

TensorFlow

▼ Lab 1: Intro to TensorFlow

In this lab, you'll get exposure to using TensorFlow and learn how it can be used for solving deep learning tasks. Go through the code and run each cell. Along the way, you'll encounter several **TODO** blocks -- follow the instructions to fill them out before running those cells and continuing.

Part 1: Intro to TensorFlow

0.1 Install TensorFlow

TensorFlow is a software library extensively used in machine learning. Here we'll learn how computations are represented and how to define a simple neural network in TensorFlow. For all the labs in 6.S191 2021, we'll be using the latest version of TensorFlow, TensorFlow 2, which affords great flexibility and the ability to imperatively execute operations, just like in Python. You'll notice that TensorFlow 2 is quite similar to Python in its syntax and imperative execution. Let's install TensorFlow and a couple of dependencies.

```
1
2 import tensorflow as tf
3
4 # Download and import the MIT 6.S191 package
5 !pip install mitdeeplearning
6 import mitdeeplearning as mdl
7
8 import numpy as np
9 import matplotlib.pyplot as plt
```

```
Collecting mitdeeplearning
  Downloading mitdeeplearning-0.2.0.tar.gz (2.1 MB)
Requirement already satisfied: numpy in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Collecting regex
  Downloading regex-2021.4.4-cp37-cp37m-win_amd64.whl (269 kB)
Requirement already satisfied: tqdm in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Collecting gym
  Downloading gym-0.18.0.tar.gz (1.6 MB)
Requirement already satisfied: scipy in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Collecting pygame<=1.5.0,>=1.4.0
  Downloading pygame-1.5.0-py2.py3-none-any.whl (1.0 MB)
Requirement already satisfied: Pillow<=7.2.0 in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Requirement already satisfied: future in c:\users\dell\.conda\envs\tensorflow\lib\site-packages
Building wheels for collected packages: mitdeeplearning, gym
```

```

Building wheel for mitdeeplearning (setup.py): started
Building wheel for mitdeeplearning (setup.py): finished with status 'done'
Created wheel for mitdeeplearning: filename=mitdeeplearning-0.2.0-py3-none-any.whl size=
Stored in directory: c:\users\dell\appdata\local\pip\cache\wheels\9a\b9\4f\99b7c8c5c7f
Building wheel for gym (setup.py): started
Building wheel for gym (setup.py): finished with status 'done'
Created wheel for gym: filename=gym-0.18.0-py3-none-any.whl size=1656453 sha256=bc98b6
Stored in directory: c:\users\dell\appdata\local\pip\cache\wheels\99\f7\e3\d6f0f120ac6
Successfully built mitdeeplearning gym
Installing collected packages: regex, pyglet, gym, mitdeeplearning
Successfully installed gym-0.18.0 mitdeeplearning-0.2.0 pyglet-1.5.0 regex-2021.4.4

```

▼ 1.1 Why is TensorFlow called TensorFlow?

TensorFlow is called 'TensorFlow' because it handles the flow (node/mathematical operation) of Tensors, which are data structures that you can think of as multi-dimensional arrays. Tensors are represented as n-dimensional arrays of base datatypes such as a string or integer -- they provide a way to generalize vectors and matrices to higher dimensions.

The `shape` of a Tensor defines its number of dimensions and the size of each dimension. The `rank` of a Tensor provides the number of dimensions (n-dimensions) -- you can also think of this as the Tensor's order or degree.

Let's first look at 0-d Tensors, of which a scalar is an example:

```

1 sport = tf.constant("Tennis", tf.string)
2 number = tf.constant(1.41421356237, tf.float64)
3
4 print("`sport` is a {}-d Tensor".format(tf.rank(sport).numpy()))
5 print("`number` is a {}-d Tensor".format(tf.rank(number).numpy()))

`sport` is a 0-d Tensor
`number` is a 0-d Tensor

```

Vectors and lists can be used to create 1-d Tensors:

```

1 sports = tf.constant(["Tennis", "Basketball"], tf.string)
2 numbers = tf.constant([3.141592, 1.414213, 2.71821], tf.float64)
3
4 print("`sports` is a {}-d Tensor with shape: {}".format(tf.rank(sports).numpy(), tf.
5 print("`numbers` is a {}-d Tensor with shape: {}".format(tf.rank(numbers).numpy(), t

`sports` is a 1-d Tensor with shape: [2]
`numbers` is a 1-d Tensor with shape: [3]

