. The larger the batch size the less noisy the behaviour.