

# Machine Learning



## Machine Learning

Lecture: Neural Networks

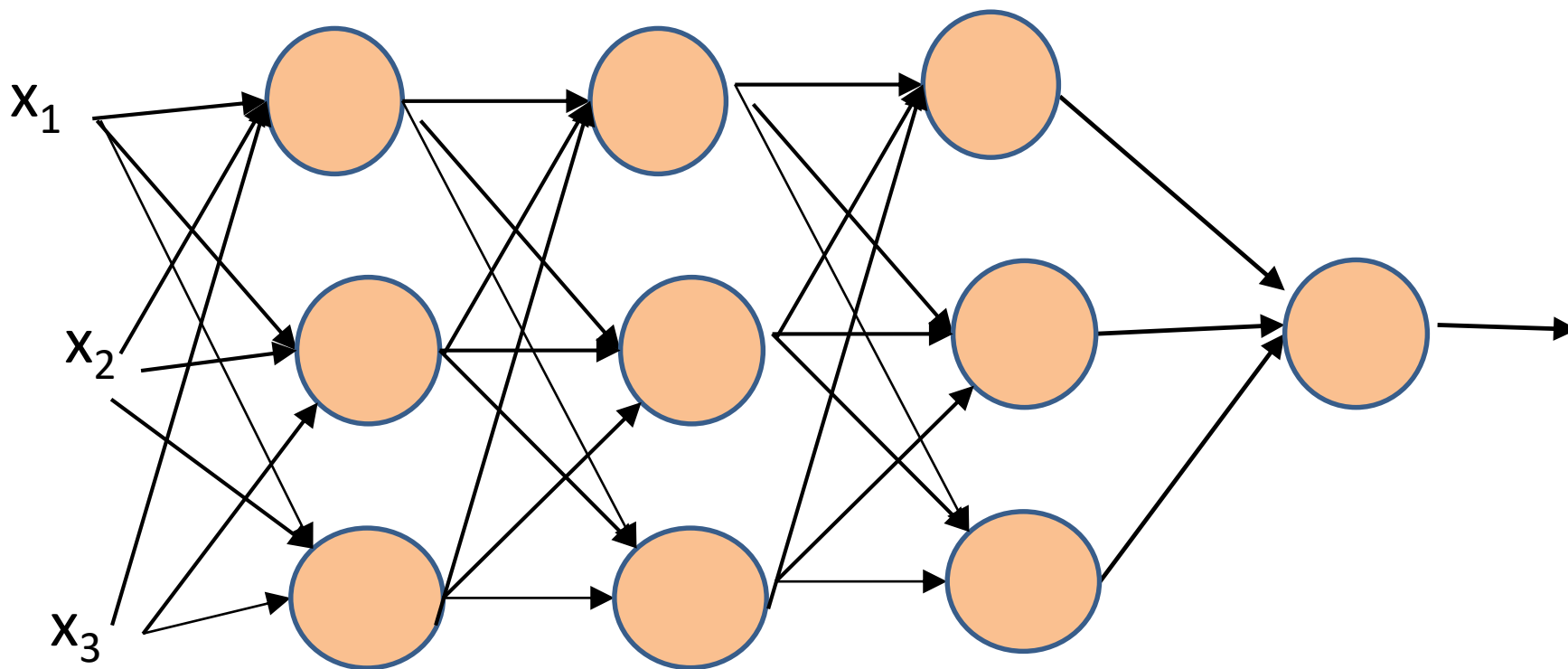
Ted Scully

# Dropout Regularization

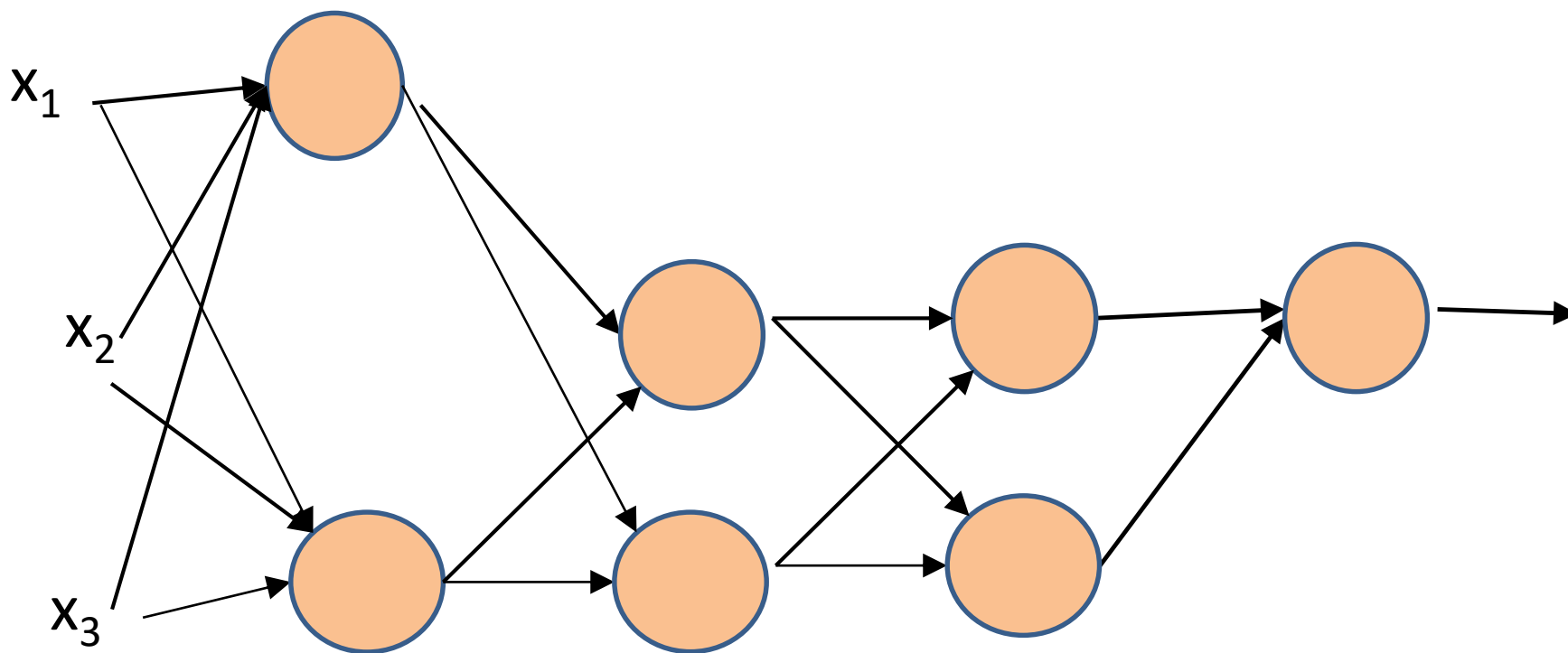
- ▶ Drop-out regularization associates a specific drop-out probability with each layer of a neural network.
