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

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
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0286 - accuracy: 0.9907
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0255 - accuracy: 0.9911
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0208 - accuracy: 0.9928

<keras.callbacks.History at 0x7ffa980f5d20>
```

Evaluate the mnistmodel4

```
mnistmodel4.evaluate(test_x, test_y)

313/313 [==============================] - 1s 2ms/step - loss: 0.0768
- accuracy: 0.9815

[0.07678413391113281, 0.9815000295639038]
```

pytorch-demo

December 27, 2023

1 PyTorch Basics: Tensors & Gradients

PyTorch is an open-source machine learning framework that is primarily used for developing and training deep learning models. It was developed by Facebook's AI Research Lab and released in 2016. PyTorch provides a flexible and dynamic approach to building neural networks, making it a popular choice among researchers and developers.

The framework is built on a dynamic computational graph concept, which means that the graph is built and modified on-the-fly as the program runs. This allows for more intuitive and flexible model development, as you can use standard Python control flow statements and debug the model easily.

PyTorch supports automatic differentiation, which enables efficient computation of gradients for training neural networks using backpropagation. It provides a rich set of tools and libraries for tasks such as data loading, model building, optimization, and evaluation.

One of the key advantages of PyTorch is its support for GPU acceleration, allowing you to train models on GPUs to significantly speed up computations. It also has a large and active community, which means there are plenty of resources, tutorials, and pre-trained models available.

PyTorch is often compared to TensorFlow, another popular deep learning framework. While TensorFlow focuses more on static computation graphs, PyTorch emphasizes dynamic computation graphs. This fundamental difference in design philosophy gives PyTorch an edge when it comes to flexibility and ease of use.

Overall, PyTorch is widely used in the research community and is gaining popularity in industry applications as well. It provides a powerful and user-friendly platform for building and training deep learning models.

1.1 installation

installation instructions here: https://pytorch.org .

[ ]:
```
# Uncomment and run the appropriate command for your operating system, if␣
↪required

# Linux / Binder
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0␣
↪-f https://download.pytorch.org/whl/torch_stable.html

# Windows
```

```
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0␣
↪-f https://download.pytorch.org/whl/torch_stable.html

# MacOS
# !pip install numpy torch torchvision torchaudio
```

Let's import the `torch` module to get started.

```
[ ]: import torch
```

1.2 Tensors

At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix, or any n-dimensional array. Let's create a tensor with a single number.

```
[ ]: # Number
     t1 = torch.tensor(4.)
     t1
```

```
[ ]: tensor(4.)
```

`4.` is a shorthand for `4.0`. It is used to indicate to Python (and PyTorch) that you want to create a floating-point number. We can verify this by checking the `dtype` attribute of our tensor.

```
[ ]: t1.dtype
```

```
[ ]: torch.float32
```

Let's try creating more complex tensors.

```
[ ]: # Vector
     t2 = torch.tensor([1., 2, 3, 4])
     t2
```

```
[ ]: tensor([1., 2., 3., 4.])
```

```
[ ]: # Matrix
     t3 = torch.tensor([[5., 6],
                        [7, 8],
                        [9, 10]])
     t3
```

```
[ ]: tensor([[ 5.,  6.],
             [ 7.,  8.],
             [ 9., 10.]])
```

```
[ ]: # 3-dimensional array
     t4 = torch.tensor([
```

```
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])
t4
```

[ ]:
```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

Tensors can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of a tensor.

[ ]:
```
print(t1)
t1.shape
```

```
tensor(4.)
```

[ ]:
```
torch.Size([])
```

[ ]:
```
print(t2)
t2.shape
```

```
tensor([1., 2., 3., 4.])
```

[ ]:
```
torch.Size([4])
```

[ ]:
```
print(t3)
t3.shape
```

```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
```

[ ]:
```
torch.Size([3, 2])
```

[ ]:
```
print(t4)
t4.shape
```

```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

[ ]:
```
torch.Size([2, 2, 3])
```

Note that it's not possible to create tensors with an improper shape.

```
[ ]: # Matrix
t5 = torch.tensor([[5., 6, 11],
                   [7, 8],
                   [9, 10]])
t5
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-83912cf67c5e> in <cell line: 2>()
      1 # Matrix
----> 2 t5 = torch.tensor([[5., 6, 11],

      3                    [7, 8],
      4                    [9, 10]])
      5 t5

ValueError: expected sequence of length 3 at dim 1 (got 2)
```

A `ValueError` is thrown because the lengths of the rows `[5., 6, 11]` and `[7, 8]` don't match.

1.3 Tensor operations and gradients

We can combine tensors with the usual arithmetic operations. Let's look at an example:

```
[ ]: # Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

```
[ ]: (tensor(3.), tensor(4., requires_grad=True), tensor(5., requires_grad=True))
```

We've created three tensors: `x`, `w`, and `b`, all numbers. `w` and `b` have an additional parameter `requires_grad` set to `True`. We'll see what it does in just a moment.

Let's create a new tensor `y` by combining these tensors.

```
[ ]: # Arithmetic operations
y = w * x + b
y
```

```
[ ]: tensor(17., grad_fn=<AddBackward0>)
```

As expected, `y` is a tensor with the value `3 * 4 + 5 = 17`. What makes PyTorch unique is that we can automatically compute the derivative of `y` w.r.t. the tensors that have `requires_grad` set to `True` i.e. w and b. This feature of PyTorch is called *autograd* (automatic gradients).

To compute the derivatives, we can invoke the `.backward` method on our result `y`.

[ ]:
```python
# Compute derivatives
y.backward()
```

The derivatives of `y` with respect to the input tensors are stored in the `.grad` property of the respective tensors.

[ ]:
```python
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

As expected, `dy/dw` has the same value as `x`, i.e., `3`, and `dy/db` has the value `1`. Note that `x.grad` is `None` because `x` doesn't have `requires_grad` set to `True`.

The "grad" in `w.grad` is short for *gradient*, which is another term for derivative. The term *gradient* is primarily used while dealing with vectors and matrices.

1.4 Tensor functions

Apart from arithmetic operations, the `torch` module also contains many functions for creating and manipulating tensors. Let's look at some examples.

[ ]:
```python
# Create a tensor with a fixed value for every element
t6 = torch.full((3, 2), 42)
t6
```

[ ]:
```
tensor([[42, 42],
        [42, 42],
        [42, 42]])
```

[ ]:
```python
# Concatenate two tensors with compatible shapes
t7 = torch.cat((t3, t6))
t7
```

[ ]:
```
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.],
        [42., 42.],
        [42., 42.],
        [42., 42.]])
```

[ ]:
```python
# Compute the sin of each element
t8 = torch.sin(t7)
t8
```

```
[ ]: tensor([[-0.9589, -0.2794],
        [ 0.6570,  0.9894],
        [ 0.4121, -0.5440],
        [-0.9165, -0.9165],
        [-0.9165, -0.9165],
        [-0.9165, -0.9165]])
```

```
[ ]: # Change the shape of a tensor
t9 = t8.reshape(3, 2, 2)
t9
```

```
[ ]: tensor([[[-0.9589, -0.2794],
         [ 0.6570,  0.9894]],

        [[ 0.4121, -0.5440],
         [-0.9165, -0.9165]],

        [[-0.9165, -0.9165],
         [-0.9165, -0.9165]]])
```

You can learn more about tensor operations here: https://pytorch.org/docs/stable/torch.html . Experiment with some more tensor functions and operations using the empty cells below.

```
[ ]:
```

1.5 Interoperability with Numpy

Numpy is a popular open-source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays and has a vast ecosystem of supporting libraries, including:

- Pandas for file I/O and data analysis
- Matplotlib for plotting and visualization
- OpenCV for image and video processing

Instead of reinventing the wheel, PyTorch interoperates well with Numpy to leverage its existing ecosystem of tools and libraries.

Here's how we create an array in Numpy:

```
[ ]: import numpy as np

x = np.array([[1, 2], [3, 4.]])
x
```

```
[ ]: array([[1., 2.],
       [3., 4.]])
```

We can convert a Numpy array to a PyTorch tensor using `torch.from_numpy`.

[ ]:
```python
# Convert the numpy array to a torch tensor.
y = torch.from_numpy(x)
y
```

[ ]:
```
tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

Let's verify that the numpy array and torch tensor have similar data types.

[ ]:
```python
x.dtype, y.dtype
```

[ ]:
```
(dtype('float64'), torch.float64)
```

We can convert a PyTorch tensor to a Numpy array using the `.numpy` method of a tensor.

[ ]:
```python
# Convert a torch tensor to a numpy array
z = y.numpy()
z
```

[ ]:
```
array([[1., 2.],
       [3., 4.]])
```

The interoperability between PyTorch and Numpy is essential because most datasets you'll work with will likely be read and preprocessed as Numpy arrays.

You might wonder why we need a library like PyTorch at all since Numpy already provides data structures and utilities for working with multi-dimensional numeric data. There are two main reasons:

1. **Autograd**: The ability to automatically compute gradients for tensor operations is essential for training deep learning models.
2. **GPU support**: While working with massive datasets and large models, PyTorch tensor operations can be performed efficiently using a Graphics Processing Unit (GPU). Computations that might typically take hours can be completed within minutes using GPUs.

[ ]:
```python

```

1.6 Linear-regression from scrach using pytorch

[ ]:
```python
import numpy as np
import torch
```

[ ]:
```python
#making training data
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')
```

[ ]:
```
# Targets (apples, oranges)
target = np.array([[56, 70],
                   [81, 101],
                   [119, 133],
                   [22, 37],
                   [103, 119]], dtype='float32')
```

[ ]:
```
#Convert input and target to tensors
inputs = torch.from_numpy(inputs)
target = torch.from_numpy(target)

print(inputs,"\n")
print(target)
```

```
tensor([[ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 87., 134.,  58.],
        [102.,  43.,  37.],
        [ 69.,  96.,  70.]])

tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

[ ]:
```
# weights and biases
w = torch.randn(2,3 , requires_grad=True)
b = torch.randn(2, requires_grad=True)

print(w)
print(b)
```

```
tensor([[-0.9360, -3.2651,  0.1739],
        [ 0.8647, -1.0314, -0.5762]], requires_grad=True)
tensor([ 0.4978, -0.2630], requires_grad=True)
```

[ ]:
```
#define the model

def model(x):
  return x @ w.t() + b
```

[ ]:
```
# prediction
preds = model(inputs)
print(preds)
```

