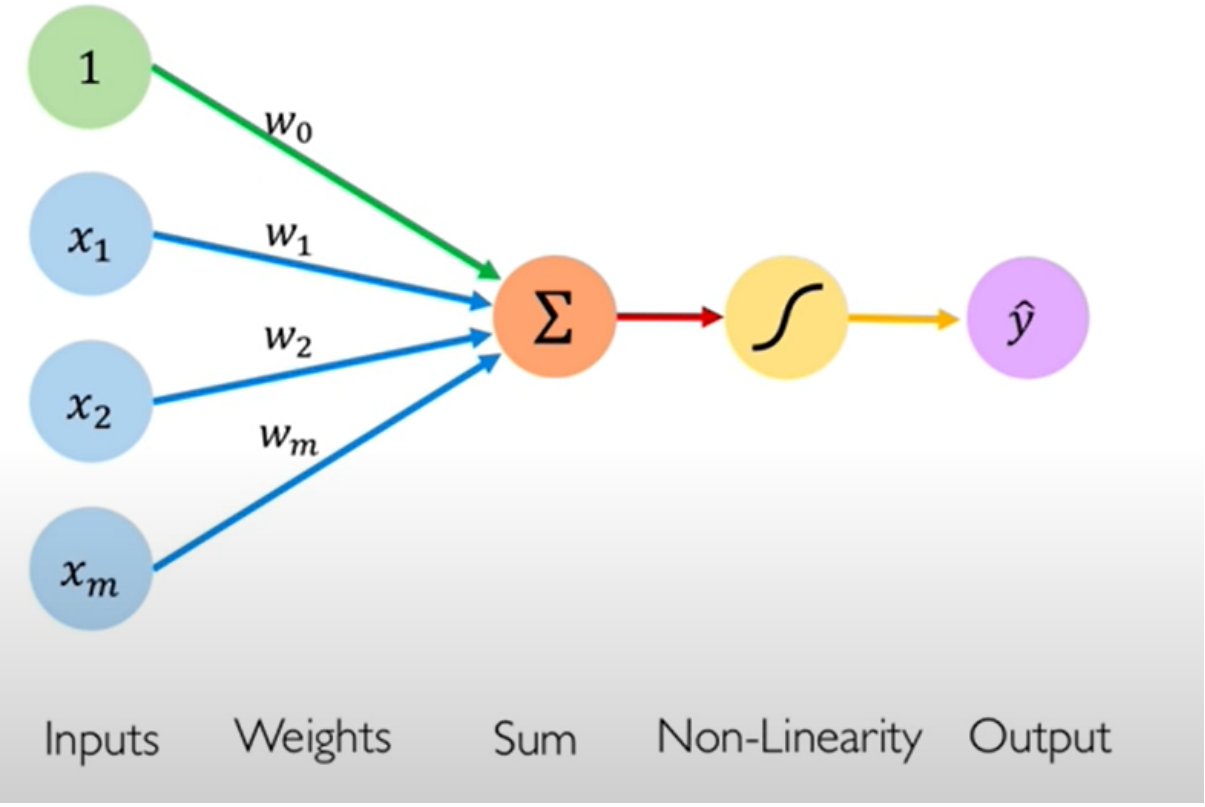
[1] Introduction to Deep Learning

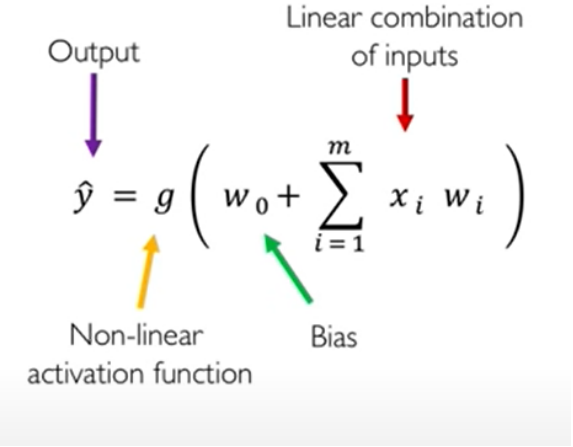
What is Deep Learning

- Artificial intelligence - Any technique that enables computers to mimic human behavior
    - Machine Learning - Ability to learn without explicitly programmed
        - Deep Learning - Extract from data using neural networks

The Perceptron: Forward Propagation



- multiply the input with the weight and bias and take the sum of all of them
- Take the single number and pass it through a non-linear activation function and you get the output



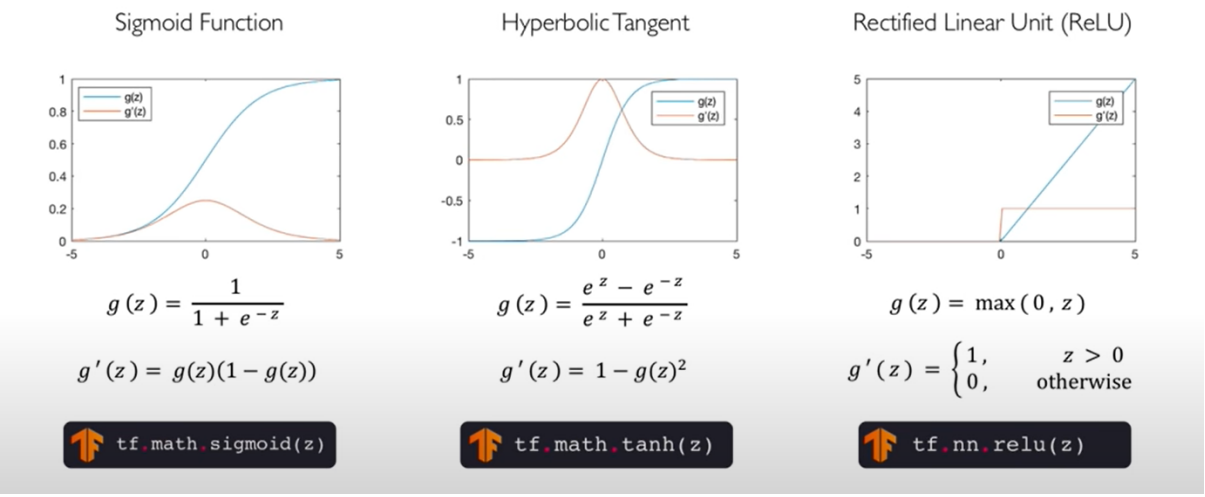
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\ w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

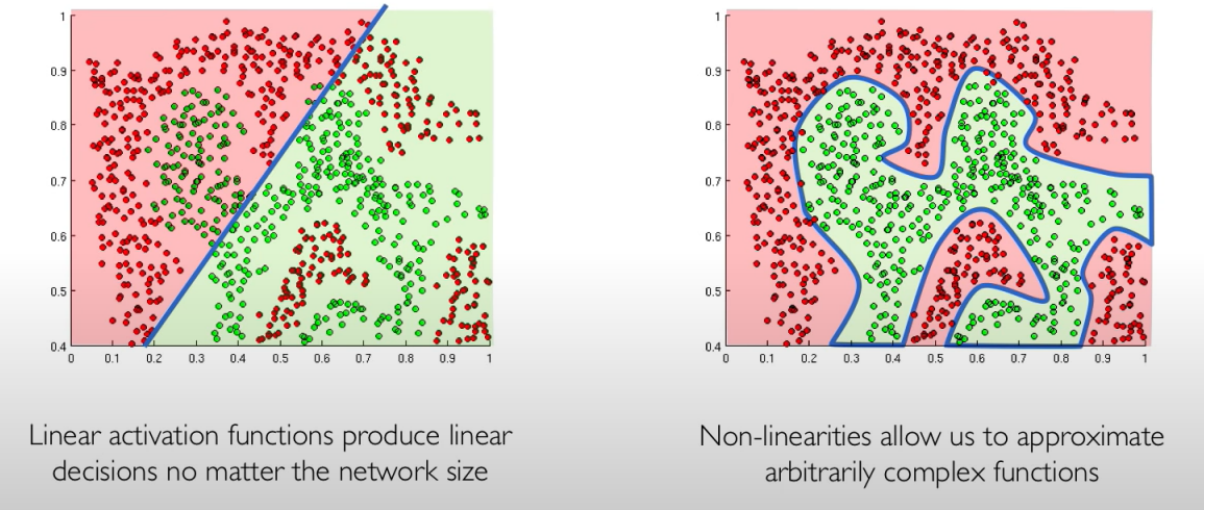
Activation Functions

- Sigmoid
  - It takes transfers the output into a scalable output between 0 and 1
- tanh
  - number between -1 to 1
- ReLU

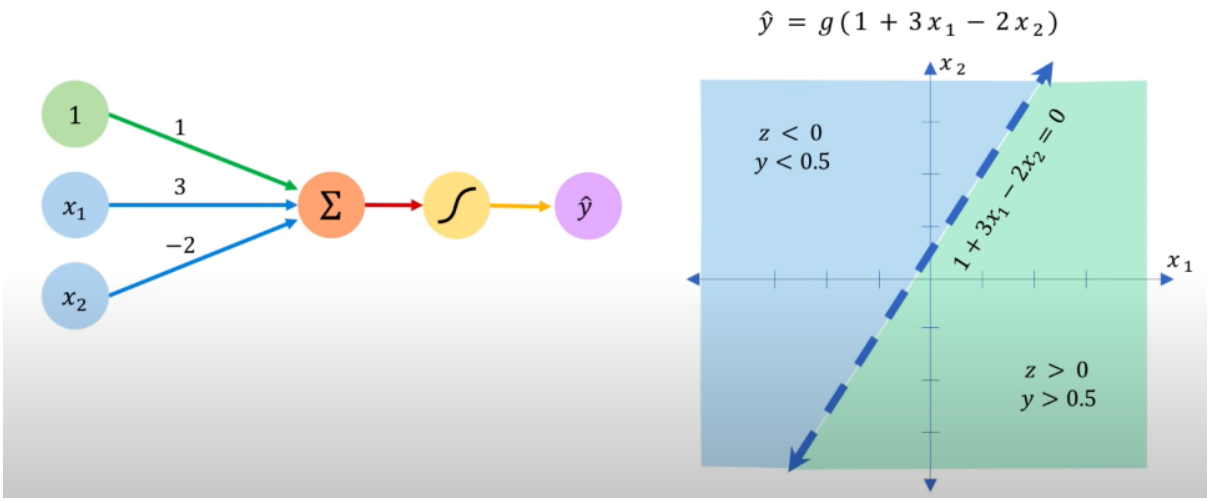


- Why activation functions?
  - Activation function introduces a complexity to your neural network

Example (separate red and green dots):



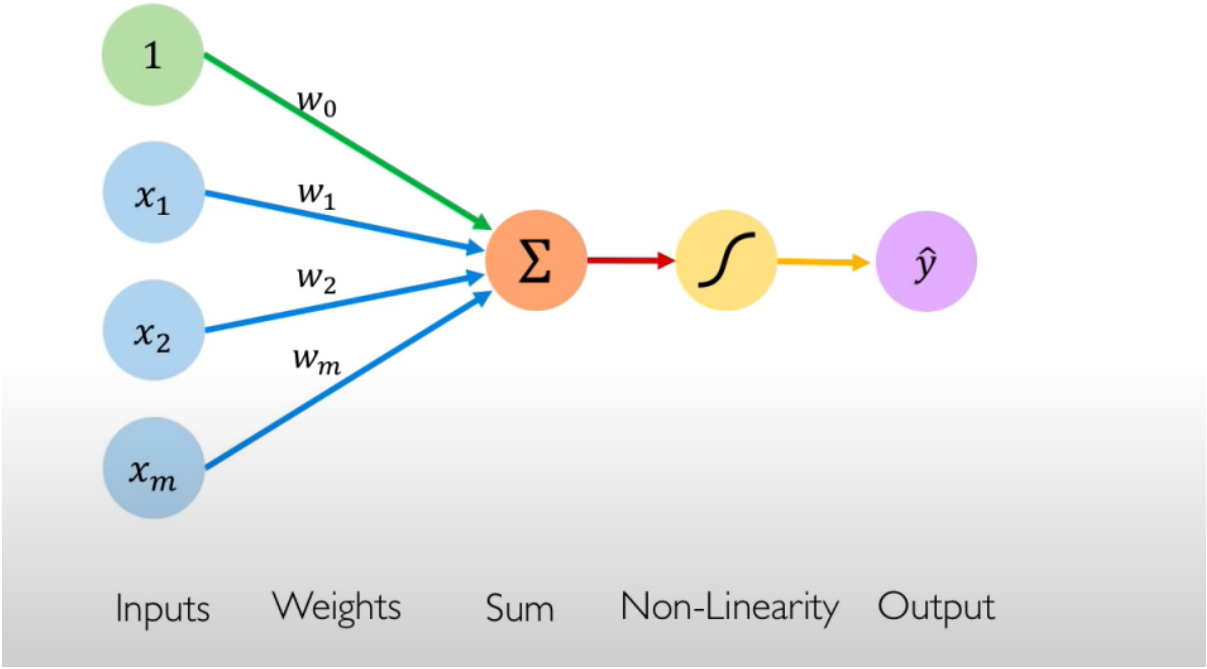
Example:



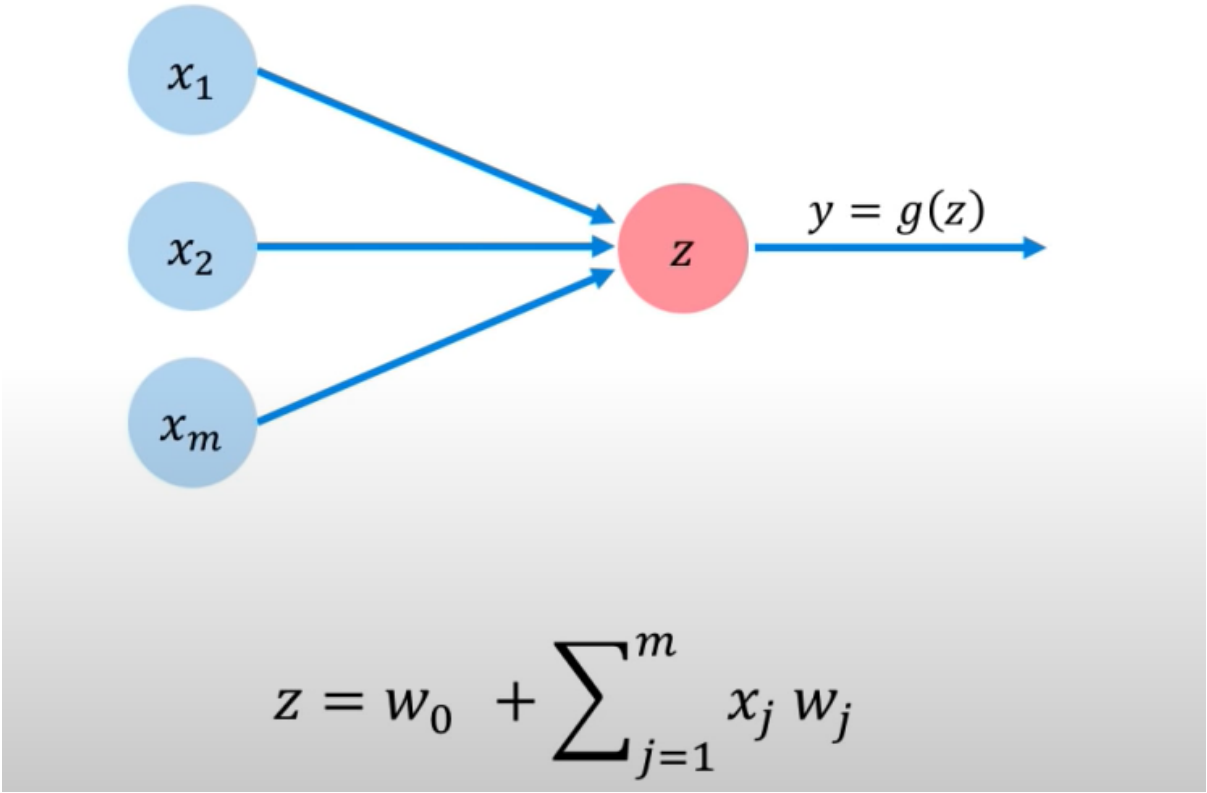
Building Neural Networks with Perceptions

The Perceptron: Simplified

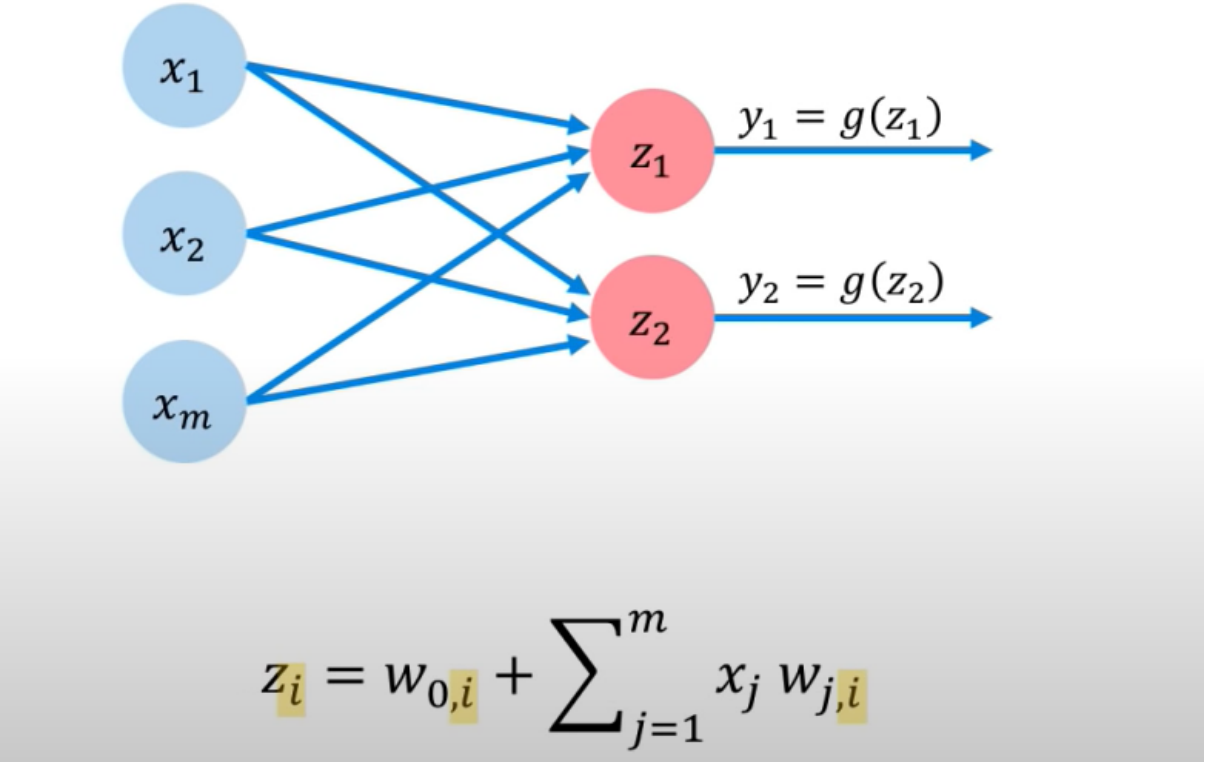
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$



- How it works
- take inputs and apply dot product to the weight, add the bias and apply a non-linearity



- The diagram above has no bias and no weight label

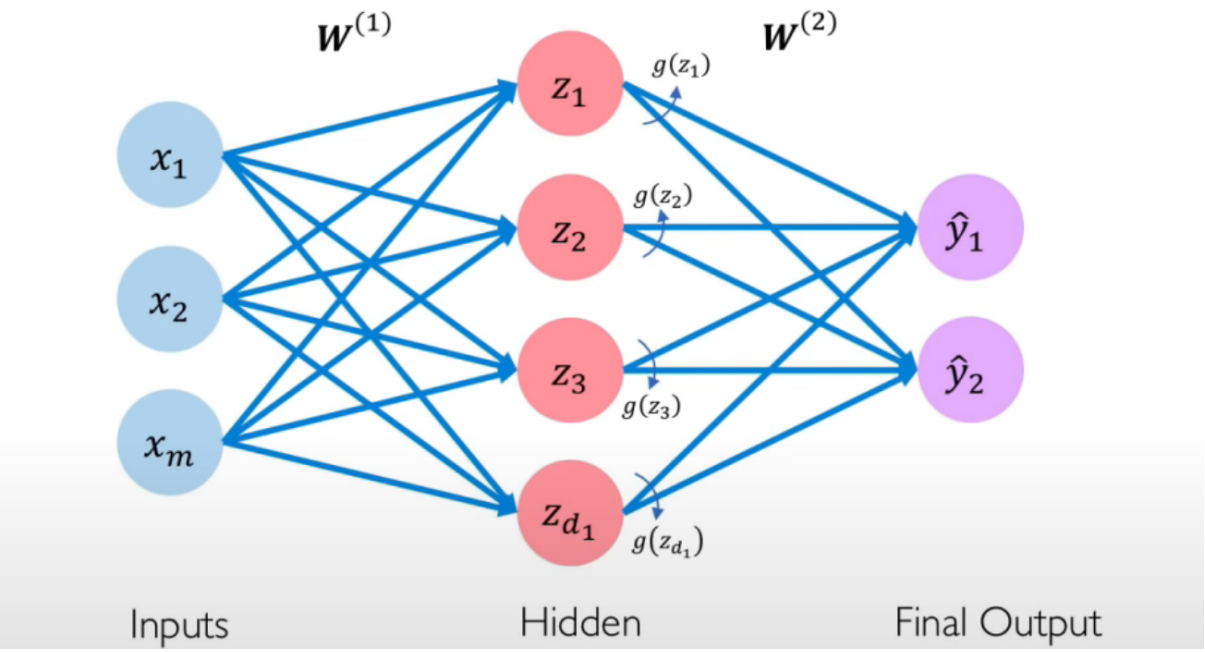


- Diagram above is a Multi-Level neural network
- The only difference between this and single level is an extra perceptron in the network
  - y1 is output one
  - y2 is output two
- Because all outputs are connected to all inputs so those layers are called **dense** layers
- How to build the a Dense Layer
  - TensorFlow have implemented a dense layer for you so you don't have to do it manually

```
import tensorflow as tf

layer = tf.keras.layers.Dense(
    units=2)
```

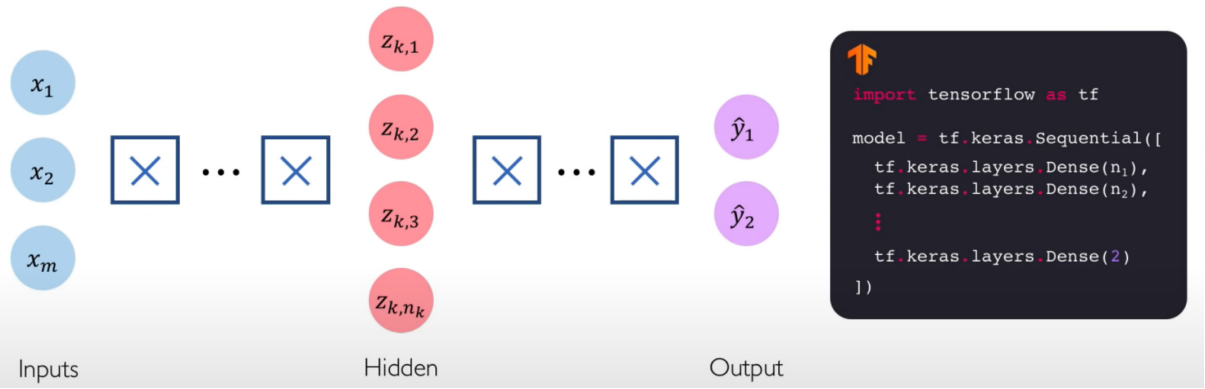
- The Diagram below is a single layer neural network



- The code below Implements a single neural network in TensorFlow

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

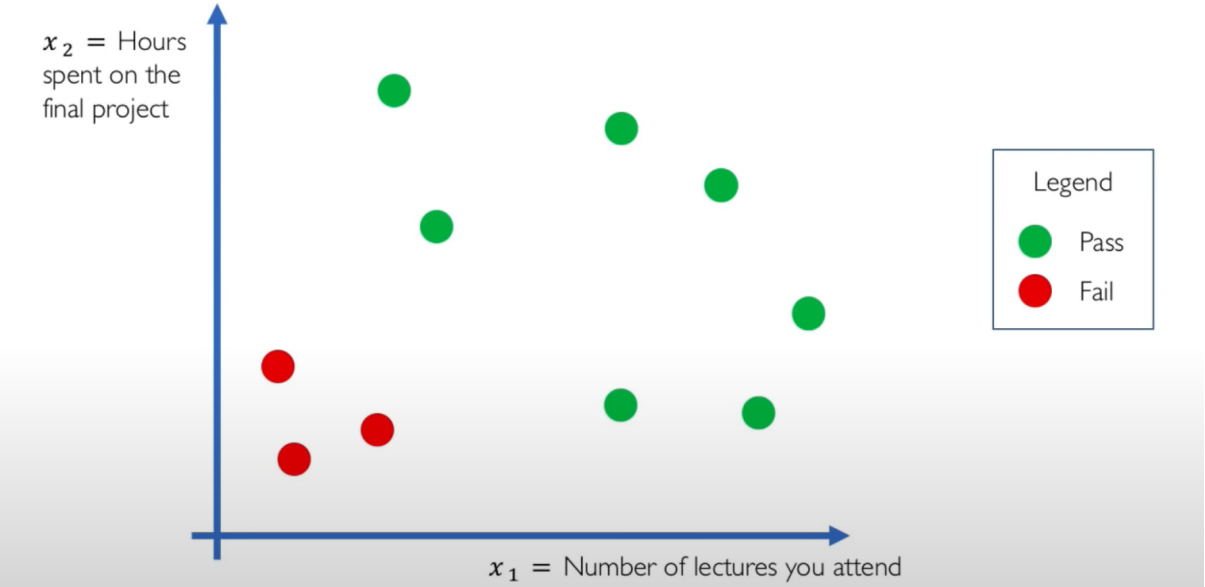
- The code below implements a deep neural network in TensorFlow



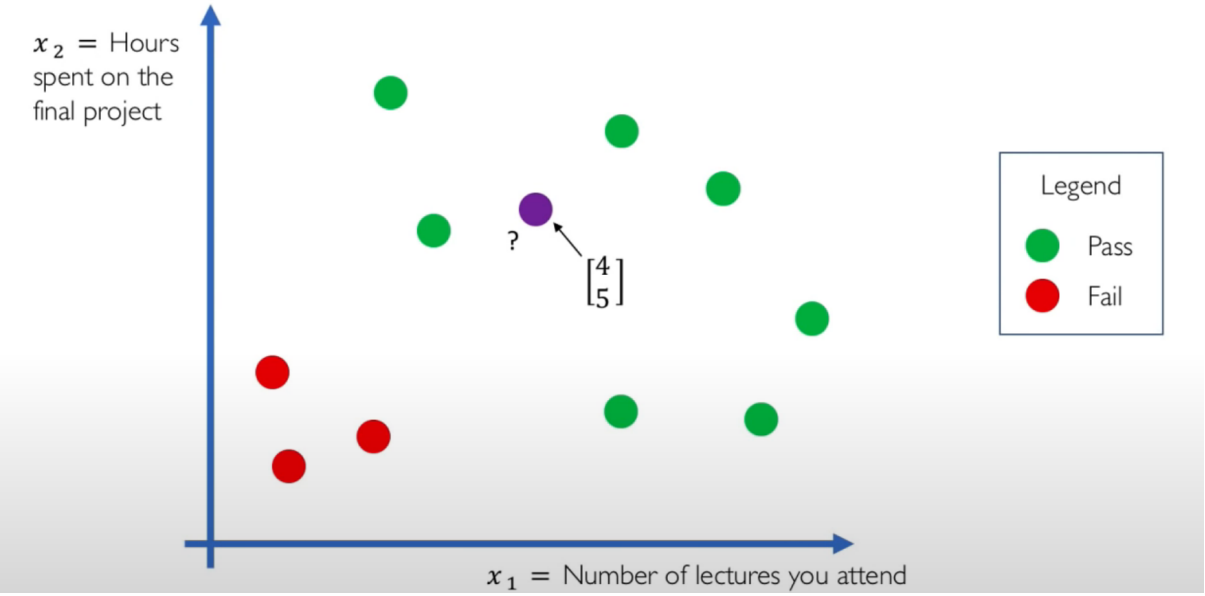
Applying Neural Network

Example Problem

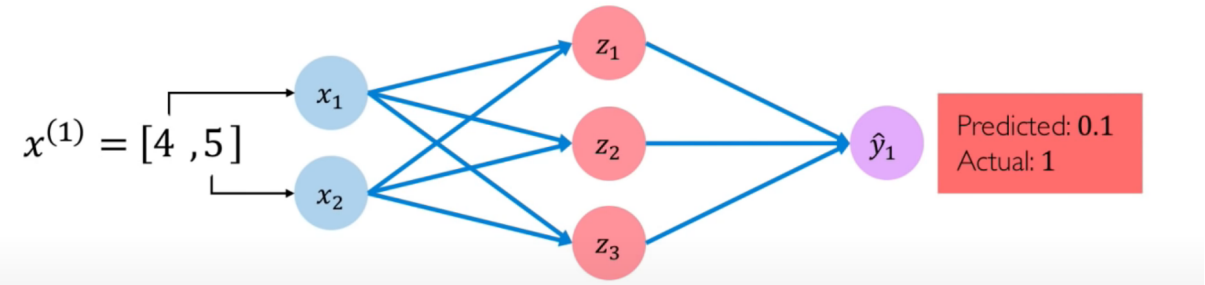
- Will I pass this class
  - x_1 = Number of lecture attened
  - x_2 = Number of hours spent on final project



- The purple dot is you, will you pass the class?



- If we just pass in your results into the network, the result you get is very wrong, you are expecting 1 but you got 0.1 which is 10 percent change for you to pass



- The reason is because the network was not trained therefore it does not know what is right and what is wrong it is just guessing the answer with a really low accuracy
- To train the network we use the loss function to determine how wrong the network is
- The loss function we will be using is softmax
- softmax compares how different the distribution are

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\left(\underbrace{f\left(x^{(i)}; W\right)}_{\text{Predicted}}\right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\left(1 - \underbrace{f\left(x^{(i)}; W\right)}_{\text{Predicted}}\right)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

- Now, let's say we want to predict your grade instead of weather you are going to pass[
- The loss function we will be using is called mean square error
- this could be used for regression models that output continues real numbers

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f\left(x^{(i)}; W\right)}_{\text{Predicted}}\right)^2$$

- Take the prediction of the network and subtract it from the actual number and square it, which would be the loss and which the model would try and change the weight to minimize the loss

Training The Neural Network

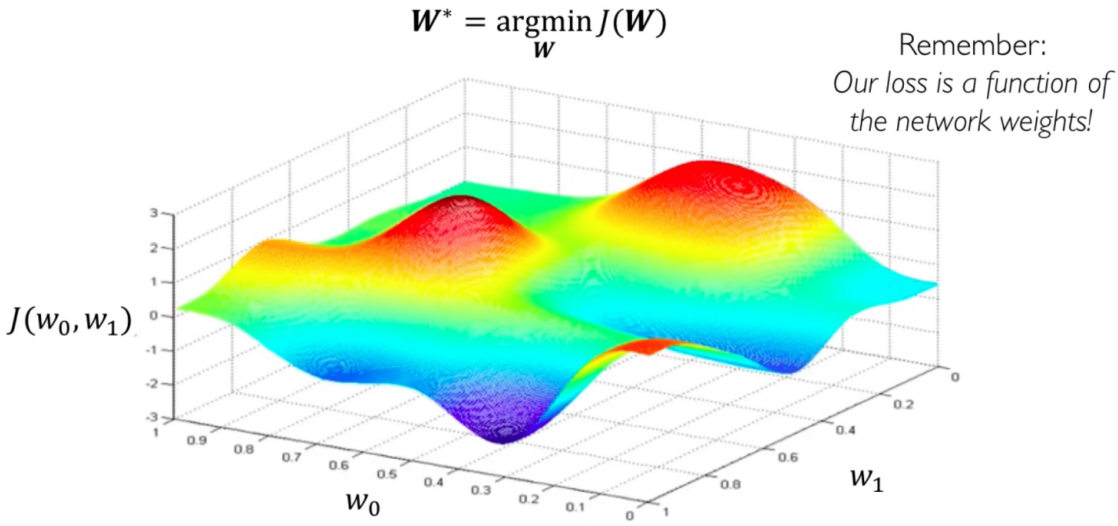
- Training the network basically tries to find the weight that achieves the lowest loss

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}}\, J(\boldsymbol{W})$$

Remember:
$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\}$

- This graph plots the weights and the loss



- This is algorithm gradient descend\

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

- Learning rate is how strong you want each gradient to be (ADA)
- Sudo code in actual code that is hard coded

```python
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

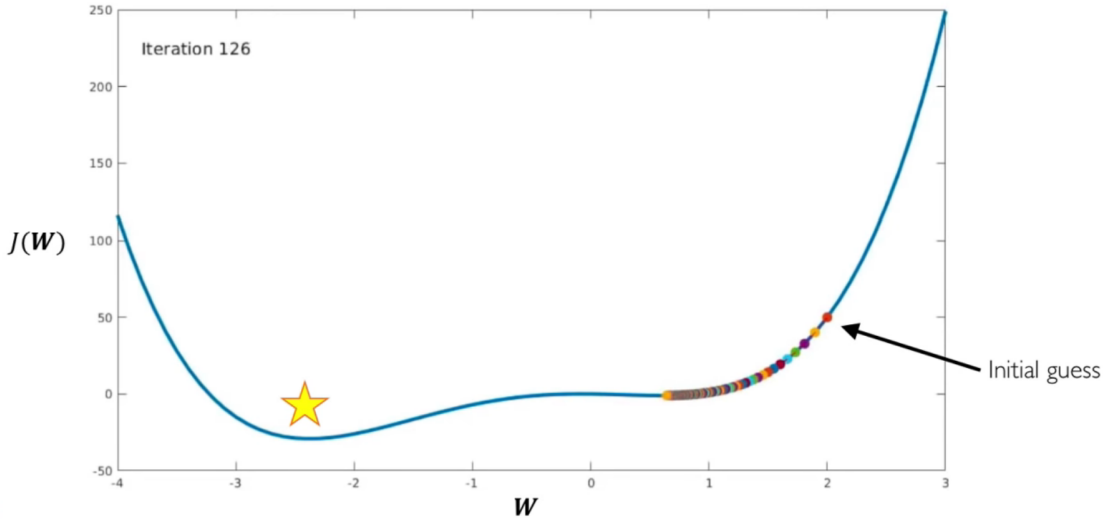
Optimization

- function for optimization

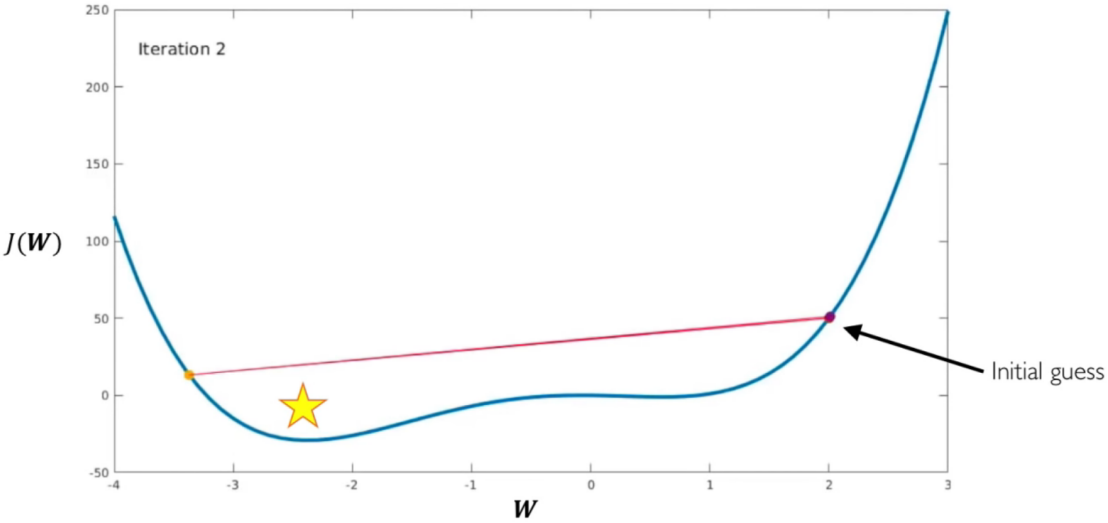
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

- Setting learning rate is important
  - If the learning rate is too small then the model could possibly be stuck at local minima
  - If the learning rate is too high then the model could overshoot

Small learning rate *converges slowly and gets stuck in false local minima*



Large learning rates *overshoot, become unstable and diverge*



- How to set a learning rate?
  - Try different ones and see which one is "just right"
  - Adaptive learning rate that changes with different landscapes
- Adaptive learning rate
  - Learning rate is no longer fixed
  - Can be larger or smaller depending on:
    - how large the gradient is
    - how fast learning is happening

    - size of particular weight
    - ETC....
- The image below shows different types of learning rates

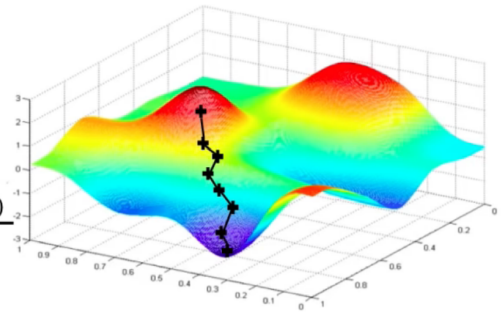
Algorithm	TF Implementation	Reference
• SGD	tf.keras.optimizers.SGD	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	tf.keras.optimizers.Adam	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelta	tf.keras.optimizers.Adadelta	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	tf.keras.optimizers.Adagrad	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	tf.keras.optimizers.RMSProp	

Mini Batches

- It is basically a approximation of the gradient

Algorithm

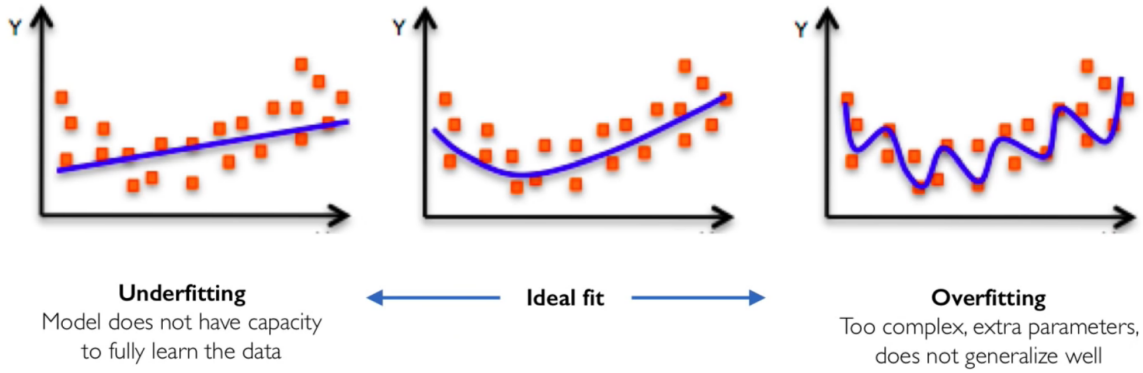
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



- Mini batches makes estimating to gradient much quicker since it takes less to compute and have multiple points
- Could increase learning rate
- Faster training because we could split it up to different GPU or computers

Overfitting

- We would need to test the model with unseen test data so the model don't overfit



Regularization

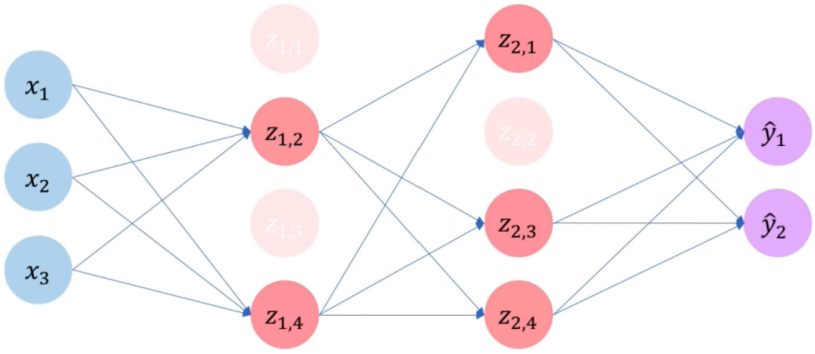
- Technique to constrain our optimization problem to discourage complex models
- Improve accuracy for unseen data

Dropouts

- Drops out a percentage of data on during each training

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.keras.layers.Dropout(p=0.5)
```



- Forced to generalize better and build a better model and prevent overfitting

Early stopping

- Stop the training before it has the chance to overfit

Summary:

The Perceptron	Neural Networks	Training in Practice
• Structural building blocks • Nonlinear activation functions	• Stacking Perceptrons to form neural networks • Optimization through backpropagation	• Adaptive learning • Batching • Regularization