PyTorch is a Python-based library that provides maximum flexibility and speed and is well known for its fast computational power.

Let's understand what PyTorch is and why it has become so popular lately, before diving into its implementation.

PyTorch is a Python-based scientific computing package that is ***similar to NumPy but with the added power of GPUs***. It is also a ***deep learning framework that provides maximum flexibility and speed*** during implementing and building deep neural network architectures.

Recently, PyTorch 1.0 was released and it was aimed to assist researchers by addressing four major challenges:

- Extensive reworking
- Time-consuming training
- Python programming language inflexibility
- Slow scale-up

So make sure you install PyTorch on your machine before proceeding. The latest version of PyTorch (PyTorch 1.2) was released on August 08 2019, and installing PyTorch is pretty easy.

You can follow the steps mentioned in the official docs and run the command as per your system specifications. For example, this is the command you can use on the basis of the options you have:

```
conda install pytorch torchvision cuda91 -c pytorch
```

Why should we use PyTorch?

The workflow of PyTorch is as close as you can get to python's scientific computing library – numpy.

Now you might ask, why would we use PyTorch to build deep learning models? Intrinsically, there are two main characteristics of PyTorch that distinguish it from other deep learning frameworks:

- *Imperative Programming*
- *Dynamic Computation Graphing*

Imperative Programming: PyTorch performs computations as it goes through each line of the written code. This is quite similar to how a Python program is executed. This concept is called imperative programming. The biggest advantage of this feature is that your code and programming logic can be debugged on the fly. PyTorch's creators say that they have a philosophy – they want to be imperative. This means that we run our computation immediately. This fits right into the python programming methodology, as **we don't have to wait for the whole code to be written before getting to know if it works or not. We can easily run a part of the code and inspect it in real-time.**

Dynamic Computation Graphing: PyTorch is referred to as a "defined by run" framework, which means that the computational graph structure (of a neural network architecture) is generated during run time. The main advantage of this property is that it provides a flexible and programmatic runtime interface that facilitates the construction and modification of systems by connecting operations. In PyTorch, a new computational graph is defined at each forward pass. This is in stark contrast to TensorFlow which uses a static graph representation.

PyTorch has a unique way of building neural networks. It creates dynamic computation graphs meaning that the graph will be created on the fly:

PyTorch 1.0 comes with an important feature called **torch.jit**, a high-level compiler that allows the user to separate the models and code. It also supports **efficient model optimization on custom hardware, such as GPUs or TPUs**.

Let's look at some of the other key features of this library which make it unique and easy to use.

TorchScript

PyTorch TorchScript helps to create serializable and optimizable models. Once we train these models in Python, they can be run independently from Python as well. This helps when we're in the model deployment stage of a data science project.

So, **you can train a model in PyTorch using Python and then export the model via TorchScript to a production environment where Python is not available.** We will discuss model deployment in more detail in the later articles of this series.

Distributed Training

PyTorch also supports distributed training which enables researchers as well as practitioners to parallelize their computations. Distributed training makes it possible to use multiple GPUs to process larger batches of input data. This, in turn, reduces the computation time.

Python Support

PyTorch has a very good interaction with Python. In fact, coding in PyTorch is quite similar to Python. So **if you are comfortable with Python, you are going to love working with PyTorch.**

And this is just skimming the surface of why PyTorch has become such a beloved framework in the data science community. Since its release in the start of January 2016, many researchers have adopted it as a go-to library because of its ease of building novel and even extremely complex graphs. Having said that, there is still some time before PyTorch is adopted by the majority of data science practitioners due to it's new and "under construction" status.

Tensors in PyTorch

Tensors are the base data structures of PyTorch which are used for building different types of neural networks. They can be considered as the generalization of arrays and matrices; in other words, tensors are N-dimensional matrices.

In other words, Tensors are multidimensional arrays. And PyTorch tensors are similar to NumPy's n-dimensional arrays. We can use these tensors on a GPU as well (this is not the case with NumPy arrays). This is a major advantage of using tensors.

PyTorch supports multiple types of tensors, including:

1. FloatTensor: 32-bit float
2. DoubleTensor: 64-bit float
3. HalfTensor: 16-bit float
4. IntTensor: 32-bit int
5. LongTensor: 64-bit int

Mathematical Operations in PyTorch(vs. NumPy)

Do you remember how to perform mathematical operations on NumPy arrays? If not, let me quickly recap that for you.

We will initialize two arrays and then perform mathematical operations like addition, subtraction, multiplication, and division, on them:

```
# initializing two arrays
a = np.array(2)
b = np.array(1)
print(a,b)
```

These are the two NumPy arrays we have initialized. Now let's see how we can perform mathematical operations on these arrays:

```
# addition
print(a+b)
```

```
# subtraction
print(b-a)

# multiplication
print(a*b)

# division
print(a/b)
```

Let's now see how we can do the same using PyTorch on tensors. So, first, let's initialize two tensors:

```
# initializing two tensors
a = torch.tensor(2)
b = torch.tensor(1)
print(a,b)
```

Next, perform the operations which we saw in NumPy:

```
# addition
print(a+b)

# subtraction
print(b-a)

# multiplication
print(a*b)

# division
print(a/b)
```

Did you see the similarities? The codes are exactly the same to perform the above mentioned mathematical operations in both NumPy and PyTorch.

Next, let's see how to initialize a matrix as well as perform matrix operations in PyTorch (along with, you guessed it, it's NumPy counterpart!)

Matrix Operations in PyTorch(vs. NumPy)

Matrix Initialization

Let's say we want a matrix of shape 3*3 having all zeros. Take a moment to think – how can we do that using NumPy?

```
# matrix of zeros
a = np.zeros((3,3))
print(a)
print(a.shape)
```

Fairly straightforward. We just have to use the *zeros()* function of NumPy and pass the desired shape ((3,3) in our case), and we get a matrix consisting of all zeros. Let's now see how we can do this in PyTorch:

```
# matrix of zeros
a = torch.zeros((3,3))
print(a)
print(a.shape)
```

Similar to NumPy, PyTorch also has the *zeros()* function which takes the shape as input and returns a matrix of zeros of a specified shape. Now, while building a neural network, we randomly initialize the weights for the model. So, let's see how we can initialize a matrix with random numbers:

```
# setting the random seed for numpy
np.random.seed(42)

#matrix of random numbers
a = np.random.randn(3,3)
a
```

We have specified the random seed at the beginning here so that every time we run the above code, the same random number will generate. **The *random.randn()* function returns random numbers that follow a standard normal distribution.**

But let's not get waylaid by the statistics part of things. We'll focus on how we can initialize a similar matrix of random numbers using PyTorch:

```
# setting the random seed for pytorch
torch.manual_seed(42)

# matrix of random numbers
a = torch.randn(3,3)
a
```

This is where even more similarities with NumPy crop up. PyTorch also has a function called *randn()* that returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

Note that we have set the random seed here as well just to reproduce the results every time you run this code. So far, we have seen how to initialize a matrix using PyTorch. Next, let's see how to perform matrix operations in PyTorch.

Matrix Operations

We will first initialize two matrices in NumPy:

```
# setting the random seed for numpy and initializing two matrices
np.random.seed(42)
a = np.random.randn(3,3)
b = np.random.randn(3,3)
```

Next, let's perform basic operations on them using NumPy:

```
# matrix addition
print(np.add(a,b), '\n')

# matrix subtraction
print(np.subtract(a,b), '\n')

# matrix multiplication
print(np.dot(a,b), '\n')

# matrix multiplication
print(np.divide(a,b))
```

In PyTorch,

```
# setting the random seed for pytorch and initializing two tensors
torch.manual_seed(42)
a = torch.randn(3,3)
b = torch.randn(3,3)
# matrix addition
print(torch.add(a,b), '\n')

# matrix subtraction
print(torch.sub(a,b), '\n')

# matrix multiplication
print(torch.mm(a,b), '\n')

# matrix division
print(torch.div(a,b))
```

Note that the *.mm()* function of PyTorch is similar to the dot product in NumPy. This function will be helpful when we create our model from scratch in PyTorch.

Matrix transpose is one technique which is also very useful while creating a neural network from scratch. So let's see how we take the transpose of a matrix in NumPy:

```
# original matrix
print(a, '\n')

# matrix transpose
print(np.transpose(a))
```

The *transpose()* function of NumPy automatically returns the transpose of a matrix. How does this happen in PyTorch? Let's find out:

```
# original matrix
print(a, '\n')

# matrix transpose
torch.t(a)
```

Next, we will look at some other common operations like concatenating and reshaping tensors.

Tensor Operations

Now, let's look at the basics of PyTorch along with how it compares against NumPy. We'll start by importing both the NumPy and the Torch libraries:

```
# importing libraries
import numpy as np
import torch
```

Now, let's see how we can assign a variable in NumPy as well as PyTorch:

```
# initializing a numpy array
a = np.array(1)

# initializing a tensor
b = torch.tensor(1)

print(a)
print(b)
```

Let's quickly look at the type of both these variables:

```
type(a), type(b)
```

Type here confirms that the first variable (a) here is a NumPy array whereas the second variable (b) is a torch tensor.

From this point forward, we will not be comparing PyTorch against NumPy as you must have got an idea of how the codes are similar.

Concatenating Tensors

Let's say we have two tensors as shown below:

```
# initializing two tensors
a = torch.tensor([[1,2],[3,4]])
b = torch.tensor([[5,6],[7,8]])
print(a, '\n')
print(b)
```

What if we want to concatenate these tensors vertically? We can use the below code:

```
# concatenating vertically
torch.cat((a,b))
```

As you can see, the second tensor has been stacked below the first tensor. We can concatenate the tensors horizontally as well by setting the dim parameter to 1:

```
# concatenating horizontally
torch.cat((a,b),dim=1)
```

Reshaping Tensors

Let's say we have the following tensor:

```
# setting the random seed for pytorch
torch.manual_seed(42)

# initializing
tensor a = torch.randn(2,4)
print(a)
a.shape
```

We can use the *.reshape()* function and pass the required shape as a parameter. Let's try to convert the above tensor of shape (2,4) to a tensor of shape (1,8):

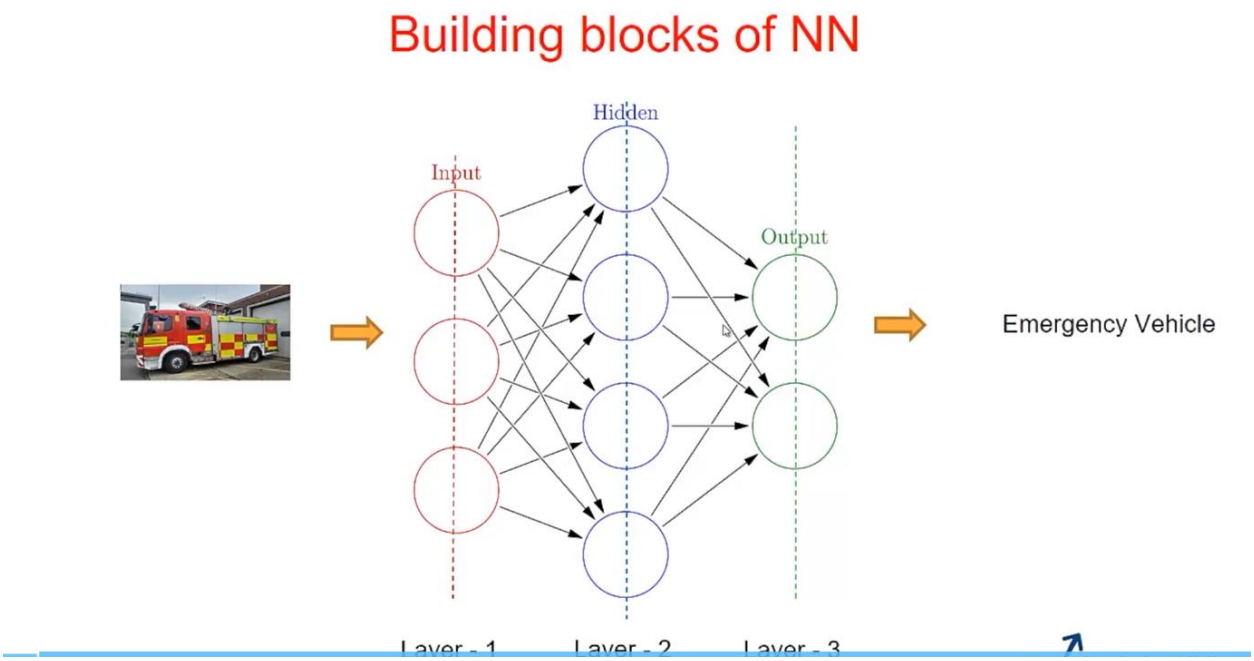
```
# reshaping tensor
b = a.reshape(1,8)
print(b)
b.shape
```

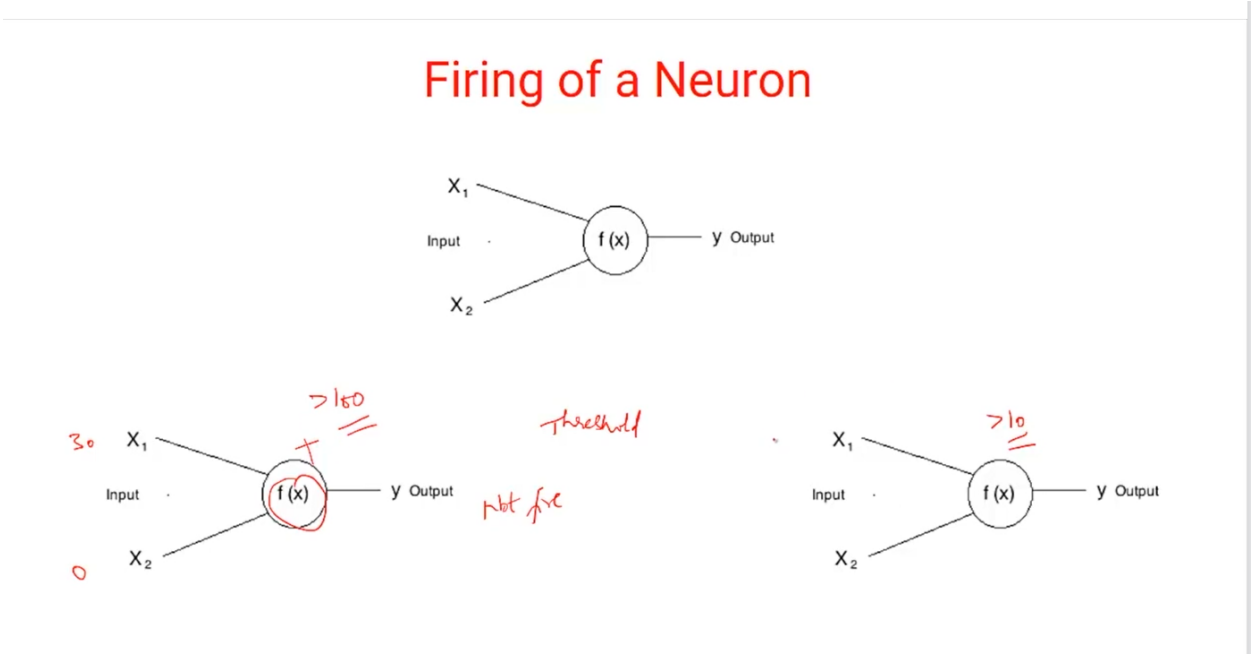
Awesome! PyTorch also provides the functionality to convert NumPy arrays to tensors. You can use the below code to do it:

```
# initializing a numpy array
a = np.array([[1,2],[3,4]])
print(a, '\n')

# converting the numpy array to tensor
tensor = torch.from_numpy(a)
print(tensor)
```

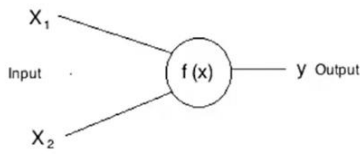
Let's move on and dive deeper into Neural networks and other various aspects of PyTorch.

Getting started with Neural Networks





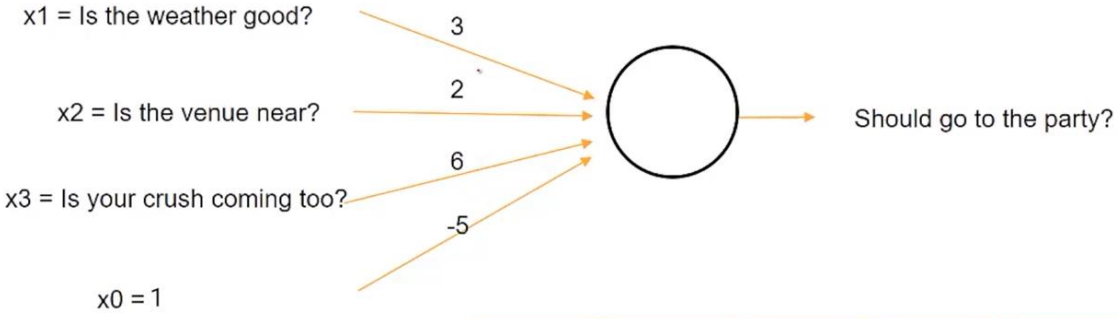
Bias of a Neuron.



$x_1 + x_2 - threshold > 0$ *Fire* $x_1 + x_2 + bias > 0$

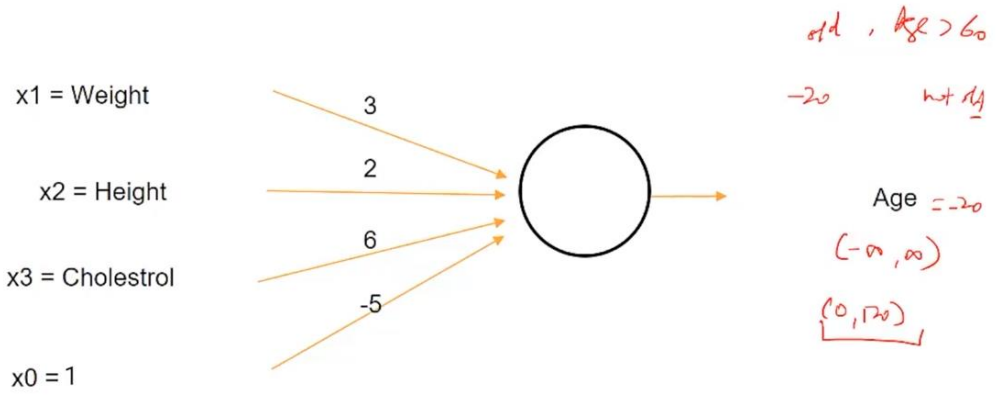
$x_1 + x_2 - threshold < 0$ *No Fire* $x_1 + x_2 + bias < 0$

Weights of a Neuron



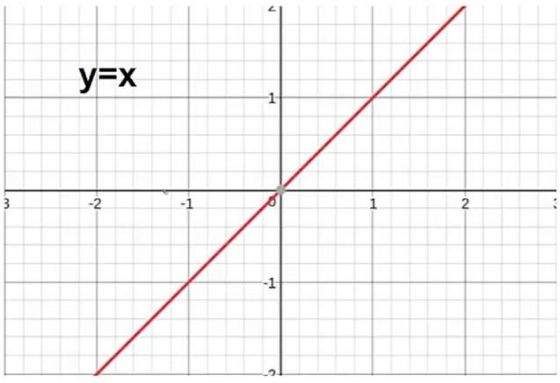
X0(w0)	X1(w1)	X2(w2)	X3(w3)	y
1(-5)	0(3)	0(2)	1(6)	

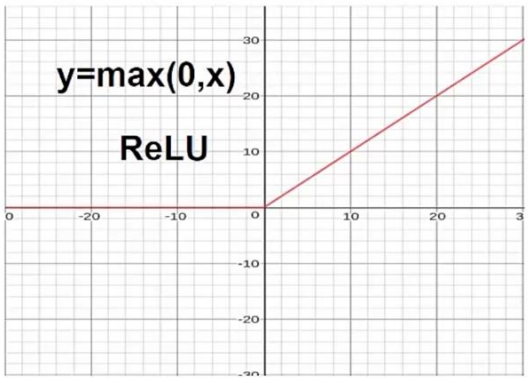
Weights of a Neuron

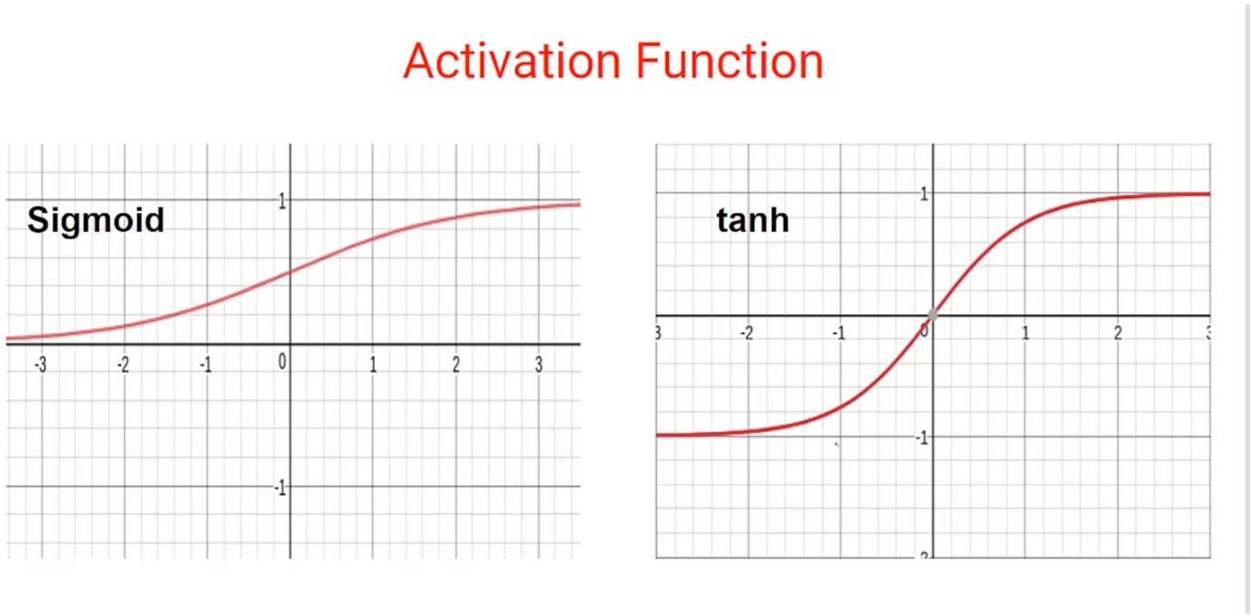


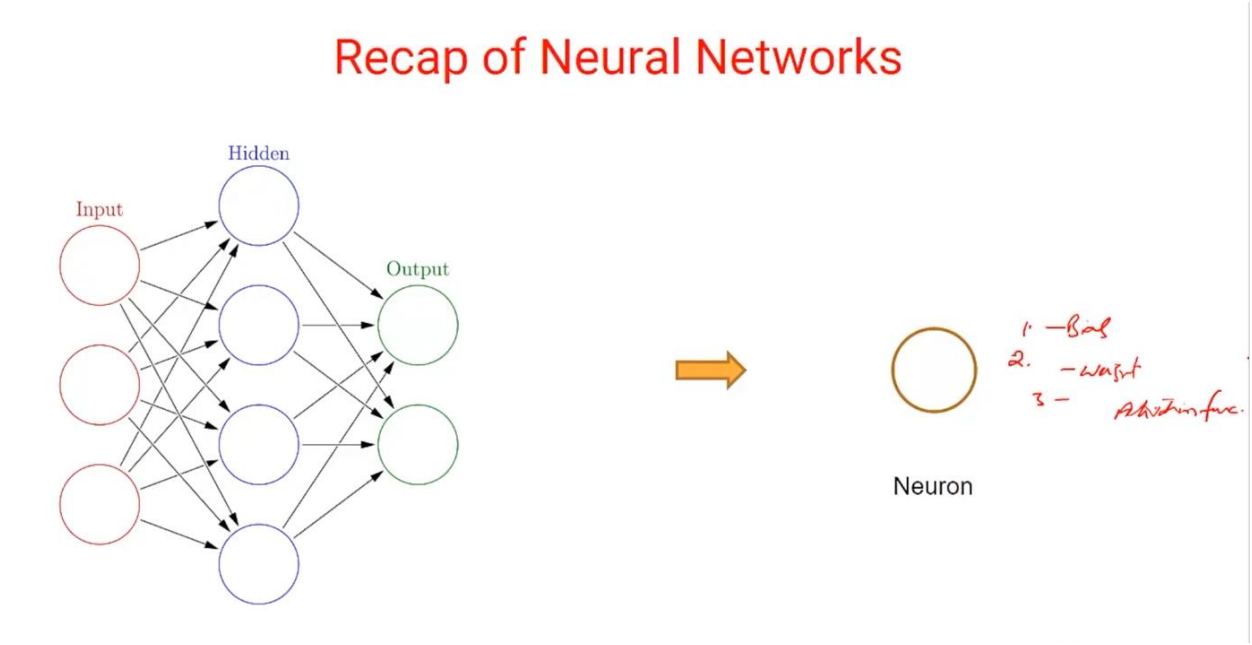
Age

Activation Function









Independent and Dependent Variables

The target variable or the variable which we are trying to predict is known as the dependent variable. Whereas all the other variables that are used to predict the target / dependent variable are known as independent variables.

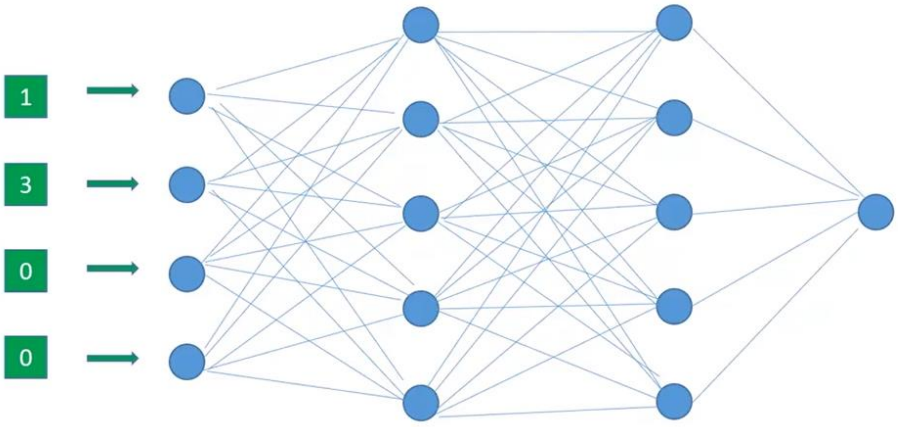
Consider we have a problem where a bank want to predict whether a loan will be given to a customer or not (Loan_Status). We have the customer details like age, salary, credit_history and based on these variables the bank want to predict whether to give a loan or not. In this example, Loan_Status is a dependent variable as its value is dependent on age, salary, credit_history of the customer. age, salary, credit_history on the other hand are independent variables as they are used to predict the Loan_Status.

