CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

TensorFlow

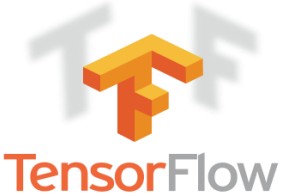
Machine Learning

Machine Learning

Lecture: TensorFlow

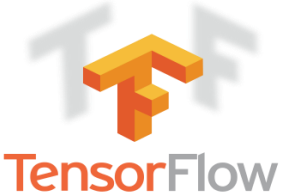
Ted Scully

TensorFlow



- We will be using Google Colab in this module.
- Colab is a cloud-based Juypter notebook style environment that comes with a free GPU or TPU (Brief introduction video here).
- A specific instance of Colab should allow you to run your code for 24 hrs without interruptions but each instance is limited to **24 hours**.
- Colab has two versions of TensorFlow installed: a 1.x version (currently 1.15) and a 2.x version (currently 2.1).
- Colab currently uses TF 1.x by default. We will be using the newer TF2.1 in this module.

TensorFlow



- To enable your notebook to run with TF2.1 include the following code in the first cell in your notebook and execute.

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)
```

- It is important that you execute this block of code before executing any other code (otherwise the TF1.X version will be used).

```
!nvidia-smi
```

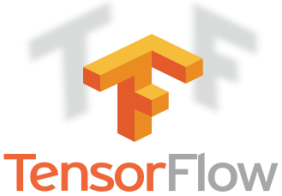
TensorFlow

- To confirm that you are using the GPU then include the following code.

```
!nvidia-smi
```

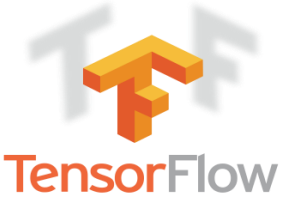
```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.59       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   36C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

TensorFlow



- You can find the notebook that contains the code from this class here.
- I have included a short presentation in the Canvas unit for this week showing how to get started with Colab.

TensorFlow

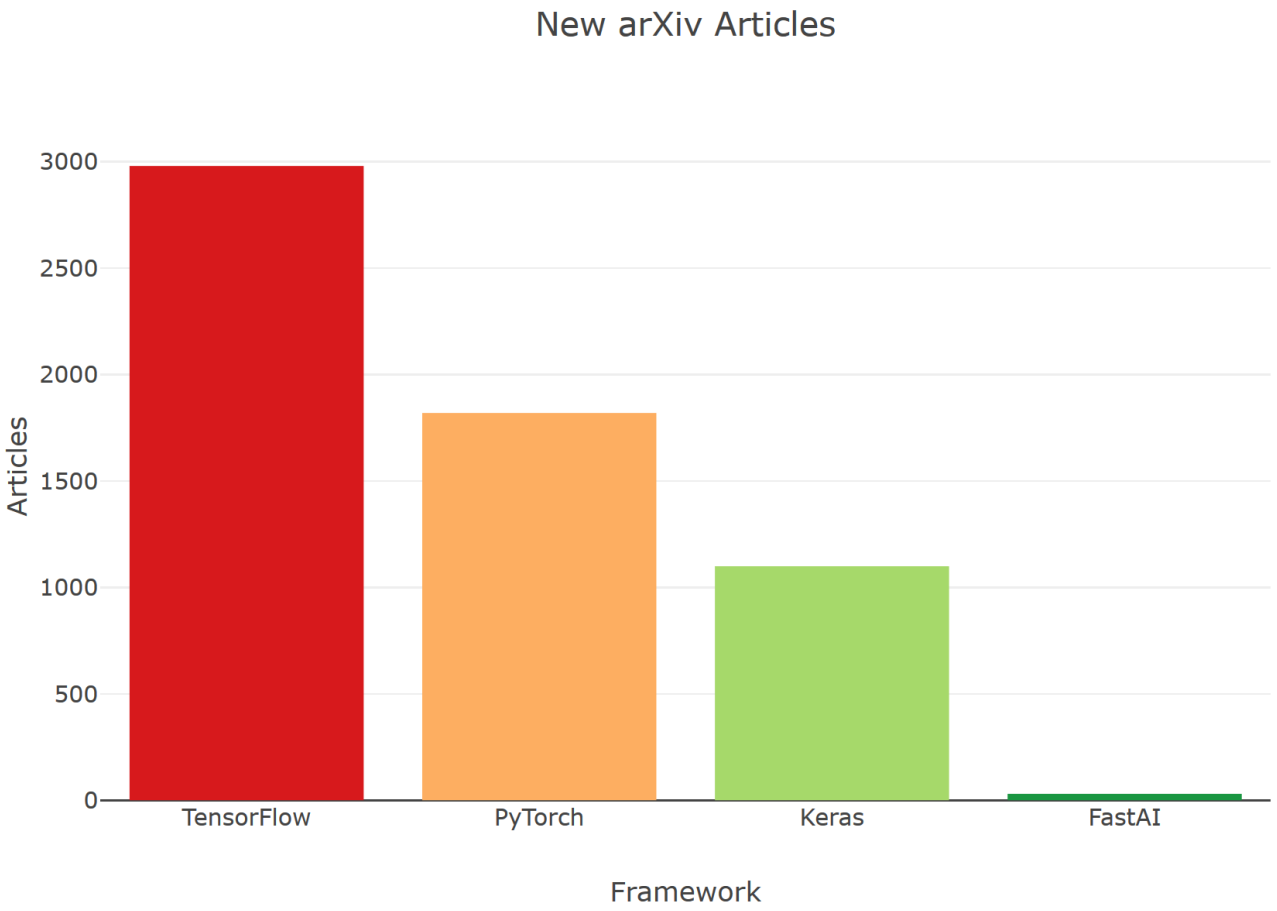


- TensorFlow was developed by the **Google Brain** team for internal Google use but it was released to the public in 2015.
  - It is used by many of Google's large-scale services, such as Google **Cloud Speech**, Google **Photos**, and Google **Search**, Google **Translate** etc.
  - It is still arguably the most **popular** Deep Learning framework (in terms of citations in papers, adoption in companies, stars on GitHub, etc.).
  - Used by companies such as Twitter, Intel, Uber, Paypal, etc. … (see #poweredbytf)

- TensorFlow 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.

