

Machine Learning



Machine Learning

Lecture: Neural Networks

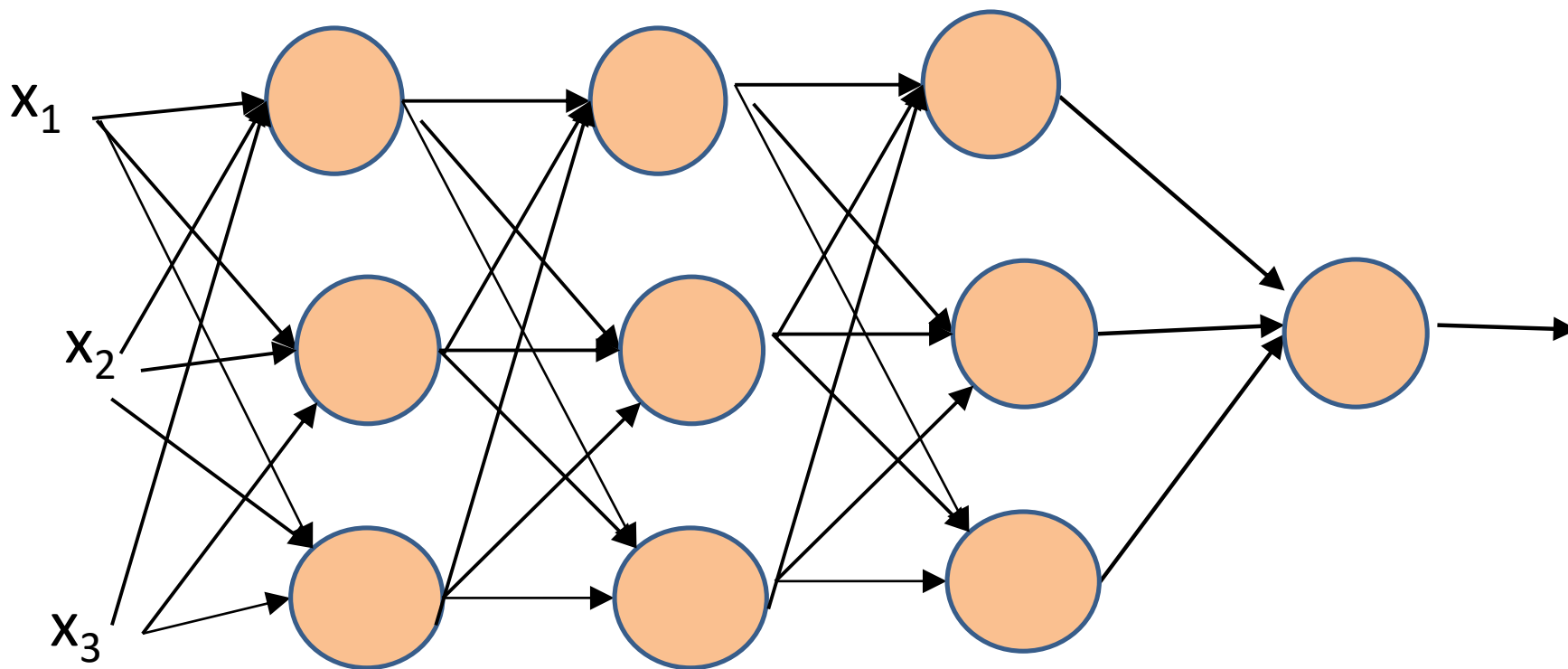
Ted Scully

Dropout Regularization

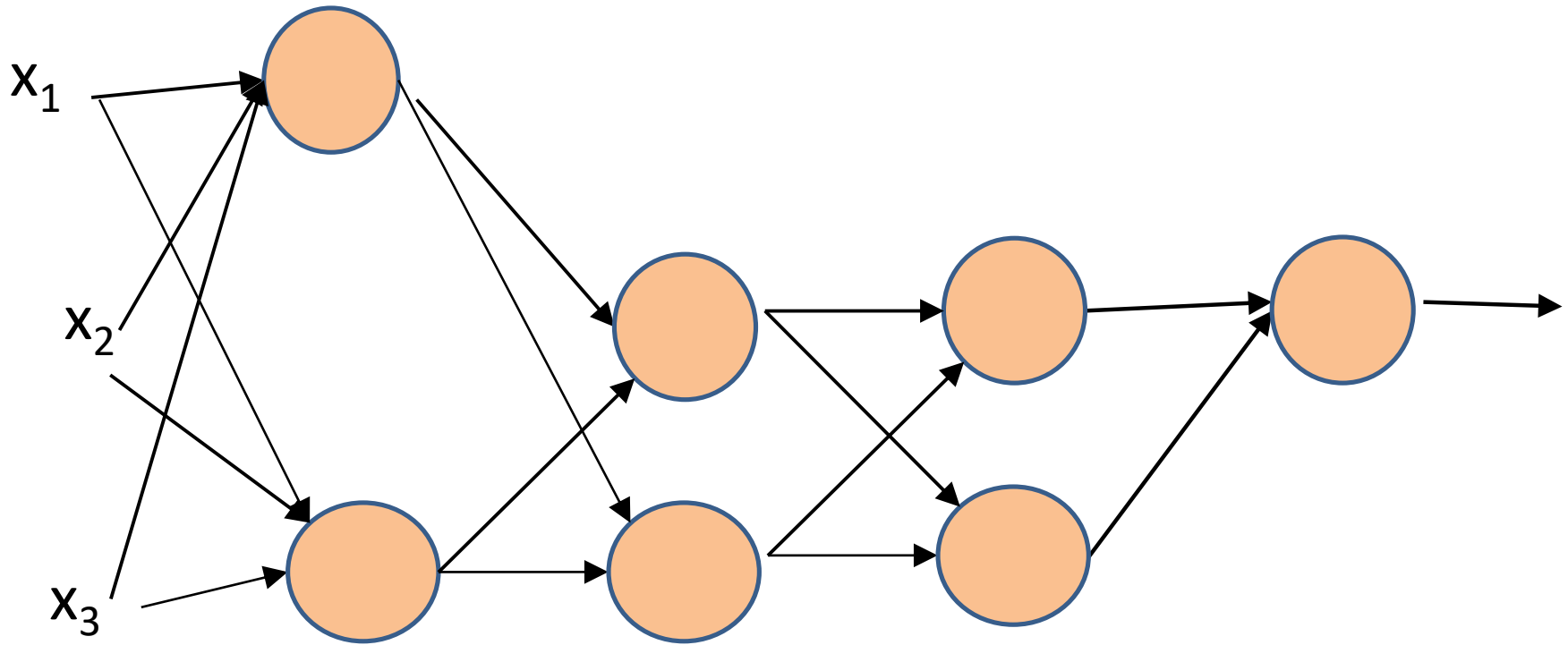
- ▶ Drop-out regularization associates a specific drop-out probability with each layer of a neural network.
- ▶ When training the network our algorithm picks the **next training example**. It then steps through each layer in the network.
 - ▶ In each layer it generates a **random probability** for each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

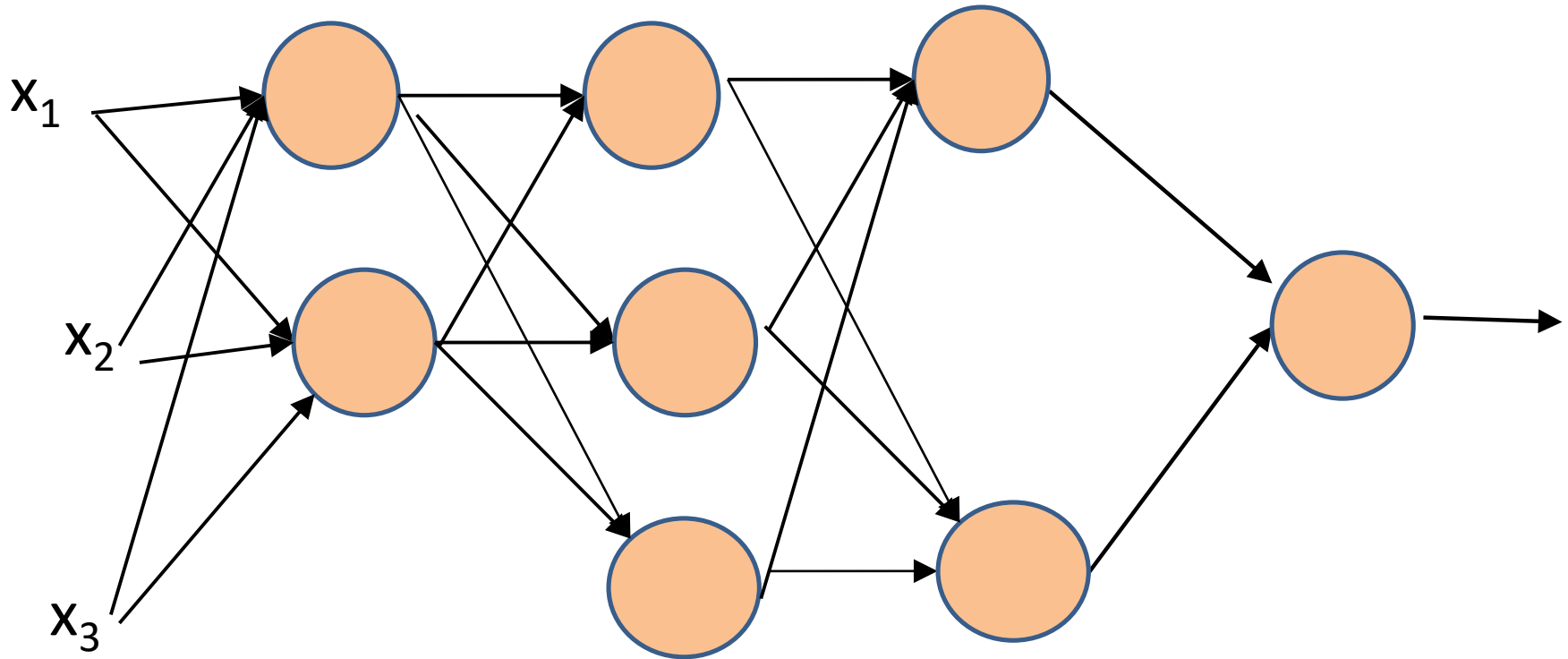
In this example, let's assume the drop probability is 0.3.



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




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.

Reminder 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.

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

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

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

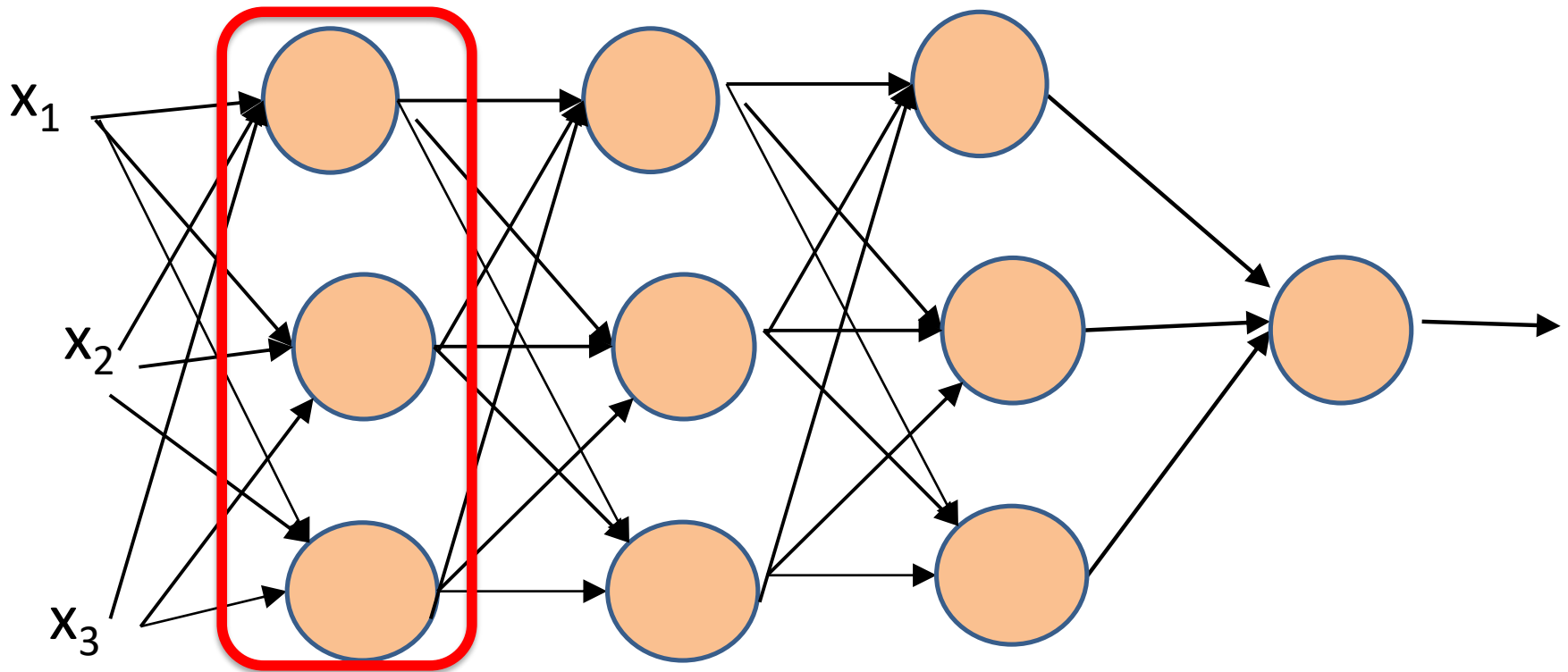
$A^{[L+1]} = W^{[L+1]} . H^{[L]} + b^{[L+1]}$

...

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

- 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.

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

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

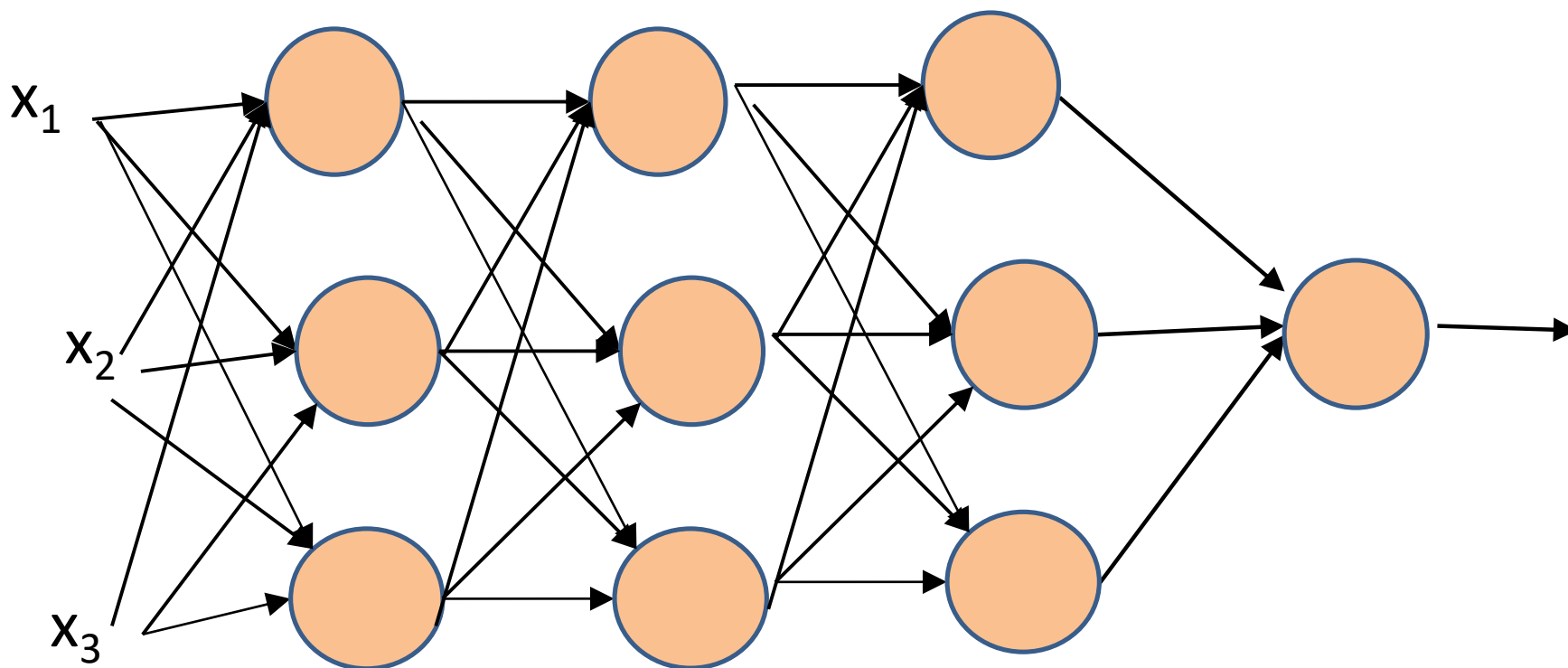
Dropout Example

- Finally we need to rescale by dividing by the 1- dropout probability.

