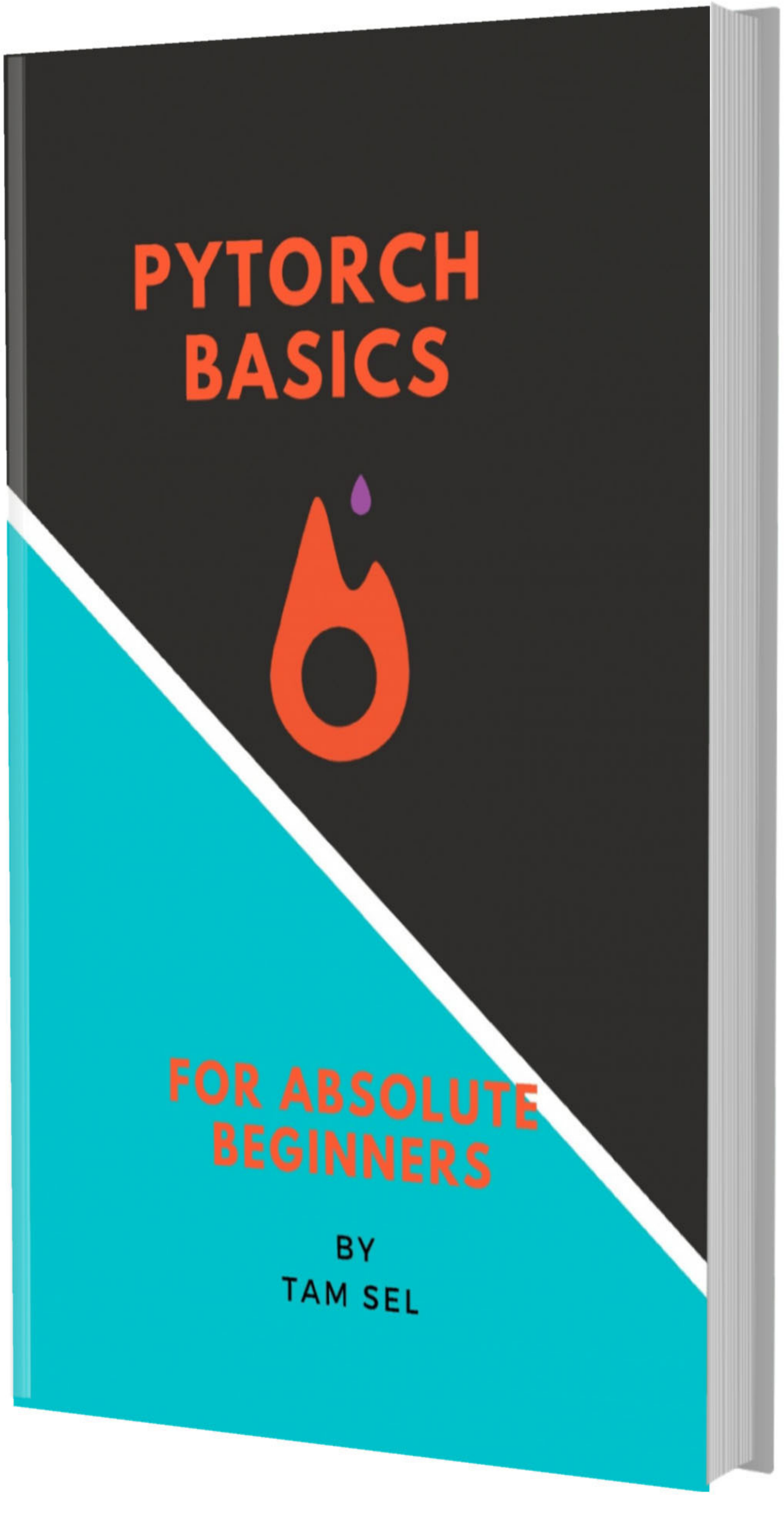
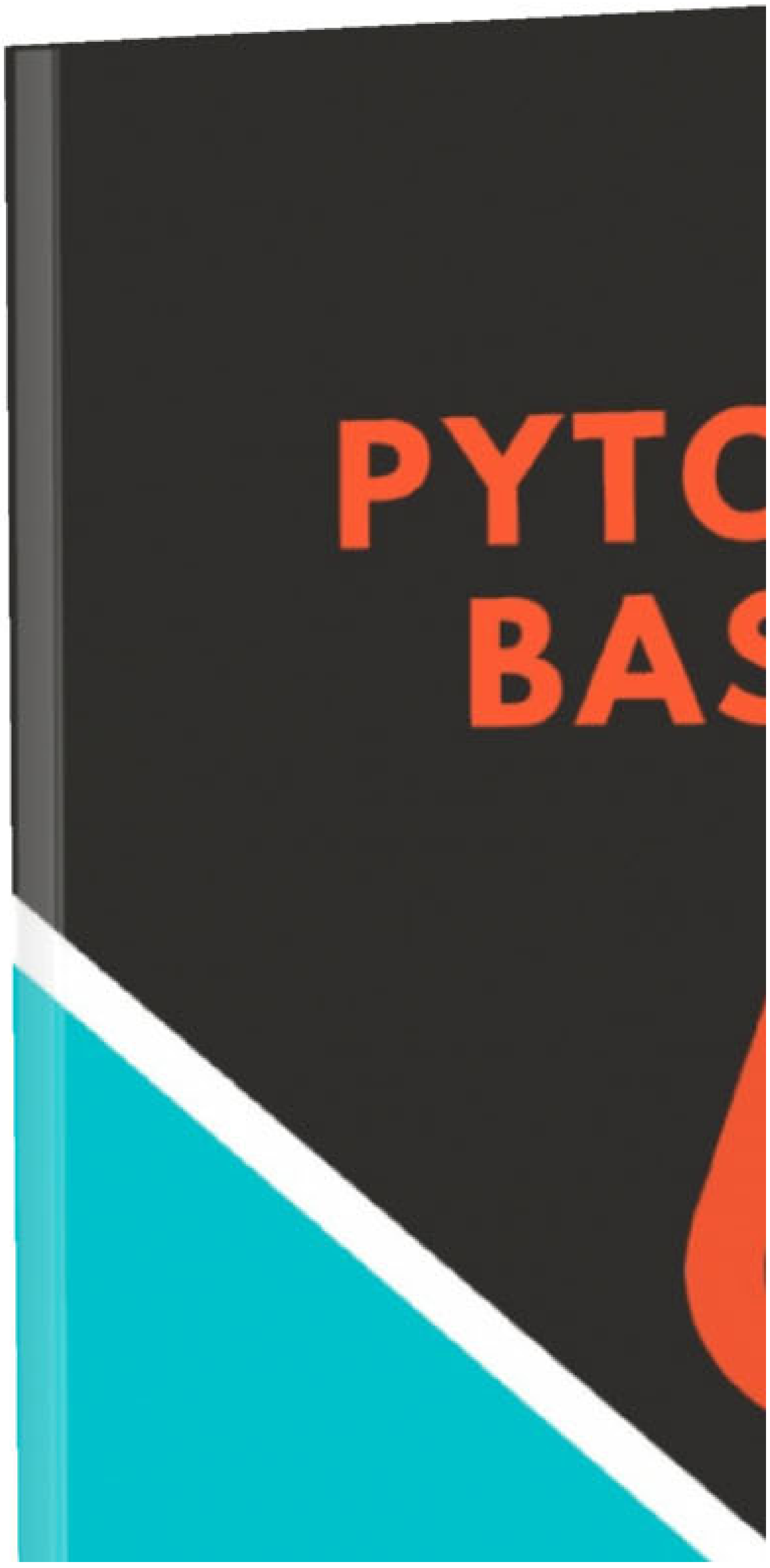


PYTORCH BASICS

FOR ABSOLUTE BEGINNERS

BY

TAM SEL


PYTO
BAS

PYTORCH BASICS FOR ABSOLUTE BEGINNERS

BY

TAM SEL

PYTORCH INTRO

The PyTorch Tutorial is intended for both beginners and experts.

To execute the PyTorch project, you must have a clear understanding of Python.

This tutorial discusses the fundamentals of PyTorch, as well as how deep neural networks are implemented and used in deep learning projects.

Obstacles We guarantee that you will not encounter any difficulties when following this PyTorch tutorial.

PyTorch is a small portion of a computer software that uses the Torch library. It's a Facebook-developed Deep Learning platform. PyTorch is a Python Machine Learning Library for applications such as Natural Language Processing.

The following are the high-level features offered by PyTorch:

1. It provides tensor computing with heavy acceleration thanks to the Graphics Processing Unit (GPU).
2. It offers a Deep Neural Network based on a tape-based auto diff scheme.

PyTorch was created with high versatility and speed in mind when it came to implementing and creating Deep Learning Neural Networks. It is a machine learning library for the Python programming language, so it is very simple to install, run, and understand. Pytorch is totally pythonic (it uses commonly accepted python idioms instead of writing Java or C++ code) and can easily construct a Neural Network Model.

HISTORY

In 2016, PyTorch was released. Many researchers are increasingly able to use PyTorch. Facebook was in charge of the operation. Caffe2 is a Facebook-owned company (Convolutional Architecture for Fast Feature Embedding). It's difficult to convert a PyTorch-defined model to Caffe2. In September 2017, Facebook and Microsoft developed the Open Neural Network Exchange (ONNX) for this reason. Simply put, ONNX was created for the purpose of translating models between frameworks. Caffe2 was merged into PyTorch in March 2018.

Why PyTorch

What's the deal with PyTorch? What makes PyTorch special in terms of building deep learning models? PyTorch is a versatile library that can be used in a number of ways. A dynamic library is a versatile library that you can use according to your needs and changes. It is currently being used by finishers in the Kaggle competition.

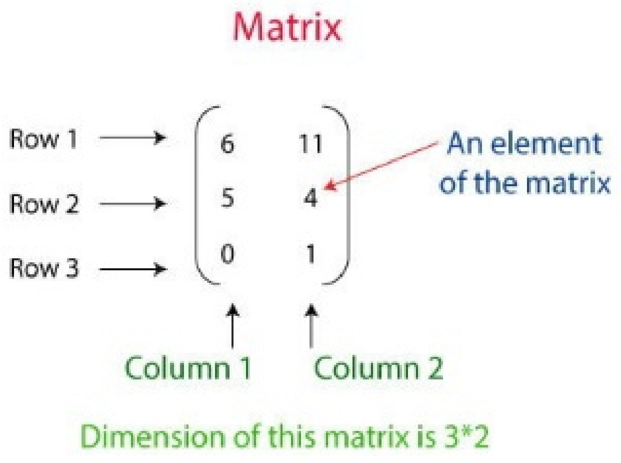
There are numerous features that encourage deep learning scientists to use it in the development of deep learning models.

PyTorch Basics

Understanding all of the fundamental principles needed to work with PyTorch is critical. Tensors are the foundation of PyTorch. Tensor has several tasks to complete. Aside from these, there are a slew of other principles that must be grasped in order to complete the mission.