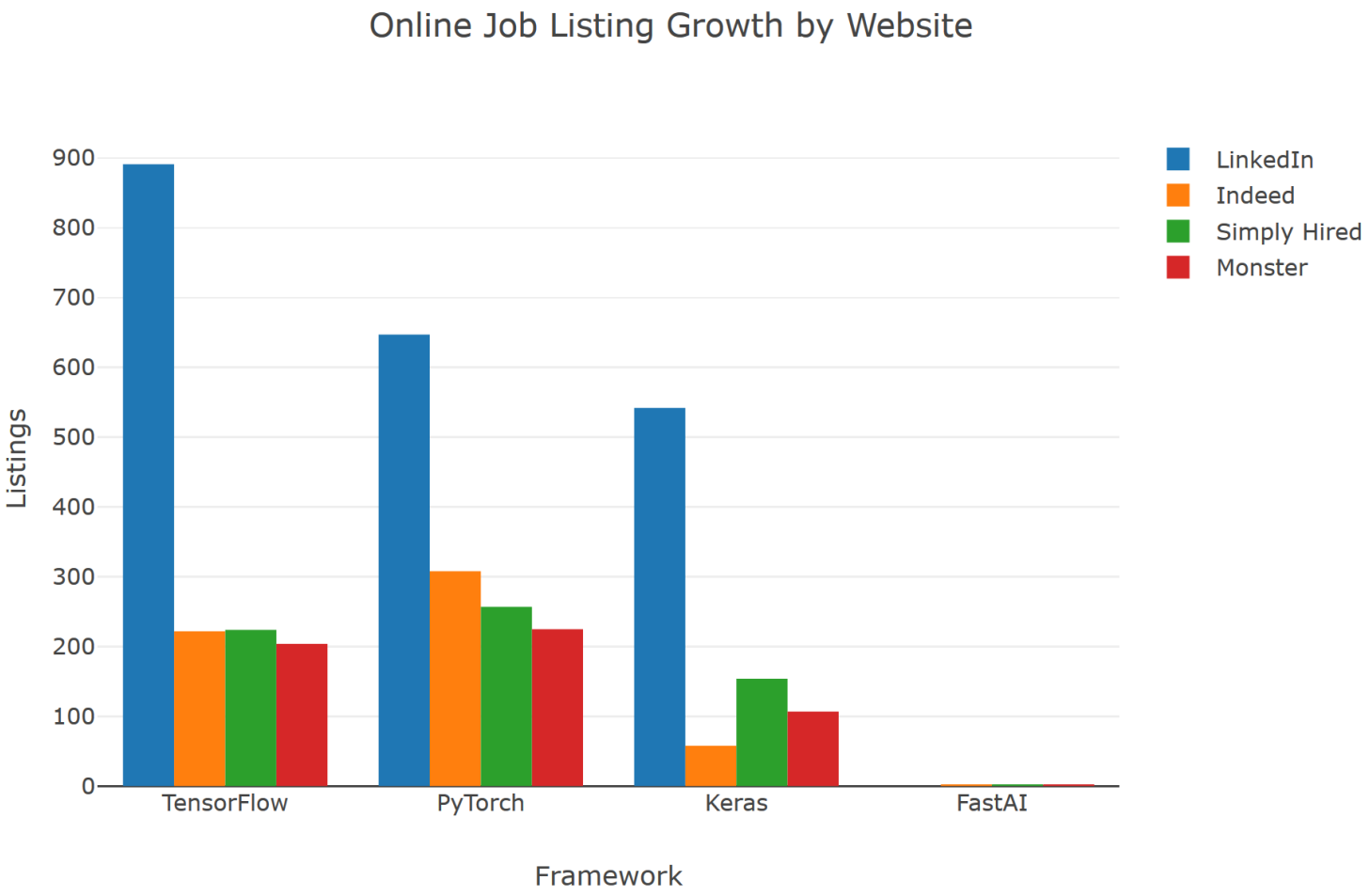
- We will be using TensorFlow 2.1, which was officially released Sept 2019.

Popularity of Deep Learning Frameworks



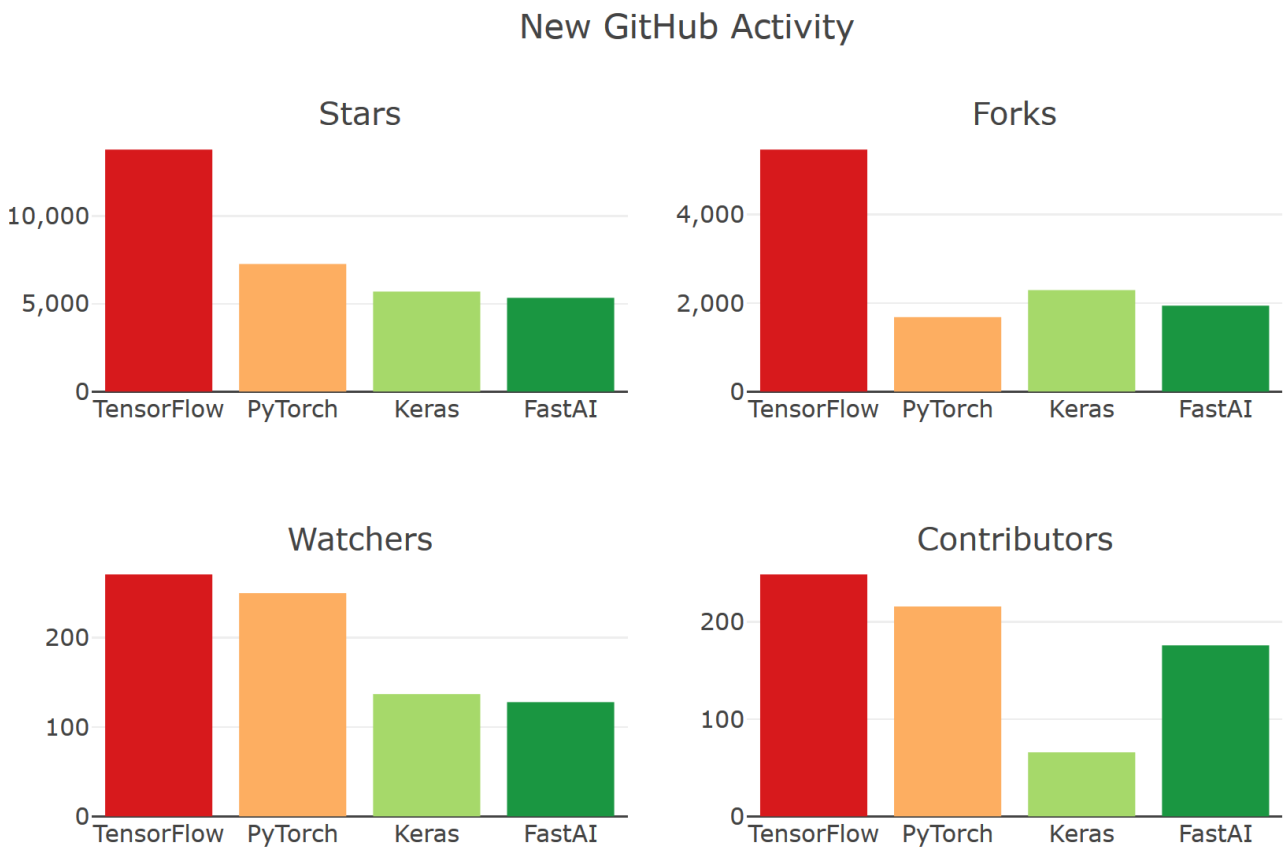
https://www.kaggle.com/discdiver/2019-deep-learning-framework-growth-scores

