Pytorch Tutorial

For Data Science HK,
Given by Pranav A

This is how our input data looks like

```
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.86,0.0,0.0,0.0,0.0,0.0,0.0,1.73,0.0,0.86,0.0,0.0,0.0,6.08,3.47,0.
0.09,0.0,0.27,0.0,0.36,0.09,0.0,0.18,0.09,0.0,0.0,0.72,0.0,0.36,0.0,0.0,0.0,0.0,2.0,0.0,3.27,0.0,0.36,0.09,0.0,
0.0,0.0,0.13,0.0,0.26,0.0,0.0,0.65,0.13,0.0,0.0,0.78,0.26,0.0,0.0,0.0,0.13,0.0,0.0,0.0,0.0,0.0,0.13,0.0,1.69,0.
0.0,1.28,0.0,0.0,0.64,0.0,0.0,0.0,0.0,1.28,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.28,0.0,0.0,0.0,0.64,0.64,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,2.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,4.0,2.0,2.0,2.
0.0,1.42,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.42,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,4.28,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.54,0.
0.0,0.0,1.11,0.0,1.11,0.0,0.74,0.0,0.0,0.0,0.74,0.37,0.0,0.0,0.0,0.0,0.37,0.0,3.35,2.98,2.61,0.0,0.0,0.37,0.0,
```

This is how input data looks like in CSV format. We have 3200 dataitems. Each dataitem has 57 features.

The output data is in the 0/1 form.

Let's do some standard imports

```
from numpy import genfromtxt
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import torch.utils.data as Data
import os
```

Set the parameters and hyperparameters

```python
# Parameters and Hyper Parameters
input_size = 57
hidden_size1 = 64
hidden_size2 = 64
num_classes = 2
num_epochs = 256
batch_size = 128
learning_rate = 0.001
```

Convert CSV into numpy array

```
data = genfromtxt('traindata.csv', delimiter=',')
labels = genfromtxt('trainlabel.csv', delimiter=',')
```

`genfromtxt()` function converts CSV file into a numpy array.

Let's divide the data into training and test datasets

```
input_data = data[:3000] # train
test_input = data[3000:] # test
output_data = labels[:3000] # train
test_output = labels[3000:] # test
```

Here we have considered first 3000 rows as our training data. Remaining of them will be used for testing.

Dataloaders

```
train_dataset = Data.TensorDataset(
    data_tensor = torch.from_numpy(input_data).float(),
    target_tensor = torch.from_numpy(output_data).long())
```

Dataloaders are one of the key parts of the Pytorch. But first we have to convert our NumPy arrays to torch tensors.

Here we are using the `TensorDataset` method.

Dataloaders

```
train_dataset = Data.TensorDataset(
    data_tensor = torch.from_numpy(input_data).float(),
    target_tensor = torch.from_numpy(output_data).long())
```

We are now wrapping data tensors and target tensors together.

Input data goes into the `data_tensor`.

Output data goes into the `target_tensor`.

Dataloaders

```
train_dataset = Data.TensorDataset(
    data_tensor = torch.from_numpy(input_data).float(),
    target_tensor = torch.from_numpy(output_data).long())
```

Step 1: We convert `input_data` array into the torch compatible tensor using `torch.from_numpy` function.

Step 2: We typecast the given dataset according datatype.

Dataloaders

```
train_dataset = Data.TensorDataset(
    data_tensor = torch.from_numpy(input_data).float(),
    target_tensor = torch.from_numpy(output_data).long())
```

Input data **HAS** to be typecasted to `float` and output data **HAS** to be typecasted to `long` (for classification problems).

Dataloaders

```
test_dataset = Data.TensorDataset(
    data_tensor = torch.from_numpy(test_input).float(),
    target_tensor = torch.from_numpy(test_output).long())
```

Similarly, we will do that for the test arrays as well.

Dataloaders

```
# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)
```

Now we make an input pipeline using dataloaders using the `Dataloader` method.

Dataloaders

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Similarly we do that for the test datasets (notice that shuffle is disabled this time).

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Now we define our neural net using the class definition.

The first step here is to define the constructor function.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 1: Define your `Net` class by taking `nn.module` as a parameter.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 2: Write your constructor function in the form of `def __init__ (parameters)`

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 3: Call the `super` function so that you can start annotating `self` attributes.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 4: Start building your layers! Here you can use `sequential` as your container.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 5: In the sequential container, make connections from input layer to the first layer. This is done through `Linear` method.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 6: After the connections, we apply activation function here. Here, we have applied `ReLU` function here.

Neural Net Definition

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2,
    num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(input_size, hidden_size1),
            nn.ReLU())
        self.fc2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU())
        self.fc3 = nn.Sequential(
            nn.Linear(hidden_size2, num_classes))
```

Step 7: Follow that for the other layers as well.

Neural Net Definition

```
def forward(self, x):
    out = self.fc1(x)
    out = self.fc2(out)
    out = self.fc3(out)
    return out
```

After defining your classes, you need to define the forward function.

Take the self as the parameter and return it as shown above.

Define some dictionaries

```
#Make a dictionary defining training and validation sets
dataloders = dict()
dataloders['train'] = train_loader
dataloders['val'] = test_loader

dataset_sizes = {'train': 3000, 'val': 220}
use_gpu = torch.cuda.is_available()
```

This is doing for making the code a bit more clean.

I have defined some dictionaries and GPU variables.

Object Declaration

```
net = Net(input_size, hidden_size1, hidden_size2, num_classes)
```

Declare an object of the class that you coded.

Loss and Optimizer

```
# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

We apply Cross Entropy Loss since this is a classification problem.