Matrices or Tensors

Pytorch's main components are tensors. PyTorch can be said to be entirely based on Tensors. A metrics is a rectangular sequence of numbers in mathematical terms. These metrics are known as ndaaray in the Numpy library. Tensor is the name given to it in PyTorch. An n-dimensional data container is known as a Tensor. For example, in PyTorch, a vector is 1d-Tensor, a metrics is 2d-Tensor, a cube is 3d-Tensor, and a cube vector is 4d-Tensor.



The matrices above describe a two-dimensional tensor with three rows and two columns.

Tensor can be generated in three different ways. Tensor is created in a different way by each of them. Tensors are made up of the following elements:

1. Create a PyTorch Tensor array.
2. Make a Tensor with just one digit and a random number.
3. From a numpy array, build a tensor.

Let's take a look at how Tensors are made.

Create a PyTorch Tensor

You must first identify the array, and then move that array as an argument to the torch's Tensor process.

FOR EXAMPLE

```
import torch
arr = [[ 3 , 4 ], [ 8 , 5 ]]
pyTensor = torch.Tensor(arr)
print(pyTensor)
```

OUTPUT

```
tensor ([[3., 4.],[8., 5.]])
```

Create a Tensor

with the random number and all one

You must use the rand() method to construct a random number Tensor, and the ones() method of the torch to create a Tensor of all ones. Another torch tool, manual seed with 0 parameters, will be used with the rand to generate random numbers.

EXAMPLE

```
import torch
ones_t = torch.ones(( 2 , 2 ))
torch.manual_seed( 0 )   //to have same values for random generation
rand_t = torch.rand(( 2 , 2 ))
print(ones_t)
print(rand_t)
```

OUTPUT

```
Tensor ([[1., 1.],[1., 1.]])
tensor ([[0.4963, 0.7682],[0.0885, 0.1320]])
```

Create a Tensor

from numpy array

We must first construct a numpy array before we can create a Tensor from it. After you've built your numpy array, you'll need to pass it as an argument to from numpy(). The numpy array is converted to a Tensor using this form.

EXAMPLE

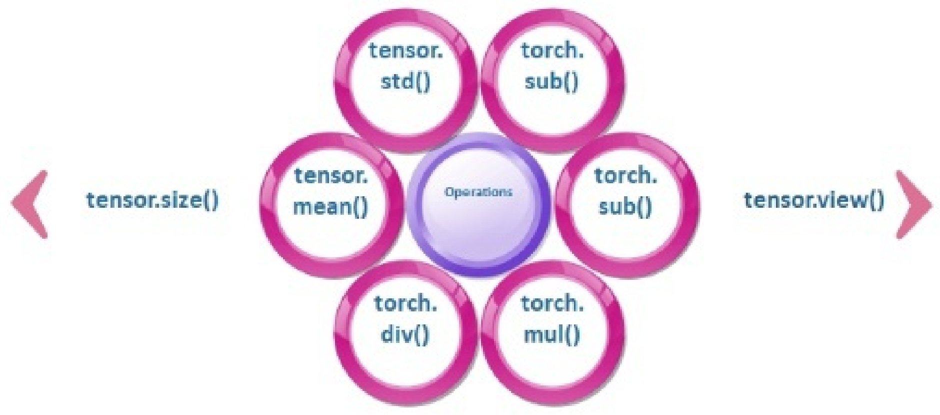
```
import torch
import numpy as np1
numpy_arr = np1.ones(( 2 , 2 ))
pyTensor = torch.from_numpy(numpy_arr)
np1_arr_from_Tensor = pyTensor.numpy()
print(np1_arr_from_Tensor)
```

OUTPUT

```
[[1. 1.] [1. 1.]]
```

Tensors Operations

Since tensors are identical to arrays, any operation that can be performed on an array can also be performed on a tensor.



Resizing a Tensor

The size property of Tensor can be used to resize the Tensor. Tensor.view() is used to resize a Tensor. Resizing a Tensor entails converting a 2*2 dimensional Tensor to a 4*1 dimensional Tensor, or a 4*4 dimensional Tensor to a 16*1 dimensional Tensor, and so on. The Tensor.size() method is used to print the Tensor size.

Let's take a look at how to resize a Tensor.

```
import torch
pyt_Tensor = torch.ones(( 2 , 2 ))
print(pyt_Tensor.size())       # shows the size of this Tensor
pyt_Tensor = pyt_Tensor.view( 4 ) # resizing 2x2 Tensor to 4x1
print(pyt_Tensor)
```

OUTPUT

```
torch.Size ([2, 2])
tensor ([1., 1., 1., 1.])
```

Mathematical Operations

Tensor can perform all mathematical operations such as addition, subtraction, division, and multiplication. The mathematical procedure can be done by the torch. Tensor operations are performed using the functions torch.add(), torch.sub(), torch.mul(), and torch.div().

Consider the following example of mathematical operations:

```
import numpy as np
import torch
Tensor_a = torch.ones(( 2 , 2 ))
Tensor_b = torch.ones(( 2 , 2 ))
result=Tensor_a+Tensor_b
result1 = torch.add(Tensor_a, Tensor_b)      //another way of addidtion
Tensor_a.add_(Tensor_b) // In-place addition
print(result)
print(result1)
print(Tensor_a)
```

OUTPUT

```
tensor ([[2., 2.], [2., 2.]])
tensor ([[2., 2.], [2., 2.]])
```

Mean and Standard-deviation

Tensor's standard deviation can be calculated in two ways: one-dimensional and multi-dimensional. We must first calculate the mean in our mathematical equation, and then apply the following formula to the given data with mean.

$$s = \frac{1}{N}\sum_{i=1}^{N}(X_i - \mu_i)^2$$

However, we can find the variance and mean of a Tensor using Tensor.mean() and Tensor.std() in Tensor.

Let's look at an example of how it worked.

```
import torch
pyTensor = torch.Tensor([ 1 , 2 , 3 , 4 , 5 ])
```

```
mean = pyt_Tensor.mean(dim= 0 )         //if multiple rows then dim = 1
std_dev = pyTensor.std(dim= 0 )        // if multiple rows then dim = 1
print(mean)
print(std_dev)
```

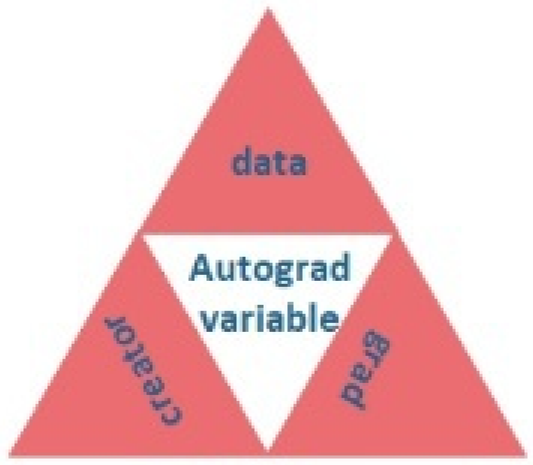
OUTPUT

```
tensor (3.)
tensor (1.5811)
```

Variables and Gradient

The package's key class is autograd.variable. The key purpose of this program is to wrap a Tensor. It can perform almost all of the operations that are specified on it. You can use.backword() to calculate the gradient only after you've finished your computation.

The row Tensor can be accessed via the.data attribute, while the gradient for this variable is accumulated in the.grad attribute.



The estimation of gradients is crucial in deep learning. In PyTorch, variables are used to measure the gradient. Variables are a wrapper around Tensors with gradient calculation features, to put it simply.

The python code for controlling variables is shown below.

```
import numpy as np
import torch
from torch.autograd import Variable
pyt_var = Variable(torch.ones(( 2 , 2 )), requires_grad = True)
```

Since the above code behaves similarly to Tensors, we may apply all operations in the same way.

Let's take a look at how we can compute the gradient in PyTorch.

EXAMPLE

```
import numpy as np
import torch
from torch.autograd import Variable
// let's consider the following equation
// y = 5(x + 1)^2
x = Variable (torch.ones( 1 ), requires_grad = True)
y = 5 * (x + 1 ) ** 2          //implementing the equation.
y.backward()                   // calculate gradient
print(x.grad)                  // get the gradient of variable x
# differentiating the above mentioned equation
// => 5(x + 1)^2 = 10(x + 1) = 10(2) = 20
```

OUTPUT

```
tensor([20.])
```

PyTorch vs. TensorFlow

Origin

PyTorch is a Python-based machine learning library based on the Torch library.

It was created by Facebook's artificial intelligence research group for use in deep learning and natural language processing.

It is open-source software that is distributed under the Modified BSD license.

Tensor flow is an open-source machine learning system that was created by Google.

Features

PyTorch has a number of appealing features, including:

1. Python has native support.
2. Graphs of dynamic computation
3. CUDA support is available.

These features reduce the amount of time it takes to run the code and improve performance. TensorFlow, on the other hand, has some unique and appealing features, such as TensorBoard, which would be a fantastic tool for visualizing a machine learning model. It also includes TensorFlow Serving, a specialized grpc server that is used during deployment and development.

Community

TensorFlow has a much wider community than PyTorch. Many researchers in different fields, such as business organizations, academics, and others, use TensorFlow. It is easier to locate tools or solutions in TensorFlow. In TensorFlow and PyTorch, there are a plethora of tutorials, codes, and help.

Level of API

If we're talking about APIs, TensorFlow is the best choice because it offers both high-level and low-level APIs. PyTorch has a lower-level API that focuses on working with array expressions directly. PyTorch has gotten a lot of attention in the last year, and it's quickly becoming the go-to tool for academic research and deep learning applications that involve custom expression optimization.

Speed

The most prominent deep learning frameworks are PyTorch and TensorFlow.

If we're working from home and introducing our first deep learning project, PyTorch is a good fit. However, when we work in our office, we use TensorFlow, and we have extensive experience with deep learning projects. When we compare the speeds of PyTorch and TensorFlow, we can see that both frameworks have a similar speed, which is fast and appropriate for efficiency.

Coverage

Certain operations, such as flipping a tensor along a dimension, checking a tensor for Nan and infinity, and Fast Fourier transforms, are natively supported by TensorFlow.

It also contains the contrib kit, which is used to build additional templates.

It facilitates the use of higher-level functionality and provides us with a diverse set of options to work with.

PyTorch is still lacking a few features, but thanks to all of the publicity, it will be bridged soon. PyTorch isn't as well-known among students and freelancers as TensorFlow.

Ramp-Up Time

PyTorch is a GPU-enabled NumPy drop-in replacement that provides higher-level functionality for deep neural network building and training.

PyTorch is simple to learn if we are familiar with Python, NumPy, and deep learning abstraction. When we write TensorFlow code, Python will "compile" it into a graph, which will then be executed by the TensorFlow execution engine.

TensorFlow has a few additional concepts to master, including the graph, session, placeholder, and variable scoping. TensorFlow's ramp-up period is significantly longer than PyTorch's.

Popularity

TensorFlow is more commonly used than PyTorch due to its popularity.

Any organization requires a simple and readable architecture that can handle large datasets quickly.

PyTorch is a newer version of TensorFlow that has exploded in popularity.

TensorFlow does not allow for customization, while PyTorch does.

The most GitHub operation, Google searches, Medium posts, Amazon books, and ArXiv articles are all linked to TensorFlow. Many developers use it, and it's described in almost all job descriptions online.

Custom Extensions

Both frameworks allow you to bind or create custom extensions that are written in C, C++, or CUDA. Of course, TensorFlow needs more boilerplate code, but it is simpler and supports a broader variety of types and devices.

We may simply write an interface and implementation for specific CPU and GPU versions in PyTorch.

For both frameworks, compiling custom extensions is a breeze (PyTorch and TensorFlow).

Outside of the pip installation, there is no need to download any headers or source code.

Tensors Introduction

Pytorch's main components are tensors. PyTorch can be said to be entirely dependent on Tensors.

A rectangular sequence of numbers is known as metrics in mathematics.

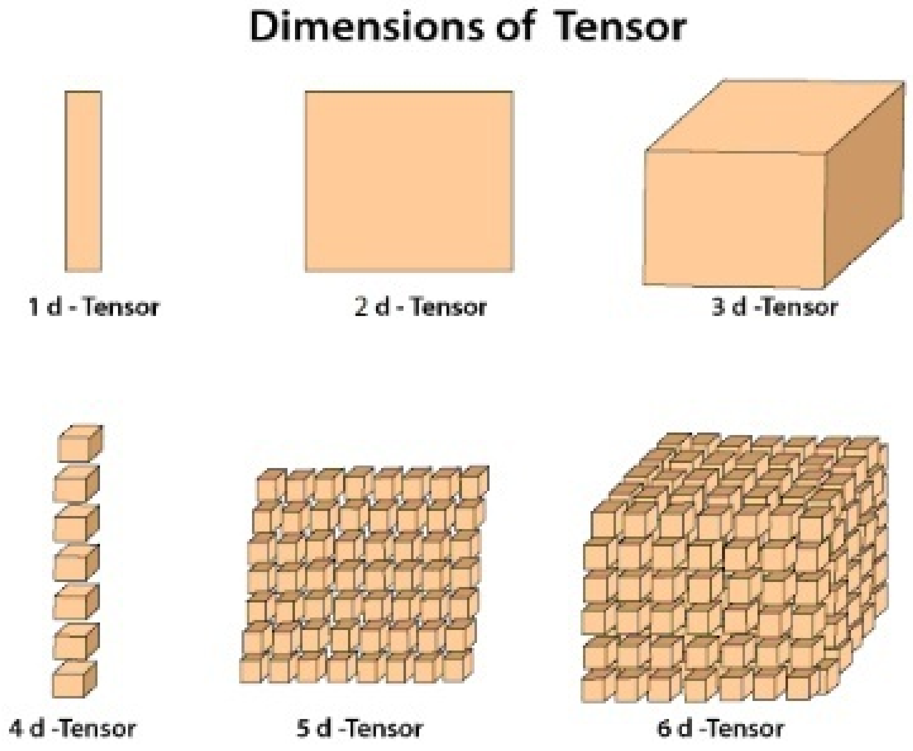
These metrics are known as ndaaray in the NumPy library. Tensor is the name given to it in PyTorch. An n-dimensional data container is known as a tensor. For example, in PyTorch, a vector is 1d-tensor, a metrics is 2d-tensor, a cube is 3d-tensor, and a cube vector is 4d-tensor.

Torch is a tensor computation library with excellent GPU acceleration.

To function with PyTorch, we must first become acquainted with the tensor data structure. It will be a necessary precondition for the implementation of neural networks.

Tensor is a critical component of deep learning, and there has been a lot of speculation about it.

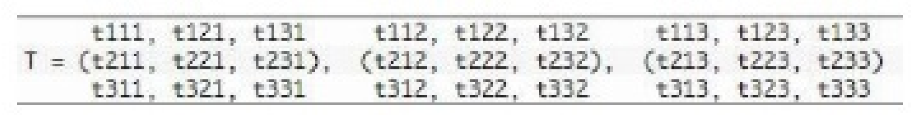
It's also in the name of TensorFlow, Google's key machine learning library.



Let's read a little bit about tensor notation.

The tensor notation resembles the metrics notation in appearance.

The tensor is represented by a capital letter, and scalar values within the tensor are represented by a lower letter with a subscript integer.



```
     t111, t121, t131      t112, t122, t132      t113, t123, t133
T = (t211, t221, t231),   (t212, t222, t232),   (t213, t223, t233)
     t311, t321, t331      t312, t322, t332      t313, t323, t333
```

How create Tensor

The Tensor can be formed in three different ways. Each one generates a Tensor in a different way and employs a different process. Tensors are a type of tensor.

From an array, create a Tensor.

Make a tensor out of all ones and a random number.

From a numpy array, build a tensor.

Create Tensor from an array

You must first identify the array, and then transfer that array as an argument to the torch's Tensor process.

FOR EXAMPLE

```
Import torch
arr = [[ 3 , 4 ], [ 8 , 5 ]]
pytensor = torch.Tensor(arr)
print(pytensor)
```

OUTPUT

```
tensor ([[3., 4.],[8., 5.]])
```

Create Tensor

with the random

number and all one

To create a tensor with a random number, we must use the torch's method rand(), and to create a tensor with all ones, we must use the torch's method ones (). Another torch method, manual seed with 0 parameters, will be used with rand to generate random numbers.

FOR EXAMPLE

```
import torch
ones_t = torch.ones(( 2 , 2 ))
torch.manual_seed( 0 )   //to have same values for random generation
rand_t = torch.rand(( 2 , 2 ))
print(ones_t)
print(rand_t)
```

OUTPUT

```
tensor([[1., 1.],[1., 1.]])
tensor([[0.4962, 0.7682],[0.0885, 0.1320]])
```

Create a Tensor

from numpy array

We must first create a numpy array before we can create a tensor from it.

After you've built your numpy array, you'll need to pass it as an argument to from numpy().

The numpy array is converted to a tensor using this form.

FOR EXAMPLE

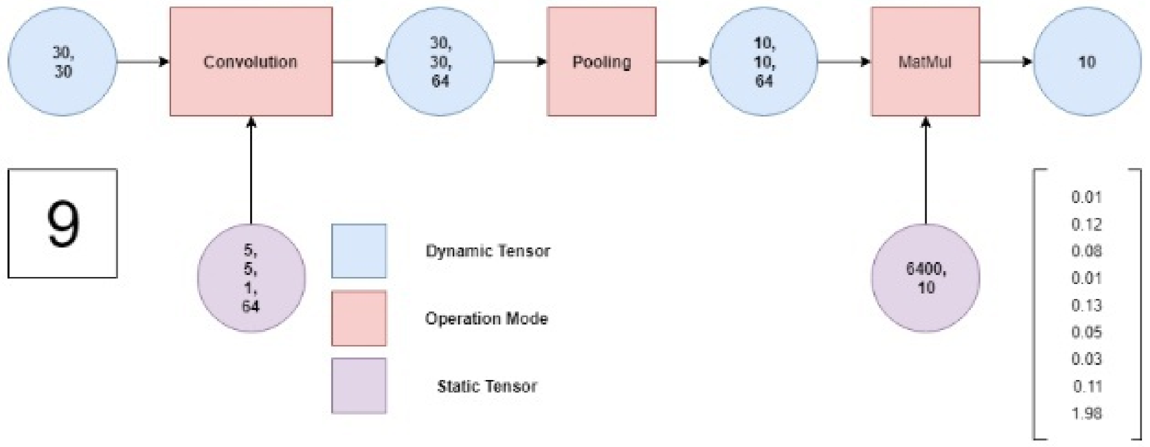
```
import torch
import numpy as np1
numpy_arr = np1.ones(( 2 , 2 ))
pytensor = torch.from_numpy(numpy_arr)
np1_arr_from_tensor = pytensor.numpy()
print(np1_arr_from_tensor)
```

OUTPUT

```
[[1. 1.][1. 1.]]
```

One Dimensional Tensors

PyTorch, as we all know, has been adopted by the Deep Learning community for its ease of use in defining neural networks. Sensors are at the heart of neural networks, and PyTorch is built around them as well. There is usually a substantial increase in efficiency. A tensor is a generalization of matrices in the broadest sense.



1D-Tensor is the same as 1D-matrix. There is just one row and one column in a one-dimensional Tensor, which is known as vector. There is also a scalar, which is a zero-dimensional tensor.

Now we'll look at how to perform operations on tensors.

We can also use Google Colab to write Tensor code. It's very easy to use Google Colab. There is no need to set up Google Colab. It is completely cloud-based.

Jupyter Notebook and Google Colab are close. When we use Google Colab, we get several packages pre-installed. Unfortunately, the torch isn't one of them, so we'll need to use the!pip3 install torch command to get it mounted.

We'll now use a one-dimensional Tensor to perform the process.

Creating one-dimensional Tensor

The tensor property of the torch library is used to build a one-dimensional Tensor. The torch.tensor() method is used to construct a tensor.

Syntax

n= torch.tensor([Tensor elements])

EXAMPLE

```
import torch
n=torch.tensor([ 1 , 2 , 3 , 4 ])
print(n)
```

OUTPUT

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

Checking data type

of elements in Tensor

The data type of the element contained in Tensor can be verified. To determine the data form, we use Tensor's dtype() process.

EXAMPLE

```
import torch
n=torch.tensor ([ 1.0 , 2.0 , 3.0 ])
print (n.dtype)
```

OUTPUT

```
torch.float32
```

Accessing of Tensor's elements

The index of a Tensor element can be used to access the elements of that Tensor. We can print the tensor variable if we want to print all of Tensor's components. Tensor index, like one-dimensional metrics index, begins at 0.

EXAMPLE

```
import torch
n=torch.tensor([ 1.0 , 2.0 , 3.0 ])
print(n[ 2 ])
```

OUTPUT

```
tensor(3.)
```

Accessing of Tensor's elements

with the specified range

By passing the beginning and ending indexes of elements separated by a colon, you can easily access elements within a given range (:). It will skip the first index element and print elements before the last index element is reached.

EXAMPLE

```
import torch
n=torch.tensor([ 1.0 , 2.0 , 3.0 ])
print(n[ 0 : 2 ])
```

OUTPUT

```
tensor (2.0,3.0)
```

Another example skips the starting index, which is initialized by us, and prints all elements.

EXAMPLE

```
import torch
n=torch.tensor ([ 1.0 , 2.0 , 3.0 ])
print(n[ 0 :])
```

OUTPUT

```
tensor (2.0,3.0)
```

Creating of Floating Point Tensor

using Integer elements

Using integer components, we can make a floating point Tensor. The FloatTensor property of the torch is used in this.

EXAMPLE

```
import torch
n=torch.FloatTensor([ 1 , 2 , 3 , 4 , 5 , 6 , 7 ])
print(n)
```

OUTPUT

```
tensor([1., 2., 3., 4., 5., 6., 7.])
```

Finding size of the Tensor

We can find the size of Tensor in the same way as we can find the size of one-dimensional metrics. To get the scale, we use Tensor's size() process.

EXAMPLE

```
import torch
n=torch.FloatTensor([ 1 , 2 , 3 , 4 , 5 , 6 , 7 ])
print(n.size())
```

OUTPUT

```
torch.Size([7])
```

Change view of Tensor

Tensor has a property that allows one to adjust the Tensor's view. If a tensor is one dimensional (one row and one column) and we want to change its view by six rows and one column, we call it changing view. Tensor's view() function can be used to make changes. It's close to the array's reshape ().

EXAMPLE

```
import torch
n=torch.FloatTensor([ 1 , 2 , 3 , 4 , 5 , 6 ])
print(n)
n.view( 6 , 1 )
```

OUTPUT

```
tensor ([1., 2., 3., 4., 5., 6.])
tensor([[1.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.]])
```

Tensor using numpy array

Tensor can also be created with a numpy array. With the assistance of the torch's from numpy (), we must transform the numpy array into a Tensor. To do so, we must first initialize numpy before creating a numpy list.

EXAMPLE

```
import torch
import numpy as np
a=np.array([ 1 , 2 , 3 , 4 , 5 , 6 ])
tensorcon=torch.from_numpy(a)
print(tensorcon)
print(tensorcon.type())
```

OUTPUT

```
tensor([1, 2, 3, 4, 5, 6])
torch.LongTensor
```

Vector Operations

We know that Tensors come in a variety of dimensions, like zero, one, and multi-dimensional.

Vectors are one-dimensional tensors that can be manipulated using a variety of operations.

Mathematical operations, dot product, and linspace are examples of vector operations.

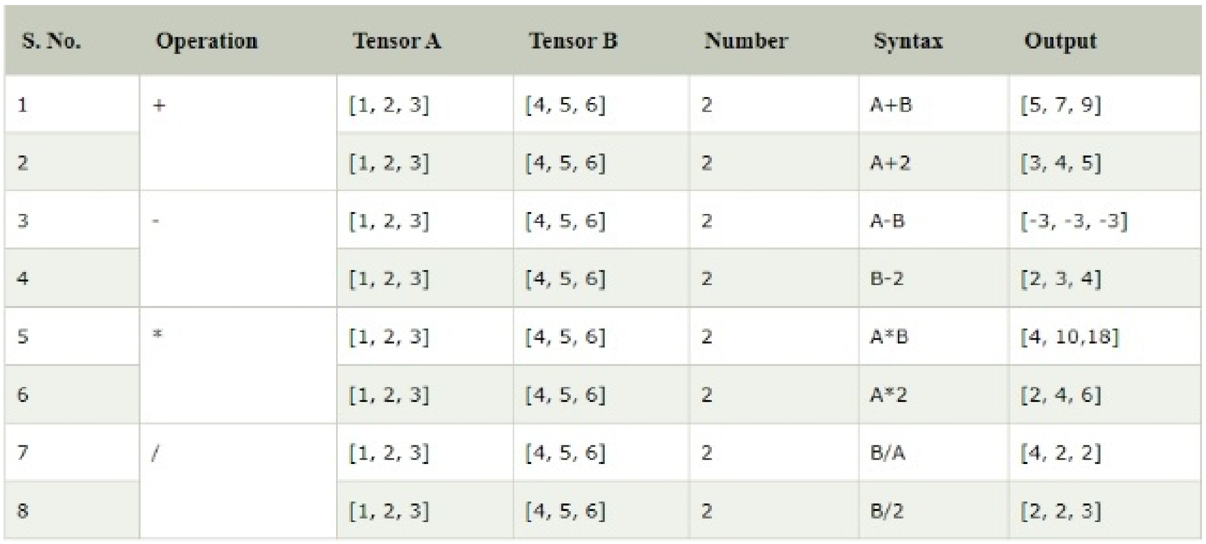
Deep learning relies heavily on vectors.

With the aid of vectors or one-dimensional tensors, we generate random points in a deep learning neural network.

On the vector, the operations mentioned below are carried out.

Mathematical Operations

The tensor can be added to, subtracted from, multiplied by, and divided by another tensor. The table below lists all mathematical operations that are performed on vectors and their predicted results.



S. No.	Operation	Tensor A	Tensor B	Number	Syntax	Output
1	+	[1, 2, 3]	[4, 5, 6]	2	A+B	[5, 7, 9]
2		[1, 2, 3]	[4, 5, 6]	2	A+2	[3, 4, 5]
3	-	[1, 2, 3]	[4, 5, 6]	2	A-B	[-3, -3, -3]
4		[1, 2, 3]	[4, 5, 6]	2	B-2	[2, 3, 4]
5	*	[1, 2, 3]	[4, 5, 6]	2	A*B	[4, 10,18]
6		[1, 2, 3]	[4, 5, 6]	2	A*2	[2, 4, 6]
7	/	[1, 2, 3]	[4, 5, 6]	2	B/A	[4, 2, 2]
8		[1, 2, 3]	[4, 5, 6]	2	B/2	[2, 2, 3]

```
import torch
A=torch.tensor([ 1 , 2 , 3 ])
```

```
B=torch.tensor([ 4 , 5 , 6 ])
A+B
A+ 2
A-B
B- 2
A*B
A* 2
B/A
B/ 2
```

OUTPUT

```
tensor([5, 7, 9])
tensor([3, 4, 5])
tensor([-3, -3, -3])
tensor([2, 3, 4])
tensor([ 4, 10, 18])
tensor([2, 4, 6])
tensor([4, 2, 2])
tensor([2, 2, 3])
```

Dot Product and linspace

We can also compute the dot product of two tensors. To measure, we use the torch's dot() form, which gives us the exact or expected result.

There's also linspace, which is a vector operation.

We use the linspace form for linspace ().

There are two parameters in this system.

The first number is the start, and the second is the end.

This method prints a hundred uniformly spaced numbers from the beginning to the end.

EXAMPLE

```
import torch
t1= torch.tensor([ 1 , 2 , 3 ])
```

```
t2= torch.tensor([ 4 , 5 , 6 ])
DotProduct= torch.dot(t1,t2)
print(DotProduct)
torch.linspace( 2 , 9 )
```

OUTPUT

```
tensor(32)

tensor([2.0000, 2.0707, 2.1414, 2.2121, 2.2828, 2.3535, 2.4242, 2.4949,
2.5657,
        2.6364, 2.7071, 2.7778, 2.8485, 2.9192, 2.9899, 3.0606, 3.1313,
3.2020,
        3.2727, 3.3434, 3.4141, 3.4848, 3.5556, 3.6263, 3.6970, 3.7677,
3.8384,
        3.9091, 3.9798, 4.0505, 4.1212, 4.1919, 4.2626, 4.3333, 4.4040,
4.4747,
        4.5455, 4.6162, 4.6869, 4.7576, 4.8283, 4.8990, 4.9697, 5.0404,
5.1111,
        5.1818, 5.2525, 5.3232, 5.3939, 5.4646, 5.5354, 5.6061, 5.6768,
5.7475,
        5.8182, 5.8889, 5.9596, 6.0303, 6.1010, 6.1717, 6.2424, 6.3131,
6.3838,
        6.4545, 6.5253, 6.5960, 6.6667, 6.7374, 6.8081, 6.8788, 6.9495,
7.0202,
        7.0909, 7.1616, 7.2323, 7.3030, 7.3737, 7.4444, 7.5152, 7.5859,
7.6566,
        7.7273, 7.7980, 7.8687, 7.9394, 8.0101, 8.0808, 8.1515, 8.2222,
8.2929,
        8.3636, 8.4343, 8.5051, 8.5758, 8.6465, 8.7172, 8.7879, 8.8586,
8.9293,
        9.0000])
```

Plotting a function

on the two-dimensional

coordinate system

When plotting a function on two-dimensional coordinate systems, the linspace function can be useful.

We'll make a land space from 0 to 10 with a 2.5-inch interval for the x-axis, and Y will be the function of each x value.

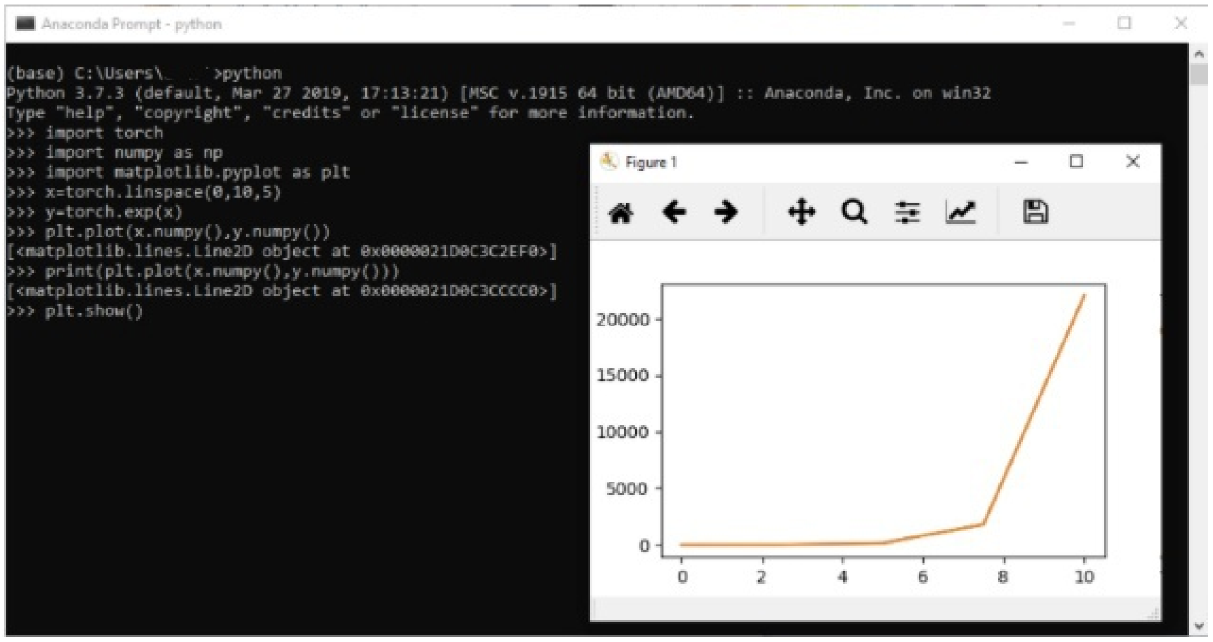
We can find the exponential of each x value for y, for example.

Now we're plotting x and y data with the Map plot lib library, which is a data visualization library.

EXAMPLE

```
import torch
import numpy as np
import matplotlib.pyplot as plt
x=torch.linspace( 0 , 10 , 100 )
y=torch.exp(x)
plt.plot(x.numpy(),y.numpy())
plt.show()
```

OUTPUT

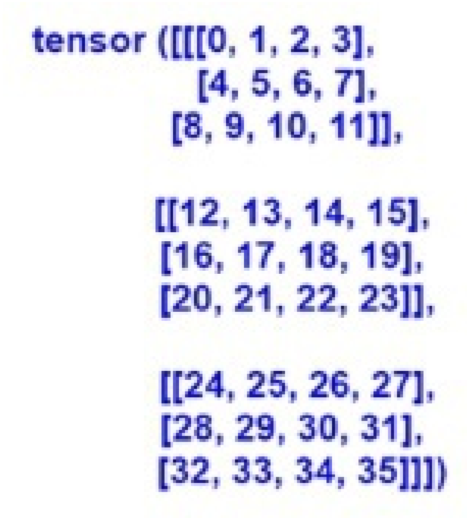


```
(base) C:\Users\[redacted]>python
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x=torch.linspace(0,10,5)
>>> y=torch.exp(x)
>>> plt.plot(x.numpy(),y.numpy())
[<matplotlib.lines.Line2D object at 0x0000021D0C3C2EF0>]
>>> print(plt.plot(x.numpy(),y.numpy()))
[<matplotlib.lines.Line2D object at 0x0000021D0C3CCCC0>]
>>> plt.show()
```

Two Dimensional Tensor

Two-dimensional tensors are similar to two-dimensional metrics in that they have two dimensions. The number of rows and columns in a two-dimensional metric is n. A two-dimensional tensor has the same number of rows and columns as a one-dimensional tensor.

The representation of a two-dimensional tensor is as follows:

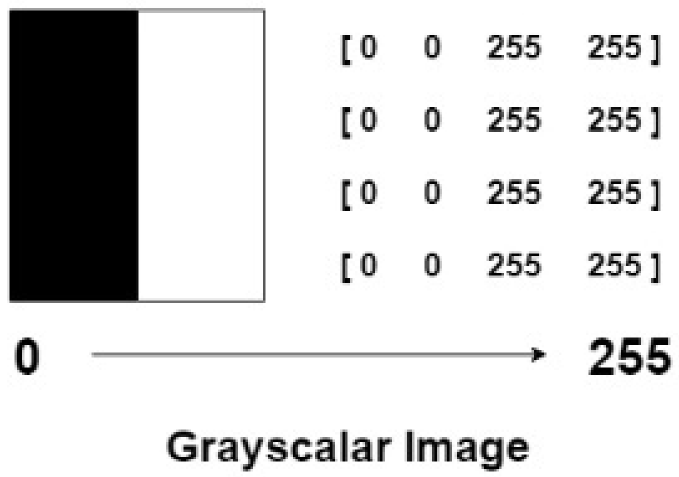


```
tensor ([[[0, 1, 2, 3],
          [4, 5, 6, 7],
         [8, 9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]],

        [[24, 25, 26, 27],
         [28, 29, 30, 31],
         [32, 33, 34, 35]]])
```

A two-dimensional matrix of pixels makes up a gray scalar image. The intensity of each pixel is represented by a numeric value that varies from 0 to 255, with a value of 0 indicating no intensity and 255 indicating maximum intensity. This two-dimensional grid of values can be saved.



```
[0   0   255   255]
[0   0   255   255]
[0   0   255   255]
[0   0   255   255]
```

Creating two-dimensional tensor

To construct a two-dimensional tensor, use the torch's arrange () method to create a one-dimensional tensor first. There are two integer-type parameters in this procedure. This method arranges the elements in a tensor according to the parameters given. After you've generated your one-dimensional tensor, the next move is to convert it to a two-dimensional view and store it in a two-dimensional variable.

Let's look at an example of a two-dimensional tensor.

```
import torch
x=torch.arange( 0 , 9 )
x
y=x.view( 3 , 3 )
y
```

OUTPUT

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

Note: The dim() method of tensor used to determine the dimension of the tensor.

```
import torch
x=torch.arange( 0 , 9 )
x
y=x.view( 3 , 3 )
y
x.dim()
y.dim()
```

OUTPUT

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
tensor([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]])
1
2
```

Accessing two-dimensional tensor elements

Let's look at a two-dimensional tensor example to see how to use index to access a specific element in a two-dimensional tensor.

EXAMPLE

```
import torch
x=torch.arange( 0 , 9 )
x
y=x.view( 3 , 3 )
y
y[ 0 , 2 ]
```

OUTPUT

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
tensor([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]])
tensor(2
```

Tensors Multiplication

The multiplication is carried out in the same way as metrics multiplication is carried out. Multiplying the corresponding row with the corresponding column is how tensor multiplication is performed. The deep learning model relies heavily on tensor multiplication. One-dimensional tensors, two-dimensional tensors, three-dimensional tensors, and so on. Tensor multiplication is only achieved with tensors of compatible dimension.

Let's look at a Tensor Multiplication example.

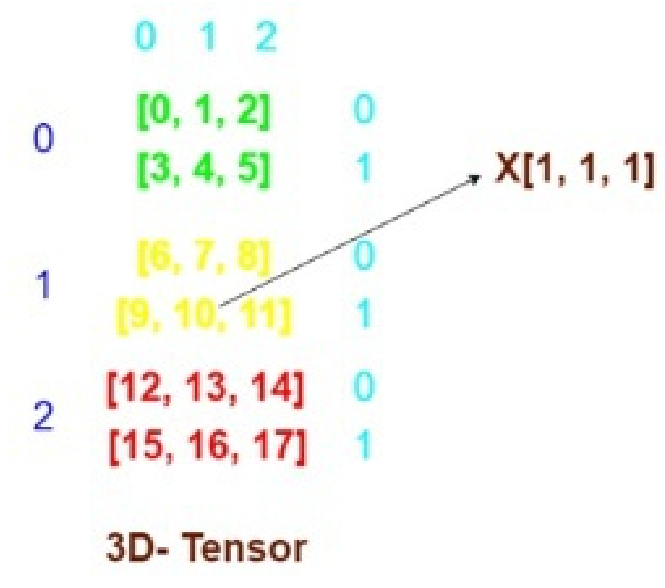
```
import torch
mat_a=torch.tensor([ 1 , 3 , 5 , 7 , 9 , 2 , 4 , 6 , 8 ])
mat_a=mat_a.view( 3 , 3 )
mat_b=torch.tensor([ 1 , 3 , 5 , 7 , 9 , 2 , 4 , 6 , 8 ])
mat_b=mat_b.view( 3 , 3 )
mat_a
mat_b
torch.matmul(mat_a,mat_b)# We can also usemat_a @ mat_b
```

OUTPUT

```
tensor([[1, 3, 5],
        [7, 9, 2],
        [4, 6, 8]])
tensor([[1, 3, 5],
        [7, 9, 2],
        [4, 6, 8]])
tensor([[ 42,  60,  51],
        [ 78, 114,  69],
        [ 78, 114,  96]])
```

Three Dimensional Tensor

The view () method is used to build a three-dimensional tensor. The structure of a three-dimensional tensor is as follows:



Accessing element from 3D- Tensor

It's easy to get elements from the 3D-tensor. It will be done with the aid of the index.

EXAMPLE

```
import torch
x=torch.arange( 18 )
y=x.view( 3 , 2 , 3 )
y
y[ 1 , 1 , 1 ]
```

OUTPUT

```
tensor([[[ 0,  1,  2],
         [ 3,  4,  5]],
        [[ 6,  7,  8],
         [ 9, 10, 11]],
        [[12, 13, 14],
         [15, 16, 17]]])
tensor(10
```

Slicing of three-dimensional tensor

The way we slice a one-dimensional tensor is very close to how we slice segment slices. Slicing a tensor refers to the method of splitting a tensor into its constituent elements to construct a new tensor.

EXAMPLE

Let's assume we have a three-dimensional tensor with elements ranging from 0 to 17 that we want to slice from 6 to 11.

```
import torch
x=torch.arange( 18 )
y=x.view( 3 , 2 , 3 )
y
y[ 1 , 0 : 2 , 0 : 3 ]      # can also apply y[ 1 , :, :]
```

OUTPUT

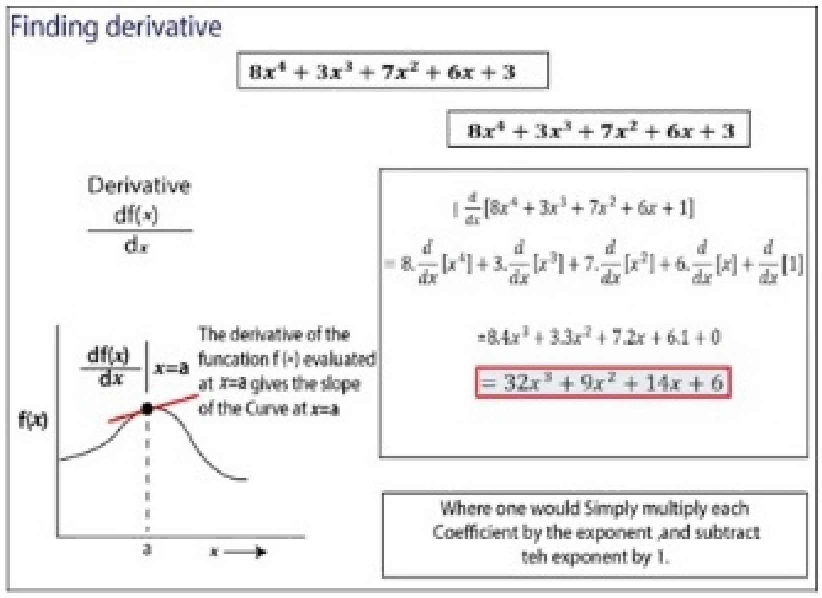
```
tensor([[[ 0,  1,  2],
          [ 3,  4,  5]],
         [[ 6,  7,  8] ,
          [ 9, 10, 11]],
         [[12, 13, 14],
          [15, 16, 17]]])
tensor([[ 6,  7,  8],
[ 9, 10, 11]])
```

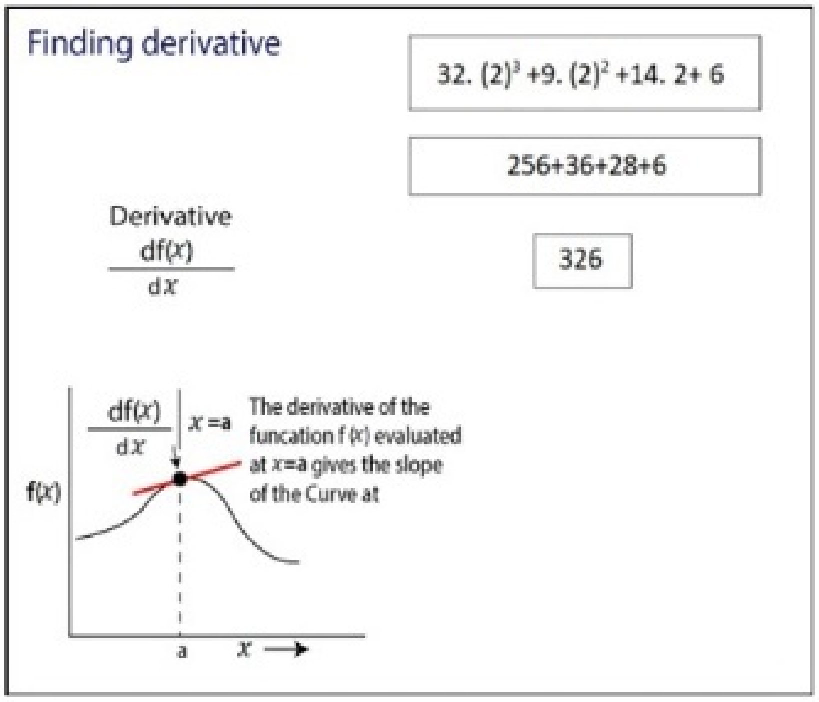
Gradient with PyTorch

In this part, we'll go over derivatives and how to use them in PyTorch. So let's get started.

The gradient is used to locate the function's derivatives. In mathematics, derivatives refer to the partial differentiation of a function and the determination of its value.

The diagram below illustrates how to measure a function's derivative.


Finding derivative
$8x^4 + 3x^3 + 7x^2 + 6x + 3$
$8x^4 + 3x^3 + 7x^2 + 6x + 3$
Derivative $\frac{df(x)}{dx}$
$\frac{d}{dx}[8x^4 + 3x^3 + 7x^2 + 6x + 1]$
$= 8.\frac{d}{dx}[x^4] + 3.\frac{d}{dx}[x^3] + 7.\frac{d}{dx}[x^2] + 6.\frac{d}{dx}[x] + \frac{d}{dx}[1]$
$= 8.4x^3 + 3.3x^2 + 7.2x + 6.1 + 0$
$= 32x^3 + 9x^2 + 14x + 6$
$\frac{df(x)}{dx}\Big|_{x=a}$
The derivative of the funcation f (•) evaluated at x=a gives the slope of the Curve at x=a
f(x)
a
x
Where one would Simply multiply each Coefficient by the exponent ,and subtract teh exponent by 1.



The work we did in the diagram above will be repeated in PyTorch with gradient. The function's derivative must be found in the next step.

1. First, we must initialize the function (y=3x3 +5x2+7x+1) for which the derivatives will be calculated.

2. The next step is to set the value of the function's variable. The following is how the value of x is set.

```
X= torch.tensor ( 2.0 , requires_grad=True)
```

To find the derivative of a function, we usually need a gradient.

3. Finally, using the backward () form, calculate the function's derivative.

4. The value of the derivative is accessed or printed using grad as the final stage.

Let's take a look at an example of finding derivatives.

```
import torch
x=torch.tensor( 2.0 , requires_grad=True)
y= 8 *x** 4 + 3 *x** 3 + 7 *x** 2 + 6 *x+ 3
y.backward()
```

```
x.grad
```

OUTPUT

```
tensor(326.)
```

EXAMPLE 2

```
import torch
x=torch.tensor( 2.0 , requires_grad=True)
z=torch.tensor( 4.0 , requires_grad=True)
y=x** 2 +z** 3
y.backward()
x.grad
z.grad
```

OUTPUT

```
tensor(4.)
tensor(48.)
```

Linear Regression

By decreasing the distance between the dependent and independent variables, linear regression can be used to find a linear relationship between them.

Linear regression is a method of supervised machine learning.

This method is used to classify order discrete categories. We'll learn how to construct a model that allows a consumer to predict the relationship between the dependent and independent variables in this section.

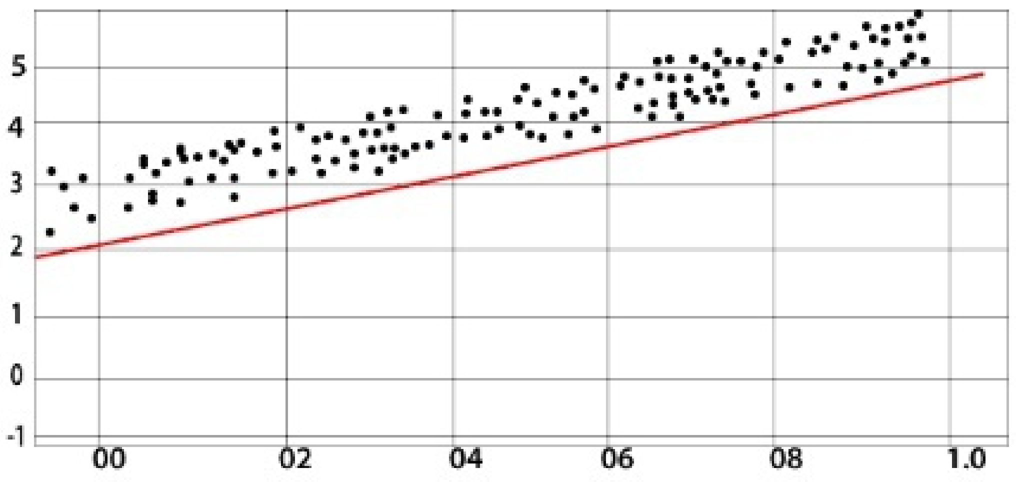
In layman's terms, the relationship between both variables, whether independent or dependent, is referred to as linear.

If Y is the dependent variable and X is the independent variable, the linear regression relationship between these two variables is as follows:

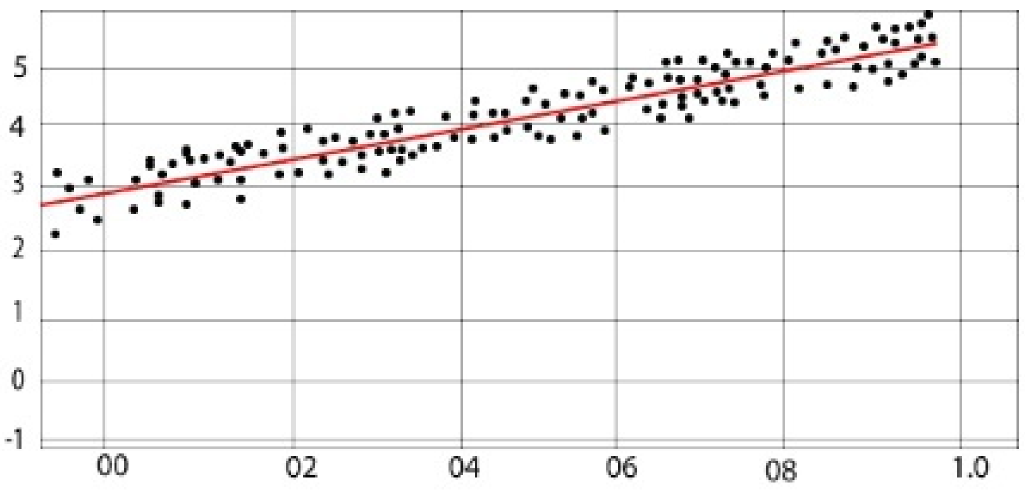
$$\mathbf{Y=AX+b}$$

- A is the **slope** .
- b is **y-intercept** .

Initial State



Final State



There are three fundamental principles that must be understood in order to construct or practice a basic linear model.

Model class

It's common practice to code all and write all the functions when they're needed, but that's not our intention.

While it is often preferable to write numeric optimization libraries rather than all of the code and functions, the business benefit can be improved if we build on top of prewritten libraries to get things done.

We employ the implementation of PyTorch's nn package for this purpose. To do so, we must first make a single sheet.

Linear layer use

Each linear module computes the output from the input and retains its own internal Tensor for weight and bias.

There are a number of other standard modules available. We'll use a model class format, which has two key methods:

1. Init: This function is used to define a linear module.
2. Forward: We will train our linear regression model on the basis of predictions made using the forwarding process.

Optimizer

One of PyTorch's most important principles is the optimizer.

It's used to find the best weight for our model to fit into the dataset.

Gradient descent and backpropagation are two optimization algorithms that optimize our weight value and best suit our model.

The torch.optim package implements a number of optimization algorithms.

To use torch.optim, build an optimizer object that will change the parameter based on the device gradient while preserving the current state.

The item is made in the following way:

1. Optimizer=optim.SGD(model.parameters(), lr=0.01 , momentum=0.9)
2. Optimizer=optim.Adam([var1, var2], lr=0.0001))

All optimizers implement the step() process, which updates the parameters. It can be used in two ways.

1) Optimizer.step()

This is a rather straightforward approach that is backed by a large number of optimizers. We may call the optimizer.step() function after computing the gradients with the backward () process.

EXAMPLE

```
for input, target in dataset:
    optimizer.zero_grad()
    output=model(input)
    loss=loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

2) Optimizer.step(closure)

Some optimization algorithms, such as LBFGS and Conjugate Gradient, require several re-evaluations of the equation, so we must move it in a closure that allows them to recompute your model.

EXAMPLE

```
for input, target in dataset:
def closure():
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    return loss
    optimizer.step(closure)
```

Criterion

Our loss function, which is used to locate loss, is the criterion. The torch nn module calls this feature.

EXAMPLE

```
criterion = torch.nn.MSELoss(size_average = False)
```

Functions and objects which are required

1. Import torch
2. From torch.autagrad import Variable

Furthermore, we must describe some data and allocate it to variables in the following manner.

```
xdata=Variable(torch.Tensor([[ 1.0 ],[ 2.0 ],[ 3.0 ]]))
ydata=Variable(torch.Tensor([[ 2.0 ],[ 4.0 ],[ 6.0 ]]))
```

Prediction and linear class

We took a quick look at how to use a machine learning-based algorithm to train a linear model to fit a collection of data points in this tutorial.

It is not necessary to have any previous knowledge of deep learning for this reason.

We'll begin by talking about supervised learning.

We'll talk about what supervised learning is and how it applies to it.

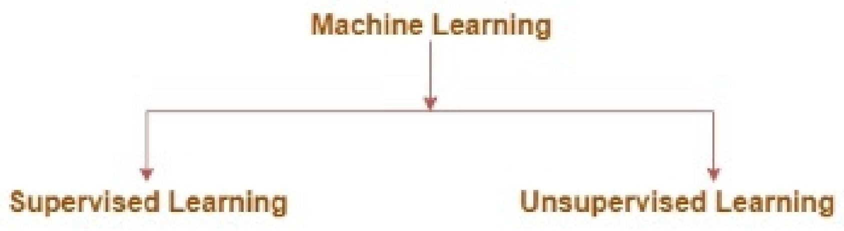
Machine Learning

AI is used in the form of machine learning. Machine Learning (ML) offers systems the opportunity to learn and improve on their own, based on their past experiences.

ML is concerned with the development of computer programs that can access data and learn from it.

Data or observation, such as examples, instructions, or direct experience, is used to search for trends in data and make informed decisions in the future based on the examples we give.

Its purpose is to allow computers to learn on their own, without the need for human interference, and change their actions accordingly.



Supervised Learning

The existence of a supervisor as a teacher is indicated by the name.

In supervised learning, we use well-labeled data to train or teach the computer.

The word "well labelled" refers to how much of the data has already been marked with the correct response.

Following that, the computer is given a new collection of data.

The supervised learning algorithm examines the training data and generates an accurate result from labeled data.

Unsupervised Learning

Unsupervised learning allows the computer to be trained using data that is neither categorized nor named, allowing the algorithm to operate on that data without being directed.

Without any previous data preparation, the role of unsupervised learning is to group unsorted information according to similarities, differences, and patterns.

Since there is no supervisor, the computer would receive no instruction.

As a result, the computer is limited in its ability to discover the secret structure on its own.

Making Prediction

The first step in creating a linear regression model is to make a prediction.

We use supervised learning in a linear regression model since regression is the second large category.

As a result, the learner is prepared and uses data sets with named features that describe the value of our training data.

Until giving the newly input data to the computer, the learner will predict a corresponding result.

Steps to find the prediction

- The first step is to set up the torch and import it so that it can be used.
- To know the equation of a line, the next step is to initialize the variables c and c.

- Initialize the line equation to y=w*x + b, where w denotes slop, b denotes bias, and y denotes prediction.
- The forward () approach is used to measure the prediction.

Let's look at an example to see how linear regression predictions are made.

For single data

