Tensorflow-2.x

> TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

Why Tensorflow?

TensorFlow is a popular and widely used open-source machine learning framework developed by Google. It offers a range of features and benefits that make it a powerful tool for building and deploying machine learning models. Here are some reasons why TensorFlow is commonly used:

- Flexibility: TensorFlow provides a flexible and modular architecture that allows developers to build and customize machine learning models for a wide variety of tasks. It supports both high-level and low-level APIs, giving users the flexibility to work at different levels of abstraction.
- Scalability: TensorFlow is designed to handle large-scale machine learning projects. It enables efficient distributed computing across multiple CPUs and GPUs, making it suitable for training models on large datasets.
- Wide range of applications: TensorFlow can be used for a diverse range of machine learning tasks, including image and speech recognition, natural language processing, recommendation systems, and more. It supports various neural network architectures, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers.
- Community and ecosystem: TensorFlow has a large and active community of developers, researchers, and enthusiasts. This community contributes to the development of the framework by sharing code, providing support, and creating libraries and tools that extend TensorFlow's functionality. This vibrant ecosystem makes it easier to find resources, tutorials, and pre-trained models.
- Visualization and debugging: TensorFlow includes tools for visualizing and debugging models, which can aid in understanding the behavior of the model during training and inference. It provides built-in support for TensorBoard, a web-based tool for visualizing metrics, model graphs, and other aspects of the training process.
- Deployment options: TensorFlow offers multiple deployment options, allowing models to be deployed in a variety of environments. It supports deployment on different platforms, including desktops, servers, mobile devices, and even specialized hardware such as Google's Tensor Processing Units (TPUs).
- Integration with other libraries and frameworks: TensorFlow can be easily integrated with other popular libraries and frameworks in the Python ecosystem, such as NumPy, Pandas, and scikit-learn. This enables seamless data manipulation, preprocessing, and post-processing tasks in conjunction with TensorFlow's capabilities.
- Continued development and support: TensorFlow is actively developed and maintained by Google and the TensorFlow community. Regular updates and improvements ensure that the framework stays up to date with the latest advancements in machine learning research and industry practices.

These are just a few reasons why TensorFlow is a popular choice for machine learning tasks. However, it's worth noting that the choice of framework ultimately depends on the specific requirements and preferences of the user.

Installation of Tensorflow

> TensorFlow is tested and supported on the following 64-bit systems:
>
> 1.Ubuntu 16.04 or later
>
> 2.Windows 7 or later
>
> 3.macOS 10.12.6 (Sierra) or later (no GPU support)
>
> 4.Raspbian 9.0 or later

For installing latest version of Tensorflow

> **pip install tensorflow**
>
> To run from Anaconda Prompt
>
> **!pip install tensorflow**
>
> To run from Jupyter Notebook

For installing a specific version of Tensorflow

> **pip install tensorflow==2.x**
>
> To run from Anaconda Prompt
>
> **!pip install tensorflow==2.x**
>
> To run from Jupyter Notebook

[Tensorflow Documentation]

Both Tensorflow 2.0 and Keras have been released for four years (Keras was released in March 2015, and Tensorflow was released in November of the same year). The rapid development of deep learning in the past days, we also know some problems of Tensorflow1.x and Keras:

- Using Tensorflow means programming static graphs, which is difficult and inconvenient for programs that are familiar with imperative programming
- Tensorflow api is powerful and flexible, but it is more complex, confusing and difficult to use.
- Keras api is productive and easy to use, but lacks flexibility for research

Version Check

```
import tensorflow as tf

print("TensorFlow version: {}".format(tf.__version__))
print("Eager execution is: {}".format(tf.executing_eagerly()))
print("Keras version: {}".format(tf.keras.__version__))
```

```
TensorFlow version: 2.12.0
Eager execution is: True
Keras version: 2.12.0
```

Tensorflow2.0 is a combination design of Tensorflow1.x and Keras. Considering user feedback and framework development over the past four years, it largely solves the above problems and will become the future machine learning platform.

> Tensorflow 2.0 is built on the following core ideas:

- The coding is more pythonic, so that users can get the results immediately like they are programming in numpy
- Retaining the characteristics of static graphs (for performance, distributed, and production deployment), this makes TensorFlow fast, scalable, and ready for production.
- Using Keras as a high-level API for deep learning, making Tensorflow easy to use and efficient
- Make the entire framework both high-level features (easy to use, efficient, and not flexible) and low-level features (powerful and scalable, not easy to use, but very flexible)

> Eager execution is the default in TensorFlow 2 and, as such, needs no special setup. The following code can be used to find out whether a CPU or GPU is in use and if it's a GPU, whether that GPU is #0.

GPU/CPU Check

```
if tf.test.is_gpu_available():
    print('Running on GPU')
else:
    print('Running on CPU')
```

```
Running on GPU
```

```
tf.config.list_physical_devices('CPU')
```

```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

```
tf.config.list_physical_devices('GPU')
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Tensor Constant

```
ineuron = tf.constant(42)
ineuron
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

```
ineuron.numpy()
```

```
42
```

```
ineuron1 = tf.constant(1, dtype = tf.int64)
ineuron1
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=1>
```

```
ineuron_x = tf.constant([[4,2],[9,5]])
print(ineuron_x)
```

```
tf.Tensor(
[[4 2]
 [9 5]], shape=(2, 2), dtype=int32)
```

```
ineuron_x.numpy()
```

```
array([[4, 2],
       [9, 5]], dtype=int32)
```

```
print('shape:',ineuron_x.shape)
print(ineuron_x.dtype)
```

```
shape: (2, 2)
<dtype: 'int32'>
```

Commonly used method is to generate constant tf.ones and the tf.zeros like of numpy np.ones & np.zeros

```
print(tf.ones(shape=(2,3)))
```

```
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
```

```
print(tf.zeros(shape=(3,2)))
```

```
tf.Tensor(
[[0. 0.]
 [0. 0.]
 [0. 0.]], shape=(3, 2), dtype=float32)
```

```
import tensorflow as tf

const2 = tf.constant([[3,4,5], [3,4,5]])
const1 = tf.constant([[1,2,3], [1,2,3]])
result = tf.add(const1, const2)

print(result)
```

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

> We have defined two constants and we add one value to the other. As a result, we got a Tensor object with the result of the adding.

Random constant

```
tf.random.normal(shape=(2,2),mean=0,stddev=1.0)
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[-0.37854993,  0.7752413 ],
       [ 1.4218645 ,  1.1185031 ]], dtype=float32)>
```

```
tf.random.uniform(shape=(2,2),minval=0,maxval=10,dtype=tf.int32)
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[7, 6],
       [7, 3]], dtype=int32)>
```

Variables

> A variable is a special tensor that is used to store variable values and needs to be initialized with some values

Declaring variables

```
var0 = 24 # python variable
var1 = tf.Variable(42) # rank 0 tensor
var2 = tf.Variable([ [ [0., 1., 2.], [3., 4., 5.] ], [ [6., 7., 8.], [9., 10., 11.] ] ]) #rank 3 tensor
var0, var1, var2
```

```
(24,
 <tf.Variable 'Variable:0' shape=() dtype=int32, numpy=42>,
 <tf.Variable 'Variable:0' shape=(2, 2, 3) dtype=float32, numpy=
 array([[[ 0.,  1.,  2.],
         [ 3.,  4.,  5.]],

        [[ 6.,  7.,  8.],
         [ 9., 10., 11.]]], dtype=float32)>)
```

> TensorFlow will infer the datatype, defaulting to tf.float32 for floats and tf.int32 for integers

The datatype can be explicitly specified

```
float_var64 = tf.Variable(89, dtype = tf.float64)
float_var64.dtype
```

```
tf.float64
```

TensorFlow has a large number of built-in datatypes.

datatype	description
tf.float16	16-bit half-precision floating-point.
tf.float32	32-bit single-precision floating-point.
tf.float64	64-bit double-precision floating-point.
tf.bfloat16	16-bit truncated floating-point.
tf.complex64	64-bit single-precision complex.
tf.complex128	128-bit double-precision complex.
tf.int8	8-bit signed integer.
tf.uint8	8-bit unsigned integer.
tf.uint16	16-bit unsigned integer.
tf.uint32	32-bit unsigned integer.
tf.uint64	64-bit unsigned integer.
tf.int16	16-bit signed integer.
tf.int32	32-bit signed integer.
tf.int64	64-bit signed integer.
tf.bool	Boolean.
tf.string	String.
tf.qint8	Quantized 8-bit signed integer.
tf.quint8	Quantized 8-bit unsigned integer.
tf.qint16	Quantized 16-bit signed integer.
tf.quint16	Quantized 16-bit unsigned integer.
tf.qint32	Quantized 32-bit signed integer.
tf.resource	Handle to a mutable resource.
tf.variant	Values of arbitrary types.

To reassign a variable, use var.assign()

```
var_reassign = tf.Variable(89.)
var_reassign
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=89.0>
```

```
var_reassign.assign(98.)
var_reassign
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=98.0>
```

```
initial_value = tf.random.normal(shape=(2,2))
a = tf.Variable(initial_value)
print(a)
```

```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[ 1.007538  ,  0.90166533],
       [-1.1097176 , -1.9102186 ]], dtype=float32)>
```

> We can assign "=" with assign (value), or assign_add (value) with "+ =", or assign_sub (value) with "-="

```
new_value = tf.random.normal(shape=(2, 2))
a.assign(new_value)
for i in range(2):
    for j in range(2):
        assert a[i, j] == new_value[i, j]
```

```
added_value = tf.random.normal(shape=(2,2))
a.assign_add(added_value)
for i in range(2):
    for j in range(2):
        assert a[i,j] == new_value[i,j]+added_value[i,j]
```

Shaping a tensor

```
tensor = tf.Variable([ [ [0., 1., 2.], [3., 4., 5.] ], [ [6., 7., 8.], [9., 10., 11.] ] ]) # tensor variable
print(tensor.shape)
```

```
(2, 2, 3)
```

Tensors may be reshaped and retain the same values, as is often required for constructing neural networks.

```
tensor1 = tf.reshape(tensor,[2,6]) # 2 rows 6 cols
tensor2 = tf.reshape(tensor,[1,12]) # 1 rows 12 cols
tensor1
```

```
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.]], dtype=float32)>
```

```
tensor2 = tf.reshape(tensor,[1,12]) # 1 row 12 columns
tensor2
```

```
<tf.Tensor: shape=(1, 12), dtype=float32, numpy=
array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.]],
      dtype=float32)>
```

Ranking (dimensions) of a tensor

> The rank of a tensor is the number of dimensions it has, that is, the number of indices that are required to specify any particular element of that tensor.

```
tf.rank(tensor)
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=3>
```

> (the shape is () because the output here is a scalar value)

Specifying an element of a tensor

```
tensor3 = tensor[1, 0, 2] # slice 1, row 0, column 2
tensor3
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Casting a tensor to a NumPy/Python variable

```
print(tensor.numpy())
```

```
[[[ 0.  1.  2.]
  [ 3.  4.  5.]]

 [[ 6.  7.  8.]
  [ 9. 10. 11.]]]
```

```
print(tensor[1, 0, 2].numpy())
```

```
8.0
```

Finding the size (number of elements) of a tensor

```
tensor_size = tf.size(input=tensor).numpy()
tensor_size
```

```
12
```

```
#the datatype of a tensor
tensor3.dtype
```

```
tf.float32
```

Tensorflow mathematical operations

> Can be used as numpy for artificial operations. Tensorflow can not execute these operations on the GPU or TPU.

```
a = tf.random.normal(shape=(2,2))
b = tf.random.normal(shape=(2,2))
c = a+b
d = tf.square(c)
e = tf.exp(c)
print(a)
print(b)
print(c)
```

```
print(d)
print(e)
```

```
tf.Tensor(
[[-0.24671447  0.8228442 ]
 [ 1.4005157  -0.21337971]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[-1.0893056  -0.98363787]
 [-0.5615219  -0.26913118]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[-1.3360201  -0.16079366]
 [ 0.8389938  -0.4825109 ]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[1.7849498  0.0258546 ]
 [0.7039106  0.23281677]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[0.26288986 0.8514677 ]
 [2.3140376  0.61723167]], shape=(2, 2), dtype=float32)
```

Performing element-wise primitive tensor operations

```
tensor*tensor
```

```
<tf.Tensor: shape=(2, 2, 3), dtype=float32, numpy=
array([[[  0.,   1.,   4.],
        [  9.,  16.,  25.]],

       [[ 36.,  49.,  64.],
        [ 81., 100., 121.]]], dtype=float32)>
```

Broadcasting

> Element-wise tensor operations support broadcasting in the same way that NumPy arrays do.

> The simplest example is that of multiplying a tensor by a scalar:

```
tensor4 = tensor*4
print(tensor4)
```

```
tf.Tensor(
[[[ 0.  4.  8.]
  [12. 16. 20.]]
```

```
 [[24. 28. 32.]
  [36. 40. 44.]]], shape=(2, 2, 3), dtype=float32)
```

> the scalar multiplier 4 is—conceptually, at least—expanded into an array that can be multiplied element-wise with t2.

Transpose Matrix multiplication

```
matrix_u = tf.constant([[3,4,3]])
matrix_v = tf.constant([[1,2,1]])

tf.matmul(matrix_u, tf.transpose(a=matrix_v))
```

```
<tf.Tensor: shape=(1, 1), dtype=int32, numpy=array([[14]], dtype=int32)>
```

Casting a tensor to another (tensor) datatype

```
i = tf.cast(tensor1, dtype=tf.int32)
i
```

```
<tf.Tensor: shape=(2, 6), dtype=int32, numpy=
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]], dtype=int32)>
```

With truncation

```
j = tf.cast(tf.constant(4.9), dtype=tf.int32)
j
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=4>
```

Declaring Ragged tensors

`A ragged tensor is a tensor with one or more ragged dimensions. Ragged dimensions are dimensions that have slices that may have different lengths.There are a variety of methods for declaring ragged arrays, the simplest being a constant ragged array.`

The following example shows how to declare a constant ragged array and the lengths of the individual slices:

```
ragged =tf.ragged.constant([[5, 2, 6, 1], [], [4, 10, 7], [8], [6,7]])

print(ragged)
print(ragged[0,:])
print(ragged[1,:])
print(ragged[2,:])
print(ragged[3,:])
print(ragged[4,:])
```

```
<tf.RaggedTensor [[5, 2, 6, 1], [], [4, 10, 7], [8], [6, 7]]>
tf.Tensor([5 2 6 1], shape=(4,), dtype=int32)
tf.Tensor([], shape=(0,), dtype=int32)
tf.Tensor([ 4 10  7], shape=(3,), dtype=int32)
tf.Tensor([8], shape=(1,), dtype=int32)
tf.Tensor([6 7], shape=(2,), dtype=int32)
```

Finding the squared difference between two tensors

```
varx = [1,3,5,7,11]
vary = 5
varz = tf.math.squared_difference(varx,vary)
varz
```

```
<tf.Tensor: shape=(5,), dtype=int32, numpy=array([16,  4,  0,  4, 36], dtype=int32)>
```

> The Python variables, varx and vary, are cast into tensors and that vary is then broadcast across varx in this example. So, for example, the first calculation is (1-5)2 = 16.

Finding the mean

> The following is the signature of tf.reduce_mean().

```
Note that this is equivalent to np.mean, except that it infers the return datatype from the input tensor, whereas np.mean
allows you to specify the output type (defaulting to float64):
```

```
tf.reduce_mean(input_tensor, axis=None, keepdims=None, name=None)
```

```
#Defining a constant
numbers = tf.constant([[4., 5.], [7., 3.]])
```

Find the mean across all axes (use the default axis = None)

```
tf.reduce_mean(input_tensor=numbers)
#( 4. + 5. + 7. + 3.)/4 = 4.75
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=4.75>
```

Find the mean across columns (that is, reduce rows) with this:

```
tf.reduce_mean(input_tensor=numbers, axis=0) # [ (4. + 7. )/2 , (5. + 3.)/2 ] = [5.5, 4.]
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([5.5, 4. ], dtype=float32)>
```

When keepdims is True, the reduced axis is retained with a length of 1:

```
tf.reduce_mean(input_tensor=numbers, axis=0, keepdims=True)
```

```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[5.5, 4. ]], dtype=float32)>
```

Find the mean across rows (that is, reduce columns) with this:

```
tf.reduce_mean(input_tensor=numbers, axis=1) # [ (4. + 5. )/2 , (7. + 3. )/2] = [4.5, 5]
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([4.5, 5. ], dtype=float32)>
```

When keepdims is True, the reduced axis is retained with a length of 1:

```
tf.reduce_mean(input_tensor=numbers, axis=1, keepdims=True)
```

```
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[4.5],
       [5. ]], dtype=float32)>
```

Generating tensors filled with random values

Using tf.random.normal()

> tf.random.normal() outputs a tensor of the given shape filled with values of the dtype type from a normal distribution.

> The required signature is as follows:

> tf. random.normal(shape, mean = 0, stddev =2, dtype=tf.float32, seed=None, name=None)

```
tf.random.normal(shape = (3,2), mean=10, stddev=2, dtype=tf.float32, seed=None, name=None)
ran = tf.random.normal(shape = (3,2), mean=10.0, stddev=2.0)
print(ran)
```

```
tf.Tensor(
[[11.012381   9.820841 ]
 [11.514592  11.424604 ]
 [ 8.9686985  7.321087 ]], shape=(3, 2), dtype=float32)
```

Using tf.random.uniform()

> The required signature is this:

> tf.random.uniform(shape, minval = 0, maxval= None, dtype=tf.float32, seed=None, name=None)

```
This outputs a tensor of the given shape filled with values from a uniform distribution in the range minval to maxval, where
the lower bound is inclusive but the upper bound isn't. Take this, for example:
```

```
tf.random.uniform(shape = (2,4), minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)
```

```
<tf.Tensor: shape=(2, 4), dtype=float32, numpy=
array([[0.69793105, 0.12818623, 0.36479974, 0.6948261 ],
       [0.99294233, 0.607391  , 0.13492548, 0.45762825]], dtype=float32)>
```

Setting the seed

```
tf.random.set_seed(11)
ran1 = tf.random.uniform(shape = (2,2), maxval=10, dtype = tf.int32)
ran2 = tf.random.uniform(shape = (2,2), maxval=10, dtype = tf.int32)
print(ran1) #Call 1
print(ran2)
```

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

```
tf.random.set_seed(11) #same seed
ran1 = tf.random.uniform(shape = (2,2), maxval=10, dtype = tf.int32)
ran2 = tf.random.uniform(shape = (2,2), maxval=10, dtype = tf.int32)
print(ran1) #Call 2
print(ran2)
```

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

Practical example of Random values using Dices

```
dice1 = tf.Variable(tf.random.uniform([10, 1], minval=1, maxval=7, dtype=tf.int32))
dice2 = tf.Variable(tf.random.uniform([10, 1], minval=1, maxval=7, dtype=tf.int32))
# We may add dice1 and dice2 since they share the same shape and size.
dice_sum = dice1 + dice2
# We've got three separate 10x1 matrices. To produce a single
# 10x3 matrix, we'll concatenate them along dimension 1.
resulting_matrix = tf.concat(values=[dice1, dice2, dice_sum], axis=1)
print(resulting_matrix)
```

```
tf.Tensor(
[[ 5  5 10]
 [ 4  3  7]
 [ 5  3  8]
```

```
 [ 3  3  6]
 [ 1  4  5]
 [ 4  1  5]
 [ 5  1  6]
 [ 6  4 10]
 [ 3  3  6]
 [ 2  3  5]], shape=(10, 3), dtype=int32)
```

Finding the indices of the largest and smallest element

> The signatures of the functions are as follows:

> `tf.argmax(input, axis=None, name=None, output_type=tf.int64 )`

> `tf.argmin(input, axis=None, name=None, output_type=tf.int64 )`

```
# 1-D tensor
t5 = tf.constant([2, 11, 5, 42, 7, 19, -6, -11, 29])
print(t5)


i = tf.argmax(input=t5)
print('index of max; ', i)
print('Max element: ',t5[i].numpy())


i = tf.argmin(input=t5,axis=0).numpy()
print('index of min: ', i)
print('Min element: ',t5[i].numpy())


t6 = tf.reshape(t5, [3,3])
print(t6)


i = tf.argmax(input=t6,axis=0).numpy() # max arg down rows
print('indices of max down rows; ', i)

i = tf.argmin(input=t6,axis=0).numpy() # min arg down rows
print('indices of min down rows ; ',i)
print(t6)
```

```
i = tf.argmax(input=t6,axis=1).numpy() # max arg across cols
print('indices of max across cols: ',i)

i = tf.argmin(input=t6,axis=1).numpy() # min arg across cols
print('indices of min across cols: ',i)
```

```
tf.Tensor([  2  11   5  42   7  19  -6 -11  29], shape=(9,), dtype=int32)
index of max;  tf.Tensor(3, shape=(), dtype=int64)
Max element:  42
index of min:  7
Min element:  -11
tf.Tensor(
[[  2  11   5]
 [ 42   7  19]
 [ -6 -11  29]], shape=(3, 3), dtype=int32)
indices of max down rows;  [1 0 2]
indices of min down rows ;  [2 2 0]
tf.Tensor(
[[  2  11   5]
 [ 42   7  19]
 [ -6 -11  29]], shape=(3, 3), dtype=int32)
indices of max across cols:  [1 0 2]
indices of min across cols:  [0 1 1]
```

Saving and restoring tensor values using a checkpoint

```
variable = tf.Variable([[1,3,5,7],[11,13,17,19]])
checkpoint= tf.train.Checkpoint(var=variable)
save_path = checkpoint.save('./vars')
variable.assign([[0,0,0,0],[0,0,0,0]])
variable
checkpoint.restore(save_path)
print(variable)
```

```
<tf.Variable 'Variable:0' shape=(2, 4) dtype=int32, numpy=
array([[ 1,  3,  5,  7],
       [11, 13, 17, 19]], dtype=int32)>
```

Using tf.function

tf.function is a function that will take a Python function and return a TensorFlow graph. The advantage of this is that graphs can apply optimizations and exploit parallelism in the Python function (func). tf.function is new to TensorFlow 2.

> Its signature is as follows:

```
tf.function( func=None, input_signature=None, autograph=True, experimental_autograph_options=None )
```

```
def f1(x, y):
    return tf.reduce_mean(input_tensor=tf.multiply(x ** 2, 5) + y**2)

f2 = tf.function(f1)
x = tf.constant([4., -5.])
y = tf.constant([2., 3.])

# f1 and f2 return the same value, but f2 executes as a TensorFlow graph
assert f1(x,y).numpy() == f2(x,y).numpy()
#The assert passes, so there is no output
```

Calculate the gradient

GradientTape

> Another difference from numpy is that it can automatically track the gradient of any variable.

> Open one GradientTape and `tape.watch()` track variables through

```
a = tf.random.normal(shape=(2,2))
b = tf.random.normal(shape=(2,2))

with tf.GradientTape() as tape:
    tape.watch(a)
    c = tf.sqrt(tf.square(a)+tf.square(b))
    dc_da = tape.gradient(c,a)
    print(dc_da)
```

```
tf.Tensor(
[[-0.469929    0.89920384]
 [-0.66446555 -0.7976701 ]], shape=(2, 2), dtype=float32)
```

> For all variables, the calculation is tracked by default and used to find the gradient, so do not `usetape.watch()`

```
a = tf.Variable(a)
with tf.GradientTape() as tape:
    c = tf.sqrt(tf.square(a)+tf.square(b))
    dc_da = tape.gradient(c,a)
    print(dc_da)

     tf.Tensor(
     [[-0.469929    0.89920384]
      [-0.66446555 -0.7976701 ]], shape=(2, 2), dtype=float32)
```

> You can GradientTapefind higher-order derivatives by opening a few more:

```
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as tape:
        c = tf.sqrt(tf.square(a)+tf.square(b))
        dc_da = tape.gradient(c,a)
    d2c_d2a = outer_tape.gradient(dc_da,a)
    print(d2c_d2a)

     tf.Tensor(
     [[0.4264985  0.6867533 ]
      [0.44663432 0.50159544]], shape=(2, 2), dtype=float32)
```

Keras - A High-Level API for TensorFlow 2

TensorFlow

Keras

The Keras Sequential model

To build a Keras Sequential model, you add layers to it in the same order that you want the computations to be undertaken by the network.

After you have built your model, you compile it; this optimizes the computations that are to be undertaken, and is where you allocate the optimizer and the loss function you want your model to use.

The next stage is to fit the model to the data. This is commonly known as training the model, and is where all the computations take place. It is possible to present the data to the model either in batches, or all at once.

Next, you evaluate your model to establish its accuracy, loss, and other metrics. Finally, having trained your model, you can use it to make predictions on new data. So, the workflow is: build, compile, fit, evaluate, make predictions. There are two ways to create a Sequential model. Let's take a look at each of them.

The first way to create a Sequential model

Firstly, you can pass a list of layer instances to the constructor, as in the following example. For now, we will just explain enough to allow you to understand what is happening here.

Acquire the data. MNIST is a dataset of hand-drawn numerals, each on a 28 x 28 pixel grid. Every individual data point is an unsigned 8-bit integer (uint8), as are the labels:

Loading the datset

```
mnist = tf.keras.datasets.mnist
(train_x,train_y), (test_x, test_y) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
```

Definning the variables

```
epochs=10
batch_size = 32 # 32 is default in fit method but specify anyway
```

> Next, normalize all the data points (x) to be in the float range zero to one, and of the float32 type.

> Also, cast the labels (y) to int64, as required:

```
train_x, test_x = tf.cast(train_x/255.0, tf.float32), tf.cast(test_x/255.0, tf.float32)
train_y, test_y = tf.cast(train_y,tf.int64),tf.cast(test_y,tf.int64)
```

Building the Architecture

```
mnistmodel1 = tf.keras.models.Sequential([
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(512,activation=tf.nn.relu),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Dense(10,activation=tf.nn.softmax)
])
```

Compiling the model

```
optimiser = tf.keras.optimizers.Adam()
mnistmodel1.compile (optimizer= optimiser, loss='sparse_categorical_crossentropy', metrics = ['accuracy'])
```

Fitting the model

```
mnistmodel1.fit(train_x, train_y, batch_size=32, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 10s 3ms/step - loss: 0.2206 - accuracy: 0.9344
Epoch 2/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0978 - accuracy: 0.9704
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0709 - accuracy: 0.9771
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0540 - accuracy: 0.9826
Epoch 5/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0439 - accuracy: 0.9860
<keras.callbacks.History at 0x7ffa98fbba30>
```

Evaluate the mnistmodel1

```
mnistmodel1.evaluate(test_x, test_y)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0629 - accuracy: 0.9801
[0.06294650584459305, 0.9800999760627747]
```

> This represents a loss of 0.09 and an accuracy of 0.9801 on the test data.

> An accuracy of 0.98 means that out of 100 test data points, 98 were, on average, correctly identified by the model.

The second way to create a Sequential model The alternative to passing a list of layers to the Sequential model's constructor is to use the add method, as follows, for the same architecture:

Building the Architecture & Compiling

```
mnistmodel2 = tf.keras.models.Sequential();
mnistmodel2.add(tf.keras.layers.Flatten())
mnistmodel2.add(tf.keras.layers.Dense(512, activation='relu'))
mnistmodel2.add(tf.keras.layers.Dropout(0.2))
mnistmodel2.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))
mnistmodel2.compile (optimizer= tf.keras.optimizers.Adam(), loss='sparse_categorical_crossentropy',metrics = ['accuracy'])
```

Fitting the mnistmodel2

```
mnistmodel2.fit(train_x, train_y, batch_size=64, epochs=5)
```

```
Epoch 1/5
938/938 [==============================] - 4s 3ms/step - loss: 0.2436 - accuracy: 0.9301
Epoch 2/5
938/938 [==============================] - 5s 5ms/step - loss: 0.1042 - accuracy: 0.9691
Epoch 3/5
938/938 [==============================] - 5s 6ms/step - loss: 0.0721 - accuracy: 0.9780
Epoch 4/5
938/938 [==============================] - 5s 6ms/step - loss: 0.0536 - accuracy: 0.9831
Epoch 5/5
938/938 [==============================] - 3s 3ms/step - loss: 0.0433 - accuracy: 0.9866
<keras.callbacks.History at 0x7ffa98508e20>
```

Evaluate the mnistmodel2

```
mnistmodel2.evaluate(test_x, test_y)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0640 - accuracy: 0.9790
[0.06400463730096817, 0.9789999723434448]
```

The Keras functional API

The functional API lets you build much more complex architectures than the simple linear stack of Sequential models we have seen previously. It also supports more advanced models. These models include multi-input and multi-output models, models with shared layers, and models with residual connections.

Here is a short example, with an identical architecture to the previous two, of the use of the functional API.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
```

```
(train_x,train_y), (test_x, test_y) = mnist.load_data()

train_x, test_x = train_x/255.0, test_x/255.0

epochs=10
```

Building the Architecture

```
inputs = tf.keras.Input(shape=(28,28)) # Returns a 'placeholder' tensor
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(512, activation='relu',name='d1')(x)
x = tf.keras.layers.Dropout(0.2)(x)
predictions = tf.keras.layers.Dense(10,activation=tf.nn.softmax, name='d2')(x)
mnistmodel3 = tf.keras.Model(inputs=inputs, outputs=predictions)
```

Compile & Fit

```
optimiser = tf.keras.optimizers.Adam()
mnistmodel3.compile (optimizer= optimiser, loss='sparse_categorical_crossentropy', metrics = ['accuracy'])
mnistmodel3.fit(train_x, train_y, batch_size=32, epochs=epochs)
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2217 - accuracy: 0.9344
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0975 - accuracy: 0.9705
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0691 - accuracy: 0.9790
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0534 - accuracy: 0.9832
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0443 - accuracy: 0.9861
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0370 - accuracy: 0.9879
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0302 - accuracy: 0.9897
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0280 - accuracy: 0.9905
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0252 - accuracy: 0.9914
Epoch 10/10
```

```
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0221 - accuracy: 0.9927
<keras.callbacks.History at 0x7ffa98444280>
```

Evaluate the mnistmodel3

```
mnistmodel3.evaluate(test_x, test_y)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0767 - accuracy: 0.9799
[0.0767001211643219, 0.9799000024795532]
```

Subclassing the Keras Model class

```
import tensorflow as tf
```

Building the subclass architecture

```
class MNISTModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MNISTModel, self).__init__()
        # Define your layers here.
        inputs = tf.keras.Input(shape=(28,28)) # Returns a placeholder tensor
        self.x0 = tf.keras.layers.Flatten()
        self.x1 = tf.keras.layers.Dense(512, activation='relu',name='d1')
        self.x2 = tf.keras.layers.Dropout(0.2)
        self.predictions = tf.keras.layers.Dense(10,activation=tf.nn.softmax, name='d2')


    def call(self, inputs):
    # This is where to define your forward pass
    # using the layers previously defined in `__init__`
        x = self.x0(inputs)
        x = self.x1(x)
        x = self.x2(x)
        return self.predictions(x)
```

```
mnistmodel4 = MNISTModel()
```

Compile & Fit

```
batch_size = 32
steps_per_epoch = len(train_x)//batch_size
print(steps_per_epoch)
mnistmodel4.compile (optimizer= tf.keras.optimizers.Adam(), loss='sparse_categorical_crossentropy',metrics = ['accuracy'])
mnistmodel4.fit(train_x, train_y, batch_size=batch_size, epochs=epochs)
```

```
1875
Epoch 1/10
1875/1875 [==============================] - 9s 4ms/step - loss: 0.2212 - accuracy: 0.9348
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0962 - accuracy: 0.9702
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0698 - accuracy: 0.9778
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0521 - accuracy: 0.9830
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0418 - accuracy: 0.9865
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0354 - accuracy: 0.9882
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0299 - accuracy: 0.9903
```