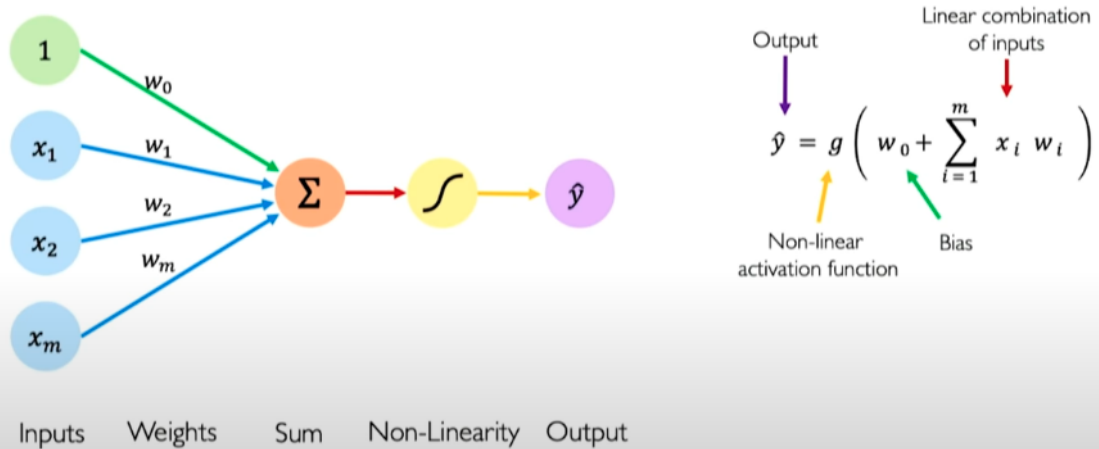


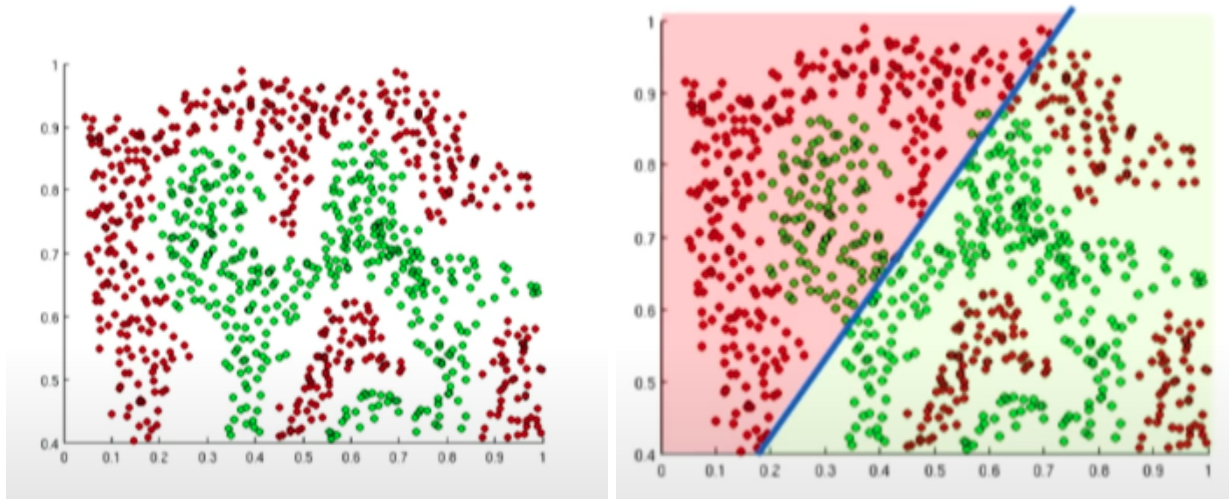
## Deep Learning

### The Perceptron: Forward Propagation



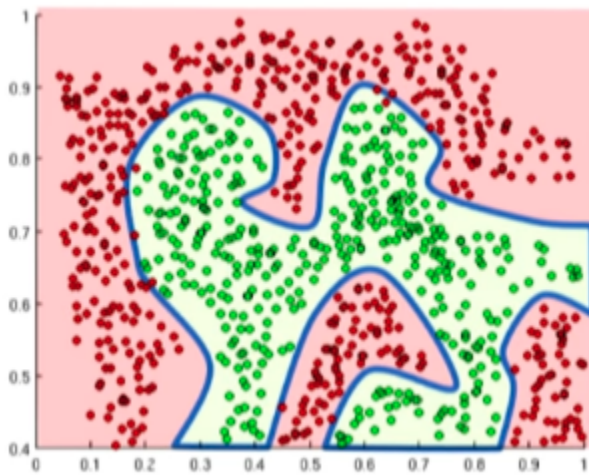
#### Activation Function:

The purpose of the activation function in a neural network is to introduce **NON - LINEARITIES** into the network.



Now, imagine, if we want to distinguish the green and red points. If we have only one line to specify the division, WE Can not. So, this is why we need to introduce non-linearity in our function.

Non-linearities allow us to approximate arbitrarily complex functions and that's what makes NN extremely powerful.

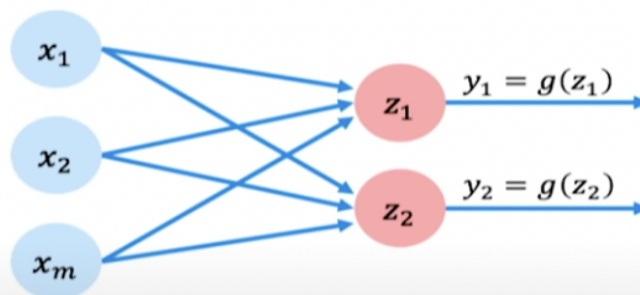


How a perceptron works:

- The dot product of inputs and weights.
- Add a bias
- Apply Non-linearity

$$z = w_0 + \sum_{j=1}^m x_j w_j$$

# Multi Output Perceptron



$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

Two different neurons would apply their own bias and weights on the inputs. In the above figure, we will get two outputs. Also, all inputs are densely connected to the neurons that's why it is called the Dense layers.

## Dense layer from scratch

```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

    return output
```

Import tensorflow as tf

Layer = tf.keras.layers.Dense(units = 2)

## Optimization

We need to find the weights that achieve the lowest loss.


For the optimization, we need to find the  $w$ 's that minimize the  $J(w)$ , our empirical loss or average loss. A loss function is just a function that takes the inputs with the set of the weights and gives out a single value which can be referred to as the error.

We compute the derivative or the gradient of the loss function to tell us the direction we need to go in order to minimize our loss and following that, we take a small step in the opposite direction of that gradient to find the lowest loss.

### Gradient Descent

#### Algorithm

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




```
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
```

### Putting it all together



```
import tensorflow as tf
model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizers.SGD()

while True:  # loop forever
    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```



Can replace with any TensorFlow optimizer!

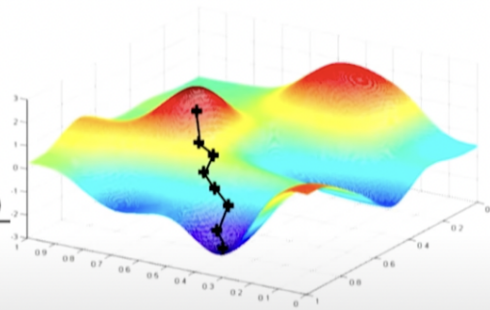
The gradient can be very heavy computationally intensive to compute. Here comes the concept SGD.

**SGD:** instead of computing the gradient on the entire dataset. We compute it only on a single data point of our whole dataset. It can be sometimes noisy. Obviously, a single datapoint won't be able to showcase the gradient of the whole, but we would be able to estimate the gradient from one.

## Stochastic Gradient Descent

### 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,  $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{w})}{\partial \mathbf{w}}$
5.     Update weights,  $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$
6. Return weights



It's not going to be the true gradient but an estimate. The major benefit of it is it is computationally less intensive. But we can refer it to be very noisy and stochastic.

So, what's the middle ground. Instead of computing the whole dataset or just a single data point. We can compute a subset of the dataset or we can call it a batch.

And, it gives us a much better estimate. Yes, it is still an estimate but it is not as noisy as when we will be using a single data point to compute the gradient.

So, it is much faster to compute and provides a better estimate.

### Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

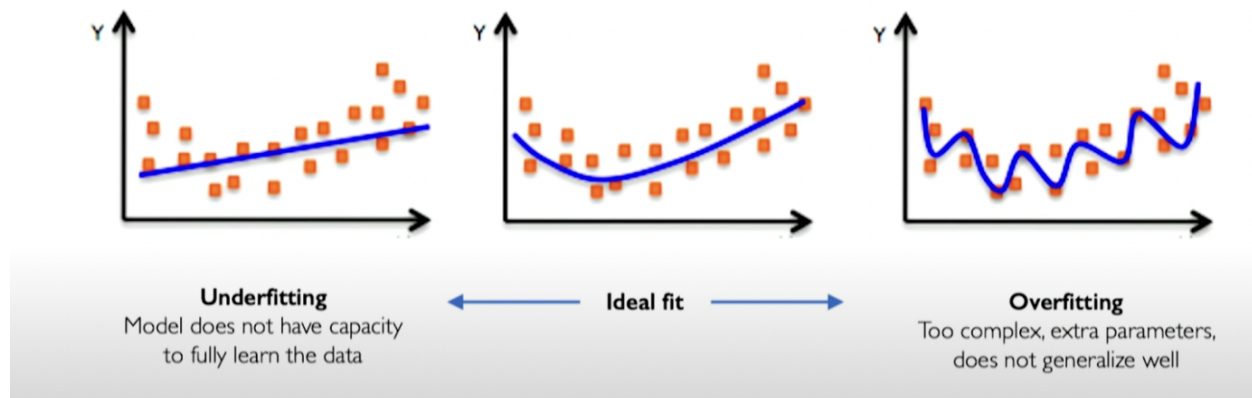
### More accurate estimation of gradient

Smoother convergence  
Allows for larger learning rates



## Overfitting in Neural Networks

### The Problem of Overfitting



To encourage a more simple model, we use regularization to discourage complex models and improve more generalization on new unseen data.

### Techniques:

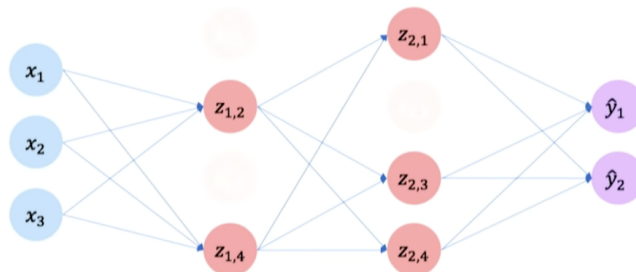
**1). Dropout:** In this technique, we randomly drop and set some activation functions to zero in our neural network. Let's say we have dropped 50% of the neurons in the network, which means we have set 50% activation functions to zero.

In the first iteration, we encourage different 50% neurons on our network, in the next iteration we choose other random 50% neurons. This makes our network choose different pathways every time and encourage our network to encourage different forms of processing the information to complete its decision-making capabilities

### Regularization I: Dropout

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

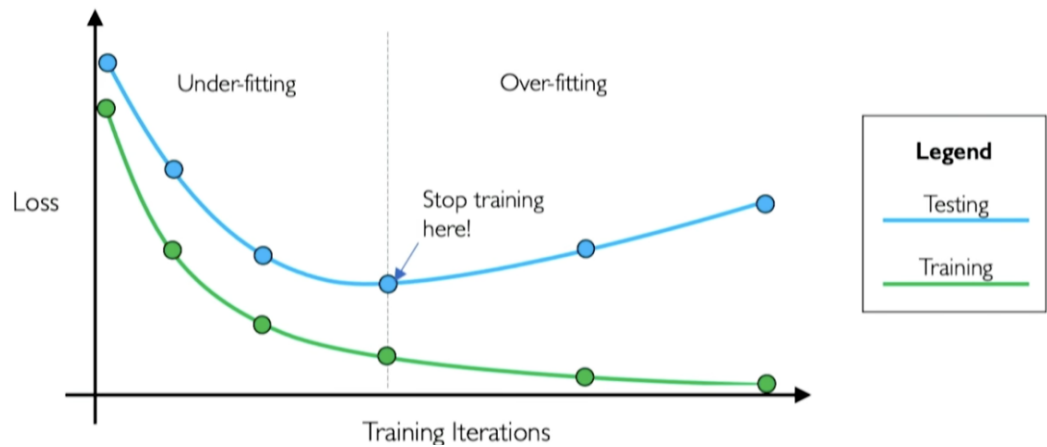
 `tf.keras.layers.Dropout(p=0.5)`



## 2). Early Stopping:

# Regularization 2: Early Stopping

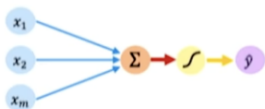
- Stop training before we have a chance to overfit



## Core Foundation Review

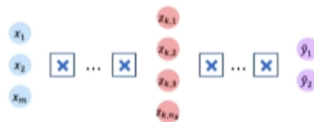
### The Perceptron

- Structural building blocks
- Nonlinear activation functions



### Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



### Training in Practice

- Adaptive learning
- Batching
- Regularization