- ▶ When training the network our algorithm picks the next training example. It then steps through each layer in the network.
  - ▶ In each layer it generates a random probability for each node 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 \(2012\)](#)

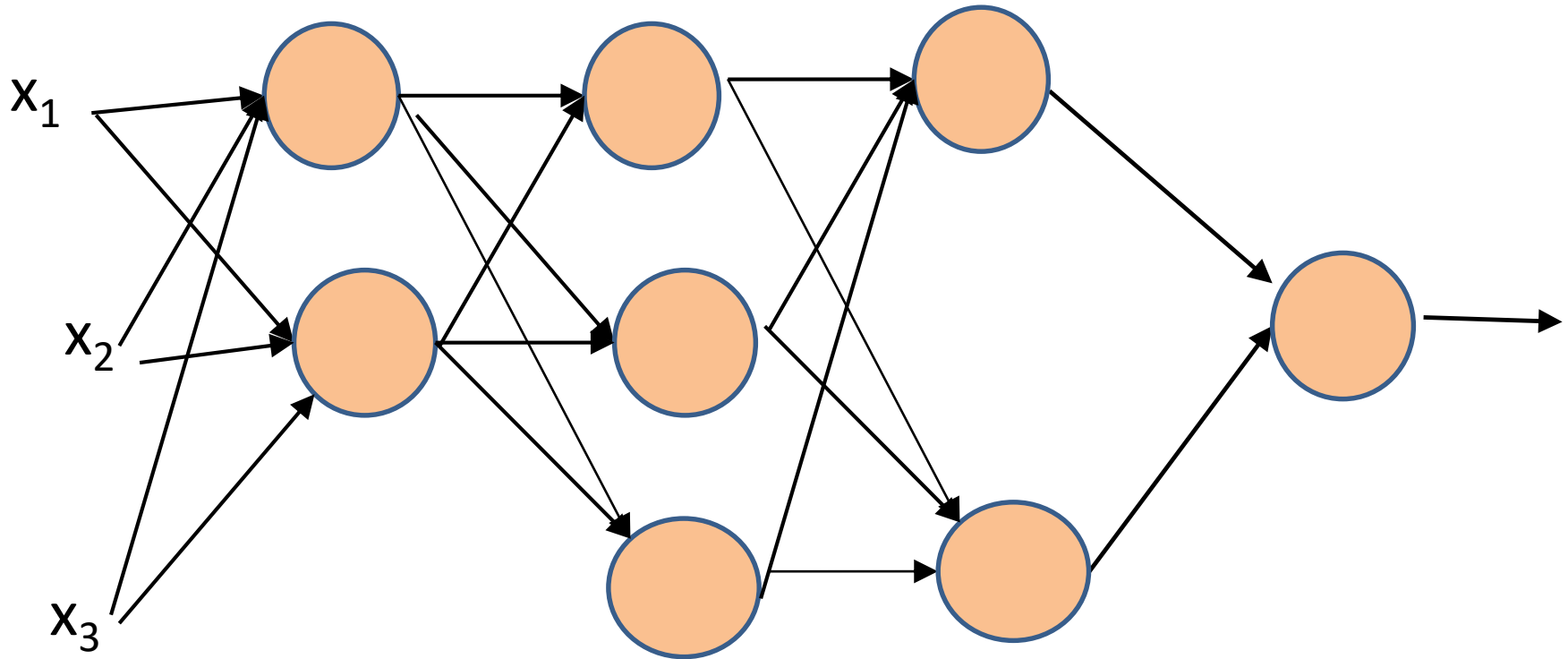
In this example, let's assume the drop probability is 0.3.



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



Sometimes 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).

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

# 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.

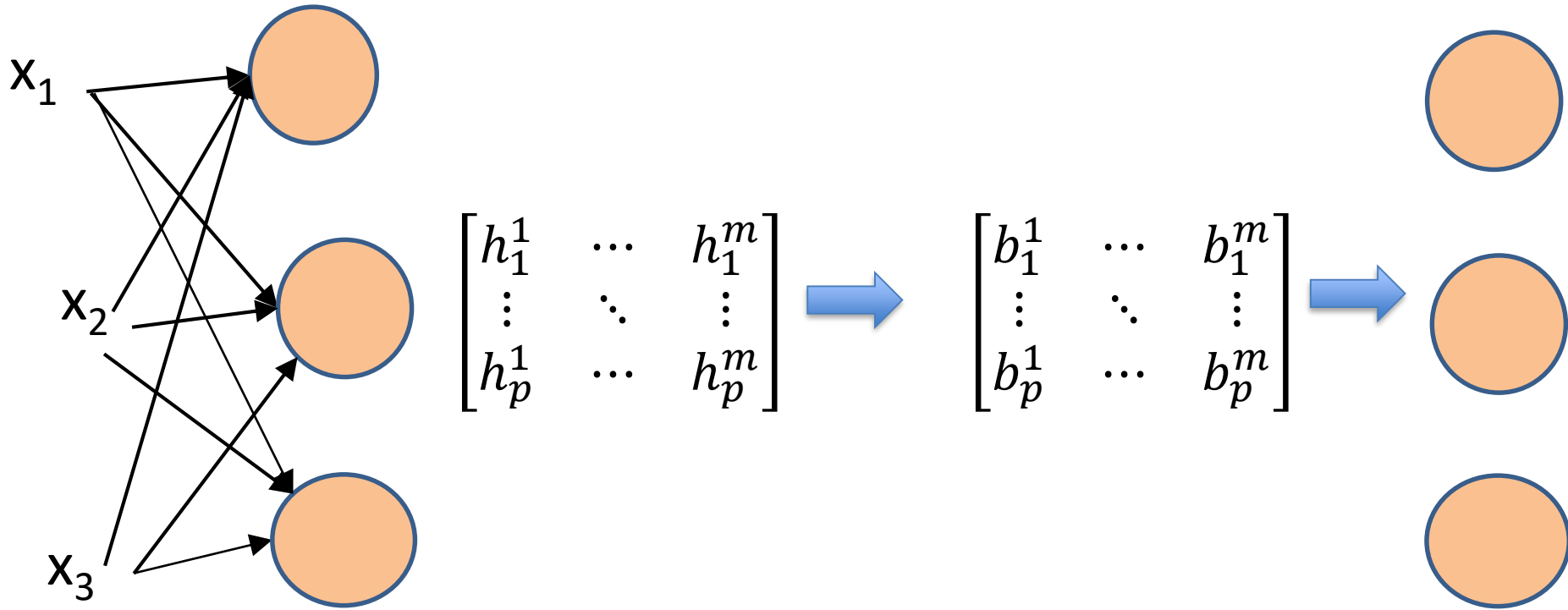

$$\begin{bmatrix} h_1^1 & \cdots & h_1^m \\ \vdots & \ddots & \vdots \\ h_p^1 & \cdots & h_p^m \end{bmatrix}$$



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 reducing 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] * dropMatrix
```

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

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

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

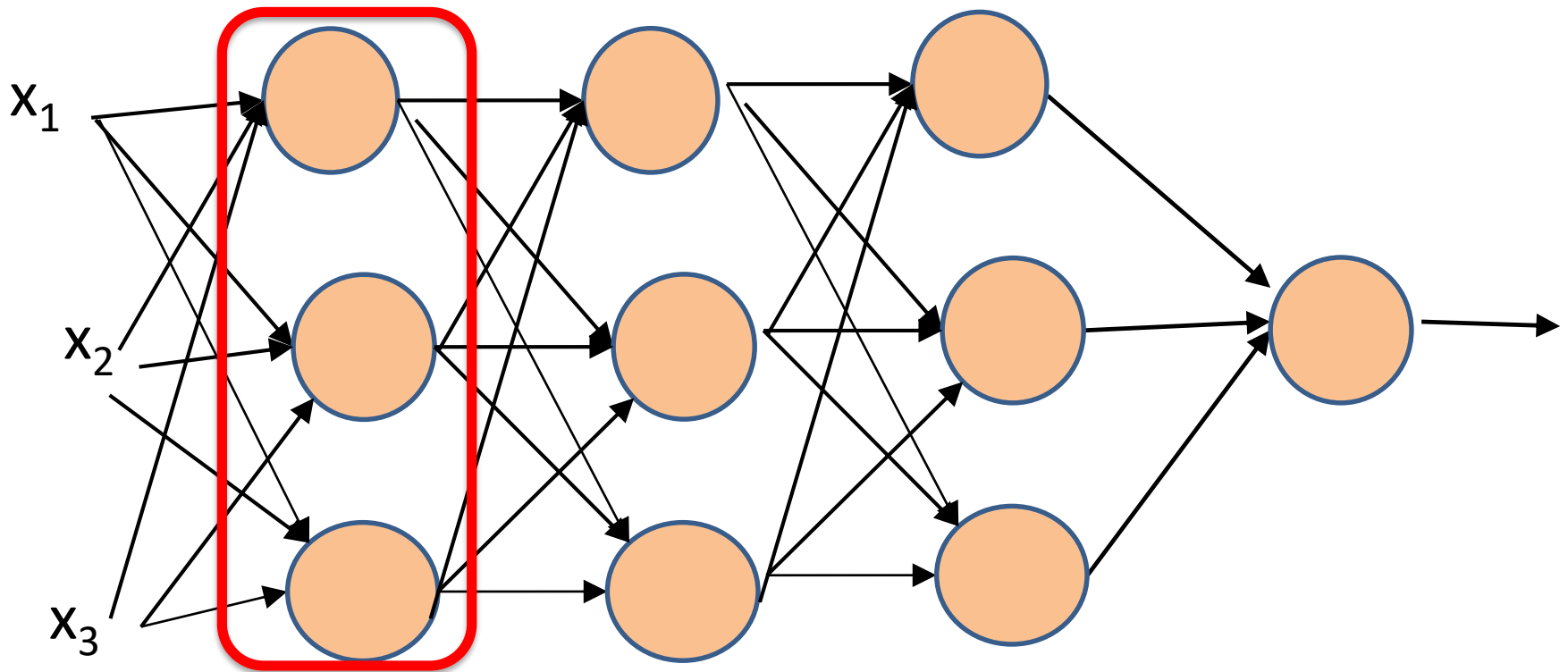
$A^{[L+1]} = W^{[L+1]} . H^{[L]} + b^{[L+1]}$

...

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

- We are going to start by taking the output from the first layer ( $H^{[1]}$ ) as shown below.
- Notice we have four training instances (four columns) and three neurons in the first layer (3 rows)

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

# 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 first step involves generate an 2D array that contains randomly generated numbers between 0 and 1 (the array should have the same dimensions as  $H^{[1]}$ )
- Now we apply a relational operation, which checks to see which elements in the random array are less than the probThreshold (probThreshold = 1- dropOutProb)
- Therefore, in this problem probTheshold = 0.7

$$\begin{bmatrix} 0.12 & 0.93 & 0.14 & 0.59 \\ 0.32 & 0.05 & 0.65 & 0.8 \\ 0.5 & 0.6 & 0.17 & 0.78 \end{bmatrix} < 0.7 = \begin{bmatrix} T & \mathbf{F} & T & T \\ T & T & T & \mathbf{F} \\ \mathbf{F} & T & T & \mathbf{F} \end{bmatrix}$$

Array with randomly generated values between 0 and 1 (same dimensions as  $H^{[1]}$ )

# Dropout Example

- Next we multiply  $H^{[1]}$  by the dropout Boolean matrix.