```
import torch
b=torch.tensor( 3.0 ,requires_grad=True)
c=torch.tensor( 1.0 ,requires_grad=True)
def forward(x):
        y=b*x+c
        return y
x=torch.tensor([ 4.0 ])
forward(x
```

OUTPUT

```
tensor([13.], grad_fn=<AddBackward0>)
```

For multiple data

```
import torch
b=torch.tensor( 3.0 ,requires_grad=True)
c=torch.tensor( 1.0 ,requires_grad=True)
def forward(x):
        y=b*x+c
        return y
x=torch.tensor([[ 4.0 ],[ 5.0 ],[ 6.0 ]])
forward(x)
```

OUTPUT

```
tensor([[13.],
        [16.],
        [19.]], grad_fn=<AddBackward0>)
```

Prediction using linear class

Another traditional method of binding prediction exists. To do so, we'll need to import the torch.nn package's linear type.

To generate random numbers, we use the manual seed() process.

When we create a model with a linear class, the linear class will be given random number values, which makes sense given the recall.

Let's look at an example of how the model and manual seed() method are used to make predictions.

For single data

```
import torch
from torch.nn import Linear
torch.manual_seed( 1 )
model=Linear(in_features= 1 ,out_features= 1 )
print(model.bias,model.weight)
x=torch.tensor([ 2.0 ])
print(model(x))
```

Output:

```
<torch._C.Generator object at 0x000001FA018DB2B0>

Parameter containing:

tensor([-0.4414], requires_grad=True) Parameter containing:

tensor([[0.5153]], requires_grad=True)

tensor([0.5891], grad_fn=<AddBackward0>)
```

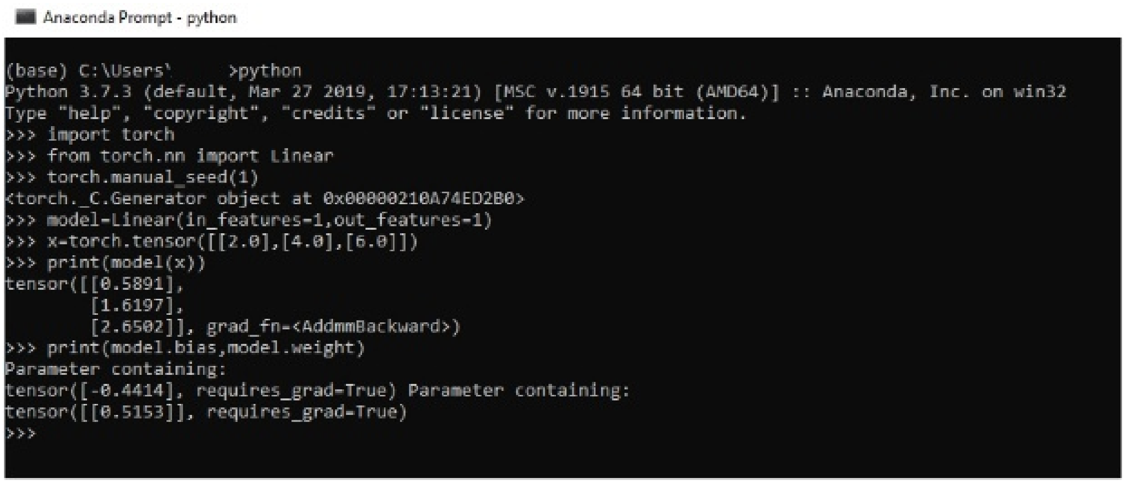
For multiple data

```
import torch
from torch.nn import Linear
torch.manual_seed( 1 )
```

```
model=Linear(in_features= 1 ,out_features= 1 )
print(model.bias,model.weight)
x=torch.tensor([[ 2.0 ],[ 4.0 ],[ 6.0 ]])
print(model(x))
```

OUTPUT

```
<torch._C.Generator object at 0x00000210A74ED2B0>
Parameter containing:
tensor([-0.4414], requires_grad=True) Parameter containing:
tensor([[0.5153]], requires_grad=True)
tensor([[0.5891],
        [1.6197],
        [2.6502]], grad_fn=<AddmmBackward>)
```



Anaconda Prompt - python

```
(base) C:\Users\      >python
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> from torch.nn import Linear
>>> torch.manual_seed(1)
<torch._C.Generator object at 0x00000210A74ED2B0>
>>> model=Linear(in_features=1,out_features=1)
>>> x=torch.tensor([[2.0],[4.0],[6.0]])
>>> print(model(x))
tensor([[0.5891],
        [1.6197],
        [2.6502]], grad_fn=<AddmmBackward>)
>>> print(model.bias,model.weight)
Parameter containing:
tensor([-0.4414], requires_grad=True) Parameter containing:
tensor([[0.5153]], requires_grad=True)
>>>
```

Creating Data Model

using Custom Module

There is another method for determining the forecast. We found the prediction in the previous section by using forward () and implementing a linear model.

This approach is extremely effective and dependable. It is easy to comprehend and execute.

We build a customize module with a class, init() and forward() methods, and a model in the Custom Module. The class's new instances are initialized using the init() process.

The first argument in this init() method is self, which indicates the instance of the class that needs to be initialized, and after that, we can add more arguments.

The next parameter is to initialize the linear model case. In the previous section, we saw how to initialize a linear model by setting the input size and output size variables to 1, but in the custom module, we set the input size and output size variables to their default values.

It is necessary to import the torch's nn kit in this case. We use inheritance in this case so that this subclass can use code from both our base class and the module.

In certain cases, the module may serve as a base class for all neural network modules. After that, we create a model and make a prediction using it.

Let's look at an example of how a custom module can be used to predict:

FOR SINGLE DATA

```
import torch
import torch.nn as nn
class LinearRegression(nn.Module):
    def __init__(self,input_size, output_size):
        super ().__init__()
        self.linear=nn.Linear(input_size,output_size)
    def forward(self,x):
        pred=self.linear(x)
        return pred
torch.manual_seed( 1 )
model=LinearRegression( 1 , 1 )
x=torch.tensor([ 1.0 ])
print(model.forward(x))
```

OUTPUT

```
<torch._C.Generator object at 0x000001B9B6C4E2B0>
tensor([0.0739], grad_fn=<AddBackward0>)
```

FOR MULTIPLE DATA

```
import torch
import torch.nn as nn
class LinearRegression(nn.Module):
    def __init__(self,input_size, output_size):
        super ().__init__()
        self.linear=nn.Linear(input_size,output_size)
    def forward(self,x):
        pred=self.linear(x)
        return pred
torch.manual_seed( 1 )
model=LinearRegression( 1 , 1 )
x=torch.tensor([[ 1.0 ],[ 2.0 ],[ 3.0 ]])
print(model.forward(x))
```

OUTPUT