Understanding Forward Propagation

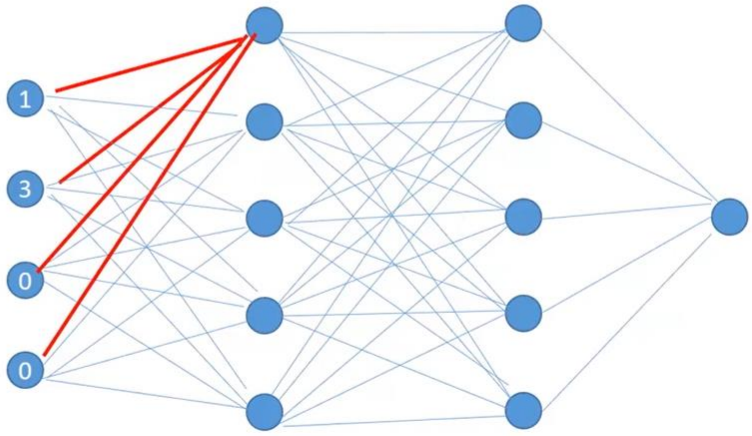
Dataset Overview

gender	age	hypertension	does_smoke	stroke
1	3	0	0	0
1	58	1	1	1
0	8	0	0	0
0	70	0	1	1
1	14	0	0	0
0	47	0	0	0
0	52	0	1	1
0	75	0	0	0
0	32	0	1	0

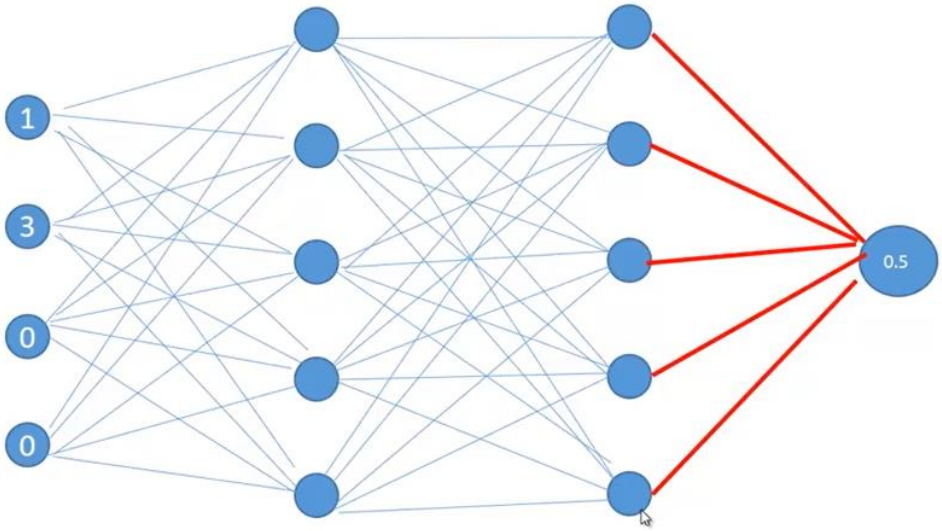
Dataset Overview



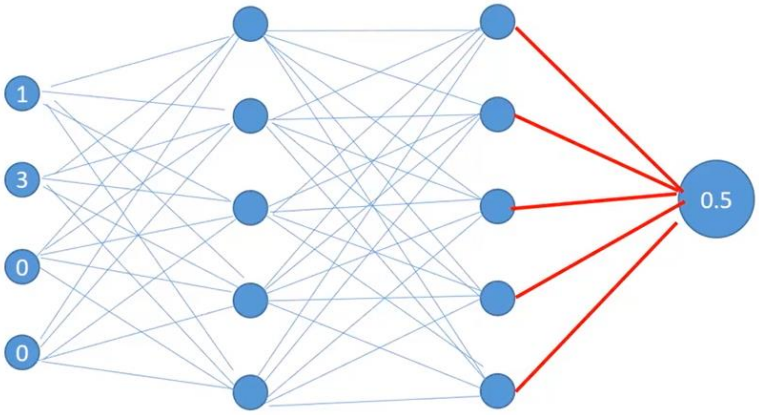
Dataset Overview



Dataset Overview



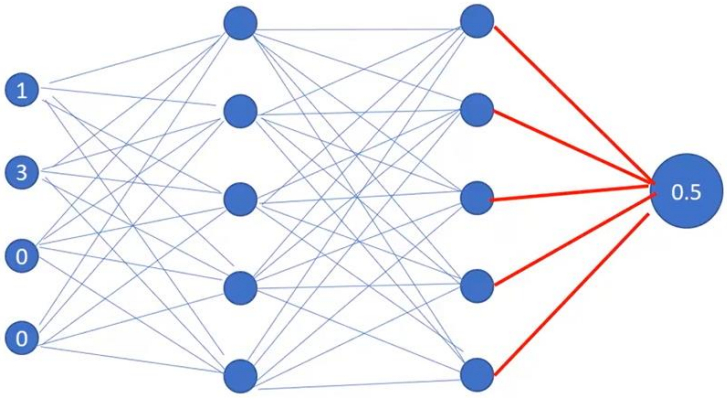
Actual vs Predicted



Actual	Predicted
0	0.5

Error and Reason for Error

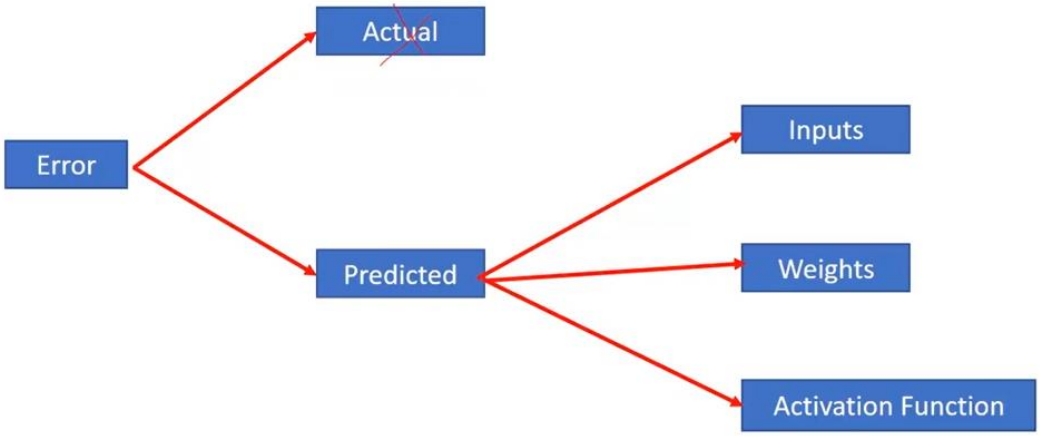
Error

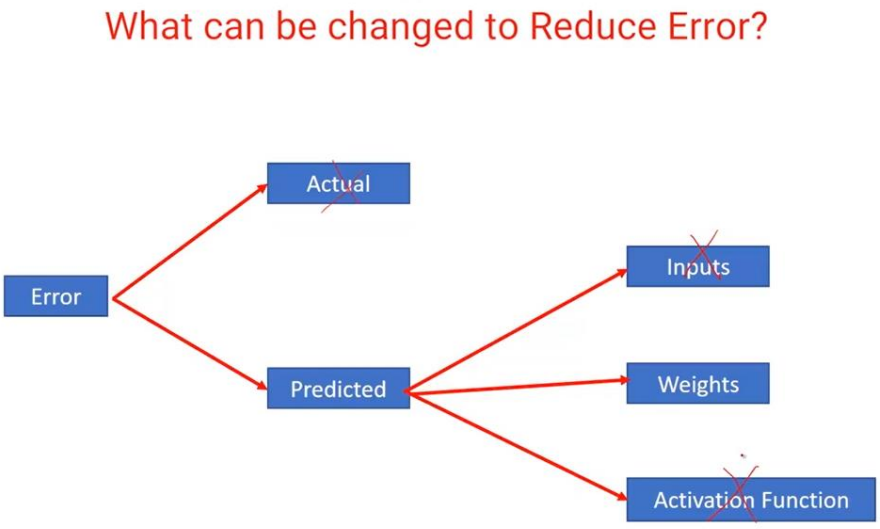


Actual	Predicted	Error
0	0.5	0.125

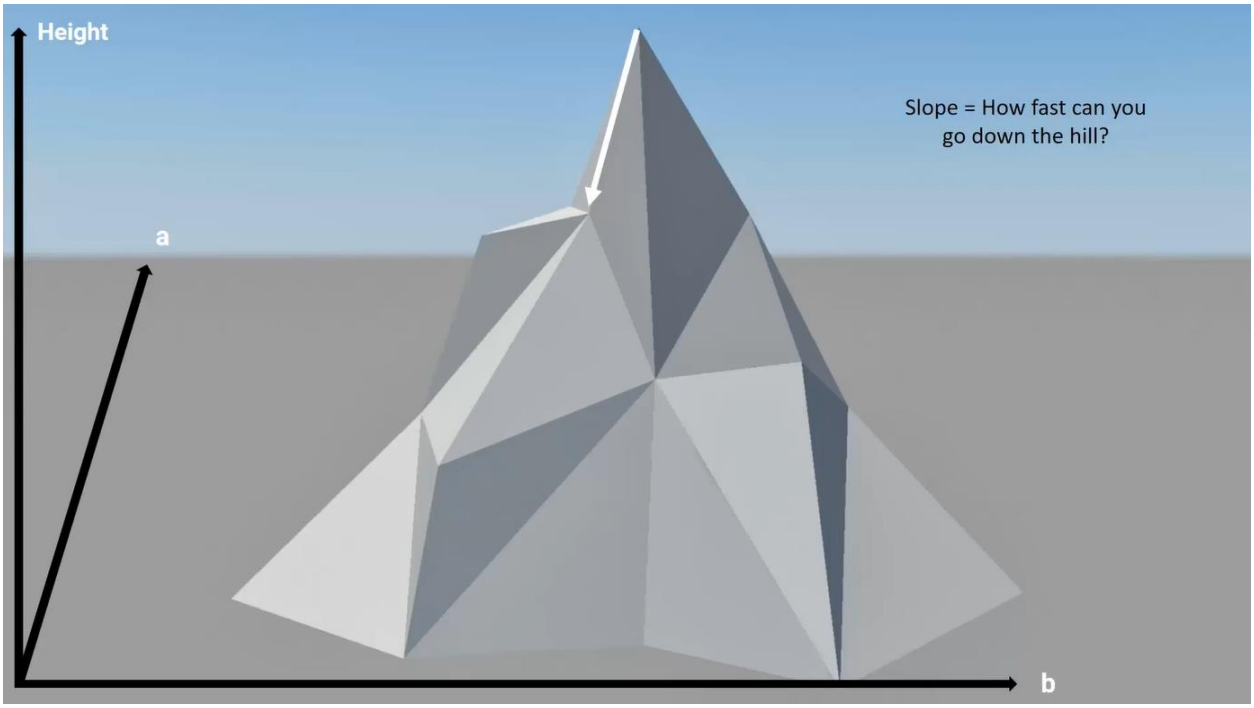
$$Error = \frac{(Y - \hat{Y})^2}{2}$$

What can be changed to Reduce Error?

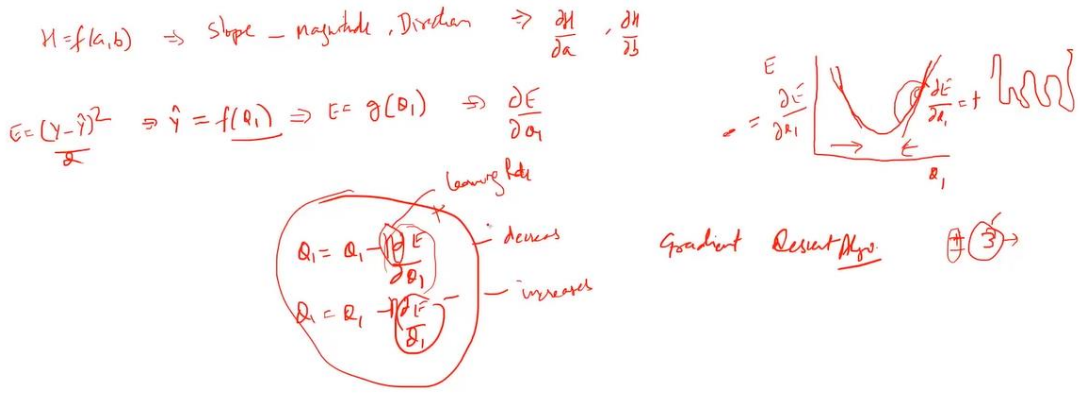


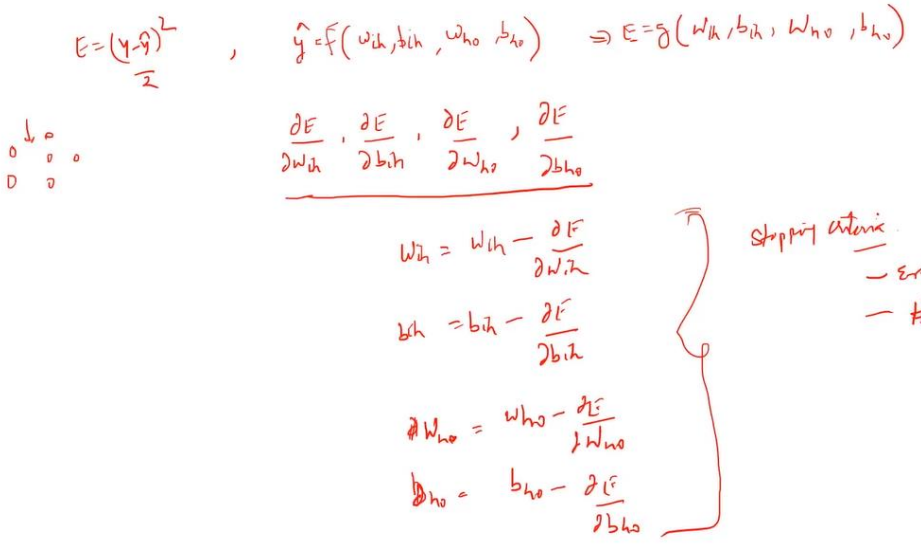


Gradient Descent Intuition



Understanding Math Behind Gradient Descent





QUESTION 3 OF 4

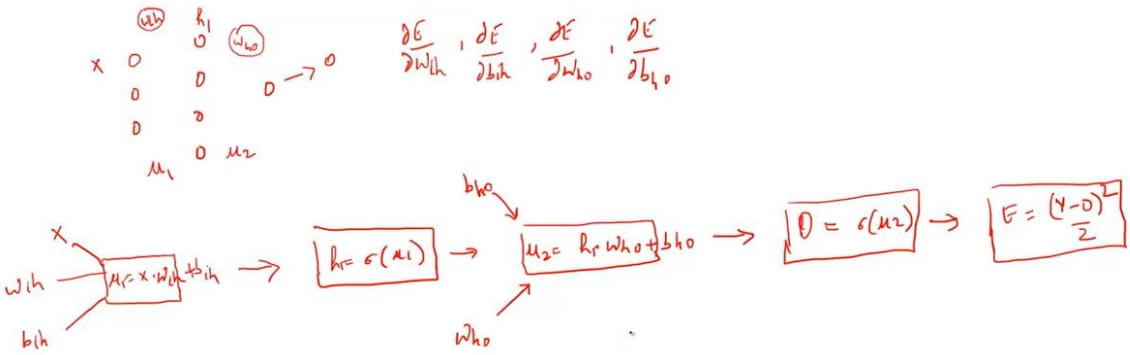
What is the update equation of gradient descent?
Note: Here, Ø is parameter and E is error function.

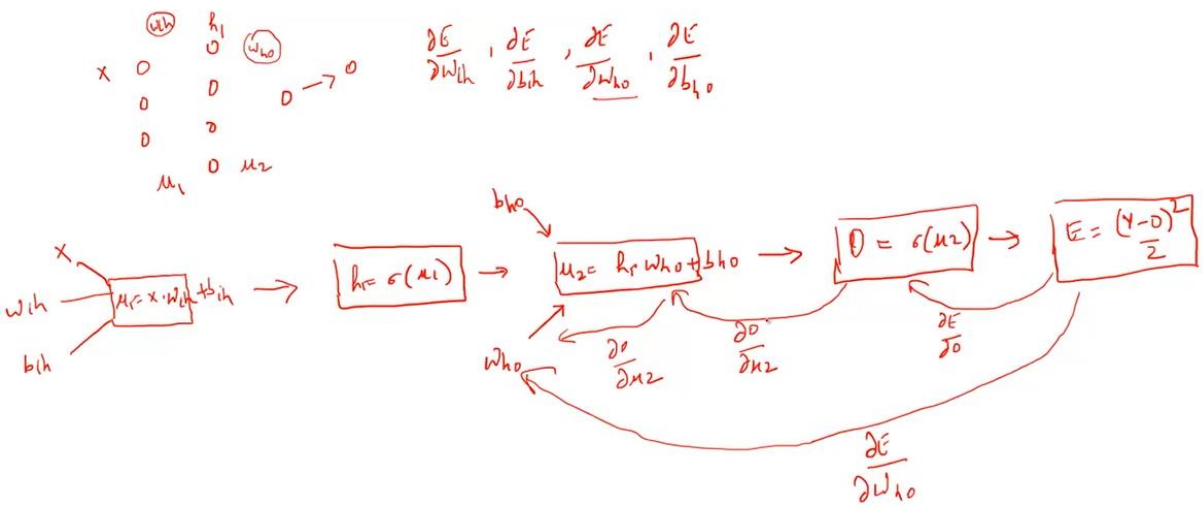
Choose only ONE best answer.

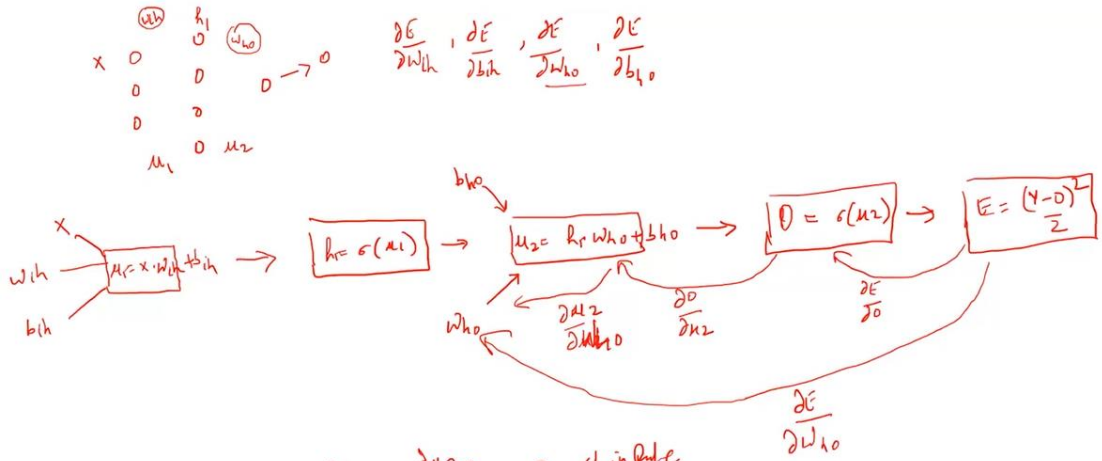
A	$Ø = Ø^2 + (d(E)/dØ)^2$
B	$Ø = Ø - d(E)/dØ$ ✓
C	$Ø = Ø * d(E)/dØ$
D	$Ø = d(E)/dØ$

This answer is correct.

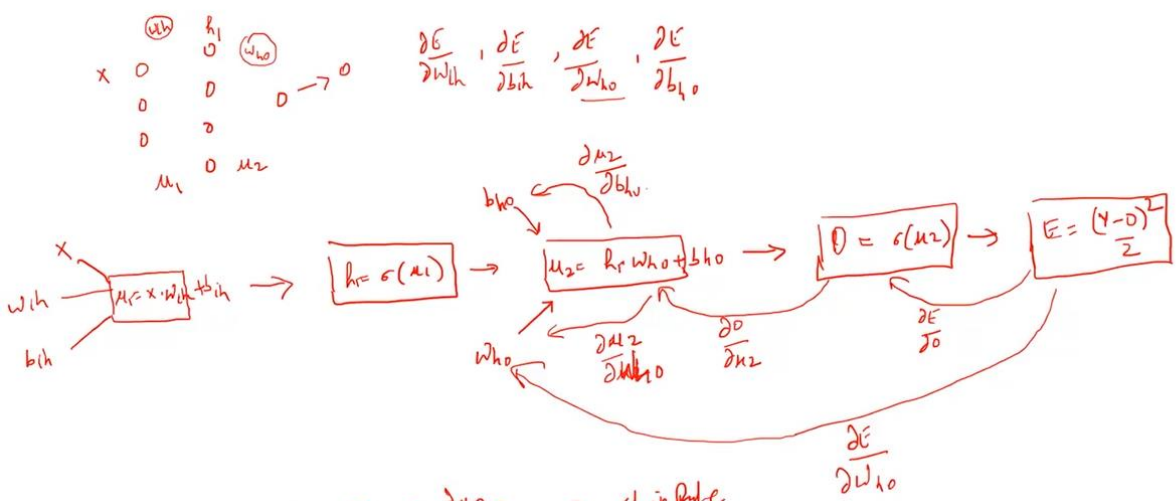
Back Propagation





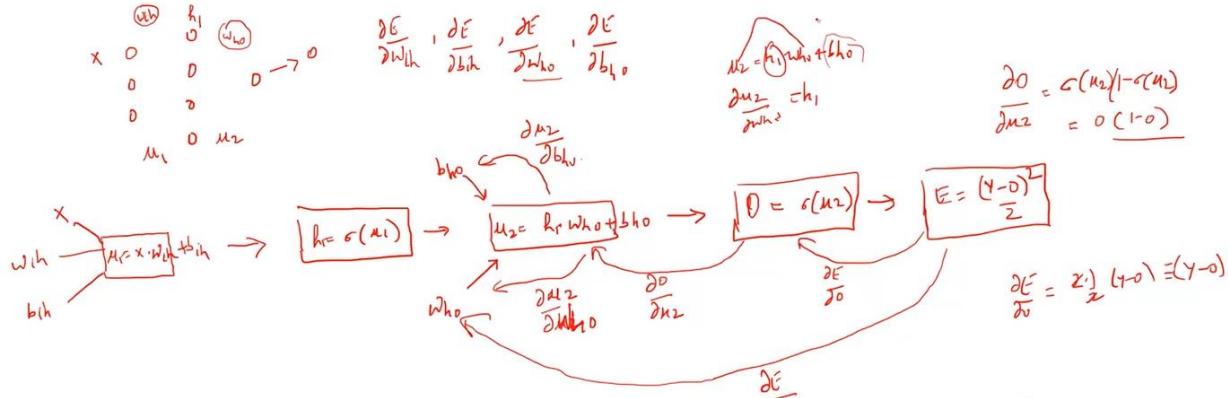


$$\frac{\partial E}{\partial w_{ho}} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial u_2} \times \frac{\partial u_2}{\partial w_{ho}} \rightarrow \text{chain Rule}$$



$$\frac{\partial E}{\partial w_{ho}} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial \mu_2} \times \frac{\partial \mu_2}{\partial w_{ho}} \rightarrow \text{chain Rule}$$

$$\frac{\partial E}{\partial b_{ho}} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial \mu_2} \times \frac{\partial \mu_2}{\partial b_{ho}}$$

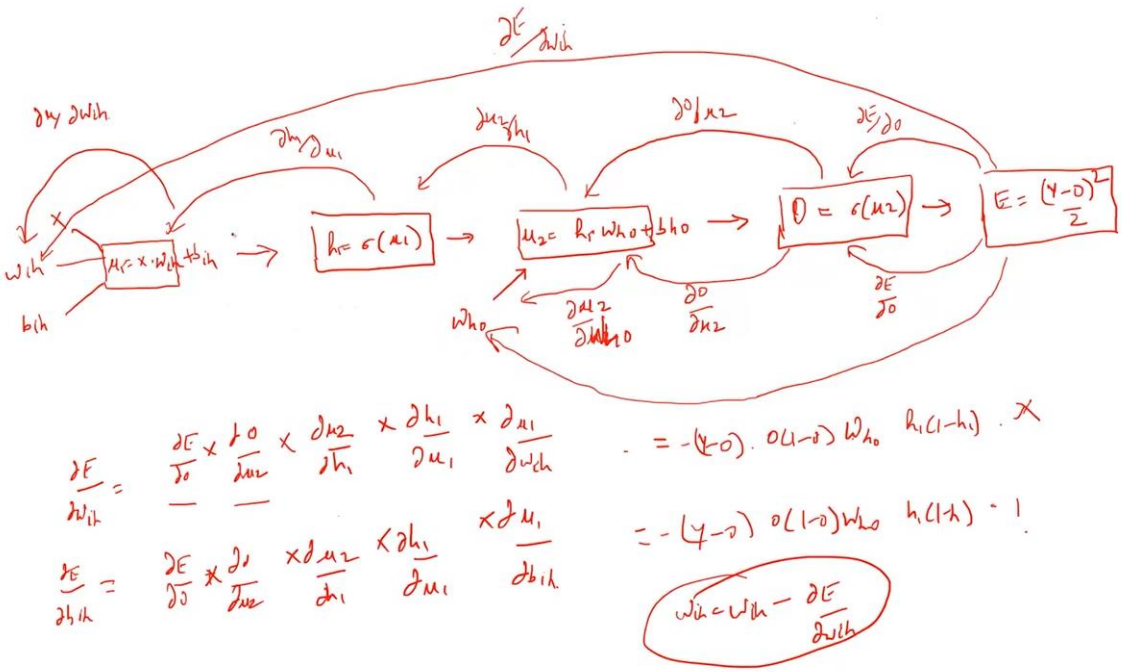


$$\frac{\partial E}{\partial w_{ho}} \Rightarrow -(Y-O) \cdot O(1-O) \cdot h_1$$

$$-(Y-O)\,O(1-O) \cdot$$

$$\frac{\partial E}{\partial w_{ho}} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial \mu_2} \times \frac{\partial \mu_2}{\partial w_{ho}} \rightarrow \text{chain Rule}$$

$$\frac{\partial E}{\partial b_{ho}} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial \mu_2} \times \frac{\partial \mu_2}{\partial b_{ho}}$$



Summary of the Module

Enable fullscreen

In this module, we understood the concepts of neural networks. We discussed various components of a neural network like neurons, layers, activation functions. We also learnt what is forward and backward propagation and how this helps to build a neural network.

Neural networks from the foundation of Deep Learning and we can make use them in the following fields:

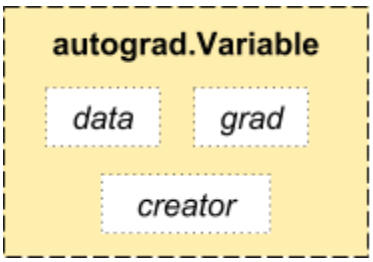
1. Computer Vision
2. Natural Language Processing
3. Audio processing

Next, we will implement a simple neural network from scratch in PyTorch.

Modules in PyTorch - Autograd

Autograd module

PyTorch uses a technique called automatic differentiation. That is, we have a recorder that records what operations we have performed, and then it replays it backward to compute our gradients. This technique is especially powerful when building neural networks, as we save time on one epoch by calculating the differentiation of the parameters at the forward pass itself.



Source: http://pytorch.org/about/

Let's look at an example to understand how the gradients are computed:

```
# initializing a tensor
a = torch.ones((2,2), requires_grad=True)
a
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

Here, we have initialized a tensor. Specifying *requires_grad* as True will make sure that the gradients are stored for this particular tensor whenever we perform some operation on it. Let's now perform some operations on the defined tensor:

```
# performing operations on the tensor
b = a + 5
c = b.mean()
print(b,c)
```

```
tensor([[6., 6.],
        [6., 6.]], grad_fn=<AddBackward0>) tensor(6., grad_fn=<MeanBackward1>)
```

First of all, we added 5 to all the elements of this tensor and then taken the mean of that tensor. We will first manually calculate the gradients and then verify that using PyTorch. We performed the following operations on *a*:

$$b = a + 5$$

$$c = mean(b) = \Sigma(a+5) / 4$$

Now, the derivative of c w.r.t. a will be ¼ and hence the gradient matrix will be 0.25. Let's verify this using PyTorch:

```
# back propagating
c.backward()

# computing gradients
print(a.grad)
```

```
tensor([[0.2500, 0.2500],
        [0.2500, 0.2500]])
```

As expected, we have the gradients.

Modules in PyTorch: Optim

The Optim module in PyTorch has pre-written codes for most of the optimizers that are used while building a neural network. We just have to import them and then they can be used to build models.

Let's see how we can use an optimizer in PyTorch:

```
# importing the optim module
from torch import optim

# adam
## adam = optim.Adam(model.parameters(), lr=learning_rate)

# sgd
## SGD = optim.SGD(model.parameters(), lr=learning_rate)
```

Above are the examples to get the ADAM and SGD optimizers. Most of the commonly used optimizers are supported in PyTorch and hence we do not have to write them from scratch. Some of them are:

- SGD
- Adam
- Adadelta
- Adagrad
- AdamW

- SparseAdam
- Adamax
- ASGD (Averaged Stochastic Gradient Descent)
- RMSprop
- Rprop (resilient backpropagation)

Modules in PyTorch: nn

The autograd module in PyTorch helps us define computation graphs as we proceed in the model. But, just using the autograd module can be low-level when we are dealing with a complex neural network.

In those cases, we can make use of the nn module. This defines a set of functions, similar to the layers of a neural network, which takes the input from the previous state and produces an output.

We will use all these modules and define our neural network to solve a case study in the later sections. For now, let's build a neural network from scratch that will help us understand how PyTorch works in a practical way.

Implementing a Neural Network from Scratch

Time to get started with neural networks! This is going to be a lot of fun so let's get right down to it. We will first initialize the input and output:

```
#Input tensor
X = torch.Tensor([[1,0,1,0],[1,0,1,1],[0,1,0,1]])

#Output
y = torch.Tensor([[1],[1],[0]])

print(X, '\n')
print(y)
```

```
tensor([[1., 0., 1., 0.],
        [1., 0., 1., 1.],
        [0., 1., 0., 1.]])

tensor([[1.],
        [1.],
        [0.]])
```

Next, we will define the sigmoid function which will act as the activation function and the derivative of the sigmoid function which will help us in the backpropagation step:

```
#Sigmoid Function
def sigmoid (x):
   return 1/(1 + torch.exp(-x))

#Derivative of Sigmoid Function/
def derivatives_sigmoid(x):
   return sigmoid(x) * (1 - sigmoid(x))
```

Next, initialize the parameters for our model including the number of epochs, learning rate, weights, biases, etc.:

```
#Variable initialization

epoch=7000 #Setting training iterations

lr=0.1 #Setting learning rate

inputlayer_neurons = X.shape[1] #number of features in data set

hiddenlayer_neurons = 3 #number of hidden layer neurons

output_neurons = 1 #number of neurons in output layer

#weight and bias initialization
wh=torch.randn(inputlayer_neurons,
hiddenlayer_neurons).type(torch.FloatTensor)
bh=torch.randn(1, hiddenlayer_neurons).type(torch.FloatTensor)
wout=torch.randn(hiddenlayer_neurons, output_neurons)
bout=torch.randn(1, output_neurons)
```

Here we have randomly initialized the weights and biases using the *.randn()* function which we saw earlier. Finally, we will create a neural network. I am taking a simple model here just to make things clear. There is a single hidden layer and an input and an output layer in the model:

```
for i in range(epoch):
   #Forward Propogation
   hidden_layer_input1 = torch.mm(X, wh)
   hidden_layer_input = hidden_layer_input1 + bh
```

```
    hidden_layer_activations = sigmoid(hidden_layer_input)

    output_layer_input1 = torch.mm(hidden_layer_activations, wout)
    output_layer_input = output_layer_input1 + bout
    output = sigmoid(output_layer_input)

    #Backpropagation
    E = y-output
    slope_output_layer = derivatives_sigmoid(output)
    slope_hidden_layer = derivatives_sigmoid(hidden_layer_activations)
    d_output = E * slope_output_layer
    Error_at_hidden_layer = torch.mm(d_output, wout.t())
    d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer

    wout += torch.mm(hidden_layer_activations.t(), d_output) *lr
    bout += d_output.sum() *lr
    wh += torch.mm(X.t(), d_hiddenlayer) *lr
    bh += d_output.sum() *lr
```

In the forward propagation step, we are calculating the output and finally, in the backward propagation step, we are calculating the error. We will then update the weights and biases using this error.

Let's now look at the output from the model:

```
print('actual :\n', y, '\n')
print('predicted :\n', output)
```

```
actual :
 tensor([[1.],
        [1.],
        [0.]])

predicted :
 tensor([[0.9834],
        [0.9780],
        [0.0330]])
```

So, the target is 1, 1, 0 and the predicted values from the model are 0.98, 0.97 and 0.03. Not bad at all!

This is how we can build and train a neural network from scratch in PyTorch. Let's now take things up a notch and dive into a case study. We will try to solve that case study using the techniques we have learned in this article.

Case Study – Solving an Image Recognition problem in PyTorch

Enable fullscreen

To get familiar with PyTorch, we will solve Analytics Vidhya's deep learning practice problem – Identify the Digits. Let's take a look at our problem statement:

Our problem is an image recognition problem, to identify digits from a given 28 x 28 image. We have a subset of images for training and the rest for testing our model.

So first, download the train and test files. The dataset contains a zipped file of all the images and both the train.csv and test.csv have the name of the corresponding train and test images. Any additional features are not provided in the datasets, just the raw images are provided in '.png' format.

Let's begin:

STEP 0: Getting Ready

a) Import all the necessary libraries

```
# import modules
%pylab inline
import os import numpy as np
import pandas as pd
from scipy.misc import imread
from sklearn.metrics import accuracy_score
```

b) Let's set a seed value, so that we can control our models randomness

```
# To stop potential randomness
seed = 128
rng = np.random.RandomState(seed)
```

c) The first step is to set directory paths, for safekeeping!

```
root_dir = os.path.abspath('.')
data_dir = os.path.join(root_dir, 'data')
# check for existence
os.path.exists(root_dir), os.path.exists(data_dir)
```

STEP 1: Data Loading and Preprocessing

a) Now let us read our datasets. These are in .csv formats and have a filename along with the appropriate labels

```
# load dataset
train = pd.read_csv(os.path.join(data_dir, 'Train', 'train.csv'))
test = pd.read_csv(os.path.join(data_dir, 'Test.csv'))

sample_submission = pd.read_csv(os.path.join(data_dir,
'Sample_Submission.csv'))

train.head()
```

	filename	label
0	0.png	4
1	1.png	9
2	2.png	1
3	3.png	7
4	4.png	3

b) Let us see what our data looks like! We read our image and display it.

```
# print an image
img_name = rng.choice(train.filename)
filepath = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)

img = imread(filepath, flatten=True)

pylab.imshow(img, cmap='gray')
pylab.axis('off')
pylab.show()
```

d) For easier data manipulation, let's store all our images as numpy arrays

```
# load images to create train and test set
temp = []
for img_name in train.filename:
    image_path = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)
    img = imread(image_path, flatten=True)
    img = img.astype('float32')
    temp.append(img

train_x = np.stack(temp)
train_x /= 255.0
train_x = train_x.reshape(-1, 784).astype('float32')

temp = []
for img_name in test.filename:
    image_path = os.path.join(data_dir, 'Train', 'Images', 'test', img_name)
    img = imread(image_path, flatten=True)
    img = img.astype('float32')
    temp.append(img) test_x = np.stack(temp)

test_x /= 255.0
test_x = test_x.reshape(-1, 784).astype('float32')

train_y = train.label.values
```

e) As this is a typical ML problem, to test the proper functioning of our model we create a validation set. Let's take a split size of 70:30 for train set vs validation set

```
# create validation set
```

```
split_size = int(train_x.shape[0]*0.7)

train_x, val_x = train_x[:split_size], train_x[split_size:]
train_y, val_y = train_y[:split_size], train_y[split_size:]
```

STEP 2: Model Building

a) Now comes the main part! Let us define our neural network architecture. We define a neural network with 3 layers input, hidden and output. The number of neurons in input and output are fixed, as the input is our 28 x 28 image and the output is a 10 x 1 vector representing the class. We take 50 neurons in the hidden layer. Here, we use Adam as our optimization algorithms, which is an efficient variant of Gradient Descent algorithm.

```
import torch from torch.autograd import Variable
# number of neurons in each layer
input_num_units = 28*28

hidden_num_units = 500

output_num_units = 10

# set remaining variables
epochs = 5
batch_size = 128
learning_rate = 0.001
```

b) It's time to train our model

```
# define model
model = torch.nn.Sequential(
    torch.nn.Linear(input_num_units, hidden_num_units),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_num_units, output_num_units),
)
loss_fn = torch.nn.CrossEntropyLoss()
# define optimization algorithm
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
## helper functions

# preprocess a batch of dataset
def preproc(unclean_batch_x):
   """Convert values to range 0-1"""
   temp_batch = unclean_batch_x / unclean_batch_x.max()

   return temp_batch

# create a batch
def batch_creator(batch_size):
    dataset_name = 'train'
    dataset_length = train_x.shape[0]

    batch_mask = rng.choice(dataset_length, batch_size)
    batch_x = eval(dataset_name + '_x')[batch_mask]
```

```
    batch_x = preproc(batch_x)

    if dataset_name == 'train':
       batch_y = eval(dataset_name).ix[batch_mask, 'label'].values

    return batch_x, batch_y
# train network
total_batch = int(train.shape[0]/batch_size)

for epoch in range(epochs):
    avg_cost = 0
    for i in range(total_batch):
       # create batch
       batch_x, batch_y = batch_creator(batch_size)

       # pass that batch for training
       x, y = Variable(torch.from_numpy(batch_x)),
Variable(torch.from_numpy(batch_y), requires_grad=False)
       pred = model(x)

       # get loss
       loss = loss_fn(pred, y)

       # perform backpropagation
       loss.backward()
       optimizer.step()

       avg_cost += loss.data[0]/total_batch
    print(epoch, avg_cost)
# get training accuracy
x, y = Variable(torch.from_numpy(preproc(train_x))),
Variable(torch.from_numpy(train_y), requires_grad=False)

pred = model(x)
final_pred = np.argmax(pred.data.numpy(), axis=1)

accuracy_score(train_y, final_pred)
# get validation accuracy
x, y = Variable(torch.from_numpy(preproc(val_x))),
Variable(torch.from_numpy(val_y), requires_grad=False)

pred = model(x)
final_pred = np.argmax(pred.data.numpy(), axis=1)

accuracy_score(val_y, final_pred)
```

The training score comes out to be:

```
0.8779008746355685
```

whereas, the validation score is:

```
0.867482993197279
```

This is a pretty impressive score especially when we have trained a very simple neural network for just five epochs!

Other Use cases for Deep Learning in PyTorch

Enable fullscreen

In this course, we understood the basic concepts of PyTorch including how it's quite intuitively similar to NumPy. We also saw how to build a neural network from scratch using PyTorch.

We then took a case study where we solved an image classification problem and got a benchmark score of around 87% on the leaderboard. We encourage you to try and improve this score by changing different parameters of the model, including the optimizer function, increasing the number of hidden layers, tuning the number of hidden units, etc.

This course is the start of our journey with PyTorch. You can study the following use cases of Deep Learning on different kinds of data using PyTorch

1. Build Your First Text Classification model using PyTorch
2. Build an Image Classification Model using Convolutional Neural Networks in PyTorch
3. Image Augmentation for Deep Learning using PyTorch – Feature Engineering for Images
4. Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch

1. text classification using pytorch

Build Your First Text Classification model using PyTorch

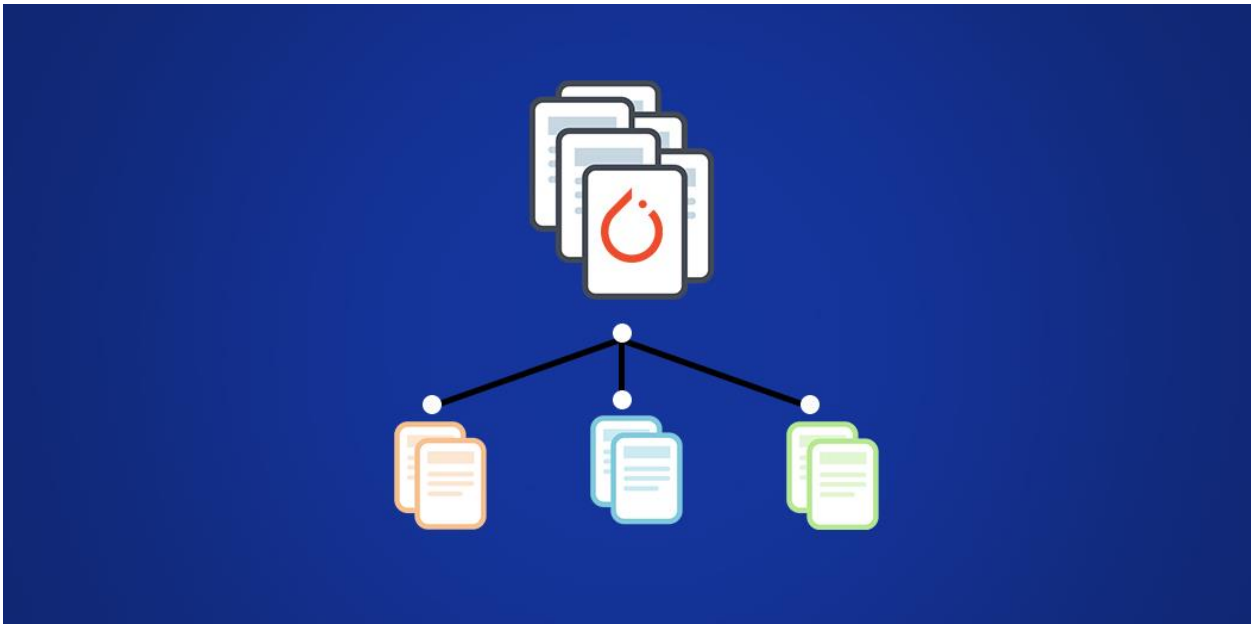
ARAVIND PAI, JANUARY 28, 2020

Overview

- Learn how to perform text classification using PyTorch
- Grasp the importance of Pack Padding feature
- Understand the key points involved while solving text classification

Introduction

I always turn to State of the Art architectures to make my first submission in data science hackathons. Implementing the State of the Art architectures has become quite easy thanks to deep learning frameworks such as PyTorch, Keras, and TensorFlow. These frameworks provide an easy way to implement complex model architectures and algorithms with least knowledge of concepts and coding skills. In short, it's a goldmine for the data science community!



In this article, we will use PyTorch, which is well known for its fast computational power. So in this article, we will walk through the key points for solving a text classification problem. And then we will implement our first text classifier in PyTorch!

Note: I highly recommend to go through the below article before moving forward with this article.

- *A Beginner-Friendly Guide to PyTorch and How it Works from Scratch*

Table of Contents

1.Why PyTorch for Text Classification?

- Dealing with Out of Vocabulary words
- Handling Variable Length sequences
- Wrappers and Pre-trained models

2.Understanding the Problem Statement
3.Implementation – Text Classification in PyTorch

Why PyTorch for Text Classification?

Before we dive deeper into the technical concepts, let us quickly familiarize ourselves with the framework that we are going to use – PyTorch. The basic unit of PyTorch is Tensor, similar to the “numpy” array in python. There are a number of benefits for using PyTorch but the two most important are:

- Dynamic networks – Change in the architecture during the run time
- Distributed training across GPUs

I am sure you are wondering – why should we use PyTorch for working with text data? Let us discuss some incredible features of PyTorch that makes it different from other frameworks, especially while working with text data.

1. Dealing with Out of Vocabulary words

A text classification model is trained on fixed vocabulary size. But during inference, we might come across some words which are not present in the vocabulary. These words are known as **Out of Vocabulary** words. Skipping Out of Vocabulary words can be a critical issue as this results in the loss of information.

In order to handle the Out Of Vocabulary words, PyTorch supports a cool feature that replaces the rare words in our training data with Unknown token. This, in turn, helps us in tackling the problem of Out of Vocabulary words.

Apart from handling Out Of Vocabulary words, PyTorch also has a feature that can handle sequences of variable length!

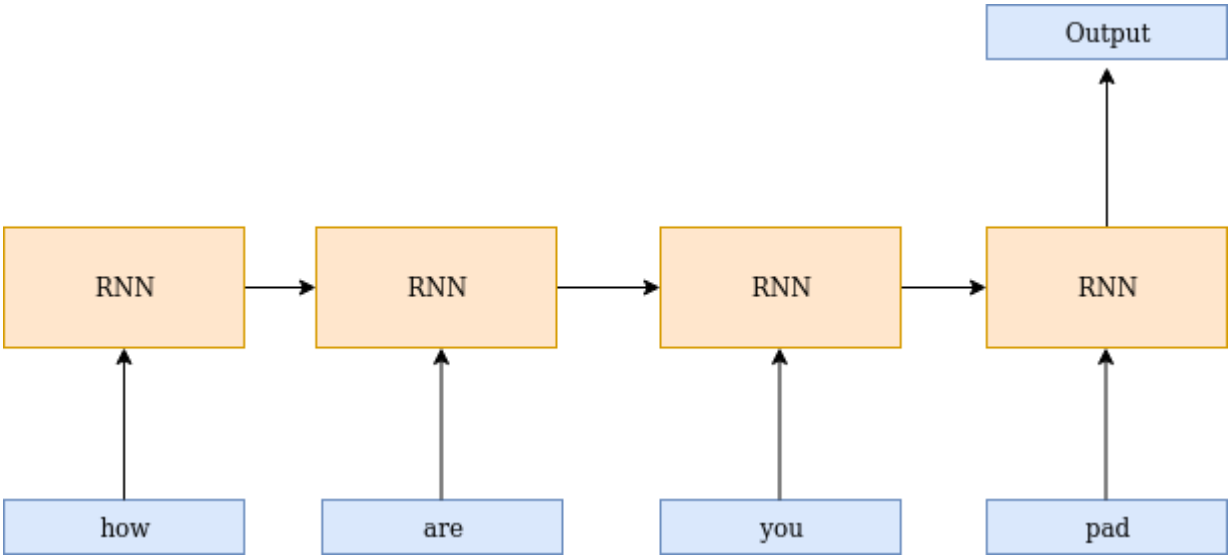
2. Handling Variable Length sequences

Have you heard of how Recurrent Neural Network is capable of handling variable-length sequences? Ever wondered how to implement it? PyTorch comes with a useful feature ‘**Packed Padding sequence**‘ that implements Dynamic Recurrent Neural Network.

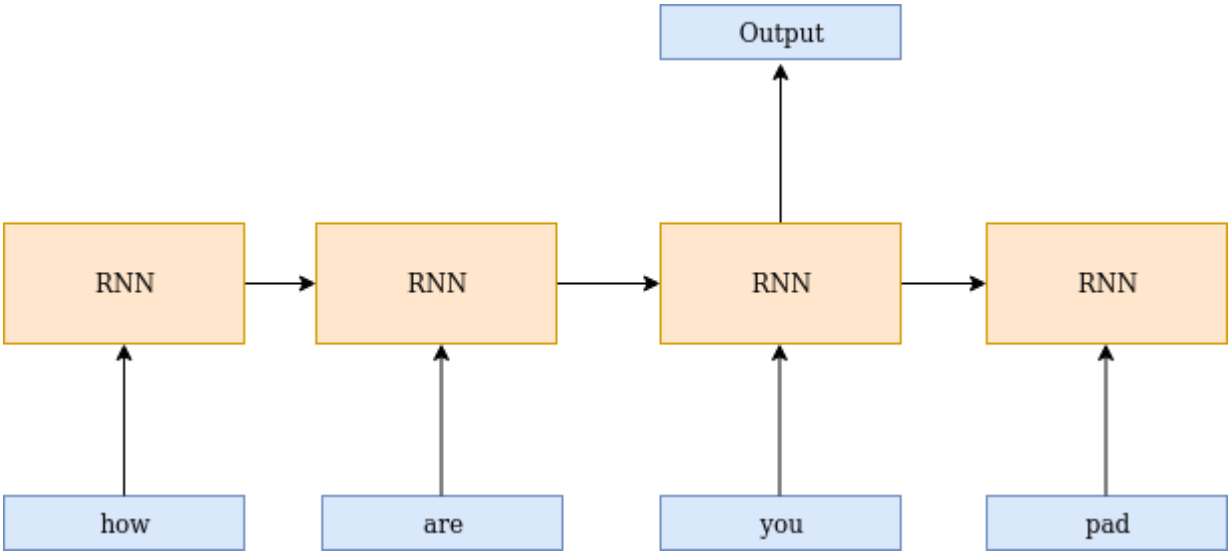
Padding is a process of adding an extra token called *padding token* at the beginning or end of the sentence. As the number of the words in each sentence varies, we convert the variable length input sentences into sentences with the same length by adding *padding tokens*.

Padding is required since most of the frameworks support static networks, i.e. the architecture remains the same throughout the model training. Although padding solves the issue of variable length sequences, there is another problem with this idea – the architectures now process these *padding token* like any other information/data. Let me explain this through a simple diagram-

As you can see in the diagram (below), the last element, which is a *padding token* is also used while generating the output. This is taken care of by the **Packed Padding sequence** in PyTorch.



Packed padding ignores the input timesteps with *padding token*. These values are never shown to the Recurrent Neural Network which helps us in building a dynamic Recurrent Neural Network.



3. Wrappers and Pretrained models

The state of the art architectures are being launched for PyTorch framework. Hugging Face released Transformers which provides more than 32 state of the art architectures for the Natural Language Understanding Generation!

Not only this, PyTorch also provides pretrained models for several tasks like Text to Speech, Object Detection and so on, which can be executed within few lines of code.

Incredible, isn't it? These are some really useful features of PyTorch among many others. Let us now use PyTorch for a text classification problem.

Understanding the problem statement

As a part of this article, we are going to work on a really interesting problem.

Quora wants to keep track of insincere questions on their platform so as to make users feel safe while sharing their knowledge. An insincere question in this context is defined as a question intended to make a statement rather than looking for helpful answers. To break this down further, here are some characteristics that can signify that a particular question is insincere:

- Has a non-neutral tone
- Is disparaging or inflammatory
- Isn't grounded in reality
- Uses sexual content (incest, bestiality, pedophilia) for shock value, and not to seek genuine answers

The training data includes the question that was asked, and a flag denoting whether it was identified as insincere (target = 1). The ground-truth labels contain some amount of noise, i.e. they are not guaranteed to be perfect. Our task will be to identify if a given question is 'insincere'. You can download the dataset for this from here.

It is time to code our own text classification model using PyTorch.

Implementation – Text Classification in PyTorch

Let us first import all the necessary libraries required to build a model. Here is a brief overview of the packages/libraries we are going to use-

- **Torch** package is used to define tensors and mathematical operations on it
- **TorchText** is a Natural Language Processing (NLP) library in PyTorch. This library contains the scripts for preprocessing text and source of few popular NLP datasets.

```
#deal with tensors

import torch



#handling text data
```

```
from torchtext import data
```

view rawlibraries.py hosted with by GitHub

In order to make the results reproducible, I have specified the seed value. Since Deep Learning model might produce different results each when it is executed due to the randomness in it, it is important to specify the seed value.

```
#Reproducing same results

SEED = 2019



#Torch

torch.manual_seed(SEED)



#Cuda algorithms

torch.backends.cudnn.deterministic = True
```

view rawseed.py hosted with by GitHub

Pre-processsing Data:

Now, let us see how to preprocess the text using field objects. There are 2 different types of field objects – Field and LabelField. Let us quickly understand the difference between the two-

1. **Field**: Field object from data module is used to specify preprocessing steps for each column in the dataset.
2. **LabelField**: LabelField object is a special case of Field object which is used only for the classification tasks. Its only use is to set the *unk_token* and sequential to None by default.

Before we use Field, let us look at the different parameters of Field and what are they used for.

Parameters of Field:

- **Tokenize**: specifies the way of tokenizing the sentence i.e. converting sentence to words. I am using spacy tokenizer since it uses novel tokenization algorithm
- **Lower**: converts text to lowercase

- **batch_first**: The first dimension of input and output is always batch size

```
TEXT = data.Field(tokenize='spacy',batch_first=True,include_lengths=True)


LABEL = data.LabelField(dtype = torch.float,batch_first=True)
```

Next we are going to create a list of tuples where first value in every tuple contains a column name and second value is a field object defined above. Furthermore we will arrange each tuple in the order of the columns of csv, and also specify as (None,None) to ignore a column from a csv file.

Let us read only required columns – question and label

```
fields = [(None, None), ('text',TEXT),('label', LABEL)]
```

In the following code block I have loaded the custom dataset by defining the field objects.

```
#loading custom dataset

training_data=data.TabularDataset(path = 'quora.csv',format = 'csv',fields =
fields,skip_header = True)


#print preprocessed text

print(vars(training_data.examples[0]))
```

view rawload.py hosted with by GitHub

Let us now split the dataset into training and validation data

```
import random

train_data, valid_data = training_data.split(split_ratio=0.7, random_state =
random.seed(SEED))
```

view rawtrain_test.py hosted with by GitHub

Preparing input and output sequences:

The next step is to build the vocabulary for the text and convert them into integer sequences. Vocabulary contains the unique words in the entire text. Each unique word is assigned an index. Below are the parameters listed for the same

Parameters:

1. **min_freq**: Ignores the words in vocabulary which has frequency less than specified one and map it to unknown token.

2. Two special tokens known as unknown and padding will be added to the vocabulary

- **Unknown** token is used to handle Out Of Vocabulary words
- **Padding** token is used to make input sequences of same length

Let us build vocabulary and initialize the words with the pretrained embeddings. Ignore the vectors parameter if you wish to randomly initialize embeddings.

```
#initialize glove embeddings

TEXT.build_vocab(train_data,min_freq=3,vectors = "glove.6B.100d")

LABEL.build_vocab(train_data)


#No. of unique tokens in text

print("Size of TEXT vocabulary:",len(TEXT.vocab))


#No. of unique tokens in label

print("Size of LABEL vocabulary:",len(LABEL.vocab))


#Commonly used words

print(TEXT.vocab.freqs.most_common(10))


#Word dictionary

print(TEXT.vocab.stoi)
```

Now we will prepare batches for training the model. *BucketIterator* forms the batches in such a way that a minimum amount of padding is required.

```
#check whether cuda is available

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


#set batch size

BATCH_SIZE = 64


#Load an iterator

train_iterator, valid_iterator = data.BucketIterator.splits(

    (train_data, valid_data),

    batch_size = BATCH_SIZE,

    sort_key = lambda x: len(x.text),

    sort_within_batch=True,

    device = device)
```

view rawiterator.py hosted with by GitHub

Model Architecture

It is now time to define the architecture to solve the binary classification problem. The nn module from torch is a base model for all the models. This means that every model must be a subclass of the nn module.

I have defined 2 functions here: init as well as forward. Let me explain the use case of both of these functions-

1. **Init**: Whenever an instance of a class is created, init function is automatically invoked. Hence, it is called as a constructor. The arguments passed to the class are initialized by the constructor.We will define all the layers that we will be using in the model

2. **Forward**: Forward function defines the forward pass of the inputs.

Finally, let's understand in detail about the different layers used for building the architecture and their parameters-

Embedding layer: Embeddings are extremely important for any NLP related task since it represents a word in a numerical format. Embedding layer creates a look up table where each row represents an embedding of a word. The embedding layer converts the integer sequence into a dense vector representation. Here are the two most important parameters of the embedding layer –

1. num_embeddings: No. of unique words in dictionary
2. embedding_dim: No. of dimensions for representing a word

LSTM: LSTM is a variant of RNN that is capable of capturing long term dependencies. Following the some important parameters of LSTM that you should be familiar with. Given below are the parameters of this layer:

1. input_size : Dimension of input
2. hidden_size : Number of hidden nodes
3. num_layers : Number of layers to be stacked
4. batch_first : If True, then the input and output tensors are provided as (batch, seq, feature)
5. dropout: If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0
6. bidirection: If True, introduces a Bi directional LSTM

Linear Layer: Linear layer refers to dense layer. The two important parameters here are described below:

1. in_features : No. of input features
2. out_features: No. of hidden nodes

Pack Padding: As already discussed, pack padding is used to define the dynamic recurrent neural network. Without pack padding, the padding inputs are also processed by the rnn and returns the hidden state of the padded element. This an awesome wrapper that does not show the inputs that are padded. It simply ignores the values and returns the hidden state of the non padded element.
Now that we have a good understanding of all the blocks of the architecture, let us go to the code!

I will start with defining all the layers of the architecture:

```
import torch.nn as nn


class classifier(nn.Module):


    #define all the layers used in model
```

```
def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
             bidirectional, dropout):

    #Constructor
    super().__init__()

    #embedding layer
    self.embedding = nn.Embedding(vocab_size, embedding_dim)

    #lstm layer
    self.lstm = nn.LSTM(embedding_dim,
                        hidden_dim,
                        num_layers=n_layers,
                        bidirectional=bidirectional,
                        dropout=dropout,
                        batch_first=True)

    #dense layer
    self.fc = nn.Linear(hidden_dim * 2, output_dim)

    #activation function
    self.act = nn.Sigmoid()

def forward(self, text, text_lengths):

    #text = [batch size,sent_length]
    embedded = self.embedding(text)
    #embedded = [batch size, sent_len, emb dim]
```

```
        #packed sequence

        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded,
text_lengths,batch_first=True)


        packed_output, (hidden, cell) = self.lstm(packed_embedded)

        #hidden = [batch size, num layers * num directions,hid dim]

        #cell = [batch size, num layers * num directions,hid dim]


        #concat the final forward and backward hidden state

        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)


        #hidden = [batch size, hid dim * num directions]

        dense_outputs=self.fc(hidden)


        #Final activation function

        outputs=self.act(dense_outputs)


        return outputs
```

view rawarchitecture.py hosted with by GitHub

The next step would be to define the hyperparameters and instantiate the model. Here is the code block for the same:

```
#define hyperparameters

size_of_vocab = len(TEXT.vocab)

embedding_dim = 100

num_hidden_nodes = 32

num_output_nodes = 1

num_layers = 2
```

```
bidirection = True

dropout = 0.2


#instantiate the model

model = classifier(size_of_vocab, embedding_dim, num_hidden_nodes,num_output_nodes,
num_layers,

                   bidirectional = True, dropout = dropout)
```

view rawinstantiate.py hosted with by GitHub

Let us look at the model summary and initialize the embedding layer with the pretrained embeddings

```
#architecture

print(model)


#No. of trianable parameters

def count_parameters(model):

    return sum(p.numel() for p in model.parameters() if p.requires_grad)


print(f'The model has {count_parameters(model):,} trainable parameters')


#Initialize the pretrained embedding

pretrained_embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy_(pretrained_embeddings)


print(pretrained_embeddings.shape)
```

view rawarch.py hosted with by GitHub

Here I have defined the optimizer, loss and metric for the model:

```
import torch.optim as optim
```

```
#define optimizer and loss

optimizer = optim.Adam(model.parameters())

criterion = nn.BCELoss()


#define metric

def binary_accuracy(preds, y):

    #round predictions to the closest integer

    rounded_preds = torch.round(preds)


    correct = (rounded_preds == y).float()

    acc = correct.sum() / len(correct)

    return acc


#push to cuda if available

model = model.to(device)

criterion = criterion.to(device)
```

view rawoptimizers.py hosted with by GitHub

There are 2 phases while building the model:

1. Training phase: model.train() sets the model on the training phase and activates the dropout layers.
2. Inference phase: model.eval() sets the model on the evaluation phase and deactivates the dropout layers.

Here is the code block to define a function for training the model

```
def train(model, iterator, optimizer, criterion):


    #initialize every epoch

    epoch_loss = 0
```

```
epoch_acc = 0

#set the model in training phase
model.train()

for batch in iterator:

    #resets the gradients after every batch
    optimizer.zero_grad()

    #retrieve text and no. of words
    text, text_lengths = batch.text

    #convert to 1D tensor
    predictions = model(text, text_lengths).squeeze()

    #compute the loss
    loss = criterion(predictions, batch.label)

    #compute the binary accuracy
    acc = binary_accuracy(predictions, batch.label)

    #backpropage the loss and compute the gradients
    loss.backward()

    #update the weights
    optimizer.step()
```

```
            #loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()

        return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

view rawtrain.py hosted with by GitHub

So we have a function to train the model, but we will also need a function to evaluate the mode. Let's do that

```
def evaluate(model, iterator, criterion):

    #initialize every epoch
    epoch_loss = 0
    epoch_acc = 0

    #deactivating dropout layers
    model.eval()

    #deactivates autograd
    with torch.no_grad():

        for batch in iterator:

            #retrieve text and no. of words
            text, text_lengths = batch.text

            #convert to 1d tensor
            predictions = model(text, text_lengths).squeeze()
```

```
            #compute loss and accuracy
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)

            #keep track of loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

view rawvalid.py hosted with by GitHub

Finally we will train the model for a certain number of epochs and save the best model every epoch.

```
N_EPOCHS = 5
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    #train the model
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)

    #evaluate the model
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    #save the best model
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
```

```
            torch.save(model.state_dict(), 'saved_weights.pt')


        print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
        print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

view rawrun.py hosted with by GitHub

Let us load the best model and define the inference function that accepts the user defined input and make predictions

```
#load weights
path='/content/saved_weights.pt'
model.load_state_dict(torch.load(path));
model.eval();


#inference
import spacy
nlp = spacy.load('en')


def predict(model, sentence):
    tokenized = [tok.text for tok in nlp.tokenizer(sentence)]  #tokenize the sentence
    indexed = [TEXT.vocab.stoi[t] for t in tokenized]          #convert to integer sequence
    length = [len(indexed)]                                    #compute no. of words
    tensor = torch.LongTensor(indexed).to(device)              #convert to tensor
    tensor = tensor.unsqueeze(1).T                             #reshape in form of batch,no. of words
    length_tensor = torch.LongTensor(length)                   #convert to tensor
    prediction = model(tensor, length_tensor)                  #prediction
    return prediction.item()
```

view rawinference.py hosted with by GitHub

Amazing! Let us use this model to make predictions for few questions:

```
#make predictions

predict(model, "Are there any sports that you don't like?")


#insincere question

predict(model, "Why Indian girls go crazy about marrying Shri. Rahul Gandhi ji?")
```

view rawtest.py hosted with by GitHub

End Notes

We have seen how to build our own text classification model in PyTorch and learnt the importance of pack padding. You can play around with the hyper-parameters of the Long Short Term Model such as number of hidden nodes, number of hidden layers and so on to improve the performance even further.

If you have any queries/feedback, leave in the comments section. I will get back to you.

Build an Image Classification Model using Convolutional Neural Networks in PyTorch

PULKIT SHARMA, OCTOBER 1, 2019

Overview

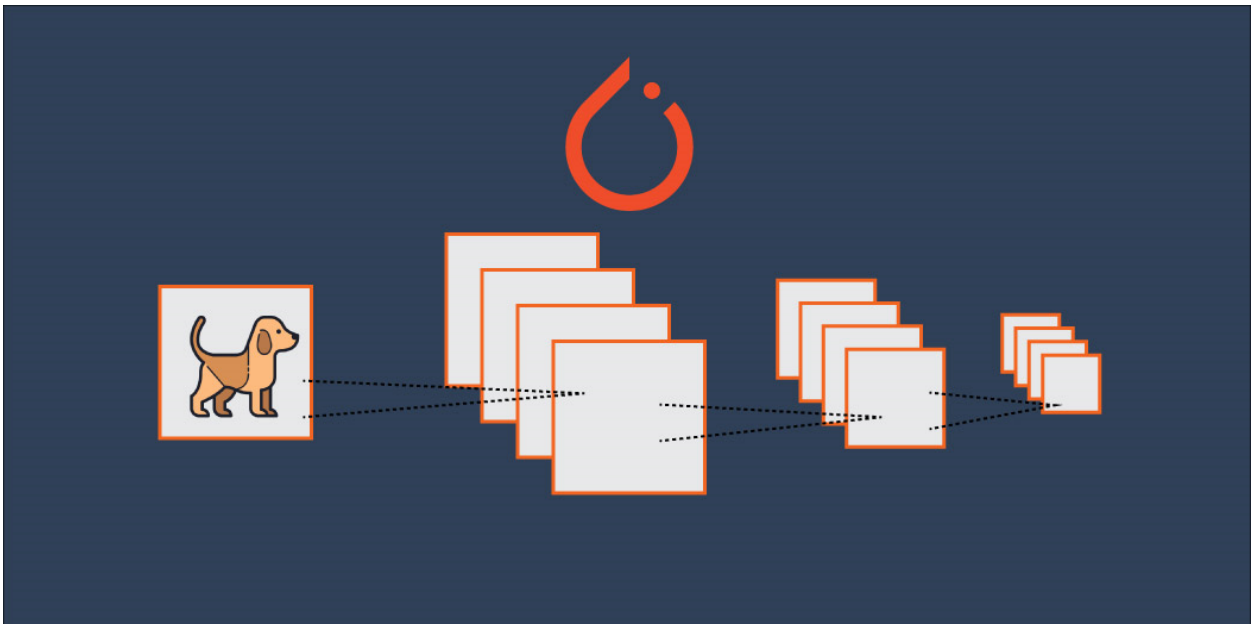
- A hands-on tutorial to build your own convolutional neural network (CNN) in PyTorch
- We will be working on an image classification problem – a classic and widely used application of CNNs

- This is part of Analytics Vidhya's series on PyTorch where we introduce deep learning concepts in a practical format

Introduction

I'm enthralled by the power and capability of neural networks. Almost every breakthrough happening in the machine learning and deep learning space right now has neural network models at its core.

This is especially prevalent in the field of computer vision. Neural networks have opened up possibilities of working with image data – whether that's simple image classification or something more advanced like object detection. In short, it's a goldmine for a data scientist like me!



Simple neural networks are always a good starting point when we're solving an image classification problem using deep learning. But they do have limitations and the model's performance fails to improve after a certain point.

This is where convolutional neural networks (CNNs) have changed the playing field. They are ubiquitous in computer vision applications. And it's honestly a concept I feel every computer vision enthusiast should pick up quickly.

This article is a continuation of my new series where I introduce you to new deep learning concepts using the popular PyTorch framework. In this article, we will understand how convolutional neural networks are helpful and how they can help us to improve our model's performance. We will also look at the implementation of CNNs in PyTorch.

If you want to comprehensively learn about CNNs, you can enrol in this free course: Convolutional Neural Networks from Scratch

This is the second article of this series and I highly recommend to go through the first part before moving forward with this article.

- *A Beginner-Friendly Guide to PyTorch and How it Works from Scratch*

Also, the third article of this series is live now where you can learn how to use pre-trained models and apply transfer learning using PyTorch:

- Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch

Table of Contents

1. A Brief Overview of PyTorch, Tensors and Numpy
2. Why Convolutional Neural Networks (CNNs)?
3. Understanding the Problem Statement: Identify the Apparels
4. Implementing CNNs using PyTorch

A Brief Overview of PyTorch, Tensors and NumPy

Let's quickly recap what we covered in the first article. We discussed the basics of PyTorch and tensors, and also looked at how PyTorch is similar to NumPy.

PyTorch is a Python-based library that provides functionalities such as:

- TorchScript for creating serializable and optimizable models
- Distributed training to parallelize computations
- Dynamic Computation graphs which enable to make the computation graphs on the go, and many more

Tensors in PyTorch are similar to NumPy's n-dimensional arrays which can also be used with GPUs. Performing operations on these tensors is almost similar to performing operations on NumPy arrays. **This makes PyTorch very user-friendly and easy to learn.**

In part 1 of this series, we built a simple neural network to solve a case study. We got a benchmark accuracy of around 65% on the test set using our simple model. Now, we will try to improve this score using Convolutional Neural Networks.

Why Convolutional Neural Networks (CNNs)?

Before we get to the implementation part, let's quickly look at why we need CNNs in the first place and how they are helpful.

We can consider Convolutional Neural Networks, or CNNs, as feature extractors that help to extract features from images.

In a simple neural network, we convert a 3-dimensional image to a single dimension, right? Let's look at an example to understand this:

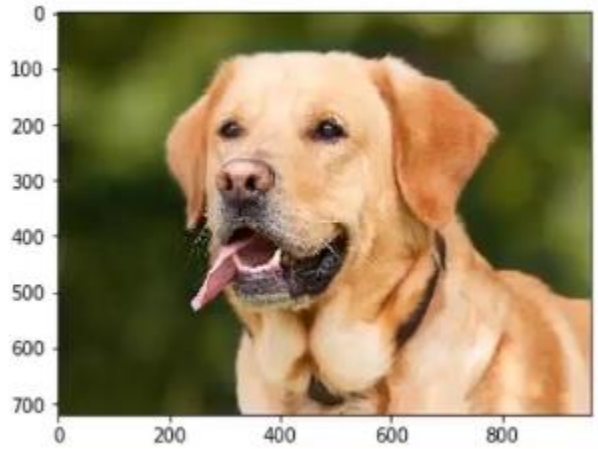
Can you identify the above image? Doesn't seem to make a lot of sense. Now, let's look at the below image:

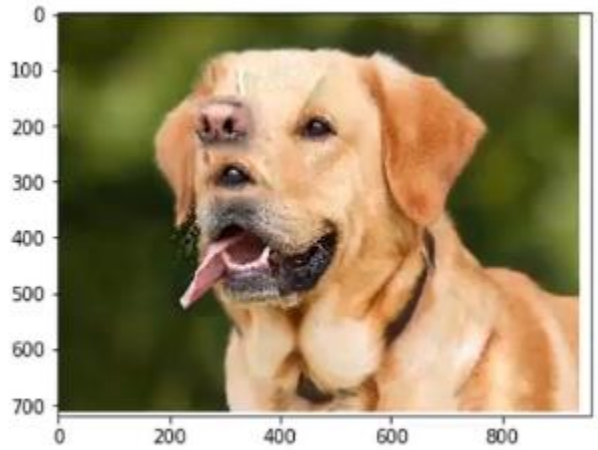
We can now easily say that it is an image of a dog. What if I tell you that both these images are the same? Believe me, they are! The only difference is that the first image is a 1-D representation whereas the second one is a 2-D representation of the same image.

Spatial Orientation

Artificial neural networks (ANNs) also lose the spatial orientation of the images. Let's again take an example and understand it:

Can you identify the difference between these two images? Well, at least I cannot. It is very difficult to identify the difference since this is a 1-D representation. Now, let's look at the 2-D representation of these images:





Don’t you love how different the same image looks by simply changing it’s representation? Here, the orientation of the images has been changed but we were unable to identify it by looking at the 1-D representation.

This is the problem with artificial neural networks – they lose spatial orientation.

Large number of parameters

Another problem with neural networks is the large number of parameters at play. Let’s say our image has a size of 28*28*3 – so the parameters here will be 2,352. What if we have an image of size 224*224*3? The number of parameters here will be 150,528.

And these parameters will only increase as we increase the number of hidden layers. So, the two major disadvantages of using artificial neural networks are:

1. Loses spatial orientation of the image
2. The number of parameters increases drastically

So how do we deal with this problem? How can we preserve the spatial orientation as well as reduce the learnable parameters?

This is where convolutional neural networks can be really helpful. **CNNs help to extract features from the images which may be helpful in classifying the objects in that image.** It starts by extracting low dimensional features (like edges) from the image, and then some high dimensional features like the shapes.

We use filters to extract features from the images and Pooling techniques to reduce the number of learnable parameters.

We will not be diving into the details of these topics in this article. If you wish to understand how filters help to extract features and how pooling works, I highly recommend you go through A Comprehensive Tutorial to learn Convolutional Neural Networks from Scratch.

Understanding the Problem Statement: Identify the Apparels

Enough theory – let's get coding! We'll be taking up the same problem statement we covered in the first article. This is because we can directly compare our CNN model's performance to the simple neural network we built there.

You can download the dataset for this 'Identify' the Apparels' problem from here.

Let me quickly summarize the problem statement. Our task is to identify the type of apparel by looking at a variety of apparel images. There are a total of 10 classes in which we can classify the images of apparels:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal

6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

The dataset contains a total of 70,000 images. 60,000 of these images belong to the training set and the remaining 10,000 are in the test set. All the images are grayscale images of size (28*28). The dataset contains two folders – one each for the training set and the test set. In each folder, there is a .csv file that has the *id* of the image and its corresponding label, and a folder containing the images for that particular set.

Ready to begin? We will start by importing the required libraries:

```
# importing the libraries
import pandas as pd
import numpy as np


# for reading and displaying images
from skimage.io import imread
import matplotlib.pyplot as plt
%matplotlib inline


# for creating validation set
from sklearn.model_selection import train_test_split


# for evaluating the model
from sklearn.metrics import accuracy_score
from tqdm import tqdm
```

```
# PyTorch libraries and modules

import torch

from torch.autograd import Variable

from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d,
Module, Softmax, BatchNorm2d, Dropout

from torch.optim import Adam, SGD
```

view rawlibrary.py hosted with by GitHub

Loading the dataset

Now, let's load the dataset, including the train, test and sample submission file:

```
# loading dataset

train = pd.read_csv('train_LbELtWX/train.csv')

test = pd.read_csv('test_ScVgIM0/test.csv')


sample_submission = pd.read_csv('sample_submission_I5njJSF.csv')


train.head()
```

view rawdataset.py hosted with by GitHub

	id	label
0	1	9
1	2	0
2	3	0
3	4	3
4	5	0

- The train file contains the id of each image and its corresponding label
- **The test file, on the other hand, only has the ids and we have to predict their corresponding labels**
- The sample submission file will tell us the format in which we have to submit the predictions

We will read all the images one by one and stack them one over the other in an array. We will also divide the pixels of images by 255 so that the pixel values of images comes in the range [0,1]. This step helps in optimizing the performance of our model.

So, let's go ahead and load the images:

```
# loading training images
train_img = []
for img_name in tqdm(train['id']):
    # defining the image path
    image_path = 'train_LbELtWX/train/' + str(img_name) + '.png'
    # reading the image
    img = imread(image_path, as_gray=True)
    # normalizing the pixel values
    img /= 255.0
    # converting the type of pixel to float 32
    img = img.astype('float32')
    # appending the image into the list
    train_img.append(img)

# converting the list to numpy array
train_x = np.array(train_img)
# defining the target
train_y = train['label'].values
train_x.shape
```

view rawtrain_data.py hosted with by GitHub

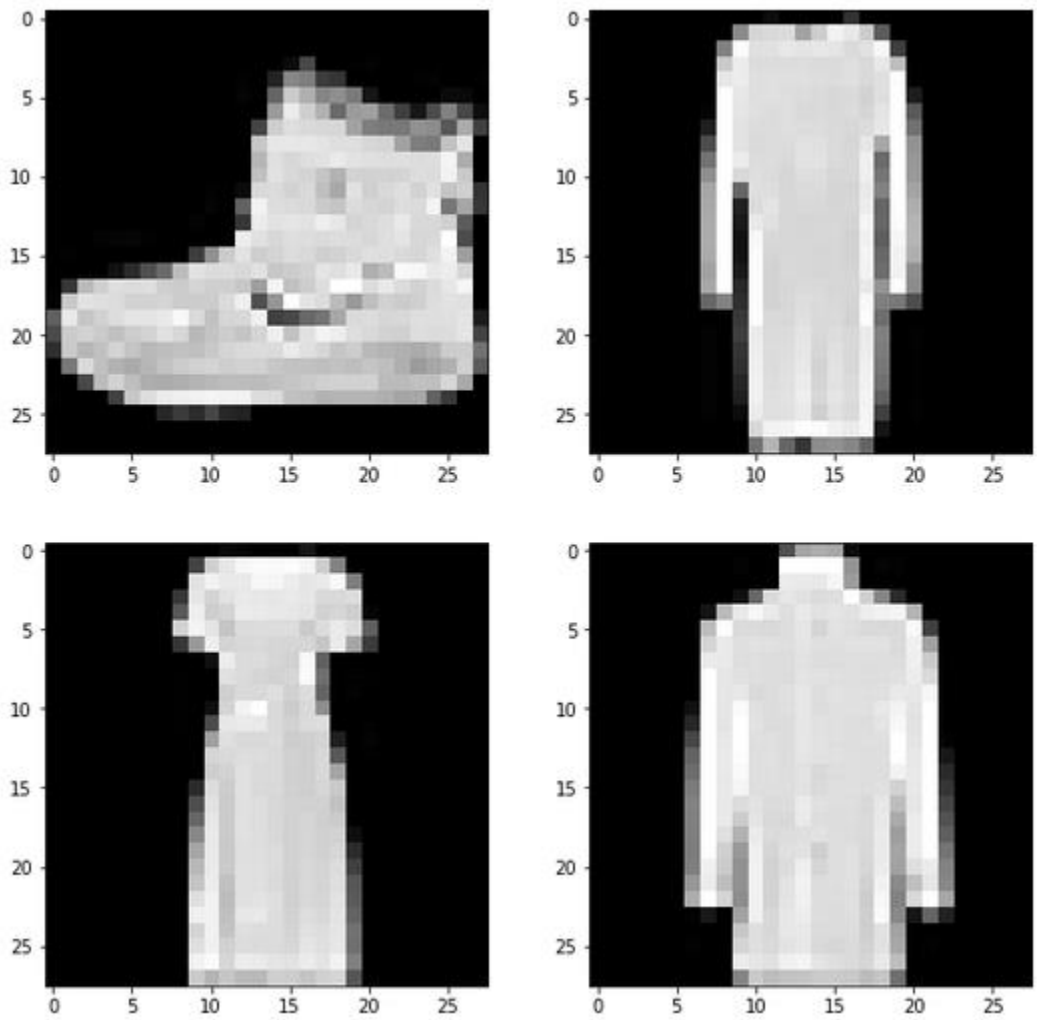
```
(60000, 28, 28)
```

As you can see, we have 60,000 images, each of size (28,28), in the training set. Since the images are in grayscale format, we only have a single-channel and hence the shape (28,28).

Let’s now explore the data and visualize a few images:

```
# visualizing images

i = 0

plt.figure(figsize=(10,10))

plt.subplot(221), plt.imshow(train_x[i], cmap='gray')

plt.subplot(222), plt.imshow(train_x[i+25], cmap='gray')

plt.subplot(223), plt.imshow(train_x[i+50], cmap='gray')

plt.subplot(224), plt.imshow(train_x[i+75], cmap='gray')
```

view rawvisualize.py hosted with by GitHub



These are a few examples from the dataset. I encourage you to explore more and visualize other images. Next, we will divide our images into a training and validation set.

Creating a validation set and preprocessing the images

```
# create validation set
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size = 0.1)
(train_x.shape, train_y.shape), (val_x.shape, val_y.shape)
```

view rawvalidation_data.py hosted with by GitHub

```
(((54000, 28, 28), (54000,)), ((6000, 28, 28), (6000,)))
```

We have kept 10% data in the validation set and the remaining in the training set. Next, let's convert the images and the targets into torch format:

```
# converting training images into torch format
train_x = train_x.reshape(54000, 1, 28, 28)
train_x  = torch.from_numpy(train_x)

# converting the target into torch format
train_y = train_y.astype(int);
train_y = torch.from_numpy(train_y)

# shape of training data
train_x.shape, train_y.shape
```

view rawtrain_torch_format.py hosted with by GitHub

```
(torch.Size([54000, 1, 28, 28]), torch.Size([54000]))
```

Similarly, we will convert the validation images:

```
# converting validation images into torch format

val_x = val_x.reshape(6000, 1, 28, 28)

val_x  = torch.from_numpy(val_x)


# converting the target into torch format

val_y = val_y.astype(int);

val_y = torch.from_numpy(val_y)


# shape of validation data

val_x.shape, val_y.shape
```

view rawvalidation_torch_format.py hosted with by GitHub

```
(torch.Size([6000, 1, 28, 28]), torch.Size([6000]))
```

Our data is now ready. Finally, it's time to create our CNN model!

Implementing CNNs using PyTorch

We will use a very simple CNN architecture with just 2 convolutional layers to extract features from the images. We'll then use a fully connected dense layer to classify those features into their respective categories.

Let's define the architecture:

```
class Net(Module):

    def __init__(self):

        super(Net, self).__init__()


        self.cnn_layers = Sequential(

            # Defining a 2D convolution layer

            Conv2d(1, 4, kernel_size=3, stride=1, padding=1),
```

```
            BatchNorm2d(4),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
            # Defining another 2D convolution layer
            Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(4),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
        )

        self.linear_layers = Sequential(
            Linear(4 * 7 * 7, 10)
        )

    # Defining the forward pass
    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x
```

view rawcnn_architecture.py hosted with by GitHub

Let's now call this model, and define the optimizer and the loss function for the model:

```
# defining the model
model = Net()
# defining the optimizer
optimizer = Adam(model.parameters(), lr=0.07)
# defining the loss function
```

```
criterion = CrossEntropyLoss()

# checking if GPU is available

if torch.cuda.is_available():

    model = model.cuda()

    criterion = criterion.cuda()



print(model)
```

view rawmodel.py hosted with by GitHub

```
Net(
  (cnn_layers): Sequential(
    (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(4, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear_layers): Sequential(
    (0): Linear(in_features=196, out_features=10, bias=True)
  )
)
```

This is the architecture of the model. We have two Conv2d layers and a Linear layer. Next, we will define a function to train the model:

```
def train(epoch):

    model.train()

    tr_loss = 0

    # getting the training set

    x_train, y_train = Variable(train_x), Variable(train_y)

    # getting the validation set

    x_val, y_val = Variable(val_x), Variable(val_y)

    # converting the data into GPU format

    if torch.cuda.is_available():
```

```
    x_train = x_train.cuda()
    y_train = y_train.cuda()
    x_val = x_val.cuda()
    y_val = y_val.cuda()

# clearing the Gradients of the model parameters
optimizer.zero_grad()

# prediction for training and validation set
output_train = model(x_train)
output_val = model(x_val)

# computing the training and validation loss
loss_train = criterion(output_train, y_train)
loss_val = criterion(output_val, y_val)
train_losses.append(loss_train)
val_losses.append(loss_val)

# computing the updated weights of all the model parameters
loss_train.backward()
optimizer.step()
tr_loss = loss_train.item()
if epoch%2 == 0:
    # printing the validation loss
    print('Epoch : ',epoch+1, '\t', 'loss :', loss_val)
```

view rawtrain_function.py hosted with by GitHub

Finally, we will train the model for 25 epochs and store the training and validation losses:

```
# defining the number of epochs

n_epochs = 25

# empty list to store training losses

train_losses = []

# empty list to store validation losses

val_losses = []

# training the model

for epoch in range(n_epochs):

    train(epoch)
```

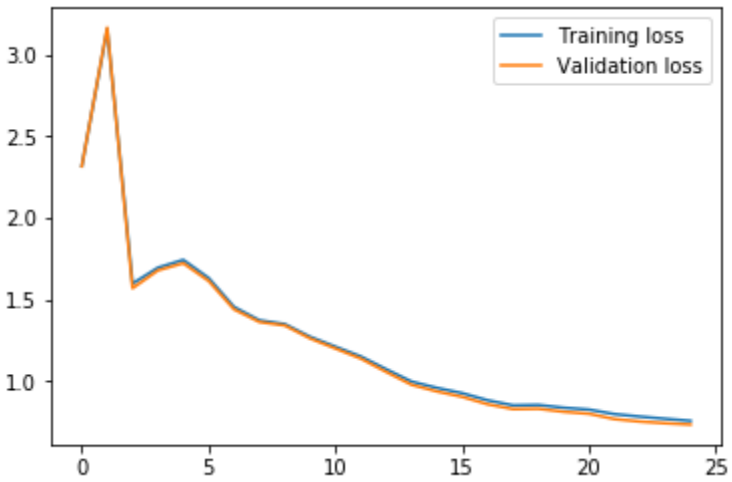
view rawtrain.py hosted with by GitHub

```
Epoch :  1      loss : tensor(2.3163, device='cuda:0')
Epoch :  3      loss : tensor(1.5701, device='cuda:0')
Epoch :  5      loss : tensor(1.7231, device='cuda:0')
Epoch :  7      loss : tensor(1.4408, device='cuda:0')
Epoch :  9      loss : tensor(1.3423, device='cuda:0')
Epoch :  11     loss : tensor(1.2012, device='cuda:0')
Epoch :  13     loss : tensor(1.0581, device='cuda:0')
Epoch :  15     loss : tensor(0.9387, device='cuda:0')
Epoch :  17     loss : tensor(0.8601, device='cuda:0')
Epoch :  19     loss : tensor(0.8327, device='cuda:0')
Epoch :  21     loss : tensor(0.8020, device='cuda:0')
Epoch :  23     loss : tensor(0.7543, device='cuda:0')
Epoch :  25     loss : tensor(0.7362, device='cuda:0')
```

We can see that the validation loss is decreasing as the epochs are increasing. Let's visualize the training and validation losses by plotting them:

```
# plotting the training and validation loss

plt.plot(train_losses, label='Training loss')

plt.plot(val_losses, label='Validation loss')

plt.legend()

plt.show()
```

view rawloss_visualization.py hosted with by GitHub



Ah, I love the power of visualization. We can clearly see that the training and validation losses are in sync. It is a good sign as the model is generalizing well on the validation set.

Let's check the accuracy of the model on the training and validation set:

```
# prediction for training set
with torch.no_grad():
    output = model(train_x.cuda())

softmax = torch.exp(output).cpu()
prob = list(softmax.numpy())
predictions = np.argmax(prob, axis=1)

# accuracy on training set
accuracy_score(train_y, predictions)
```

view rawtrain_accuracy.py hosted with by GitHub

```
0.7231851851851852
```

An accuracy of ~72% accuracy on the training set is pretty good. Let's check the accuracy for the validation set as well:

```
# prediction for validation set
```

```
with torch.no_grad():

    output = model(val_x.cuda())


softmax = torch.exp(output).cpu()

prob = list(softmax.numpy())

predictions = np.argmax(prob, axis=1)


# accuracy on validation set

accuracy_score(val_y, predictions)
```

view rawvalidation_accuracy.py hosted with by GitHub

```
0.727
```

As we saw with the losses, the accuracy is also in sync here – we got ~72% on the validation set as well.

Generating predictions for the test set

It's finally time to generate predictions for the test set. We will load all the images in the test set, do the same pre-processing steps as we did for the training set and finally generate predictions.

So, let's start by loading the test images:

```
# loading test images

test_img = []

for img_name in tqdm(test['id']):

    # defining the image path

    image_path = 'test_ScVgIM0/test/' + str(img_name) + '.png'

    # reading the image

    img = imread(image_path, as_gray=True)
```

```
        # normalizing the pixel values
        img /= 255.0
        # converting the type of pixel to float 32
        img = img.astype('float32')
        # appending the image into the list
        test_img.append(img)


    # converting the list to numpy array
    test_x = np.array(test_img)
    test_x.shape
```

view rawtest_data.py hosted with by GitHub

```
(10000, 28, 28)
```

Now, we will do the pre-processing steps on these images similar to what we did for the training images earlier:

```
    # converting training images into torch format
    test_x = test_x.reshape(10000, 1, 28, 28)
    test_x  = torch.from_numpy(test_x)
    test_x.shape
```

view rawtest_torch_format.py hosted with by GitHub

```
torch.Size([10000, 1, 28, 28])
```

Finally, we will generate predictions for the test set:

```
    # generating predictions for test set
    with torch.no_grad():
        output = model(test_x.cuda())
```

```
softmax = torch.exp(output).cpu()

prob = list(softmax.numpy())

predictions = np.argmax(prob, axis=1)
```

view rawtest_prediction.py hosted with by GitHub

Replace the labels in the sample submission file with the predictions and finally save the file and submit it on the leaderboard:

```
# replacing the label with prediction

sample_submission['label'] = predictions

sample_submission.head()
```

view rawsubmission.py hosted with by GitHub

	id	label
0	60001	9
1	60002	2
2	60003	1
3	60004	1
4	60005	2

```
# saving the file

sample_submission.to_csv('submission.csv', index=False)
```

view rawsubmission_file.py hosted with by GitHub

You will see a file named *submission.csv* in your current directory. You just have to upload it on the solution checker of the problem page which will generate the score.

Our CNN model gave us an accuracy of around 71% on the test set. That is quite an improvement on the 65% we got using a simple neural network in our previous article.

End Notes

In this article, we looked at how CNNs can be useful for extracting features from images. They helped us to improve the accuracy of our previous neural network model from 65% to 71% – a significant upgrade.

You can play around with the hyperparameters of the CNN model and try to improve accuracy even further. Some of the hyperparameters to tune can be the number of convolutional layers, number of filters in each convolutional layer, number of epochs, number of dense layers, number of hidden units in each dense layer, etc.

In the next article of this series, we will learn how to use pre-trained models like VGG-16 and model checkpointing steps in PyTorch. And as always, if you have any doubts related to this article, feel free to post them in the comments section below!

Image Augmentation for Deep Learning using PyTorch – Feature Engineering for Images

PULKIT SHARMA, DECEMBER 5, 2019

Overview

- Image augmentation is a super effective concept when we don't have enough data with us
- We can use image augmentation for deep learning in any setting – hackathons, industry projects, and so on
- We'll also build an image classification model using PyTorch to understand how image augmentation fits into the picture

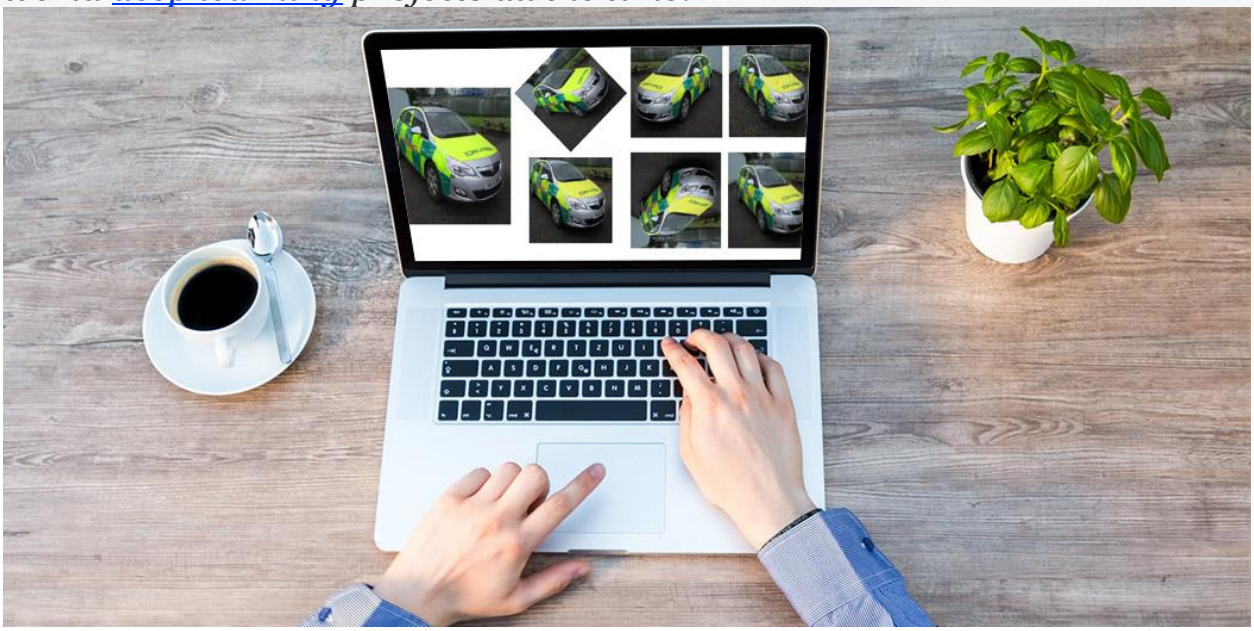
Introduction

The trick to do well in deep learning hackathons (or frankly any data science hackathon) often comes down to feature engineering. How much creativity can you muster when you're given data that simply isn't enough to build a winning deep learning model?

I'm talking from my own experience of participating in multiple deep learning hackathons where we were given a dataset of a few hundred images – simply not enough to win or even finish in the top echelons of the leaderboard. So how can we deal with this problem?

The answer? Well, that lies deep in a data scientist's skillset! This is where our curiosity and creativity come to the fore. That's the idea behind feature engineering – how well we can come up with new features given existing ones. And the same idea applies when we're working with image data.

And that's where image augmentation plays a major role. This concept isn't limited to hackathons – we use it in the industry and in real-world deep learning projects all the time!



I love how image augmentation helps spruce up my existing dataset with more data without having to put manual time taking efforts. And I'm sure you're going to find this technique very helpful for your own projects.

So in this article, we will understand the concept of image augmentation, why it's helpful, and what are the different image augmentation techniques. We'll also implement these image augmentation techniques to build an image classification model using PyTorch.

This is the fifth article of PyTorch for beginners series which I have been working on. You can access the previous articles here:

- *A Beginner-Friendly Guide to PyTorch and How it Works from Scratch*

- *Build an Image Classification Model using Convolutional Neural Networks in PyTorch*
- *Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch*
- *4 Proven Tricks to Improve your Deep Learning Model's Performance*

Table of Contents

1. Why Do We Need Image Augmentation?
2. Different Image Augmentation Techniques
3. Basic Guidelines for Selecting the Right Augmentation Technique
4. Case Study: Solving an Image Classification Problem using Image Augmentation

Why Do We Need Image Augmentation?

Deep learning models usually require a lot of data for training. In general, the more the data, the better the performance of the model. But acquiring massive amounts of data comes with its own challenges. Not everyone has the deep pockets of the big firms.

And the problem with a lack of data is that our deep learning model might not learn the pattern or function from the data and hence it might not give a good performance on unseen data.

So what can we do in that case? Instead of spending days manually collecting data, we can make use of **Image Augmentation techniques**.

> *Image Augmentation is the process of generating new images for training our deep learning model. These new images are generated using the existing training images and hence we don't have to collect them manually.*

There are multiple image augmentation techniques and we will discuss some of the common and most widely used ones in the next section.

Different Image Augmentation Techniques

Image Rotation

Image rotation is one of the most commonly used augmentation techniques. It can help our model become robust to the changes in the orientation of objects. Even if we rotate the image, the information of the image remains the same. A car is a car even if we see it from a different angle:

Hence, we can use this technique to increase the size of our data by creating rotated images from the original ones. Let's see how we can rotate images:

```
# importing all the required libraries

import warnings

warnings.filterwarnings('ignore')

import numpy as np

import skimage.io as io

from skimage.transform import rotate, AffineTransform, warp

from skimage.util import random_noise

from skimage.filters import gaussian

import matplotlib.pyplot as plt

% matplotlib inline
```

view rawlibrary.py hosted with by GitHub

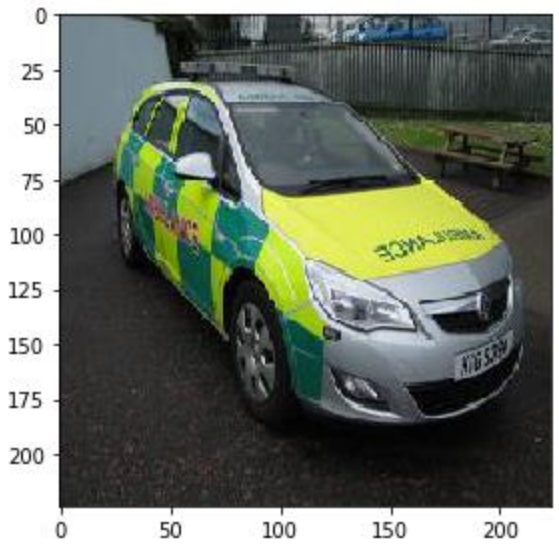
I will be using this image to demonstrate different image augmentation techniques. You can try other images as well as per your requirement.

Let's import the image and visualize it first:

```
# reading the image using its path

image = io.imread('emergency_vs_non-emergency_dataset/images/0.jpg')


# shape of the image

print(image.shape)


# displaying the image

io.imshow(image)
```

view rawreading_image.py hosted with by GitHub

```
(224, 224, 3)
<matplotlib.image.AxesImage at 0x7f5ceb6d4240>
```



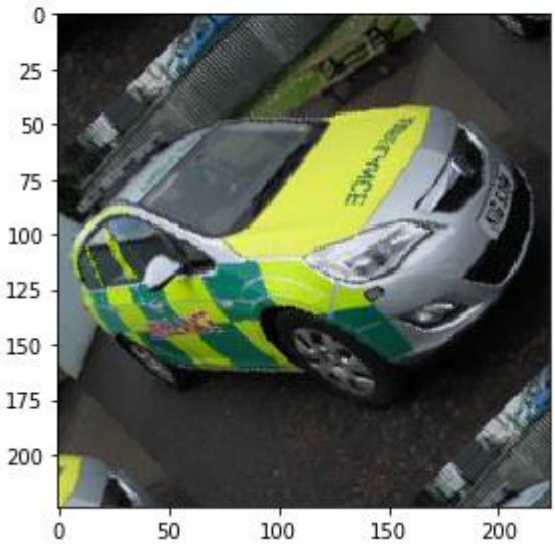
This is the original image. Let's now see how we can rotate it. I will use the rotate function of the *skimage* library to rotate the image:

```
print('Rotated Image')

#rotating the image by 45 degrees

rotated = rotate(image, angle=45, mode = 'wrap')

#plot the rotated image
```

```
io.imshow(rotated)
```

view rawimage_rotation.py hosted with by GitHub

```
Rotated Image

<matplotlib.image.AxesImage at 0x7f5ce9dfa470>
```



Nice! Setting mode as ‘wrap’ fills the points outside the boundaries of the input with the remaining pixels of the image.

Shifting Images

There might be scenarios when the objects in the image are not perfectly central aligned. In these cases, image shift can be used to add shift-invariance to the images.

By shifting the images, we can change the position of the object in the image and hence give more variety to the model. This will eventually lead to a more generalized model.

Image shift is a geometric transformation that maps the position of every object in the image to a new location in the final output image.

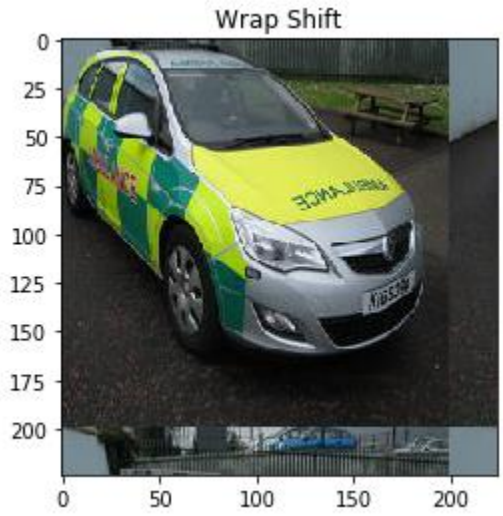
After the shift operation, an object present at a location (x,y) in the input image is shifted to a new position (X, Y):

- X = x + dx
- Y = y + dy

Here, dx and dy are the respective shifts along different dimensions. Let's see how we can apply shift to an image:

```
#apply shift operation

transform = AffineTransform(translation=(25,25))

wrapShift = warp(image,transform,mode='wrap')

plt.imshow(wrapShift)

plt.title('Wrap Shift')
```

view rawimage_shift.py hosted with by GitHub



The translation hyperparameter defines the number of pixels by which the image should be shifted. Here, I have shifted the image by (25, 25) pixels. You can play around with the values of this hyperparameter.

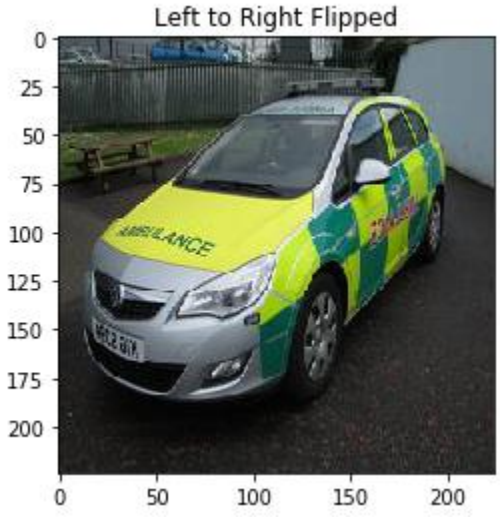
Again I have used the mode as 'wrap' which fills the points outside the boundaries of the input with the remaining pixels of the image. In the output above, you can see that both the height and width of the image have been shifted by 25 pixels.

Flipping Images

Flipping is an extension of rotation. It allows us to flip the image in the left-right as well as up-down direction. Let's see how we can implement flipping:

```
#flip image left-to-right

flipLR = np.fliplr(image)


plt.imshow(flipLR)

plt.title('Left to Right Flipped')
```

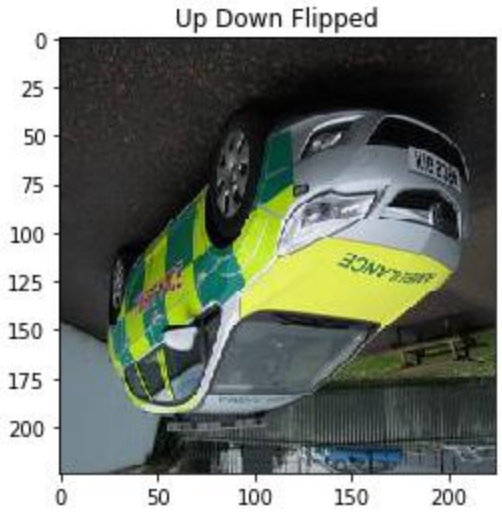
view rawflip_1.py hosted with by GitHub



Here, I have used the *fliplr* function of NumPy to flip the image from left to right. It flips the pixel values of each row and the output confirms the same. Similarly, we can flip the images in an up-down direction:

```
#flip image up-to-down

flipUD = np.flipud(image)


plt.imshow(flipUD)

plt.title('Up Down Flipped')
```

view rawflip_2.py hosted with by GitHub



This is how we can flip the image and make more generalized models that will learn the patterns of the original as well as the flipped images. Adding random noise to the images is also an image augmentation technique. Let's understand it using an example.

Adding Noise to Images

Image noising is an important augmentation step that allows our model to learn how to separate signal from noise in an image. This also makes the model more robust to changes in the input.

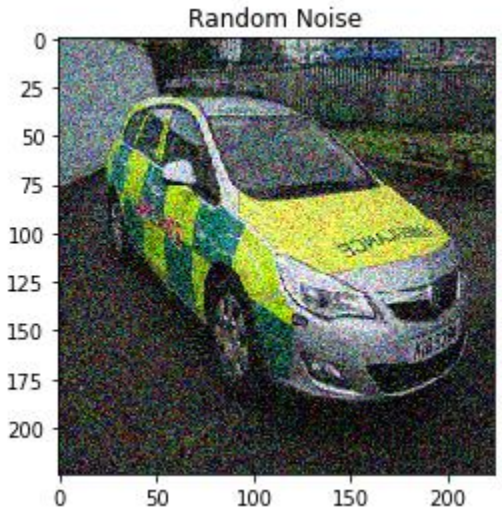
We will use the *random_noise* function of the *skimage* library to add some random noise to our original image.

I will take the standard deviation of the noise to be added as 0.155 (you can change this value as well). Just keep in mind that increasing this value will add more noise to the image and vice versa:

```
#standard deviation for noise to be added in the image

sigma=0.155

#add random noise to the image

noisyRandom = random_noise(image,var=sigma**2)


plt.imshow(noisyRandom)
```

```
plt.title('Random Noise')
```

view rawadding_noise.py hosted with by GitHub



We can see that random noise has been added to our original image. Play around with the standard deviation value and see the different results you get.

Blurring Images

All photography lovers will instantly understand this idea.

Images come from different sources. And hence, the quality of the images will not be the same from each source. Some images might be of very high quality while others might be just plain bad.

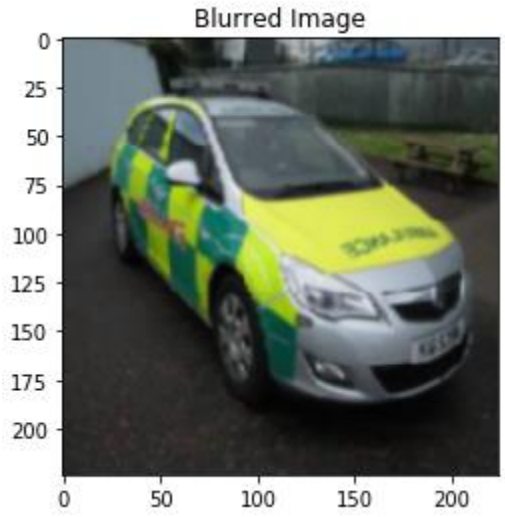
In such scenarios, we can blur the image. How will that help? Well, this helps make our deep learning model more robust.

Let's see how we can do that. We will use a Gaussian filter for blurring the image:

```
#blur the image

blurred = gaussian(image,sigma=1,multichannel=True)


plt.imshow(blurred)
```

```
    plt.title('Blurred Image')
```

view rawblurring.py hosted with by GitHub



Sigma here is the standard deviation for the Gaussian filter. I have taken it as 1. The higher the sigma value, the more will be the blurring effect. **Setting *Multichannel* to true ensures that each channel of the image is filtered separately.**

Again, you can try different sigma values to change the magnitude of blurriness.

These are some of the image augmentation techniques which help to make our deep learning model robust and generalizable. This also helps increase the size of the training set.

We're almost at the implementation part of this tutorial. Before that, let's look at some of the basic guidelines for deciding the right image augmentation technique.

Basic Guidelines for Selecting the Right Augmentation Technique

There are a few guidelines that I think are important while deciding the augmentation technique based on the problem that you are trying to solve. Here is a brief summary of these guidelines:

1. The first step in any model building process is to make sure that the size of our input matches what is expected by the model. We also have to make sure that the size of

all the images should be similar. For this, we can resize our images to the appropriate size.
2. Let's say you are working on a classification problem and have relatively less number of data samples. In such scenarios, you can use different augmentation techniques like image rotation, image noising, flipping, shift, etc. Remember all these operations are applicable for classification problems where the location of objects in the image does not matter.
3. If you are working on an object detection task, where the location of objects is what we want to detect, these techniques might not be appropriate.
4. Normalizing image pixel values is always a good strategy to ensure better and faster convergence of the model. If there are some specific requirements of the model, we must pre-process the images as per the model's requirement.

Now, without waiting further, let's move on to the model building part. We will apply the augmentation techniques that are discussed in this article to generate images and then use those images to train the model.

Case Study: Solving an Image Classification Problem and Applying Image Augmentation

We will be working on the emergency vs non-emergency vehicle classification problem. You should be familiar with the problem statement if you've gone through my previous PyTorch articles.

The aim of this project is to classify the images of vehicles as emergency or non-emergency. And you guessed it – it's an image classification problem. You can **download the dataset from here**.

Loading the dataset

Let's begin! We'll start by loading the data into our notebook. Then, we'll apply image augmentation techniques and finally, build a convolutional neural network (CNN) model.

Let's import the required libraries:

```
# importing the libraries

from torchsummary import summary

import pandas as pd
```

```
import numpy as np
from skimage.io import imread, imsave
from tqdm import tqdm
import matplotlib.pyplot as plt
%matplotlib inline


from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


from skimage.transform import rotate
from skimage.util import random_noise
from skimage.filters import gaussian
from scipy import ndimage
```

view rawimporting_library.py hosted with by GitHub

Now, we will read the CSV file that contains the names of images and their corresponding labels:

```
# loading dataset
data = pd.read_csv('emergency_vs_non-emergency_dataset/emergency_train.csv')
data.head()
```

view rawloading_dataset.py hosted with by GitHub

	image_names	emergency_or_not
0	1503.jpg	0
1	1420.jpg	0
2	1764.jpg	0
3	1356.jpg	0
4	1117.jpg	0

0 here represents that the vehicle is non-emergency and 1 means it's an emergency vehicle. Let's now import all the images from our dataset:

```
# loading images

train_img = []

for img_name in tqdm(data['image_names']):

    image_path = 'emergency_vs_non-emergency_dataset/images/' + img_name

    img = imread(image_path)

    img = img/255

    train_img.append(img)



train_x = np.array(train_img)

train_y = data['emergency_or_not'].values

train_x.shape, train_y.shape
```

view rawloading_images.py hosted with by GitHub

```
100%|██████████| 1646/1646 [00:03<00:00, 528.57it/s]
((1646, 224, 224, 3), (1646,))
```

We have a total of 1,646 images in the dataset. Let's split this data into training and validation set. We will use the validation set to evaluate how well the model will perform on unseen data:

```
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size = 0.1,
random_state = 13, stratify=train_y)

(train_x.shape, train_y.shape), (val_x.shape, val_y.shape)
```

view rawvalidation.py hosted with by GitHub

```
(((1481, 224, 224, 3), (1481,)), ((165, 224, 224, 3), (165,)))
```

I have kept the *test_size* as 0.1 and hence 10% data will be randomly selected as the validation set and the remaining 90% will be used to train the model. We have 1,481 images in the training set which is quite less to train a deep learning model.

So next, we will augment these training images to increase the training set and possibly improve our model's performance.

Augmenting the Images

We will be using the image augmentation techniques we discussed earlier:

```
final_train_data = []

final_target_train = []

for i in tqdm(range(train_x.shape[0])):

    final_train_data.append(train_x[i])

    final_train_data.append(rotate(train_x[i], angle=45, mode = 'wrap'))

    final_train_data.append(np.fliplr(train_x[i]))

    final_train_data.append(np.flipud(train_x[i]))

    final_train_data.append(random_noise(train_x[i],var=0.2**2))

    for j in range(5):

        final_target_train.append(train_y[i])
```

view rawaugmentation.py hosted with by GitHub

```
100%|██████████| 1481/1481 [00:20<00:00, 73.39it/s]
```

We have generated 4 augmented images for each of the 1,481 images in the training set. Let's convert the images in the form of an array and verify the size of our dataset:

```
len(final_target_train), len(final_train_data)

final_train = np.array(final_train_data)

final_target_train = np.array(final_target_train)
```

view rawaugmented_data.py hosted with by GitHub

```
(7405, 7405)
```

This confirms that we have augmented the images and increased the size of our training set. Let's visualize these augmented images:

```
fig,ax = plt.subplots(nrows=1,ncols=5,figsize=(20,20))
for i in range(5):
    ax[i].imshow(final_train[i+30])
    ax[i].axis('off')
```

view rawaugmentation_visualization.py hosted with by GitHub

The first image here is the original image from the dataset. The remaining four images are generated using different image augmentation techniques – rotation, left-to-right flip, up-down flip and adding random noise respectively.

Our dataset is now ready. It's time to define the architecture of our deep learning model and then train it on the augmented training set. Let's first import all the functions from PyTorch:

```
# PyTorch libraries and modules
import torch
from torch.autograd import Variable
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d,
Module, Softmax, BatchNorm2d, Dropout
from torch.optim import Adam, SGD
```

view rawpytorch_library.py hosted with by GitHub

We'll have to convert both the training and validation sets into PyTorch format:

```
# converting training images into torch format
final_train = final_train.reshape(7405, 3, 224, 224)
final_train  = torch.from_numpy(final_train)
final_train = final_train.float()
```

```
# converting the target into torch format
final_target_train = final_target_train.astype(int)
final_target_train = torch.from_numpy(final_target_train)
```

view rawtraining_set.py hosted with by GitHub

Similarly, we will convert the validation set:

```
# converting validation images into torch format
val_x = val_x.reshape(165, 3, 224, 224)
val_x  = torch.from_numpy(val_x)
val_x = val_x.float()

# converting the target into torch format
val_y = val_y.astype(int)
val_y = torch.from_numpy(val_y)
```

view rawvalidation_set.py hosted with by GitHub

Model Architecture

Next, we will define the architecture of the model. This is a bit complex since the architecture has 4 convolutional blocks and then 4 fully connected dense layers:

```
torch.manual_seed(0)


class Net(Module):
    def __init__(self):
        super(Net, self).__init__()
```

```
self.cnn_layers = Sequential(
    # Defining a 2D convolution layer
    Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
    ReLU(inplace=True),
    # adding batch normalization
    BatchNorm2d(32),
    MaxPool2d(kernel_size=2, stride=2),
    # adding dropout
    Dropout(p=0.25),
    # Defining another 2D convolution layer
    Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
    ReLU(inplace=True),
    # adding batch normalization
    BatchNorm2d(64),
    MaxPool2d(kernel_size=2, stride=2),
    # adding dropout
    Dropout(p=0.25),
    # Defining another 2D convolution layer
    Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    ReLU(inplace=True),
    # adding batch normalization
    BatchNorm2d(128),
    MaxPool2d(kernel_size=2, stride=2),
    # adding dropout
    Dropout(p=0.25),
    # Defining another 2D convolution layer
    Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
    ReLU(inplace=True),
```

```
            # adding batch normalization
            BatchNorm2d(128),
            MaxPool2d(kernel_size=2, stride=2),
            # adding dropout
            Dropout(p=0.25),
        )

        self.linear_layers = Sequential(
            Linear(128 * 14 * 14, 512),
            ReLU(inplace=True),
            Dropout(),
            Linear(512, 256),
            ReLU(inplace=True),
            Dropout(),
            Linear(256,10),
            ReLU(inplace=True),
            Dropout(),
            Linear(10,2)
        )

    # Defining the forward pass
    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x
```

Let’s define the other hyperparameters of the model as well, including the optimizer, learning rate, and the loss function:

```
# defining the model

model = Net()

# defining the optimizer

optimizer = Adam(model.parameters(), lr=0.000075)

# defining the loss function

criterion = CrossEntropyLoss()

# checking if GPU is available

if torch.cuda.is_available():

    model = model.cuda()

    criterion = criterion.cuda()


print(model)
```

```
Net(
  (cnn_layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
    (5): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Dropout(p=0.25, inplace=False)
    (10): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (14): Dropout(p=0.25, inplace=False)
    (15): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU(inplace=True)
    (17): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Dropout(p=0.25, inplace=False)
  )
  (linear_layers): Sequential(
    (0): Linear(in_features=25088, out_features=512, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=512, out_features=256, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=256, out_features=10, bias=True)
    (7): ReLU(inplace=True)
    (8): Dropout(p=0.5, inplace=False)
    (9): Linear(in_features=10, out_features=2, bias=True)
  )
)
```

Training the Model

Let’s train our deep learning model for 20 epochs:

```
torch.manual_seed(0)


# batch size of the model

batch_size = 64


# number of epochs to train the model

n_epochs = 20
```

```
for epoch in range(1, n_epochs+1):


    train_loss = 0.0


    permutation = torch.randperm(final_train.size()[0])


    training_loss = []
    for i in tqdm(range(0,final_train.size()[0], batch_size)):


        indices = permutation[i:i+batch_size]
        batch_x, batch_y = final_train[indices], final_target_train[indices]


        if torch.cuda.is_available():
            batch_x, batch_y = batch_x.cuda(), batch_y.cuda()


        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs,batch_y)


        training_loss.append(loss.item())
        loss.backward()
        optimizer.step()


    training_loss = np.average(training_loss)
    print('epoch: \t', epoch, '\t training loss: \t', training_loss)
```

```
100%|██████████| 116/116 [00:11<00:00, 10.36it/s]
  2%|          | 2/116 [00:00<00:10, 11.38it/s]epoch:    4       training loss:          0.6411901748385923
100%|██████████| 116/116 [00:11<00:00, 10.36it/s]
  2%|          | 2/116 [00:00<00:09, 11.53it/s]epoch:    5       training loss:          0.6165799809427097
100%|██████████| 116/116 [00:11<00:00, 10.35it/s]
  2%|          | 2/116 [00:00<00:09, 11.47it/s]epoch:    6       training loss:          0.5999701357093351
100%|██████████| 116/116 [00:11<00:00, 10.33it/s]
  2%|          | 2/116 [00:00<00:09, 11.42it/s]epoch:    7       training loss:          0.5808765243353515
100%|██████████| 116/116 [00:11<00:00, 10.33it/s]
  2%|          | 2/116 [00:00<00:09, 11.44it/s]epoch:    8       training loss:          0.5642456068047161
100%|██████████| 116/116 [00:11<00:00, 10.37it/s]
  2%|          | 2/116 [00:00<00:09, 11.51it/s]epoch:    9       training loss:          0.532082457737676
100%|██████████| 116/116 [00:11<00:00, 10.35it/s]
  2%|          | 2/116 [00:00<00:09, 11.45it/s]epoch:    10      training loss:          0.5162202844332004
100%|██████████| 116/116 [00:11<00:00, 10.32it/s]
  2%|          | 2/116 [00:00<00:09, 11.42it/s]epoch:    11      training loss:          0.4971266509643916
100%|██████████| 116/116 [00:11<00:00, 10.35it/s]
  2%|          | 2/116 [00:00<00:10, 11.36it/s]epoch:    12      training loss:          0.4814850780470618
100%|██████████| 116/116 [00:11<00:00, 10.35it/s]
  2%|          | 2/116 [00:00<00:09, 11.51it/s]epoch:    13      training loss:          0.45296267574203425
100%|██████████| 116/116 [00:11<00:00, 10.32it/s]
  2%|          | 2/116 [00:00<00:09, 11.41it/s]epoch:    14      training loss:          0.4241378130070094
100%|██████████| 116/116 [00:11<00:00, 10.36it/s]
  2%|          | 2/116 [00:00<00:09, 11.74it/s]epoch:    15      training loss:          0.4075380291404395
100%|██████████| 116/116 [00:11<00:00, 10.36it/s]
  2%|          | 2/116 [00:00<00:09, 11.43it/s]epoch:    16      training loss:          0.37495105032776965
100%|██████████| 116/116 [00:11<00:00, 10.34it/s]
  2%|          | 2/116 [00:00<00:10, 11.36it/s]epoch:    17      training loss:          0.35450426386348133
100%|██████████| 116/116 [00:11<00:00, 10.23it/s]
  2%|          | 2/116 [00:00<00:09, 11.41it/s]epoch:    18      training loss:          0.32281466457864333
100%|██████████| 116/116 [00:11<00:00, 10.33it/s]
  2%|          | 2/116 [00:00<00:09, 11.53it/s]epoch:    19      training loss:          0.2918576194808401
100%|██████████| 116/116 [00:11<00:00, 10.32it/s]epoch:          20      training loss:          0.27600731430896397
```

This is a summary of the training phase. You'll notice that the training loss decreases as we increase the epochs. Let's save the weights of the trained model so we can use them in the future without retraining the model:

```
torch.save(model, 'model.pt')
```

view rawsaving_model.py hosted with by GitHub

If you do not wish to train the model at your end, you can download the weights of the model which I trained for 20 epochs using this link.

Next, let's load this model:

```
the_model = torch.load('model.pt')
```

view rawloading_model.py hosted with by GitHub

Checking Our Model's Performance

Finally, let's make predictions for the training and validation set and check the respective accuracy:

```
torch.manual_seed(0)
```

```
# prediction for training set
prediction = []
target = []
permutation = torch.randperm(final_train.size()[0])
for i in tqdm(range(0,final_train.size()[0], batch_size)):
    indices = permutation[i:i+batch_size]
    batch_x, batch_y = final_train[indices], final_target_train[indices]

    if torch.cuda.is_available():
        batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

    with torch.no_grad():
        output = model(batch_x.cuda())

    softmax = torch.exp(output).cpu()
    prob = list(softmax.numpy())
    predictions = np.argmax(prob, axis=1)
    prediction.append(predictions)
    target.append(batch_y)

# training accuracy
accuracy = []
for i in range(len(prediction)):
    accuracy.append(accuracy_score(target[i].cpu(),prediction[i]))

print('training accuracy: \t', np.average(accuracy))
```

```
100%|██████████| 116/116 [00:04<00:00, 24.55it/s]
training accuracy:       0.9114104406130268
```

We got an accuracy of more than 91% on the training set! That's quite promising. But let's wait before we celebrate. We need to check the same for the validation set:

```
# checking the performance on validation set

torch.manual_seed(0)

output = model(val_x.cuda())

softmax = torch.exp(output).cpu()

prob = list(softmax.detach().numpy())

predictions = np.argmax(prob, axis=1)

accuracy_score(val_y, predictions)
```

view rawvalidation_accuracy.py hosted with by GitHub

```
0.7757575757575758
```

The validation accuracy is around 78%. That's quite good!

End Notes

This is how we can use image augmentation techniques when we are given less training data to begin with.

In this article, we covered most of the commonly used image augmentation techniques. We learned how to rotate, shift, and flip images. We also learned how we can add random noise to images or blur them. Then we discussed basic guidelines for selecting the right augmentation technique.

You can try these image augmentation techniques on any image classification problem and then compare the performance with and without augmentation. Feel free to share your results in the comments section below.

And if you're new to deep learning, computer vision and image data in general, I suggest going through the below course:

- Computer Vision using Deep Learning 2.0

Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch

PULKIT SHARMA, OCTOBER 22, 2019

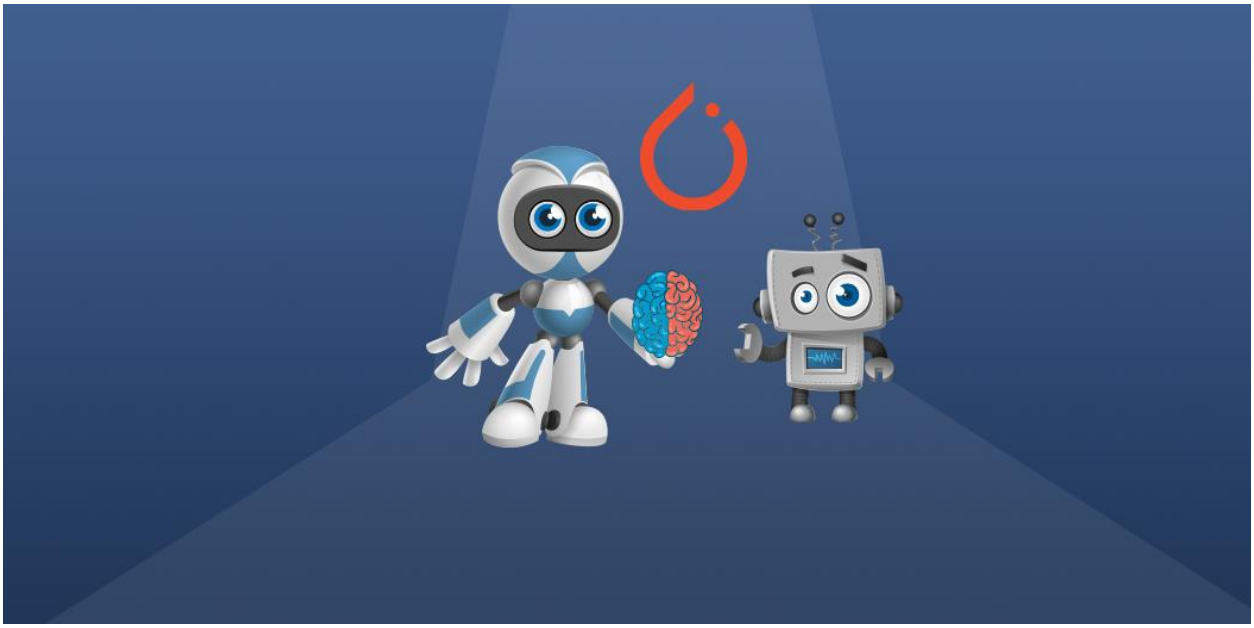
Overview

- The art of transfer learning could transform the way you build machine learning and deep learning models
- Learn how transfer learning works using PyTorch and how it ties into using pre-trained models
- We'll work on a real-world dataset and compare the performance of a model built using convolutional neural networks (CNNs) versus one built using transfer learning

Introduction

I was working on a computer vision project last year where we had to build a robust face detection model. The concept behind that is fairly straightforward – it's the execution part that always sticks in my mind.

Given the size of the dataset we had, building a model from scratch was a real challenge. It was going to be potentially time-consuming and a strain on the computational resources we had. We had to figure out a solution quickly because we were working with a tight deadline.



This is when the powerful concept of transfer learning came to our rescue. It is a really helpful tool to have in your data scientist armoury, especially when you're working with limited time and computational power.

So in this article, we will learn all about transfer learning and how to leverage it on a real-world project using Python. We'll also discuss the role of pre-trained models in this space and how they'll change the way you build machine learning pipelines.

This article is part of my PyTorch series for beginners. I strongly believe PyTorch is one of the best deep learning frameworks right now and will only go from strength to strength in the near future. This is a great time to learn how it works and get onboard. Make sure you check out the previous articles in this series:

- *A Beginner-Friendly Guide to PyTorch and How it Works from Scratch*
- *Build an Image Classification Model using Convolutions Neural Networks (CNNs) in PyTorch*

If you are completely new to CNNs, you can learn them comprehensively by enrolling in this free course: Convolutional Neural Networks (CNN) from Scratch

Table of Contents

1. Introduction to Transfer Learning
2. What are Pre-trained Models? And how to Pick the Right Pre-trained Model?
3. Case Study: Emergency vs Non-Emergency Vehicle Classification
4. Solving the Challenge using Convolutional Neural Networks (CNNs)

5. Solving the Challenge using Transfer Learning and PyTorch
6. Performance Comparison of CNN and Transfer Learning

Introduction to Transfer Learning

Let me illustrate the concept of transfer learning using an example. Picture this – you want to learn a topic from a domain you're completely new to. Pick any domain and any topic – you can think of deep learning and neural networks as well.

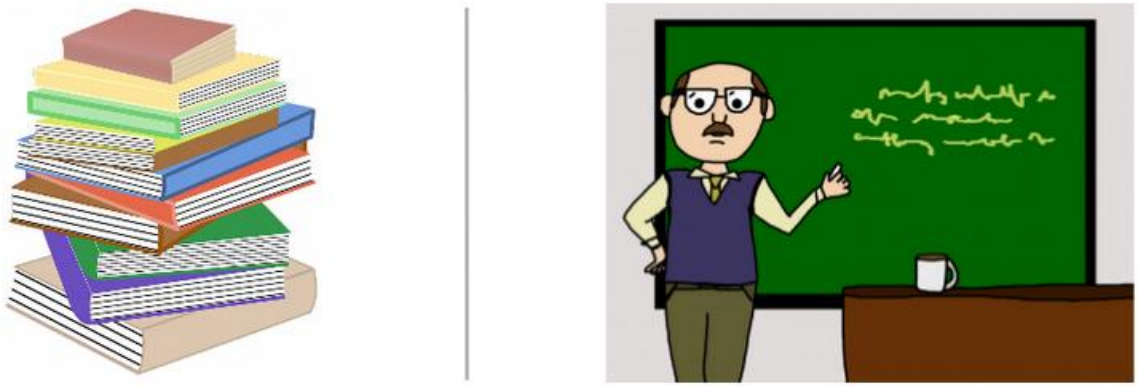
What are the different approaches you would take to understand the topic? Off the top of my head:

- Search online for resources
- Read articles and blogs
- Refer to books
- Look out for video tutorials, and so on

All of these will help you get comfortable with the topic. In this situation, you are the only person who is putting in all the effort.

But there's another approach, which might yield better results in a short amount of time.

You can consult a domain/topic expert who has a solid grasp on the topic you want to learn. This person will transfer his/her knowledge to you. thus expediting your learning process.



Learning from Scratch

Transfer Learning

The first approach, where you are putting in all the effort alone, is an example of learning from scratch. The second approach is referred to as transfer learning. There is a knowledge transfer happening from an expert in that domain to a person who is new to it.

Yes, the idea behind transfer learning is that straightforward!

Neural Networks and Convolutional Neural Networks (CNNs) are examples of learning from scratch. Both these networks extract features from a given set of images (in case of an image related task) and then classify the images into their respective classes based on these extracted features.

This is where transfer learning and pre-trained models are so useful. Let's understand a bit about the latter concept in the next section.

What are Pre-trained Models and how to Pick the Right Pre-trained Model?

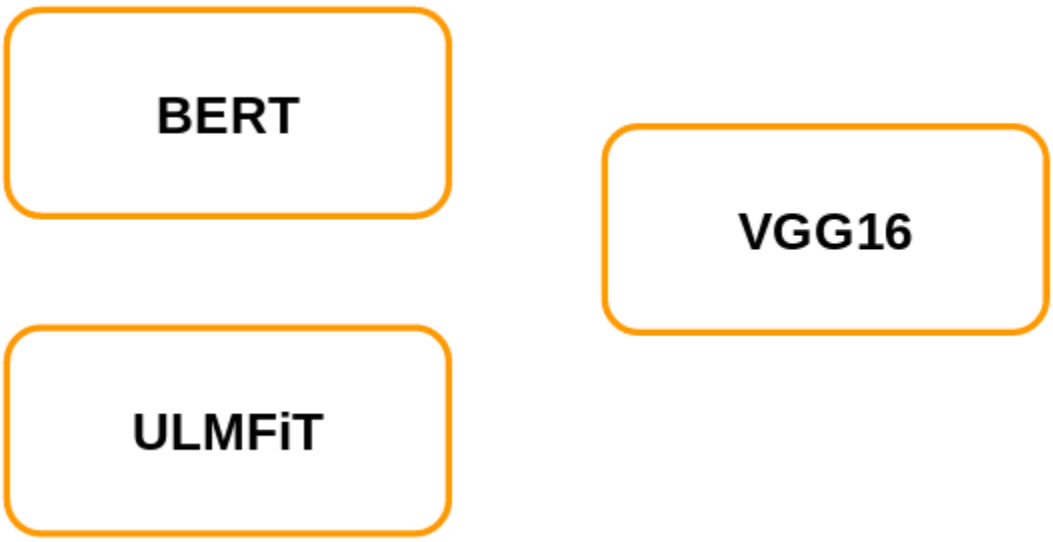
Pre-trained models are super useful in any deep learning project that you'll work on. Not all of us have the unlimited computational power of the top tech behemoths. We need to make do with our local machines so pre-trained models are a blessing there.

> *A pre-trained model, as you might have surmised already, is a model already designed and trained by a certain person or team to solve a specific problem.*

Recall that we learn the weights and biases while training models like Neural Network and CNNs. These weights and biases, when multiplied with the image pixels, help to generate features.

Pre-trained models share their learning by passing their weights and biases matrix to a new model. So, whenever we do transfer learning, we will first select the right pre-trained model and then pass its weight and bias matrix to the new model.

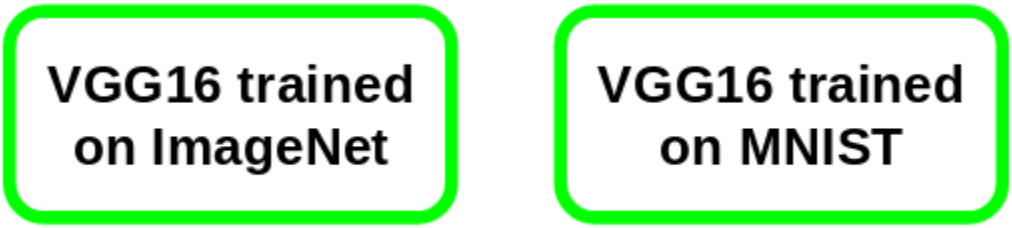
There are *n* number of pre-trained models available out there. We need to decide which will be the best-suited model for our problem. For now, let's consider that we have three pre-trained networks available – BERT, ULMFiT, and VGG16.



Our task is to classify the images (as we have been doing in the previous articles of this series). So, which of these pre-trained models will you pick? Let me first give you a quick overview of these pre-trained networks which will help us to decide the right pre-trained model.

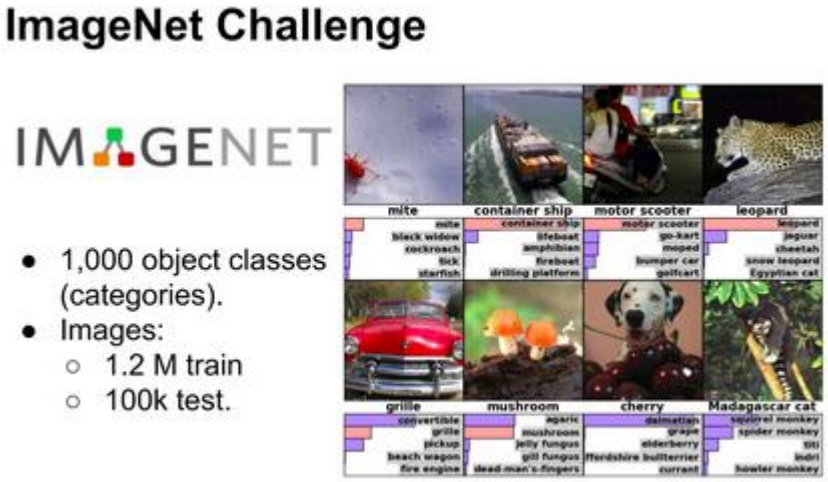
BERT and ULMFiT are used for language modeling and VGG16 is used for image classification tasks. And if you look at the problem at hand, it is an image classification one. So it stands to reason that we will pick VGG16.

Now, VGG16 can have different weights, i.e. VGG16 trained on ImageNet or VGG16 trained on MNIST:

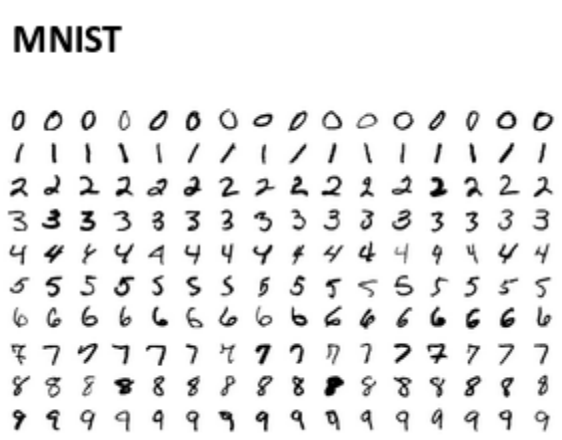


ImageNet vs. MNIST

Now, to decide the right pre-trained model for our problem, we should explore these ImageNet and MNIST datasets. The ImageNet dataset consists of 1000 classes and a total of 1.2 million images. Some of the classes in this data are animals, cars, shops, dogs, food, instruments, etc.:



MNIST, on the other hand, is trained on handwritten digits. It includes 10 classes from 0 to 9:



We will be working on a project where we need to classify images into emergency and non-emergency vehicles (we will discuss this in more detail in the next section). This dataset includes images of vehicles so a VGG16 model trained on the ImageNet dataset would be more useful for us as it has images of vehicles.

This, in a nutshell, is how we should decide the right pre-trained model based on our problem.

Case Study: Emergency vs Non-Emergency Vehicle Classification

Ideally, we would be using the Identify the Apparels problem for this article. We've worked on it in the previous two articles of this series and that would help in comparing our progress.

Unfortunately, this isn't possible here because VGG16 requires that the images should be of the shape (224,224,3) (the images in the other problem are of shape (28,28)). One way to combat this could have been to resize these (28,28) images to (224,224,3) but this will not make sense intuitively.

Here's the good part – we'll be working on a brand new project! Here, our aim is to classify the vehicles as emergency or non-emergency.

This project is also a part of the Computer Vision using Deep Learning course by Analytics Vidhya. To work on more such interesting projects and learn the concepts of computer vision in much more detail, feel free to check out the course.

Let's now start with understanding the problem and visualizing a few examples. ***You can download the images using this link***. First, import the required libraries:

```
# importing the libraries

import pandas as pd

import numpy as np

from tqdm import tqdm


# for reading and displaying images

from skimage.io import imread

from skimage.transform import resize

import matplotlib.pyplot as plt

%matplotlib inline


# for creating validation set

from sklearn.model_selection import train_test_split


# for evaluating the model

from sklearn.metrics import accuracy_score
```

```
# PyTorch libraries and modules

import torch

from torch.autograd import Variable

from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d,
Module, Softmax, BatchNorm2d, Dropout

from torch.optim import Adam, SGD


# torchvision for pre-trained models

from torchvision import models
```

view rawlibrary.py hosted with by GitHub

Next, we will read the .csv file containing the image name and the corresponding label:

```
# loading dataset

train = pd.read_csv('emergency_train.csv')

train.head()
```

view rawload_dataset.py hosted with by GitHub

	image_names	emergency_or_not
0	1503.jpg	0
1	1420.jpg	0
2	1764.jpg	0
3	1356.jpg	0
4	1117.jpg	0

There are two columns in the .csv file:

1. **image_names:** It represents the name of all the images in the dataset
2. **emergency_or_no:** It specifies whether that particular image belongs to the emergency or non-emergency class. 0 means that the image is a non-emergency vehicle and 1 represents an emergency vehicle

Next, we will load all the images and store them in an array format:

```
# loading training images
train_img = []
for img_name in tqdm(train['image_names']):
    # defining the image path
    image_path = 'images/' + img_name
    # reading the image
    img = imread(image_path)
    # normalizing the pixel values
    img = img/255
    # resizing the image to (224,224,3)
    img = resize(img, output_shape=(224,224,3), mode='constant', anti_aliasing=True)
    # converting the type of pixel to float 32
    img = img.astype('float32')
    # appending the image into the list
    train_img.append(img)

# converting the list to numpy array
train_x = np.array(train_img)
train_x.shape
```

view rawtrain_image.py hosted with by GitHub

```
100%|██████████| 1646/1646 [00:12<00:00, 136.81it/s]
(1646, 224, 224, 3)
```

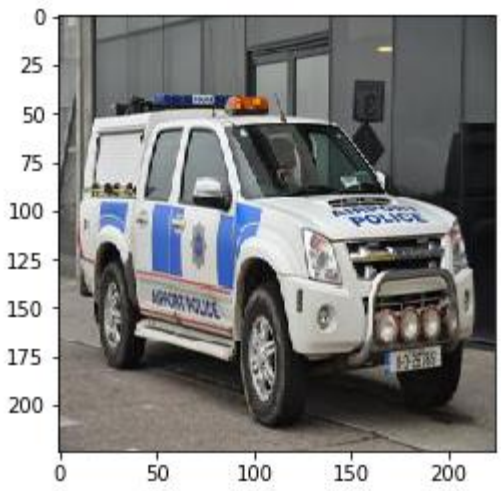
It took approximately 12 seconds to load these images. There are 1,646 images in our dataset and we have reshaped all of them to (224,224,3) since VGG16 requires all the images in this particular shape. Let's now visualize a few images from the dataset:

```
# Exploring the data
index = 10
plt.imshow(train_x[index])
```

```
if (train['emergency_or_not'][index] == 1):
    print('It is an Emergency vehicle')
else:
    print('It is a Non-Emergency vehicle')
```

view rawsample_image.py hosted with by GitHub

```
It is an Emergency vehicle
```



This is a police car and hence has a label of Emergency vehicle. Now we will store the target in a separate variable:

```
# defining the target
train_y = train['emergency_or_not'].values
```

view rawlabel.py hosted with by GitHub

Let's create a validation set to evaluate our model:

```
# create validation set
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size = 0.1,
random_state = 13, stratify=train_y)
(train_x.shape, train_y.shape), (val_x.shape, val_y.shape)
```

view rawcreating_validation_set.py hosted with by GitHub

```
(((1481, 224, 224, 3), (1481,)), ((165, 224, 224, 3), (165,)))
```

We have 1,481 images in the training set and remaining 165 images in the validation set. We now have to convert the dataset into torch format:

```
# converting training images into torch format
train_x = train_x.reshape(1481, 3, 224, 224)
train_x  = torch.from_numpy(train_x)

# converting the target into torch format
train_y = train_y.astype(int)
train_y = torch.from_numpy(train_y)

# shape of training data
train_x.shape, train_y.shape
```

view rawtrain_image_to_torch.py hosted with by GitHub

```
(torch.Size([1481, 3, 224, 224]), torch.Size([1481]))
```

Similarly, we will convert the validation set:

```
# converting validation images into torch format
val_x = val_x.reshape(165, 3, 224, 224)
val_x  = torch.from_numpy(val_x)

# converting the target into torch format
val_y = val_y.astype(int)
val_y = torch.from_numpy(val_y)

# shape of validation data
val_x.shape, val_y.shape
```

view rawvalidation_image_to_torch.py hosted with by GitHub

```
(torch.Size([165, 3, 224, 224]), torch.Size([165]))
```

Our data is ready! In the next section, we will build a Convolutional Neural Network (CNN) before we use the pre-trained model to solve this problem.

Solving the Challenge using Convolutional Neural Networks (CNNs)

We are finally at the model building part! Before using transfer learning to solve the problem, let's use a CNN model and set a benchmark for ourselves.

We will build a very simple CNN architecture with two convolutional layers to extract features from images and a dense layer at the end to classify these features:

```
class Net(Module):
    def __init__(self):
        super(Net, self).__init__()

        self.cnn_layers = Sequential(
            # Defining a 2D convolution layer
            Conv2d(3, 4, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(4),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
            # Defining another 2D convolution layer
            Conv2d(4, 8, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(8),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
```

```
        )


        self.linear_layers = Sequential(
            Linear(8 * 56 * 56, 2)
        )


    # Defining the forward pass
    def forward(self, x):
        x = self.cnn_layers(x)
        x = x.view(x.size(0), -1)
        x = self.linear_layers(x)
        return x
```

view rawcnn_pytorch.py hosted with by GitHub

Let's now define the optimizer, learning rate and the loss function for our model and use a GPU to train the model:

```
# defining the model
model = Net()
# defining the optimizer
optimizer = Adam(model.parameters(), lr=0.0001)
# defining the loss function
criterion = CrossEntropyLoss()
# checking if GPU is available
if torch.cuda.is_available():
    model = model.cuda()
    criterion = criterion.cuda()


print(model)
```

view rawcnn_model_pytorch.py hosted with by GitHub

```
Net(
  (cnn_layers): Sequential(
    (0): Conv2d(3, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear_layers): Sequential(
    (0): Linear(in_features=25088, out_features=2, bias=True)
  )
)
```

This is how the architecture of the model looks like. Finally, we will train the model for 15 epochs. I am setting the *batch_size* of the model to 128 (you can play around with this):

```
# batch size of the model

batch_size = 128



# number of epochs to train the model

n_epochs = 15



for epoch in range(1, n_epochs+1):



    # keep track of training and validation loss

    train_loss = 0.0



    permutation = torch.randperm(train_x.size()[0])



    training_loss = []

    for i in tqdm(range(0,train_x.size()[0], batch_size)):
```

```
            indices = permutation[i:i+batch_size]
            batch_x, batch_y = train_x[indices], train_y[indices]

            if torch.cuda.is_available():
                batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

            optimizer.zero_grad()
            # in case you wanted a semi-full example
            outputs = model(batch_x)
            loss = criterion(outputs,batch_y)

            training_loss.append(loss.item())
            loss.backward()
            optimizer.step()

        training_loss = np.average(training_loss)
        print('epoch: \t', epoch, '\t training loss: \t', training_loss)
```

view rawtraining_cnn_pytorch.py hosted with by GitHub

```
 73%|███████   | 11/15 [00:08<00:02,  1.46it/s]
epoch:   11      training loss:          0.5055197974046072
 80%|████████  | 12/15 [00:08<00:02,  1.48it/s]
epoch:   12      training loss:          0.5015172933538755
 87%|████████  | 13/15 [00:09<00:01,  1.49it/s]
epoch:   13      training loss:          0.4827266087134679
 93%|█████████ | 14/15 [00:10<00:00,  1.50it/s]
epoch:   14      training loss:          0.47801993290583294
100%|██████████| 15/15 [00:10<00:00,  1.50it/s]
epoch:   15      training loss:          0.4621712590257327
```

This will print a summary of the training as well. The training loss is decreasing after each epoch and that's a good sign. Let's check the training as well as the validation accuracy:

```
# prediction for training set
prediction = []
target = []
permutation = torch.randperm(train_x.size()[0])
for i in tqdm(range(0,train_x.size()[0], batch_size)):
    indices = permutation[i:i+batch_size]
    batch_x, batch_y = train_x[indices], train_y[indices]

    if torch.cuda.is_available():
        batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

    with torch.no_grad():
        output = model(batch_x.cuda())

    softmax = torch.exp(output).cpu()
    prob = list(softmax.numpy())
    predictions = np.argmax(prob, axis=1)
    prediction.append(predictions)
    target.append(batch_y)

# training accuracy
accuracy = []
for i in range(len(prediction)):
    accuracy.append(accuracy_score(target[i],prediction[i]))

print('training accuracy: \t', np.average(accuracy))
```

view rawprediction_train_cnn.py hosted with by GitHub

```
100%|██████████| 12/12 [00:00<00:00, 18.86it/s]
training accuracy:       0.815728453196347
```

We got a training accuracy of around 82% which is a good score. Let's now check the validation accuracy:

```
# prediction for validation set
prediction_val = []
target_val = []
permutation = torch.randperm(val_x.size()[0])
for i in tqdm(range(0,val_x.size()[0], batch_size)):
    indices = permutation[i:i+batch_size]
    batch_x, batch_y = val_x[indices], val_y[indices]

    if torch.cuda.is_available():
        batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

    with torch.no_grad():
        output = model(batch_x.cuda())

    softmax = torch.exp(output).cpu()
    prob = list(softmax.numpy())
    predictions = np.argmax(prob, axis=1)
    prediction_val.append(predictions)
    target_val.append(batch_y)

# validation accuracy
accuracy_val = []
```

```
for i in range(len(prediction_val)):

    accuracy_val.append(accuracy_score(target_val[i],prediction_val[i]))


print('validation accuracy: \t', np.average(accuracy_val))
```

view rawprediction_validation_cnn.py hosted with by GitHub

```
100%|██████████| 2/2 [00:00<00:00, 24.96it/s]
validation accuracy:     0.7626689189189189
```

The validation accuracy comes out to be 76%. Now that we have a benchmark with us, it's time to use transfer learning to solve this emergency versus non-emergency vehicle classification problem. Let's get rolling!

Solving the Challenge using Transfer Learning

I've touched on this above and I'll reiterate it here – we will be using the VGG16 pre-trained model trained on the ImageNet dataset. Let's look at the steps we will be following to train the model using transfer learning:

1. First, we will load the weights of the pre-trained model – VGG16 in our case
2. Then we will fine tune the model as per the problem at hand
3. Next, we will use these pre-trained weights and extract features for our images
4. Finally, we will train the fine tuned model using the extracted features

So, let's start by loading the weights of the model:

```
# loading the pretrained model

model = models.vgg16_bn(pretrained=True)
```

view rawloading_pretrained_model.py hosted with by GitHub

We will now fine tune the model. We will not be training the layers of the VGG16 model and hence let's freeze the weights of these layers:

```
# Freeze model weights

for param in model.parameters():
```

```
        param.requires_grad = False
```

view rawfreezing_model_weights.py hosted with by GitHub

Since we only have 2 classes to predict and VGG16 is trained on ImageNet which has 1000 classes, we need to update the final layer as per our problem:

Since we will be training only the last layer, I have set the *requires_grad* as True for the last layer. Let's set the training to GPU:

```
# checking if GPU is available
if torch.cuda.is_available():
    model = model.cuda()
```

view rawgpu.py hosted with by GitHub

```
# Add on classifier
model.classifier[6] = Sequential(
                      Linear(4096, 2))
for param in model.classifier[6].parameters():
    param.requires_grad = True
```

view rawadding_classifier.py hosted with by GitHub

We'll now use the model and extract features for both the training and validation images. I will set the *batch_size* as 128 (again, you can increase or decrease this *batch_size* per your requirement):

```
# batch_size
batch_size = 128


# extracting features for train data
data_x = []
label_x = []


inputs,labels = train_x, train_y
```

```
for i in tqdm(range(int(train_x.shape[0]/batch_size)+1)):
    input_data = inputs[i*batch_size:(i+1)*batch_size]
    label_data = labels[i*batch_size:(i+1)*batch_size]
    input_data , label_data = Variable(input_data.cuda()),Variable(label_data.cuda())
    x = model.features(input_data)
    data_x.extend(x.data.cpu().numpy())
    label_x.extend(label_data.data.cpu().numpy())
```

view rawfeatures_train.py hosted with by GitHub

Similarly, let's extract features for our validation images:

```
# extracting features for validation data
data_y = []
label_y = []

inputs,labels = val_x, val_y

for i in tqdm(range(int(val_x.shape[0]/batch_size)+1)):
    input_data = inputs[i*batch_size:(i+1)*batch_size]
    label_data = labels[i*batch_size:(i+1)*batch_size]
    input_data , label_data = Variable(input_data.cuda()),Variable(label_data.cuda())
    x = model.features(input_data)
    data_y.extend(x.data.cpu().numpy())
    label_y.extend(label_data.data.cpu().numpy())
```

view rawfeatures_validation.py hosted with by GitHub

Next, we will convert these data into torch format:

```
# converting the features into torch format
```

```
x_train  = torch.from_numpy(np.array(data_x))
x_train = x_train.view(x_train.size(0), -1)
y_train  = torch.from_numpy(np.array(label_x))
x_val  = torch.from_numpy(np.array(data_y))
x_val = x_val.view(x_val.size(0), -1)
y_val  = torch.from_numpy(np.array(label_y))
```

view rawfeatures_in_torch.py hosted with by GitHub

We also have to define the optimizer and the loss function for our model:

```
import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate
optimizer = optim.Adam(model.classifier[6].parameters(), lr=0.0005)
```

view rawpre-trained.py hosted with by GitHub

It's time to train the model. We will train it for 30 epochs with a batch_size set to 128:

```
# batch size
batch_size = 128

# number of epochs to train the model
n_epochs = 30

for epoch in tqdm(range(1, n_epochs+1)):

    # keep track of training and validation loss
```

```
train_loss = 0.0


permutation = torch.randperm(x_train.size()[0])


training_loss = []
for i in range(0,x_train.size()[0], batch_size):


    indices = permutation[i:i+batch_size]
    batch_x, batch_y = x_train[indices], y_train[indices]


    if torch.cuda.is_available():
        batch_x, batch_y = batch_x.cuda(), batch_y.cuda()


    optimizer.zero_grad()
    # in case you wanted a semi-full example
    outputs = model.classifier(batch_x)
    loss = criterion(outputs,batch_y)


    training_loss.append(loss.item())
    loss.backward()
    optimizer.step()


training_loss = np.average(training_loss)
print('epoch: \t', epoch, '\t training loss: \t', training_loss)
```

```
epoch:   26      training loss:          0.3568757598598798
 90%|[redacted] | 27/30 [00:06<00:00,  4.42it/s]
epoch:   27      training loss:          0.3510003586610158
 93%|[redacted] | 28/30 [00:06<00:00,  4.42it/s]
epoch:   28      training loss:          0.3469578077395757
 97%|[redacted] | 29/30 [00:06<00:00,  4.42it/s]
epoch:   29      training loss:          0.33534184098243713
100%|[redacted]| 30/30 [00:06<00:00,  4.42it/s]
epoch:   30      training loss:          0.35356302311023075
```

Here is a summary of the model. **You can see that the loss has decreased and hence we can say that the model is improving.** Let's validate this by looking at the training and validation accuracies:

```
# prediction for training set
prediction = []
target = []
permutation = torch.randperm(x_train.size()[0])
for i in tqdm(range(0,x_train.size()[0], batch_size)):
    indices = permutation[i:i+batch_size]
    batch_x, batch_y = x_train[indices], y_train[indices]

    if torch.cuda.is_available():
        batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

    with torch.no_grad():
        output = model.classifier(batch_x.cuda())

    softmax = torch.exp(output).cpu()
    prob = list(softmax.numpy())
```

```
        predictions = np.argmax(prob, axis=1)
        prediction.append(predictions)
        target.append(batch_y)

    # training accuracy
    accuracy = []
    for i in range(len(prediction)):
        accuracy.append(accuracy_score(target[i],prediction[i]))

    print('training accuracy: \t', np.average(accuracy))
```

view rawtrain_accuracy_transfer_learning.py hosted with by GitHub

```
100%|██████████| 12/12 [00:00<00:00, 47.52it/s]
training accuracy:       0.8370522973744293
```

We got an accuracy of ~ 84% on the training set. Let's now check the validation accuracy:

```
    # prediction for validation set
    prediction_val = []
    target_val = []
    permutation = torch.randperm(x_val.size()[0])
    for i in tqdm(range(0,x_val.size()[0], batch_size)):
        indices = permutation[i:i+batch_size]
        batch_x, batch_y = x_val[indices], y_val[indices]

        if torch.cuda.is_available():
            batch_x, batch_y = batch_x.cuda(), batch_y.cuda()

        with torch.no_grad():
```

```
            output = model.classifier(batch_x.cuda())


        softmax = torch.exp(output).cpu()
        prob = list(softmax.numpy())
        predictions = np.argmax(prob, axis=1)
        prediction_val.append(predictions)
        target_val.append(batch_y)


    # validation accuracy
    accuracy_val = []
    for i in range(len(prediction_val)):
        accuracy_val.append(accuracy_score(target_val[i],prediction_val[i]))


    print('validation accuracy: \t', np.average(accuracy_val))
```

view rawvalidation_accuracy_transfer_learning.py hosted with by GitHub

```
100%|██████████| 2/2 [00:00<00:00, 68.50it/s]
validation accuracy:     0.8347761824324325
```

The validation accuracy of the model is also similar, i,e, 83%. **The training and validation accuracies are almost in sync and hence we can say that the model is generalized.** Here is the summary of our results:

Model	Training Accuracy	Validation Accuracy
CNN	81.57%	76.26%
VGG16	83.70%	83.47%

We can infer that the accuracies have improved by using the VGG16 pre-trained model as compared to the CNN model. Got to love the art of transfer learning!

End Notes

In this article, we learned how to use pre-trained models and transfer learning to solve an image classification problem. We first understood what pre-trained models are and how to choose the right pre-trained model depending on the problem at hand. Then we took a case study of classifying images of vehicles as emergency or non-emergency. We solved this case study using a CNN model first and then we used the VGG16 pre-trained model to solve the same problem.

We found that using the VGG16 pre-trained model significantly improved the model performance and we got better results as compared to the CNN model. I hope you now have a clear understanding of how to use transfer learning and the right pre-trained model to solve problems using PyTorch.

I encourage you to take other image classification problems and try to apply transfer learning to solve them. This will help you to grasp the concept much more clearly.

As always, if you have some feedback or doubts related to this tutorial, feel free to post them in the comments section below and I will be happy to answer them.

Get Started with PyTorch – Learn How to Build Quick & Accurate Neural Networks (with 4 Case Studies!)

SHIVAM BANSAL, JANUARY 14, 2019

Introduction

PyTorch v TensorFlow – how many times have you seen this polarizing question pop up on social media? The rise of deep learning in recent times has been fuelled by the popularity of

these frameworks. There are staunch supporters of both, but a clear winner has started to emerge in the last year.

PyTorch was one of the most popular frameworks in 2018. It quickly became the preferred go-to deep learning framework among researchers in both academia and the industry. After using PyTorch for the last few weeks, I can confirm that it is highly flexible and an easy-to-use deep learning library.

In this article, we will explore what PyTorch is all about. But our learning won't stop with the theory – we will code through 4 different use cases and see how well PyTorch performs. Building deep learning models has never been this fun!

Note: This article assumes that you have a basic understanding of deep learning concepts. If not, I recommend going through this article.

If you prefer to approach the following concepts in a structured format, you can enrol for this free course on PyTorch and follow them chapter-wise.

Contents

- What is PyTorch?
- Building Neural Nets using PyTorch
- Use Case 1: Handwritten Digits Classification (Numerical Data, MLP)
- Use Case 2: Objects Image Classification (Image Data, CNN)
- Use Case 3: Sentiment Text Classification (Text Data, RNN)
- Use Case 4: Image Style Transfer (Transfer Learning)

What is PyTorch?

Let's understand what PyTorch is and why it has become so popular lately, before diving into it's implementation.

PyTorch is a Python-based scientific computing package that is ***similar to NumPy, but with the added power of GPUs***. It is also a ***deep learning framework that provides maximum flexibility and speed*** during implementing and building deep neural network architectures.

Recently, PyTorch 1.0 was released and it was aimed to assist researchers by addressing four major challenges:

- Extensive reworking
- Time-consuming training
- Python programming language inflexibility
- Slow scale-up

Intrinsically, there are two main characteristics of PyTorch that distinguish it from other deep learning frameworks like Tensorflow:

- *Imperative Programming*
- *Dynamic Computation Graphing*

Imperative Programming: PyTorch performs computations as it goes through each line of the written code. This is quite similar to how a Python program is executed. This concept is called imperative programming. The biggest advantage of this feature is that your code and programming logic can be debugged on the fly.

Dynamic Computation Graphing: PyTorch is referred to as a "defined by run" framework, which means that the computational graph structure (of a neural network architecture) is generated during run time. The main advantage of this property is that it provides a flexible and programmatic runtime interface that facilitates the construction and modification of systems by connecting operations. In PyTorch, a new computational graph is defined at each forward pass. This is in stark contrast to TensorFlow which uses a static graph representation.

PyTorch 1.0 comes with an important feature called **torch.jit**, a high-level compiler that allows the user to separate the models and code. It also supports **efficient model optimization on custom hardware, such as GPUs or TPUs**.

Building Neural Nets using PyTorch

Let's understand PyTorch through a more practical lens. Learning theory is good, but it isn't much use if you don't put it into practice!

A PyTorch implementation of a neural network looks exactly like a NumPy implementation. The goal of this section is to showcase the equivalent nature of PyTorch and NumPy. For this purpose, let's create a simple three-layered network having 5 nodes in the input layer, 3

in the hidden layer, and 1 in the output layer. We will use only one training example with one row which has five features and one target.

```
import torch


n_input, n_hidden, n_output = 5, 3, 1
```

The first step is to do ***parameter initialization***. Here, the weights and bias parameters for each layer are initialized as the tensor variables. ***Tensors*** are the base data structures of PyTorch which are used for building different types of neural networks. They can be considered as the generalization of arrays and matrices; in other words, tensors are N-dimensional matrices.

```
## initialize tensor for inputs, and outputs


x = torch.randn((1, n_input))


y = torch.randn((1, n_output))


## initialize tensor variables for weights


w1 = torch.randn(n_input, n_hidden) # weight for hidden layer


w2 = torch.randn(n_hidden, n_output) # weight for output layer


## initialize tensor variables for bias terms


b1 = torch.randn((1, n_hidden)) # bias for hidden layer


b2 = torch.randn((1, n_output)) # bias for output layer
```

After the parameter initialization step, a neural network can be defined and trained in four key steps:

- *Forward Propagation*
- *Loss computation*
- *Backpropagation*
- *Updating the parameters*

Let's see each of these steps in a bit more detail.

Forward Propagation: In this step, activations are calculated at every layer using the two steps shown below. These activations flow in the forward direction from the input layer to the output layer in order to generate the final output.

1. *z = weight * input + bias*
2. *a = activation_function (z)*

The following code blocks show how we can write these steps in PyTorch. Notice that most of the functions, such as exponential and matrix multiplication, are similar to the ones in NumPy.

```
## sigmoid activation function using pytorch


def sigmoid_activation(z):


    return 1 / (1 + torch.exp(-z))


## activation of hidden layer


z1 = torch.mm(x, w1) + b1


a1 = sigmoid_activation(z1)


## activation (output) of final layer


z2 = torch.mm(a1, w2) + b2
```

```
output = sigmoid_activation(z2)
```

Loss Computation: In this step, the error (also called loss) is calculated in the output layer. A simple loss function can tell the difference between the actual value and the predicted value. Later, we will look at different loss functions available in PyTorch.

```
loss = y - output
```

Backpropagation: The aim of this step is to minimize the error in the output layer by making marginal changes in the bias and the weights. These marginal changes are computed using the derivatives of the error term.

Based on the Calculus principle of the Chain rule, the delta changes are back passed to hidden layers where corresponding changes in their weights and bias are made. This leads to an adjustment in the weights and bias until the error is minimized.

```
## function to calculate the derivative of activation


def sigmoid_delta(x):


  return x * (1 - x)


## compute derivative of error terms


delta_output = sigmoid_delta(output)


delta_hidden = sigmoid_delta(a1)


## backpass the changes to previous layers
```

```
d_outp = loss * delta_output


loss_h = torch.mm(d_outp, w2.t())


d_hidn = loss_h * delta_hidden
```

Updating the Parameters: Finally, the weights and bias are updated using the delta changes received from the above backpropagation step.

```
learning_rate = 0.1


w2 += torch.mm(a1.t(), d_outp) * learning_rate


w1 += torch.mm(x.t(), d_hidn) * learning_rate


b2 += d_outp.sum() * learning_rate


b1 += d_hidn.sum() * learning_rate
```

Finally, when these steps are executed for a number of epochs with a large number of training examples, the loss is reduced to a minimum value. The final weight and bias values are obtained which can then be used to make predictions on the unseen data.

Use Case 1: Handwritten Digital Classification

In the previous section, we saw a simple use case of PyTorch for writing a neural network from scratch. In this section, we will use different utility packages provided within PyTorch (nn, autograd, optim, torchvision, torchtext, etc.) to build and train neural networks.

Neural networks can be defined and managed easily using these packages. In our use case, we will create a ***Multi-Layered Perceptron (MLP)*** network for building a handwritten digit classifier. We will make use of the MNIST dataset included in the torchvision package.

The first step, as with any project you'll work on, is **data preprocessing**. We need to transform the raw dataset into tensors and normalize them in a fixed range. The torchvision package provides a utility called ***transforms*** which can be used to combine different transformations together.

```
from torchvision import transforms


_tasks = transforms.Compose([

    transforms.ToTensor(),

    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

    ])
```

The first transformation converts the raw data into tensor variables and the second transformation performs normalization using the below operation:

$$x\_normalized = x\text{-}mean / std$$

The values 0.5 and 0.5 represent the mean and standard deviation for 3 channels: red, green, and blue.

```
from torchvision.datasets import MNIST


## Load MNIST Dataset and apply transformations


mnist = MNIST("data", download=True, train=True, transform=_tasks)
```

Another excellent utility of PyTorch is ***DataLoader*** iterators which provide the ability to batch, shuffle and load the data in parallel using multiprocessing workers. For the purpose of evaluating our model, we will partition our data into training and validation sets.

```
from torch.utils.data import DataLoader


from torch.utils.data.sampler import SubsetRandomSampler


## create training and validation split


split = int(0.8 * len(mnist))


index_list = list(range(len(mnist)))


train_idx, valid_idx = index_list[:split], index_list[split:]


## create sampler objects using SubsetRandomSampler


tr_sampler = SubsetRandomSampler(train_idx)


val_sampler = SubsetRandomSampler(valid_idx)


## create iterator objects for train and valid datasets
```

```
trainloader = DataLoader(mnist, batch_size=256, sampler=tr_sampler)


validloader = DataLoader(mnist, batch_size=256, sampler=val_sampler)
```

The neural network architectures in PyTorch can be defined in a class which inherits the properties from the base class from nn package called ***Module***. This inheritance from the `nn.Module` class allows us to implement, access, and call a number of methods easily. We can define all the layers inside the constructor of the class, and the forward propagation steps inside the forward function.

We will define a network with the following layer configurations: [784, 128,10]. This configuration represents the 784 nodes (28*28 pixels) in the input layer, 128 in the hidden layer, and 10 in the output layer. Inside the forward function, we will use the sigmoid activation function in the hidden layer (which can be accessed from the nn module).

```
import torch.nn.functional as F


class Model(nn.Module):


    def __init__(self):


        super().__init__()


        self.hidden = nn.Linear(784, 128)


        self.output = nn.Linear(128, 10)




   def forward(self, x):


        x = self.hidden(x)
```

```
        x = F.sigmoid(x)

        x = self.output(x)

        return x

model = Model()
```

Define the loss function and the optimizer using the *nn* and *optim* package:

```
from torch import optim

loss_function = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay= 1e-6, momentum = 0.9
, nesterov = True)
```

We are now ready to train the model. The core steps will remain the same as we saw earlier: Forward Propagation, Loss Computation, Backpropagation, and updating the parameters.

```
for epoch in range(1, 11): ## run the model for 10 epochs

    train_loss, valid_loss = [], []

    ## training part

    model.train()

    for data, target in trainloader:
```

```
        optimizer.zero_grad()

        ## 1. forward propagation

        output = model(data)



        ## 2. loss calculation

        loss = loss_function(output, target)



        ## 3. backward propagation

        loss.backward()



        ## 4. weight optimization

        optimizer.step()



        train_loss.append(loss.item())
```

```
    ## evaluation part

    model.eval()

    for data, target in validloader:

        output = model(data)

        loss = loss_function(output, target)

        valid_loss.append(loss.item())

    print ("Epoch:", epoch, "Training Loss: ", np.mean(train_loss), "Valid Loss: ", n
p.mean(valid_loss))
```

```
>> Epoch: 1  Training Loss:  0.645777 Valid Loss:  0.344971

>> Epoch: 2  Training Loss:  0.320241 Valid Loss:  0.299313

>> Epoch: 3  Training Loss:  0.278429 Valid Loss:  0.269018

>> Epoch: 4  Training Loss:  0.246289 Valid Loss:  0.237785

>> Epoch: 5  Training Loss:  0.217010 Valid Loss:  0.217133

>> Epoch: 6  Training Loss:  0.193017 Valid Loss:  0.206074

>> Epoch: 7  Training Loss:  0.174385 Valid Loss:  0.180163
```

```
>> Epoch: 8  Training Loss:  0.157574 Valid Loss:  0.170064

>> Epoch: 9  Training Loss:  0.144316 Valid Loss:  0.162660

>> Epoch: 10 Training Loss:  0.133053 Valid Loss:  0.152957
```

Once the model is trained, make the predictions on the validation data.

```
## dataloader for validation dataset

dataiter = iter(validloader)

data, labels = dataiter.next()

output = model(data)

_, preds_tensor = torch.max(output, 1)

preds = np.squeeze(preds_tensor.numpy())

print ("Actual:", labels[:10])

print ("Predicted:", preds[:10])

>>> Actual: [0 1 1 1 2 2 8 8 2 8]

>>> Predicted: [0 1 1 1 2 2 8 8 2 8]
```

Use Case 2: Object Image Classification

Let’s take things up a notch.

In this use case, we will create convolutional neural network (CNN) architectures in PyTorch. We will perform object image classification using the popular CIFAR-10 dataset. This dataset is also included in the torchvision package. The entire procedure to define and train the model will remain the same as the previous use case, except the introduction of additional layers in the network.

Let’s load and transform the dataset:

```
## load the dataset


from torchvision.datasets import CIFAR10


cifar = CIFAR10('data', train=True, download=True, transform=_tasks)


## create training and validation split


split = int(0.8 * len(cifar))
```

```
index_list = list(range(len(cifar)))

train_idx, valid_idx = index_list[:split], index_list[split:]

## create training and validation sampler objects

tr_sampler = SubsetRandomSampler(train_idx)

val_sampler = SubsetRandomSampler(valid_idx)

## create iterator objects for train and valid datasets

trainloader = DataLoader(cifar, batch_size=256, sampler=tr_sampler)

validloader = DataLoader(cifar, batch_size=256, sampler=val_sampler)
```

We will create the architecture with three convolutional layers for low-level feature extraction, three pooling layers for maximum information extraction, and two linear layers for linear classification.

```
class Model(nn.Module):

    def __init__(self):

        super(Model, self).__init__()



        ## define the layers

        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
```

```
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        self.linear1 = nn.Linear(1024, 512)

        self.linear2 = nn.Linear(512, 10)



    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))

        x = self.pool(F.relu(self.conv2(x)))

        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 1024) ## reshaping

        x = F.relu(self.linear1(x))

        x = self.linear2(x)

        return x
```

```
model = Model()
```

Define the loss function and the optimizer:

```
import torch.optim as optim

loss_function = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay= 1e-6, momentum = 0.9
, nesterov = True)

## run for 30 Epochs

for epoch in range(1, 31):

    train_loss, valid_loss = [], []

    ## training part

    model.train()

    for data, target in trainloader:

        optimizer.zero_grad()

        output = model(data)

        loss = loss_function(output, target)

        loss.backward()
```

```
        optimizer.step()

        train_loss.append(loss.item())



    ## evaluation part

    model.eval()

    for data, target in validloader:

        output = model(data)

        loss = loss_function(output, target)

        valid_loss.append(loss.item())
```

Once the model is trained, we can generate predictions on the validation set.

```
## dataloader for validation dataset

dataiter = iter(validloader)

data, labels = dataiter.next()

output = model(data)

_, preds_tensor = torch.max(output, 1)
```

```
preds = np.squeeze(preds_tensor.numpy())


print ("Actual:", labels[:10])


print ("Predicted:", preds[:10])


Actual: ['truck', 'truck', 'truck', 'horse', 'bird', 'truck', 'ship', 'bird', 'deer',

'bird']


Pred:   ['truck', 'automobile', 'automobile', 'horse', 'bird', 'airplane', 'ship', 'b

ird', 'deer', 'bird']
```

Use Case 3: Sentiment Text Classification

We'll pivot from computer vision use cases to natural language processing. The idea is to showcase the utility of PyTorch in a variety of domains in deep learning.

In this section, we'll leverage PyTorch for text classification tasks using RNN (Recurrent Neural Networks) and LSTM (Long Short Term Memory) layers. First, we will load a dataset containing two fields—text and target. The target contains two classes, class1 and class2, and our task is to classify each text into one of these classes.

You can download the dataset here.

```
train = pd.read_csv("train.csv")


x_train = train["text"].values


y_train = train['target'].values
```

I highly recommend setting seeds before getting into the heavy coding. This ensures that the results you will see are the same as mine – a very useful (and helpful) feature when learning new concepts.

```
np.random.seed(123)

torch.manual_seed(123)

torch.cuda.manual_seed(123)

torch.backends.cudnn.deterministic = True
```

In the preprocessing step, convert the text data into a padded sequence of tokens so that it can be passed into embedding layers. I will use the utilities provided in the Keras package, but the same can be done using the torchtext package as well.

```
from keras.preprocessing import text, sequence

## create tokens

tokenizer = Tokenizer(num_words = 1000)

tokenizer.fit_on_texts(x_train)

word_index = tokenizer.word_index

## convert texts to padded sequences

x_train = tokenizer.texts_to_sequences(x_train)

x_train = pad_sequences(x_train, maxlen = 70)
```

Next, we need to convert the tokens into vectors. I will use pretrained GloVe word embeddings for this purpose. We will load these word embeddings and create an embedding matrix containing the word vector for every word in the vocabulary.

```
EMBEDDING_FILE = 'glove.840B.300d.txt'

embeddings_index = {}

for i, line in enumerate(open(EMBEDDING_FILE)):

    val = line.split()

    embeddings_index[val[0]] = np.asarray(val[1:], dtype='float32')

embedding_matrix = np.zeros((len(word_index) + 1, 300))

for word, i in word_index.items():

    embedding_vector = embeddings_index.get(word)

    if embedding_vector is not None:

        embedding_matrix[i] = embedding_vector
```

Define the model architecture with embedding layers and LSTM layers:

```
class Model(nn.Module):

    def __init__(self):

        super(Model, self).__init__()
```

```
        ## Embedding Layer, Add parameter
        self.embedding = nn.Embedding(max_features, embed_size)
        et = torch.tensor(embedding_matrix, dtype=torch.float32)
        self.embedding.weight = nn.Parameter(et)
        self.embedding.weight.requires_grad = False
        self.embedding_dropout = nn.Dropout2d(0.1)
        self.lstm = nn.LSTM(300, 40)
        self.linear = nn.Linear(40, 16)
        self.out = nn.Linear(16, 1)
        self.relu = nn.ReLU()
    def forward(self, x):
        h_embedding = self.embedding(x)
        h_lstm, _ = self.lstm(h_embedding)
        max_pool, _ = torch.max(h_lstm, 1)
```

```
        linear = self.relu(self.linear(max_pool))

        out = self.out(linear)

        return out

model = Model()
```

Create training and validation sets:

```
from torch.utils.data import TensorDataset

## create training and validation split

split_size = int(0.8 * len(train_df))

index_list = list(range(len(train_df)))

train_idx, valid_idx = index_list[:split], index_list[split:]

## create iterator objects for train and valid datasets

x_tr = torch.tensor(x_train[train_idx], dtype=torch.long)

y_tr = torch.tensor(y_train[train_idx], dtype=torch.float32)

train = TensorDataset(x_tr, y_tr)

trainloader = DataLoader(train, batch_size=128)
```

```
x_val = torch.tensor(x_train[valid_idx], dtype=torch.long)

y_val = torch.tensor(y_train[valid_idx], dtype=torch.float32)

valid = TensorDataset(x_val, y_val)

validloader = DataLoader(valid, batch_size=128)
```

Define loss and optimizers:

```
loss_function = nn.BCEWithLogitsLoss(reduction='mean')

optimizer = optim.Adam(model.parameters())
```

Training the model:

```
## run for 10 Epochs

for epoch in range(1, 11):

    train_loss, valid_loss = [], []

## training part

    model.train()

    for data, target in trainloader:

        optimizer.zero_grad()
```

```
        output = model(data)

        loss = loss_function(output, target.view(-1,1))

        loss.backward()

        optimizer.step()

        train_loss.append(loss.item())



    ## evaluation part

    model.eval()

    for data, target in validloader:

        output = model(data)

        loss = loss_function(output, target.view(-1,1))

        valid_loss.append(loss.item())
```

Finally, we can obtain the predictions:

```
dataiter = iter(validloader)

data, labels = dataiter.next()
```

```
output = model(data)


_, preds_tensor = torch.max(output, 1)


preds = np.squeeze(preds_tensor.numpy())


Actual: [0 1 1 1 1 0 0 0 0]


Predicted: [0 1 1 1 1 1 1 1 0 0]
```

Use Case #4: Image Style Transfer

Let's look at one final use case where we will perform artistic style transfer. This is one of the most creative projects I have worked on and hopefully you'll have fun with this as well. The basic idea behind the style transfer concept is:

- Take the objects/context from one image
- Take the style/texture from a second image
- Generate a final image which is a mixture of the two

This concept was introduced in the paper: **"Image Style Transfer using Convolutional Networks".** An example of style transfer is shown below:

Awesome, right? Let’s look at it’s implementation in PyTorch. The process involves six steps:

- **Low-level feature extraction from both the input images**. This can be done using pretrained deep learning models such as VGG19.

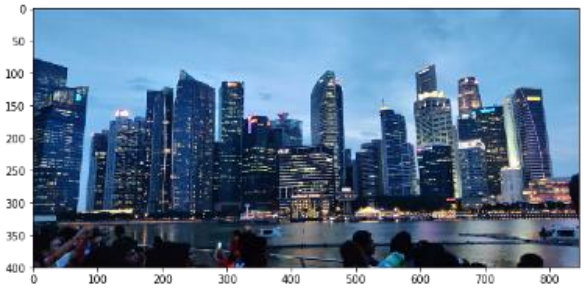
```
from torchvision import models


# get the features portion from VGG19


vgg = models.vgg19(pretrained=True).features




# freeze all VGG parameters


for param in vgg.parameters():
```

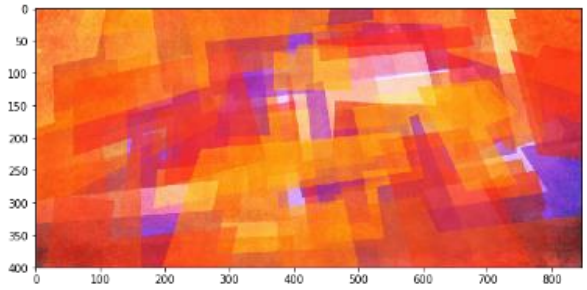
```
    param.requires_grad_(False)

# check if GPU is available

device = torch.device("cpu")

if torch.cuda.is_available():

    device = torch.device("cuda")

vgg.to(device)
```

- Load the two images on the device and obtain the features from VGG. Also, apply the transformations: resize to tensor, and normalization of values.

```
from torchvision import transforms as tf

def transformation(img):

    tasks = tf.Compose([tf.Resize(400), tf.ToTensor(),

                tf.Normalize((0.44,0.44,0.44),(0.22,0.22,0.22))])

    img = tasks(img)[:3,:,:].unsqueeze(0)

    return img

img1 = Image.open("image1.jpg").convert('RGB')

img2 = Image.open("image2.jpg").convert('RGB')
```

```
img1 = transformation(img1).to(device)


img2 = transformation(img2).to(device)
```



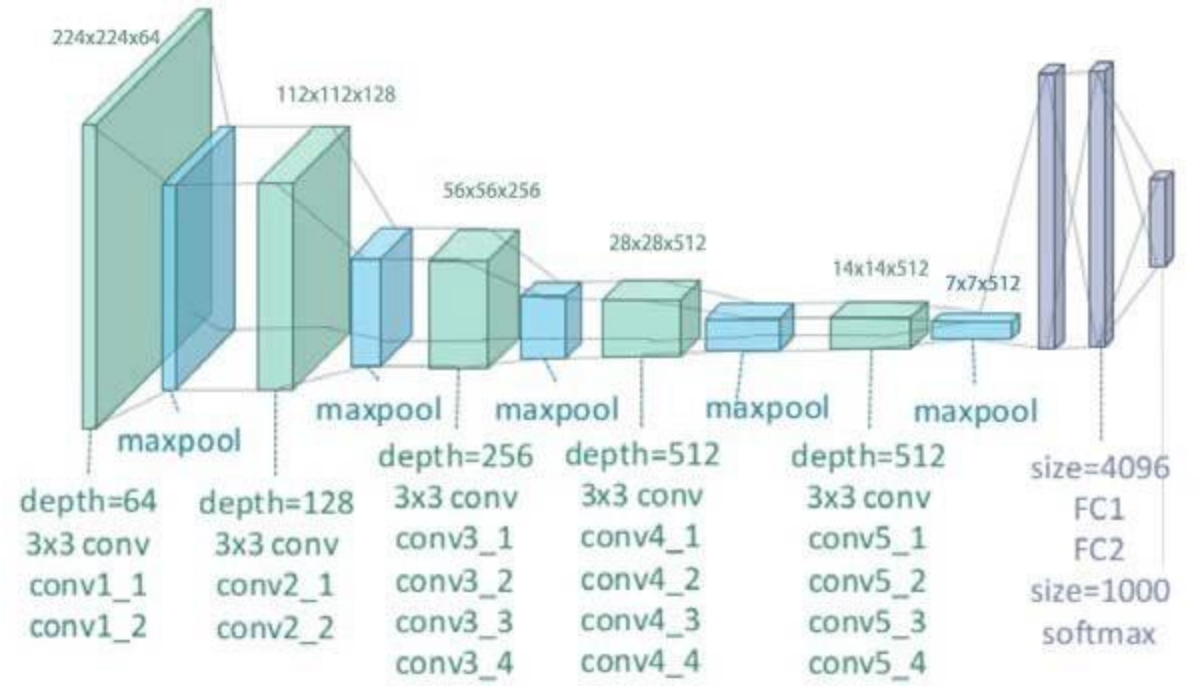


- Now, we need to obtain the relevant features of the two images. From the first image, we need to extract features related to the context or the objects present. From the second image, we need to extract features related to styles and textures.

Object Related Features: In the original paper, the authors have suggested that more valuable information about objects and context can be extracted from the initial layers of the network. This is because in the higher layers, the information space becomes more complex and detailed pixel information is lost.

Style Related Features: In order to obtain the texture information from the second image, the authors used ***correlations*** between different features in different layers. This is explained in detail in point 4 below.

But before getting there, let's look at the structure of a typical VGG19 model:



For object information extraction, Conv42 was the layer of interest. It's present in the 4th convolutional block with a depth of 512. For style representation, the layers of interest were the first convolutional layer of every convolutional block in the network, i.e., conv11, conv21, conv31, conv41, and conv51. These layers were selected purely based on the author's experiments and I am only replicating their results in this article.

```
def get_features(image, model):

    layers = {'0': 'conv1_1', '5': 'conv2_1',  '10': 'conv3_1',

              '19': 'conv4_1', '21': 'conv4_2', '28': 'conv5_1'}

    x = image

    features = {}

    for name, layer in model._modules.items():

        x = layer(x)
```

```
        if name in layers:
            features[layers[name]] = x
    return features
img1_features = get_features(img1, vgg)
img2_features = get_features(img2, vgg)
```

- As mentioned in the previous point, the authors used correlations in different layers to obtain the style related features. These feature correlations are given by the Gram matrix G, where every cell (i, j) in G is the inner product between the vectorised feature maps *i* and *j* in a layer.

```
def correlation_matrix(tensor):
    _, d, h, w = tensor.size()
    tensor = tensor.view(d, h * w)
    correlation = torch.mm(tensor, tensor.t())
    return correlation
correlations = {l: correlation_matrix(img2_features[l]) for l in
                                                    img2_features}
```

- We can finally perform style transfer using these features and correlations,. Now in order to transfer the style from one image to the other, we need to set the weight of every layer used to obtain style features. As mentioned above, the initial layers

provide more information so we'll set more weight for these layers. Also, define the optimizer function and the target image which will be the copy of image1.

```
weights = {'conv1_1': 1.0, 'conv2_1': 0.8, 'conv3_1': 0.25,

           'conv4_1': 0.21, 'conv5_1': 0.18}



target = img1.clone().requires_grad_(True).to(device)

optimizer = optim.Adam([target], lr=0.003)
```

- Start the loss minimization process in which we run the loop for a large number of steps and calculate the loss related to object feature extraction and style feature extraction. Using the minimized loss, the network parameters are updated which further updates the target image. After some iterations, the updated image will be generated.

```
for ii in range(1, 2001):



    ## calculate the content loss (from image 1 and target)

    target_features = get_features(target, vgg)

    loss = target_features['conv4_2'] - img1_features['conv4_2']

    content_loss = torch.mean((loss)**2)
```

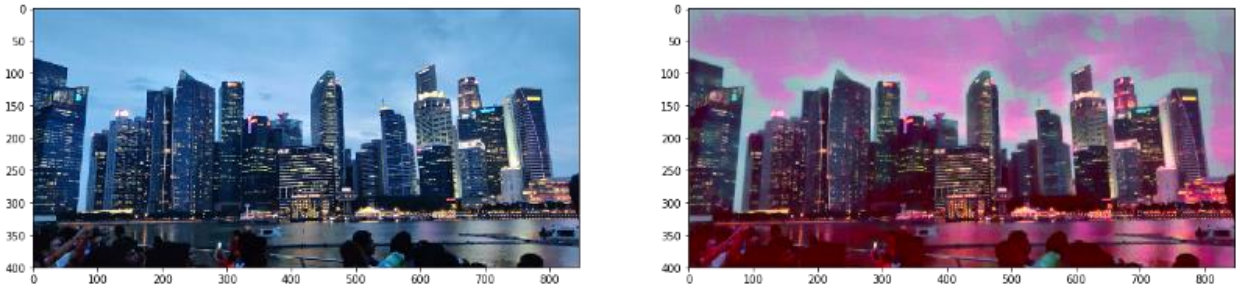
```
## calculate the style loss (from image 2 and target)

style_loss = 0

for layer in weights:



    target_feature = target_features[layer]

    target_corr = correlation_matrix(target_feature)

    style_corr = correlations[layer]



    layer_loss = torch.mean((target_corr - style_corr)**2)

    layer_loss *= weights[layer]



    _, d, h, w = target_feature.shape

    style_loss += layer_loss / (d * h * w)



total_loss = 1e6 * style_loss + content_loss
```

```
    optimizer.zero_grad()

    total_loss.backward()

    optimizer.step()
```

In the end, we can view the predicted results. I ran it for only a small number of iterations, but one can run up to 3000 iterations (if computation resources are no bar!).

```
def tensor_to_image(tensor):

    image = tensor.to("cpu").clone().detach()

    image = image.numpy().squeeze()

    image = image.transpose(1, 2, 0)

    image *= np.array((0.22, 0.22, 0.22))

                        + np.array((0.44, 0.44, 0.44))

    image = image.clip(0, 1)

    return image

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

ax1.imshow(tensor_to_image(img1))
```

```
ax2.imshow(tensor_to_image(target))
```



End Notes

There are plenty of other use cases where PyTorch can, and has been used. It has quickly become the darling of researchers around the globe. The majority of the open-source libraries and developments you'll see happening nowadays have a PyTorch implementation available on GitHub.

In this article, I have illustrated what PyTorch is and how you can get started with implementing it in different use cases in deep learning. One should treat this guide as the starting point. The performance in every use case can be improved with more data, more fine-tuning of network parameters, and most importantly, applying creative skills while building network architectures. Thanks for reading and do leave your feedback in the comments section below.

References

1. Official PyTorch Guide: https://pytorch.org/tutorials/
2. Deep Learning with PyTorch: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
3. Faizan's Article on AnalyticsVidhya: https://www.analyticsvidhya.com/blog/2018/02/pytorch-tutorial/
4. Udacity Deep Learning using Pytorch: https://github.com/udacity/deep-learning-v2-pytorch
5. Image Style Transfer Original Paper: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf