

---

# Introduction to Tensorflow and GradientTape

Shivam Goel(190805)    Astha Pant(190199)

---

February 1, 2022

## Contents

<b>1</b>	<b>What is TensorFlow?</b>	<b>3</b>
<b>2</b>	<b>Why Tensorflow?</b>	<b>3</b>
<b>3</b>	<b>GradientTape</b>	<b>3</b>
<b>4</b>	<b>Practical Examples of Calculating Derivatives Using GradientTape</b>	<b>4</b>
4.1	Derivatives of basic polynomial function . . . . .	4
4.2	Derivatives of basic polynomial function with parameter x as constant . . . . .	5
4.3	Calculating Double and Other Higher Order Derivatives	5
4.4	Calculating Multivariate Derivatives . . . . .	6
<b>5</b>	<b>Application of GradientTape in Linear Regression</b>	<b>7</b>
<b>6</b>	<b>Application of GradientTape in Neural Networks</b>	<b>8</b>

## 1 What is TensorFlow?

TensorFlow is an open-source library used for numerical computation and large scale machine learning. It allows developers to create machine learning applications using various tools, libraries, and community resources. More information about TensorFlow can be collected from its [website](#).

## 2 Why Tensorflow?

Learning that TensorFlow is used for numerical computation, a question arises why not use the Numpy library which also provides various numerical computing tools. So here are some advantages of TensorFlow over Numpy which signifies its vast usage in machine learning applications:-

- TensorFlow can be integrated with GPU which puts it in great advantage when it comes to large data as the computation speed increases with GPU.
- TensorFlow provides us with automatic gradient computation which is of a lot of utility especially in backpropagation.
- TensorFlow can also be used to store the relationship between the functions and its variables which can be used to calculate derivatives and other functions

Some other utilities of TensorFlow are:-

- TensorFlow provides a better way of visualizing data with its graphical approach using TensorBoard.
- TensorFlow offers high-level API in the form of [Keras](#) which allows us to build and train models easily and offer high-level functionality at the same time.

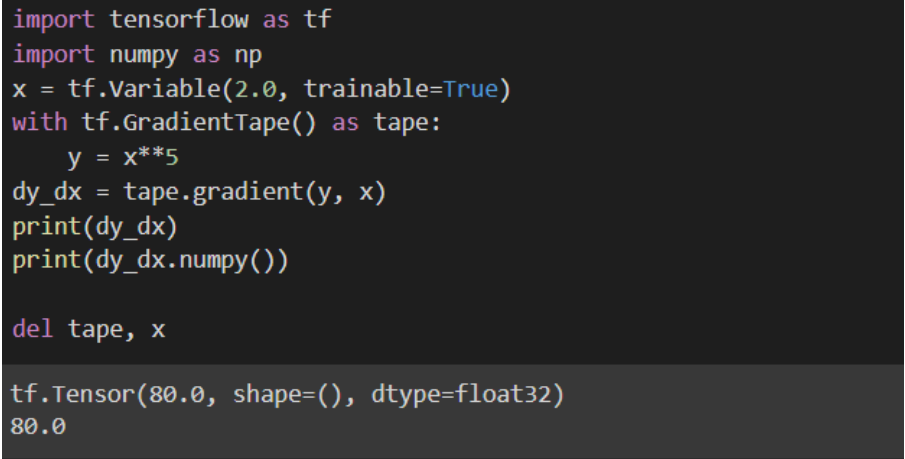
## 3 GradientTape

GradientTape is a mathematical tool for automatic differentiation, which is the core functionality of TensorFlow. It is used to record a sequence of operations performed upon some input and producing some output,executed inside the context of a `tf.GradientTape` onto a “tape”, so that the output can be differentiated with respect to the input. The variables used in calculating are generally TensorFlow

Variables (“tf.Variable(value)”) and not numpy variables as the variable tensor stores the operation between input and output useful in calculating the derivatives. Also the TensorFlow variables are trainable in nature which is automatically watched by GradientTape.

## 4 Practical Examples of Calculating Derivatives Using GradientTape

### 4.1 Derivatives of basic polynomial function



```
import tensorflow as tf
import numpy as np
x = tf.Variable(2.0, trainable=True)
with tf.GradientTape() as tape:
    y = x**5
dy_dx = tape.gradient(y, x)
print(dy_dx)
print(dy_dx.numpy())

del tape, x

tf.Tensor(80.0, shape=(), dtype=float32)
80.0
```

Figure 1: Derivative of  $y = x^5$  using GradientTape

In the above example, we have x as a TensorFlow variable whose initial assigned value is 2.0. We want to calculate the derivative of y with respect to x where y depends on x by the following relation -  $y = x^5$ . So we have created a tape which has stored this relation and using the method “tape.gradient(y,x)” the derivative is calculated and the value is stored within the variable “dy\_dx”. Using maths, the derivative of y w.r.t x is  $y = 5x^4$  which calculates to 80.0 on substituting x with its value 2.0 which is the value displayed in the output. since “dy\_dx” is a TensorFlow variable the output is of tensor data type and to get only the value we have converted it to numpy variable using method “numpy()”. To avoid use of excess memory we have deleted the tape and variable x at the end.

## 4.2 Derivatives of basic polynomial function with parameter x as constant

```
import tensorflow as tf
import numpy as np
x = tf.constant(2.0)
with tf.GradientTape() as tape:
    tape.watch(x)
    y = x**5
dy_dx = tape.gradient(y, x)
print(dy_dx)
print(dy_dx.numpy())

del tape, x

tf.Tensor(80.0, shape=(), dtype=float32)
80.0
```

Figure 2: Derivative of  $y = x^5$  using GradientTape with x as constant

The above example is same as the previous one except for the fact that x here is of the type “tf.constant” so it is not a trainable thus would not be automatically watched by GradientTape. Therefore, to tape the relationship between y and x, we invoke the method “tape.watch()” thereby getting the function  $y = x^5$  recorded. The output and rest things are same as the previous example.

## 4.3 Calculating Double and Other Higher Order Derivatives

```
x = tf.Variable(2.0, trainable=True)
with tf.GradientTape() as tape1:
    with tf.GradientTape() as tape2:
        y = x ** 5
        dy_dx = tape2.gradient(y, x)
    d2y_dx2 = tape1.gradient(dy_dx, x)

print(dy_dx.numpy())
print(d2y_dx2.numpy())

del x, tape1, tape2

80.0
160.0
```

Figure 3: Second-Order Derivative of  $y = x^5$  using GradientTape

In the above example we have calculated the double derivative of  $y = x^5$  using Nested GradientTape. Here, the tape2 stores the relation  $y = x^5$  and the visibility of tape2 ends here. The tape1 then stores the relation first derivative of y w.r.t x which is  $y = 5x^4$  and the visibility

of `tape1` ends here. Now, we use the method “`tape1.gradient()`” to stores the double derivative of  $y$  w.r.t  $x$  which is  $y = 20x^3$  and the value is stored in the variable “`d2y_dx2`” which calculates to 160 which can be seen in the output.

Similarly, the third and other higher order derivatives can be calculated through the same nested concept where the base relation of  $y$  w.r.t  $x$  is stored in innermost tape and subsequent derivatives in each above tape.

#### 4.4 Calculating Multivariate Derivatives

```
x1 = tf.Variable(2.0, trainable=True)
x2 = tf.Variable(1.0, trainable=True)
with tf.GradientTape(persistent=True) as tape: # persistent=True
    y = x1 ** 5 + x2**3
dy_dx1 = tape.gradient(y, x1)
dy_dx2 = tape.gradient(y, x2)
dy_dx1_dx2 = tape.gradient(y, [x1,x2])
print(dy_dx1.numpy())
print(dy_dx2.numpy())
print(dy_dx1_dx2)
del x1, x2, tape

80.0
3.0
[<tf.Tensor: shape=(), dtype=float32, numpy=80.0>, <tf.Tensor: shape=(), dtype=float32, numpy=3.0>]
```

Figure 4: Multivariate Derivative of  $y = x_1^5 + x_2^3$  using GradientTape

GradientTape can also be used to calculate Multivariate Derivatives, that is,  $y$  depends on two or more than two variables.

Here, we see a new parameter “persistent” which is set as True in GradientTape. By default, this parameter is set False which implies that the tape can be used only once but in the given example case we have used it thrice so to be able to do so we have set the parameter to true.

We can get the derivatives w.r.t to  $x_1$  and  $x_2$  by using gradient method twice or can get the same using it only once and passing the list  $[x_1, x_2]$  instead of a single variable. Note, here the output is also of the type list storing the derivative of  $y$  w.r.t  $x_1$  first and then  $x_2$ .  $dy_{dx1}$  has the relation  $y = 5x_1^4$  which calculates to 80.0 as seen in the output and  $dy_{dx2}$  has relation  $y = 3x_2^2$  calculating to 3.0 same as that shown in the output.

## 5 Application of GradientTape in Linear Regression

```
# Training data
x_train = np.asarray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y_train = np.asarray([6*i**2 + 8*i + 2 for i in x_train]) # y = 6x^2 + 8x + 2
# Trainable variables
a = tf.Variable(np.random.random(), trainable=True)
b = tf.Variable(np.random.random(), trainable=True)
c = tf.Variable(np.random.random(), trainable=True)
# Loss function
def loss(real_y, pred_y):
    return tf.abs(real_y - pred_y)
# Step function
def step(real_x, real_y):
    with tf.GradientTape(persistent=True) as tape:
        # Make prediction
        pred_y = a*real_x**2 + b*real_x + c
        # Calculate loss
        poly_loss = loss(real_y, pred_y)
        # Calculate gradients
        a_gradients, b_gradients, c_gradients = tape.gradient(poly_loss, (a, b, c))
        # Update variables
        # a = a - a_gradients * 0.001 # Gives error
        a.assign(a - a_gradients * 0.001)
        b.assign_sub(b_gradients * 0.001)
        c.assign_sub(c_gradients * 0.001)
# Training loop
for _ in range(10000):
    step(x_train, y_train)
print(f'y ≈ {a.numpy()}x^2 + {b.numpy()}x + {c.numpy()}')
del a, b, x_train, y_train, step, loss

y ≈ 5.99811315536499x^2 + 7.555886745452881x + 1.9986547231674194
```

Figure 5: Estimating the parameters of Quadratic Function using Linear Regression

Now we start to see the applications of TensorFlow GradientTape in various machine learning applications. One such algorithm is Linear Regression which has been previously taught in the course EE698V: Machine Learning for Signal Processing. To understand Linear Regression, one may use the lectures of the course [EE698V](#)

We have to predict here the parameters(a,b,c) of the quadratic function  $y = ax^2 + bx + c$ . The real values of the parameters are 6,8,2 respectively.

So we have taken TensorFlow trainable variables a,b and c and have assigned them random values. The loss function is taken as absolute difference between real values and predicted values.

Now we define the step function, which uses the GradientTape where we tape the predictive function as  $pred_y = a * real_x^2 + b * real_x + c$  and also tape the result of the loss function defined previously in the variable “poly\_loss”. We store the gradients of all the variables differentiating the “poly\_loss” function w.r.t each variable and then updating

the variables as done in gradient descent.

Finally, we iterate the step function 10000 times which updates the variables a,b,c to values 5.998, 7.555 and 1.998 respectively which are quite close the original parameters.

## 6 Application of GradientTape in Neural Networks

```
from tensorflow.keras.layers import Conv2D, Flatten, Dense, Dropout, MaxPooling2D
from tensorflow.keras.models import Sequential
from tensorflow.keras.initializers import RandomNormal
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import math
%matplotlib inline
# Load and pre-process training data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = (x_train / 255).reshape((-1, 28, 28, 1))
y_train = tf.keras.utils.to_categorical(y_train, 10)
x_test = (x_test / 255).reshape((-1, 28, 28, 1))
y_test = tf.keras.utils.to_categorical(y_test, 10)
# Hyperparameters
batch_size = 128
epochs = 25
optimizer = Adam(lr=0.001)
weight_init = RandomNormal()
# Build model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', kernel_initializer=weight_init, input_shape=(28, 28, 1)))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer=weight_init))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer=weight_init))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax', kernel_initializer=weight_init))
```

Figure 6: Building the model using TensorFlow Keras

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
conv2d_3 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling 2D)	(None, 12, 12, 64)	0
dropout_2 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_2 (Dense)	(None, 128)	1179776
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1290

=====  
 Total params: 1,199,882  
 Trainable params: 1,199,882  
 Non-trainable params: 0

Figure 7: Summary of the Convolution Neural Network Model



We have developed the Neural Network Model above which has been done through previous understand of CNN through the Course EE698V whose lectures can be accessed from [here](#).

```
# Step function
def step(real_x, real_y):
    with tf.GradientTape() as tape:
        # Make prediction
        pred_y = model(real_x.reshape((-1, 28, 28, 1)))
        # Calculate loss
        model_loss = tf.keras.losses.categorical_crossentropy(real_y, pred_y)

    # Calculate gradients
    model_gradients = tape.gradient(model_loss, model.trainable_variables)
    # Update model
    optimizer.apply_gradients(zip(model_gradients, model.trainable_variables))

# Training loop
bat_per_epoch = math.floor(len(x_train) / batch_size)
for epoch in range(epochs):
    print('-', end='')
    for i in range(bat_per_epoch):
        n = i*batch_size
        step(x_train[n:n+batch_size], y_train[n:n+batch_size])

# Calculate accuracy
model.compile(optimizer=optimizer, loss=tf.losses.categorical_crossentropy, metrics=['acc']) # Compile just for evaluation
print('\nAccuracy:', model.evaluate(x_test, y_test, verbose=0)[1])

=====
Accuracy: 0.9876999855041504

y = model.predict(x_test)
print(y.shape)

(10000, 10)
```

Figure 8: Use of GradientTape in CNN

Now we move on to the Gradient Descent part which is quite similar to the Linear Regression example which we have done previously. The only change here is of the loss function as here we have used the Categorical Cross Entropy present in the Keras library and have added epochs to train the model better so that we could achieve a higher accuracy which we did by achieving 98.77%.