We use Adam as the optimizer.

It's time to train!

```
if use_gpu:
    model_ft, train_acc, test_acc = train_model(net.cuda(), criterion,
    optimizer, num_epochs)
else:
    model_ft, train_acc, test_acc = train_model(net, criterion,
    optimizer, num_epochs)
```

We will now implement these training models.
Structure is shown above.

Training function

```
def train_model(model, criterion, optimizer, num_epochs):
    f = open("Iterations.txt", "w+")
    best_model_wts = model.state_dict()
    best_val_acc = 0.0
    best_train_acc = 0.0
```

It would be cleaner if we log our results into a file, which we have done in the second line.

We will reload the weights of `model` using `state_dict()`.

Epoch Iteration

```
for epoch in range(num_epochs):
    print('Epoch {}/{}'.format(epoch, num_epochs - 1))
    print('-' * 10)
    for phase in ['train', 'val']:
        if phase == 'train':
            model.train(True)  # Set model to training mode
        else:
            model.train(False)  # Set model to evaluate mode
        running_loss = 0.0
        running_corrects = 0
```

Now we will iterate through each epoch.

If we are training, we will set `model.train` to `True`. Vice versa for the testing part.

Dataloaders iterations

```
# Iterate over data.
for data in dataloders[phase]:
    # get the inputs
    inputs, label = data
    # wrap them in Variable
    if use_gpu:
        inputs = Variable(inputs.cuda())
        labels = Variable(label.cuda())
    else:
        inputs, labels = Variable(inputs), Variable(label)
```

We iterate over dataloaders and wrap the tensors into `Variable()`.

Optimizers and loss

```
# zero the parameter gradients
optimizer.zero_grad()
# forward
outputs = model(inputs)
_, preds = torch.max(outputs.data, 1)
loss = criterion(outputs, labels)
```

Step 1: Zero the gradient vector, so that we can fill it.

Step 2: Retrieve the outputs.

Optimizers and loss

```
# zero the parameter gradients
optimizer.zero_grad()
# forward
outputs = model(inputs)
_, preds = torch.max(outputs.data, 1)
loss = criterion(outputs, labels)
```

Step 3: Take the index which has the maximum value.

Step 4: Calculate loss between outputs and labels using `criterion`.

Calculate loss

```
# backward + optimize only if in training phase
if phase == 'train':
    loss.backward()
    optimizer.step()
# statistics
running_loss += loss.data[0]
running_corrects += torch.sum(preds == label)
```

If loop is in the training phase, then backprop using `loss.backward()` and take a gradient “step” using `optimizer.step()`.

Calculate loss

```python
# backward + optimize only if in training phase
if phase == 'train':
    loss.backward()
    optimizer.step()
# statistics
running_loss += loss.data[0]
running_corrects += torch.sum(preds == label)
```

In order to log, we will calculate losses.

We can retrieve it using `loss.data[0]`.

Calculate loss

```
# backward + optimize only if in training phase
if phase == 'train':
    loss.backward()
    optimizer.step()
# statistics
running_loss += loss.data[0]
running_corrects += torch.sum(preds == label)
```

To calculate number of corrects, we will sum up the predictions which are equal to the correct labels.

Printing stuff

```
epoch_loss = running_loss / dataset_sizes[phase]
epoch_acc = running_corrects / dataset_sizes[phase]
#Print it out Loss and Accuracy and in the file torchvision
print('{} Loss: {:.8f} Accuracy: {:.4f}'.format(phase,
epoch_loss, epoch_acc))
f.write('{} Loss: {:.8f} Accuracy: {:.4f}\n'.format(phase,
epoch_loss, epoch_acc))
```

This is how we can calculate epoch losses, accuracy and store them in the file.

Copying the model

```python
# deep copy the model
if phase == 'val' and epoch_acc > best_val_acc:
    best_val_acc = epoch_acc
    best_model_wts = model.state_dict()
if phase == 'train' and epoch_acc > best_train_acc:
    best_train_acc = epoch_acc
    best_model_wts = model.state_dict()
```

This is how we are deep copying the model and model weights so that we can return them as variables.

Returning the variables

```
f.close()
print('Best val Acc: {:4f}'.format(best_val_acc))
model.load_state_dict(best_model_wts)
return model, best_train_acc, best_val_acc
```

For the best model weights, load them into the model and return the variables.

We are done!

```
Epoch 248/255
----------
train Loss: 0.00035002 Accuracy: 0.9847
val Loss: 0.00110338 Accuracy: 0.9682
Epoch 249/255
----------
train Loss: 0.00048751 Accuracy: 0.9810
val Loss: 0.00124002 Accuracy: 0.9545
Epoch 250/255
----------
train Loss: 0.00037690 Accuracy: 0.9830
val Loss: 0.00109878 Accuracy: 0.9636
Epoch 251/255
----------
train Loss: 0.00045727 Accuracy: 0.9787
val Loss: 0.00127925 Accuracy: 0.9545
Epoch 252/255
----------
train Loss: 0.00066674 Accuracy: 0.9743
val Loss: 0.00155925 Accuracy: 0.9455
Epoch 253/255
----------
train Loss: 0.00047886 Accuracy: 0.9780
val Loss: 0.00163075 Accuracy: 0.9500
Epoch 254/255
----------
train Loss: 0.00043893 Accuracy: 0.9813
val Loss: 0.00132550 Accuracy: 0.9591
Epoch 255/255
----------
train Loss: 0.00049601 Accuracy: 0.9777
val Loss: 0.00129811 Accuracy: 0.9636
Best val Acc: 0.981818
```