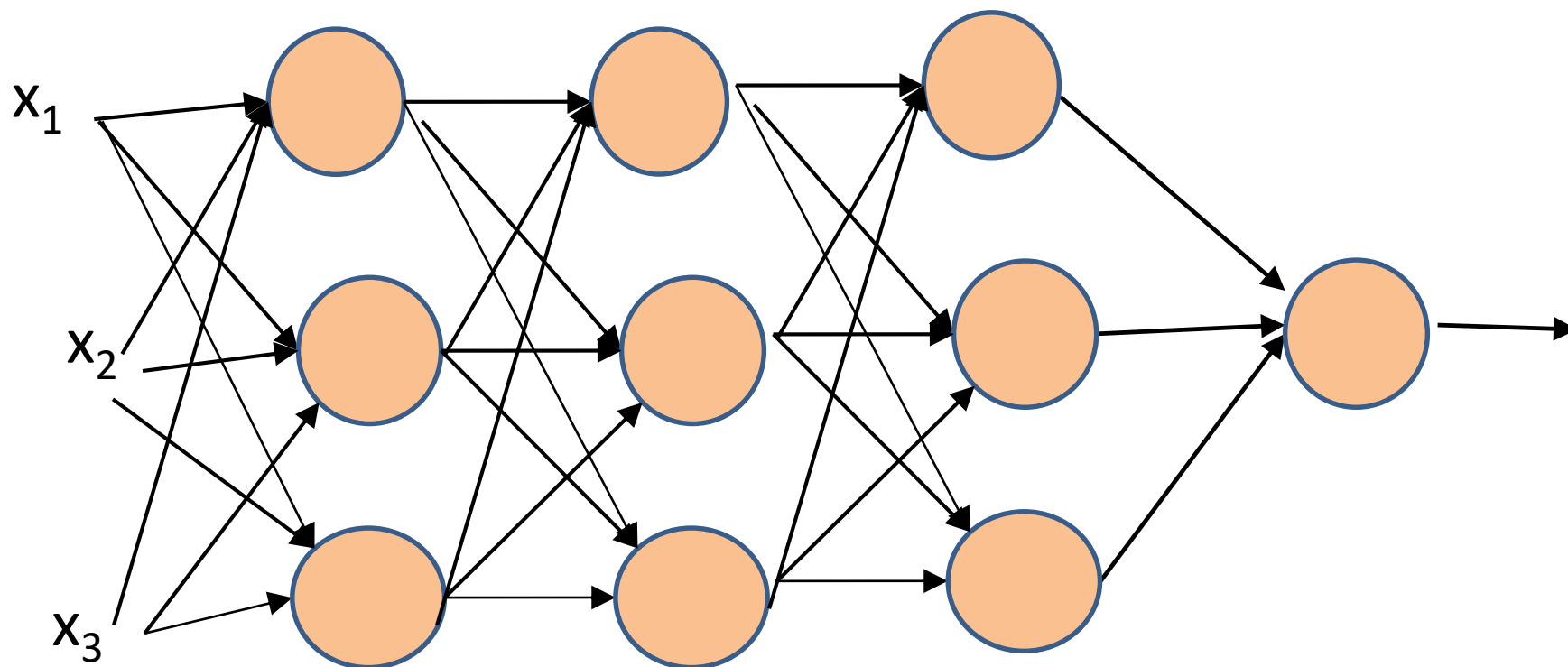
$$H^{[1]} = \begin{bmatrix} 0.1 & 0.2 & 0.4 & 0.5 \\ 0.2 & 0.05 & 0.1 & 0.4 \\ 0.25 & 0.1 & 0.6 & 0.1 \end{bmatrix} = \begin{bmatrix} T & \mathbf{F} & T & T \\ T & T & T & \mathbf{F} \\ \mathbf{F} & T & T & \mathbf{F} \end{bmatrix}$$

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

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

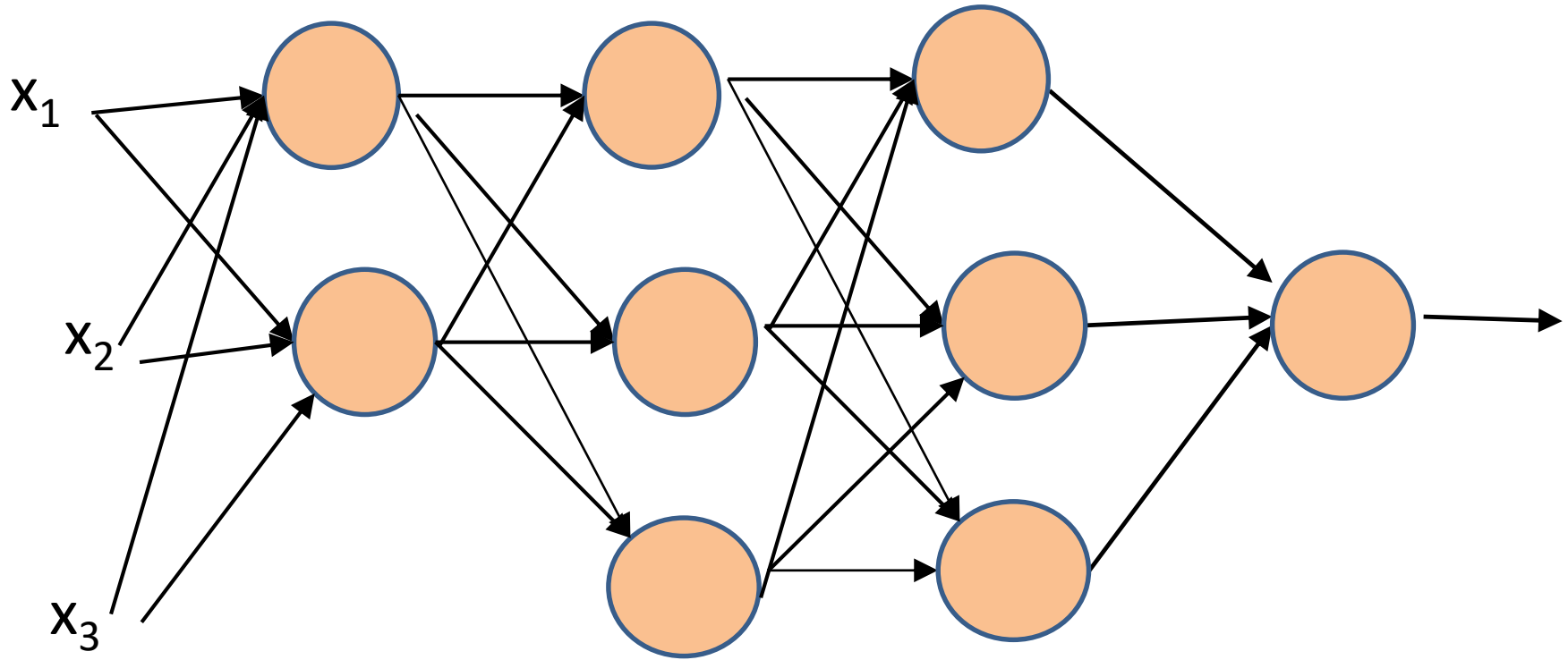
