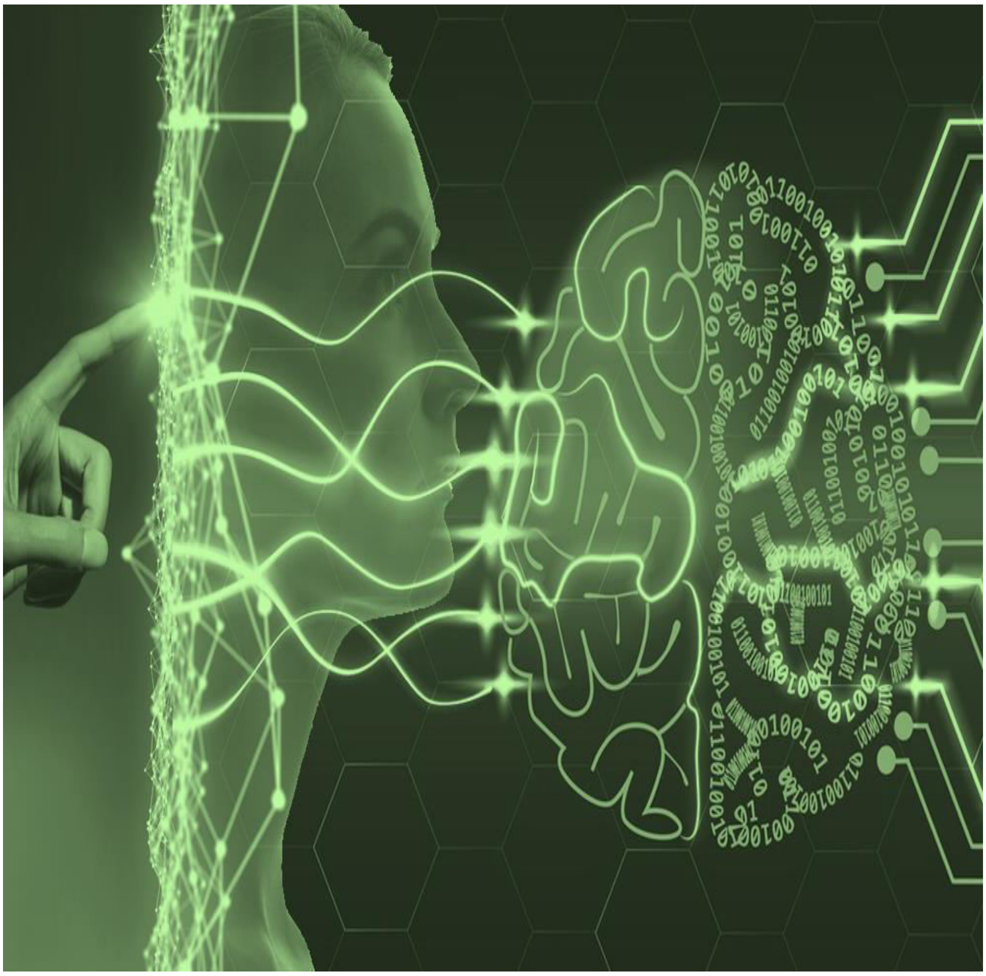
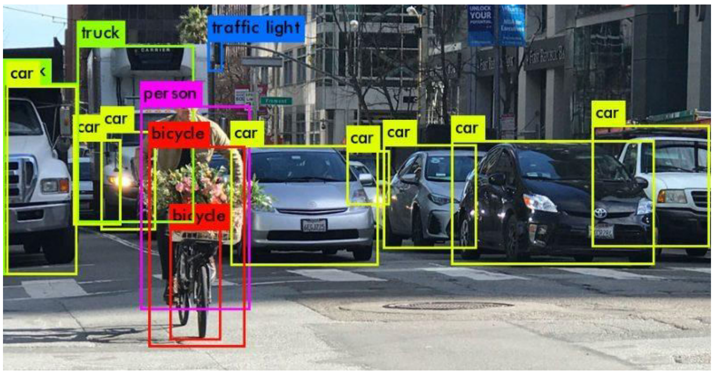
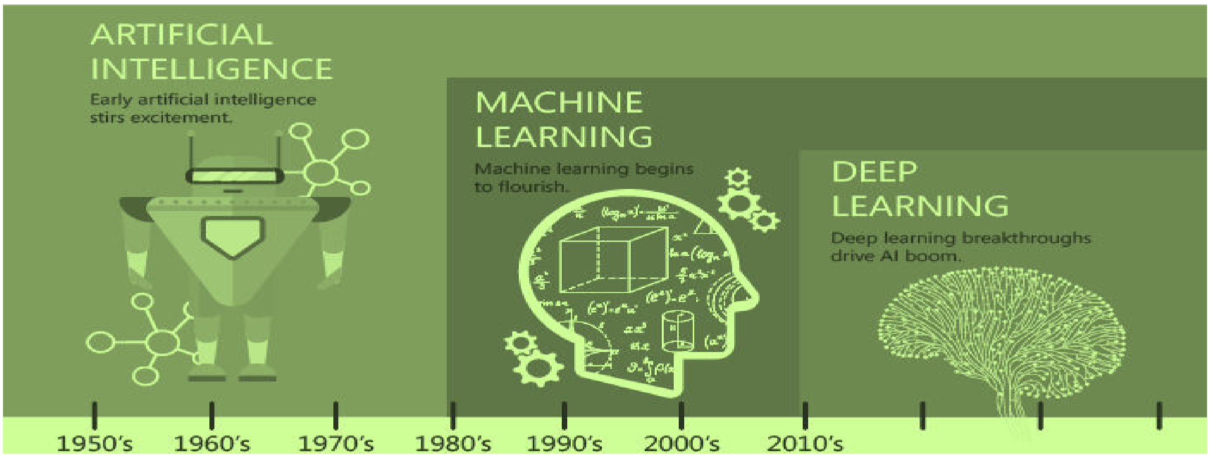


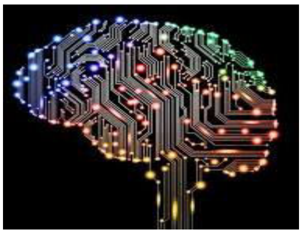
INTRODUCTION TO DEEP LEARNING

Application of deep learning

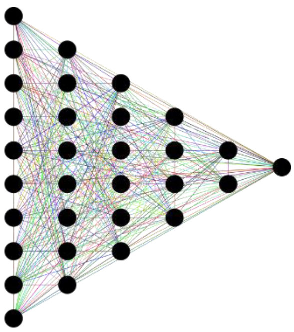


What is Deep Learning?

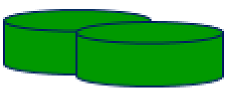




- Deep learning is a type of machine learning that mimics the neuron of the neural networks present in the human brain.

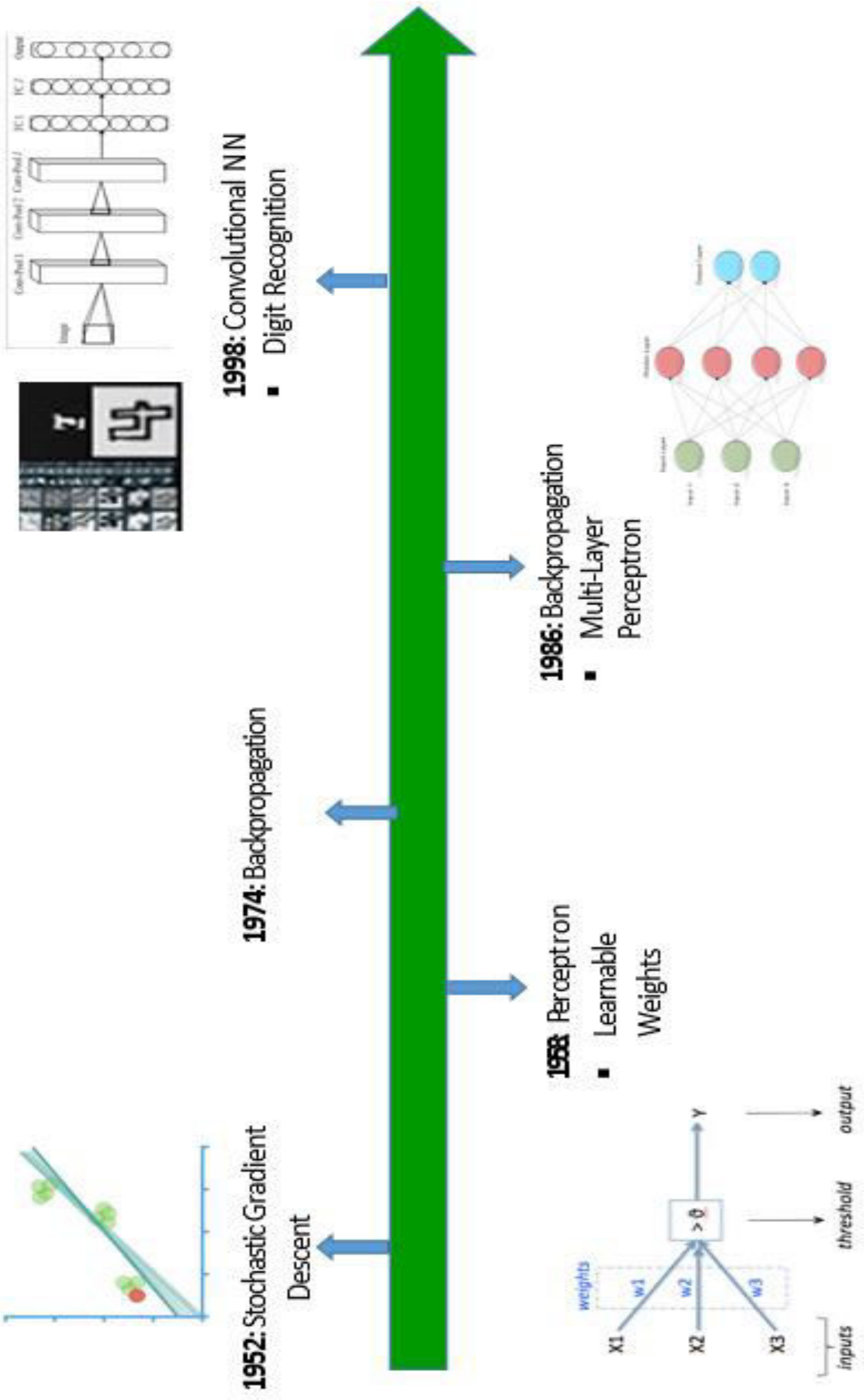


- Deep learning algorithms learn progressively about the input data as it goes through each neural network layer.



- If the system provided with tons of information, it begins to understand it and respond in useful ways.

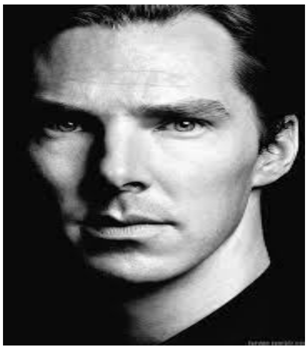
A long Time Ago…



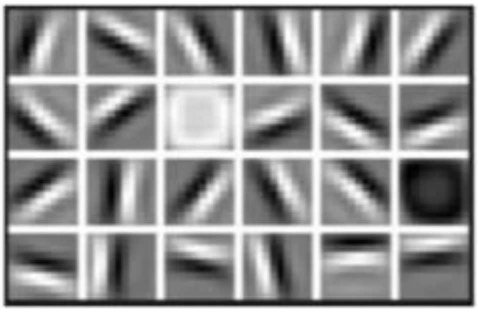
Why using Deep Learning Now?

- Neural Network date back, so why resurgence?

1. Big Data
    a. Larger Datasets
    b. Easier Collection and storage
2. Hardware
    a. Graphics Processing Units (GPUs)
    b. Massively Parallelizable
3. Software
    a. Improved Techniques
    b. New Models
    c. Toolboxes



Low Level Features

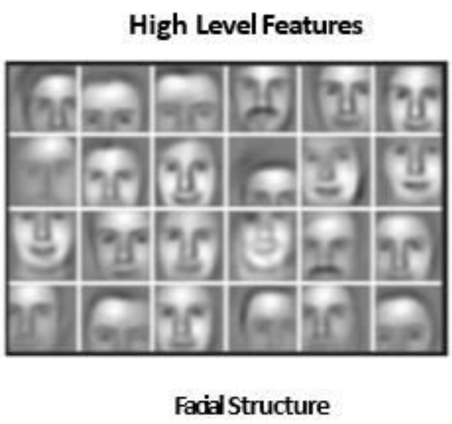


Lines & Edges

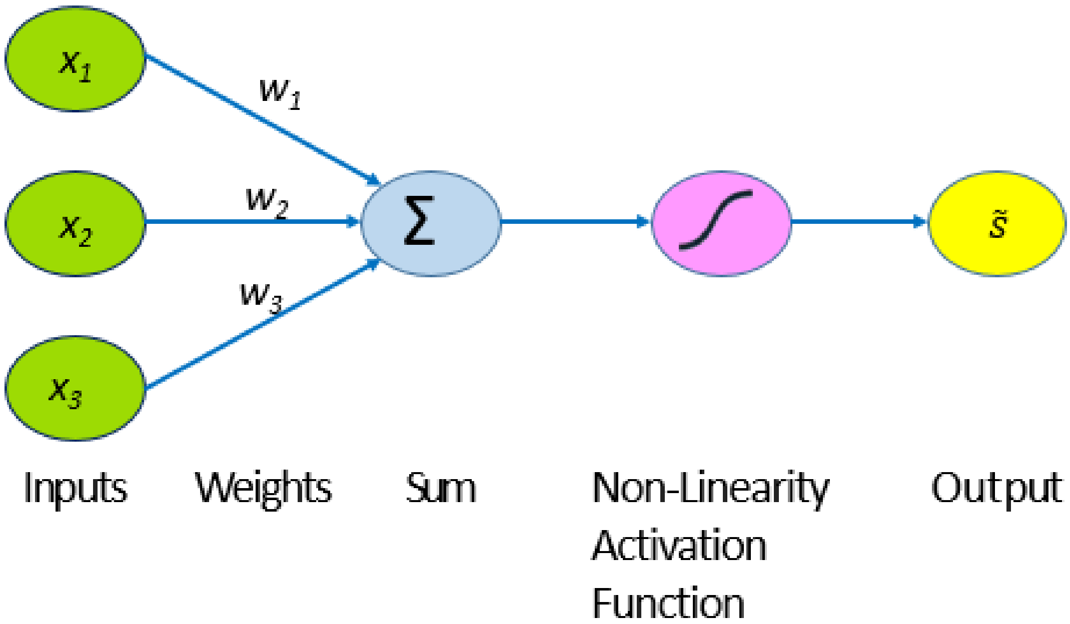
Why Deep Learning?

Mid Level Features

Eyes& Nose & Ears



The Perceptron: Basic neural network building block

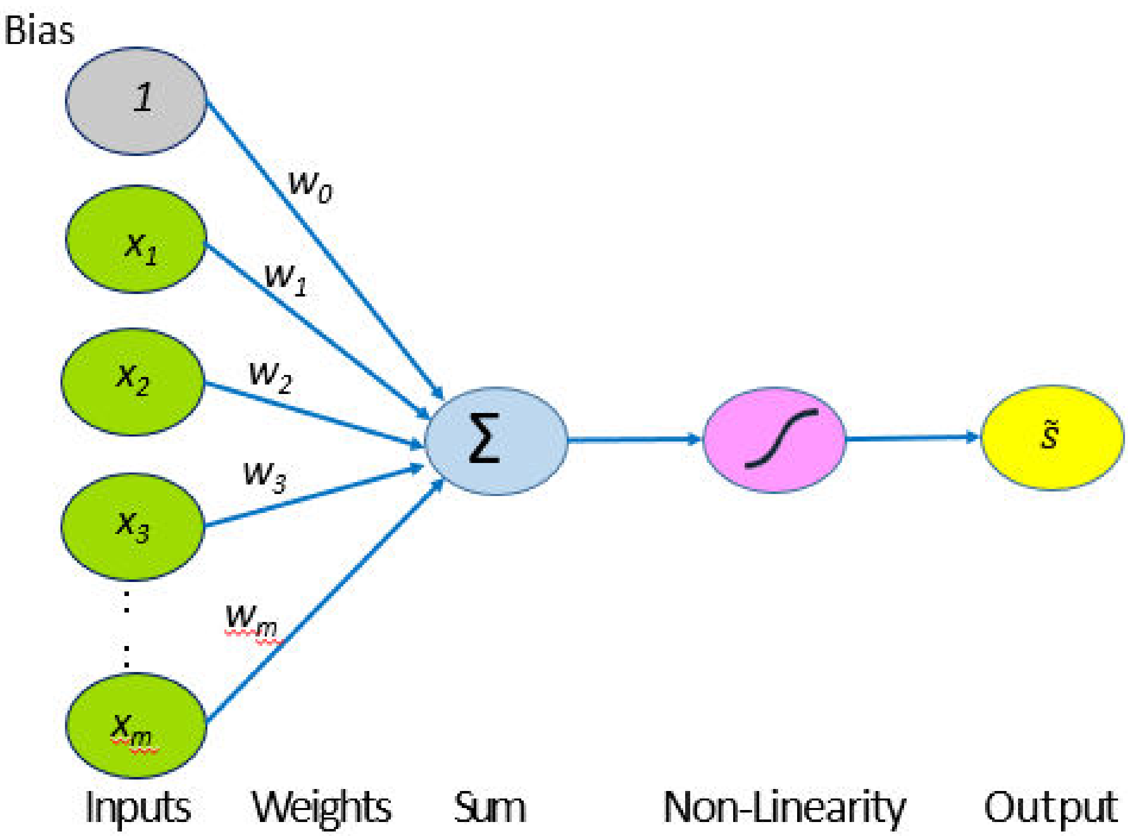


Non-linear activation function

Linear combination of inputs

$$\tilde{s} = F\left(\sum_{i=1}^{m} x_i w_i\right)$$

The Perceptron: Forward Propagation



Mathematical Function

Output

Linear combination of inputs

$$\tilde{S} = F(w_0 + \sum_{i=1}^{m} x_i w_i)$$

Non-linear activation function

Bias

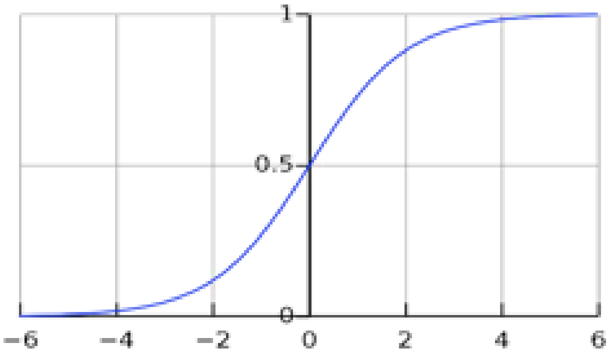
$$\tilde{s} = F(w_0 + \sum_{i=1}^{m} x_i w_i)$$

$$\tilde{s} = F(w_0 + X^T W)$$

where: $X = \begin{bmatrix} X_1 \\ \vdots \\ X_m \end{bmatrix}$ and W $= \begin{bmatrix} W_1 \\ \vdots \\ W_m \end{bmatrix}$

Activation Functions

$$\tilde{y} = FF(\ w_o + X^T W\ )$$

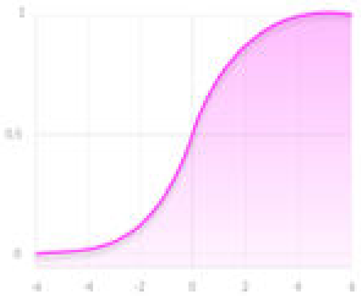


Sigmoid function

$$F(z) = \sigma\ (z) = \frac{1}{1+e^{-z}}$$

Some Non-Linear Activation Functions

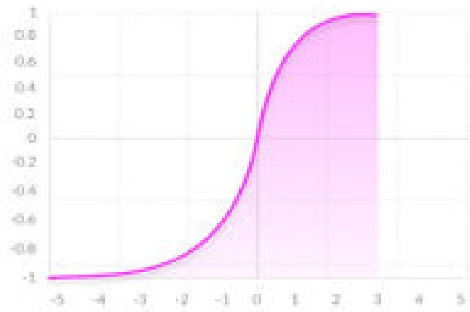
Sigmoid Function



$$F(z) = \frac{1}{1 + e^{-z}}$$

tf.keras.activations.sigmoid(x)

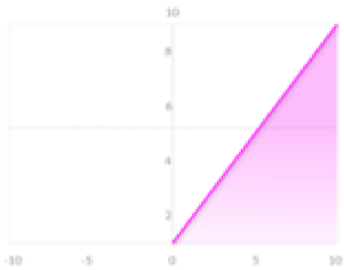
Tanh/ Hyperbolic Tangent



$$F(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

tf.keras.activations.sigmoid(x)

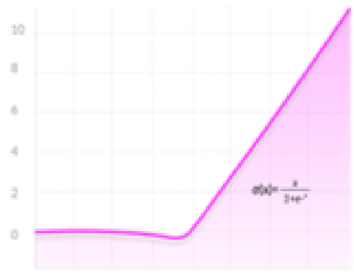
Rectified Linear Unit (ReLU)



$$F(z) = max\ (\ 0,\ z)$$

tf.keras.activations.relu(x)

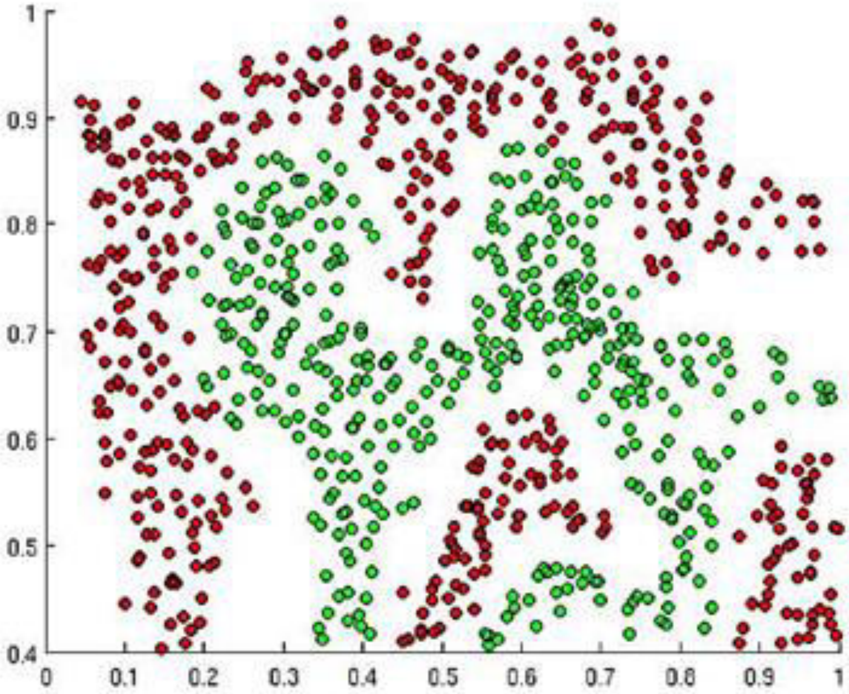
Swish



$$F(x) = x \cdot \frac{1}{1 + e^{-z}}$$

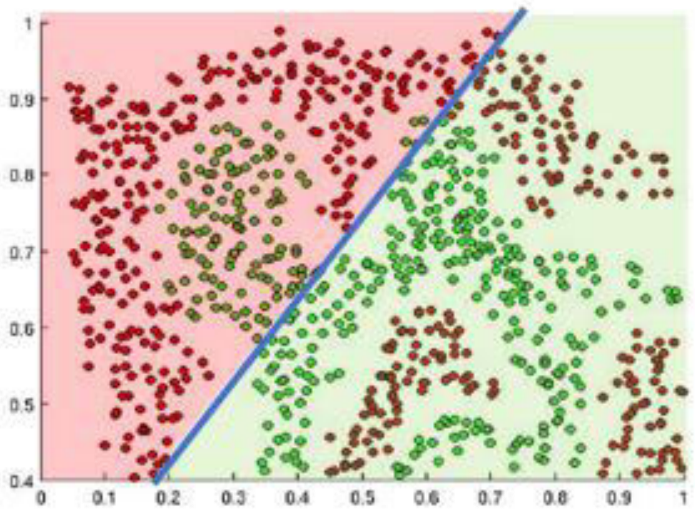
tf.keras.activations.awish(x)

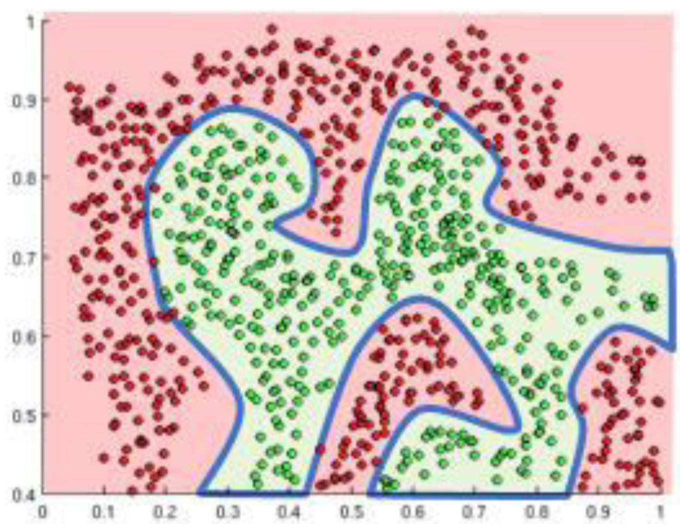
Why Activation Functions?



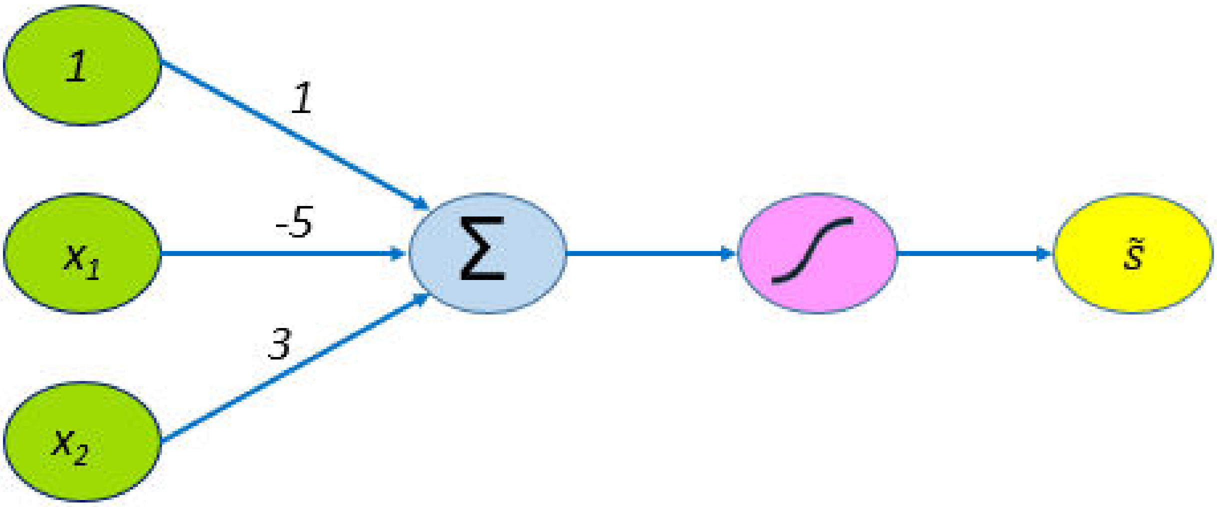
Example : **Separating green points from red points in the graph.**

Importance of Activation Functions

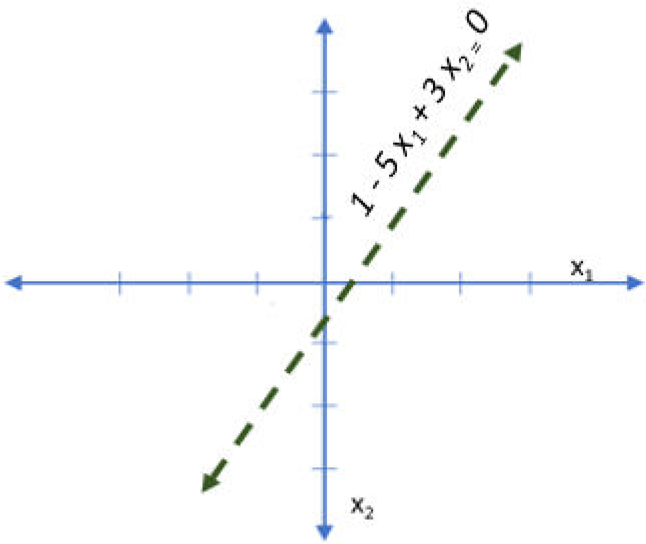




Example for Perceptron



$\tilde{s} = (1 - 5\,x_1 + 3\,x_2)$

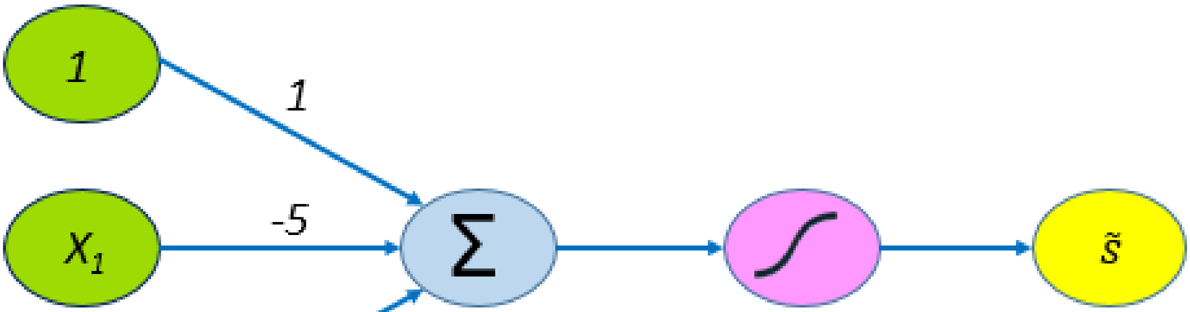


$W_0 = 1 \quad and \quad \mathbf{W} = \begin{bmatrix} -5 \\ 3 \end{bmatrix}$

$$\tilde{s} = g(\ W_0 + X^T W\ )$$

$$= g\left(\ 1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} -5 \\ 3 \end{bmatrix}\right)$$

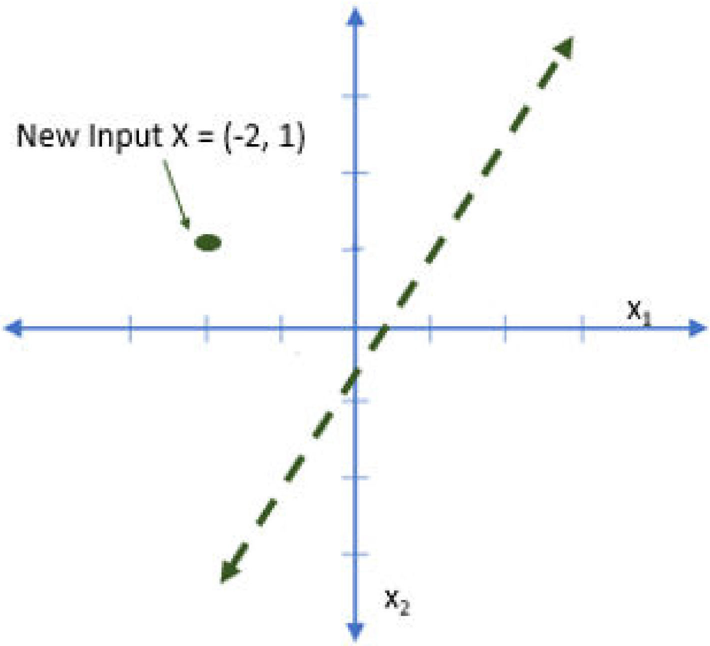
$$\tilde{s} = \underbrace{(\ 1 - 5\,x_1 + 3\,x_2)}_{Output}$$



Let new input X = $\begin{bmatrix} -2 \\ 1 \end{bmatrix}$

$$\tilde{s} = F(1 - (5 * -2) + (3 * 1))$$
$$= F(-6) \approx 0.002$$

$$\tilde{s} = (1 - 5x_1 + 3x_2)$$

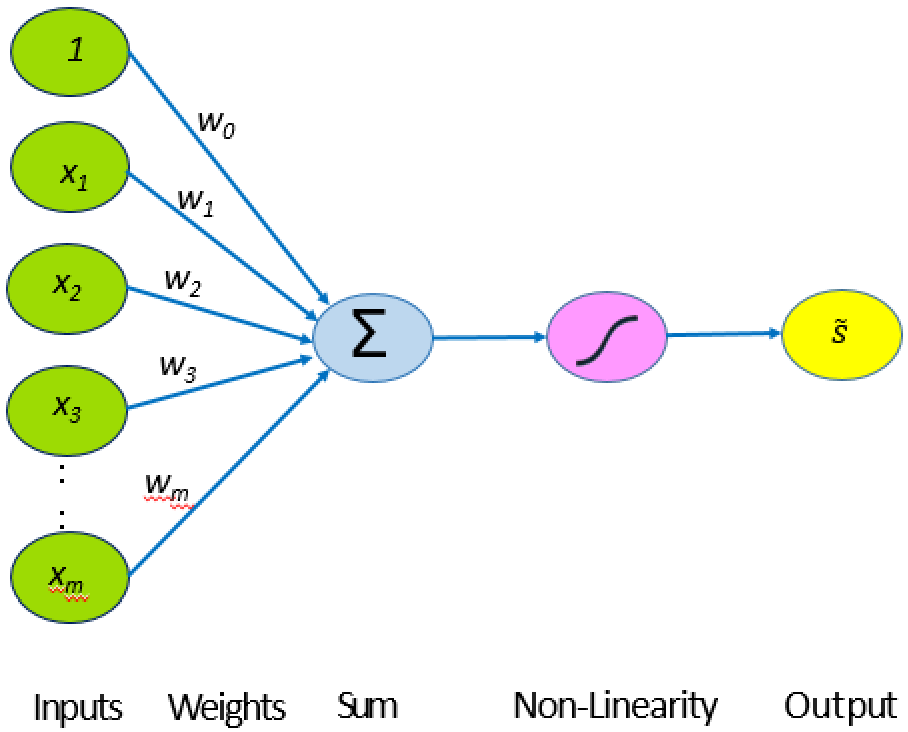


Building Neural Networks with Perceptron

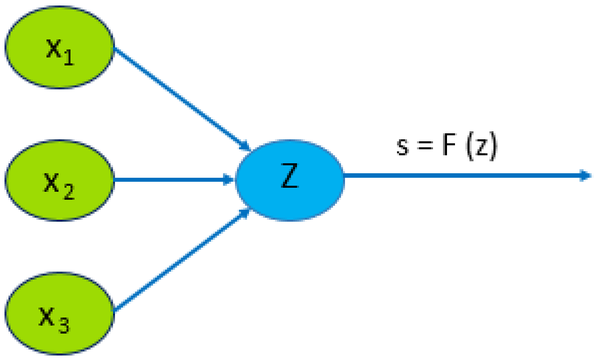
The Perceptron: Simplified

Simple equation of perceptron working:

$$\tilde{s} = F(\ w_0 + X^T W\ )$$

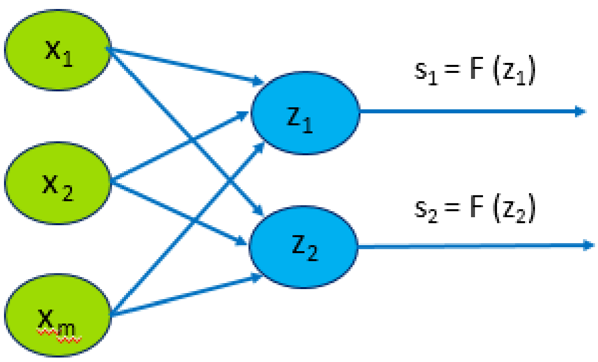


The Perceptron: Simplified



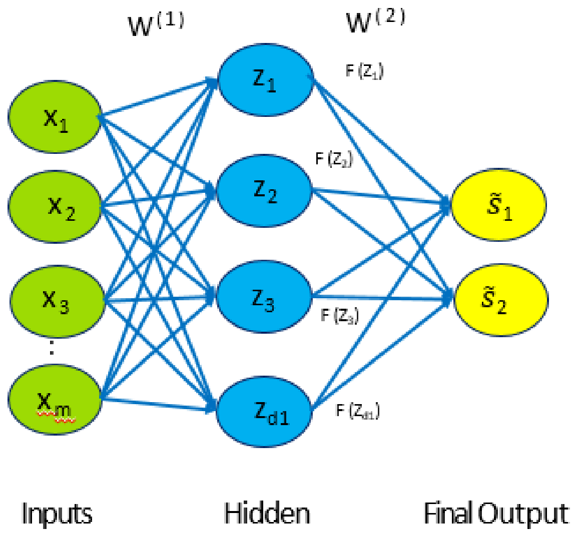
$$Z = w_0 + \sum_{j=1}^{m} x_j w_j$$

Multi Output Perceptron



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

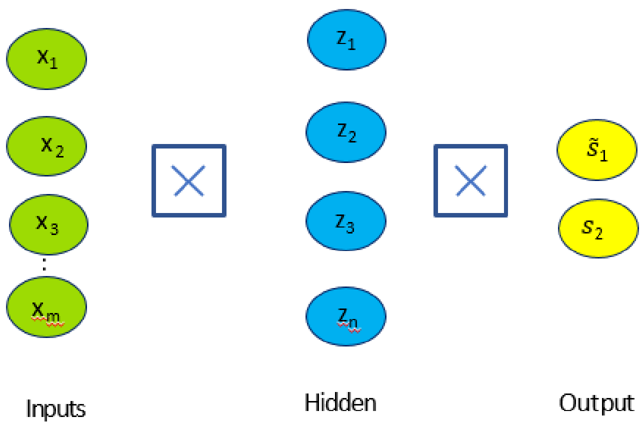
Single Layer Neural Network



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

$$\tilde{s} = F\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

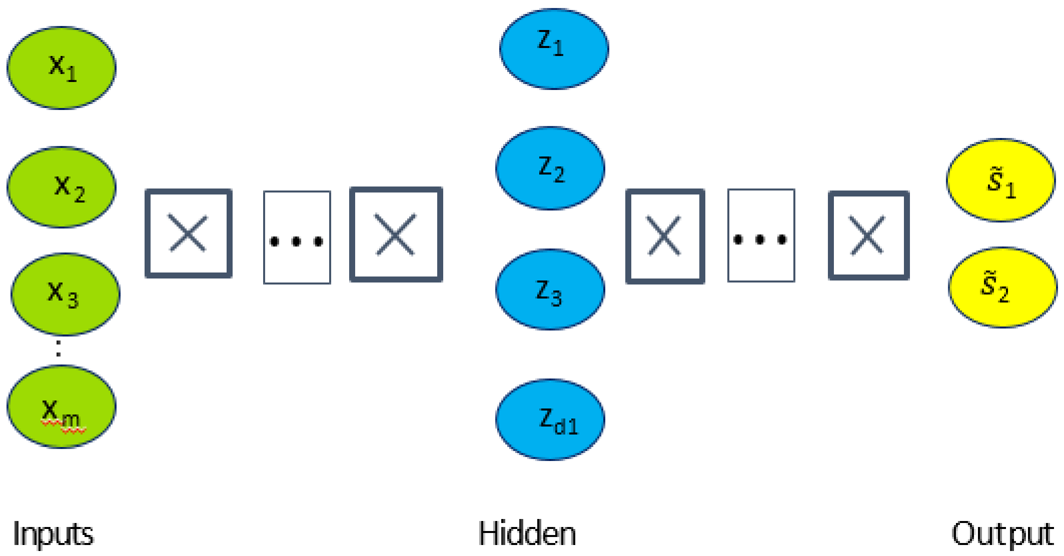
Multi Output Perceptron




```
from tf.keras.layers import *

inputs = Inputs(m)
hidden = Dense(d1)(inputs)  outputs =
Dense(2)(hidden)  model =
Model(inputs, outputs)
```

Deep Neural Network




```
from tf.keras.layers import *
inputs = Inputs(m)
hidden = Dense(d1)(inputs)
outputs = Dense(2)(hidden)
model = Model(inputs,
outputs)
```

$$Z_{k,i} = w_{0,i}^{(K)} + \sum_{j=1}^{d_{k-1}} F(z_{k-1,j})\, w_{j,i}^{(k)}$$

Applying Neural Networks

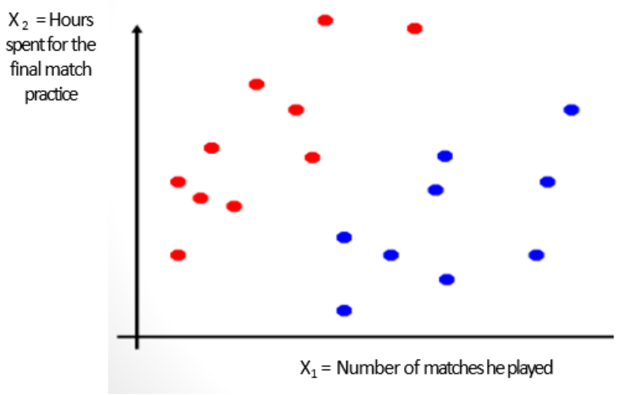
Example Problem

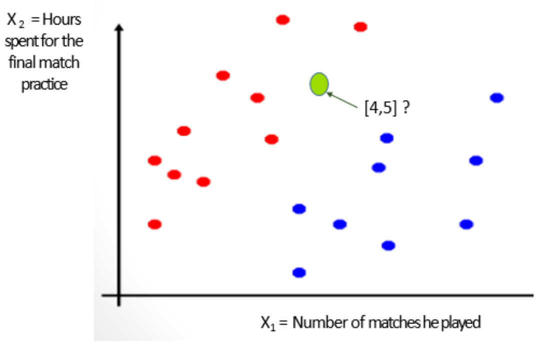
Will Mr. X Win the match?

Let's start with a simple two feature model

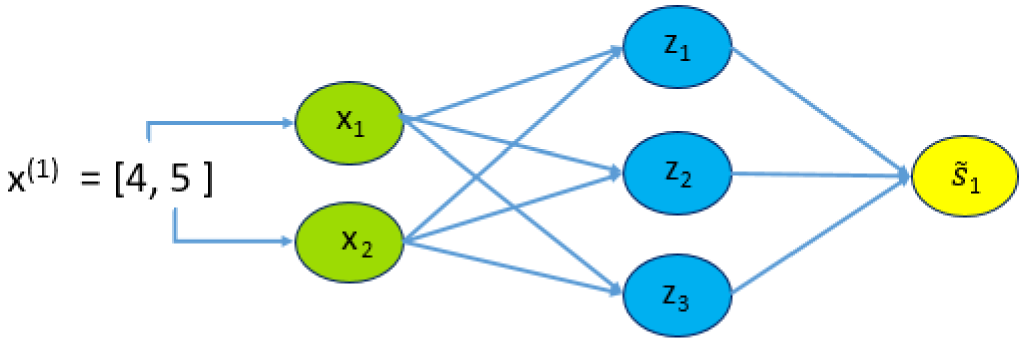
X_1 = Number of matches he played

X_2 = Hours spent for the final match practice

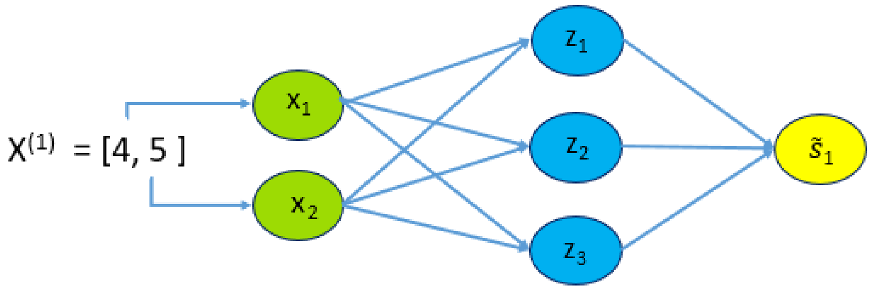




Example Problem: Will Mr. X Win the match?

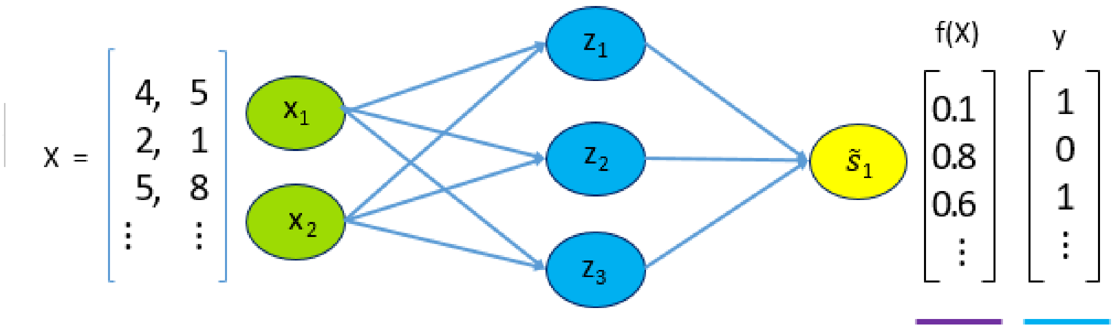


What is Quantifying Loss?



$$L\,(\,\underbrace{f\,(\,xi\,;\,\boldsymbol{W}\,)}_{\text{Predicted}},\ \underbrace{s^i}_{\text{Actual}}\,)$$

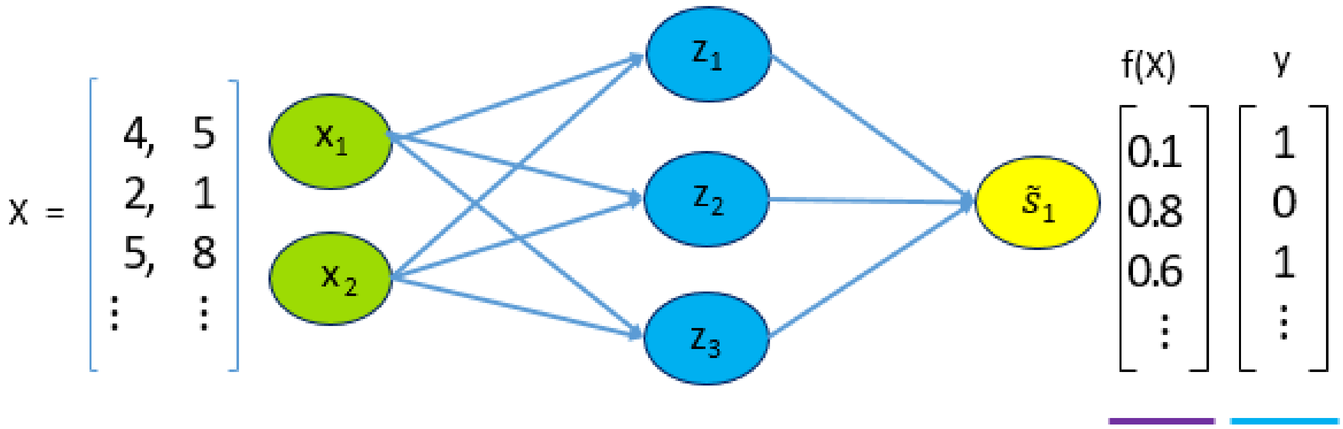
Empirical Loss



Also Known as
Empiirical loss
Cost function
Objective function
Empirical Risk

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} L\,(\,\underbrace{f\,(\,xi\,;\,\boldsymbol{W}),}_{\text{Predicted}}\ \underbrace{s^i}_{\text{Actual}}\,)$$

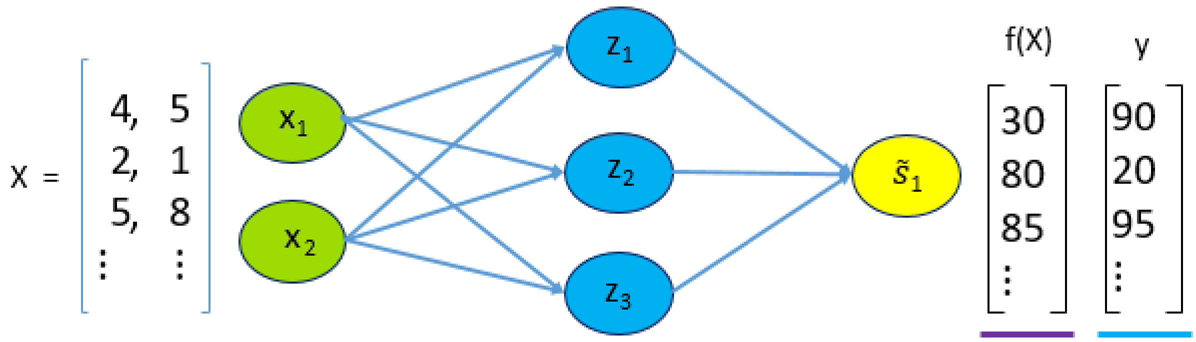
Binary Cross Entropy Loss



$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{s^i}_{\text{Actual}} \, log(\underbrace{f\, xi; W)}_{\text{Predicted}}) + (\underbrace{1 - si}_{\text{Actual}})\, log(\underbrace{1 - f(xi; W)}_{\text{Predicted}})$$

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

Mean Squared Error Loss



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

```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred)
```

Training Neural Networks

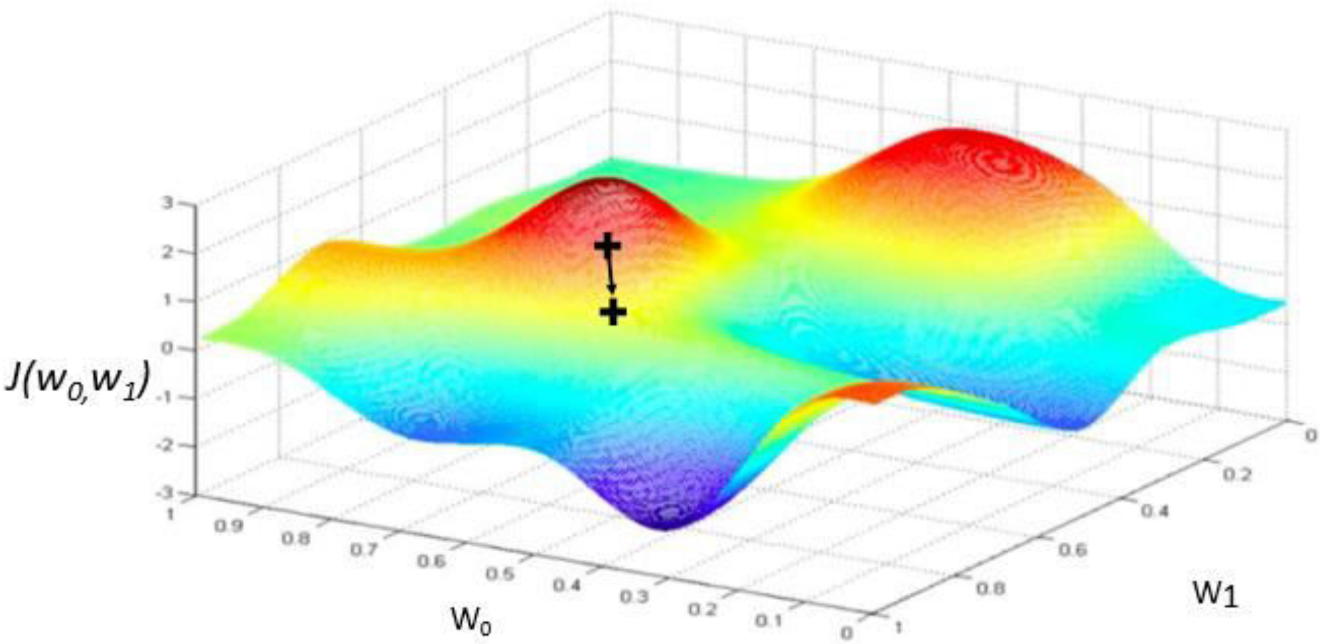
Loss Optimization

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin} \frac{1}{n} \sum_{i=1}^{n} L(f(x^i; \boldsymbol{W}), si)$$

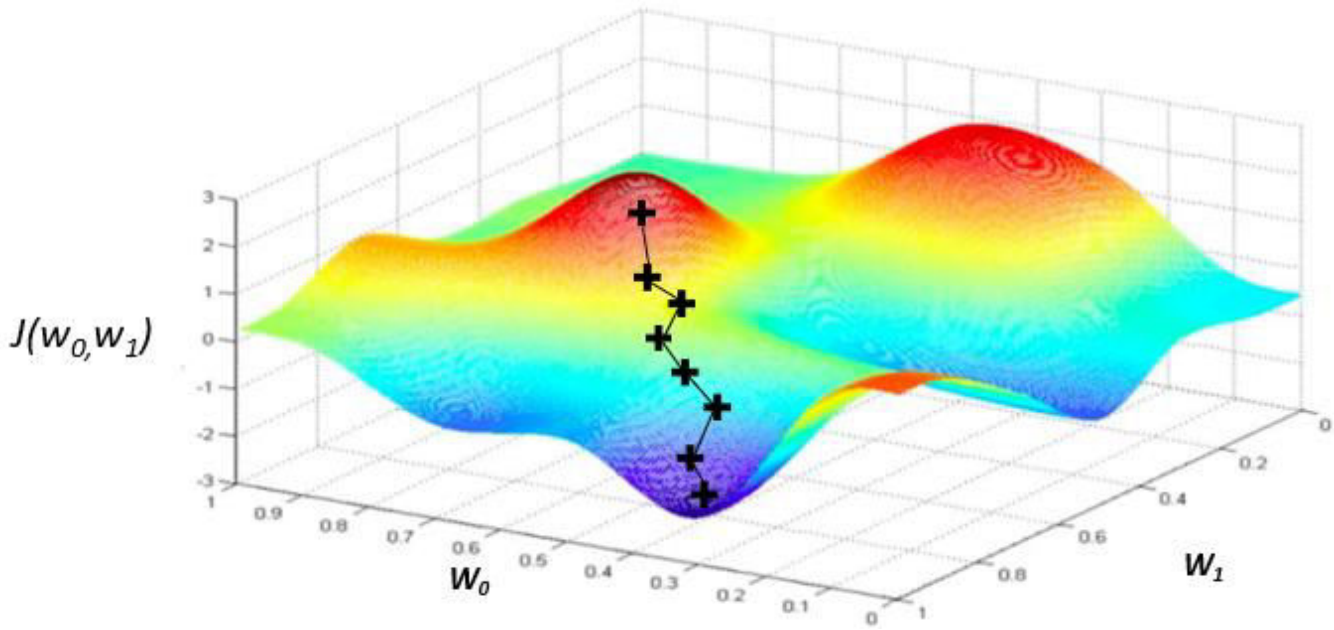
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{argmin}\, J(\boldsymbol{W})$$

Remember:

$$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \cdots\}$$



Gradient Descent



Algorithm for gradient descent:

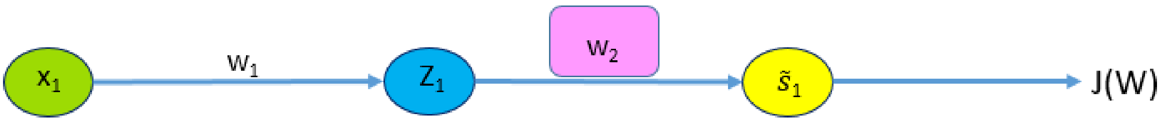
1. Initialize the weights randomly $\sim N(0,\sigma^2)$
2. Loop until finding the convergence:
3. Compute gradient, $\frac{\partial J(w)}{\partial w}$
4. Update weight, $W \leftarrow W - \eta \frac{\partial J(w)}{\partial W}$
5. Return weights

```
weights = tf.random_normal( )
```

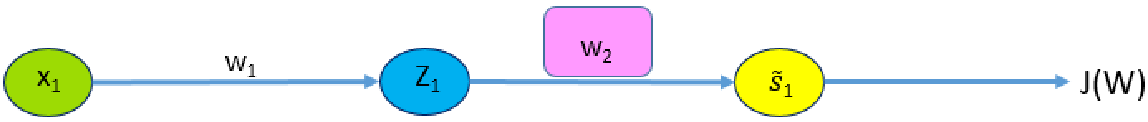
```
grads = tf.gradients(loss, weights)
```

```
weights_new = weights.assign(weights – lr * grads)
```

Computing Gradients: Backpropagation

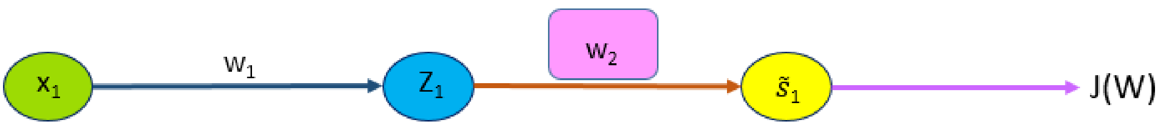


How does a small change in one weight (ex. W_2*) affect the final loss* **J(W)**?



Gradient loss of W with respect to w_2 $\quad \dfrac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w_2}}$

Let's use the chainrule!



$$\frac{\partial J(W)}{\partial (w_2)} = \frac{\partial J(W)}{\partial \tilde{s}} * \frac{\partial \tilde{s}}{\partial (w_2)}$$

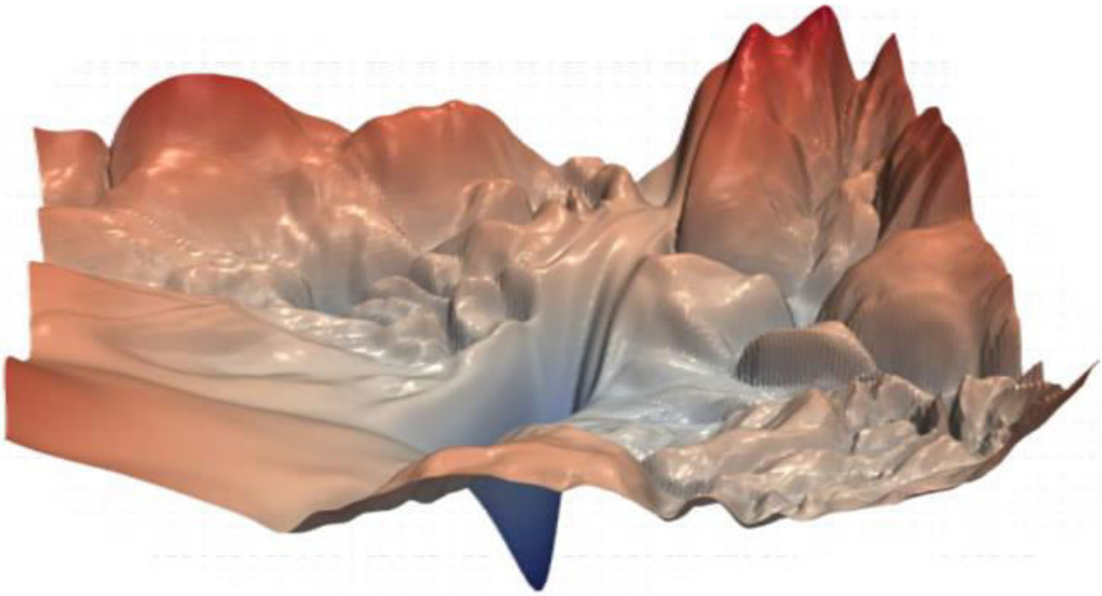
$$\frac{\partial J(W)}{\partial(w_1)} = \frac{\partial J(W)}{\partial \tilde{s}} * \frac{\partial \tilde{s}}{\partial(w_1)}$$

Apply chain rule! Apply chain rule!

$$\frac{\partial J(W)}{\partial(w_1)} = \frac{\partial J(W)}{\partial \tilde{s}} * \frac{\partial \tilde{s}}{\partial(Z_1)} * \frac{\partial Z_1}{\partial w_1}$$

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Loss Functions Can Be Difficult to Optimize

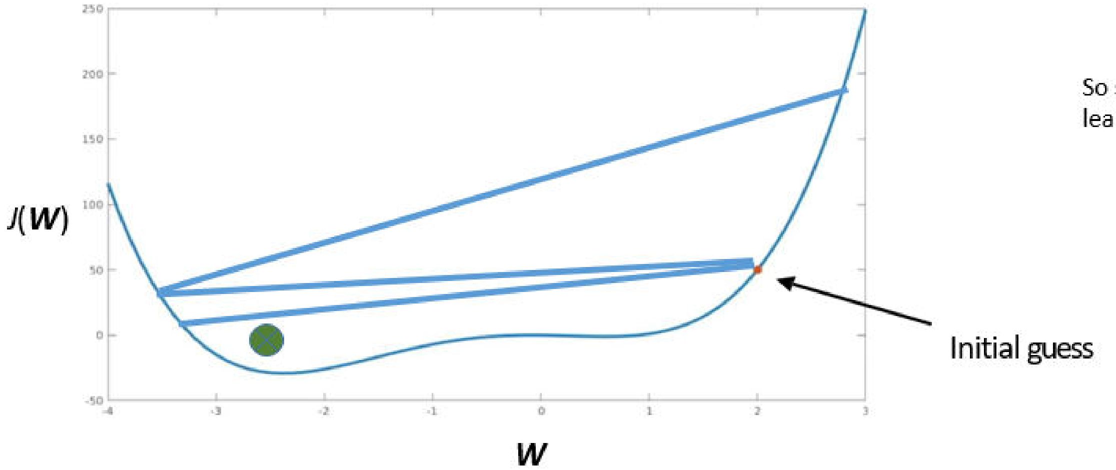
Remember:

Optimization through gradient descent

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

Setting the Learning Rate

- Large learning rates overshoot, become unstable and diverge which is more undesirable.



So setting a learning rate

How to deal with setting learning rate?

Idea 1:

Hit and trial Method: Trying different learning rates and see what works correctly

Idea 2:

Do something smarter!

Design an adaptive learning rate: Which "adapts" to the landscape

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Adaptive Learning Rate Algorithms

• Adagrad	tf.train.AdagradOptimizer	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• Adadelta	tf.train.AdadeltaOptimizer	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adam	tf.train.AdamOptimizer	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Momentum	tf.train.MomentumOptimizer	Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.
• RMSProp	tf.train.RMSPropOptimizer	

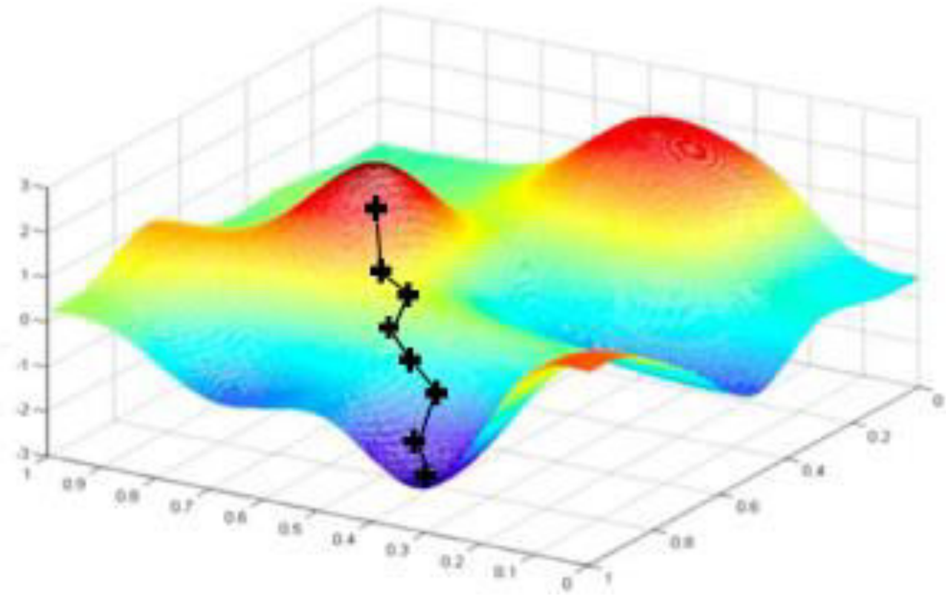
Neural Networks in Practice: Mini-batches

Gradient Descent

Algorithm for gradient descent:

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

Can be very computational to compute!



Stochastic Gradient Descent

Algorithm for gradient descent:

1. Initialize weights the randomly $\sim N(0, \sigma^2)$
2. Loop until finding the convergence:
3. Pick single data point i
4. Compute gradient, $\boxed{\frac{\partial J(w)}{\partial w}}$
5. Update weight, $W \leftarrow W - \eta \frac{\partial J(w)}{\partial W}$
6. Return weights

Easy to compute but very noisy (stochastic)!

Algorithm for gradient descent:

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

Fast to compute and a much better estimate of the true gradient!

Mini-batches while training

- Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.
- Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.
- **More accurate estimation of gradient**
  - Smoother convergence Allows for larger learning rates

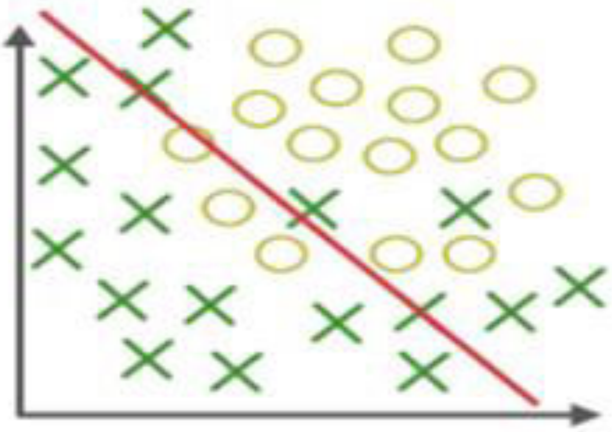
More accurate estimation of gradient
Smoother convergence Allows for larger learning rates

Mini-batches lead to fast training

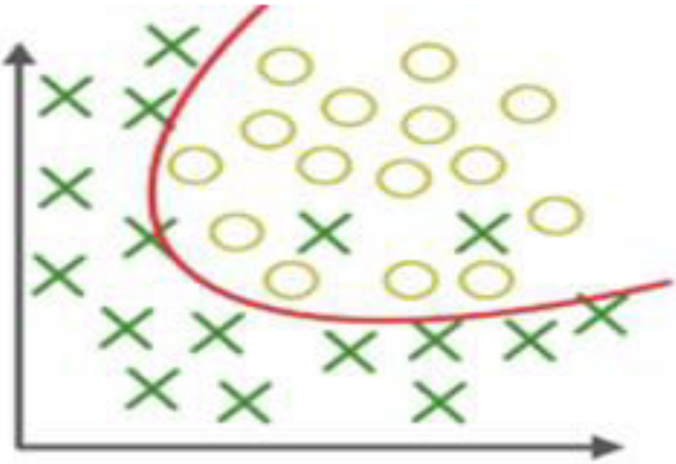
Increase the computation and achieve increased speed on GPU’s

Neural Networks in Practice: Overfitting

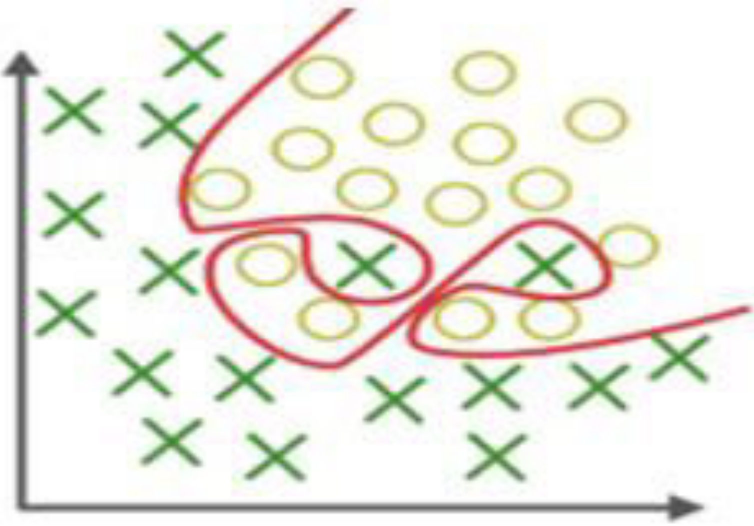
The Problem of Overfitting



Under-fitting
Too simple to explain the variance



Appropriate-Fitting



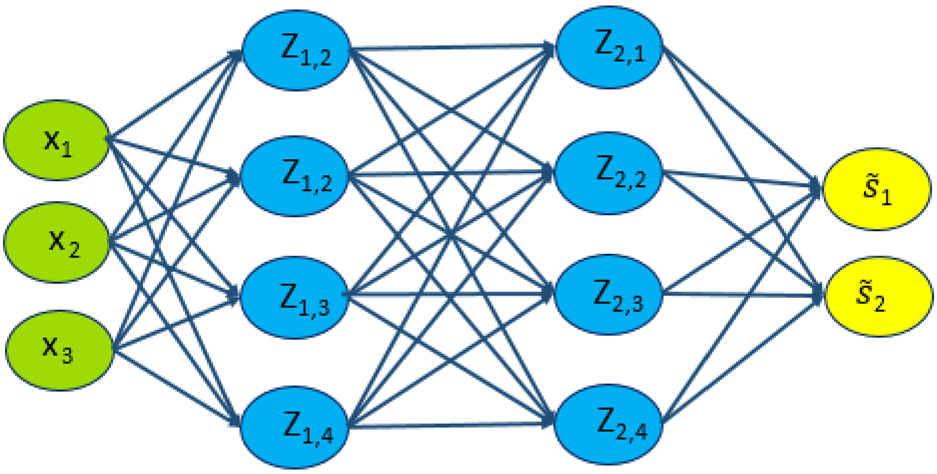
Over-fitting
Force fitting, too complex, extra parameters

Regularization

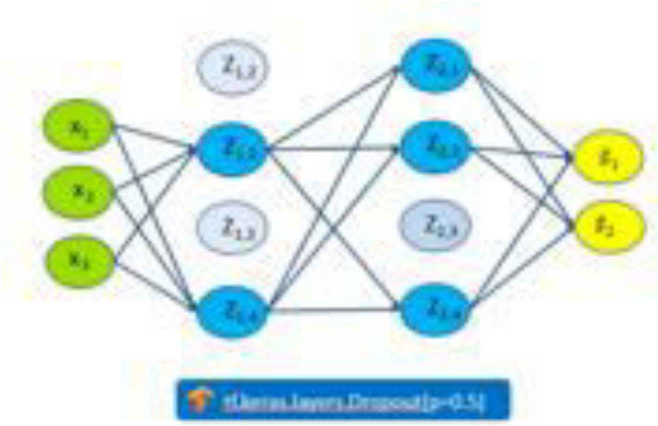
What is it?
Technique that constrains our optimization problem to discourage complex models

Why do we need it?
Improve generalization of our model on unseen data

Regularization 1: Dropout



- During training, randomly set some activations to 0

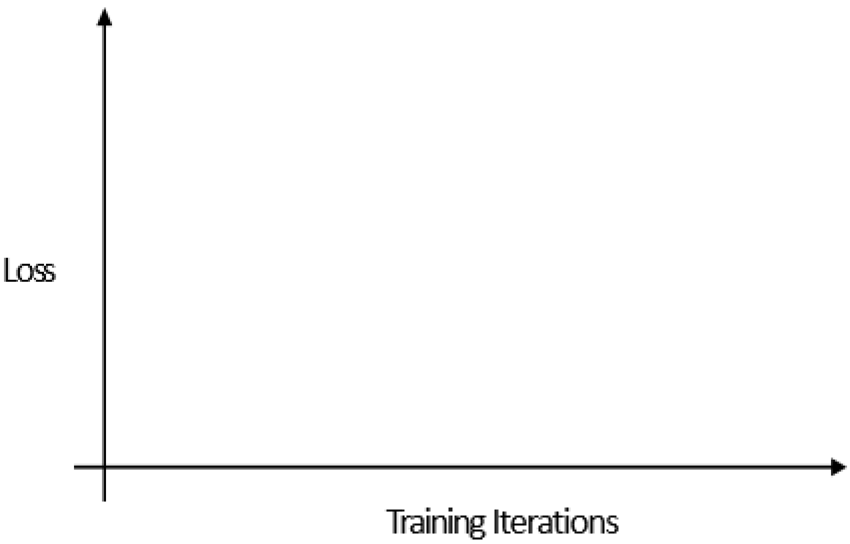


- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

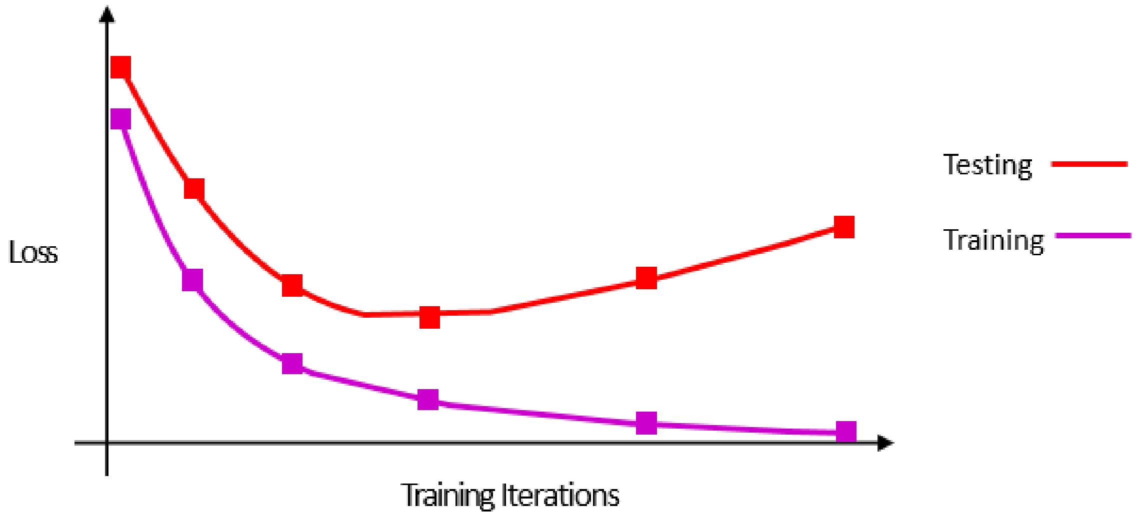
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

Regularization 2: Early Stopping

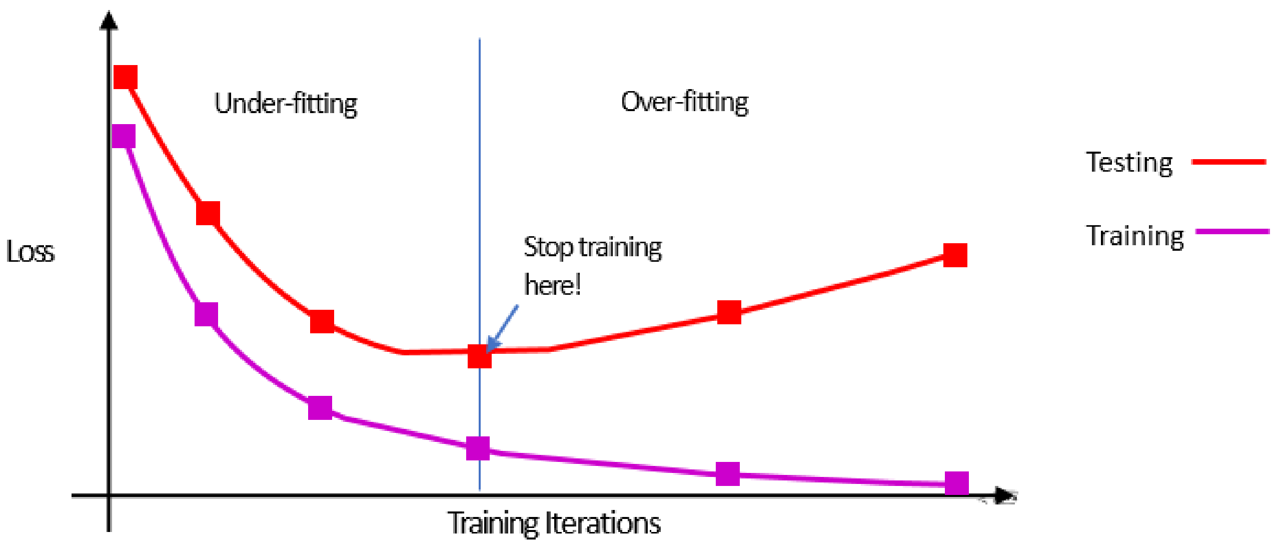
- Stop training before we have a chance to overfit



- Stop training before we have a chance to overfit



- Stop training before we have a chance to overfit



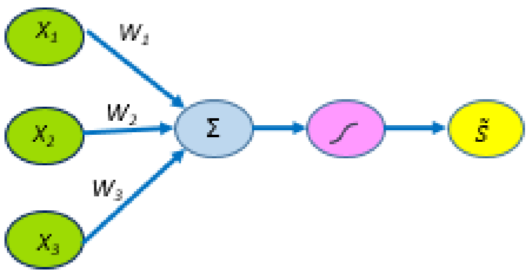
Core Foundation Review

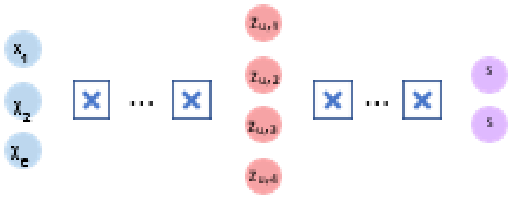
Perceptron

- Structural building blocks
- Nonlinear activation functions

Neural Networks

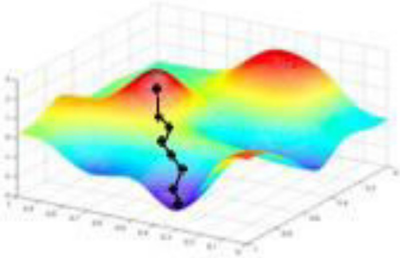
- Merging Perceptrons to form neural networks
- Optimization through backpropagation method





Training in Practice

- Adaptive learning
- Batching
- Regularization



Queries ??