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

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

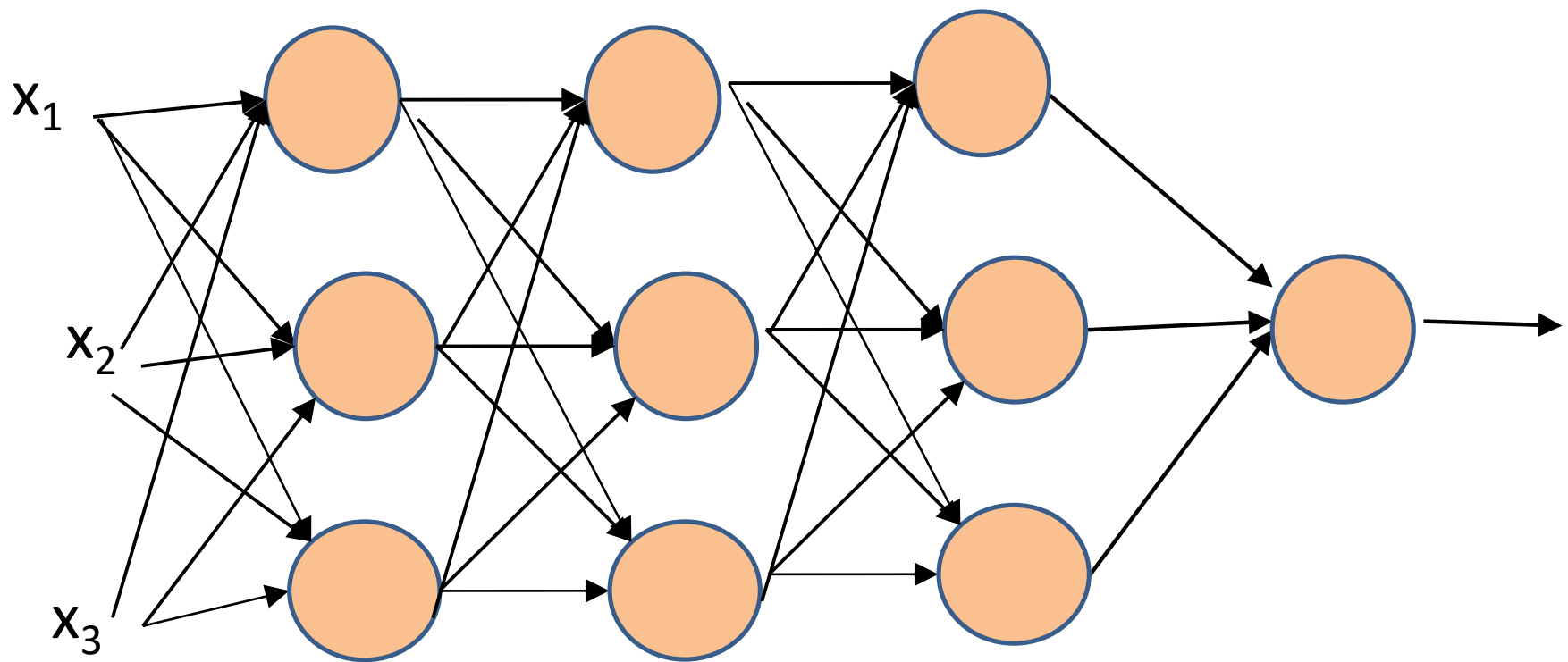
$$H^{[1]} = \begin{bmatrix} 0.1/0.7 & \mathbf{0}/0.7 & 0.4/0.7 & 0.5/0.7 \\ 0.2/0.7 & 0.05/0.7 & 0.1/0.7 & \mathbf{0}/0.7 \\ \mathbf{0}/0.7 & 0.1/0.7 & 0.6/0.7 & \mathbf{0}/0.7 \end{bmatrix} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85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



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

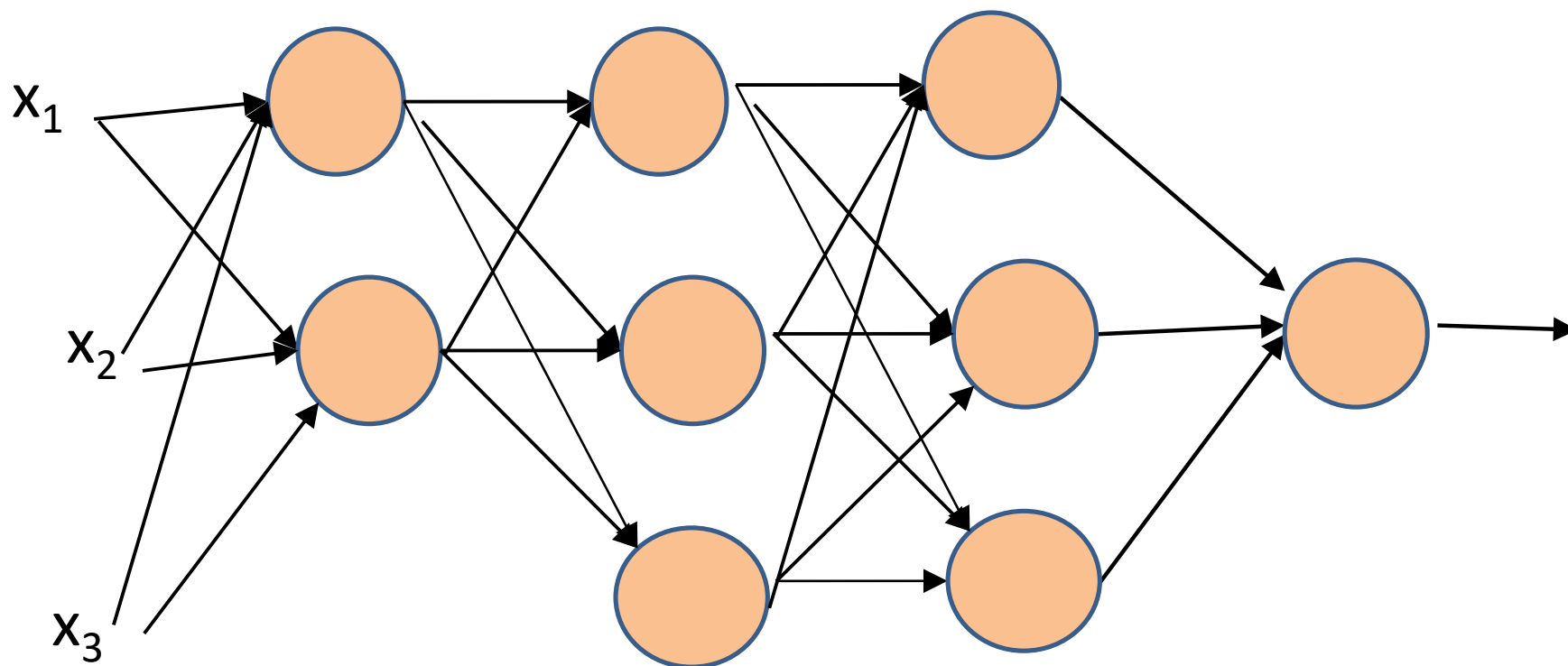
$$A^{[2]} = w^{[2]} H^{[1]} + b^{[2]}$$

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



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

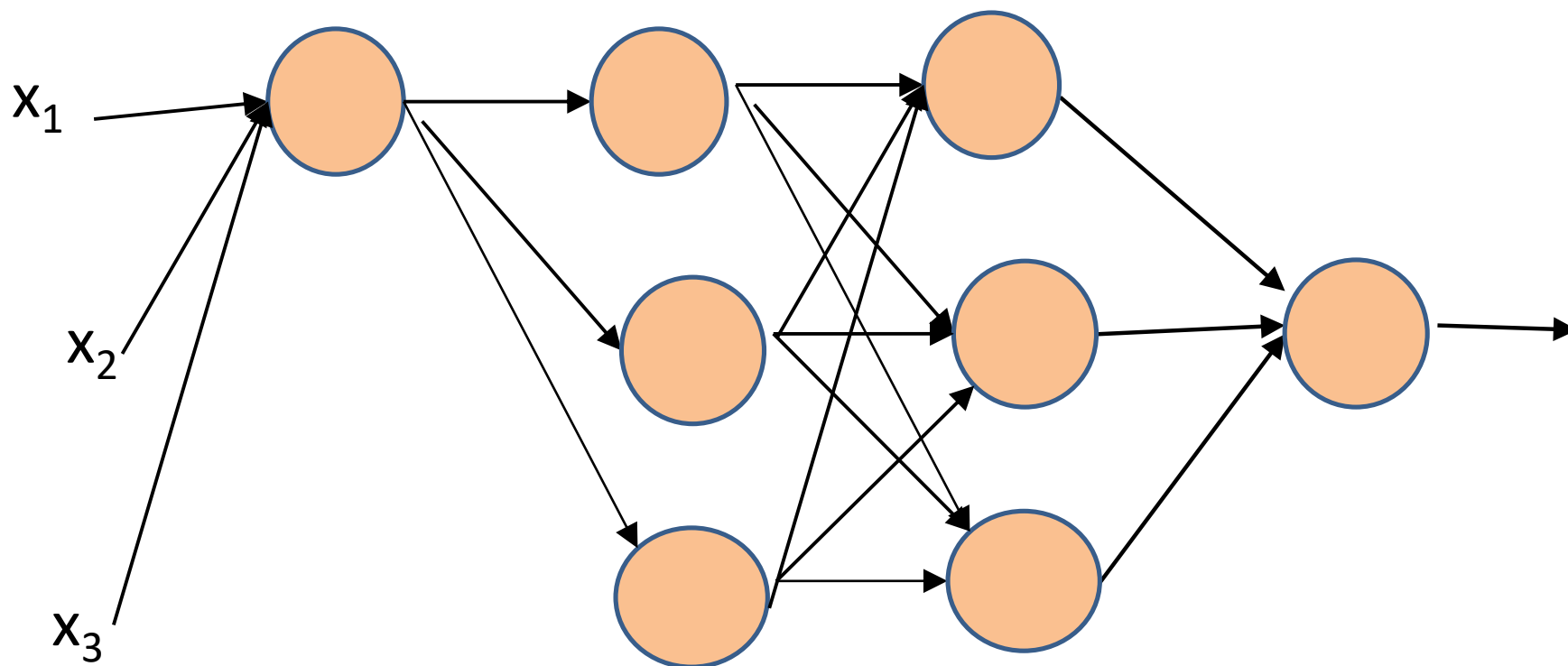
Let's look at example 1. Notice it will have no output from the their neuron.



↓

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

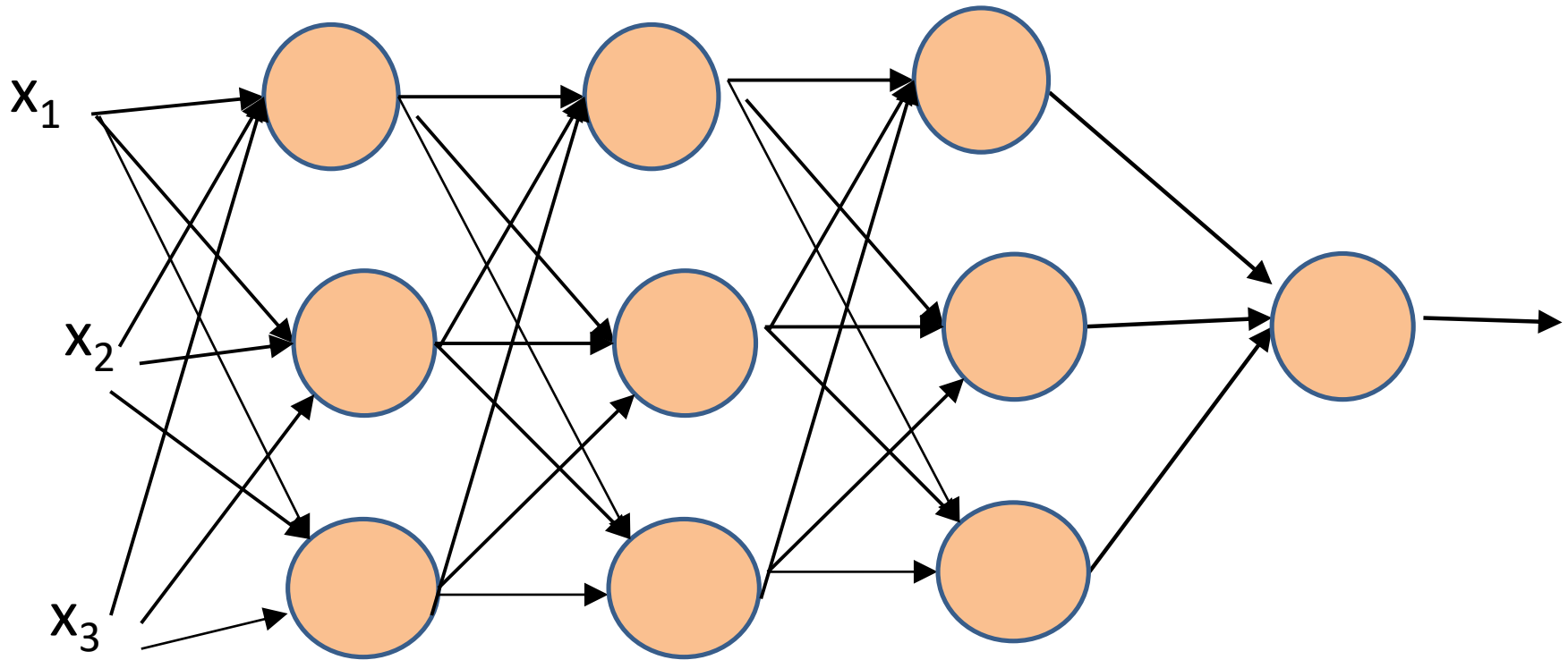
Let's look at training example 1.
Notice it will have no output
from the third neuron.



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

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.



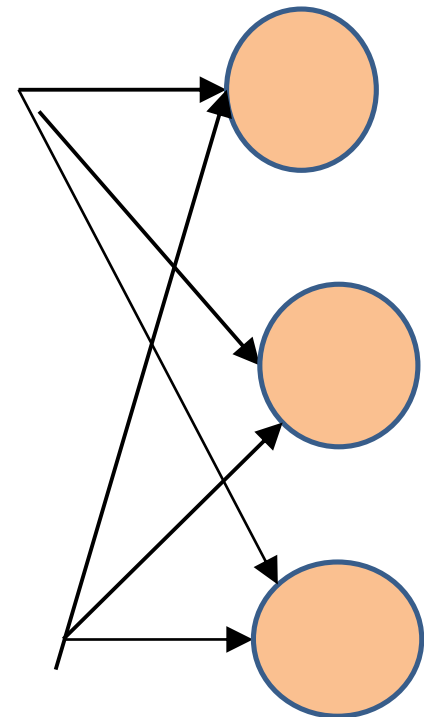
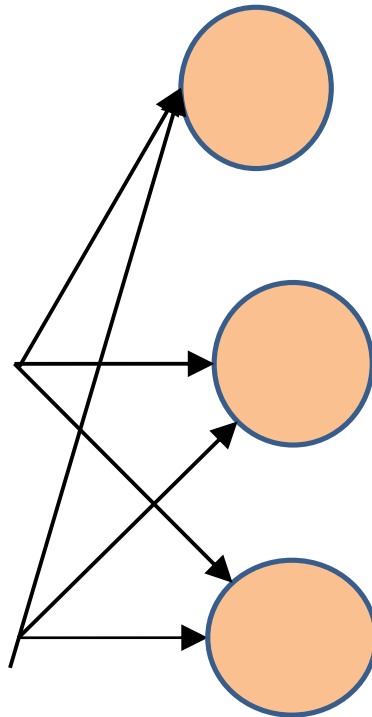
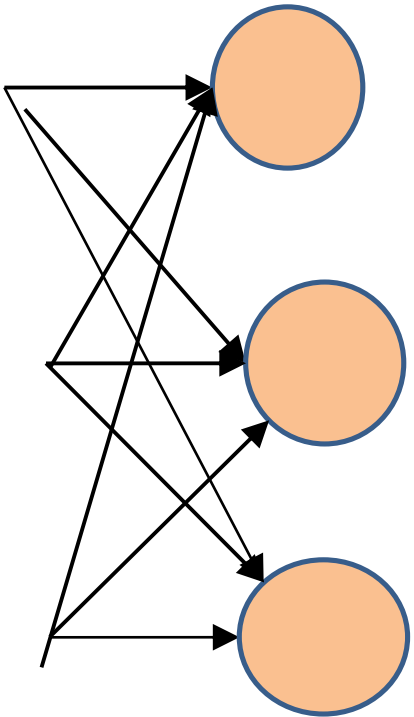
$$H^{[1]} = \begin{bmatrix} 0.14 & \mathbf{0} & 0.57 & 0.71 \\ 0.28 & 0.07 & 0.14 & \mathbf{0} \\ \mathbf{0} & 0.14 & 0.85 & \mathbf{0} \end{bmatrix}$$

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

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

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**.
- ▶ Normally you apply drop-out or other regularization techniques such as L2 if there is **evidence of overfitting**.
- ▶ Another important consideration with dropout is that our **cost function may not always be decreasing** with every iteration.

Neural Networks

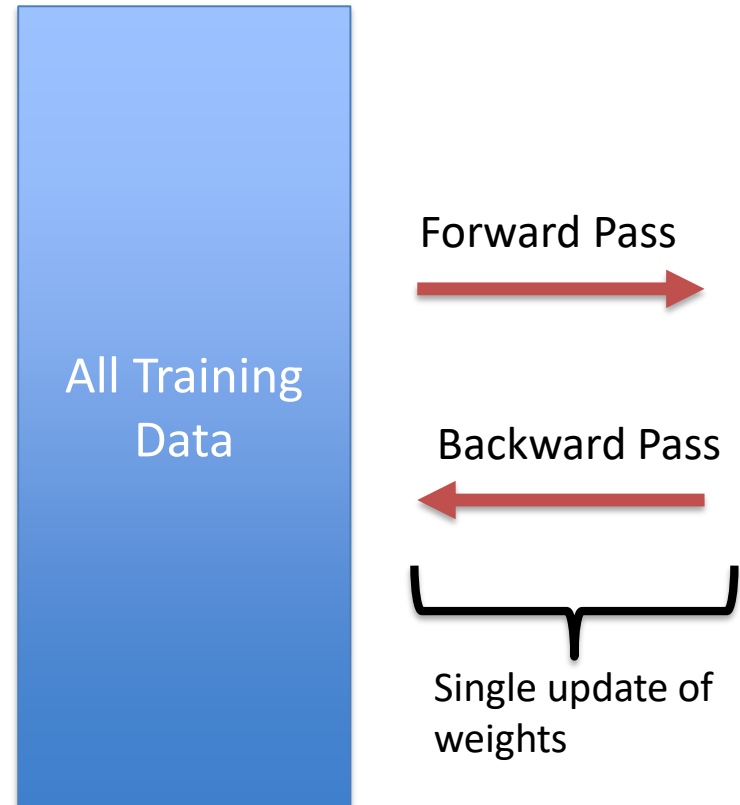
1. Neural Networks and Activation Functions
2. Vectorized Neural Networks
3. Softmax Activation Layer
4. Regularization and Dropout
5. Optimization

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

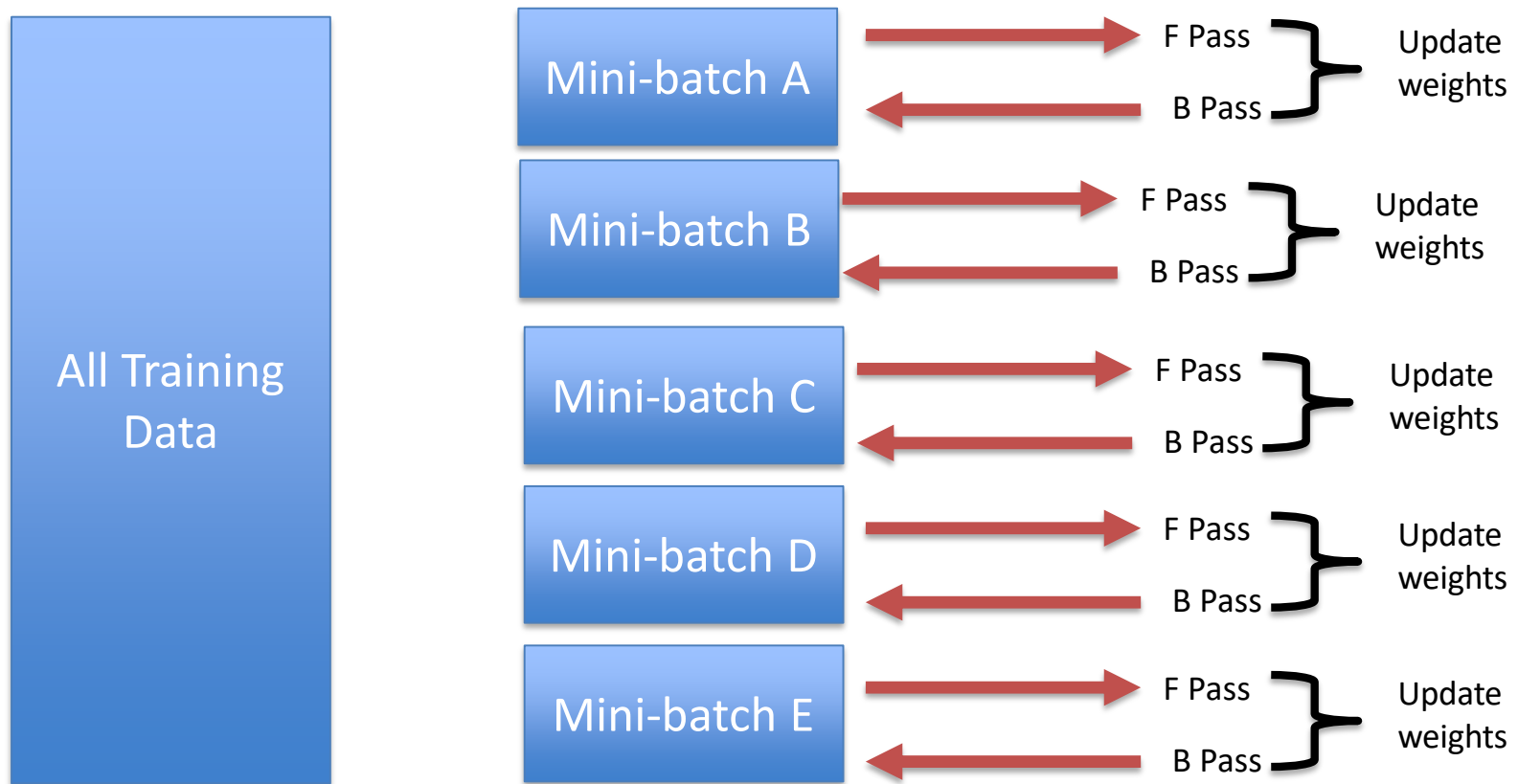
- ▶ Consider a situation where you have a very large number of train examples (**4,000,000**).
- ▶ The issue that arises in such scenarios is that we must push all 4M data points through our graph in order to determine a single update for our weights.
- ▶ This can take a very long time and can mean that training the model takes a very very long time. Given that the process of 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)



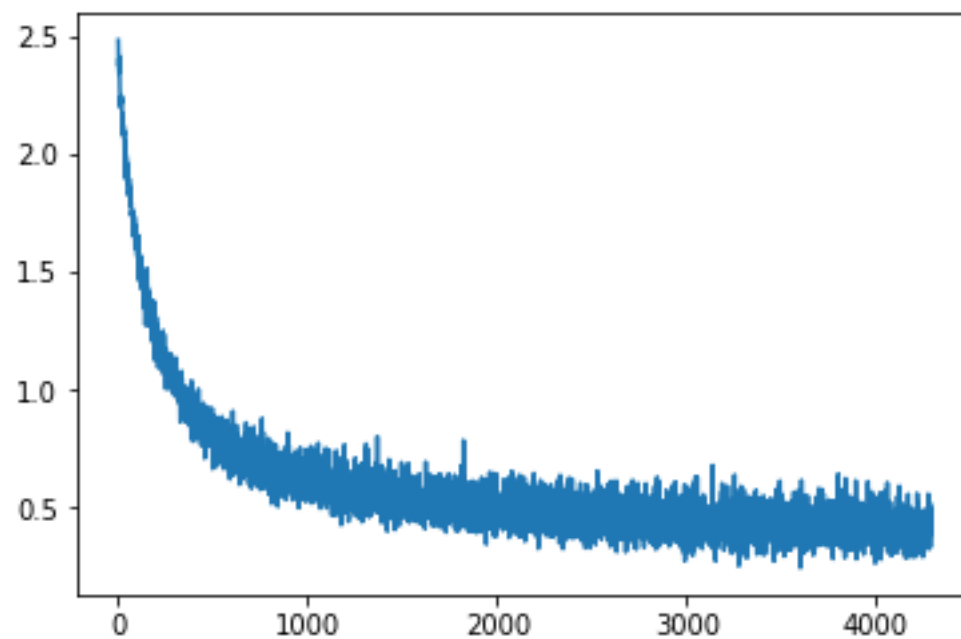
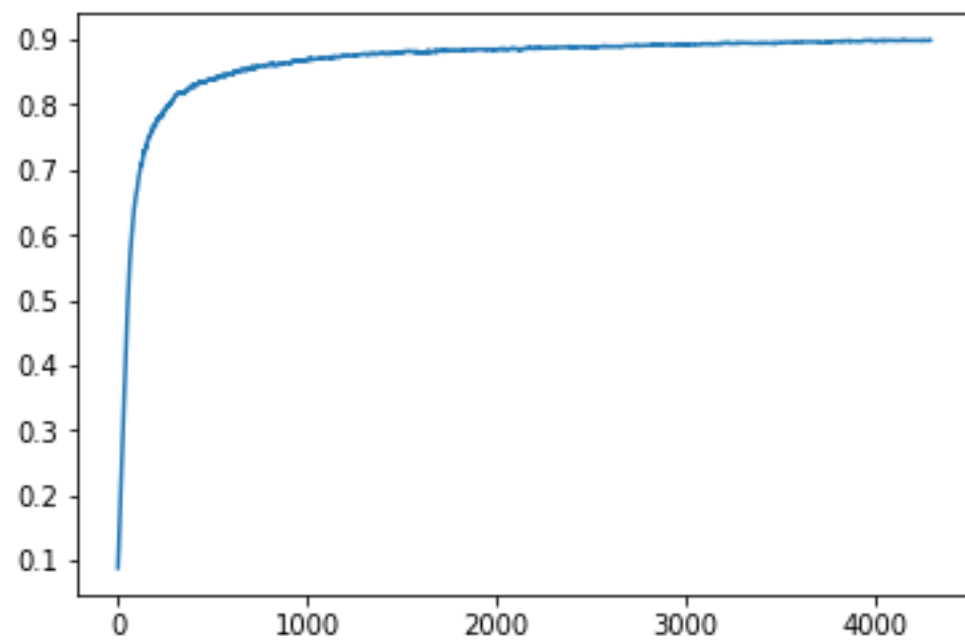
Batch v's Mini-Batch Gradient Descent

- ▶ 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.

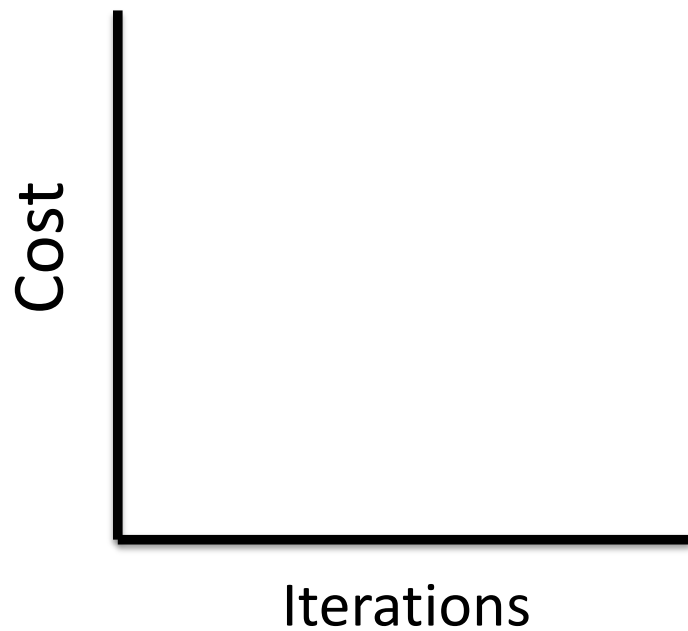
Batch v's Mini-Batch Gradient Descent



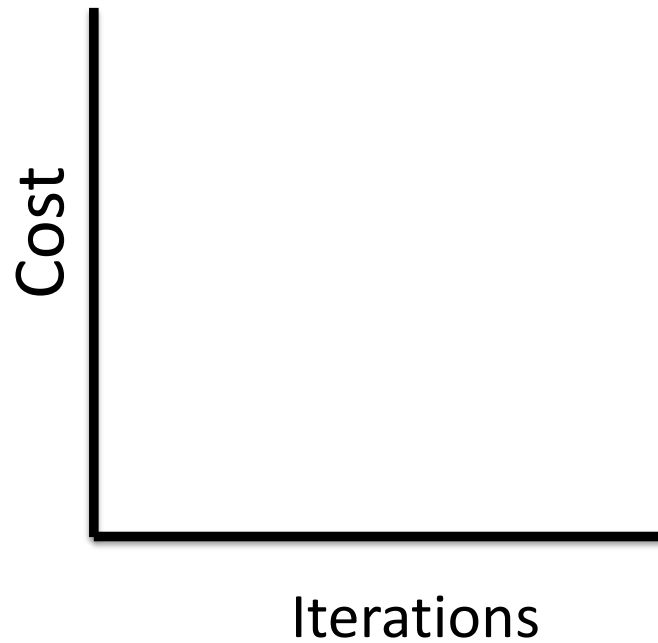
- ▶ The main difference is that when using mini-batch, after one epoch the weights will have been updated mini-batch number of times ... in our example the weights would have been updated 8000 times.
- ▶ However, after a single epoch with batch gradient descent the weight have only been updated once.



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

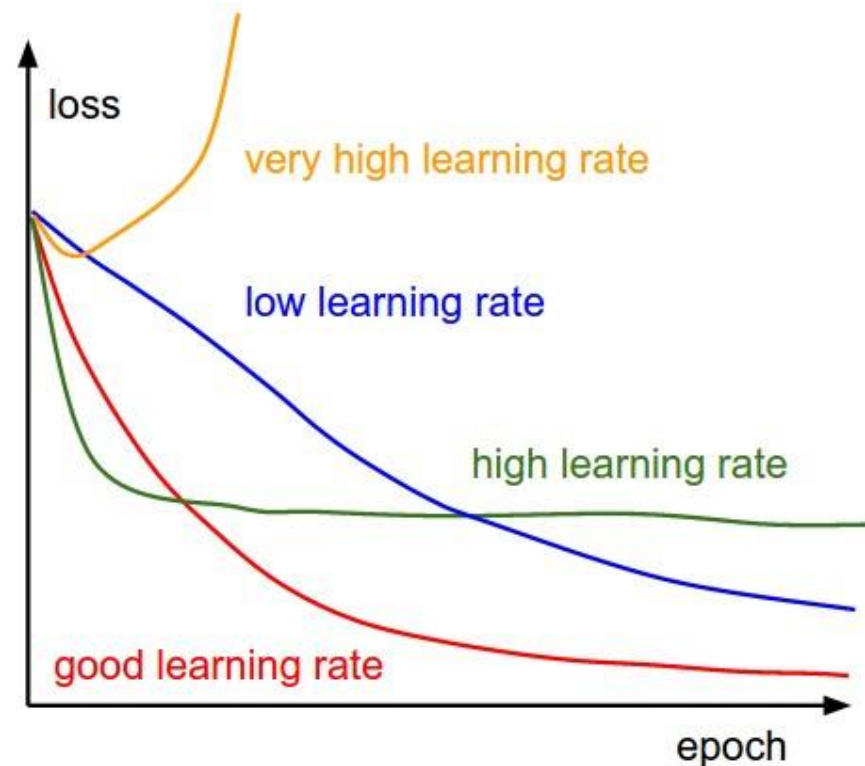
- ▶ If ***batch size*** > 1 and $< m$ then we are using what is called mini-batch gradient descent.
 - ▶ Typically sizes for mini-batch is 32, 64, 128, 256, 512, 1024.
 - ▶ The advantage of this technique is that we can achieve a relatively high frequency of weight updates while still retain the benefit 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.

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

Learning Rate Decay

- ▶ Learning Rate Decay is a simple concept that gradually reduces the learning rate (α) as we iterate through gradient descent.
- ▶ High learning rates may reduce the loss quickly, but they can often get stuck at values of loss (green line). They end up jumping around the minimum rather than converging properly.
- ▶ The idea behind learning rate decay is that we want to initially move quickly in the direction of the true minimum but as we approach we want to gradually reduce the learning rate to ensure smaller steps and better convergence.



Learning Rate Decay – Step Decay

- ▶ There are a range of methods that can be adopted for implementing learning rate decay.
- ▶ One of the simplest and most effective is called **step decay**.
- ▶ In step decay you reduce the learning rate by some factor every few epochs.
- ▶ For example, you might **multiply the learning rate by 0.5 after each five epochs**. Please note that the appropriate value can vary significantly from one problem to another.
- ▶ In practice one way of implementing this approach is to run your model initially with a fixed learning rate. Observe the epoch where the validation loss **stops reducing**.
- ▶ You can then apply a decay of the learning rate at this epoch. In other words you can reduce the learning rate by some value (e.g. 0.5) when the validation stops decreasing.

Learning Rate Decay

- ▶ Another common technique is an exponential rate decay.
- ▶ For each epoch we update our learning rate according to the following rule:
- ▶ $\alpha_{epoch} = \alpha_0 e^{-epoch}$

Epoch	α_{epoch}
1	0.18
2	0.06
3	0.02
4	0.009
5	0.003

In this example $\alpha_0 = 0.5$

As you can see the learning rate decreases rapidly. This rate of decay can be adjusted by adding an additional hyper-parameter h to the equation.

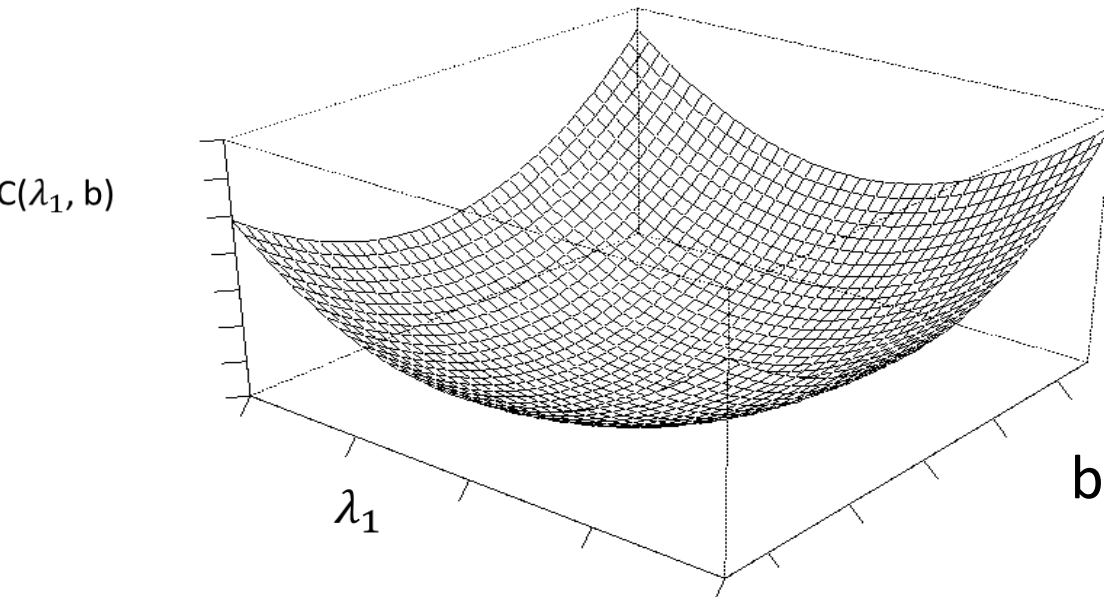
$$\alpha_{epoch} = \alpha_0 e^{-epoch * h}$$

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**

Adaptive Learning Rates

- ▶ You may notice when we use learning rate decay we adjust the learning rate. However, that single adjusted learning rate is used to update the learning rate for all training parameters.
- ▶ A preferable approach could be to employ an adaptive rate of update for each of the training parameters.
- ▶ This has been a very significant amount of research work invested in this area over the last number of years.