```
<torch._C.Generator object at 0x000001B9B6C4E2B0>
tensor([[0.0739],
        [0.5891],
        [1.1044]], grad_fn=<AddmmBackward>)
```

Gradient Descent in PyTorch

Our main concern is how to train a model to find the weight parameters that minimize our error function.

Let's look at how gradient descent can assist us in training our model.

When we initialize the linear model with the linear function, it will start with a random initial parameter recall.

It did, in reality, provide us with a random initial parameter.

Let's disregard the bias value for the time being and concentrate on the error associated with parameter A.

Our aim is to step in the direction with the least amount of error.

The derivatives of the slope of the tangent at the current value that we encountered will take us in the direction of the highest error if we take the gradient of error function.

$\text{Loss}=(3+3A)^{2}$

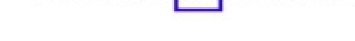
$f'(A)=18(A+1)$

As a consequence, we transfer it in the negative direction of the gradient, which will lead us to the lowest error.

At the same point, we convert current to weight and deduct the derivatives of that function.

$A1=A0-f'(A)$

It will lead us in the direction of making the fewest mistakes.



In a nutshell, we must first compute the derivatives of the loss function before submitting the line's current weight value.

They will give you the gradient value regardless of the weight.

The new revised weight A1 is obtained by subtracting the gradient value from the original weight A0. The new weight should produce a smaller error than the old one.

We'll keep doing this until we find the best parameter for our line model to match the data.

However, we are descending with the gradient to ensure the best results.

The descent should be performed in small measures. As a result, we'll divide the gradient by a small number called the learning rate.

The learning rate has an objective validity. Although a reasonable standard starting value is one over ten or one over one hundred, the learning rate must be small enough to avoid the line moving too far in one direction, which can cause unnecessary divergent behaviour.

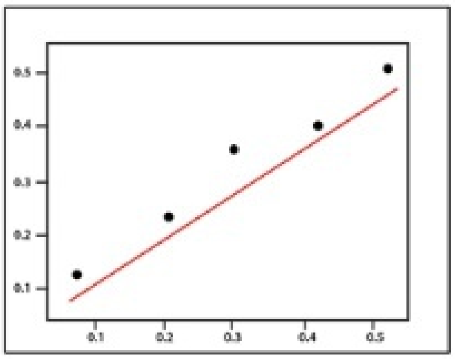
Mean Squared Error

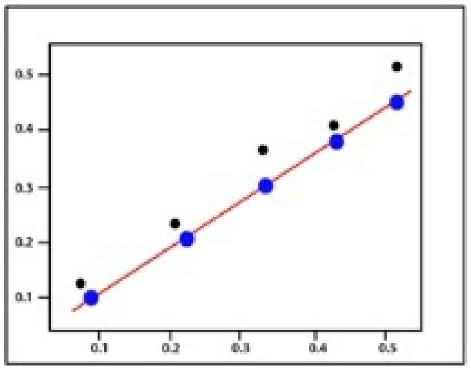
The Mean Squared Error is determined in the same way that the general loss equation was calculated earlier. We'll also take into account the bias value, which is another parameter that needs to be changed during the training phase.

$$(y-Ax+b)^2$$

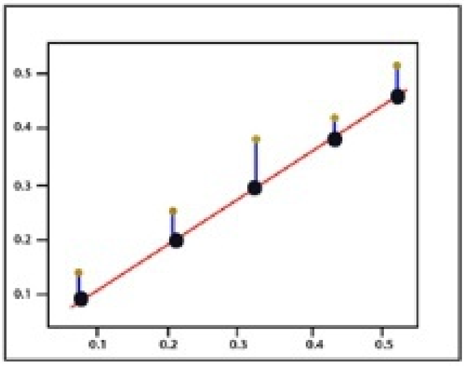
An example is the best way to describe the mean squared error.

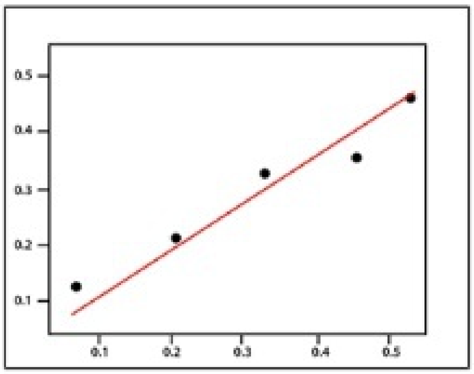
Assume we have a set of values and begin by drawing a regression line parameter sized by a random set of weight and bias values, just as we did before.





The error is the difference between the real and expected values - the actual distance between them.





The error is determined for each point by comparing the predicted values produced by our line model with the actual value using the formula below.

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - y_i^{\wedge})^2$$

Every point has an error associated with it, so we must add up all of the errors for each point. We know that the prediction can be rewritten in the following way:

$$Ax + b$$

So,

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (Ax_i + b))^2$$

We must take the average by dividing by the number of data points when measuring the mean squared error. As previously reported, the error function's gradient should point us in the direction of the greatest increase in error.

We travel in the direction of the smallest error by heading towards the negative of our cost function's gradient. We'll use this gradient as a compass to guide us downhill at all times. The existence of bias is ignored in gradient descent, but both parameters A and b must be specified for the error.

Now we'll calculate the partially derivatives for each, starting with any A and b value pair, as we did before.

$$f(A, b) = \frac{1}{N}\sum_{i=1}^{n}(y_i - (Ax_i + b))^2$$

$$\left[\frac{df}{dm}\right] = [\frac{1}{N}\sum 2x_i(y_i - (Ax_i + b))]$$

$$\left[\frac{df}{db}\right] = [\frac{1}{N}\sum 2(y_i - (Ax_i + b))]$$

Based on the two partial derivatives described earlier, we use a gradient descent algorithm to update A and b in the direction of least error. The current weight is equal to the previous weight for each iteration.

$$\mathbf{A_1 = A_0 - \propto f'(A)}$$

And the latest bias value is the same as before.

$$\mathbf{b_1 = b_0 - \propto f'(b)}$$

The key concept behind the code is that we start with a random model with a random collection of weight and bias parameter values. We use gradient descent to update the weight of our model in the direction of the least error since this random model has a large error function and a large cost function. Optimizing the outcome by minimizing the error.

Perceptron Introduction

A perceptron is a single-layer neural network, or a multi-layer perceptron.

A binary classifier, the perceptron is used in supervised learning. Perceptron is a basic model of a biological neuron in an artificial neural network.

Binary classifiers are functions that can determine whether or not an input represented by a vector of numbers belongs to a particular class.

A type of linear classifier is the binary classifier. A linear classifier is a classification algorithm that uses a linear predictor function that combines a collection of weights with the feature vector to make predictions.

The perceptron algorithm was developed to divide subjects into two categories, distinguish visual data, and draw a line between classes. Image processing and machine learning include classification. The perceptron algorithm classifies patterns, i.e., identifies and classifies them using a machine learning algorithm, and groups them by determining the linear separation between different objects and patterns obtained through numeric or visual input.

Perceptrons are made up of four components, all of which must be understood before the perceptron model can be implemented in PyTorch.

Input values or one input layer

A perceptron's input layer is made up of artificial input neurons that carry the initial data into the system for processing.

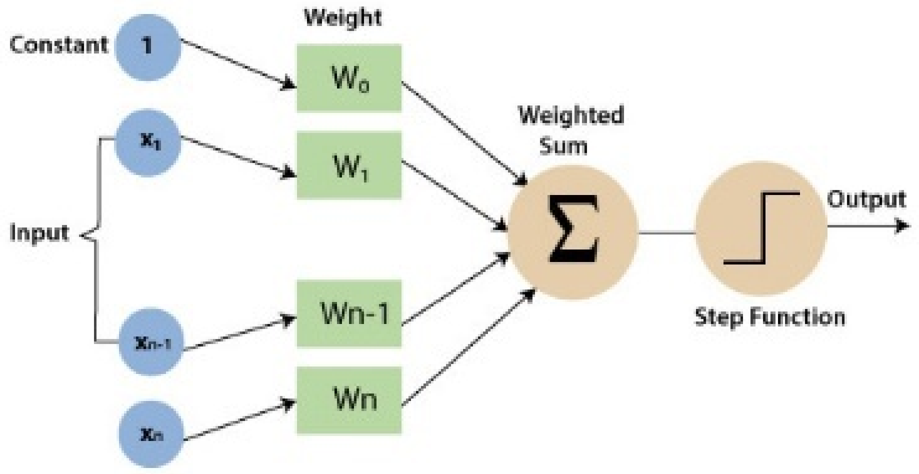
Weights and bias

The intensity or dimension of a relationship between units is represented by its weight. If the weight from node 1 to node 2 is greater, neuron 1 would have a greater impact on neuron 2. The amount of impact the input has on the output is determined by the weight.

In a linear equation, bias is identical to the intercept. It's an extra parameter whose job is to change the output as well as the weighted number of the neuron's inputs.

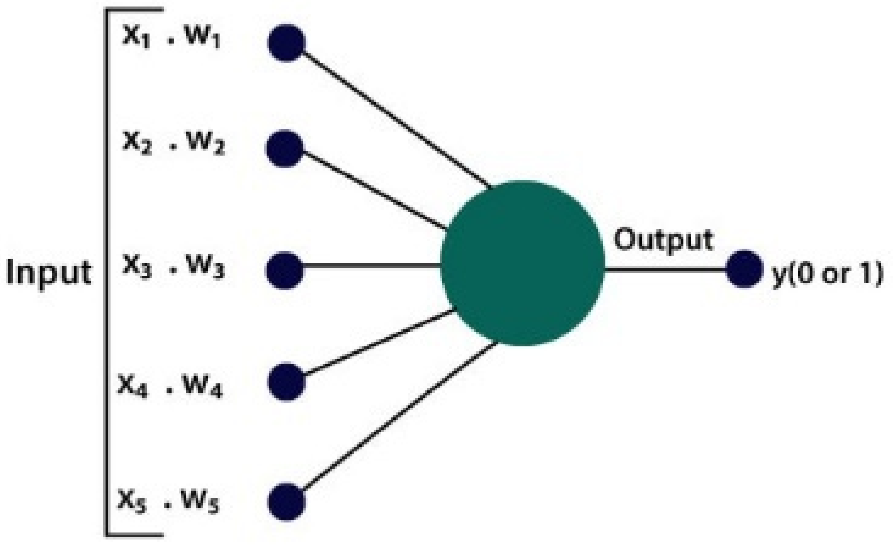
Activation Function

An activation mechanism determines whether a neuron should be stimulated or not. To get the result, the activation function calculates a weighted sum and then applies bias to it.

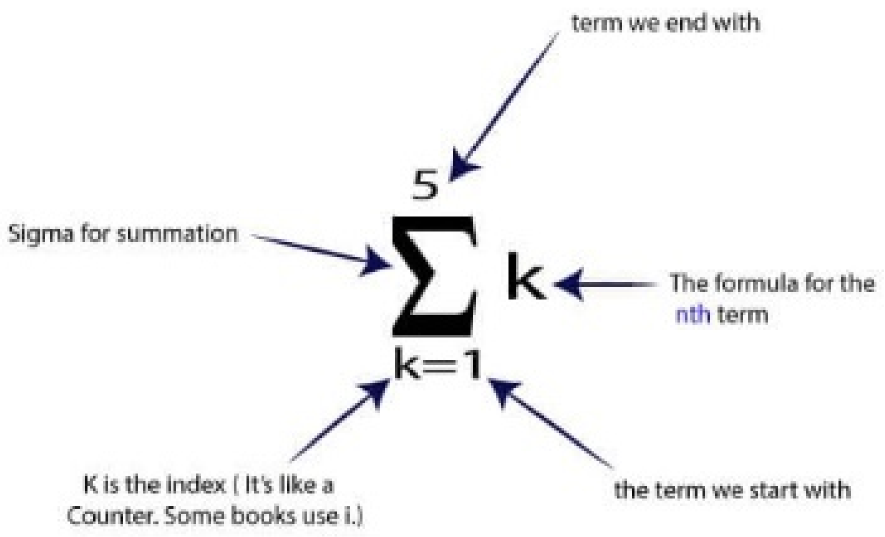


The Perceptron in three steps:

a) The first step involves multiplying all of the input x by their weights, which are denoted by K. This step is important because the output from it will be used as input for the next step.



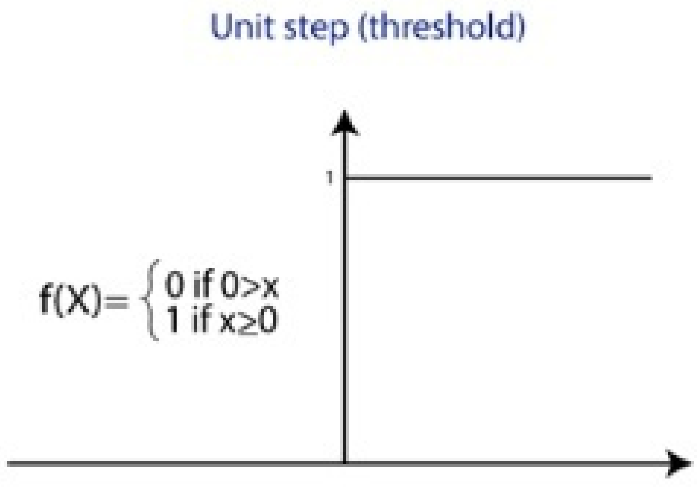
b) Next, apply all of the compounded values from K1 to Kn together. The weighted total is what it's called. This weighted sum will be used as input in the following stage.


term we end with
5
Sigma for summation
$\sum$ k
The formula for the nth term
k=1
K is the index (It's like a Counter. Some books use i.)
the term we start with

c) The weighted sum determined in the previous step is added to the right activation function in the next step.

For example

A unit step activation function


Unit step (threshold)
$f(X)=\begin{cases}0 \text{ if } 0>x\\1 \text{ if } x\geq 0\end{cases}$