$$H^{[1]} = \begin{bmatrix} 0.1/0.7 & \mathbf{0}/0.7 & 0.4/0.7 & 0.5/0.7 \\ 0.2/0.7 & 0.05/0.7 & 0.1/0.7 & \mathbf{0}/0.7 \\ \mathbf{0}/0.7 & 0.1/0.7 & 0.6/0.7 & \mathbf{0}/0.7 \end{bmatrix} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85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





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

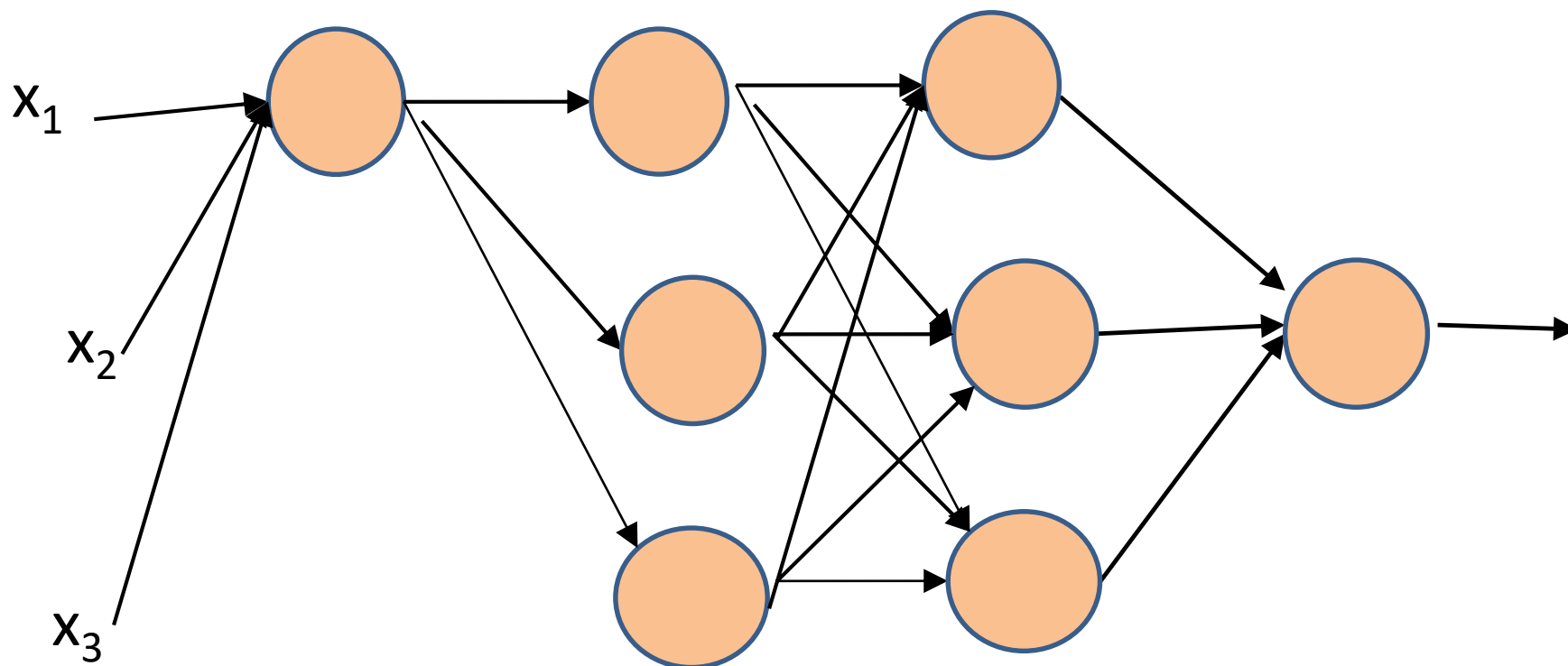
Let's look at example 1. Notice it will have no output from the third neuron.