Popularity of Deep Learning Frameworks



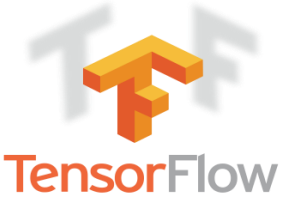
https://www.kaggle.com/discdiver/2019-deep-learning-framework-growth-scores

Popularity of Deep Learning Frameworks



https://www.kaggle.com/discdiver/2019-deep-learning-framework-growth-scores

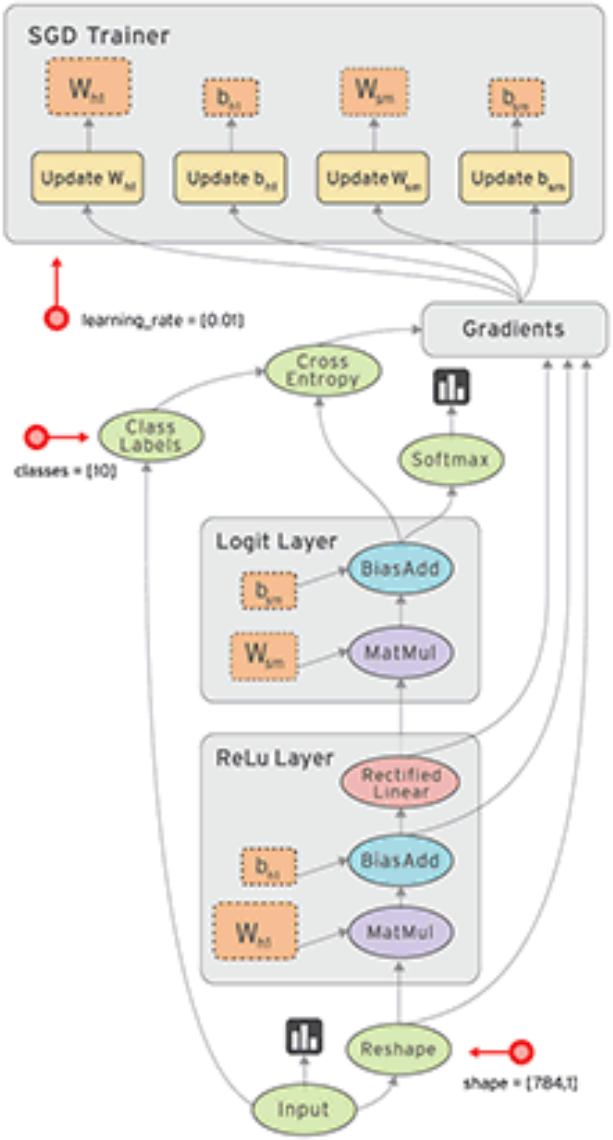
TensorFlow



- Its core is very similar to **NumPy**, but unlike NumPy it provides GPU and TPU support.
- It also supports **distributed computing**. For example, you can distribute the training of a neural network model across multiple GPUs .
- It has both a **high level (Keras)** and **low level** API, which provides excellent flexibility (if specific functionality is not available from the high level API then you can move to the low level API).
- As with all deep learning frameworks it provides **automatic differentiation (autodiff)**, which automatically take care of calculating the gradients for you

TensorFlow – Computational Graphs

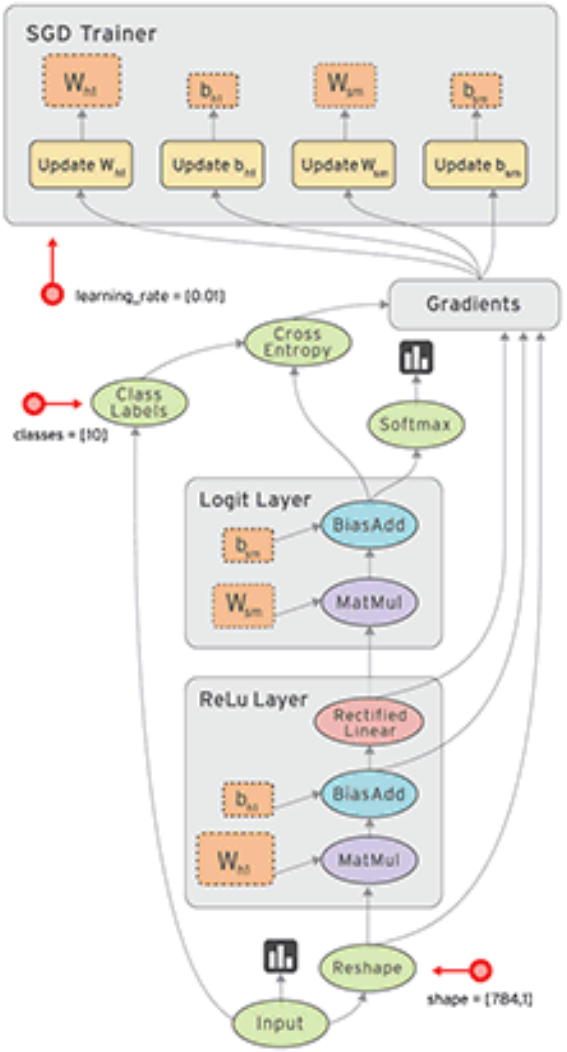
- TF also supports the use of computational graphs
- One of the basic advantages of graphs is that they easily **facilitates parallelism**. The edges of the graph represent dependencies between operations. It is easy for the system to identify operations that can execute in parallel.
- Can be executed directly on a **GPU**.
- 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 – Computational Graphs

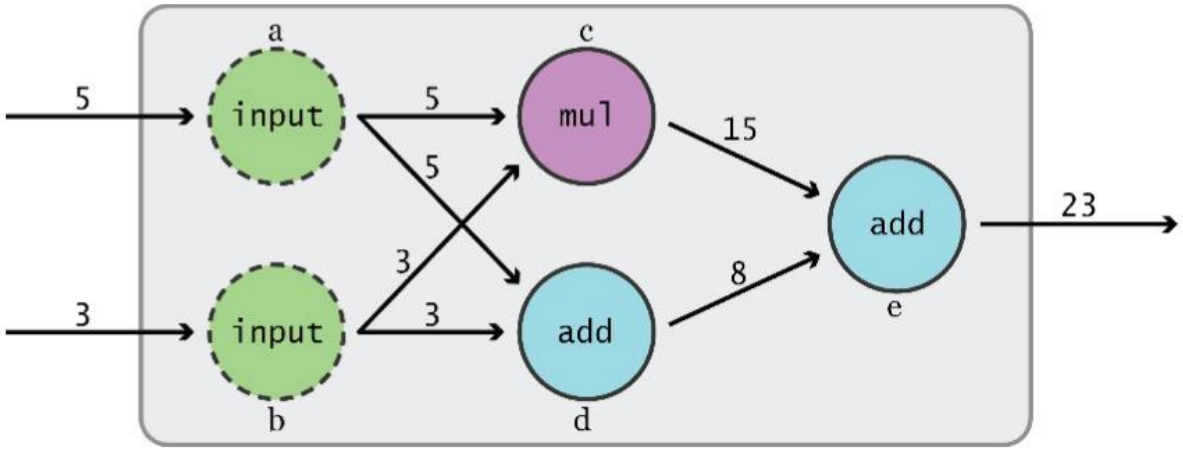
- A graph is a **language-independent** representation of your model. You can build a graph in Python, store it, and restore it in another language.

- 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 1.X Graphs

- Prior to TF2.0, graph based execution was the **default** method of building models.
  - With graph based execution you first had to **define and build an abstract representation** of your model as a computational graph. The graph defines the set of operations that you want performed on data structures known as Tensors.
  - Next you had to use a special **Session** environment to initialize the variables and evaluate the operations defined in the graph.
- Building graphs using this method proved to be very cumbersome. It made **debugging** code very difficult and the model itself was a black box.
- It also represented a **barrier for new users** who are more used to imperative languages.



Graph from TensorFlow for Machine Intelligence

TensorFlow 1.X Graphs

- For example if you were using TF1.X and you execute the following code you might expect that when we print *result* it would print out 8 ((2*3)+2).

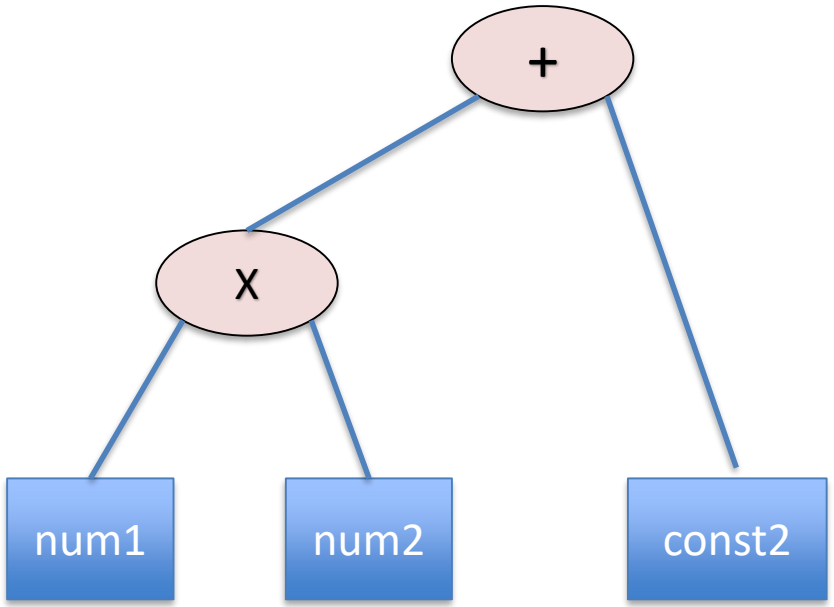
- However, what it would print out is `Tensor("sum:0", shape=(), dtype=float32)`

This indicates that *result* is actually a node within the graph that it provides a rank 0 tensor and dtype of float 32. No calculations have been performed.

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2, name = "product")
result = tf.add(product, const2, name="sum")

print (result)
```

TensorFlow 1.X Graphs

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2, name =
"product")
result = tf.add(product, const2, name="sum")
print (result)
```



- It is important to understand that the code written above does not perform any computation (it just allows us to manually build our model as a graph).
- It creates a computation graph. Even the variables are not initialized yet.
- To evaluate this graph, you need to open a **TensorFlow session** and use it to initialize the variables and evaluate result.

TensorFlow 2.X using Eager Execution

- Since the release of TensorFlow 2.0, **eager execution** is now the default.
  - Eager execution is an **imperative programming environment** that evaluates operations immediately, without the need to first build a computational graph.
  - All operations return **concrete values** instead of constructing a computational graph that the user can then execute later.
  - This makes it much **easier to get started** with TF. It makes it easier to build and train models in TF and reduces the need for much of the boilerplate code that was needed in TF1.X.
  - Code written in Eager execution is also much **easier to debug** as you can use standard Python debuggers such as pdb.

TensorFlow 2.X using Eager Execution

- If we run the following code in Colab (here is a link) it will generate the output immediately as a tensor containing the value 8.0.

```
%tensorflow_version 2.x
import tensorflow as tf
print(tf.__version__)

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2)
print (product)

result = tf.add(product, const2)
print (result)
```

You only need to use this line if using Colab.
%tensorflow_version 2.x

At the time of writing Google Colab has two versions of TensorFlow installed: a 1.15 version and a 2.1 version.

Colab currently uses TF 1.15 by default To enable TF2 execute the following code

Unlike the previous code, this code will execute the operation and output the results directly.

tf.Tensor(**6.0**, shape=(), dtype=float32)
tf.Tensor(**8.0**, shape=(), dtype=float32)

TensorFlow Graph Mode v Eager Execution

- While we will be using Eager Execution throughout this module, it is important to understand that TF **facilitates both Eager and Graph mode**.
- Eager mode is now the default. It is easier to use but this can come at the expense of both **performance** and **deployability**.

```
@tf.function
def circleArea(radius):
  return (radius**2)*3.14
```

- To overcome this issue you can use **tf.function** to **covert your function into a TensorFlow graph**. This allows us to still utilize graphs and their advantages in your code.
- This is most commonly achieved by adding the **@tf.function** decoration before the function (as shown above).
- When we use tf.function() TF analyzes the computations performed by the function and generates an **equivalent computational graph**.

TensorFlow – Components

- Tensorflow programs are mainly composed of two types of objects.
  - **Tensors**: These represent the values on which operations are performed (In a graph, tensors flow through the graph, along the **edges** of the graph.).
  - **Operations**: Operations describe calculations that consume and produce tensors (in graph mode operations are the nodes of the graph.).

Tensors

- A tensor is typically a **multidimensional array** (like what we have used with NumPy ndarray). One minor point is that a tensor can also hold a single scalar value (unlike NumPy).
- The dimensions of a tensor is referred to it's **rank**.
- There are a host of operations built into TF that allow us to create Tensors.

```
import tensorflow as tf

t1 = tf.zeros((4, 5))
t2 = tf.range(3, 15, 2)
t3 = tf.ones((2,2))
t4 = tf.random.uniform(shape=[2,2],dtype=tf.float32)

print (t1)
print (t4)
```

```
tf.Tensor(
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]], shape=(4, 5),
dtype=float32)

tf.Tensor(
[[0.82380414 0.21950543]
 [0.3624072  0.16305423]], shape=(2, 2),
dtype=float32)
```

TensorFlow and NumPy

- As you begin to gain exposure to TF you should also being seeing a high degree of similarity between TF and NumPy operations.

```
import tensorflow as tf
import numpy as np

a = np.zeros((4, 4))
a_t = tf.zeros((4, 4))

b = np.ones((4, 4))
b_t = tf.ones((4, 4))

print (np.sum(b, axis=1))
print (tf.reduce_sum(a, reduction_indices=[1]))

print (a.shape)
print (a_t.get_shape())

np.dot(a, b)
tf.matmul(a_t, b_t)

print (a[:, 1])
print (a_t[:, 1])
```

The main difference between NumPy arrays and TensorFlow Tensors are:
1. Tensors can use be used on a GPU or TPU.
2. Does support automatic calculation of derivatives

Tensors – tf.constant

- You can create an **immutable tensors** using tf.constant.
- Notice below we pass it an pyton 1D and 2D list and NumPy array.
- Notice we can perform **slicing** operations in the same way as we did with NumPy

```
import numpy as np

a = tf.constant([1, 2, 3, 4, 5, 6, 7, 8, 9])
print (a)
print (a.shape)

a = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print (a)
print (a.shape)

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
tf.constant(a)
print (a)

print (a[1:,1:])
```

```
tf.Tensor([1 2 3 4 5 6 7 8 9], shape=(9,),
dtype=int32)
(9,)

tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
(3, 3)

[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[5 6]
 [8 9]]
```

Tensors – tf.Variable

- Clearly it is important that we are also able to modify tensor values. For, example tensors may represent the weights in our neural network and these will need to be modified over time.
- To facilitate this we can use a tf.Variable, which represents **a tensor whose value can be changed by running ops on it**.

- A tf. Variable can be **modified in place** using the **assign()** method (or **assign_add()** or
- **assign_sub()**, which increment or decrement the variable by the given value).
- Specific ops allow you to read and modify the values of this tensor.

```
v = tf.Variable([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print (v)
print (v.shape)

v.assign(v-2)
print (v)

v.assign_sub(v+1)
print (v)
```

```
<tf.Variable 'Variable:0' shape=(3, 3) dtype=int32,
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=int32)>
(3, 3)
<tf.Variable 'Variable:0' shape=(3, 3) dtype=int32,
array([[-1,  0,  1],
       [ 2,  3,  4],
       [ 5,  6,  7]], dtype=int32)>
<tf.Variable 'Variable:0' shape=(3, 3) dtype=int32,
array([[-1, -1, -1],
       [-1, -1, -1],
       [-1, -1, -1]], dtype=int32)>
```

Tensors – tf.Variable

- In the previous slide we use **assign** to change the value of a tf.Variable. Note this is used just to illustrate the that the variable value of mutable (rarely used in practice). Model weights (tf.variables) will generally be updated directly by the optimizers. However, if implementing low level code then we can just use assign.

- Another important point about tf.Variable is that it provides **parameter called trainable**. This allows us to distinguish between variables holding trainable model parameters and other variables.

- For example if we set **trainable = True** then it will be updated as part of the model training process. **By default it is set to True**. More on this later.

Basic Operations in TF

You can a large number of mathematical operations here in TF.math.

tf.add(x,y)	Returns tensor with x + y element-wise.
tf.subtract(x,y)	Returns tensor with x - y element-wise.
tf.multiply(x,y)	Returns tensor x * y element-wise.
tf.scalar_mul(scalar,x)	Multiplies scalar with tensor (scalor*x) and returns multiplied tensor.
tf.div(x,y,name=none)	Divides x / y elementwise and returns tensor
tf.reduce_max(x)	Computes the maximum of elements across dimensions of a tensor.
tf.reduce_mean(x)	Computes the mean of elements across dimensions of a tensor.
tf.round(x)	Rounds the values of a tensor to the nearest integer, element-wise.
tf.matmul(a, b)	Performs matrix multiplication on the arguments

Overloading of Basic Operations in TF

The following are the most commonly **overloaded** methods in TensorFlow. Please note that if only one of the operands is a tensor then the result will still be a tensor.

- tf.negative (unary -)
- tf.add (binary +)
- tf.subtract (binary -)
- tf.multiply (binary elementwise *)
- tf.floordiv (binary // in Python 3)
- tf.truediv (binary / in Python 3)
- tf.pow (binary **)
- tf.logical_and (binary &)
- tf.logical_or (binary |)

It is important to note that == is **not overloaded**. **x == y** will return a Python boolean whether x and y refer to the same tensor. You need to use **tf.equal()** to check for element wise equality.

Operations and Tensors

- As you would expect operations are performed on Tensors. This is our example from earlier.

```
# Basic Example of Eager Execution

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2)
print (product)

result = tf.add(product, const2)
print (result)
```

Operations and Tensors

- As you would expect operations are performed on Tensors. This is our example from earlier.

```
# Basic Example of Eager Execution

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = num1*num2
print (product)

result = product + const2
print (result)
```

Type Conversions in TensorFlow

- Unlike other languages TF **doesn't perform automatic type conversions** when performing operations.
- For example, in TF you **cannot** add a 32 and 64 bit float value. The rationale behind this is that type conversion can be **costly** and **impact performance**.
- Therefore, TF will just raise an **exception** if you try to perform an operation on tensors with incompatible types.
- This is just something to be aware of as it can be a frustrating error when you first start off with TensorFlow.

```
# first line in no problem as we add two float32 tensors
tf.constant(10.5) + tf.constant(5.0)

# the second line in contrast will generate an exception
# we attempt to add a 32 bit float and an integer value
tf.constant(10.5) + tf.constant(5)
```

```
<tf.Tensor: shape=(),
dtype=float32, numpy=15.5>
```

Type Conversions in TensorFlow

- Unlike other languages TF **doesn't perform automatic type conversions** when performing operations.
- For example, in TF you cannot add a 32 and 64 bit float value. The rationale behind this is that type conversion can be **costly** and **impact performance**.
- Therefore, TF will just raise an exception if you try to perform an operation on tensors with incompatible types.
- This is just something to be aware of as it can be a frustrating error when you first start off with TensorFlow.

```
# first line in no problem as we add two float32 tensors
tf.constant(10.5) + tf.constant(5.0)

# the second line in contrast will generate an exception
# we attempt to add a 32 bit float and an integer value
tf.constant(10.5) + tf.constant(5)
```

InvalidArgumentError: cannot compute AddV2 as input #1(zero-based) was expected to be a float tensor but is a int32 tensor [Op:AddV2] name: add/

Type Conversions in TensorFlow

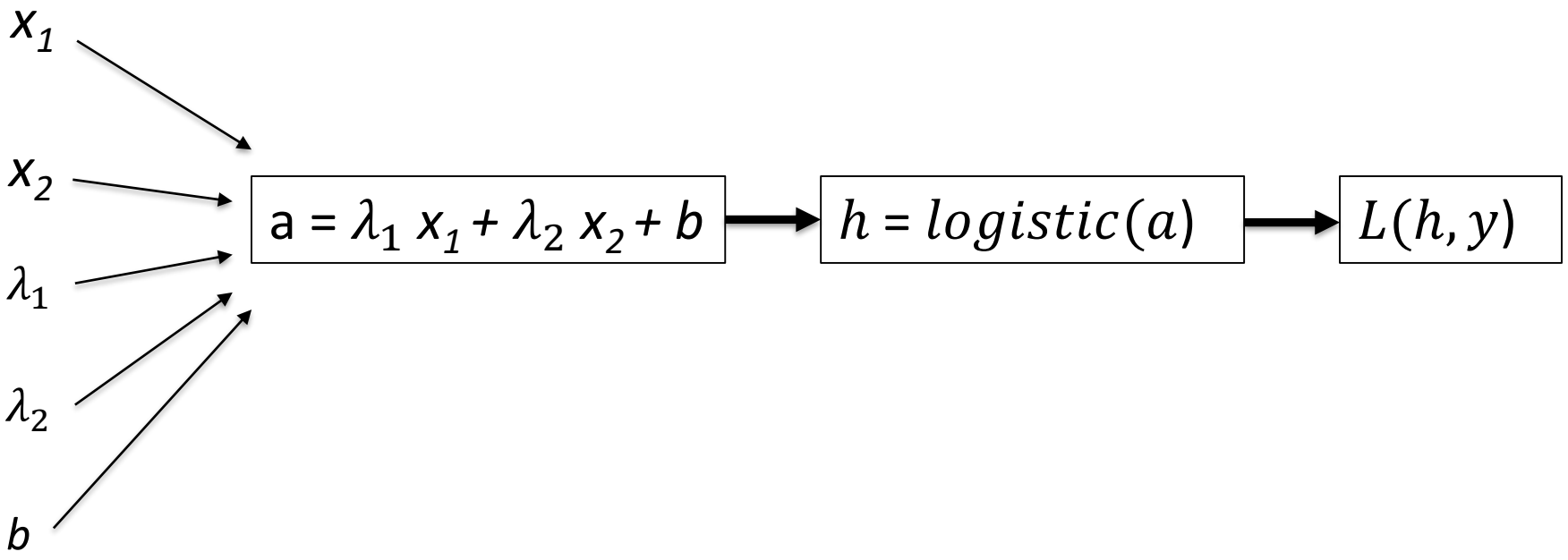
- One common way of addressing this problem is to initially convert your tensors to the appropriate type using casting.
- For example, we can use tf.cast
- It takes in as arguments (i) the original data and (ii) the destination data type.
- It returns the original data casted to the destination data type.

```
num1 = tf.constant(10.5)
num2 = tf.constant(5)

num2 = tf.cast(num2, tf.float32)

print (num1+num2)
```

Automatic Differentiation in TensorFlow

- TensorFlow provides **tf.GradientTape** to facilitate automatic differentiation. GradientTape allows us to calculate the gradient of a function with respect to it's trainable input varaibles.

- Tensorflow "**records**" all operations executed in the context of a tf.GradientTape. It then uses the tape to compute the **gradients of a function** with respect to input variables.



Automatic Differentiation in TensorFlow

- We initially create an **instance of tf.GradientTape**, which records all (forward-pass) operations once it has been created.

- To determine the **gradients of the loss function** with respect to some variables, we call **tape.gradient**.
  - The first argument is the "**target**" for the calculation, i.e. typically the output of the loss function (or whatever function you are trying to optimize)
  - The second argument is the "**source**" these are the variables values you can alter in order to optimize the target loss function.

- The **gradient** function will **return** the partial derivatives of each of the listed trainable variables with respect to the (loss) function.

- By default, the resources held by a **GradientTape are released** as soon as GradientTape.gradient() method is called.

- We can either update the variables ourselves or more commonly we can use an existing optimizer to update the gradients for us.

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2
```

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2

# Create a tensorflow variable with an itial value of 10.0
x = tf.Variable(10.0, tf.float32)
```

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2

# Create a tensorflow variable with an itial value of 10.0
x = tf.Variable(10.0, tf.float32)

with tf.GradientTape() as tape:
  prediction = simpleFunc(x)
  current_grad = tape.gradient(prediction, x)

print (current_grad)
```

GradientTape will record all operation that take place with the with statement.

```
tf.Tensor(920.0, shape=(), dtype=float32)
```

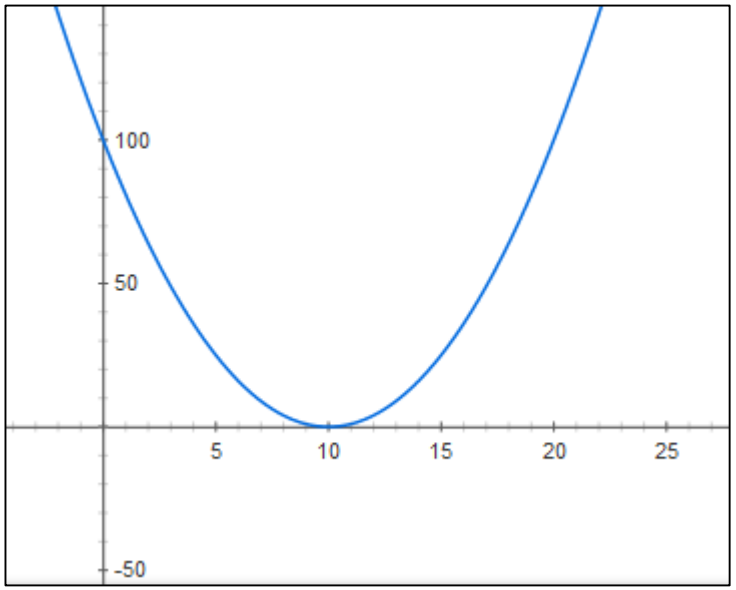
Using TensorFlow to Minimize a Function

- To illustrate the use of automatic differentiation we will start with the following example, we are going to write a TensorFlow program to find the minimum value of x in the following function:

- $X^2 – 20X + 100$

- The minimum value of X is obvious but we will illustrate the process of building a simple TensorFlow program to solve this problem.

- Let’s first built a function representing our quadratic equation above.



```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic
```

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)
```

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)
```

We want to determine the value of x that will minimize our quadratic equation. Therefore, we iterate 50 times. Each time we iterate we update the value of the variable x using our gradient descent rule.

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)
```

Gradient tape once initialized records all operations performed within it's context.

Next we call gradient, which calculate the gradients of the prediction function with respect to x the input variable.

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)

  x.assign_sub(gradient * learning_rate)

  print (x)
```

The next step is to use my normal gradient descent update rule to update the tensorflow variable x.

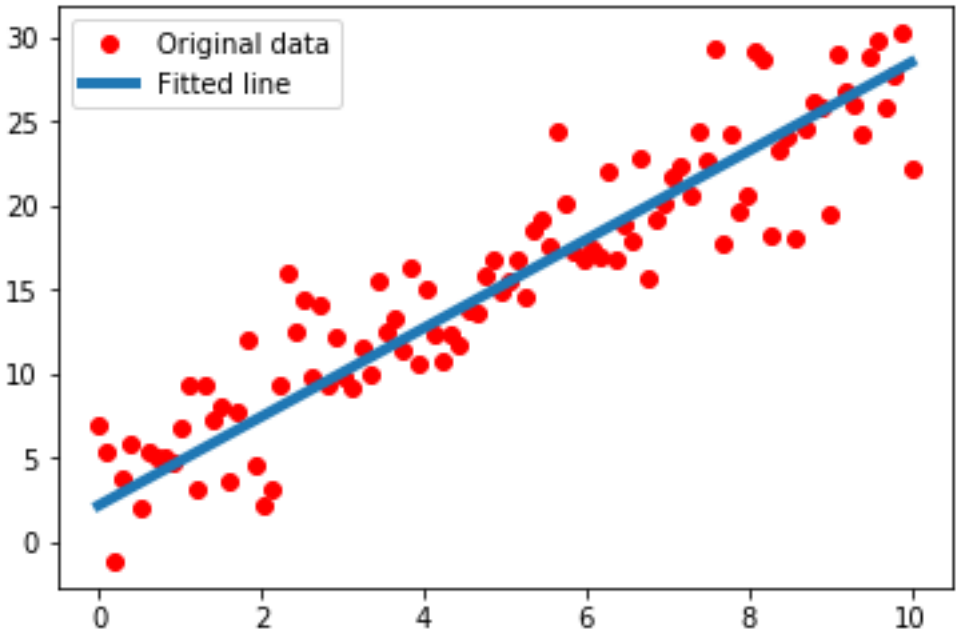
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=42.0>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=35.6>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=30.48>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=26.383999>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=23.107199>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=20.48576>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=18.388607>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=16.710886>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=15.368709>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=14.294967>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=13.435973>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=12.748778>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=12.199022>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.759218>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.407374>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.125899>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.90072>
.....
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.020282>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.016226>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.01298>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.010385>

Linear Regression Using TensorFlow

- In this example we are going to build a Linear Regression algorithm using TensorFlow using GradientTape to calculate the derivatives.

  - $h(x) = \lambda_1 x + b$

Which of the above should be a variable in our TensorFlow program?

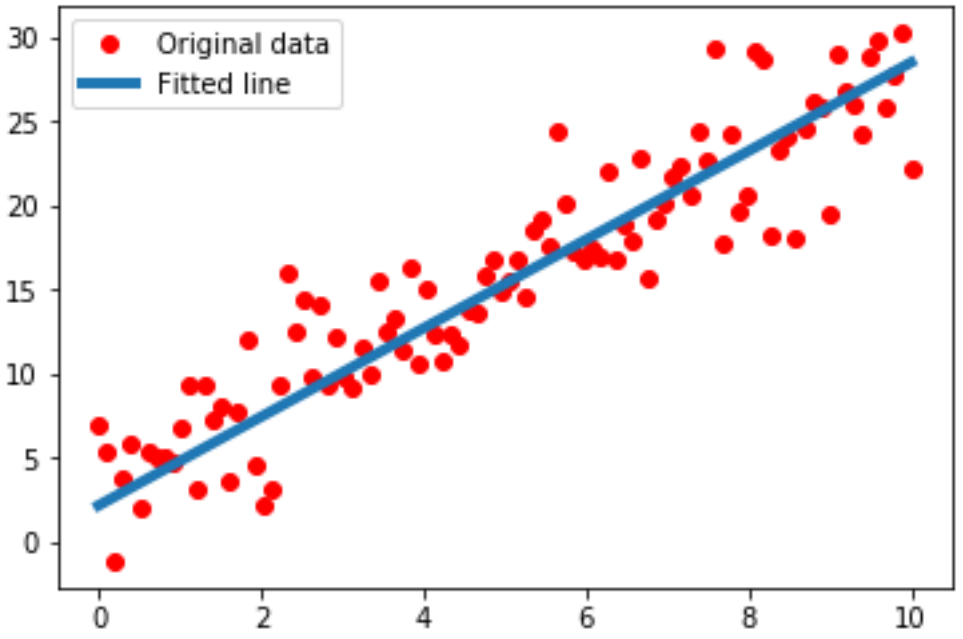


Linear Regression Using TensorFlow

- In this example we are going to build a Linear Regression algorithm using TensorFlow using GradientTape to calculate the derivitatives.

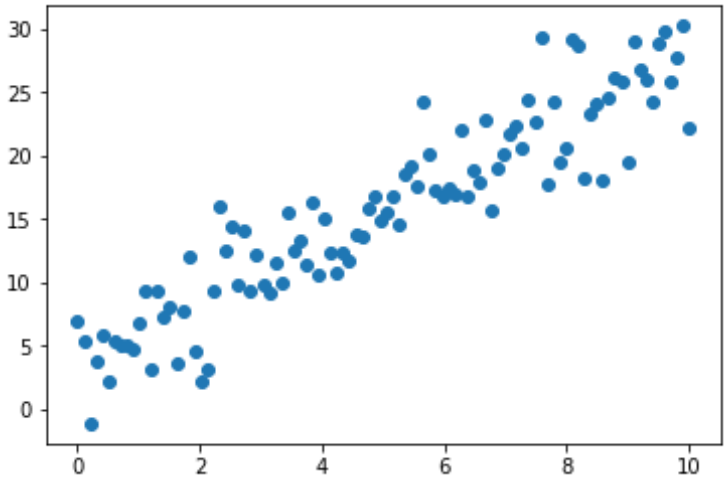
  - $h(x) = \lambda_1 x + b$

- You will remember our equation for linear regression above. There are two variables that we must adjust in order to minimize the loss function.
- Those are $\boldsymbol{\lambda_1}$ and **b**. Note in this example x is just the training data, which will not change.



Linear Regression – Preparing Data

- In the code we create some sample data for our linear regression model.
- Notice we also specify some of the parameters for our linear regression model.




```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np


def generateData():

  np.random.seed(10)
  train_X=np.linspace(0,10,100)
  noise=np.random.normal(0,1.5,100)

  train_Y=((2.5*train_X)+3) +noise

  return train_X, train_Y


x_train, y_train = generateData()
plt.scatter(x_train,y_train)
plt.legend()
plt.show()
```

Linear Regression – Model and Loss Function

- Next we create a function for our linear regression model (takes in training data and output results)

- We also create a function for our squared error cost. . Notice we are comparing the predicted output with the actual labels Y.

```
def predict(x, w, c):
  y = w * x + c
  return y


def loss_func(y_pred, y_true):
  error = y_pred - y_true
  sumSqErrors = tf.reduce_sum(error**2)
  currentLoss = sumSqErrors/(2*len(y_pred))
  return currentLoss
```

$$\frac{1}{2m}\sum_{i=0}^{m}((h(x^i)-y^i))^2$$

Linear Regression – Training Process

- We create a variable w and c and set their values to 0.
- As with the previous example each time we iterate, we create an instance of GradientTape, which tracks the forward pass operations. We then calculate the derivatives of our loss function with respect to the variables w and c. Next we use these gradients in the normal way to the variable values for our linear regression model.

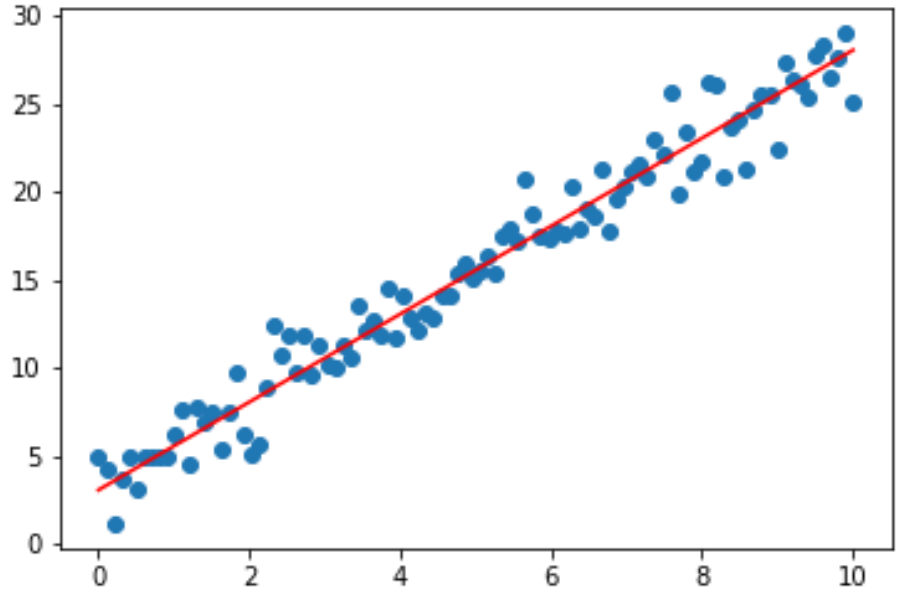
```
w = tf.Variable(0.)
c = tf.Variable(0.)

learning_rate = 0.001
steps = 10000

for i in range(steps):

  with tf.GradientTape() as tape:
    predictions = predict(x_train, w, c)
    loss = loss_func(predictions, y_train)

  gradients = tape.gradient(loss, [w, c])

  m.assign_sub(gradients[0] * learning_rate)
  c.assign_sub(gradients[1] * learning_rate)

  if i % 100 == 0:
        print("Step ",i,", Loss ",loss.numpy())
```

Linear Regression – Training Process

- As we run the code we should see the loss gradually decline.

```
plt.scatter(x_train,y_train)
plt.plot(x_train,predict(x_train),"r")
plt.show()
```

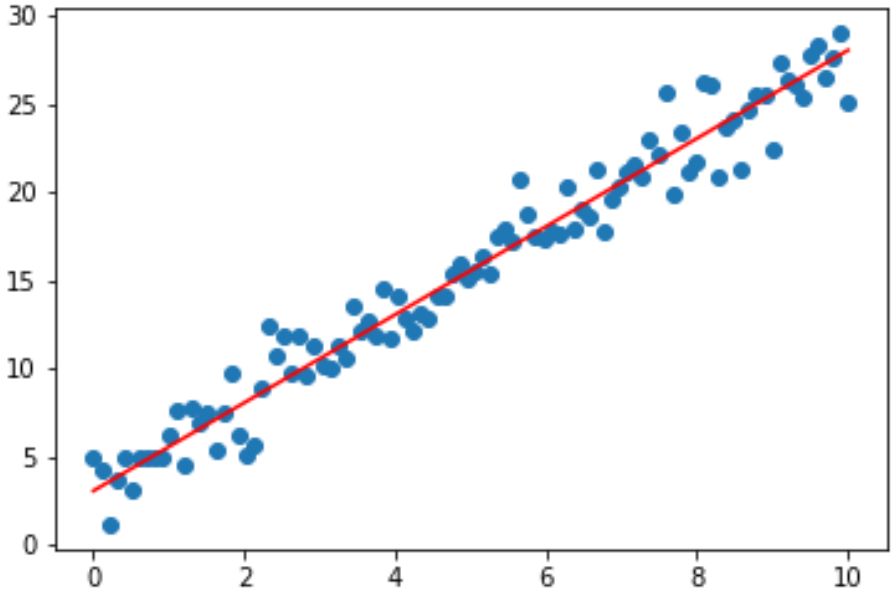
- The following code generates a graph showing the linear regression model generated after our gradient descent has updated the values of m and c.



Linear Regression – Training Process

- In the previous examples we have only used standard gradient descent.
- However, TensorFlow offers a host of in-built optimizers (Adam, Adagrad, RMSProp, etc) that we can plug into our code.
- You can find the full list of optimizers at tf.keras.optimizers

- The code on the next slide will demonstrate how to alter our linear regression model to use the inbuilt Adam optimizer.



Linear Regression – Training with Adam Optimizer

```
w = tf.Variable(0.)
c = tf.Variable(0.)
adam_optimizer = tf.keras.optimizers.Adam()


learning_rate = 0.001
steps = 10000

for i in range(steps):

  with tf.GradientTape() as tape:
    predictions = predict(x_train,w, c)
    loss = loss_func(predictions, y_train)

  gradients = tape.gradient(loss, [w, c])

  adam_optimizer.apply_gradients(zip(gradients, [w,c]))
```

As you can see there has been little change to the code from the previous example.

1. We create an instance of the Adam optimizers.
2. Each time we iterate through out training loop we calculate the gradient for m and c. We then call the Adam apply_gradients method in the Adam optimizer and pass it the gradient and the associate variables and that's it!