Deep Learning and TensorFlow

Part 1: Intro to TensorFlow

The following concent is based on MIT 6.S191 Lab1 Part1. See © MIT 6.S191: Introduction to Deep Learning.

1.0 Install TensorFlow

TensorFlow is a software library extensively used in machine learning. 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. Let's install TensorFlow 2, and a couple of dependencies.

```
%tensorflow_version 2.x
import tensorflow as tf

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

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 dataypes* 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:

```
sport = tf.constant("Tennis", tf.string)
number = tf.constant(1.41421356237, tf.float64)

print("`sport` is a {}-d Tensor".format(tf.rank(sport).numpy()))
print("`number` is a {}-d Tensor".format(tf.rank(number).numpy()))
```

The output is:

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

Vectors and lists can be used to create 1-d and 2-d Tensors:

```
sports = tf.constant(["Tennis", "Basketball"], tf.string)
numbers = tf.constant([[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]])

print("`sports` is a {}-d Tensor with shape: {}".format(tf.rank(sports).numpy(), tf.shape(sports)))
print("`numbers` is a {}-d Tensor with shape: {}".format(tf.rank(numbers).numpy(), tf.shape(numbers)))
```

The output is:

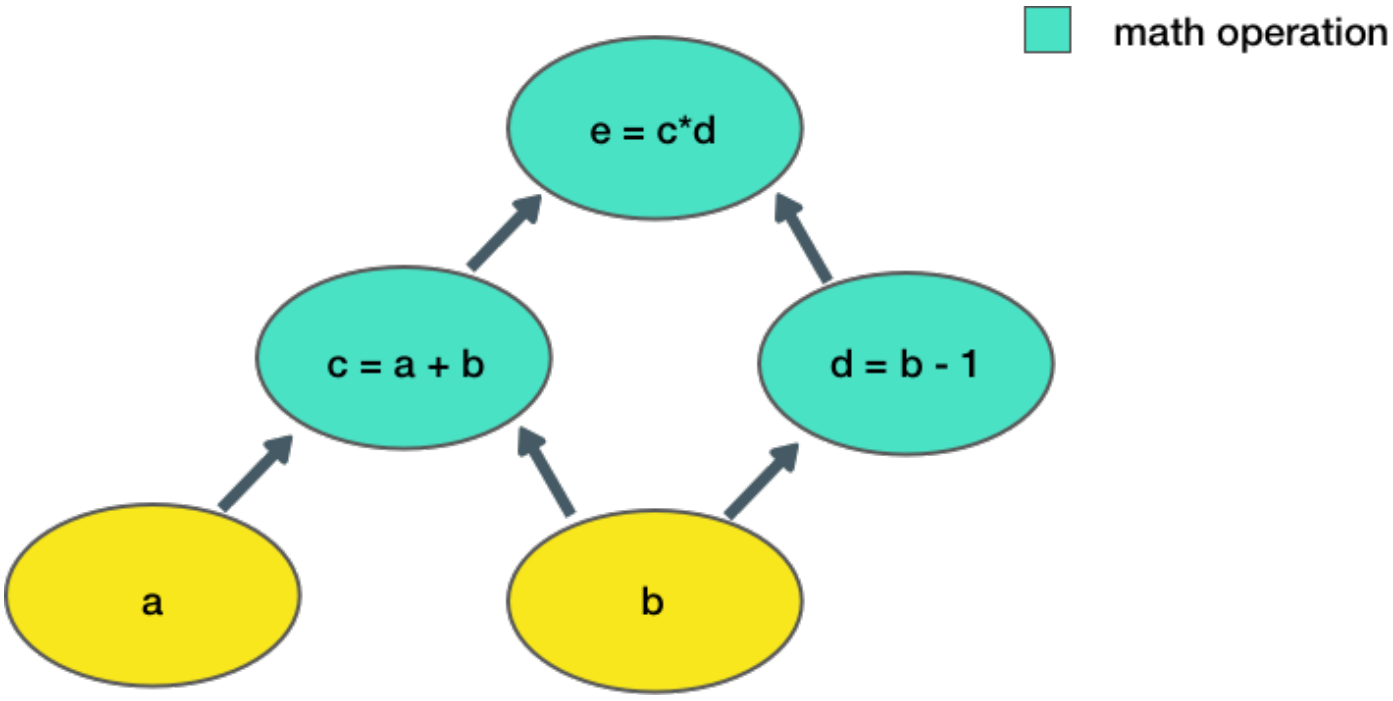
> `sports` is a 1-d Tensor with shape: [2]
>
> `numbers` is a a 2-d Tensor with shape: [2 4]

The next line creates a 4-d Tensor using `tf.zeros`:

```
images = tf.zeros([10, 256, 256, 3])
```

1.2 Computations on Tensors

Now let's consider an example:



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:

```
# Construct a simple computation function
def func(a,b):
  c = tf.add(a, b)
  d = tf.subtract(b, 1)
  e = tf.multiply(c, d)
  return e
```

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

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

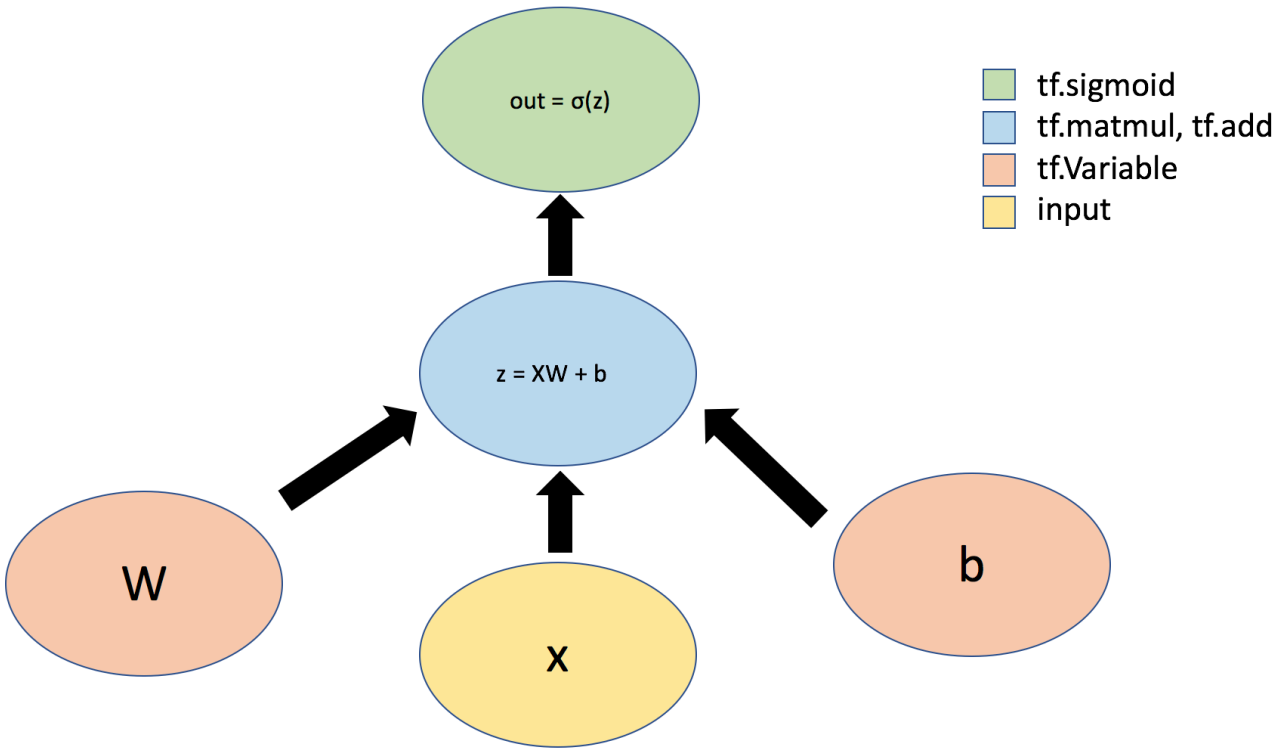
> tf.Tensor(6.0, shape=(), dtype=float32)

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:



1.3.1 A network Layer

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. (For a tutorial on Classes in Python, click here)

```
### Defining a network Layer ###

# n_output_nodes: number of output nodes
# input_shape: shape of the input
# x: input to the layer

class OurDenseLayer(tf.keras.layers.Layer):
  def __init__(self, n_output_nodes):
    super(OurDenseLayer, self).__init__()
    self.n_output_nodes = n_output_nodes

  def build(self, input_shape):
    d = int(input_shape[-1])
    # Define and initialize parameters: a weight matrix W and bias b
    # Note that parameter initialization is random!
    self.W = self.add_weight("weight", shape=[d, self.n_output_nodes]) # note the dimensionality
    self.b = self.add_weight("bias", shape=[1, self.n_output_nodes]) # note the dimensionality

  def call(self, x):
    # Define the operation for z
    z = tf.matmul(x, self.W) + self.b
    # Define the operation for out
    y = tf.sigmoid(z)
    return y

# Since layer parameters are initialized randomly, we will set a random seed for reproducibility
tf.random.set_seed(1)
layer = OurDenseLayer(3)
layer.build((1,2))
x_input = tf.constant([[1,2.]], shape=(1,2))
y = layer.call(x_input)

# print the output
print(y.numpy())
```

```
[[0.2697859  0.45750412 0.66536945]]
```

1.3.2 Using the Sequential API

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.

```
### Defining a neural network using the Sequential API ###

# Import relevant packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Define the number of outputs
n_output_nodes = 3

# First define the model
model = Sequential()

# Define a dense (fully connected) layer to compute z
# Remember: dense layers are defined by the parameters W and b!
# You can read more about the initialization of W and b in the TF documentation :)
# https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense?version=stable
dense_layer = Dense(n_output_nodes, activation='sigmoid')

# Add the dense layer to the model
model.add(dense_layer)
```

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

```
# Test model with example input
x_input = tf.constant([[1,2.]], shape=(1,2))

# feed input into the model and predict the output
model_output = model(x_input).numpy()
print(model_output)
```

```
[[0.5607363 0.6566898 0.1249697]]
```

1.3.3 Subclassing the Model class

In addition to defining models using the `Sequential` API, we can also define neural networks by directly *subclassing* the `Model` class, which groups layers together to enable model training and inference. The `Model` class captures what we refer to as a "model" or as a "network". Using Subclassing, we can create a class for our model, and then define the *forward pass* through the network using the `call` function.

Let's define the same neural network as above now using Subclassing rather than the `Sequential` model.

```
### Defining a model using subclassing ###

from tensorflow.keras import Model
from tensorflow.keras.layers import Dense

class SubclassModel(tf.keras.Model):

  # In __init__, we define the Model's layers
  def __init__(self, n_output_nodes):
    super(SubclassModel, self).__init__()

    self.dense_layer = Dense(n_output_nodes, activation='sigmoid')

  # In the call function, we define the Model's forward pass.
  def call(self, inputs):
    return self.dense_layer(inputs)
```

Just like the model we built using the `Sequential` API, let's test out our `SubclassModel` using an example input.

```
n_output_nodes = 3
model = SubclassModel(n_output_nodes)

x_input = tf.constant([[1,2.]], shape=(1,2))

print(model.call(x_input))
```

```
tf.Tensor([[0.6504887  0.47828162 0.8373661 ]], shape=(1, 3), dtype=float32)
```

Importantly, Subclassing affords us a lot of flexibility to define custom layers, custom training loops, custom activation functions and custom models. For example, we can use boolean arguments in the `call` function to specify different network behaviors, for example different behaviors during training and inference. Let's suppose under some instances we want our network to simply output the input, without any perturbation. We define a boolean argument `isidentity` to control this behavior:

```
### Defining a model using subclassing and specifying custom behavior ###

class IdentityModel(tf.keras.Model):

  # As before, in __init__ we define the Model's layers
  # Since our desired behavior involves the forward pass, this part is unchanged
  def __init__(self, n_output_nodes):
    super(IdentityModel, self).__init__()
    self.dense_layer = tf.keras.layers.Dense(n_output_nodes, activation='sigmoid')

  # Implement the behavior where the network outputs the input, unchanged, under
control of the isidentity argument
  def call(self, inputs, isidentity=False):
```

```
    x = self.dense_layer(inputs)
    if isidentity:
      return inputs
    return x

# Let's test this behavior:
n_output_nodes = 3
model = IdentityModel(n_output_nodes)

x_input = tf.constant([[1,2.]], shape=(1,2))
# pass the input into the model and call with and without the input identity option
out_activate = model.call(x_input)

out_identity = model.call(x_input, isidentity=True)

print("Network output with activation: {}; network identity output
{}".format(out_activate.numpy(), out_identity.numpy()))
```

> Network output with activation: [[0.29996255 0.62776643 0.48460066]]; network identity output: [[1. 2.]]

1.4 Automatic differentiation in TensorFlow

Automatic differentiation is one of the most important parts of TensorFlow and is the backbone of training with backpropagation. We will use the TensorFlow GradientTape `tf.GradientTape` to trace operations for computing gradients later.

When a forward pass is made through the network, all forward-pass operations get recorded to a "tape"; then, to compute the gradient, the tape is played backwards. By default, the tape is discarded after it is played backwards; this means that a particular `tf.GradientTape` can only compute one gradient, and subsequent calls throw a runtime error. However, we can compute multiple gradients over the same computation by creating a `persistent` gradient tape.

First, we will look at how we can compute gradients using GradientTape and access them for computation. We define the simple function $y = x^2$ and compute the gradient:

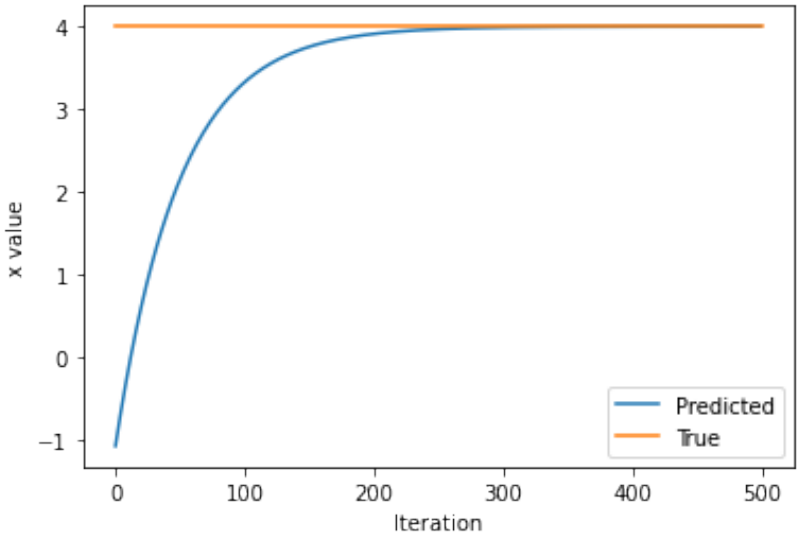
```
### Gradient computation with GradientTape ###

# y = x^2
# Example: x = 3.0
x = tf.Variable(3.0) # Notice that x is a tf.Variable

# Initiate the gradient tape
with tf.GradientTape() as tape:
  y = x * x

# Access the gradient -- derivative of y with respect to x
dy_dx = tape.gradient(y, x)

```

```
assert dy_dx.numpy() == 6.0
```

In training neural networks, we use differentiation and stochastic gradient descent (SGD) to optimize a loss function. Now that we have a sense of how `GradientTape` can be used to compute and access derivatives, we will look at an example where we use automatic differentiation and SGD to find the minimum of $L = (x - x_f)^2$. Here x_f is a variable for a desired value we are trying to optimize for; L represents a loss that we are trying to minimize. While we can clearly solve this problem analytically ($x_{min} = x_f$), considering how we can compute this using `GradientTape` sets us up nicely for future labs where we use gradient descent to optimize entire neural network losses.

```
### Function minimization with automatic differentiation and SGD ###

# Initialize a random value for our initial x
x = tf.Variable([tf.random.normal([1])])
print("Initializing x={}".format(x.numpy()))

learning_rate = 1e-2 # learning rate for SGD
history = []
# Define the target value
x_f = 4

# We will run SGD for a number of iterations. At each iteration, we compute the loss,
#  compute the derivative of the loss with respect to x, and perform the SGD update.
for i in range(500):
  with tf.GradientTape() as tape:
    loss = (x - x_f)**2 # "forward pass": record the current loss on the tape

  # loss minimization using gradient tape
  grad = tape.gradient(loss, x) # compute the derivative of the loss with respect to x
  new_x = x - learning_rate * grad # sgd update
  x.assign(new_x) # update the value of x
  history.append(x.numpy()[0])

# Plot the evolution of x as we optimize towards x_f!
plt.plot(history)
plt.plot([0, 500],[x_f,x_f])
plt.legend(('Predicted', 'True'))
plt.xlabel('Iteration')
plt.ylabel('x value')
```

```
Initializing x=[[-1.1771784]]
Text(0, 0.5, 'x value')
```



`GradientTape` provides an extremely flexible framework for automatic differentiation. In order to back propagate errors through a neural network, we track forward passes on the Tape, use this information to determine the gradients, and then use these gradients for optimization using SGD.