↓

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

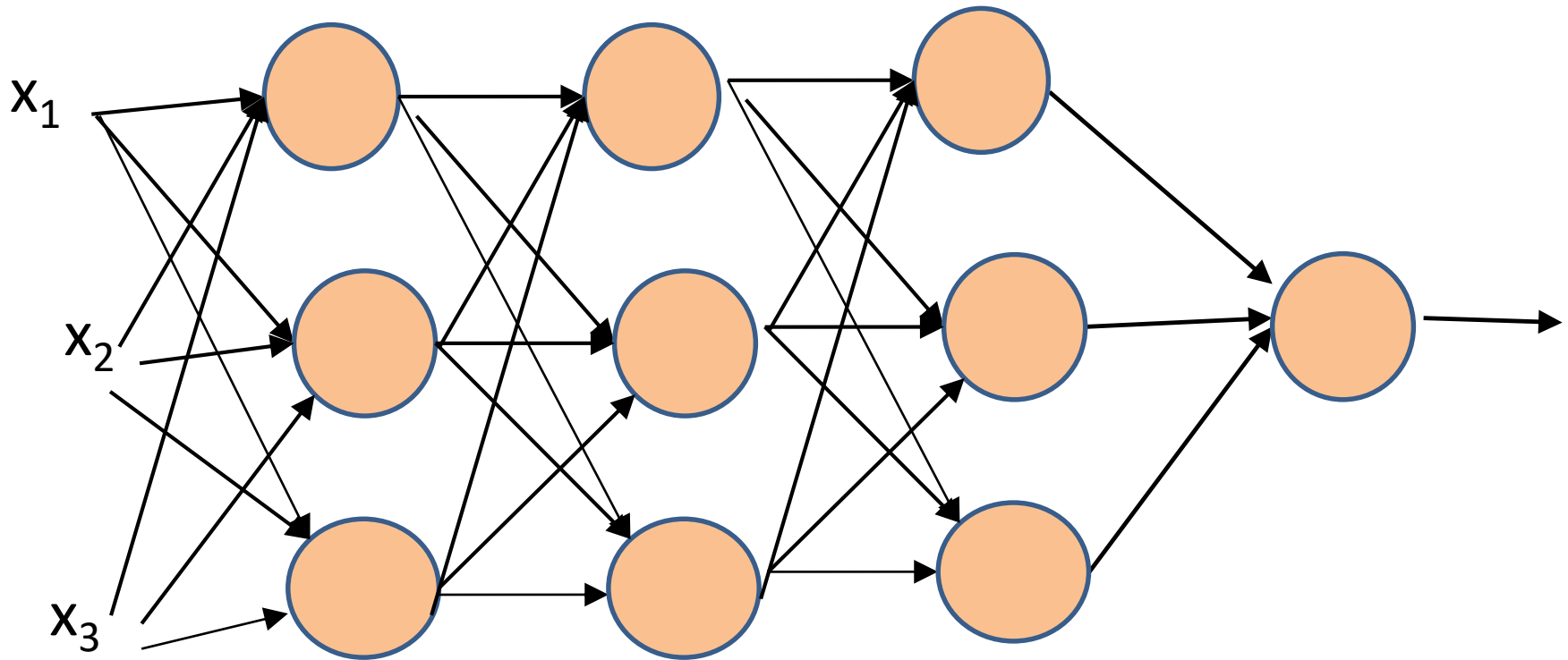
Let's look at training example 1.  
Notice it will have no output  
from the third neuron.



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

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 layer. Once we have calculated the output of the second layer  $H^{[2]}$  then we can repeat the same process again and apply dropout again.



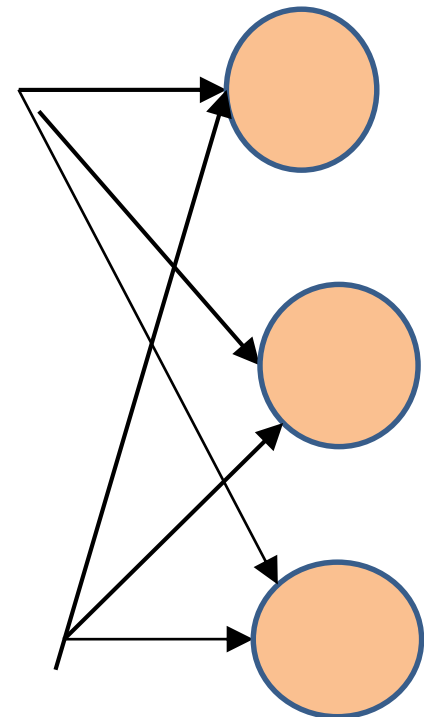
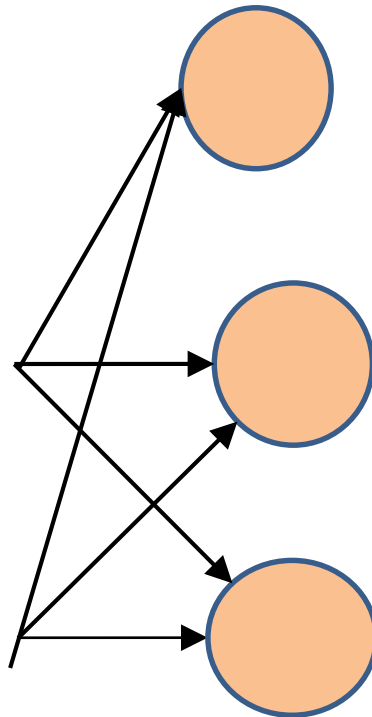
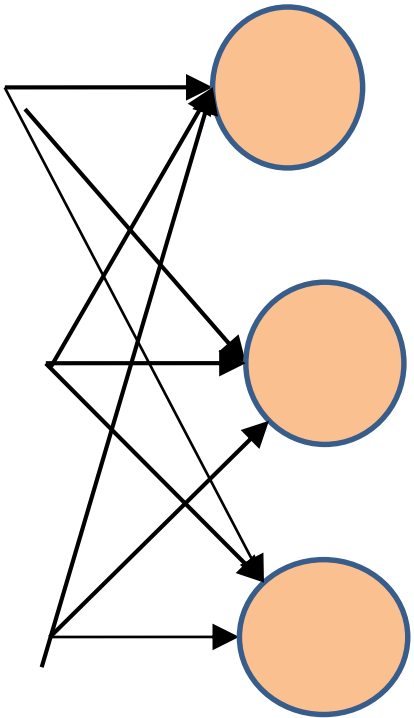
$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

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

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