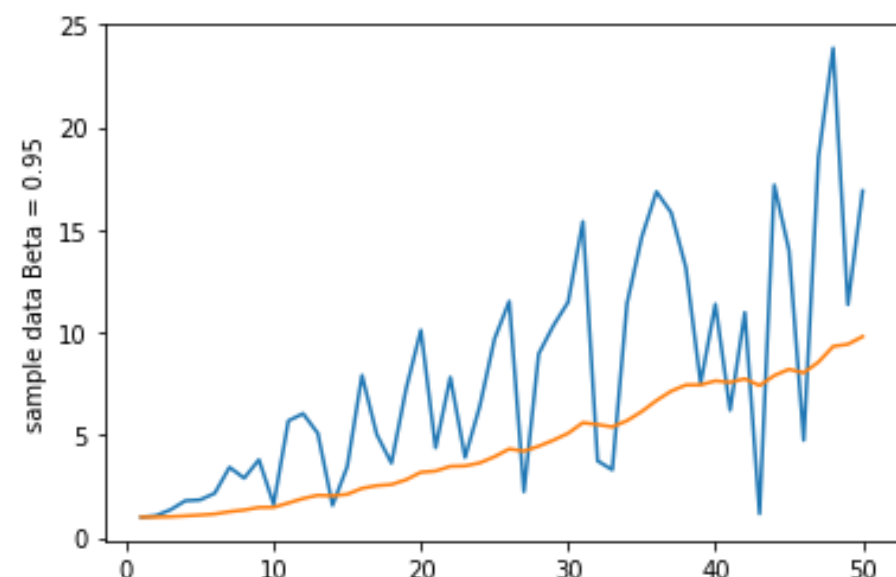
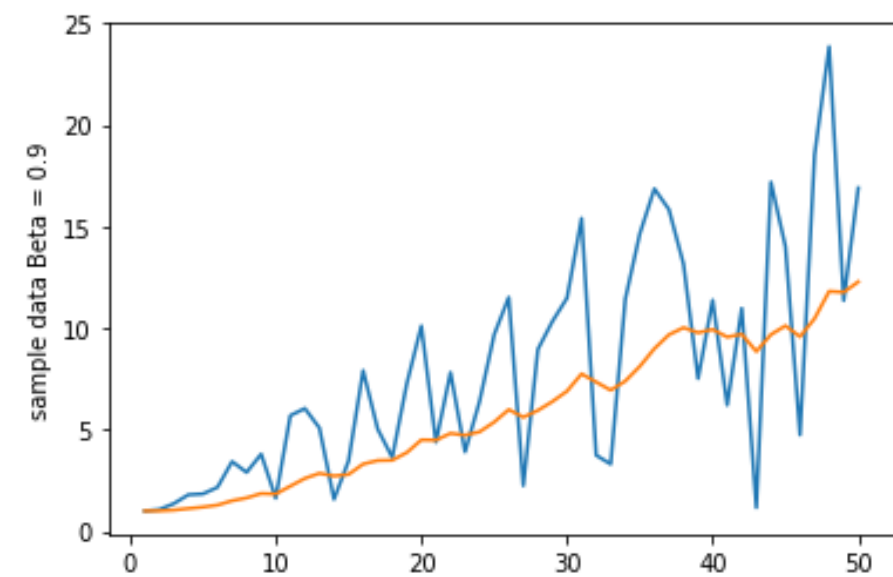
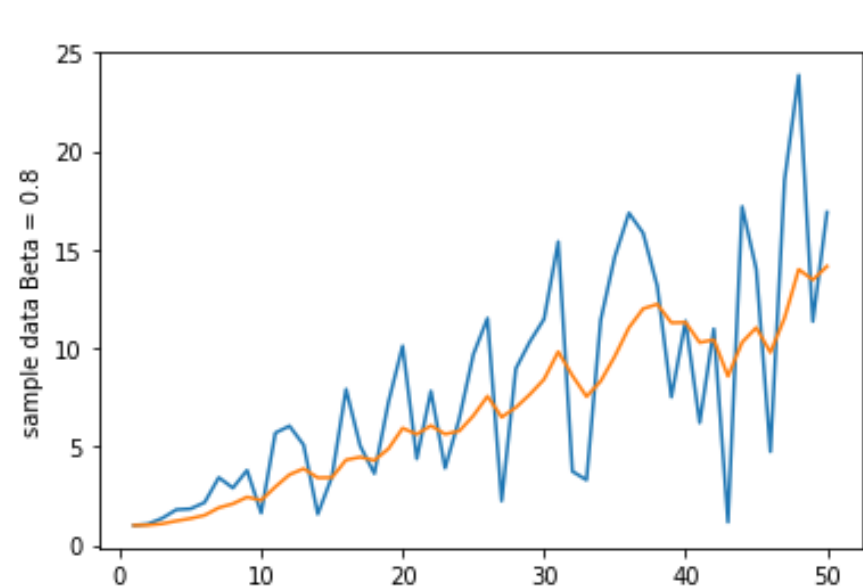
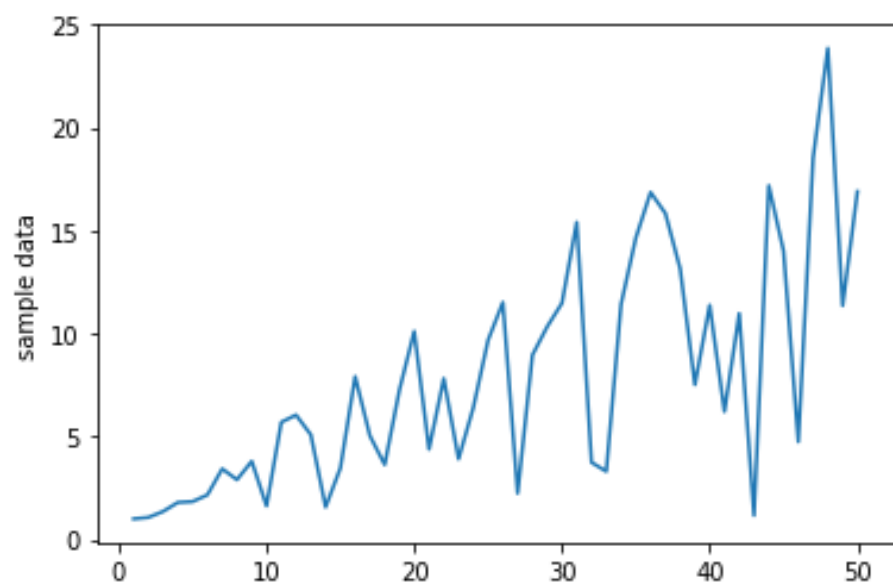


$$\lambda_1 = \lambda_1 - \alpha (d\lambda_1)$$

$$b = b - \alpha (db)$$

Exponentially Moving Average

- ▶ To understand the operation of the majority of algorithms that employ adaptive learning you first need to grasp the idea of exponentially moving average (also called exponentially weighted average).
- ▶ Assume we have a continuous sequence of time series data **Y** containing **n data points**.
- ▶ **EMA is a weighted average of these n data points, where the weighting applied to the data points decreases exponentially over time.**
- ▶ That is EMA ensures that more recent data points obtain a higher weighting.
- ▶ The equation for EMA for the series data Y is as follows:
$$E^t = \begin{cases} Y^1 & \text{when } t = 1 \\ ((1 - \beta) * Y_t) + (\beta * E^{t-1}) & \text{when } t > 1 \end{cases}$$
- ▶ The higher the value of the variable β the slower the old observations are discounted and less emphasis put on the new observation.



Exponentially Moving Average

- ▶ When using the exponentially moving average in ML we don't keep track of each average value of time. Instead we just keep track of one variable the current average. Therefore, when using it in ML it becomes

- ▶
$$E = \begin{cases} Y^1 & \text{when } t = 1 \\ ((1 - \beta) * Y_t) + (\beta * E) & \text{when } t > 1 \end{cases}$$

- ▶ This is one of the benefits of using EMA. It has a low memory footprint (and is computationally inexpensive).
- ▶ One of the benefits of EMA that you see in the previous is that it can smoothen out large oscillations in the value of a variable.

Gradient Descent with Momentum

- ▶ The idea behind gradient descent with momentum is to calculate an exponential moving average of your gradients for each trainable parameter and you use that average to update your weights.
- ▶ So let's look at it for the simple problem where just have two learnable parameters our bias b and a single coefficient λ_1 .

repeat

$$\lambda_1 = \lambda_1 - \alpha (d\lambda_1)$$

$$b = b - \alpha (db)$$

Gradient Descent with Momentum

- ▶ Notice in the code we maintain an EMA for both trainable parameters. It is this value that now becomes core to the update of these parameters.
- ▶ Notice if there is a high level of oscillation for one parameter such as bias then it will smoothen out this value and slow down the rate of update.
- ▶ In turn this can allow us to use a large learning rate and obtain convergence faster.

repeat

$$E_{\lambda_1}^t = \beta E_{\lambda_1}^{t-1} + (1 - \beta) d\lambda_1$$

$$E_b^t = \beta E_b^{t-1} + (1 - \beta) db$$

$$\lambda_1 = \lambda_1 - \alpha (E_{\lambda_1}^t)$$

$$b = b - \alpha (E_b^t)$$

Machine Learning



Machine Learning

Lecture: TensorFlow

Ted Scully

TensorFlow

- ▶ TensorFlow is a powerful open source software library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning
- ▶ TensorFlow was developed by the Google Brain team for internal Google use but it was released to the public in 2015.

Library	Rank	Overall	Github	Stack Overflow	Google Results
tensorflow	1	10.87	4.25	4.37	2.24
keras	2	1.93	0.61	0.83	0.48
caffe	3	1.86	1.00	0.30	0.55
theano	4	0.76	-0.16	0.36	0.55
pytorch	5	0.48	-0.20	-0.30	0.98
sonnet	6	0.43	-0.33	-0.36	1.12
mxnet	7	0.10	0.12	-0.31	0.28
torch	8	0.01	-0.15	-0.01	0.17
cntk	9	-0.02	0.10	-0.28	0.17
dlib	10	-0.60	-0.40	-0.22	0.02
caffe2	11	-0.67	-0.27	-0.36	-0.04
chainer	12	-0.70	-0.40	-0.23	-0.07
paddlepaddle	13	-0.83	-0.27	-0.37	-0.20
deeplearning4j	14	-0.89	-0.06	-0.32	-0.51
lasagne	15	-1.11	-0.38	-0.29	-0.44
bigdl	16	-1.13	-0.46	-0.37	-0.30
dynet	17	-1.25	-0.47	-0.37	-0.42
apache singa	18	-1.34	-0.50	-0.37	-0.47
nvidia digits	19	-1.39	-0.41	-0.35	-0.64
matconvnet	20	-1.41	-0.49	-0.35	-0.58
tflearn	21	-1.45	-0.23	-0.28	-0.94
nervana neon	22	-1.65	-0.39	-0.37	-0.89
opennn	23	-1.97	-0.53	-0.37	-1.07

Some of the companies that use TensorFlow

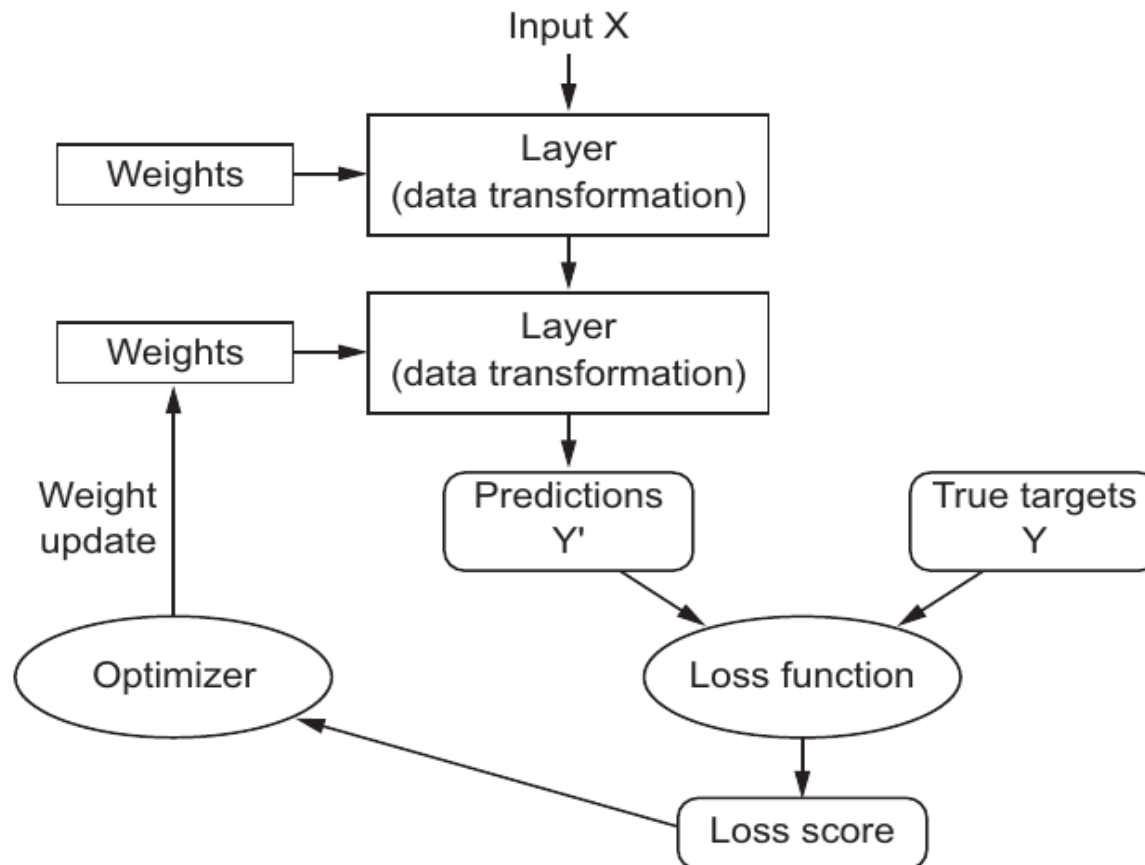
Companies using TensorFlow



TensorFlow

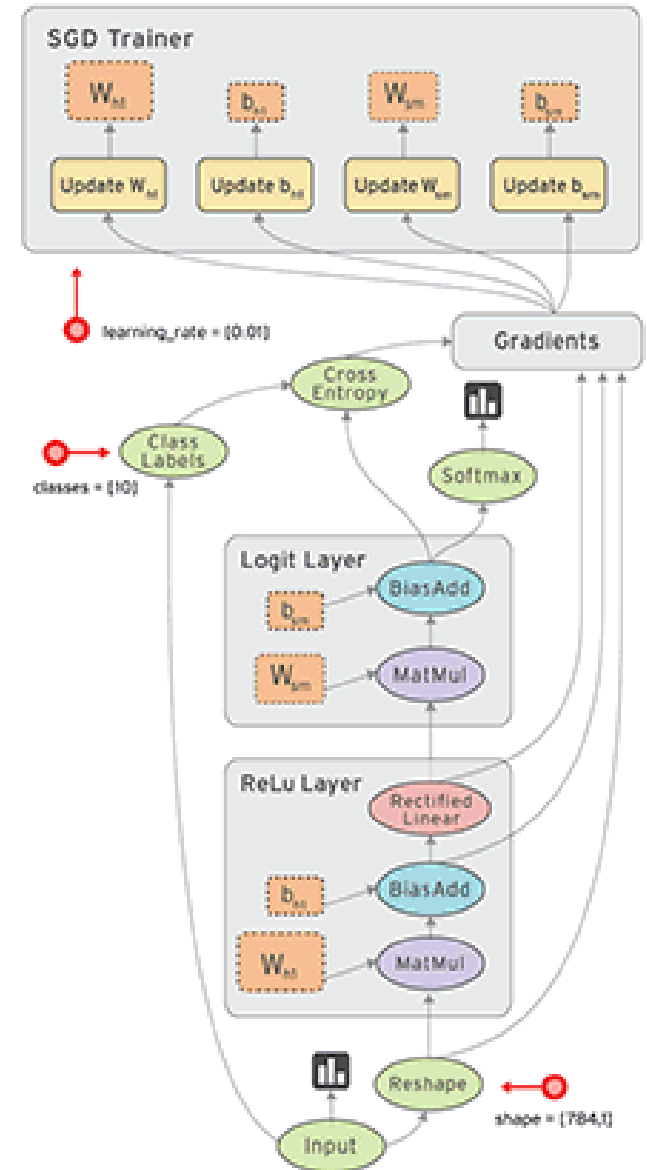
- ▶ It is **open-source project** and has a very active community contributing to improving it. At the moment it is currently one of the most popular open source projects on GitHub.
- ▶ It has both a **high level (Keras and TF Estimator)** and **low level** API, which provides great flexibility (if specific functionality is not available from the high level API then you can move to the low level API).
- ▶ **Keras** has now been adopted as the official high-level API for TensorFlow.
- ▶ It also comes with a strong visualization tool called **TensorBoard** that allows you to browse through the computation graph, view learning curves, and more.
- ▶ As with all deep learning frameworks it provides **automatic differentiation**, which automatically take care of calculating the gradients for you.

How Neural Networks Work



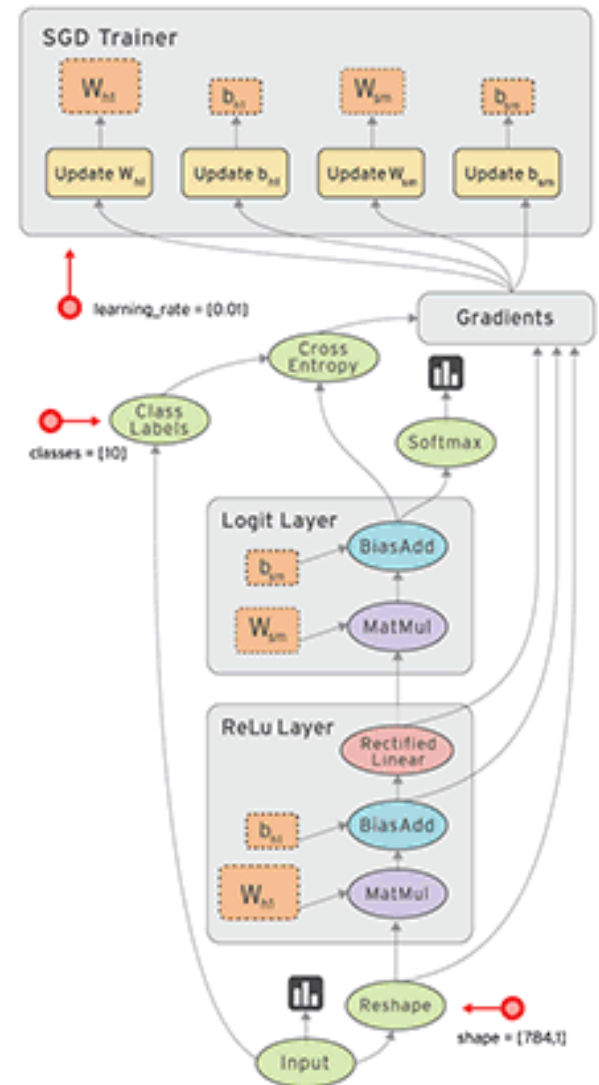
TensorFlow

- ▶ TensorFlow is graph based, that is you define your operations as a computation graph.
- ▶ One of the basic advantages is that graph easily facilitates parallelism. The edges of the graph represent dependencies between operations. It is easy for the system to identify operations that can execute in parallel.
- ▶ The advantages of using a graph based approach is that it can significantly boost performance (TF also has an excellent package called XLA that can optimize computation on a graph).



TensorFlow

- ▶ The graph is a language-independent representation of your model. You can build a graph in Python, store it, and restore it in another language.
- ▶ As such graphs make it easy to deploy your models to any device. You aren't tied to a specific language. It runs on Windows, Linux, and MacOS, and also on **mobile devices**, including both iOS and Android.



TensorFlow

- ▶ The graph based approach also makes it possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines
- ▶ In other words TF allows you to train very large neural networks on massive training sets in a reasonable amount of time by splitting the computations across many different machines/GPUs/TPUs..

