Writing Low-Level TensorFlow Code

Learning Objectives

1. Practice defining and performing basic operations on constant Tensors
2. Use Tensorflow's automatic differentiation capability
3. Learn how to train a linear regression from scratch with TensorFLow

Introduction

In this notebook, we will start by reviewing the main operations on Tensors in TensorFlow and understand how to manipulate TensorFlow Variables. We explain how these are compatible with python built-in list and numpy arrays.

Then we will jump to the problem of training a linear regression from scratch with gradient descent. The first order of business will be to understand how to compute the gradients of a function (the loss here) with respect to some of its arguments (the model weights here). The TensorFlow construct allowing us to do that is `tf.GradientTape`, which we will describe.

At last we will create a simple training loop to learn the weights of a 1-dim linear regression using synthetic data generated from a linear model.

As a bonus exercise, we will do the same for data generated from a non linear model, forcing us to manual engineer non-linear features to improve our linear model performance.

Each learning objective will correspond to a #TODO in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

In [ ]:
```
# Here we'll import data processing libraries like Numpy and Tensorflow
import numpy as np
import tensorflow as tf
# Use matplotlib for visualizing the model
from matplotlib import pyplot as plt
```

In [ ]:
```
# Here we'll show the currently installed version of TensorFlow
print(tf.__version__)
```

```
2.3.0
```

Operations on Tensors

Variables and Constants

Tensors in TensorFlow are either constant (`tf.constant`) or variables (`tf.Variable`). Constant values can not be changed, while variables values can be.

The main difference is that instances of `tf.Variable` have methods allowing us to change their values while tensors constructed with `tf.constant` don't have these methods, and therefore their values can not be changed. When you want to change the value of a `tf.Variable` `x` use one of the following method:

- `x.assign(new_value)`
- `x.assign_add(value_to_be_added)`
- `x.assign_sub(value_to_be_subtracted`

In [ ]:
```
# Creates a constant tensor from a tensor-like object.
x = tf.constant([2, 3, 4])
x
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([2, 3, 4], dtype=int32)>
```

In [ ]:
```
# The Variable() constructor requires an initial value for the variable, which can be a
x = tf.Variable(2.0, dtype=tf.float32, name='my_variable')
```

In [ ]:
```
# The .assign() method will assign the value to referance object.
x.assign(45.8)
x
```

```
<tf.Variable 'my_variable:0' shape=() dtype=float32, numpy=45.8>
```

In [ ]:
```
# The .assign_add() method will update the referance object by adding value to it.
x.assign_add(4)
x
```

```
<tf.Variable 'my_variable:0' shape=() dtype=float32, numpy=49.8>
```

In [ ]:
```
# The .assign_add() method will update the referance object by subtracting value to it.
x.assign_sub(3)
x
```

```
<tf.Variable 'my_variable:0' shape=() dtype=float32, numpy=46.8>
```

Point-wise operations

Tensorflow offers similar point-wise tensor operations as numpy does:

- `tf.add` allows to add the components of a tensor
- `tf.multiply` allows us to multiply the components of a tensor
- `tf.subtract` allow us to substract the components of a tensor
- `tf.math.*` contains the usual math operations to be applied on the components of a tensor
- and many more...

Most of the standard arithmetic operations (`tf.add` , `tf.substrac` , etc.) are overloaded by the usual corresponding arithmetic symbols (`+` , `-` , etc.)

In [ ]:
```
# Creates a constant tensor from a tensor-like object.
a = tf.constant([5, 3, 8]) # TODO 1a
```

```
b = tf.constant([3, -1, 2])
# Using the .add() method components of a tensor will be added.
c = tf.add(a, b)
d = a + b

# Let's output the value of `c` and `d`.
print("c:", c)
print("d:", d)
```

```
# Here tf.Tensor() represents a multidimensional array of elements.
c: tf.Tensor([ 8  2 10], shape=(3,), dtype=int32)
d: tf.Tensor([ 8  2 10], shape=(3,), dtype=int32)
```

In [ ]:

```
# Creates a constant tensor from a tensor-like object.
a = tf.constant([5, 3, 8]) # TODO 1b
b = tf.constant([3, -1, 2])
# Using the .multiply() method components of a tensor will be multiplied.
c = tf.multiply(a, b)
d = a * b

# Let's output the value of `c` and `d`.
print("c:", c)
print("d:", d)
```

```
c: tf.Tensor([15 -3 16], shape=(3,), dtype=int32)
d: tf.Tensor([15 -3 16], shape=(3,), dtype=int32)
```

In [ ]:

```
# TODO 1c
# tf.math.exp expects floats so we need to explicitly give the type
a = tf.constant([5, 3, 8], dtype=tf.float32)
b = tf.math.exp(a)

# Let's output the value of `b`.
print("b:", b)
```

```
b: tf.Tensor([ 148.41316    20.085537 2980.958   ], shape=(3,), dtype=float32)
```

NumPy Interoperability

In addition to native TF tensors, tensorflow operations can take native python types and NumPy arrays as operands.

In [ ]:

```
# native python list
a_py = [1, 2]
b_py = [3, 4]
```

In [ ]:

```
# Using the .add() method components of a tensor will be added.
tf.add(a_py, b_py)
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([4, 6], dtype=int32)>
```

In [ ]:

```
# numpy arrays
a_np = np.array([1, 2])
b_np = np.array([3, 4])
```

In [ ]:
```
# Using the .add() method components of a tensor will be added.
tf.add(a_np, b_np)
```

```
<tf.Tensor: shape=(2,), dtype=int64, numpy=array([4, 6])>
```

In [ ]:
```
# native TF tensor
a_tf = tf.constant([1, 2])
b_tf = tf.constant([3, 4])
```

In [ ]:
```
# Using the .add() method components of a tensor will be added.
tf.add(a_tf, b_tf)
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([4, 6], dtype=int32)>
```

In [ ]:
```
# Here using the .numpy() method we'll convert a `native TF tensor` to a `NumPy array`.
a_tf.numpy()
```

```
array([1, 2], dtype=int32)
```

Linear Regression

Now let's use low level tensorflow operations to implement linear regression.

Later in the course you'll see abstracted ways to do this using high level TensorFlow.

Toy Dataset

We'll model the following function:

\begin{equation} y= 2x + 10 \end{equation}

In [ ]:
```
# Creates a constant tensor from a tensor-like object.
X = tf.constant(range(10), dtype=tf.float32)
Y = 2 * X + 10

# Let's output the value of `X` and `Y`.
print("X:{}".format(X))
print("Y:{}".format(Y))
```

```
X:[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Y:[10. 12. 14. 16. 18. 20. 22. 24. 26. 28.]
```

Let's also create a test dataset to evaluate our models:

In [ ]:
```
# Creates a constant tensor from a tensor-like object.
X_test = tf.constant(range(10, 20), dtype=tf.float32)
Y_test = 2 * X_test + 10

# Let's output the value of `X_test` and `Y_test`.
print("X_test:{}".format(X_test))
print("Y_test:{}".format(Y_test))
```

```
X_test:[10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
Y_test:[30. 32. 34. 36. 38. 40. 42. 44. 46. 48.]
```

Loss Function

The simplest model we can build is a model that for each value of x returns the sample mean of the training set:

In [ ]:
```
# The numpy().mean() will compute the arithmetic mean or average of the given data (arr
y_mean = Y.numpy().mean()


# Let's define predict_mean() function.
def predict_mean(X):
    y_hat = [y_mean] * len(X)
    return y_hat

Y_hat = predict_mean(X_test)
```

Using mean squared error, our loss is: \begin{equation} MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{Y}_i-Y_i)^2 \end{equation}

For this simple model the loss is then:

In [ ]:
```
# Let's evaluate the loss.
errors = (Y_hat - Y)**2
loss = tf.reduce_mean(errors)
loss.numpy()
```

```
33.0
```

This value for the MSE loss above will give us a baseline to compare how a more complex model is doing.

Now, if $\hat{Y}$ represents the vector containing our model's predictions when we use a linear regression model \begin{equation} \hat{Y} = w_0X + w_1 \end{equation}

In [ ]:
```
# Let's define loss_mse() function which is taking arguments as coefficients of the mod
def loss_mse(X, Y, w0, w1):
    Y_hat = w0 * X + w1
    errors = (Y_hat - Y)**2
    return tf.reduce_mean(errors)
```

Gradient Function

To use gradient descent we need to take the partial derivatives of the loss function with respect to each of the weights. We could manually compute the derivatives, but with Tensorflow's automatic differentiation capabilities we don't have to!

During gradient descent we think of the loss as a function of the parameters w_0 and w_1. Thus, we want to compute the partial derivative with respect to these variables.

For that we need to wrap our loss computation within the context of `tf.GradientTape` instance which will record gradient information:

```
with tf.GradientTape() as tape:
    loss = # computation
```

This will allow us to later compute the gradients of any tensor computed within the `tf.GradientTape` context with respect to instances of `tf.Variable`:

```
gradients = tape.gradient(loss, [w0, w1])
```

In [ ]:
```
# Let's define compute_gradients() procedure for computing the loss gradients with resp
# TODO 2
def compute_gradients(X, Y, w0, w1):
    with tf.GradientTape() as tape:
        loss = loss_mse(X, Y, w0, w1)
    return tape.gradient(loss, [w0, w1])
```

In [ ]:
```
# The Variable() constructor requires an initial value for the variable, which can be a
w0 = tf.Variable(0.0)
w1 = tf.Variable(0.0)

dw0, dw1 = compute_gradients(X, Y, w0, w1)
```

In [ ]:
```
# Let's output the value of `dw0`.
print("dw0:", dw0.numpy())
```

```
dw0 -204.0
```

In [ ]:
```
# Let's output the value of `dw1`.
print("dw1", dw1.numpy())
```

```
dw1 -38.0
```

Training Loop

Here we have a very simple training loop that converges. Note we are ignoring best practices like batching, creating a separate test set, and random weight initialization for the sake of simplicity.

In [ ]:
```
# TODO 3
STEPS = 1000
LEARNING_RATE = .02
MSG = "STEP {step} - loss: {loss}, w0: {w0}, w1: {w1}\n"


# The Variable() constructor requires an initial value for the variable, which can be a
w0 = tf.Variable(0.0)
w1 = tf.Variable(0.0)


for step in range(0, STEPS + 1):

    dw0, dw1 = compute_gradients(X, Y, w0, w1)
    w0.assign_sub(dw0 * LEARNING_RATE)
    w1.assign_sub(dw1 * LEARNING_RATE)

    if step % 100 == 0:
        loss = loss_mse(X, Y, w0, w1)
        print(MSG.format(step=step, loss=loss, w0=w0.numpy(), w1=w1.numpy()))
```

```
STEP 0 - loss: 35.70719528198242, w0: 4.079999923706055, w1: 0.7599999904632568
STEP 100 - loss: 2.6017532348632812, w0: 2.4780430793762207, w1: 7.002389907836914
STEP 200 - loss: 0.26831889152526855, w0: 2.153517961502075, w1: 9.037351608276367
STEP 300 - loss: 0.027671903371810913, w0: 2.0493006706237793, w1: 9.690855979919434
STEP 400 - loss: 0.0028539239428937435, w0: 2.0158326625823975, w1: 9.90071964263916
STEP 500 - loss: 0.0002943490108009428, w0: 2.005084753036499, w1: 9.96811580657959
STEP 600 - loss: 3.0356444767676294e-05, w0: 2.0016329288482666, w1: 9.989760398864746
STEP 700 - loss: 3.1322738323069643e-06, w0: 2.0005245208740234, w1: 9.996710777282715
STEP 800 - loss: 3.2238213520940917e-07, w0: 2.0001683235168457, w1: 9.998944282531738
STEP 900 - loss: 3.369950718479231e-08, w0: 2.000054359436035, w1: 9.999658584594727
STEP 1000 - loss: 3.6101481803996194e-09, w0: 2.0000178813934326, w1: 9.99988842010498
```

In [ ]:

```
# Here we can compare the test loss for this linear regression to the test loss from th
# Its output will always be the mean of the training set:
loss = loss_mse(X_test, Y_test, w0, w1)
loss.numpy()
```

```
2.4563633e-08
```

This is indeed much better!

Bonus

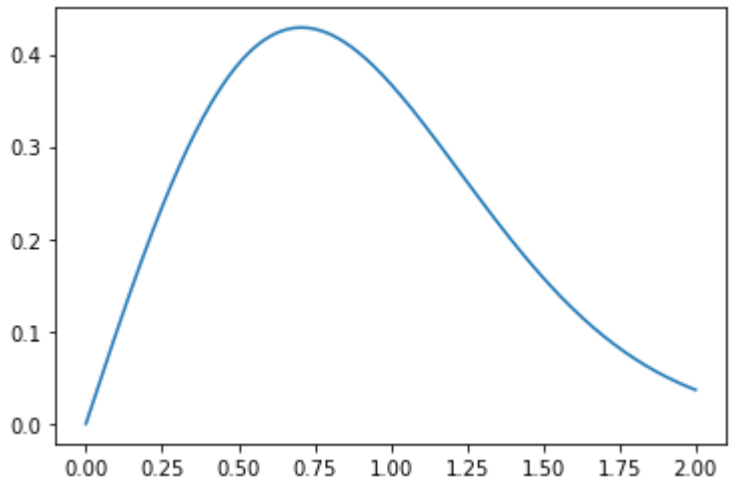
Try modeling a non-linear function such as: $y=xe^{-x^2}$

In [ ]:

```
X = tf.constant(np.linspace(0, 2, 1000), dtype=tf.float32)
Y = X * tf.exp(-X**2)
```

In [ ]:

```
%matplotlib inline

# The .plot() is a versatile function, and will take an arbitrary number of arguments.
plt.plot(X, Y)
```



In [ ]:

```
# Let's make_features() procedure.
def make_features(X):
# The tf.ones_like() method will create a tensor of all ones that has the same shape as
    f1 = tf.ones_like(X)
    f2 = X
# The tf.square() method will compute square of input tensor element-wise.
    f3 = tf.square(X)
# The tf.sqrt() method will compute element-wise square root of the input tensor.
```

```
    f4 = tf.sqrt(X)
# The tf.exp() method will compute exponential of input tensor element-wise.
    f5 = tf.exp(X)
# The tf.stack() method will stacks a list of rank-R tensors into one rank-(R+1) tensor
    return tf.stack([f1, f2, f3, f4, f5], axis=1)
```

In [ ]:

```
# Let's define predict() procedure that will remove dimensions of size 1 from the shape
def predict(X, W):
    return tf.squeeze(X @ W, -1)
```

In [ ]:

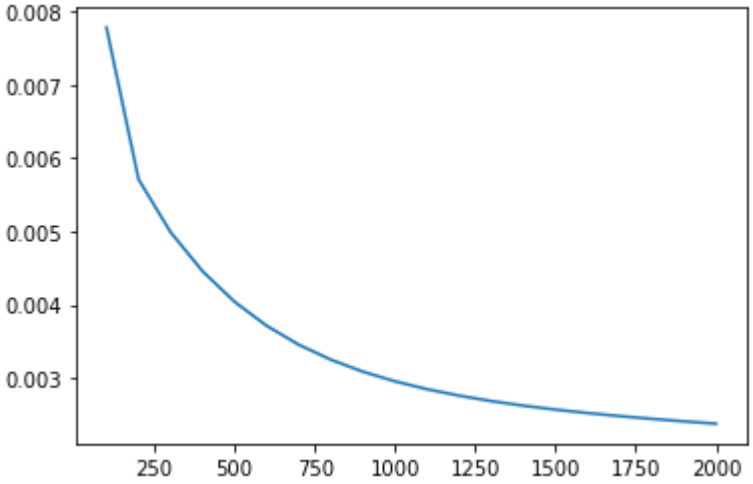
```
# Let's define loss_mse() procedure that will evaluate the mean of elements across dime
def loss_mse(X, Y, W):
    Y_hat = predict(X, W)
    errors = (Y_hat - Y)**2
    return tf.reduce_mean(errors)
```

In [ ]:

```
# Let's define compute_gradients() procedure for computing the loss gradients.
def compute_gradients(X, Y, W):
    with tf.GradientTape() as tape:
        loss = loss_mse(Xf, Y, W)
    return tape.gradient(loss, W)
```

In [ ]:

```
STEPS = 2000
LEARNING_RATE = .02


Xf = make_features(X)
n_weights = Xf.shape[1]

W = tf.Variable(np.zeros((n_weights, 1)), dtype=tf.float32)

# For plotting
steps, losses = [], []
plt.figure()


for step in range(1, STEPS + 1):

    dW = compute_gradients(X, Y, W)
    W.assign_sub(dW * LEARNING_RATE)

    if step % 100 == 0:
        loss = loss_mse(Xf, Y, W)
        steps.append(step)
        losses.append(loss)
        plt.clf()
        plt.plot(steps, losses)


print("STEP: {} MSE: {}".format(STEPS, loss_mse(Xf, Y, W)))
```



In [ ]:

```
# The .figure() method will create a new figure, or activate an existing figure.
plt.figure()
# The .plot() is a versatile function, and will take an arbitrary number of arguments.
plt.plot(X, Y, label='actual')
plt.plot(X, predict(Xf, W), label='predicted')
# The .legend() method will place a legend on the axes.
plt.legend()
```