# 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.



# Observations

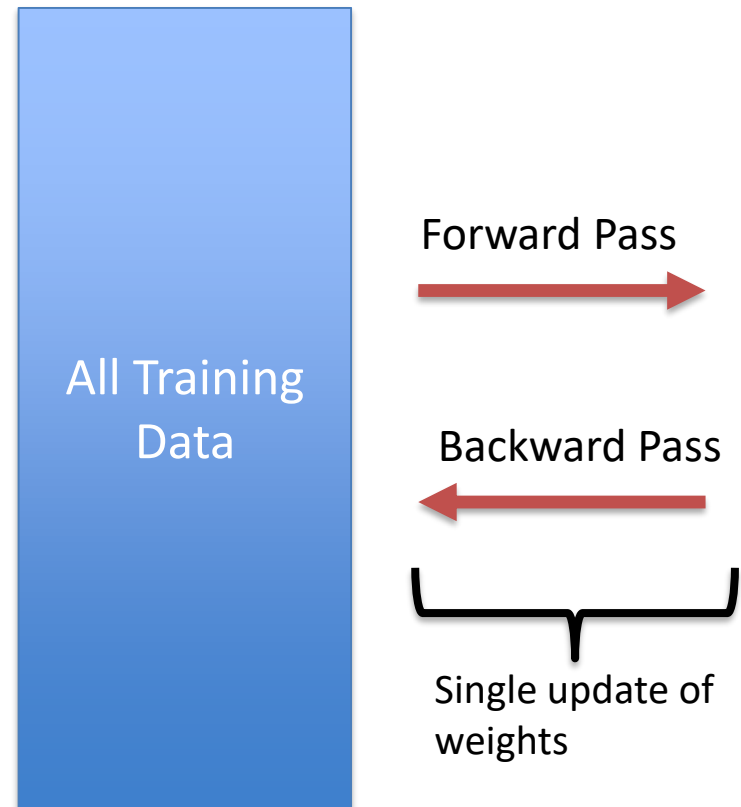
- ▶ Of course the consequence of using dropout is that we now have yet another **hyper-parameters** to estimate for our neural network.
  - ▶ The appropriate range of values is an open question and can vary depending on the type of network you are training. For example, Hinton's original paper used a 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)

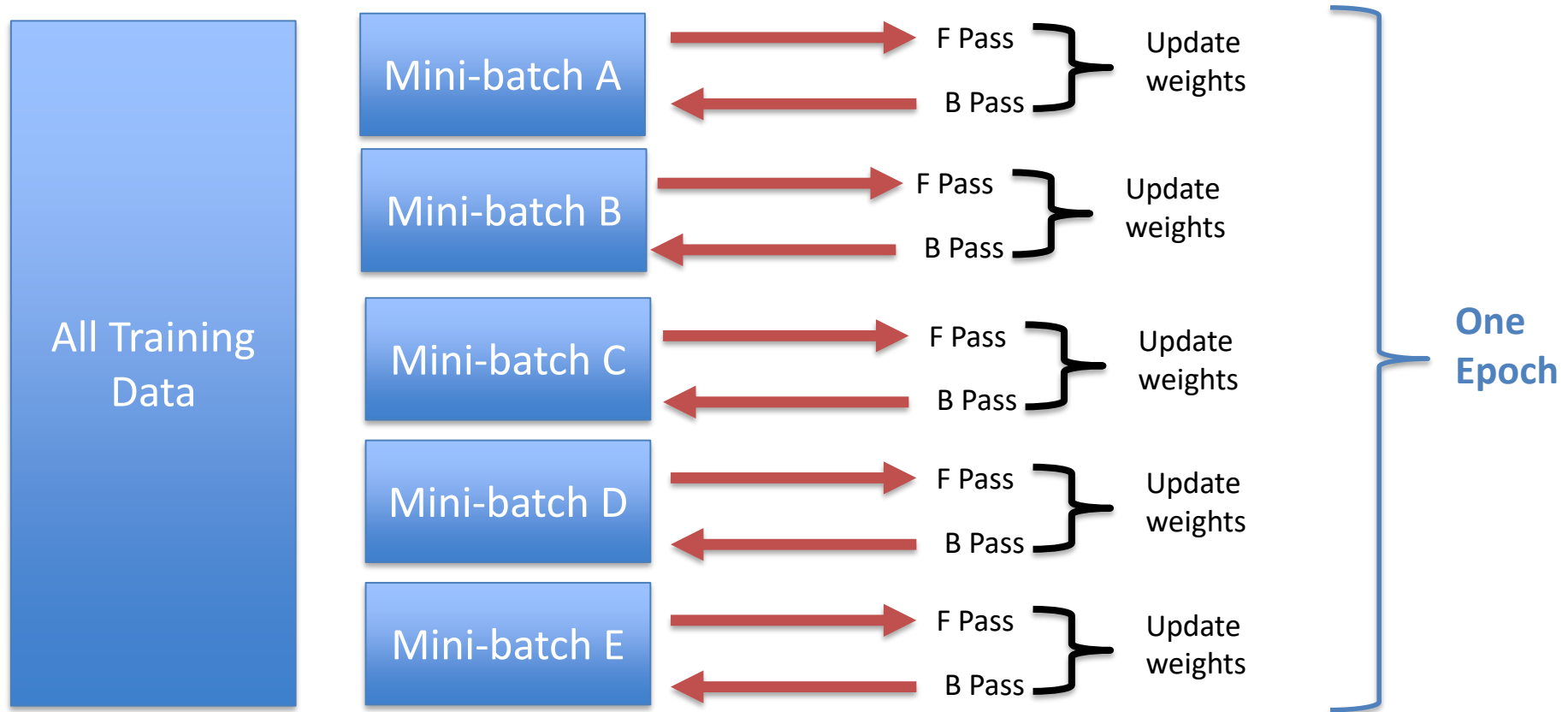




# Batch v's Mini-Batch Gradient Descent

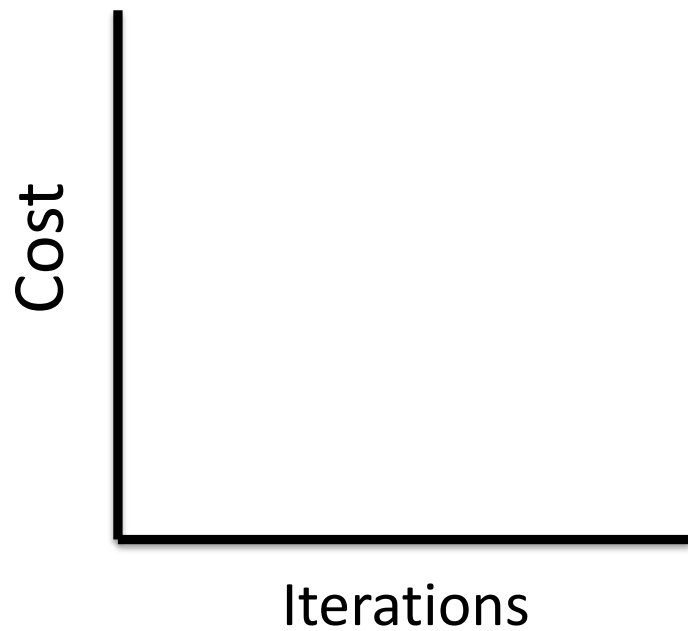
- ▶ An alternative approach that significantly alleviates the problem outlined in the previous slide is to **break-up the training data** into exclusive partitions, referred to as **mini-batches**.
- ▶ For example, we might break our **4M** training examples in **8000 mini-batches** with **500 training 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

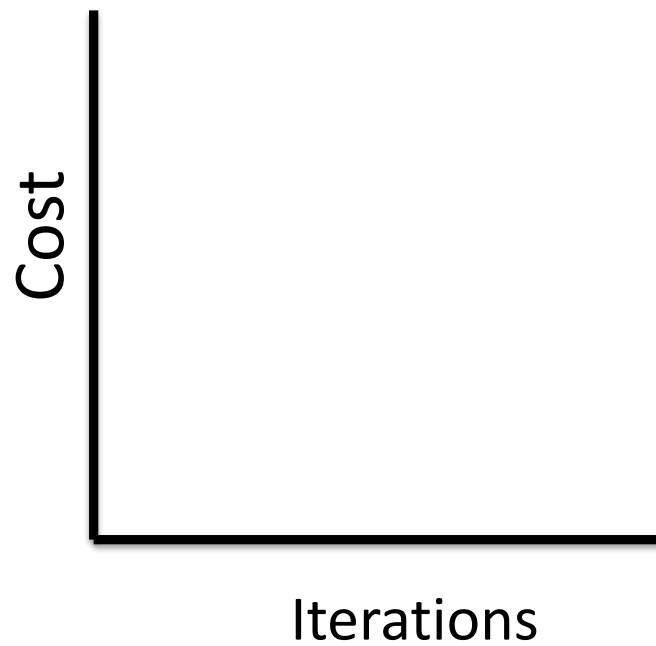


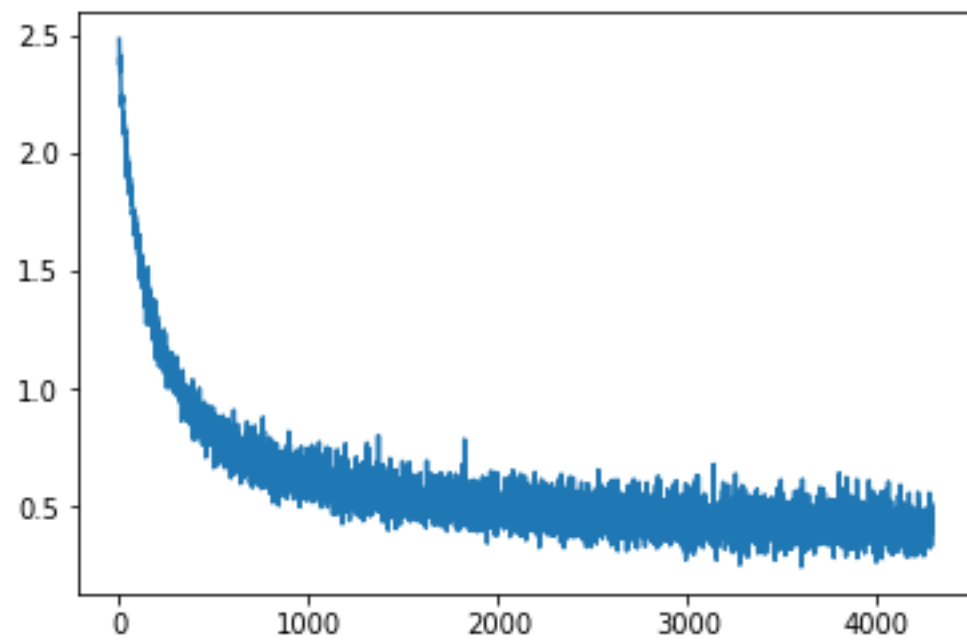
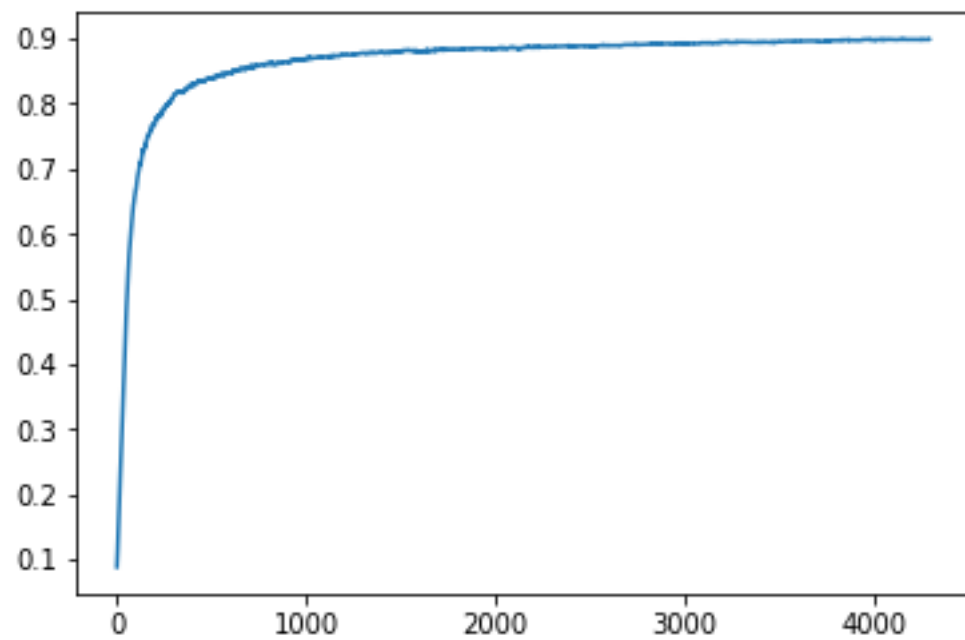
- ▶ The main difference is that when using mini-batch, after one epoch the weights will have been updated mini-batch number of times ... in our example the weights would have been updated 8000 times.
- ▶ However, after a single epoch with batch gradient descent the weight have only been updated once.

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 drawback 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.