```

Next we consider creating 2-d (i.e., matrices) and higher-rank Tensors. For examples, in future labs involving image processing and computer vision, we will use 4-d Tensors. Here the dimensions correspond to the number of example images in our batch, image height, image width, and the number of color channels.

```
1 ### Defining higher-order Tensors ###
2
3 '''TODO: Define a 2-d Tensor'''
4 matrix = tf.constant([[3.141592, 1.414213, 2.71821],[3.141592, 1.414213, 2.71821]],
5
6 assert isinstance(matrix, tf.Tensor), "matrix must be a tf Tensor object"
7 assert tf.rank(matrix).numpy() == 2
8 print("`matrix` is a {}-d Tensor with shape: {}".format(tf.rank(matrix).numpy(), tf.
    `matrix` is a 2-d Tensor with shape: [2 3]
```

```
1 '''TODO: Define a 4-d Tensor.'''
2 # Use tf.zeros to initialize a 4-d Tensor of zeros with size 10 x 256 x 256 x 3.
3 # You can think of this as 10 images where each image is RGB 256 x 256.
4 images = tf.zeros(shape=(10,256,256,3))
5
6 assert isinstance(images, tf.Tensor), "matrix must be a tf Tensor object"
7 assert tf.rank(images).numpy() == 4, "matrix must be of rank 4"
8 assert tf.shape(images).numpy().tolist() == [10, 256, 256, 3], "matrix is incorrect
```

As you have seen, the `shape` of a Tensor provides the number of elements in each Tensor dimension. The `shape` is quite useful, and we'll use it often. You can also use slicing to access subtensors within a higher-rank Tensor:

```
1 type(matrix)

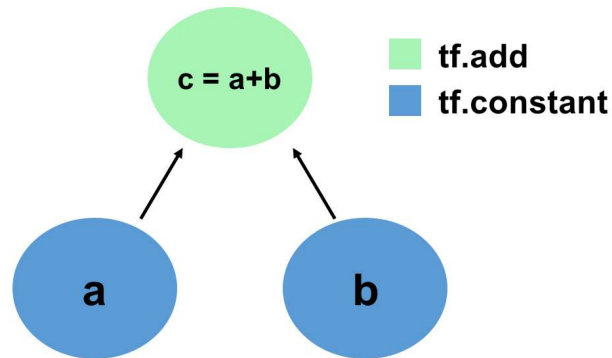
tensorflow.python.framework.ops.EagerTensor

1 row_vector = matrix[0]
2 column_vector = matrix[:,2]
3 scalar = matrix[1, 2]
4
5 print("`row_vector`: {}".format(row_vector.numpy()))
6 print("`column_vector`: {}".format(column_vector.numpy()))
7 print("`scalar`: {}".format(scalar.numpy()))

`row_vector`: [3.141592 1.414213 2.71821 ]
`column_vector`: [2.71821 2.71821]
`scalar`: 2.71821
```

▼ 1.2 Computations on Tensors

A convenient way to think about and visualize computations in TensorFlow is in terms of graphs. We can define this graph in terms of Tensors, which hold data, and the mathematical operations that act on these Tensors in some order. Let's look at a simple example, and define this computation using TensorFlow:




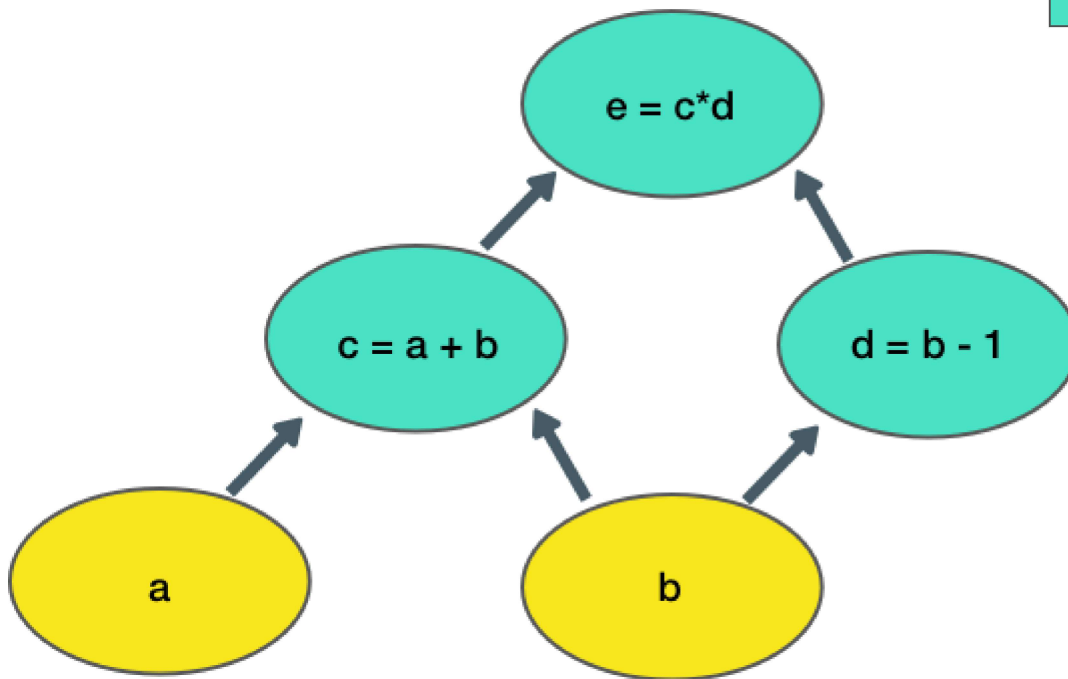
```
1 # Create the nodes in the graph, and initialize values
2 a = tf.constant(15)
3 b = tf.constant(61)
4
5 # Add them!
6 c1 = tf.add(a,b)
7 c2 = a + b # TensorFlow overrides the "+" operation so that it is able to act on Ten
8 print(c1)
9 print(c2)
```

`tf.Tensor(76, shape=(), dtype=int32)`
`tf.Tensor(76, shape=(), dtype=int32)`

Notice how we've created a computation graph consisting of TensorFlow operations, and how the output is a Tensor with value 76 -- we've just created a computation graph consisting of operations, and it's executed them and given us back the result.

Now let's consider a slightly more complicated example:

 math operation



Here, we take two inputs, a , b , and compute an output e . Each node in the graph represents an operation that takes some input, does some computation, and passes its output to another node.

Let's define a simple function in TensorFlow to construct this computation function:

```
1 ### Defining Tensor computations ###
2
3 # Construct a simple computation function
4 def func(a,b):
5     '''TODO: Define the operation for c, d, e (use tf.add, tf.subtract, tf.multiply).'''
6     c = a+b
7     d = b-1
8     e = c*d
9     return e
```

Now, we can call this function to execute the computation graph given some inputs a, b :

```
1 # Consider example values for a,b
2 a, b = 1.5, 2.5
3 # Execute the computation
4 e_out = func(a,b)
5 print(e_out)
```

6.0

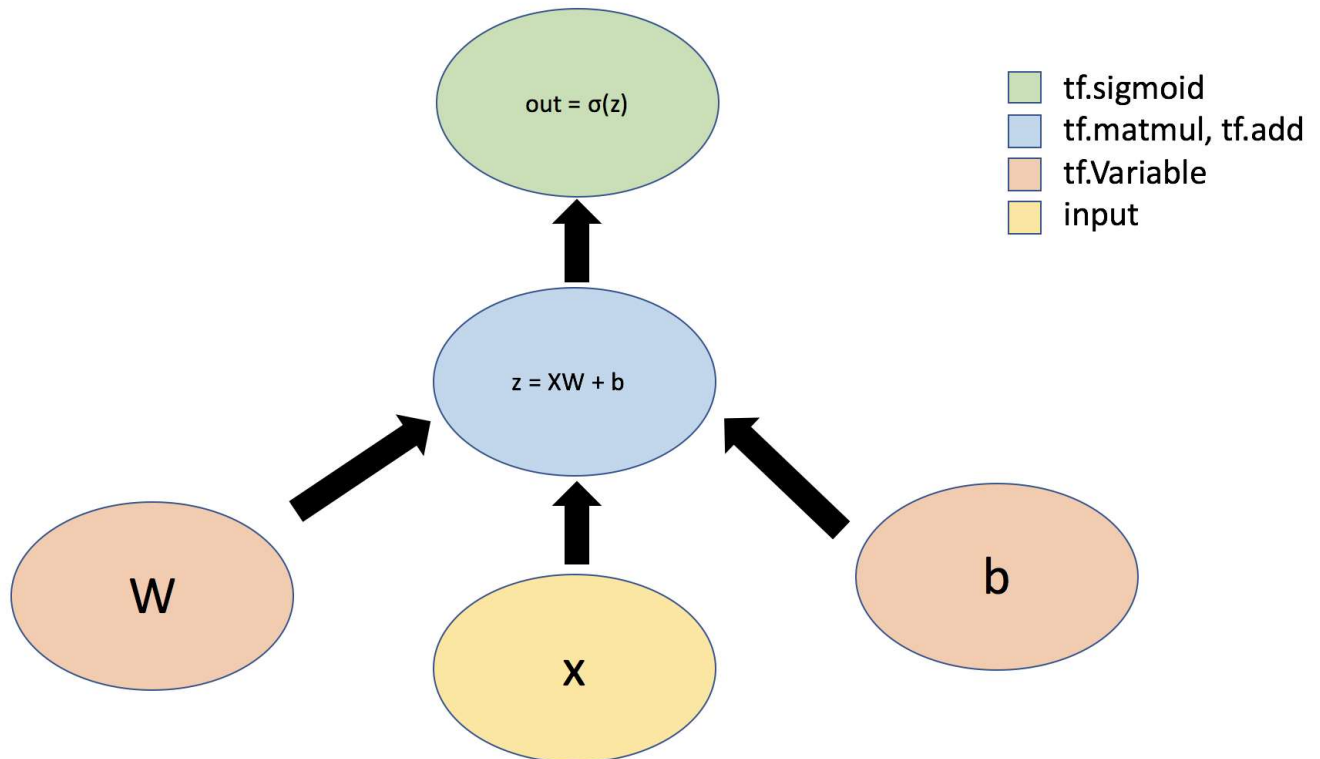
Notice how our output is a Tensor with value defined by the output of the computation, and that the output has no shape as it is a single scalar value.

▼ 1.3 Neural networks in TensorFlow

We can also define neural networks in TensorFlow. TensorFlow uses a high-level API called [Keras](#) that provides a powerful, intuitive framework for building and training deep learning models.

Let's first consider the example of a simple perceptron defined by just one dense layer:

$y = \sigma(Wx + b)$, where W represents a matrix of weights, b is a bias, x is the input, σ is the sigmoid activation function, and y is the output. We can also visualize this operation using a graph:



Tensors can flow through abstract types called [Layers](#) -- the building blocks of neural networks.

Layers implement common neural networks operations, and are used to update weights, compute losses, and define inter-layer connectivity. We will first define a Layer to implement the simple perceptron defined above.

```
1 ### Defining a network Layer ###
2
3 # n_output_nodes: number of output nodes
4 # input_shape: shape of the input
5 # x: input to the layer
6
7 class OurDenseLayer(tf.keras.layers.Layer):
8     def __init__(self, n_output_nodes):
9         super(OurDenseLayer, self).__init__()
10        self.n_output_nodes = n_output_nodes
11
```

```

--
12 def build(self, input_shape): #      a=[1,2,3,4,5] --> a[-1]==5
13     d = int(input_shape[-1])
14     # Define and initialize parameters: a weight matrix W and bias b
15     # Note that parameter initialization is random!
16     self.W = self.add_weight("weight", shape=[d, self.n_output_nodes]) # note the di
17     self.b = self.add_weight("bias", shape=[1, self.n_output_nodes]) # note the dime
18
19 def call(self, x):
20     '''TODO: define the operation for z (hint: use tf.matmul)'''
21     z = tf.add(tf.matmul(x,self.W),self.b)
22
23     '''TODO: define the operation for out (hint: use tf.sigmoid)'''
24     y = tf.sigmoid(z)
25     return y
26
27 # Since layer parameters are initialized randomly, we will set a random seed for rep
28 tf.random.set_seed(1)
29 layer = OurDenseLayer(3)
30 layer.build((1,2))
31 x_input = tf.constant([[1,2.]], shape=(1,2))
32 y = layer.call(x_input)
33
34 # test the output!
35 print(y.numpy())
36 mdl.lab1.test_custom_dense_layer_output(y)

[[0.2697859  0.45750418 0.66536945]]
[PASS] test_custom_dense_layer_output
True

```

Conveniently, TensorFlow has defined a number of `Layers` that are commonly used in neural networks, for example a `Dense`. Now, instead of using a single `Layer` to define our simple neural network, we'll use the `Sequential` model from Keras and a single `Dense` layer to define our network. With the `Sequential` API, you can readily create neural networks by stacking together layers like building blocks.

```

1 ### Defining a neural network using the Sequential API ###
2
3 # Import relevant packages
4 from tensorflow.keras import Sequential
5 from tensorflow.keras.layers import Dense
6
7 # Define the number of outputs
8 n_output_nodes = 3
9
10 # First define the model
11 model = Sequential()

```

```

12
13 '''TODO: Define a dense (fully connected) layer to compute z'''
14 # Remember: dense layers are defined by the parameters W and b!
15 # You can read more about the initialization of W and b in the TF documentation :)
16 # https://www.tensorflow.org/api\_docs/python/tf/keras/layers/Dense?version=stable
17 dense_layer = Dense(n_output_nodes, activation='sigmoid')
18
19 # Add the dense layer to the model
20 model.add(dense_layer)
21

```

That's it! We've defined our model using the Sequential API. Now, we can test it out using an example input:

```

1 # Test model with example input
2 x_input = tf.constant([[1,2.]], shape=(1,2))
3
4 '''TODO: feed input into the model and predict the output!'''
5 model_output = model.predict(x_input)
6 print(model_output)

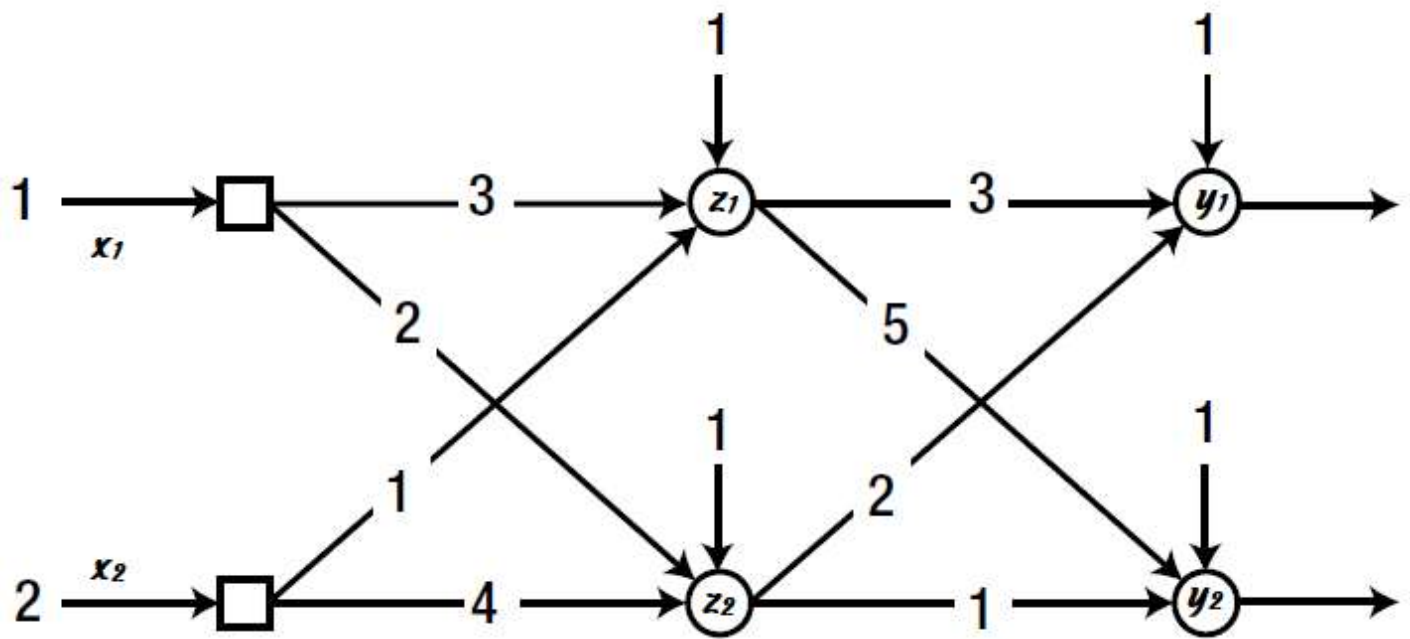
[[0.5607363 0.6566898 0.1249697]]

```

Now that we have learned how to define `Layers` as well as neural networks in TensorFlow using the `Sequential` APIs, we're ready to turn our attention to actually implement neural networks.

▼ Labwork

Implement the Neural Network created in lab# 1 using TensorFlow library



```

1 model=Sequential()
2 model.add(Dense(2, activation='relu', input_shape=(2,1)))
3 model.add(Dense(2, activation='relu'))
4 model.add(Dense(2, activation='sigmoid'))

```

1