```
tensor([[-279.1155,  -31.0234],
        [-360.8790,  -49.2193],
```

```
        [-508.3732,  -96.6649],
        [-228.9407,   22.2628],
        [-365.3642,  -79.9504]], grad_fn=<AddBackward0>)
```

[ ]:
```python
#actual
print(target)
```

```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

[ ]:
```python
# loss function MSE
def MSE(actual, target):
  diff = actual - target
  return torch.sum(diff * diff) / diff.numel()
```

[ ]:
```python
# error
loss = MSE(target, preds)
print(loss)
```

```
tensor(110880.8750, grad_fn=<DivBackward0>)
```

[ ]:
```python
# compute gradients
loss.backward()
```

[ ]:
```python
print(w, "\n")
print(w.grad)
```

```
tensor([[-0.9360, -3.2651,  0.1739],
        [ 0.8647, -1.0314, -0.5762]], requires_grad=True)

tensor([[-35433.7969, -40231.9023, -24229.6328],
        [-11251.2559, -14099.1797,  -8350.0820]])
```

[ ]:
```python
print(b, "\n")
print(b.grad)
```

```
tensor([ 0.4978, -0.2630], requires_grad=True)

tensor([-424.7345, -138.9190])
```

[ ]:
```python
#reset grad
w.grad.zero_()
b.grad.zero_()

print(w.grad)
```

```
print(b.grad)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([0., 0.])
```

[ ]:
```
# adjust params

preds = model(inputs)
print(preds)
```

```
tensor([[-279.1155,  -31.0234],
        [-360.8790,  -49.2193],
        [-508.3732,  -96.6649],
        [-228.9407,   22.2628],
        [-365.3642,  -79.9504]], grad_fn=<AddBackward0>)
```

[ ]:
```
# loss
loss = MSE(target, preds)
print(loss)
```

```
tensor(110880.8750, grad_fn=<DivBackward0>)
```

[ ]:
```
loss.backward()

print(w.grad, "\n")
print(b.grad)
```

```
tensor([[-35433.7969, -40231.9023, -24229.6328],
        [-11251.2559, -14099.1797,  -8350.0820]])

tensor([-424.7345, -138.9190])
```

[ ]:
```
# adjust weight & reset grad
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
    w.grad.zero_()
    b.grad.zero_()
```

[ ]:
```
print(w)
print(b)
```

```
tensor([[-0.5817, -2.8628,  0.4162],
        [ 0.9772, -0.8904, -0.4927]], requires_grad=True)
tensor([ 0.5020, -0.2616], requires_grad=True)
```

[ ]:
```
# calculate again
preds = model(inputs)
loss = MSE(target, preds)
print(loss)
```

```
tensor(75765.4062, grad_fn=<DivBackward0>)
```

[ ]:
```
# Training for multiple epochs
for i in range(400):
  preds = model(inputs)
  loss = MSE(target, preds)
  loss.backward()

  with torch.no_grad():
     w -= w.grad * 1e-5 # learning rate
     b -= b.grad * 1e-5
     w.grad.zero_()
     b.grad.zero_()
  print(f"Epochs({i}/{100}) & Loss {loss}")
```

```
Epochs(0/100) & Loss 75765.40625
Epochs(1/100) & Loss 52088.75
Epochs(2/100) & Loss 36120.72265625
Epochs(3/100) & Loss 25347.583984375
Epochs(4/100) & Loss 18075.365234375
Epochs(5/100) & Loss 13162.5283203125
Epochs(6/100) & Loss 9839.7890625
Epochs(7/100) & Loss 7588.75
Epochs(8/100) & Loss 6060.0634765625
Epochs(9/100) & Loss 5018.30615234375
Epochs(10/100) & Loss 4304.82958984375
Epochs(11/100) & Loss 3812.721435546875
Epochs(12/100) & Loss 3469.931640625
Epochs(13/100) & Loss 3227.90576171875
Epochs(14/100) & Loss 3053.92236328125
Epochs(15/100) & Loss 2925.927734375
Epochs(16/100) & Loss 2829.060302734375
Epochs(17/100) & Loss 2753.3017578125
Epochs(18/100) & Loss 2691.900146484375
Epochs(19/100) & Loss 2640.30419921875
Epochs(20/100) & Loss 2595.443359375
Epochs(21/100) & Loss 2555.24853515625
Epochs(22/100) & Loss 2518.322998046875
Epochs(23/100) & Loss 2483.724853515625
Epochs(24/100) & Loss 2450.81640625
Epochs(25/100) & Loss 2419.167236328125
Epochs(26/100) & Loss 2388.48583984375
Epochs(27/100) & Loss 2358.57470703125
```

```
Epochs(28/100) & Loss 2329.297607421875
Epochs(29/100) & Loss 2300.56298828125
Epochs(30/100) & Loss 2272.30615234375
Epochs(31/100) & Loss 2244.484375
Epochs(32/100) & Loss 2217.064453125
Epochs(33/100) & Loss 2190.025390625
Epochs(34/100) & Loss 2163.35009765625
Epochs(35/100) & Loss 2137.02587890625
Epochs(36/100) & Loss 2111.04345703125
Epochs(37/100) & Loss 2085.39501953125
Epochs(38/100) & Loss 2060.07275390625
Epochs(39/100) & Loss 2035.0718994140625
Epochs(40/100) & Loss 2010.387451171875
Epochs(41/100) & Loss 1986.0140380859375
Epochs(42/100) & Loss 1961.9476318359375
Epochs(43/100) & Loss 1938.18359375
Epochs(44/100) & Loss 1914.71875
Epochs(45/100) & Loss 1891.548583984375
Epochs(46/100) & Loss 1868.669189453125
Epochs(47/100) & Loss 1846.076904296875
Epochs(48/100) & Loss 1823.7685546875
Epochs(49/100) & Loss 1801.739501953125
Epochs(50/100) & Loss 1779.987548828125
Epochs(51/100) & Loss 1758.507568359375
Epochs(52/100) & Loss 1737.297607421875
Epochs(53/100) & Loss 1716.3531494140625
Epochs(54/100) & Loss 1695.671630859375
Epochs(55/100) & Loss 1675.2496337890625
Epochs(56/100) & Loss 1655.0836181640625
Epochs(57/100) & Loss 1635.1702880859375
Epochs(58/100) & Loss 1615.506591796875
Epochs(59/100) & Loss 1596.0894775390625
Epochs(60/100) & Loss 1576.9154052734375
Epochs(61/100) & Loss 1557.982177734375
Epochs(62/100) & Loss 1539.2857666015625
Epochs(63/100) & Loss 1520.8238525390625
Epochs(64/100) & Loss 1502.5931396484375
Epochs(65/100) & Loss 1484.5909423828125
Epochs(66/100) & Loss 1466.8143310546875
Epochs(67/100) & Loss 1449.2601318359375
Epochs(68/100) & Loss 1431.926025390625
Epochs(69/100) & Loss 1414.8089599609375
Epochs(70/100) & Loss 1397.906494140625
Epochs(71/100) & Loss 1381.2152099609375
Epochs(72/100) & Loss 1364.733154296875
Epochs(73/100) & Loss 1348.4573974609375
Epochs(74/100) & Loss 1332.385498046875
Epochs(75/100) & Loss 1316.5146484375
```

```
Epochs(76/100) & Loss 1300.8424072265625
Epochs(77/100) & Loss 1285.366455078125
Epochs(78/100) & Loss 1270.083984375
Epochs(79/100) & Loss 1254.992919921875
Epochs(80/100) & Loss 1240.090576171875
Epochs(81/100) & Loss 1225.374755859375
Epochs(82/100) & Loss 1210.8428955078125
Epochs(83/100) & Loss 1196.492919921875
Epochs(84/100) & Loss 1182.322265625
Epochs(85/100) & Loss 1168.328857421875
Epochs(86/100) & Loss 1154.510498046875
Epochs(87/100) & Loss 1140.86474609375
Epochs(88/100) & Loss 1127.3897705078125
Epochs(89/100) & Loss 1114.0830078125
Epochs(90/100) & Loss 1100.942626953125
Epochs(91/100) & Loss 1087.966552734375
Epochs(92/100) & Loss 1075.1524658203125
Epochs(93/100) & Loss 1062.4984130859375
Epochs(94/100) & Loss 1050.0025634765625
Epochs(95/100) & Loss 1037.662841796875
Epochs(96/100) & Loss 1025.477294921875
Epochs(97/100) & Loss 1013.4439697265625
Epochs(98/100) & Loss 1001.5606689453125
Epochs(99/100) & Loss 989.8259887695312
Epochs(100/100) & Loss 978.2376098632812
Epochs(101/100) & Loss 966.7937622070312
Epochs(102/100) & Loss 955.4929809570312
Epochs(103/100) & Loss 944.3331909179688
Epochs(104/100) & Loss 933.3126220703125
Epochs(105/100) & Loss 922.4293212890625
Epochs(106/100) & Loss 911.6820068359375
Epochs(107/100) & Loss 901.0687255859375
Epochs(108/100) & Loss 890.5877075195312
Epochs(109/100) & Loss 880.2374877929688
Epochs(110/100) & Loss 870.0162353515625
Epochs(111/100) & Loss 859.9222412109375
Epochs(112/100) & Loss 849.9542846679688
Epochs(113/100) & Loss 840.1105346679688
Epochs(114/100) & Loss 830.3895263671875
Epochs(115/100) & Loss 820.7892456054688
Epochs(116/100) & Loss 811.3089599609375
Epochs(117/100) & Loss 801.9466552734375
Epochs(118/100) & Loss 792.7008056640625
Epochs(119/100) & Loss 783.5700073242188
Epochs(120/100) & Loss 774.5531616210938
Epochs(121/100) & Loss 765.6484375
Epochs(122/100) & Loss 756.8543090820312
Epochs(123/100) & Loss 748.169921875
```

```
Epochs(124/100) & Loss 739.5933837890625
Epochs(125/100) & Loss 731.1236572265625
Epochs(126/100) & Loss 722.7591552734375
Epochs(127/100) & Loss 714.4987182617188
Epochs(128/100) & Loss 706.3410034179688
Epochs(129/100) & Loss 698.2847900390625
Epochs(130/100) & Loss 690.32861328125
Epochs(131/100) & Loss 682.4713134765625
Epochs(132/100) & Loss 674.7117919921875
Epochs(133/100) & Loss 667.0484619140625
Epochs(134/100) & Loss 659.4802856445312
Epochs(135/100) & Loss 652.0061645507812
Epochs(136/100) & Loss 644.6248779296875
Epochs(137/100) & Loss 637.3352661132812
Epochs(138/100) & Loss 630.1359252929688
Epochs(139/100) & Loss 623.0261840820312
Epochs(140/100) & Loss 616.0045166015625
Epochs(141/100) & Loss 609.06982421875
Epochs(142/100) & Loss 602.2213134765625
Epochs(143/100) & Loss 595.4576416015625
Epochs(144/100) & Loss 588.7779541015625
Epochs(145/100) & Loss 582.180908203125
Epochs(146/100) & Loss 575.6657104492188
Epochs(147/100) & Loss 569.2310791015625
Epochs(148/100) & Loss 562.8763427734375
Epochs(149/100) & Loss 556.6002197265625
Epochs(150/100) & Loss 550.40185546875
Epochs(151/100) & Loss 544.2801513671875
Epochs(152/100) & Loss 538.2343139648438
Epochs(153/100) & Loss 532.2632446289062
Epochs(154/100) & Loss 526.3660888671875
Epochs(155/100) & Loss 520.5418090820312
Epochs(156/100) & Loss 514.7894897460938
Epochs(157/100) & Loss 509.1084899902344
Epochs(158/100) & Loss 503.49749755859375
Epochs(159/100) & Loss 497.9559631347656
Epochs(160/100) & Loss 492.48291015625
Epochs(161/100) & Loss 487.077392578125
Epochs(162/100) & Loss 481.7386779785156
Epochs(163/100) & Loss 476.4658203125
Epochs(164/100) & Loss 471.25811767578125
Epochs(165/100) & Loss 466.1146545410156
Epochs(166/100) & Loss 461.0345764160156
Epochs(167/100) & Loss 456.01727294921875
Epochs(168/100) & Loss 451.0616760253906
Epochs(169/100) & Loss 446.1673889160156
Epochs(170/100) & Loss 441.33331298828125
Epochs(171/100) & Loss 436.5586853027344
```

```
Epochs(172/100) & Loss 431.8429260253906
Epochs(173/100) & Loss 427.185302734375
Epochs(174/100) & Loss 422.5850524902344
Epochs(175/100) & Loss 418.0413513183594
Epochs(176/100) & Loss 413.5536193847656
Epochs(177/100) & Loss 409.12103271484375
Epochs(178/100) & Loss 404.74298095703125
Epochs(179/100) & Loss 400.41876220703125
Epochs(180/100) & Loss 396.14776611328125
Epochs(181/100) & Loss 391.9290466308594
Epochs(182/100) & Loss 387.7622985839844
Epochs(183/100) & Loss 383.6466064453125
Epochs(184/100) & Loss 379.5816345214844
Epochs(185/100) & Loss 375.56640625
Epochs(186/100) & Loss 371.6004943847656
Epochs(187/100) & Loss 367.6831970214844
Epochs(188/100) & Loss 363.8139343261719
Epochs(189/100) & Loss 359.9920959472656
Epochs(190/100) & Loss 356.21710205078125
Epochs(191/100) & Loss 352.4883728027344
Epochs(192/100) & Loss 348.8052978515625
Epochs(193/100) & Loss 345.167236328125
Epochs(194/100) & Loss 341.5737609863281
Epochs(195/100) & Loss 338.0242614746094
Epochs(196/100) & Loss 334.51812744140625
Epochs(197/100) & Loss 331.054931640625
Epochs(198/100) & Loss 327.63397216796875
Epochs(199/100) & Loss 324.2547302246094
Epochs(200/100) & Loss 320.91693115234375
Epochs(201/100) & Loss 317.6197814941406
Epochs(202/100) & Loss 314.36285400390625
Epochs(203/100) & Loss 311.1457214355469
Epochs(204/100) & Loss 307.96783447265625
Epochs(205/100) & Loss 304.82867431640625
Epochs(206/100) & Loss 301.727783203125
Epochs(207/100) & Loss 298.6646423339844
Epochs(208/100) & Loss 295.6387634277344
Epochs(209/100) & Loss 292.6497802734375
Epochs(210/100) & Loss 289.6971130371094
Epochs(211/100) & Loss 286.78033447265625
Epochs(212/100) & Loss 283.89910888671875
Epochs(213/100) & Loss 281.05279541015625
Epochs(214/100) & Loss 278.24114990234375
Epochs(215/100) & Loss 275.4635314941406
Epochs(216/100) & Loss 272.71978759765625
Epochs(217/100) & Loss 270.0091857910156
Epochs(218/100) & Loss 267.33148193359375
Epochs(219/100) & Loss 264.6863098144531
```

```
Epochs(220/100) & Loss 262.0731506347656
Epochs(221/100) & Loss 259.4916076660156
Epochs(222/100) & Loss 256.94134521484375
Epochs(223/100) & Loss 254.4219207763672
Epochs(224/100) & Loss 251.9329833984375
Epochs(225/100) & Loss 249.47409057617188
Epochs(226/100) & Loss 247.04501342773438
Epochs(227/100) & Loss 244.645263671875
Epochs(228/100) & Loss 242.27444458007812
Epochs(229/100) & Loss 239.9322509765625
Epochs(230/100) & Loss 237.6182861328125
Epochs(231/100) & Loss 235.3321533203125
Epochs(232/100) & Loss 233.07369995117188
Epochs(233/100) & Loss 230.8423614501953
Epochs(234/100) & Loss 228.63790893554688
Epochs(235/100) & Loss 226.4600372314453
Epochs(236/100) & Loss 224.30819702148438
Epochs(237/100) & Loss 222.18240356445312
Epochs(238/100) & Loss 220.08203125
Epochs(239/100) & Loss 218.0068359375
Epochs(240/100) & Loss 215.95669555664062
Epochs(241/100) & Loss 213.93099975585938
Epochs(242/100) & Loss 211.9296417236328
Epochs(243/100) & Loss 209.9523468017578
Epochs(244/100) & Loss 207.9987030029297
Epochs(245/100) & Loss 206.06832885742188
Epochs(246/100) & Loss 204.1610870361328
Epochs(247/100) & Loss 202.27671813964844
Epochs(248/100) & Loss 200.41476440429688
Epochs(249/100) & Loss 198.57504272460938
Epochs(250/100) & Loss 196.75735473632812
Epochs(251/100) & Loss 194.96133422851562
Epochs(252/100) & Loss 193.18667602539062
Epochs(253/100) & Loss 191.4331817626953
Epochs(254/100) & Loss 189.70053100585938
Epochs(255/100) & Loss 187.9885711669922
Epochs(256/100) & Loss 186.29684448242188
Epochs(257/100) & Loss 184.6253204345703
Epochs(258/100) & Loss 182.97361755371094
Epochs(259/100) & Loss 181.3415069580078
Epochs(260/100) & Loss 179.728759765625
Epochs(261/100) & Loss 178.1351318359375
Epochs(262/100) & Loss 176.56033325195312
Epochs(263/100) & Loss 175.0042266845703
Epochs(264/100) & Loss 173.46646118164062
Epochs(265/100) & Loss 171.94699096679688
Epochs(266/100) & Loss 170.4453582763672
Epochs(267/100) & Loss 168.96144104003906
```

```
Epochs(268/100) & Loss 167.4951171875
Epochs(269/100) & Loss 166.04598999023438
Epochs(270/100) & Loss 164.61404418945312
Epochs(271/100) & Loss 163.19882202148438
Epochs(272/100) & Loss 161.80030822753906
Epochs(273/100) & Loss 160.41824340820312
Epochs(274/100) & Loss 159.0523681640625
Epochs(275/100) & Loss 157.7025604248047
Epochs(276/100) & Loss 156.3686065673828
Epochs(277/100) & Loss 155.05020141601562
Epochs(278/100) & Loss 153.74722290039062
Epochs(279/100) & Loss 152.4595947265625
Epochs(280/100) & Loss 151.18695068359375
Epochs(281/100) & Loss 149.92918395996094
Epochs(282/100) & Loss 148.68612670898438
Epochs(283/100) & Loss 147.45748901367188
Epochs(284/100) & Loss 146.24325561523438
Epochs(285/100) & Loss 145.04306030273438
Epochs(286/100) & Loss 143.85690307617188
Epochs(287/100) & Loss 142.6845245361328
Epochs(288/100) & Loss 141.52572631835938
Epochs(289/100) & Loss 140.38037109375
Epochs(290/100) & Loss 139.24835205078125
Epochs(291/100) & Loss 138.1294403076172
Epochs(292/100) & Loss 137.0233917236328
Epochs(293/100) & Loss 135.93019104003906
Epochs(294/100) & Loss 134.849609375
Epochs(295/100) & Loss 133.78152465820312
Epochs(296/100) & Loss 132.72569274902344
Epochs(297/100) & Loss 131.6820831298828
Epochs(298/100) & Loss 130.65045166015625
Epochs(299/100) & Loss 129.63070678710938
Epochs(300/100) & Loss 128.62261962890625
Epochs(301/100) & Loss 127.62614440917969
Epochs(302/100) & Loss 126.64106750488281
Epochs(303/100) & Loss 125.66734313964844
Epochs(304/100) & Loss 124.70469665527344
Epochs(305/100) & Loss 123.7530517578125
Epochs(306/100) & Loss 122.8122787475586
Epochs(307/100) & Loss 121.88226318359375
Epochs(308/100) & Loss 120.96281433105469
Epochs(309/100) & Loss 120.0538330078125
Epochs(310/100) & Loss 119.1552734375
Epochs(311/100) & Loss 118.2667465209961
Epochs(312/100) & Loss 117.38846588134766
Epochs(313/100) & Loss 116.52009582519531
Epochs(314/100) & Loss 115.66153717041016
Epochs(315/100) & Loss 114.81266784667969
```

```
Epochs(316/100) & Loss 113.97335052490234
Epochs(317/100) & Loss 113.14361572265625
Epochs(318/100) & Loss 112.32319641113281
Epochs(319/100) & Loss 111.51203918457031
Epochs(320/100) & Loss 110.70989990234375
Epochs(321/100) & Loss 109.9168701171875
Epochs(322/100) & Loss 109.1327133178711
Epochs(323/100) & Loss 108.3573226928711
Epochs(324/100) & Loss 107.59063720703125
Epochs(325/100) & Loss 106.83248138427734
Epochs(326/100) & Loss 106.08284759521484
Epochs(327/100) & Loss 105.34153747558594
Epochs(328/100) & Loss 104.60848236083984
Epochs(329/100) & Loss 103.88360595703125
Epochs(330/100) & Loss 103.16676330566406
Epochs(331/100) & Loss 102.4578857421875
Epochs(332/100) & Loss 101.75688171386719
Epochs(333/100) & Loss 101.0635757446289
Epochs(334/100) & Loss 100.37793731689453
Epochs(335/100) & Loss 99.6999282836914
Epochs(336/100) & Loss 99.0293960571289
Epochs(337/100) & Loss 98.36617279052734
Epochs(338/100) & Loss 97.7103042602539
Epochs(339/100) & Loss 97.06159210205078
Epochs(340/100) & Loss 96.42002868652344
Epochs(341/100) & Loss 95.7854232788086
Epochs(342/100) & Loss 95.15788269042969
Epochs(343/100) & Loss 94.53712463378906
Epochs(344/100) & Loss 93.92306518554688
Epochs(345/100) & Loss 93.31578063964844
Epochs(346/100) & Loss 92.71503448486328
Epochs(347/100) & Loss 92.12081146240234
Epochs(348/100) & Loss 91.53311920166016
Epochs(349/100) & Loss 90.95165252685547
Epochs(350/100) & Loss 90.37657165527344
Epochs(351/100) & Loss 89.80763244628906
Epochs(352/100) & Loss 89.244873046875
Epochs(353/100) & Loss 88.6881103515625
Epochs(354/100) & Loss 88.13738250732422
Epochs(355/100) & Loss 87.5925521850586
Epochs(356/100) & Loss 87.0535659790039
Epochs(357/100) & Loss 86.52027893066406
Epochs(358/100) & Loss 85.99272918701172
Epochs(359/100) & Loss 85.47074127197266
Epochs(360/100) & Loss 84.95435333251953
Epochs(361/100) & Loss 84.44343566894531
Epochs(362/100) & Loss 83.93795013427734
Epochs(363/100) & Loss 83.43780517578125
```

```
Epochs(364/100) & Loss 82.94291687011719
Epochs(365/100) & Loss 82.45328521728516
Epochs(366/100) & Loss 81.96879577636719
Epochs(367/100) & Loss 81.48942565917969
Epochs(368/100) & Loss 81.0150375366211
Epochs(369/100) & Loss 80.54563903808594
Epochs(370/100) & Loss 80.08116149902344
Epochs(371/100) & Loss 79.62149810791016
Epochs(372/100) & Loss 79.16664123535156
Epochs(373/100) & Loss 78.7164535522461
Epochs(374/100) & Loss 78.27107238769531
Epochs(375/100) & Loss 77.8301773071289
Epochs(376/100) & Loss 77.39387512207031
Epochs(377/100) & Loss 76.96209716796875
Epochs(378/100) & Loss 76.53474426269531
Epochs(379/100) & Loss 76.11177062988281
Epochs(380/100) & Loss 75.69317626953125
Epochs(381/100) & Loss 75.27879333496094
Epochs(382/100) & Loss 74.86875915527344
Epochs(383/100) & Loss 74.46283721923828
Epochs(384/100) & Loss 74.06108093261719
Epochs(385/100) & Loss 73.663330078125
Epochs(386/100) & Loss 73.26969146728516
Epochs(387/100) & Loss 72.88002014160156
Epochs(388/100) & Loss 72.49427795410156
Epochs(389/100) & Loss 72.11234283447266
Epochs(390/100) & Loss 71.7343521118164
Epochs(391/100) & Loss 71.36018371582031
Epochs(392/100) & Loss 70.9896469116211
Epochs(393/100) & Loss 70.62281799316406
Epochs(394/100) & Loss 70.25973510742188
Epochs(395/100) & Loss 69.90019226074219
Epochs(396/100) & Loss 69.54423522949219
Epochs(397/100) & Loss 69.1917724609375
Epochs(398/100) & Loss 68.84284973144531
Epochs(399/100) & Loss 68.4973373413086
```

[ ]:
```
preds = model(inputs)
loss = MSE(target, preds)
print(loss)
```

```
tensor(68.1553, grad_fn=<DivBackward0>)
```

[ ]:
```
from math import sqrt
sqrt(loss)
```

[ ]: 8.255619331662308

[ ]:
```
preds
```

[ ]:
```
tensor([[ 58.0557,  72.1434],
        [ 88.5395,  97.2732],
        [102.8608, 137.7344],
        [ 26.3404,  47.4669],
        [109.9683, 107.0769]], grad_fn=<AddBackward0>)
```

[ ]:
```
target
```

[ ]:
```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

[ ]:
```
## You can see they are almost close earch other
```

[ ]:
```

```

1.7 Neural Network using Pytorch

[ ]:
```
# To check GPU
!nvidia-smi
```

```
Wed May 24 08:25:17 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   42C    P8     9W /  70W |      3MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[ ]:
```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt
```

[ ]:
```python
# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz

100%|      | 26421880/26421880 [00:01<00:00, 15982108.83it/s]

Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

100%|      | 29515/29515 [00:00<00:00, 271635.48it/s]

Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 4422102/4422102 [00:00<00:00, 5083357.14it/s]
```

```
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 5148/5148 [00:00<00:00, 6987791.91it/s]

Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
data/FashionMNIST/raw
```

[ ]:
```python
type(training_data)
```

[ ]:
```
torchvision.datasets.mnist.FashionMNIST
```

[ ]:
```python
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    # print(X)
    # print(y)
    break
```

```
Shape of X [N, C, H, W]:  torch.Size([64, 1, 28, 28])
Shape of y:  torch.Size([64]) torch.int64
```

[ ]:
```python
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

```
Using cuda device
```

[ ]:
```python
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
```

```
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

[ ]:
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

[ ]:
```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

[ ]:
```python
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:␣
↪{test_loss:>8f} \n")
```

[ ]:
```python
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

```
Epoch 1
-------------------------------
loss: 2.316672  [    0/60000]
loss: 2.299611  [ 6400/60000]
loss: 2.276056  [12800/60000]
loss: 2.267389  [19200/60000]
loss: 2.250305  [25600/60000]
loss: 2.210238  [32000/60000]
loss: 2.231450  [38400/60000]
loss: 2.190959  [44800/60000]
loss: 2.200532  [51200/60000]
loss: 2.153820  [57600/60000]
Test Error:
 Accuracy: 36.2%, Avg loss: 2.147636

Epoch 2
-------------------------------
loss: 2.171121  [    0/60000]
loss: 2.154280  [ 6400/60000]
loss: 2.090852  [12800/60000]
loss: 2.105507  [19200/60000]
loss: 2.051154  [25600/60000]
loss: 1.985906  [32000/60000]
loss: 2.021355  [38400/60000]
```

```
loss: 1.935196  [44800/60000]
loss: 1.952712  [51200/60000]
loss: 1.864827  [57600/60000]
Test Error:
 Accuracy: 51.2%, Avg loss: 1.863941

Epoch 3
-------------------------------
loss: 1.910603  [    0/60000]
loss: 1.873535  [ 6400/60000]
loss: 1.752214  [12800/60000]
loss: 1.789106  [19200/60000]
loss: 1.676216  [25600/60000]
loss: 1.630203  [32000/60000]
loss: 1.652746  [38400/60000]
loss: 1.553478  [44800/60000]
loss: 1.586449  [51200/60000]
loss: 1.473644  [57600/60000]
Test Error:
 Accuracy: 58.0%, Avg loss: 1.494110

Epoch 4
-------------------------------
loss: 1.571199  [    0/60000]
loss: 1.534655  [ 6400/60000]
loss: 1.384191  [12800/60000]
loss: 1.451482  [19200/60000]
loss: 1.334421  [25600/60000]
loss: 1.332630  [32000/60000]
loss: 1.344593  [38400/60000]
loss: 1.269534  [44800/60000]
loss: 1.310125  [51200/60000]
loss: 1.211971  [57600/60000]
Test Error:
 Accuracy: 61.5%, Avg loss: 1.237874

Epoch 5
-------------------------------
loss: 1.320967  [    0/60000]
loss: 1.302911  [ 6400/60000]
loss: 1.136904  [12800/60000]
loss: 1.240004  [19200/60000]
loss: 1.119631  [25600/60000]
loss: 1.143259  [32000/60000]
loss: 1.161697  [38400/60000]
loss: 1.096488  [44800/60000]
loss: 1.141878  [51200/60000]
loss: 1.062178  [57600/60000]
```

```
Test Error:
 Accuracy: 64.1%, Avg loss: 1.082332

Done!
```

[ ]:
```
#save model
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")
```

```
Saved PyTorch Model State to model.pth
```

[ ]:
```
#load model
model = NeuralNetwork()
model.load_state_dict(torch.load("model.pth"))
```

[ ]: <All keys matched successfully>

[ ]:
```
## Prediction

classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

model.eval()
x, y = test_data[0][0], test_data[0][1]
with torch.no_grad():
    pred = model(x)
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f'Predicted: "{predicted}", Actual: "{actual}"')
```

```
Predicted: "Ankle boot", Actual: "Ankle boot"
```

[ ]: