Chapter 10

PyTorch Basics

PyTorch is one of the most widely used package for artificial intelligence and machine learning. PyTorch is maintained by Facebook. PyTorch by design has been developed to be as close to numpy for an easy transition to PyTorch for those who are familiar with scientific computation in Python .

The building block for PyTorch and for that matter most of deep learning tools is a Tensor. Tensor is similar to numpy array. Let's start with importing torch and torchvision modules.

```
import torch
import torchvision
import torch.nn as nn
import numpy as np
import torchvision.transforms as transforms
```

To start off, we start by defined zero dimensional, one and two dimensional tensors. Notice that similar to numpy, associated with a PyTorch tensor is the notion of datatype, shape and size.

```
x=torch.tensor(2.)
print(x)
print(x.dtype)
print(x.shape)


y=torch.tensor([1,2,3])
print(y)
print(y.dtype)
print(y.shape)


z=torch.tensor([[1,2],[3,4]])
print(z)
print(z.dtype)
print(z.shape)
```

Similar to array in numpy the basic object for PyTorch is a tensor. Tensor can be thought of as a multi-dimensional array. We can generate different types of tensor in PyTorch .

```
#computational graph
x=torch.tensor(2.,requires_grad=True)
y=x*x +2*x+3
```

```
y.backward()
print(x.grad)
```

The fundamental block of computation defined in PyTorch is a computation graph. The forward pass in the computational graph is just going from input to output. The backward() command computes the gradient with respect to all the variables for which the "require_grad" flag is set to "True" the default value of this flag is "True".

```
x=torch.tensor(2.,requires_grad=True)

lr=torch.tensor([1e-2])
for i in range(1000):
  print('iteraion is:', i)
  x=torch.tensor(x,requires_grad=True)
  y=x*x +2*x+3
  y.backward()
  with torch.no_grad():
    x=x-lr*x.grad.data
  print(x,y)
```

As a first example in PyTorch , we will implement gradient descent algorithm on a simple function. Notice, how convenient it is to implement the algorithm in PyTorch . We don't have to compute the gradients explicitly rather PyTorch keeps track of the gradients for us. All we have to do is invoke the "backward()" command for the function and then we can use the gradient with respect to independent variable to update the variable until we reach the local minima.

Question: Implement the gradient descent using PyTorch for the following functions:

```
 dx_in, hl1, dy_out, n = 32, 50, 2 , 1000

x = torch.randn(dx_in, n)
y = torch.randn(dy_out, n)


w1 = torch.randn(hl1, x_in, requires_grad=True)
w2 = torch.randn(y_out, hl1,requires_grad=True)

learning_rate = 1e-6
for t in range(100):
  z=w1.mm(x).clamp(min=0)
  y_pred = w2.mm(z)


  loss = (y_pred - y).pow(2).sum()
  print(t, loss.item())

  loss.backward()


  with torch.no_grad():
    w1 -= learning_rate * w1.grad
```

```
    w2 -= learning_rate * w2.grad

    w1.grad.zero_()
    w2.grad.zero_()
```

The above example implements a two layer simple neural network in PyTorch . You can select the input dimensions, data size and the number of hidden unit and output units on the first lime of the code.

Let's take a deeper look at the code and try to see what the code is doing exactly. We are trying to setup a regression problem and hence you can see that both x and y are sampled from normal distribution. If it were a classification we would have sample y from Bernoulli distribution.

Similarly, we can see that weights of both layers also sampled from normal distribution. Notice that the require_grad flag has been set to "True" because we need to optimize the network with respect to these weights.

Note: Notice, that we have to initialize the weights of the network randomly, otherwise we will that gradient of the loss function for multiple weights of the network will be exactly same and they will evolve to same values.

In this example we are setting the learning rate of $1e-6$. Some good question to ask here are: Is this the best learning weight we can select for this problem? Should we keep the learning rate constant during the learning process?

Example motivated from https://github.com/jcjohnson/pytorch-examples#pytorch-autograd

```
 x = torch.randn(1000, 10)
y = torch.randn(1000, 2)

# Build a fully connected layer.
linear1 = nn.Linear(10, 50)
linear2 = nn.Linear(50,2)


# Build loss function and optimizer.
criterion = nn.MSELoss()
optimizer1 = torch.optim.SGD(linear1.parameters(), lr=0.01)
optimizer2 = torch.optim.SGD(linear2.parameters(), lr=0.01)


for i in range(100):
    optimizer1.zero_grad()
    optimizer2.zero_grad()
    h1 = linear1(x).clamp(min=0)
    pred = linear2(h1)
    loss = criterion(pred, y)
    loss.backward()
    optimizer1.step()
    optimizer2.step()
    print('loss after step optimization: ',i, loss.item())
```

```
 # Fully connected neural network with one hidden layer
class Net(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, out_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, out_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out


x = torch.randn(1000, 10)
y = torch.randn(1000, 2)

net=Net(10,50,2)

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

for i in range(100):
    optimizer.zero_grad()
    pred = net(x)
    loss = criterion(pred, y)
    loss.backward()
    optimizer.step()
    print('loss after step optimization: ',i, loss.item())
```

Examples motivated form https://github.com/yunjey/pytorch-tutorial

10.1 Data Loading and Transformation

An import aspect of deep learning is the manipulation and augmentation of data. Notice, that in most cases the datasets are huge and can't be loaded in memory, not to mention the fact that we have to select random batches for the data to implement methods like stochastic gradient decent. Both `PyTorch` and `TensorFlow` provide tools for data manipulation and augmentation. In this section we will see an overview of these methods in `PyTorch` and see how these methods work.

```
import torch
import numpy as np
import os
import imageio as io
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from torchvision import transforms, utils
from skimage import io, transform
```

For the dataloader we will use the above packages. We will explaing the packages as they will be used in the code below.

```
%matplotlib inline
```

```
dataDir='' #directory here
D=os.listdir(dataDir)
print(len(D))
Dr=[]
for name in D:
    if not 'jpg' in name:
        Dr.append(name)
        name

for name in Dr:
    name
    D.remove(name)
```

In the code above we are trying to make a list of all jpg images in the `dataDir` folder. The command `os.listdir` gives a list of all the files in the folder. Notice that we might have other type of files in the folder as well. Just to remove the chances of any error we remove the non jpg extension files from the list. We loop over the directory `D` and collect all the non jpg images in a list `Dr` and in the last part of the code we remove elements of `Dr` from `D`.

```
class KaustDataset(Dataset):
    """Our dataset ."""

    def __init__(self,img_list, root_dir, transform=None):

        self.images = img_list
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = os.path.join(self.root_dir,
                                self.images[idx])
        image = io.imread(img_name)
        sample = {'image': image}

        if self.transform:
            sample = self.transform(sample)

        return sample

kaust_dataset = KaustDataset(img_list=D,
                                    root_dir=dataDir)
```

In the code above we extend the `Dataset` class of `PyTorch` and define a new child class called `KaustDataset`. We then define three methods for this class. The first method is the `__init__` method which is used to pass the directory and transformation information that we need for our data pipeline.

The next method that we need to define is the `__len__` method which return the number of elements in our dataset. And lastly we define a method `__getitem__` which gets us an element of the dataset. Everything else is handled by the `Dataset` which is the parent class of our class. Notice that we have an if condition to convert the variable `id` to list type if it is not a list and we apply transform to the item if a transform is provided, by default the value of `transform` is `None`.

In the last part of the above code we define an object of the class `KaustDataset` with no transforms.

```
for i in range(len(kaust_dataset)):
    print(D[i])
    sample = kaust_dataset[i]

    print(i, sample['image'].shape)

    plt.imshow(sample['image'])


    plt.show()
```

The above code displays all images in the dataset.

Next, we want to apply some transformation to the dataset in the data loader pipeline. We define two classes of transformation, one to resize the image and the other to normalize the image. Notice that these type of transforms are very important in data pipelines and are used for data pre-processing and adjustment.

```
class Resize(object):
    """Resize the image."""

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image = sample['image']

        h, w = image.shape[:2]
        img = transform.resize(image,
        (self.output_size, self.output_size))


        return {'image': img}
```

To use PyTorch `transform` we define classes for our transformation, where we have two methods, `__init__` for passing arguments to the class and `__call__` to perform the transformation on a sample of the data. In the above code we resize the image to the input size.

```
class Normalize(object):
    """Normalize the image"""
```

```
    def __init__(self):
        print("init")
    def __call__(self, sample):
        image = sample['image']

        h, w,c = image.shape
        img=copy.copy(image)
        for i in range(c):
            mean=sum(sum(image[:,:,i]))/(h*w)
            std=(image[:,:,i]-mean)
            std=abs(std)
            img[:,:,i]=(image[:,:,i]-mean)/std



        return {'image': img}
```

Similar to the Resize class for Normalize has two methods __init__ for passing arguments to the class and __call__ to perform the transformation on a sample of the data.

Exercise 10.1: We have used Python 2 convention for class definitions. Try Python 3 class definitions and see if it is still works.

```
transformed_dataset = KaustDataset(img_list=D,
                                   root_dir=dataDir,
                                   transform=transforms.Compose
                                   ([
                                           Resize(124),
                                           Normalize()
                                                          ]))



print(len(transformed_dataset))

for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]
    print(sample['image'].shape)
```

In the above code we generate an instance of the dataset from KaustDataset but this time we use the transform argument as well and ask for composition of resize and normalize operation.

```
dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)

print(dataloader)
for batch in dataloader:
    print(batch['image'].shape)
    for i in range(batch['image'].shape[0]):
```

```
        plt.imshow(batch['image'][i,:,:,:])
        plt.show()
```

As a final step for the application of the stochastic gradient dataset we need batches of the dataset rather then the entire dataset. To do this we can use the DataLoader of PyTorch and it will generate the batch of the size provided.

Question: Modify the data loader code to add a horizontal flip and to add label from mask directory for each figure.

?

Exercises

Note: Below codes use the following plotting function:

```
def plotClass(X,y,p):
    plt.figure()
    for i in range(y.shape[0]):
        if y[i]==0:
            plt.plot(X[i,0],X[i,1],'r'+p)
        else:
            plt.plot(X[i,0],X[i,1],'b'+p)

    plt.show()
```

!

Exercise 10.2: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the value of offset the number of layers and hidden units on the classification result.

```
num_data=100 # data points per class
offset=5
x1=np.random.randn(2,num_data)+offset
x0=np.random.randn(2,num_data)

y1=np.ones((1,num_data))
y0=np.zeros((1,num_data))

x=np.concatenate((x1,x0),axis=1)
y=np.concatenate((y1,y0), axis=1)

print(x.shape)
print(y.shape)


plt.plot(x[0,:100],x[1,:100],'b*')
plt.plot(x[0,100:],x[1,100:],'r*')
x=x.T
```

```
x=torch.from_numpy(x).float()
y=torch.from_numpy(y.T).float()
```

Exercise 10.3: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=200

x=np.random.randn(2,num_data)
x=np.random.randn(2,num_data)

y=(x[0,:]**2+x[1,:]**2)>0.5
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```

Exercise 10.4: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=200
x=np.random.randn(2,num_data)


y=(x[0,:]*x[1,:])>0
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```

Exercise 10.5: Use simple neural networks to train a binary classifier for the following data. Study the effect of changing the number of layers and hidden units on the classification result.

```
num_data=500

x=np.random.uniform(-5,5,(2,num_data))


y=(np.floor(x[0,:]%2)!=np.floor(x[1,:]%2))>0
y.astype(int)
x=x.T
y=y.T
y=np.reshape(y,(num_data,1))
plotClass(x,y,'o')

x=torch.from_numpy(x).float()
y=torch.from_numpy(y).float()
```