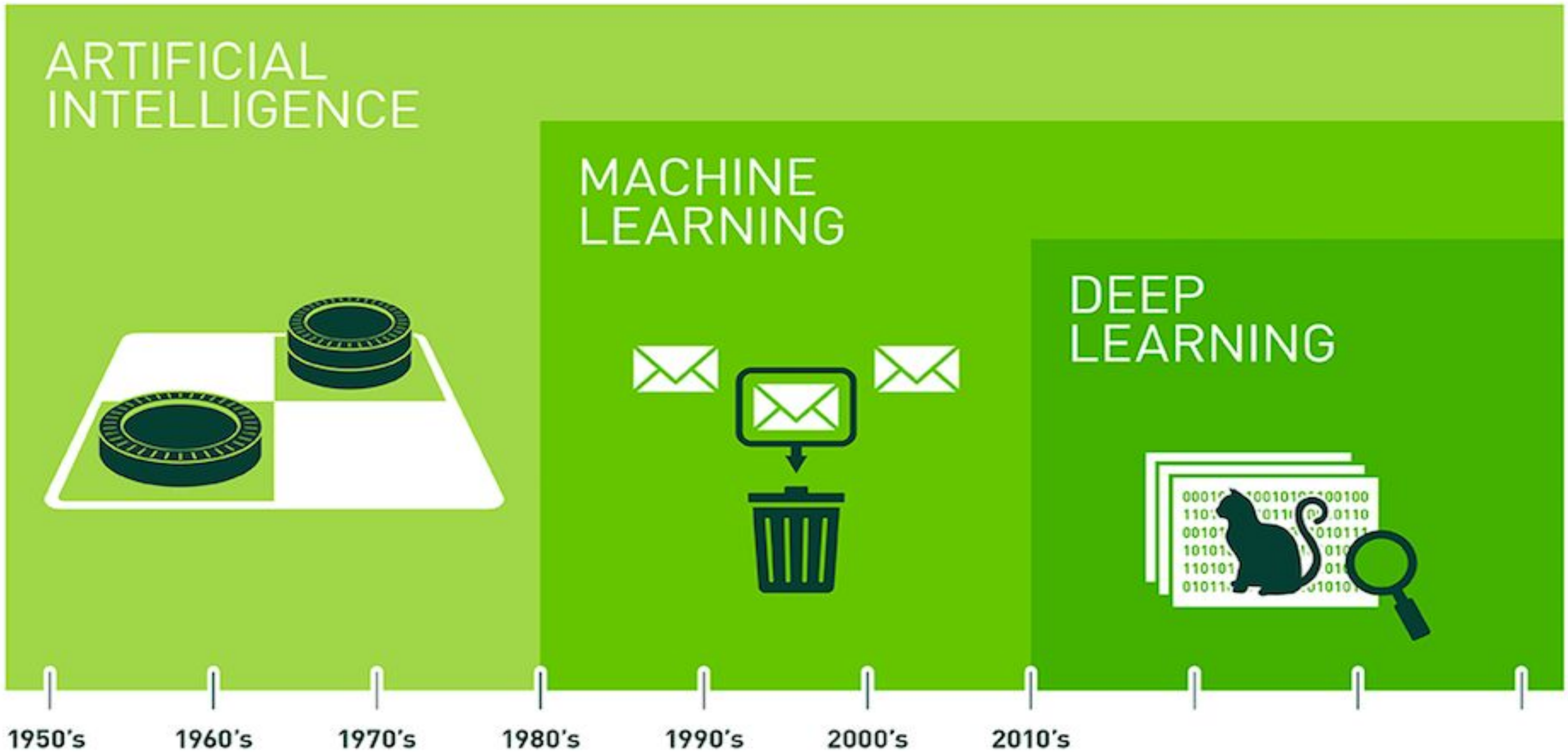
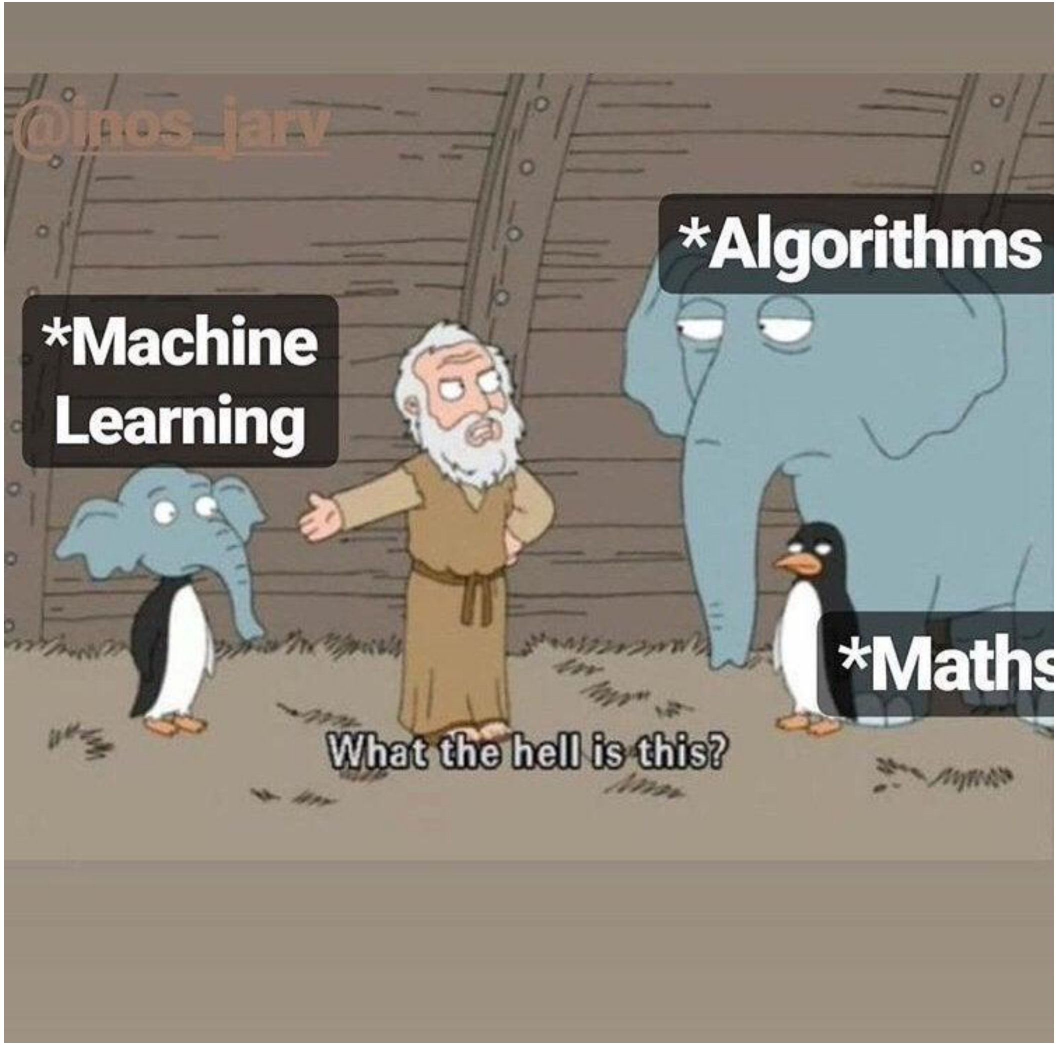
Introduction to Deep Learning

ML and DL


ARTIFICIAL INTELLIGENCE
MACHINE LEARNING
DEEP LEARNING
1950's
1960's
1970's
1980's
1990's
2000's
2010's


@inos_jarv
*Algorithms
*Machine Learning
*Maths
What the hell is this?

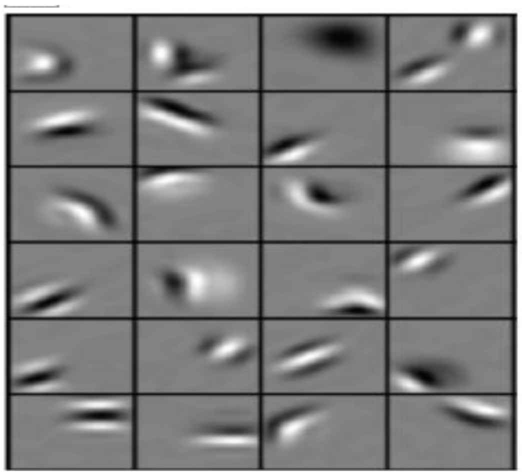
i am ai

Why Deep Learning and Why Now?

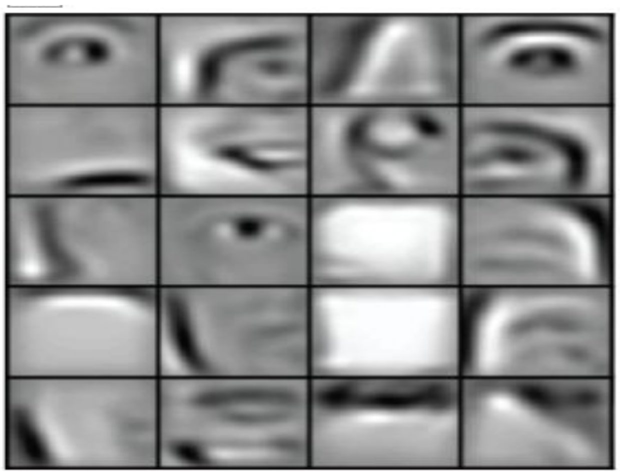
Why Deep Learning?

Hand engineered features are time consuming, brittle, and not scalable in practice
Can we learn the underlying features directly from data ?

Low Level Features



Mid Level Features



High Level Features

Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

⋮

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

⋮

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

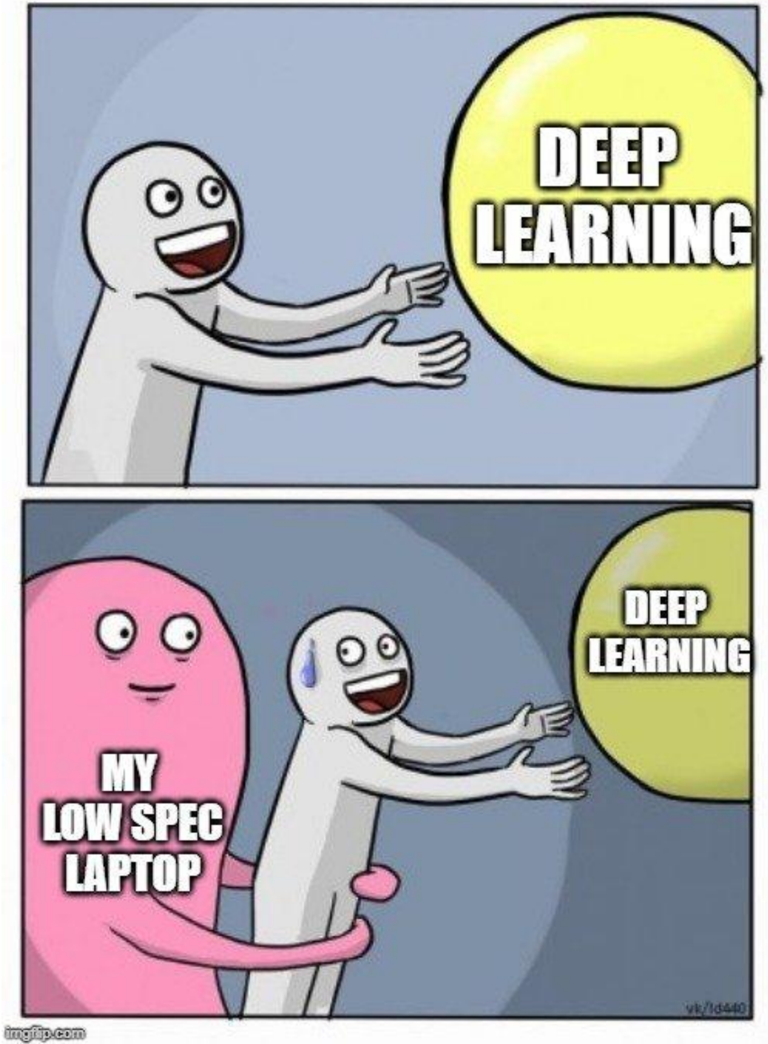
WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow


TensorFlow
OP 1
OP 2
A


DEEP
LEARNING
MY
LOW SPEC
LAPTOP
DEEP
LEARNING
imgflip.com

TensorFlow behind the scenes

What is TensorFlow?

What is TensorFlow?

- Open Source **Machine Learning** Library, **Apache 2.0** License
- Useful for **Deep Learning**
- Research **and** Production

. . .

Model building: from simple to arbitrarily flexible

Progressive disclosure of complexity

Sequential API
+ built-in layers

New users, simple models

Functional API
+ built-in layers

Engineers with standard use cases

Functional API
- **+ Custom layers**
- **+ Custom metrics**
- **+ Custom losses**

Engineers requiring increasing control

Subclassing: write everything yourself from scratch

Researchers

TensorFlow behind the scenes

What is TensorFlow?

Computational Graph

$$\boxed{(a+b)\times(c+d)}$$

- Sequential → $\underbrace{(a + b)} \times \underbrace{(c + d)}$

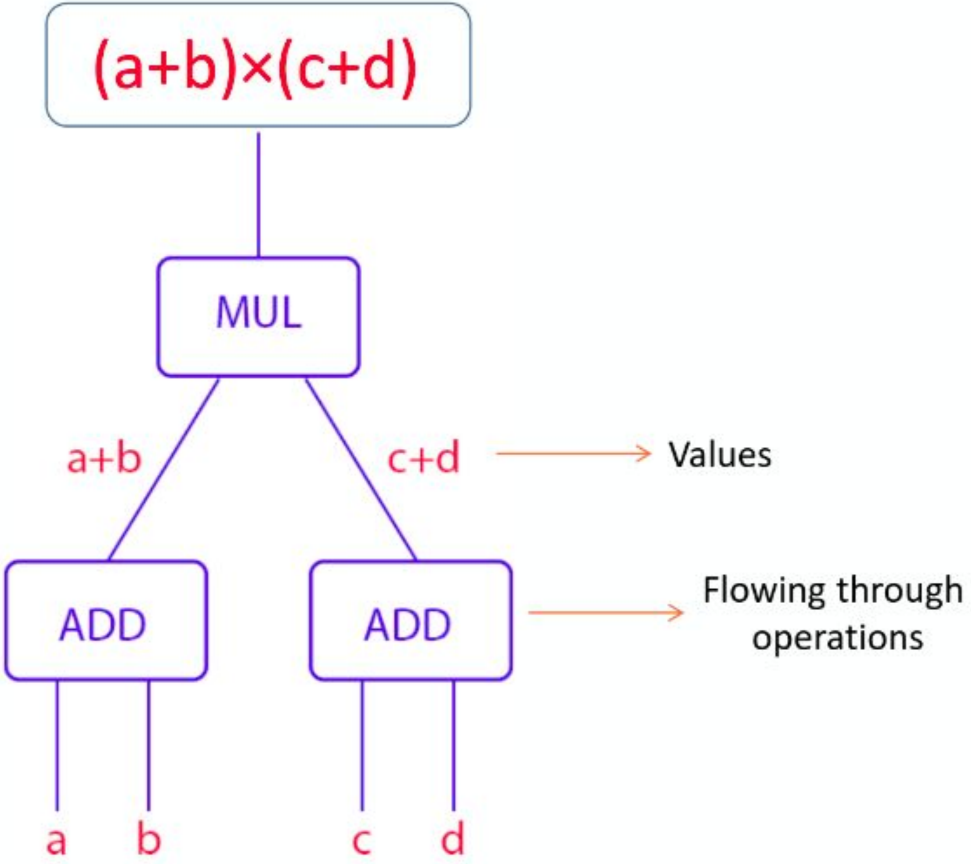
INDEPENDENT!

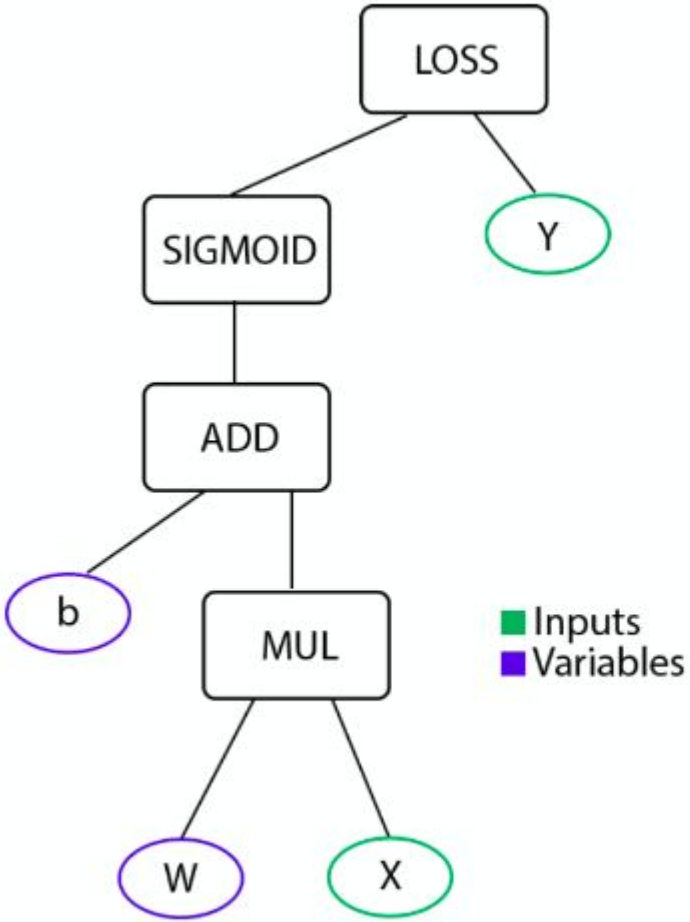
$(a+b)\times(c+d)$

- Sequential → $(a + b) \times (c + d)$

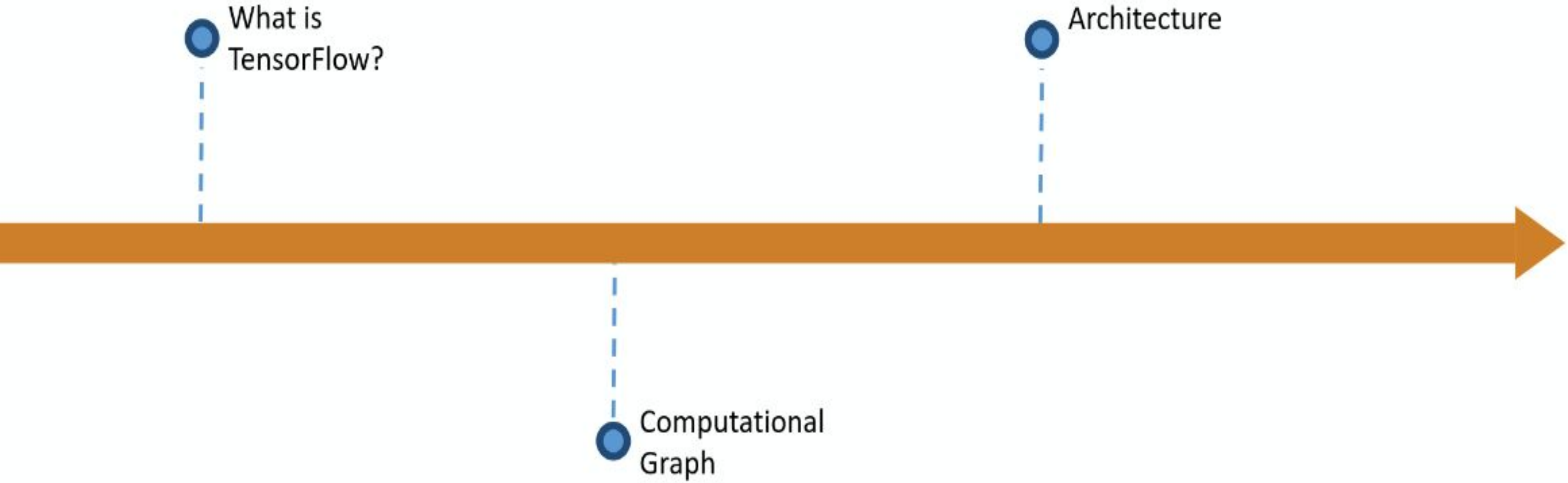
- Parallel → $\begin{cases} (a + b) \\ \times \\ (c + d) \end{cases}$

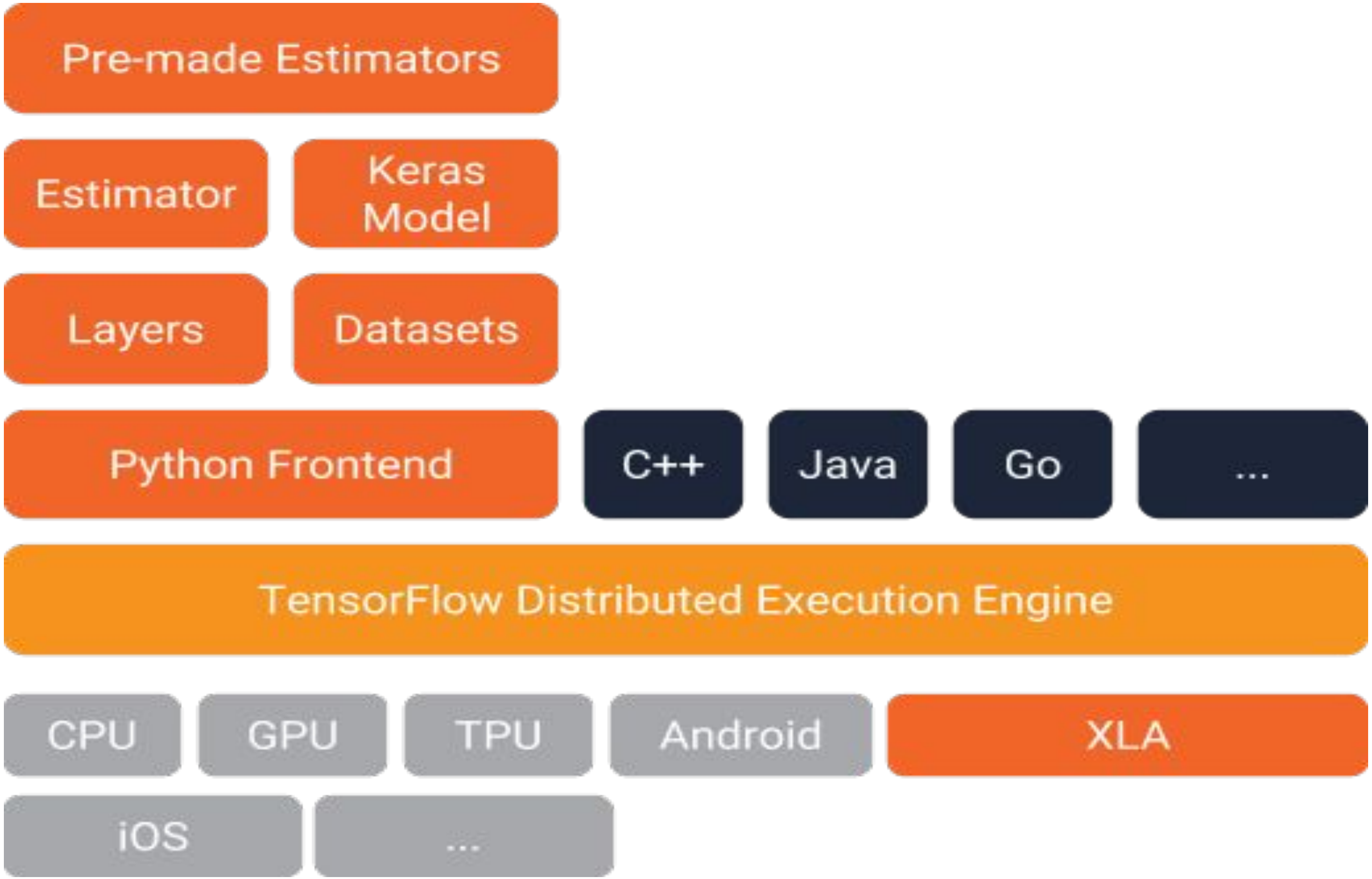
Computational Graph




LOSS
SIGMOID
Y
ADD
b
MUL
Inputs
Variables
W
X

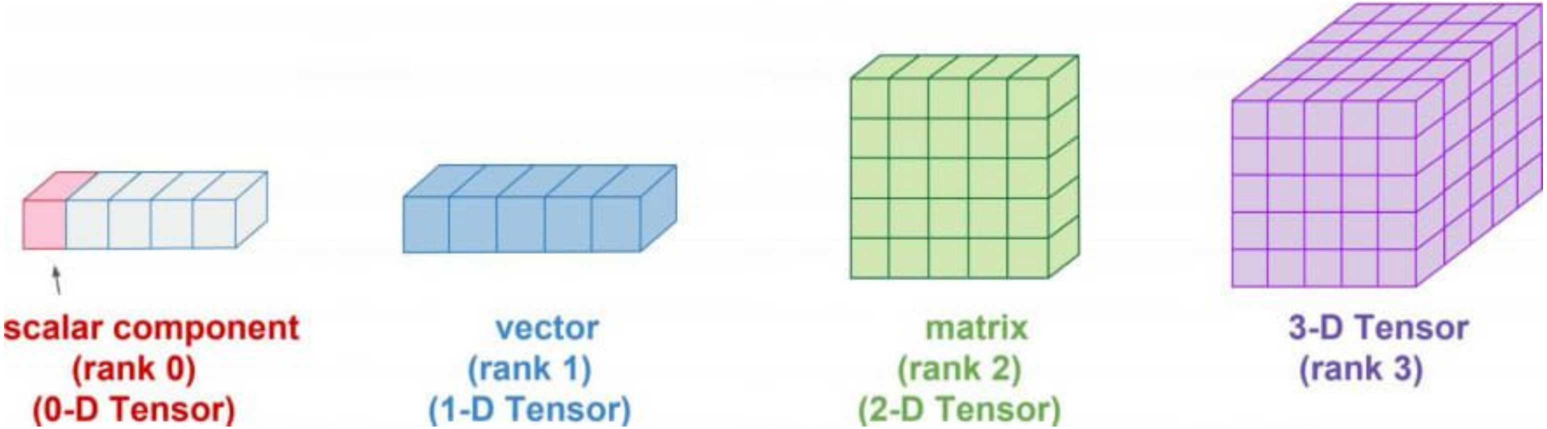
TensorFlow behind the scenes




Pre-made Estimators
Estimator
Keras Model
Layers
Datasets
Python Frontend
C++
Java
Go
...
TensorFlow Distributed Execution Engine
CPU
GPU
TPU
Android
XLA
iOS
...

Tensors

A Tensor is a multi-dimensional array. Similar to NumPy ndarray objects, tf.Tensor objects have a data type and a shape. Additionally, tf.Tensors can reside in accelerator memory (like a GPU). TensorFlow offers a rich library of operations (tf.add, tf.matmul, tf.multiply etc.) that consume and produce tf.Tensors. These operations automatically convert native Python types.



tensors

```
tf.ones([3,4],dtype=tf.float32)
```

```
<tf.Tensor: shape=(3, 4), dtype=float32,
numpy=
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)>
```

```
tf.zeros([2,2],dtype=tf.float32)
```

```
tf.random.normal(shape=[2,3])
```

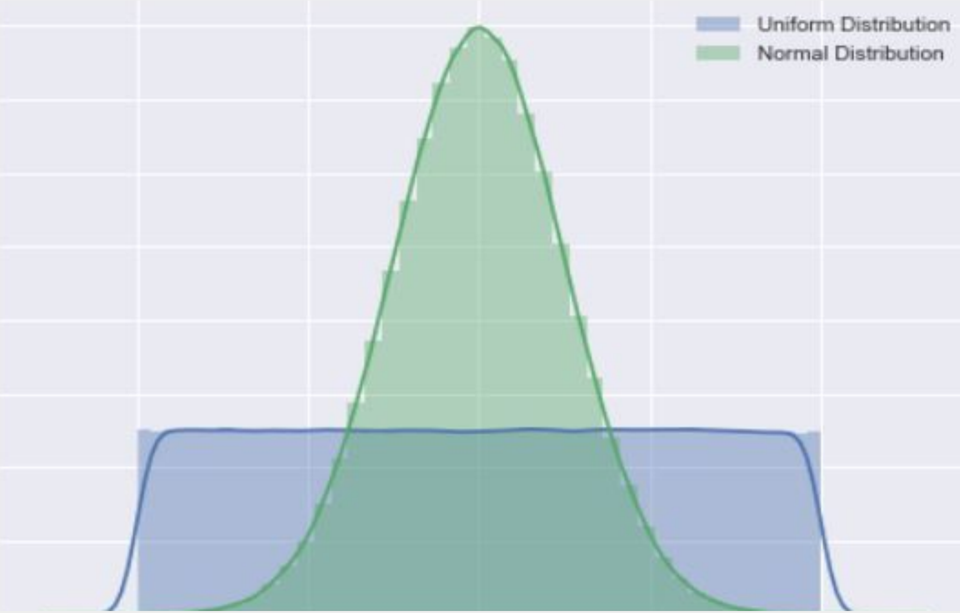
```
<tf.Tensor: shape=(2, 2), dtype=float32,
numpy=
array([[ 0.4381552, -1.5851315],
       [ 1.5874195, -2.1158957]],
dtype=float32)>
```

```
tf.random.uniform(shape=[2,3])
```

```
<tf.Tensor: shape=(2, 2), dtype=float32,
numpy=
array([[0.2659924 , 0.08474123],
       [0.36174548, 0.07563508]],
dtype=float32)>
```

Normal Distribution Vs Uniform Distribution

Normal Distribution is a probability distribution which peaks out in the middle and gradually decreases towards both ends of axis. It is also known as gaussian distribution and bell curve because of its bell like shape.
Uniform Distribution is a probability distribution where probability of x is constant. That is to say, all points in range are equally likely to occur consequently it looks like a rectangle.

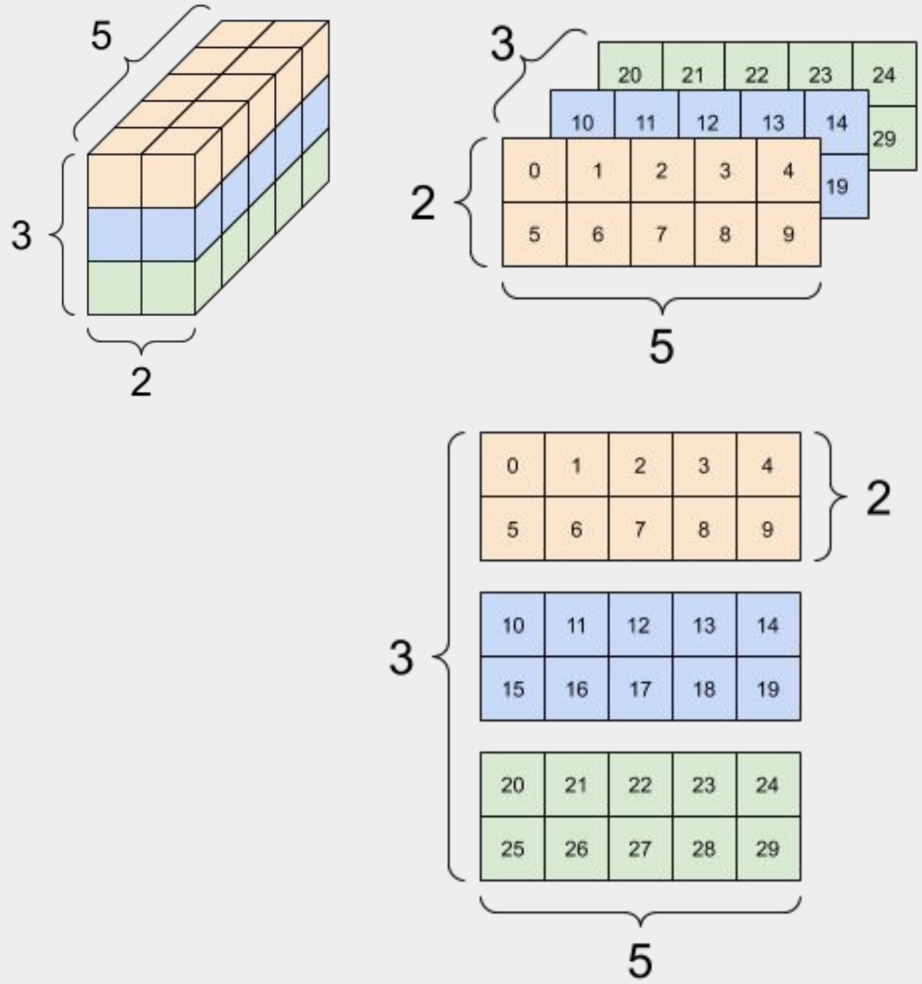


Constant

Creates a constant tensor from a tensor-like object.

A 3-axis tensor, shape: [3, 2, 5]

```
tf.constant([[
  [[0, 1, 2, 3, 4],
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]]]])
```



Variables

A TensorFlow variable is the recommended way to represent shared, persistent state your program manipulates.

Variables are created and tracked via the **tf.Variable** class. A **tf.Variable** represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like **tf.keras** use **tf.Variable** to store model parameters.

```
tf.Variable(
    initial_value=None, trainable=None, validate_shape=True,
    name=None, dtype=None)
```

basic math on tensors

You can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.

```
tf.add(a,b)
```

```
tf.multiply(a,b)
```

```
tf.matmul(a,b)
```

Input:	Result:
$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 2 \\ 2 & 7 \end{pmatrix}$	$\begin{pmatrix} 13 & 27 \\ 11 & 17 \end{pmatrix}$

Input: $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 2 \\ 2 & 7 \end{pmatrix}$	(1*1) + (2*3) + (3*2) = 13
Input: $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 2 \\ 2 & 7 \end{pmatrix}$	(1*2) + (2*2) + (3*7) = 27
Input: $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 2 \\ 2 & 7 \end{pmatrix}$	(3*1) + (2*3) + (1*2) = 11
Input: $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 2 \\ 2 & 7 \end{pmatrix}$	(3*2) + (2*2) + (1*7) = 17

$$\begin{bmatrix} 4 & 8 \\ 3 & 7 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 4+1 & 8+0 \\ 3+5 & 7+2 \end{bmatrix}$$

Tensor to Numpy

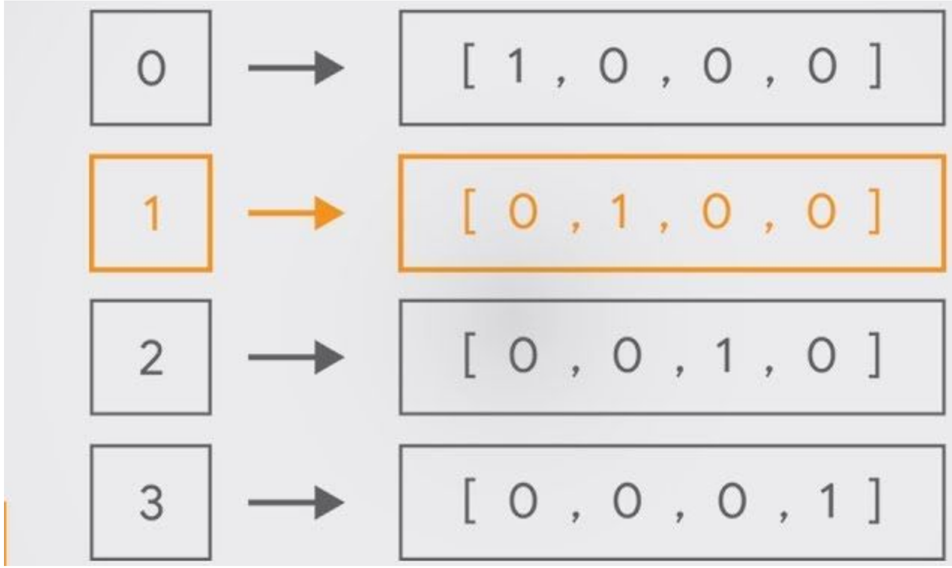
You can convert a tensor to a NumPy array either using np.array or the tensor.numpy method:

```
np.array(tensor)
```

```
tensor.numpy()
```

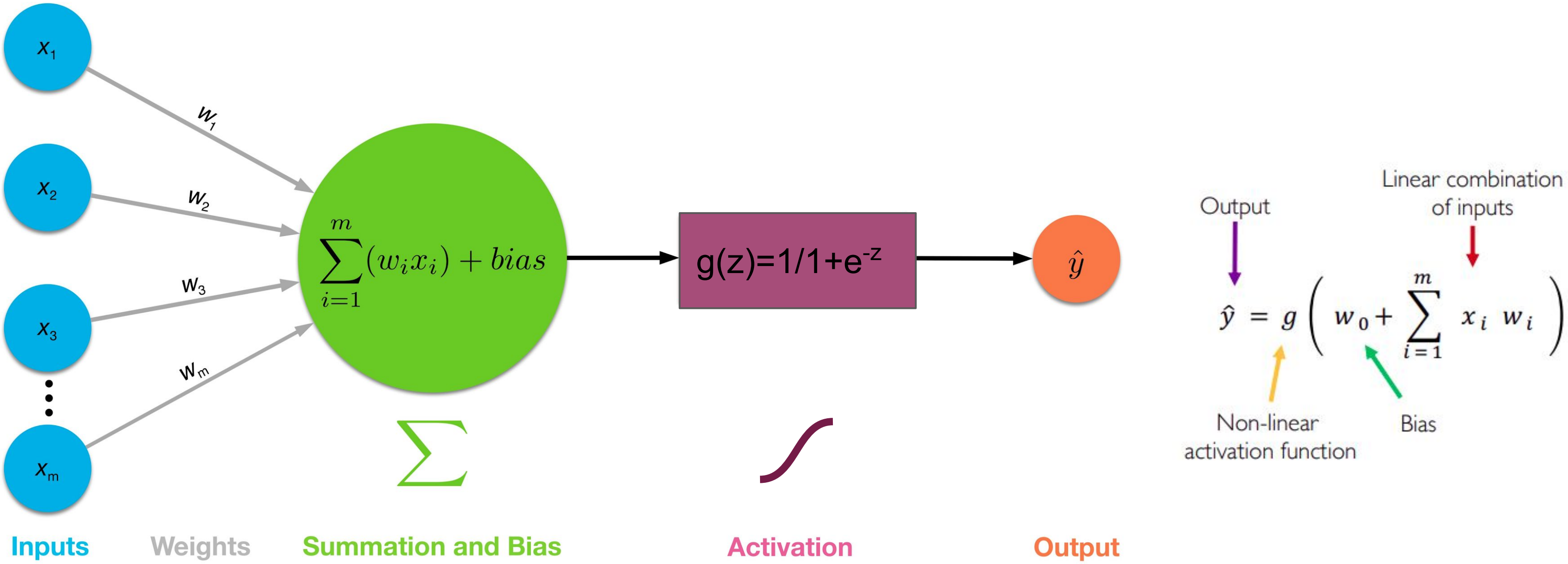
One-hot encoding

In machine learning, a one-hot is a group of bits among which the legal combinations of values are only those with a single high bit and all the others low. A similar implementation in which all bits are '1' except one '0' .

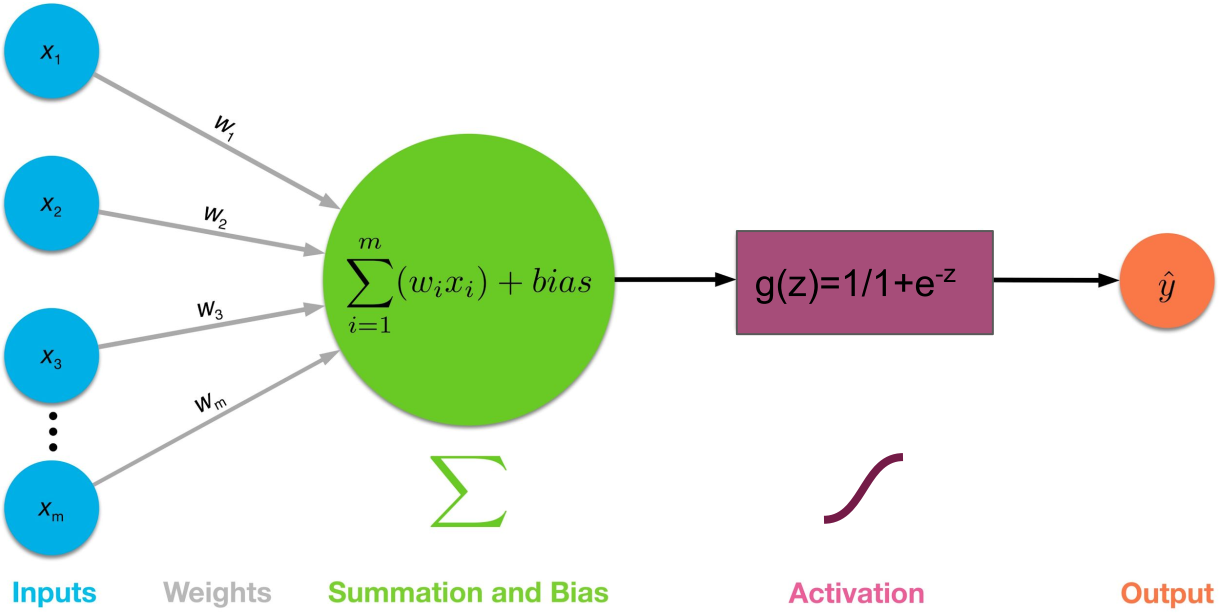


The Perceptron The structural building block of deep learning

The Perceptron: Forward Propagation



The Perceptron: Forward Propagation

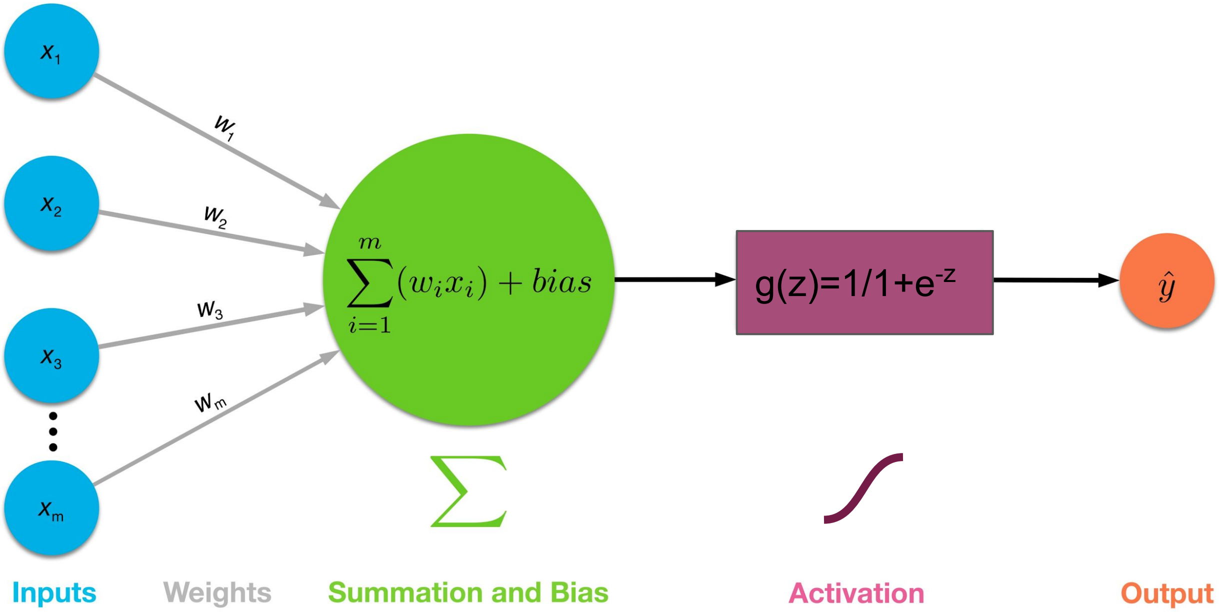


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i \right)$$

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

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

The Perceptron: Forward Propagation

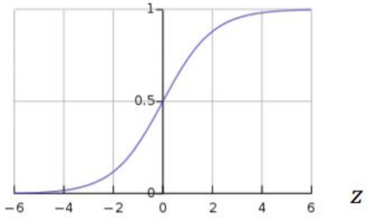


Activation Functions

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

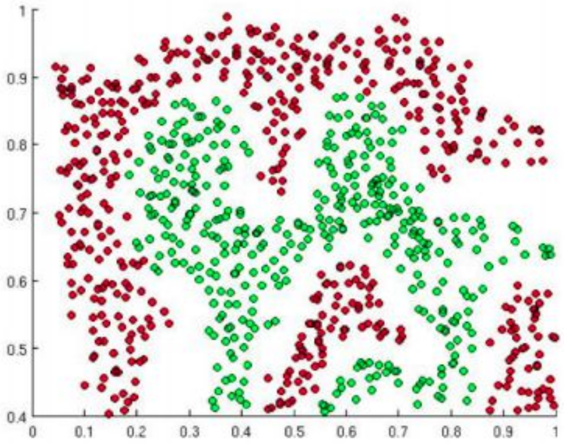
- Example: sigmoid function

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



Importance of Activation Functions

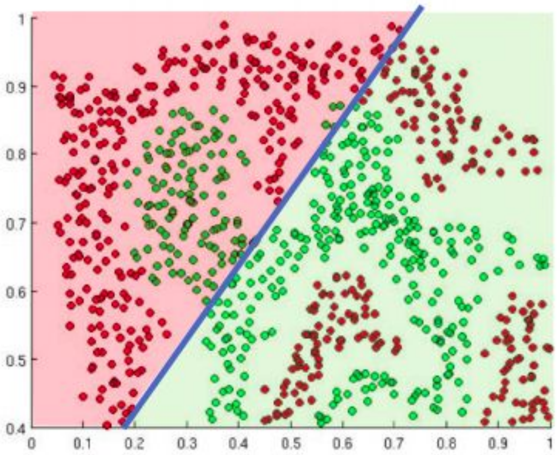
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



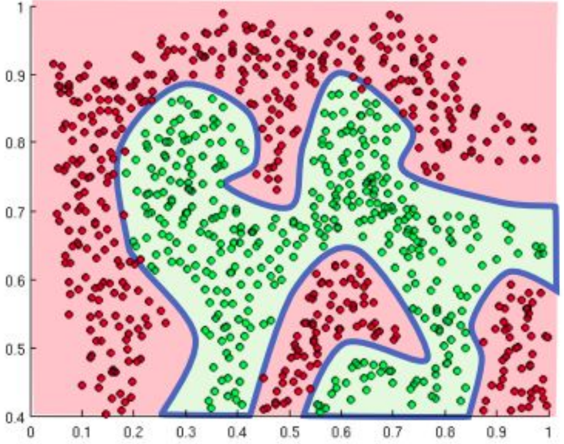
What if we wanted to build a Neural Network to
distinguish green vs red points?

Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*



Linear Activation functions produce linear decisions no matter the network size

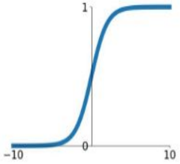


Non-linearities allow us to approximate arbitrarily complex functions

Common Activation Functions

Sigmoid

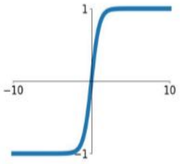
$\sigma(x) = \frac{1}{1+e^{-x}}$



tf.nn.sigmoid(x)

tanh

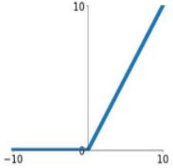
$\tanh(x)$



tf.nn.tanh(x)

ReLU

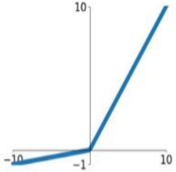
$\max(0, x)$



tf.nn.relu(x)

Leaky ReLU

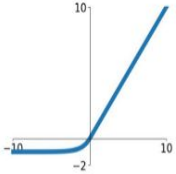
$\max(0.1x, x)$



tf.nn. leaky_relu(x)

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

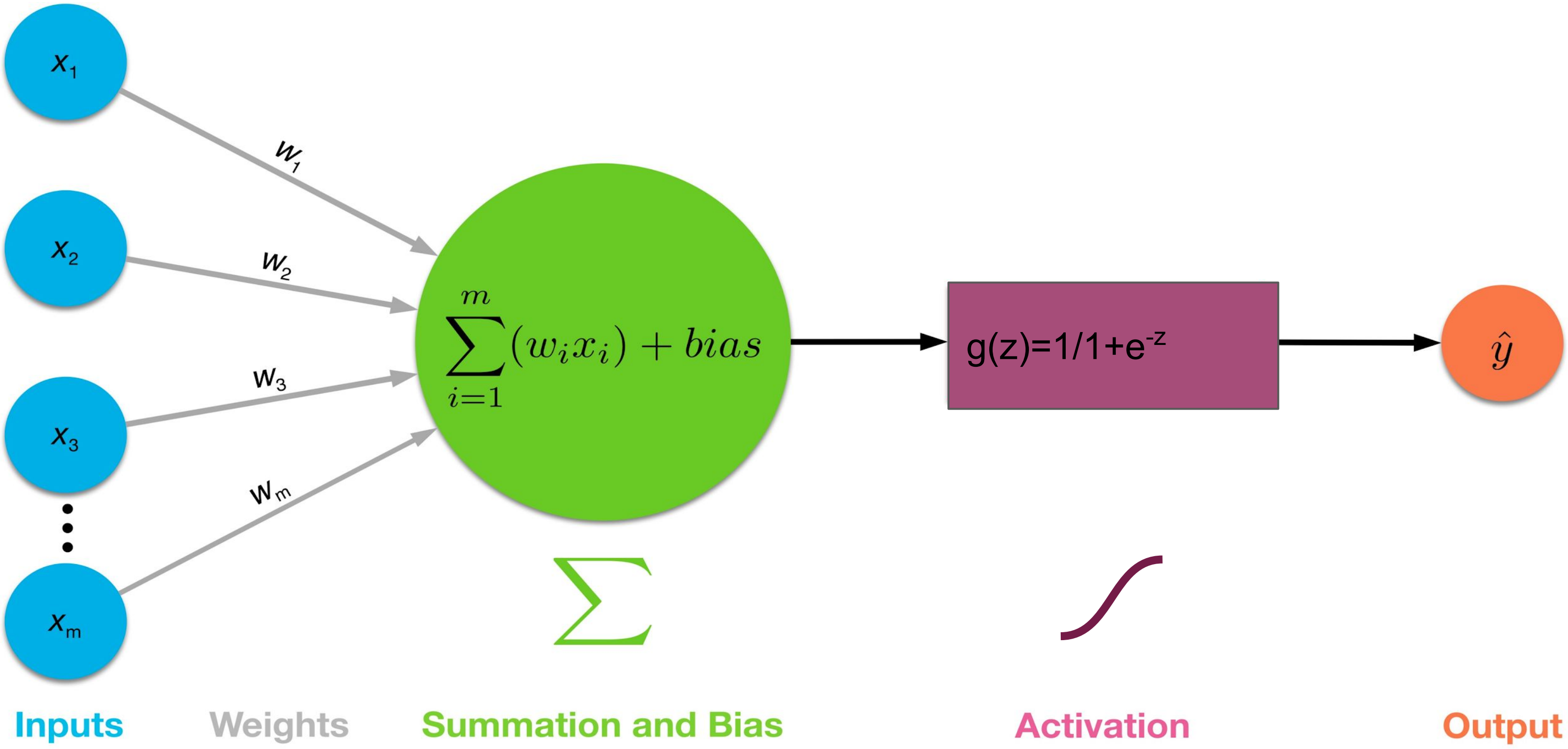


tf.nn.elu(x)

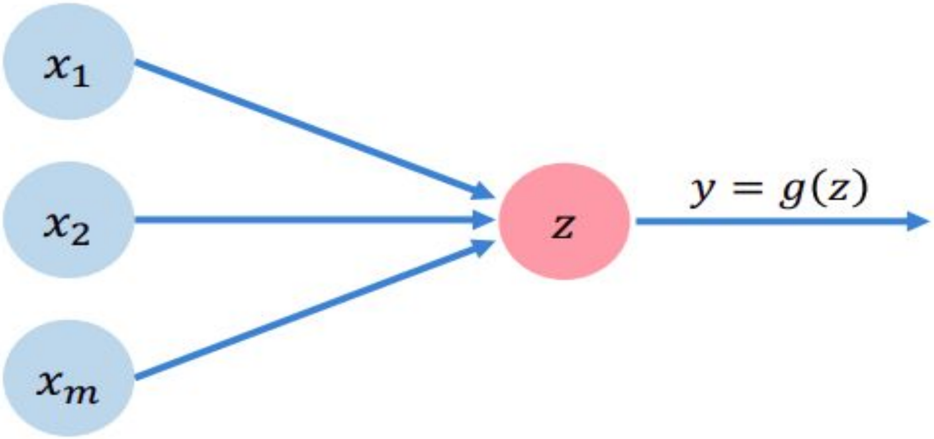

Wx + b < 0
Next
Layer
ReLU

Building Neural Networks with Perceptrons

The Perceptron: Simplified

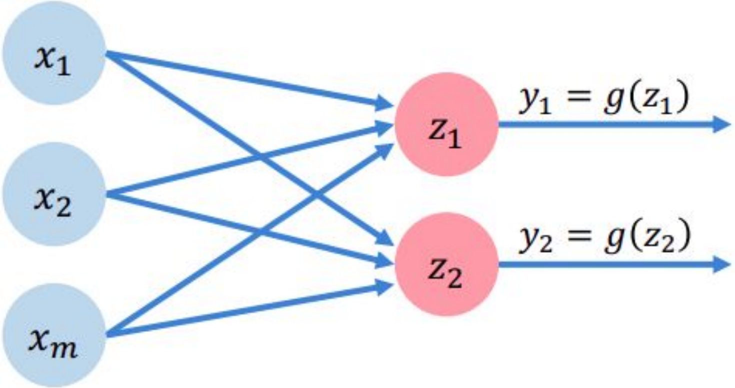


The Perceptron: Simplified



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

Multi Output Perceptron



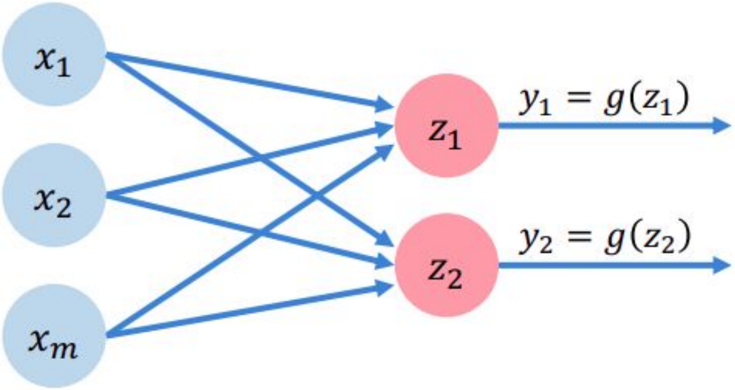
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

```python
class myDenseLayer(tf.keras.layers.Layer):
    def __init__(self,input_dim,output_dim):
        super(myDenseLayer,self).__init__()
        init_w=tf.random_normal_initializer()
        self.w=tf.Variable(initial_value=init_w(shape=(input_dim,output_dim),dtype='float32'),
                           trainable=True)
        init_b=tf.zeros_initializer()
        self.b=tf.Variable(initial_value=init_b(shape=(output_dim,),dtype='float32'),
                           trainable=True)
    def call(self,inputs):
        z=tf.matmul(inputs,self.w)+self.b
        output=tf.math.sigmoid(z)
        return output
```

```python
x=tf.ones((4,4))
model=myDenseLayer(4,2)
y=model(x)
print(y)
```

```
tf.Tensor(
[[0.52305216 0.5289087 ]
 [0.52305216 0.5289087 ]
 [0.52305216 0.5289087 ]
 [0.52305216 0.5289087 ]], shape=(4, 2), dtype=float32)
```

Multi Output Perceptron

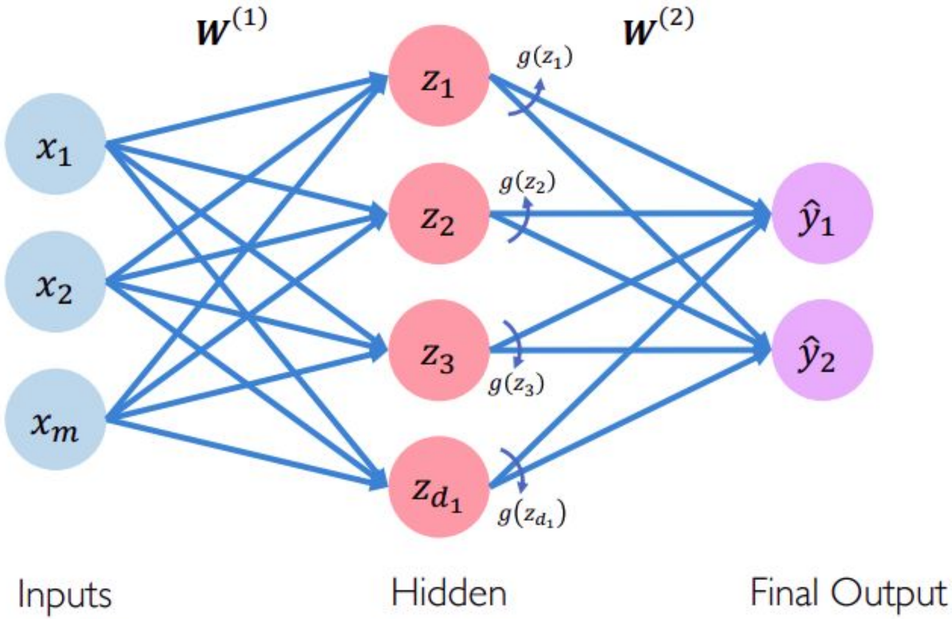


$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

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


PERCEPTRONS ARE TOO LINEAR
WE MUST GO DEEPER

Single Layer Neural Network

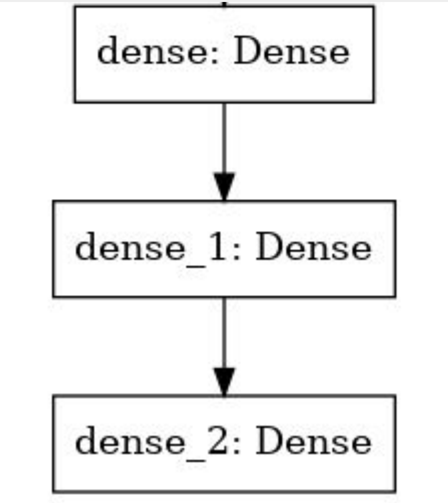


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)} \right)$$

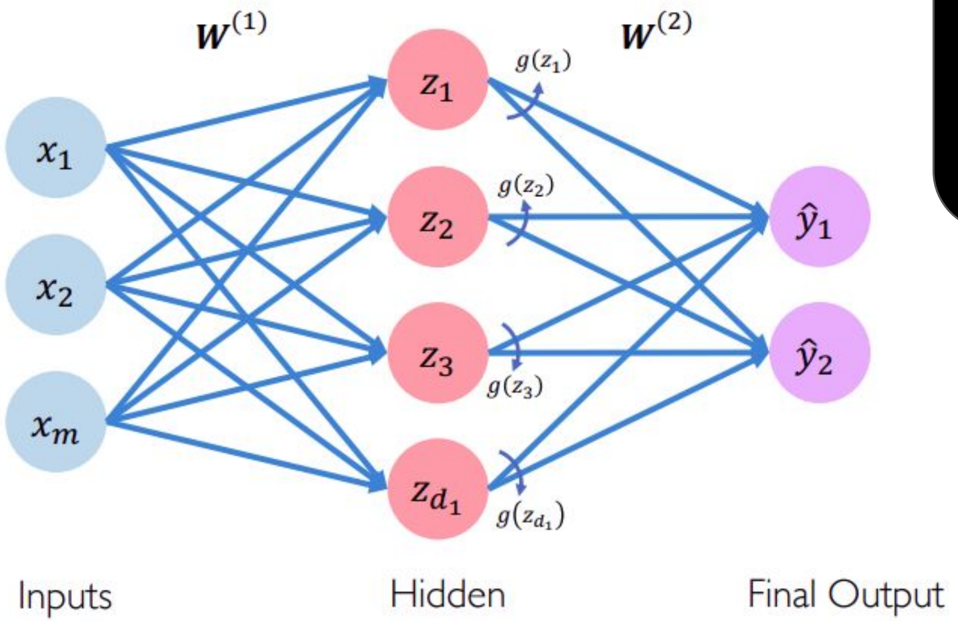
The Sequential model

When to use a Sequential model
A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

```python
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
```



Multi Output Perceptron

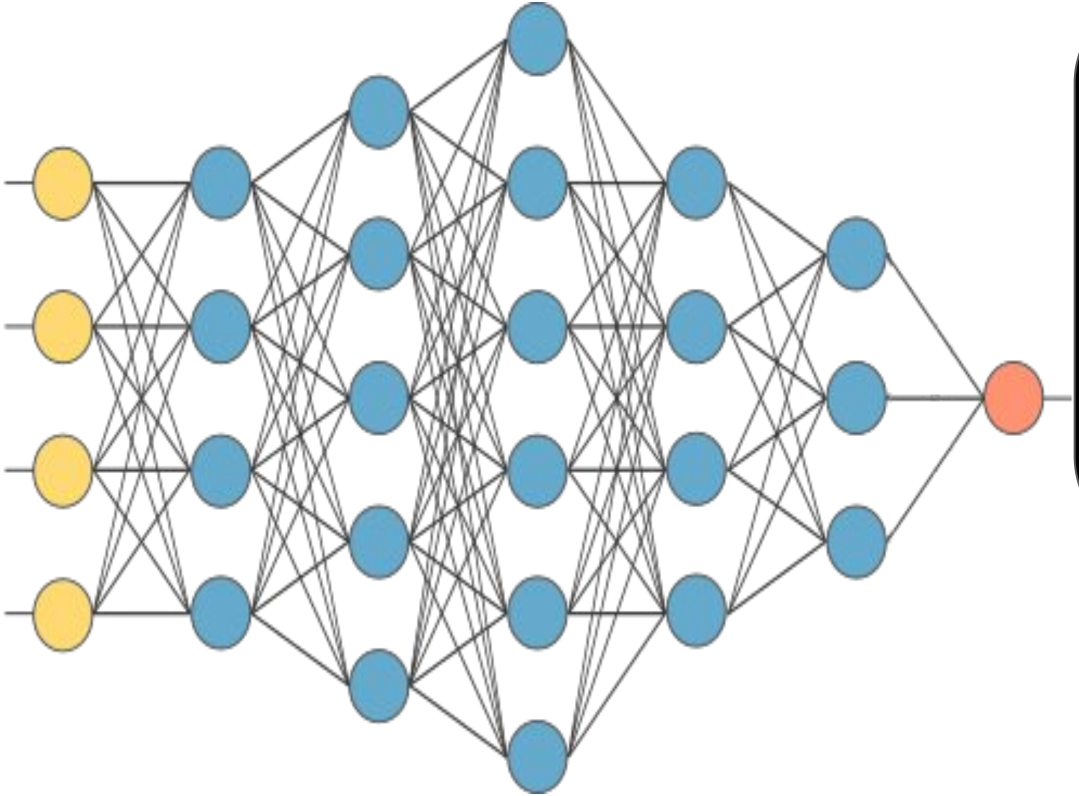


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)} \right)$$

```
from tensorflow..keras.layers
import *
model=tf.keras.Sequential([
nputs(m),
Dense(4,activation="relu"),
Dense(2,activation="softmax")])
```


WE NEED TO GO
DEEPER

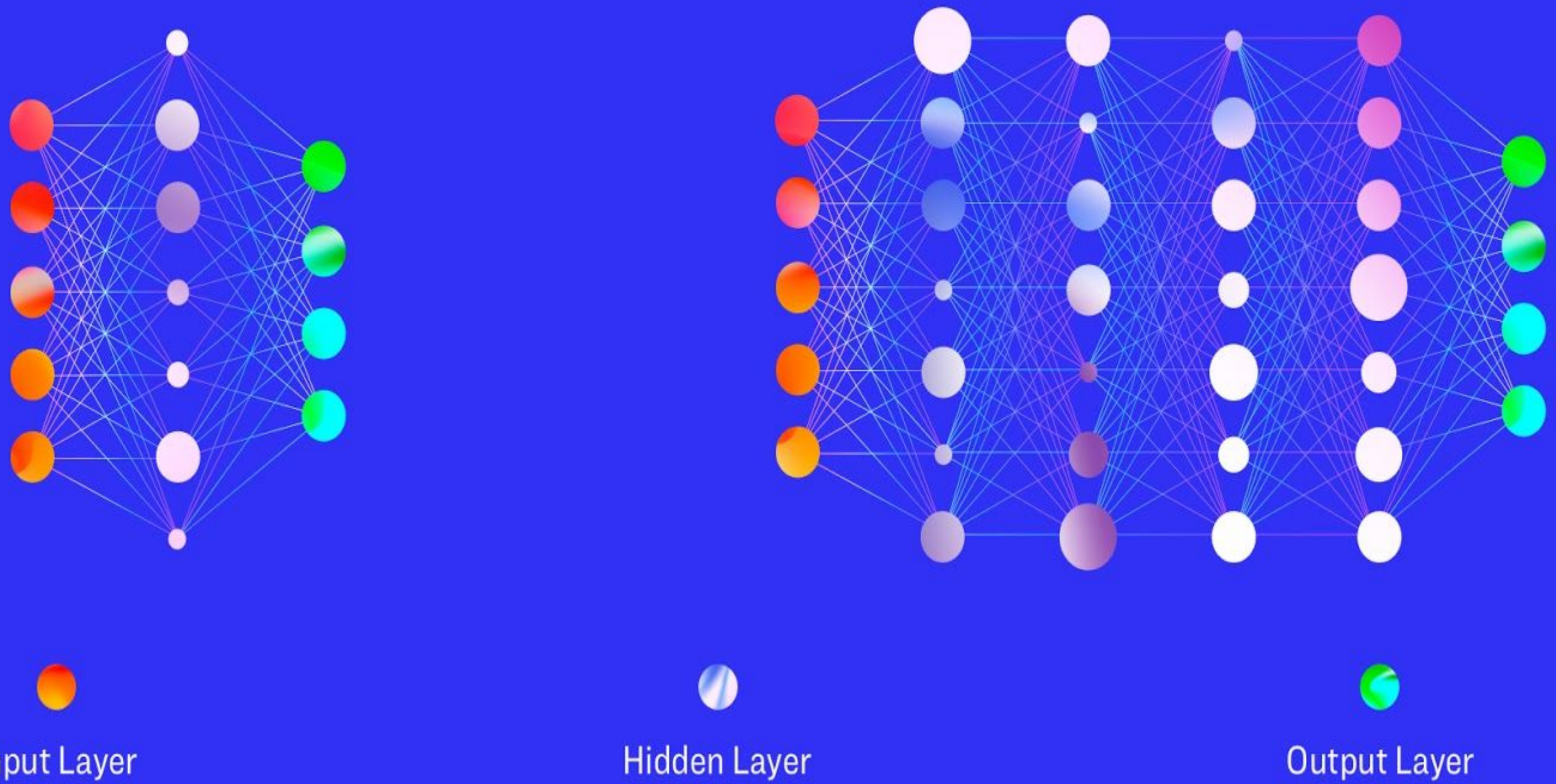
Deep Neural Network



```
from tensorflow..keras.layers import *
model=tf.keras.Sequential([
        Inputs(m),
        Dense(4,activation="relu"),
        Dense(5,activation="relu"),
        Dense(6,activation="relu"),
         Dense(4,activation="relu"),
        Dense(3,activation="relu"),
        Dense(1,activation="sigmoid")
])
```

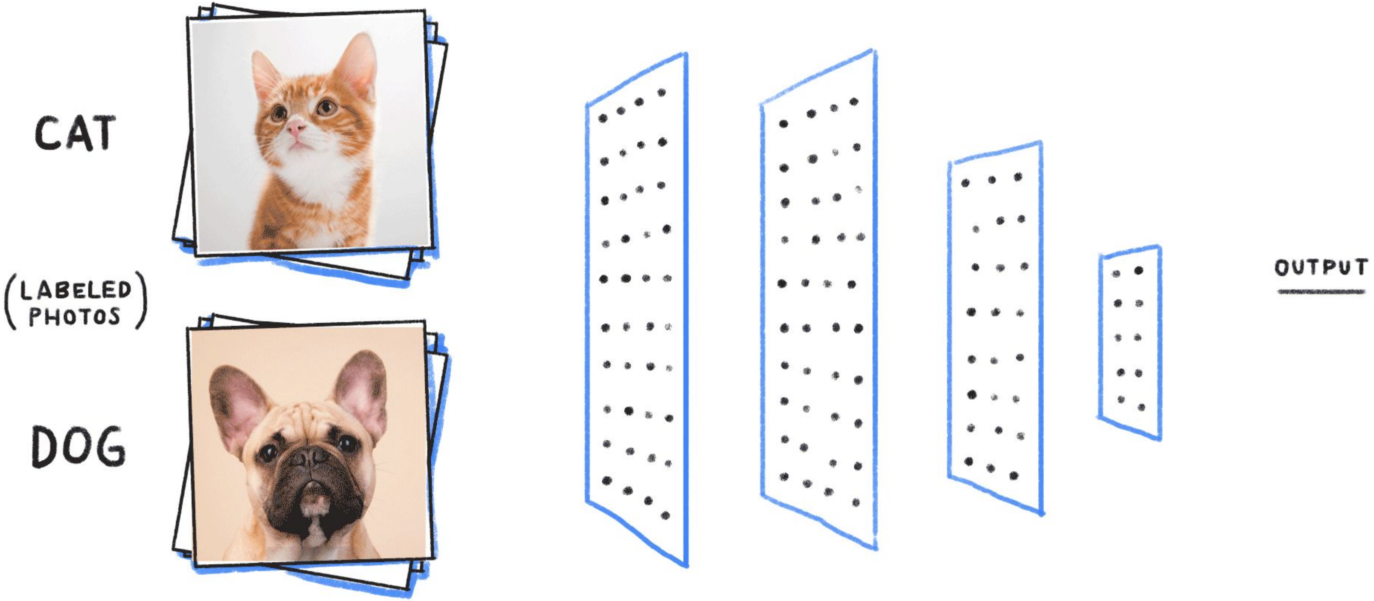
Simple Neural Network

Deep Learning Neural Network

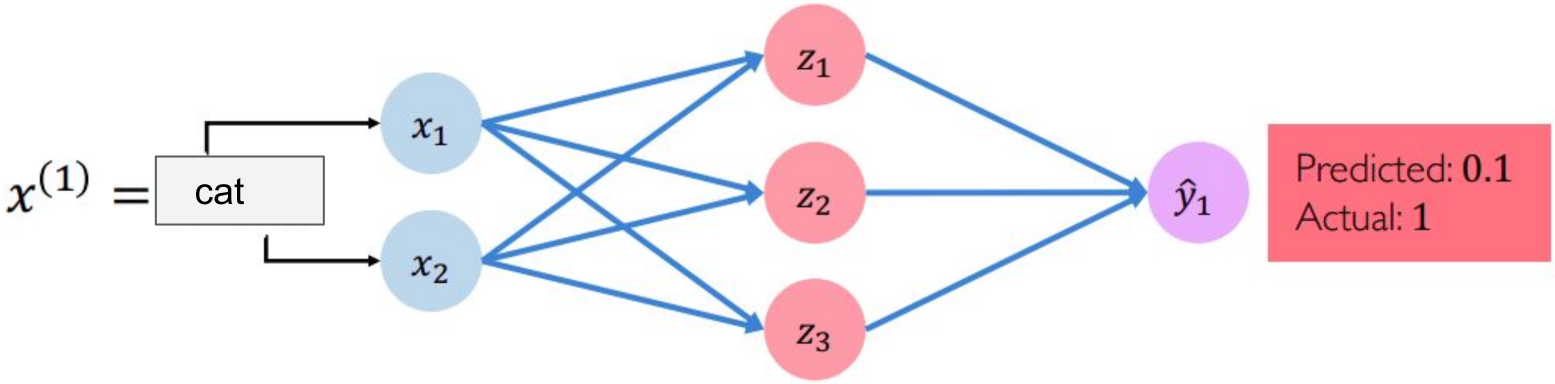


Applying Neural Networks

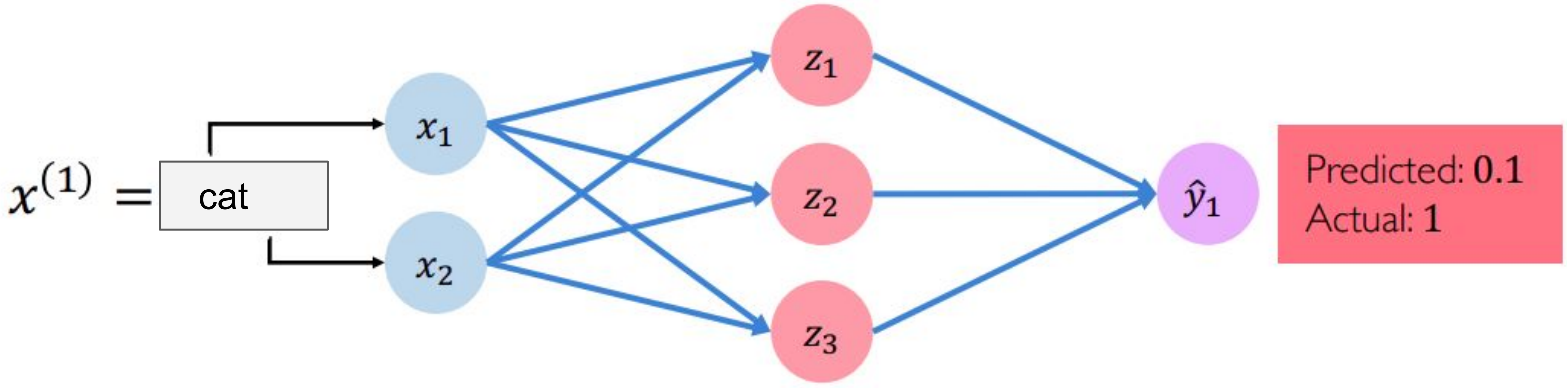
Example Problem : Cat Vs Dog



Example Problem: Will I pass this class?



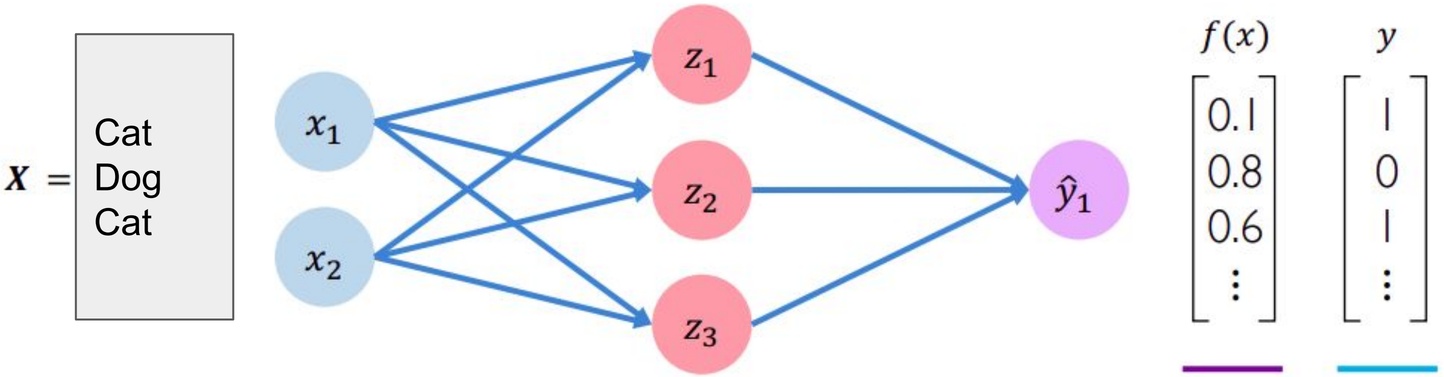
Quantifying Loss



$$\mathcal{L}\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

Cross entropy loss

***Cross entropy loss** can be used with models that output a probability between 0 and 1*

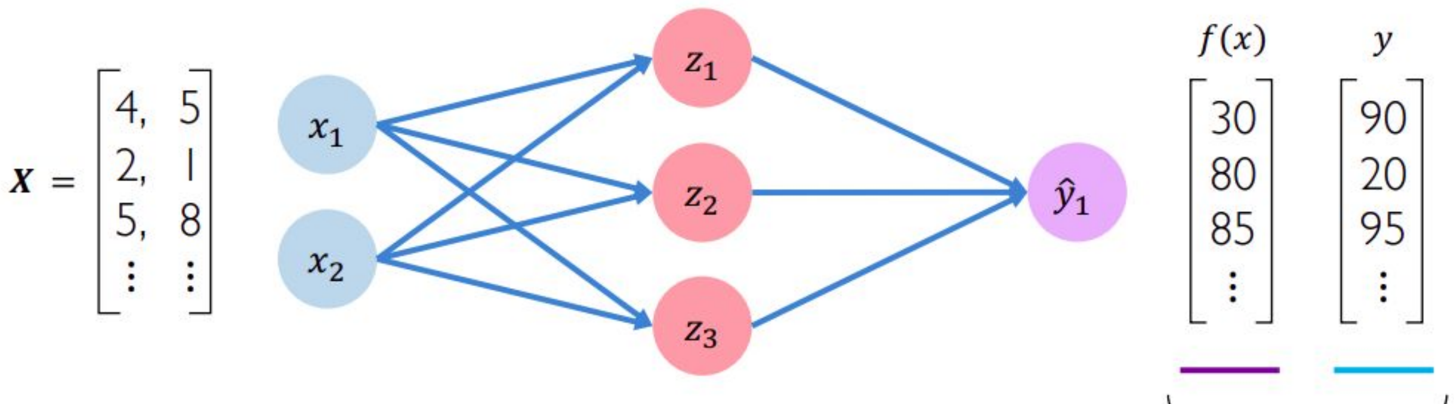


$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

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

Mean Square Error

***Mean squared error loss** can be used with regression models that output continuous real numbers*



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

Final Grades
(percentage)

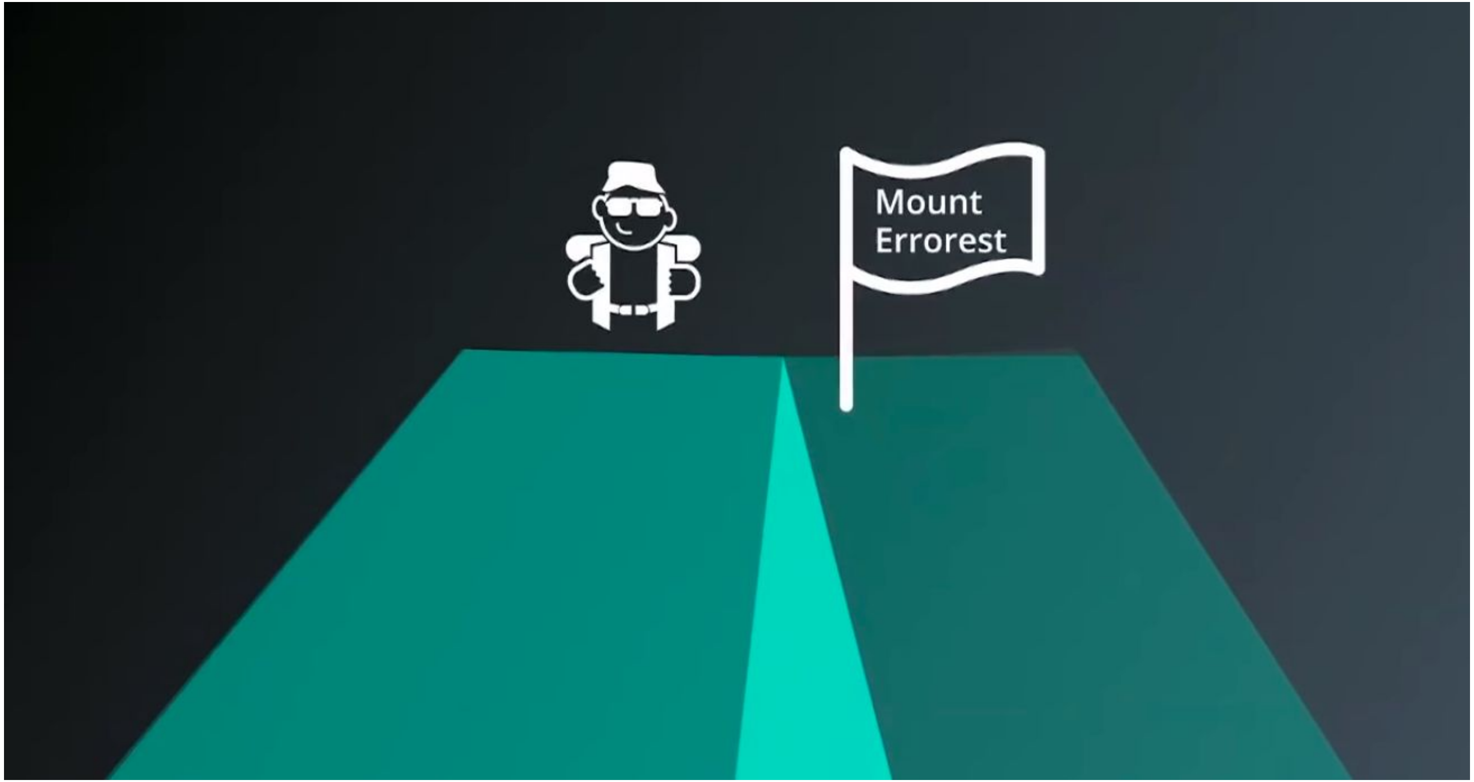
```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
```

People with no idea about AI, telling me my AI will destroy the world

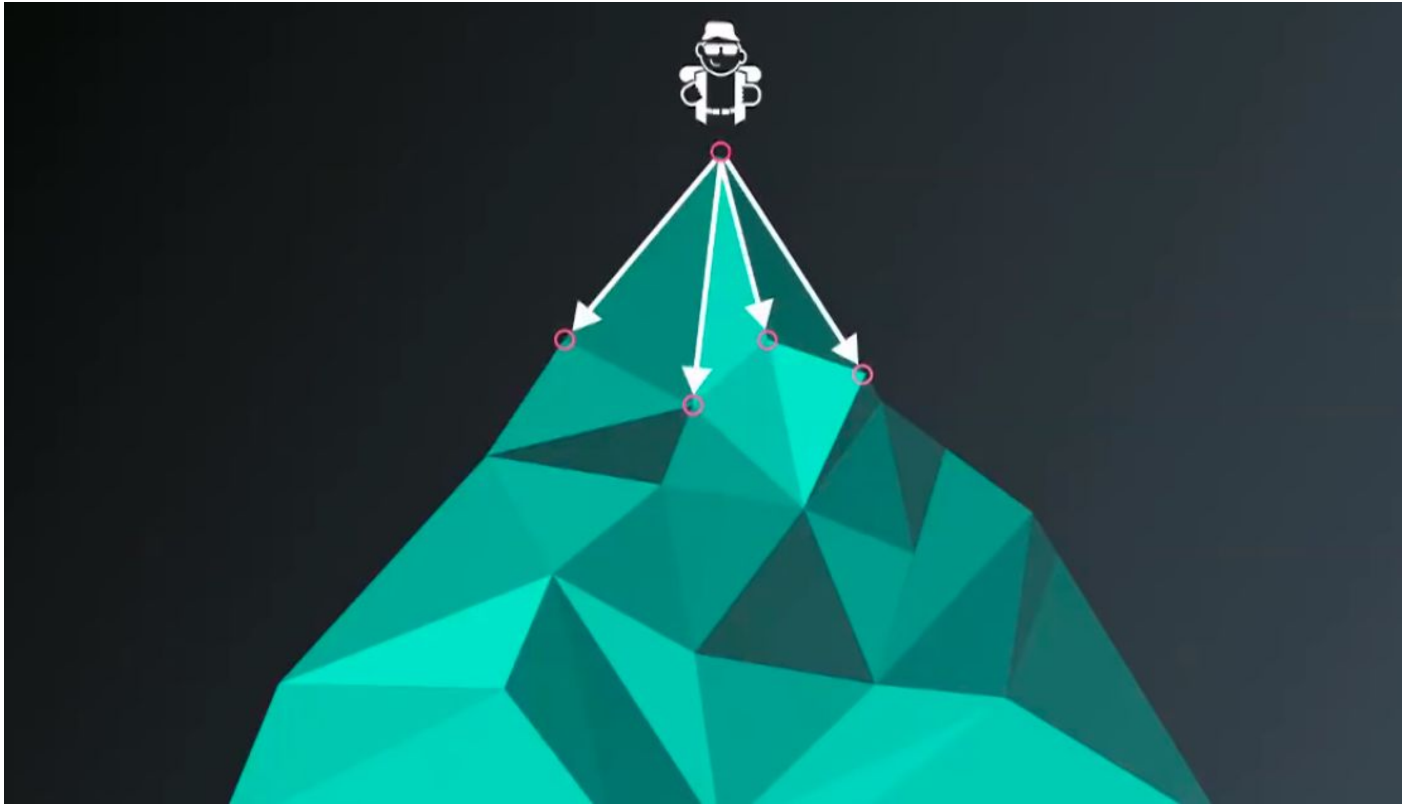
Me wondering why my neural network is classifying a cat as a dog..

Training Neural Networks

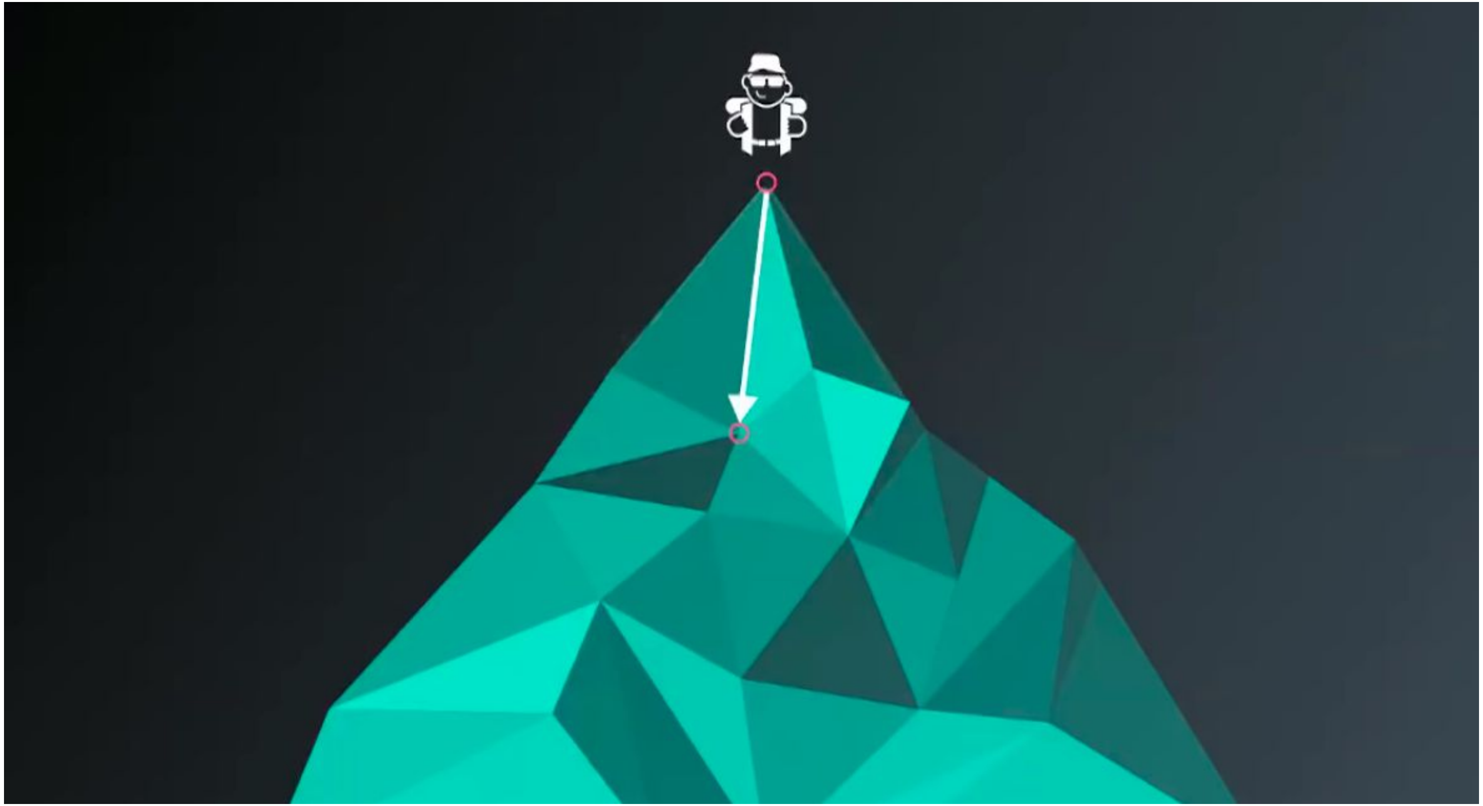
Loss Optimization



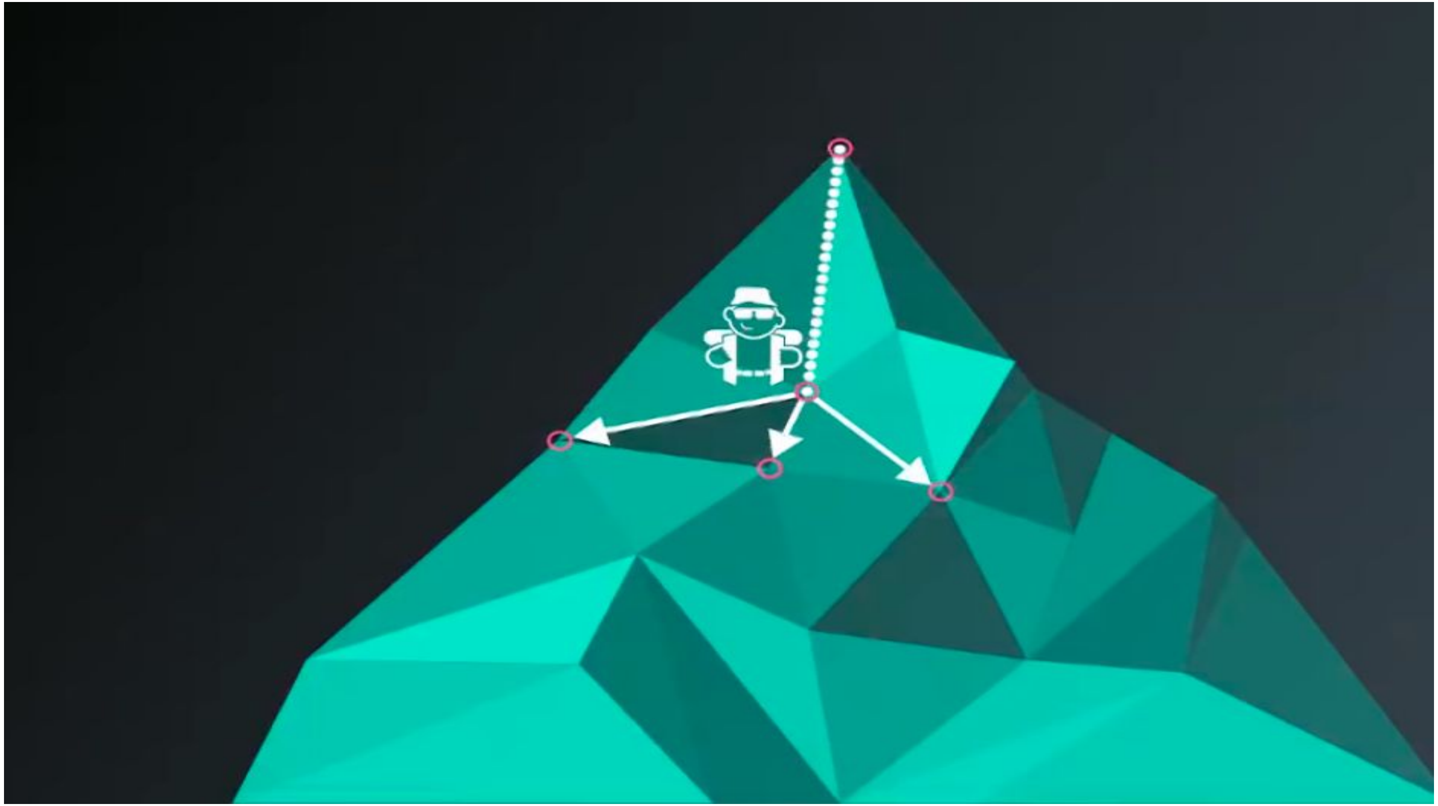
Loss Optimization



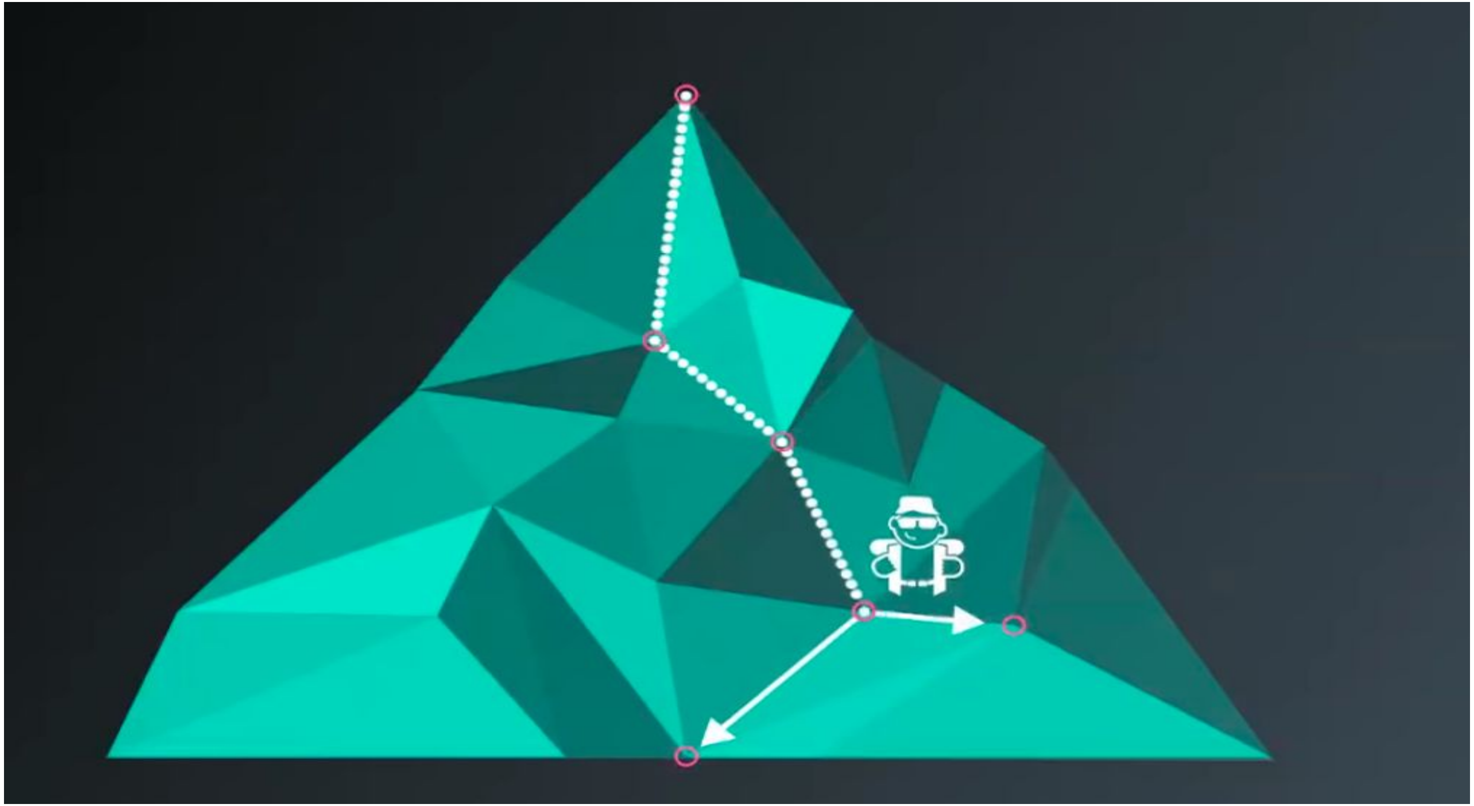
Loss Optimization



Loss Optimization

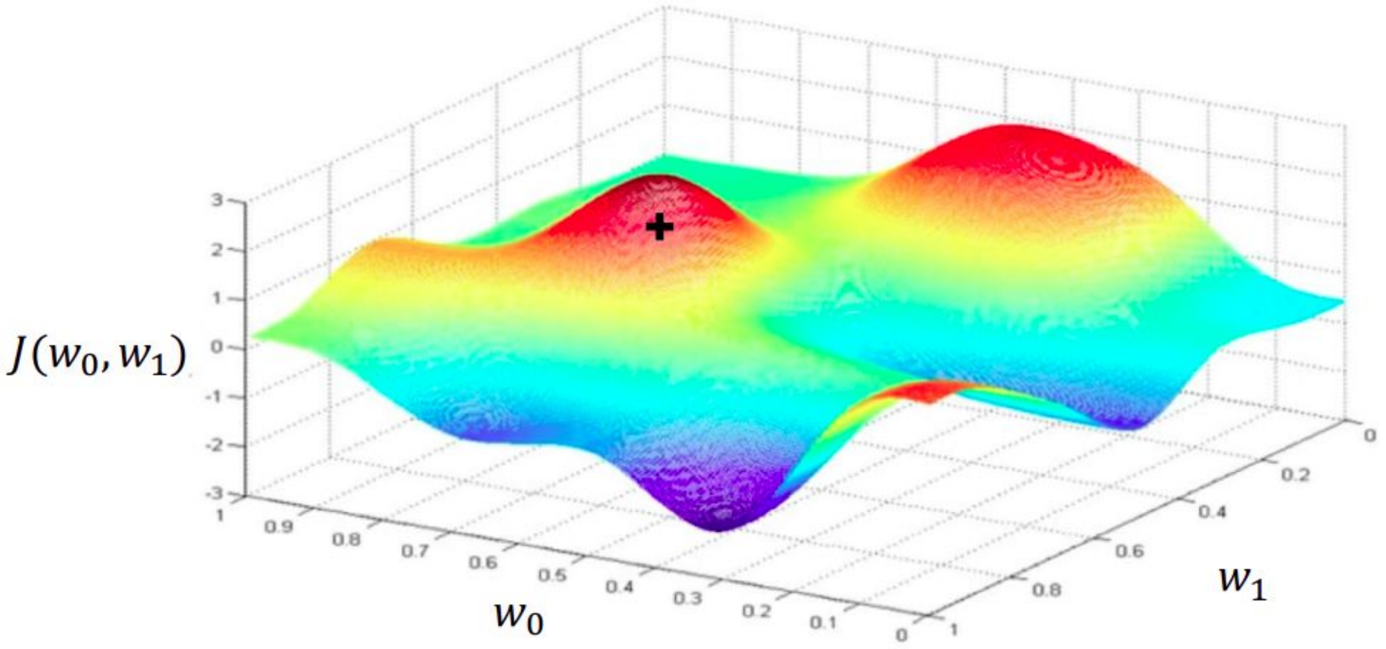


Loss Optimization



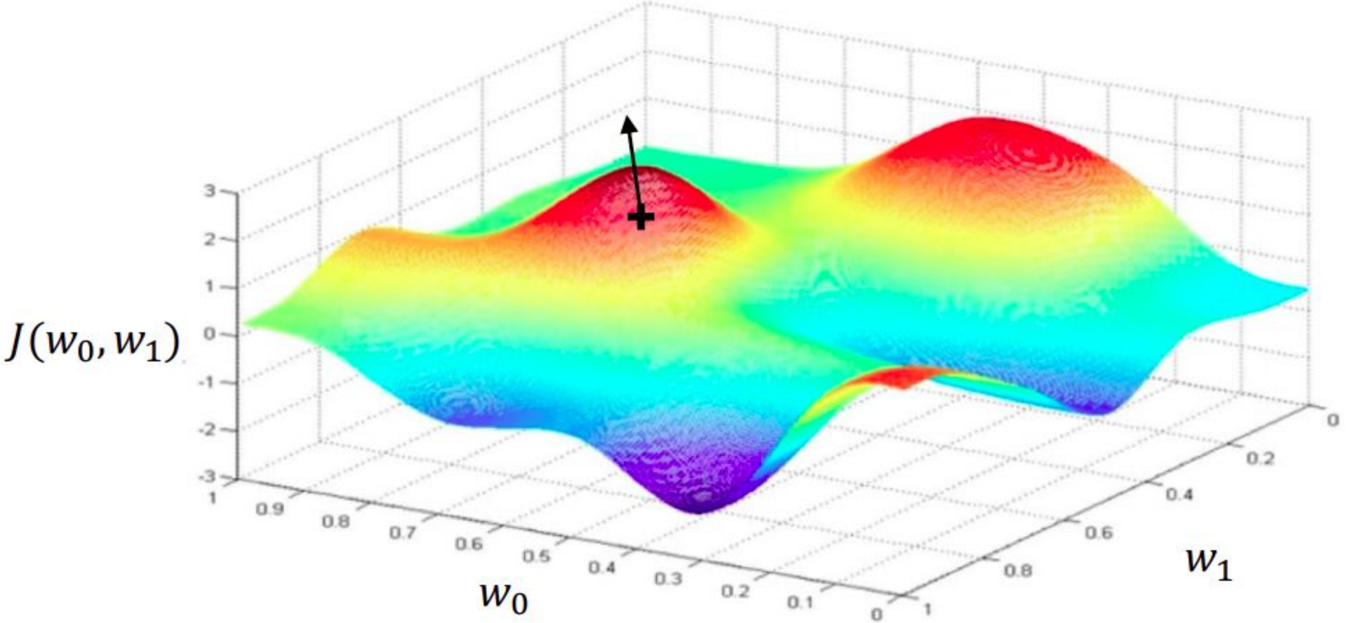
Loss Optimization

Randomly pick an initial (w_0, w_1)

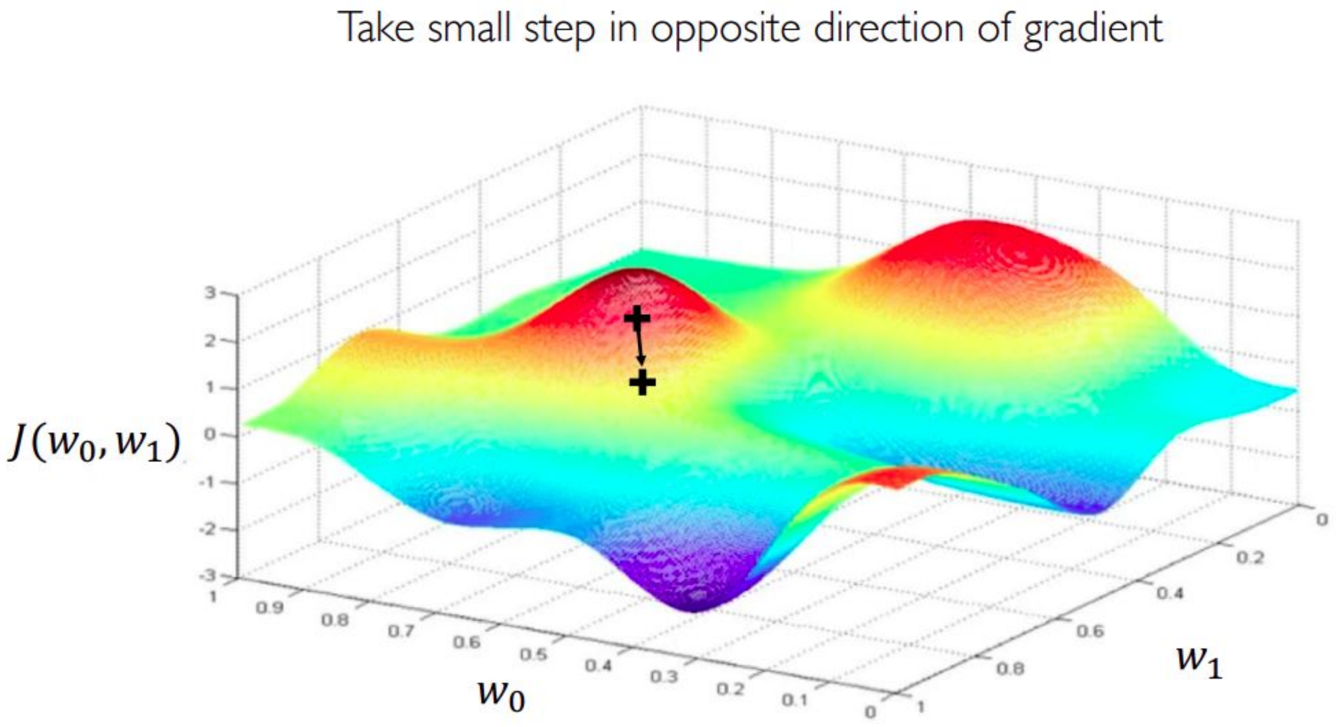


Loss Optimization

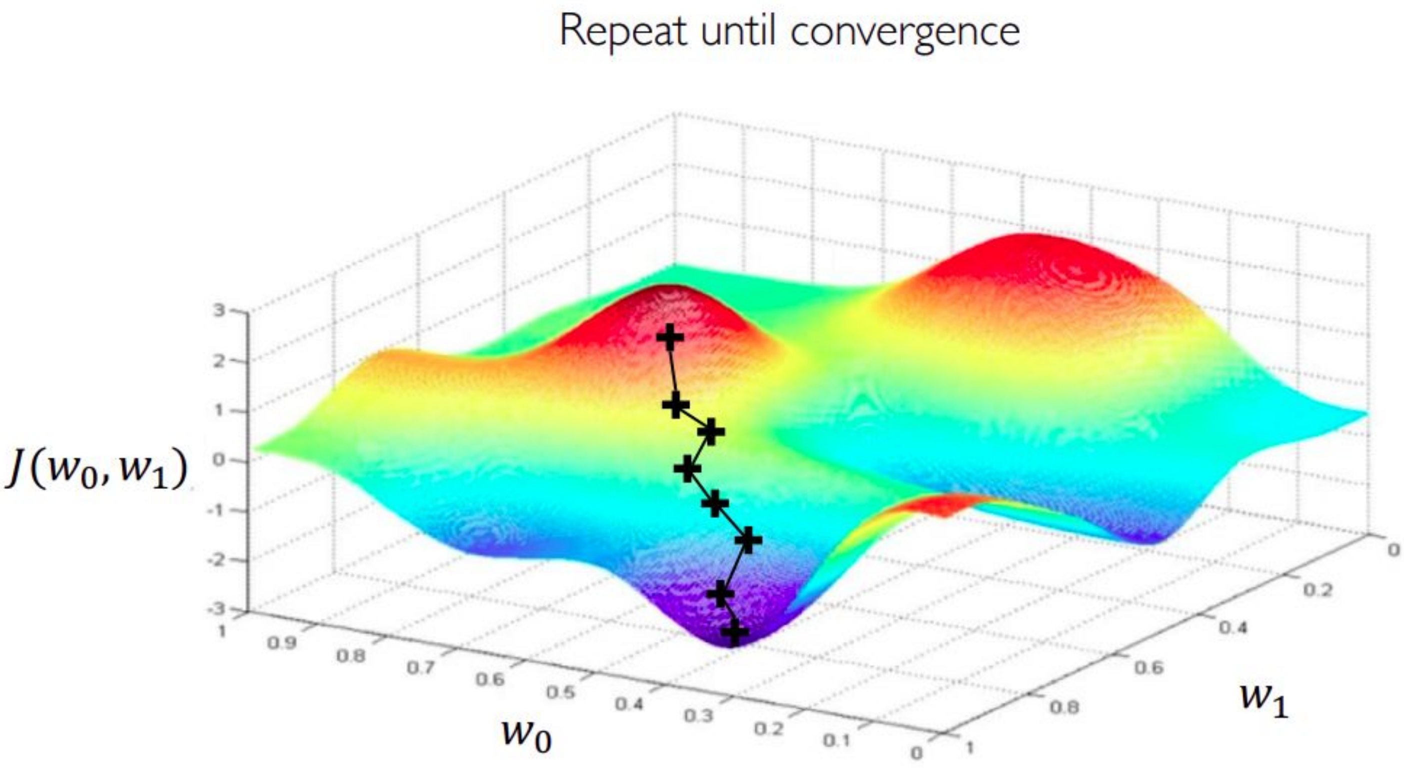
Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$



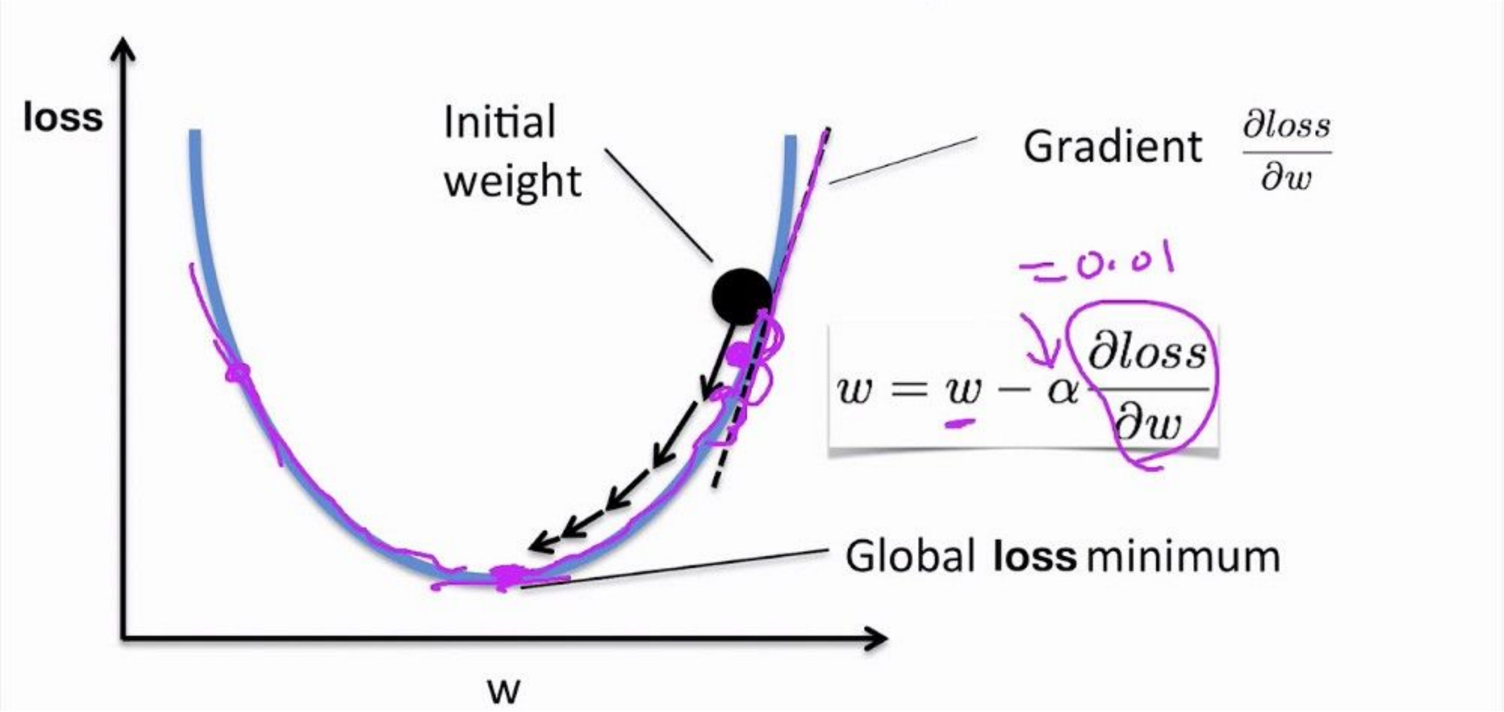
Loss Optimization



Gradient Descent



Gradient Descent



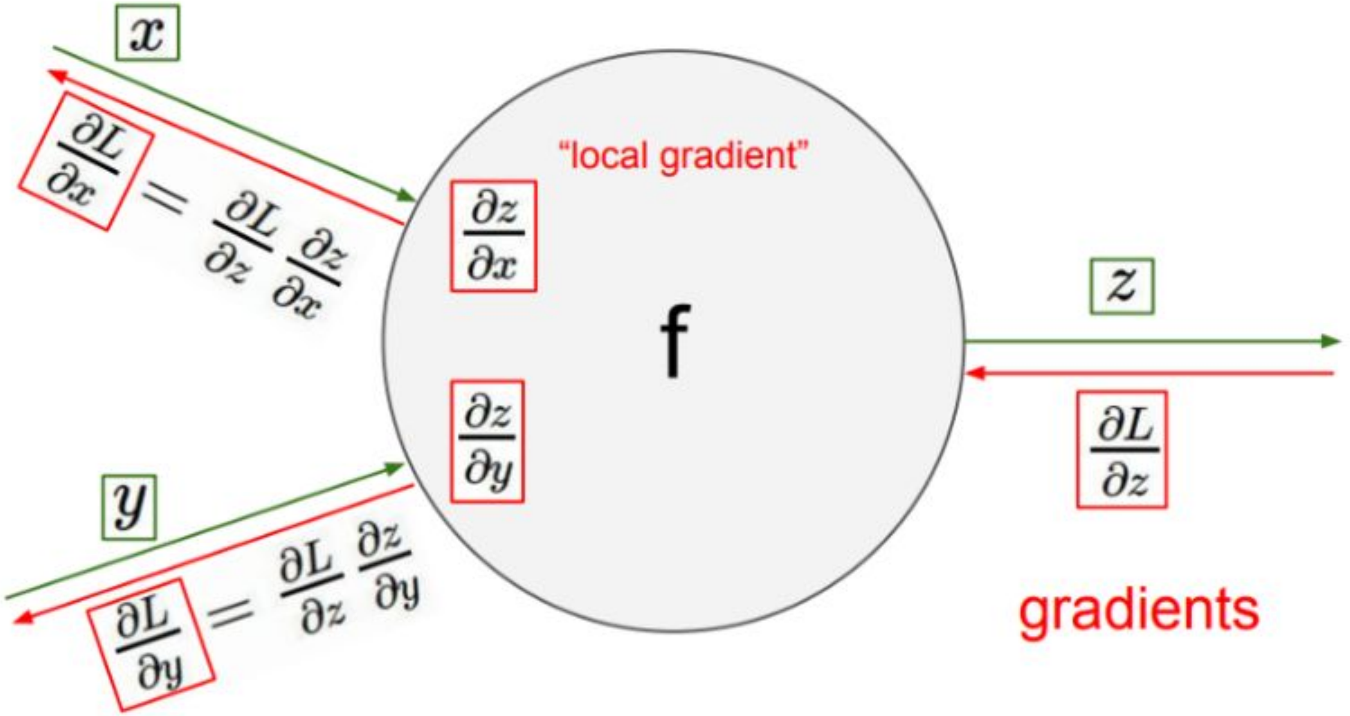
Gradient Descent

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

```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
```

Computing Gradients: Backpropagation



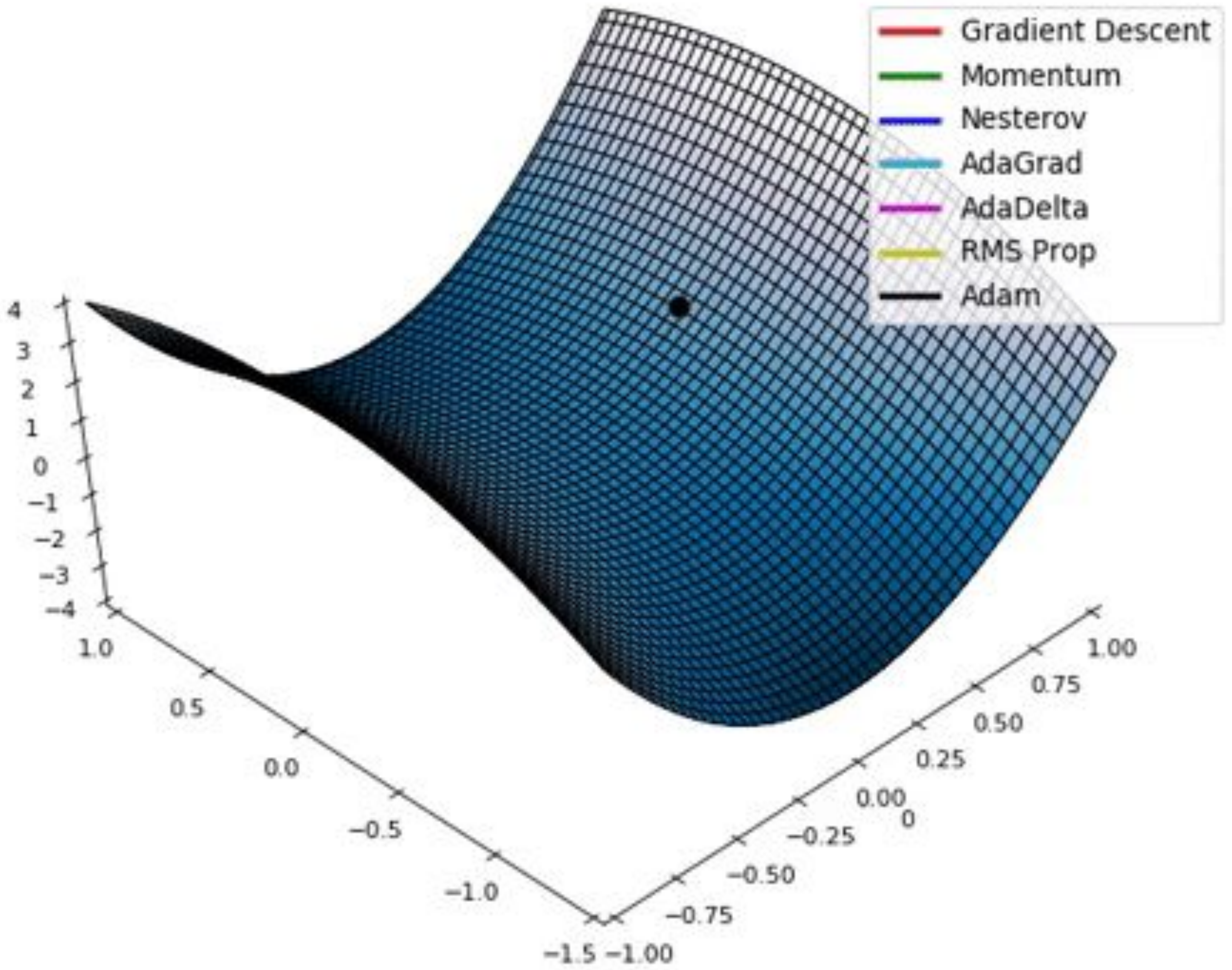
Optimization

Gradient Descent Algorithms

Algorithm	***TF Implementation***
SGD	`tf.optimizers.SGD( )`
Adam	`tf.optimizers.Adam( )`
Adadelta	`tf.optimizers.Adadelta( )`
Adagrad	`tf.optimizers.Adagrad( )`
RMSProp	`tf.optimizers.RMSprop( )`

Reference

Qian et al. “On the momentum term in gradient descent learning algorithms.” 1999. Duchi et al. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.” 2011. Zeiler et al. “ADADELTA: An Adaptive Learning Rate Method.” 2012. Kingma et al. “Adam: A Method for Stochastic Optimization.” 2014.


Gradient Descent
Momentum
Nesterov
AdaGrad
AdaDelta
RMS Prop
Adam
4
3
2
1
0
−1
−2
−3
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.00
−0.75
−0.50
−0.25
0.00
0
0.25
0.50
0.75
1.00

Neural Networks in Practice: Mini-batches

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

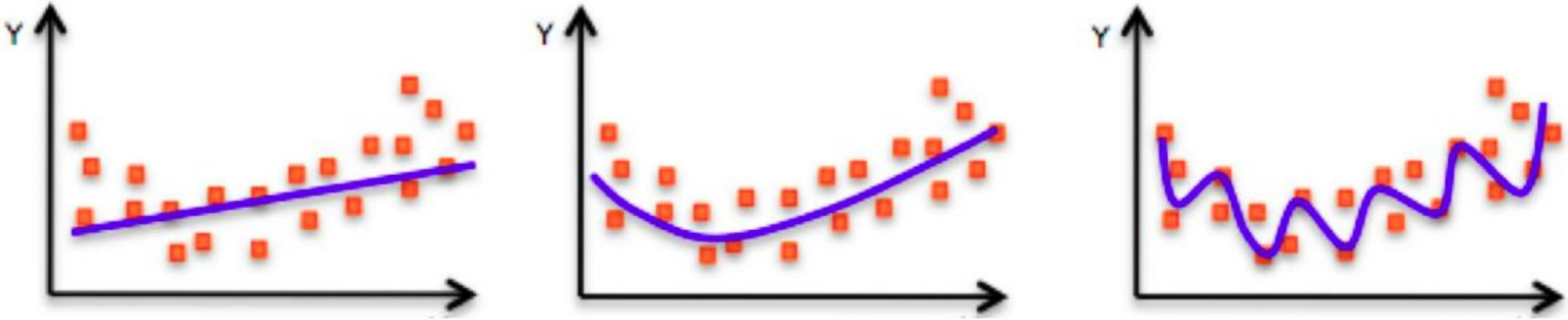
Allows for larger learning rates

Mini-batches lead to fast training!

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

Neural Networks in Practice: Overfitting

The Problem of Overfitting

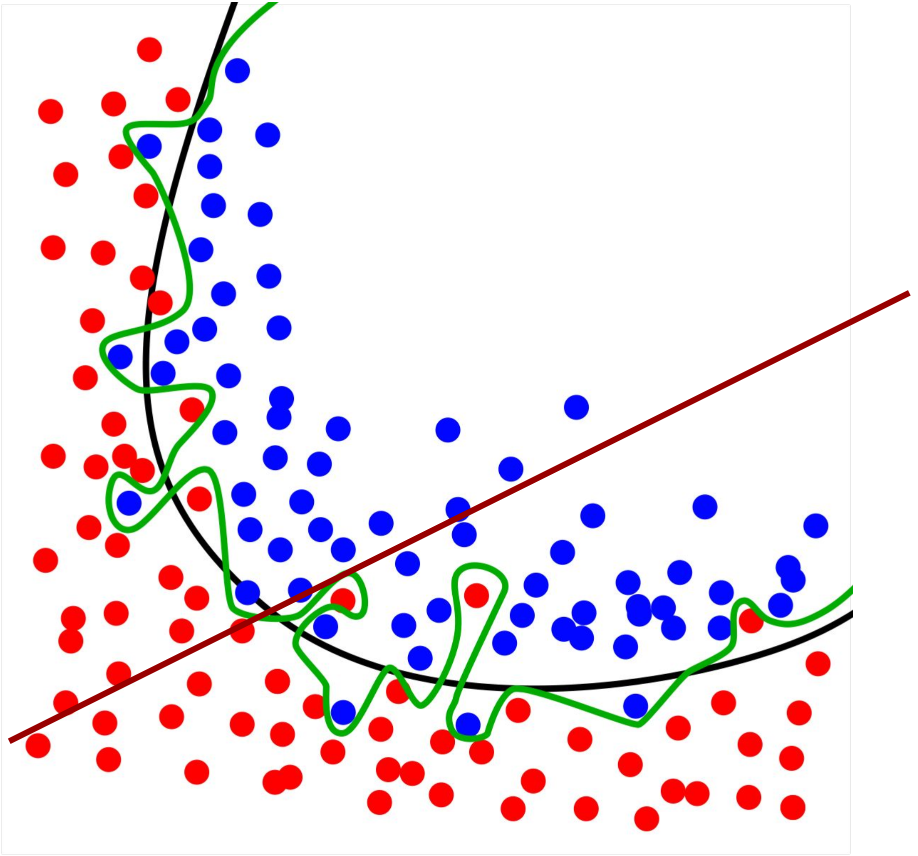


Underfitting
Model does not have capacity
to fully learn the data

← **Ideal fit** →

Overfitting
Too complex, extra parameters,
does not generalize well

The Problem of Overfitting



Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

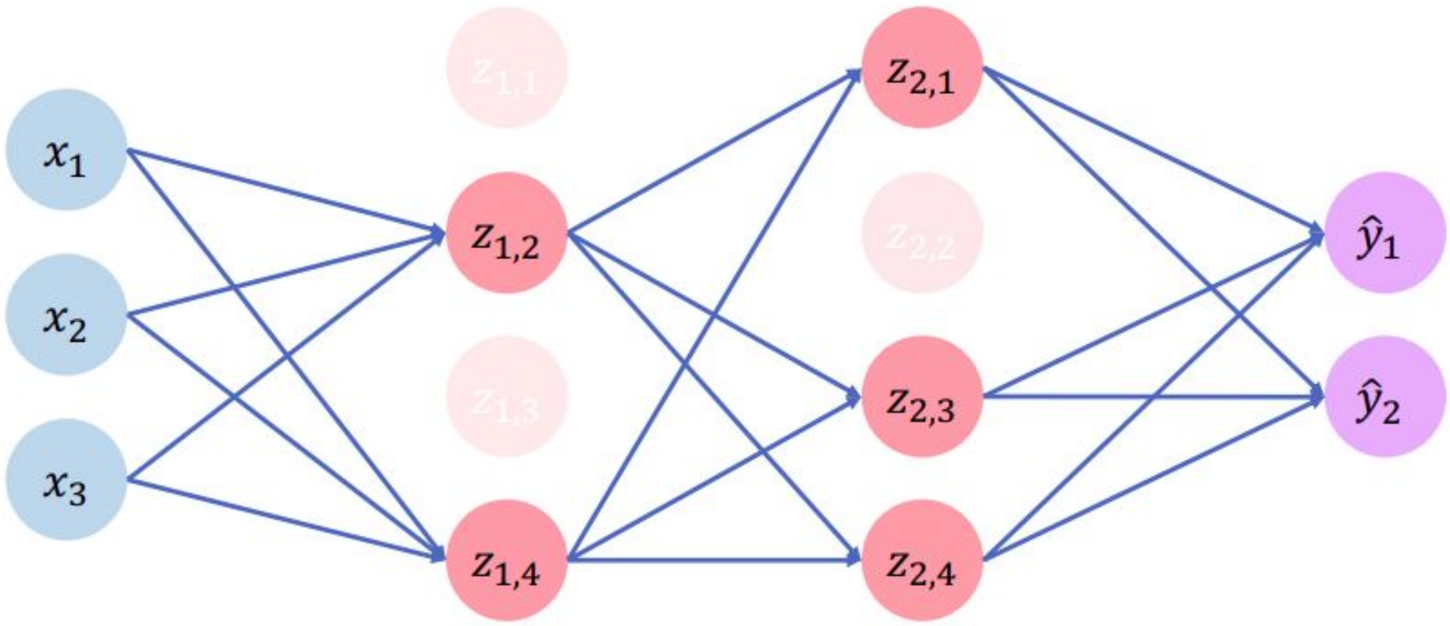
Why do we need it?

Improve generalization of our model on unseen data

Regularization 1: Dropout

- During training, randomly set some activations to 0

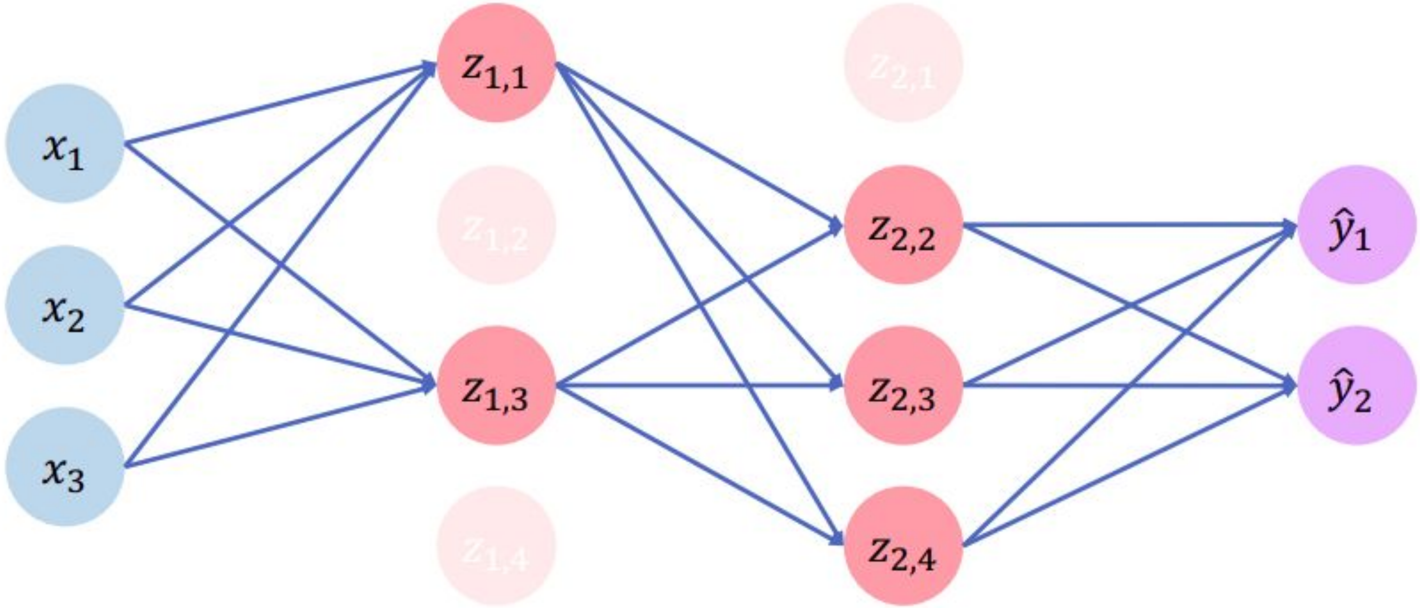
`tf.keras.layers.Dropout( )`



Regularization 1: Dropout

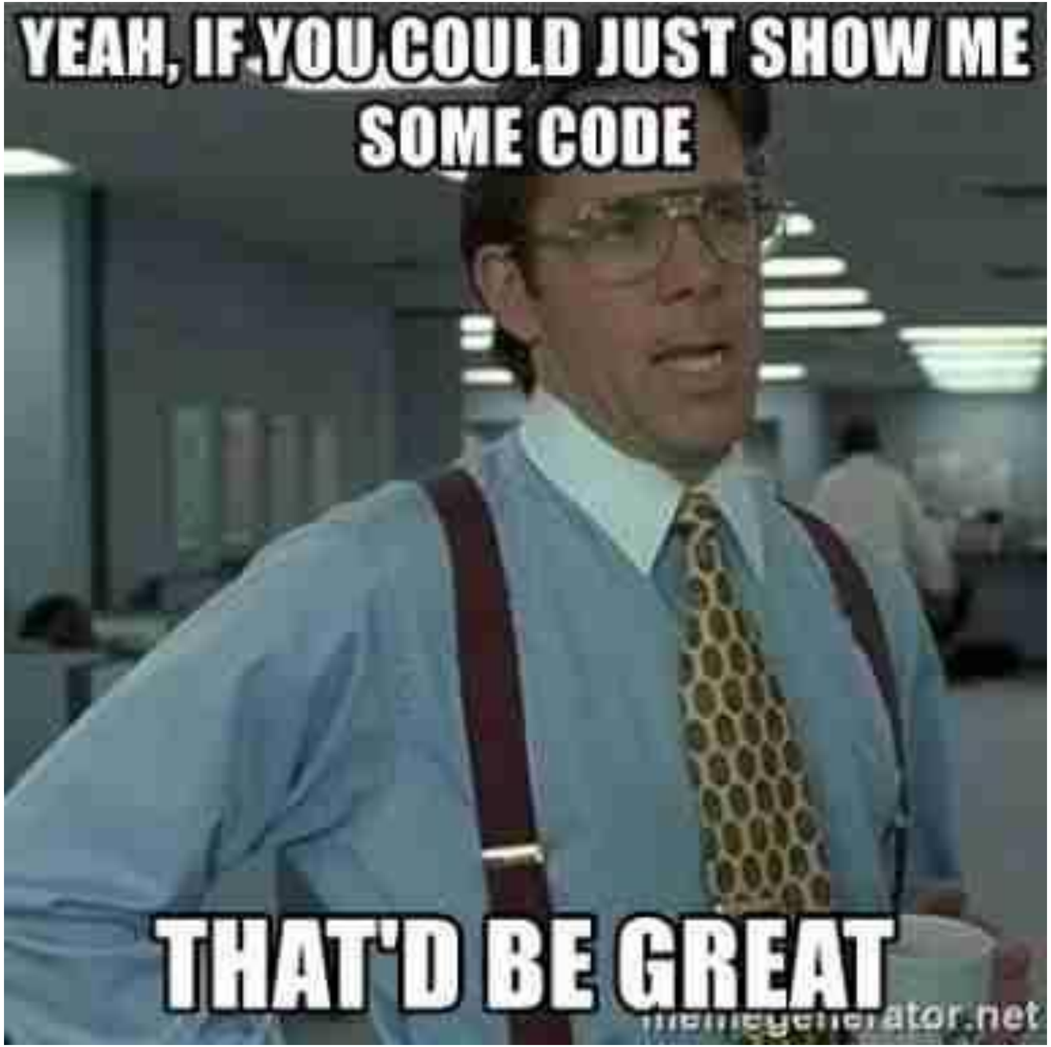
- During training, randomly set some activations to 0
- Typically 'drop' 50% of activations in layer
- Forces network to not rely on any 1 node

`tf.keras.layers.Dropout( )`



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit


YEAH, IF YOU COULD JUST SHOW ME SOME CODE
THAT'D BE GREAT


ITS OVER
ITS FINALLY OVER
quickmeme.com