# BU.330.760 Deep Learning with Unstructured Data
## Lab 6. PyTorch and Recurrent Neural Network

Learning Goal: practice using PyTorch package to train a recurrent neural network on MNIST dataset. Please note that a convolutional neural network would be better suited for image classification, but this lab makes for a simple example that you are already familiar with
Required Skills: knowledge on recurrent neural network and PyTorch

1. We will use PyTorch (https://pytorch.org) in lab 6, which is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing.
2. You have two ways to use PyTorch, one with Anaconda, and the other with Google Colab
    a. For those who are comfortable with Anaconda so far, install PyTorch packages locally with conda (https://pytorch.org/get-started/locally/). You can use Terminal on Mac system, or on Windows system, **Aanconda Navigator > JupyterLab > Launcher > Terminal**. There is no need for a new environment, you can install it under the main (base) environment.

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

    b. pyTorch is already installed on Google CoLab, so you do not need to install anything for lab 6 if you are using Google CoLab.
3. build the following scripts in Jupyter notebook:
    a. Import required packages.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import os
import numpy as np
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
```

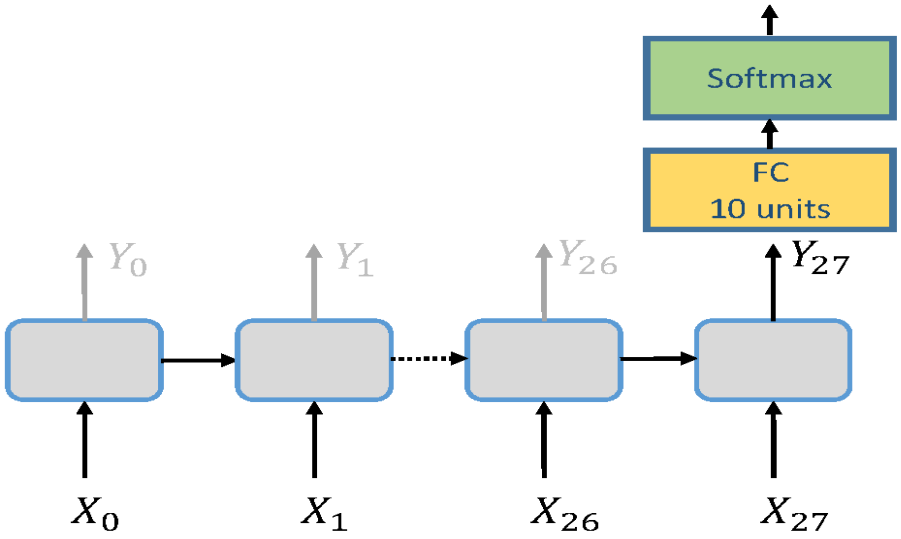
    b. let's import the MNIST dataset, transform the image in the original range [0, 255] to a **torch.FloatTensor** of shape (C x H x W) in the range [0.0, 1.0], and split the dataset into test and train portions.

    Here, **torch.utils.data.DataLoader** class will be used to load multiple samples parallelly. The parameter **batch_size** defines how many samples per batch to load; **shuffle** is set to **True** for training set to have the data reshuffled at every epoch; and **num_workers** defines how many subprocesses to use for data loading (for parallelism).

```
BATCH_SIZE = 64
```

```
transform = transforms.Compose(
    [transforms.ToTensor()])
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)
testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE,
                                         shuffle=False, num_workers=2)
```

c. Next let's define the parameters for the RNN model. We will treat each image as a sequence of 28 rows of 28 pixels each (since each MNIST image is 28x28 pixels). We will use cells of 150 recurrent neurons, plus a fully connected layer containing 10 neurons (one per class) connected to the output of the last time step, followed by a softmax layer. The structure is shown below:




```
N_STEPS = 28
N_INPUTS = 28
N_NEURONS = 150
N_OUTPUTS = 10
N_EPHOCS = 10
```

d. Then let's set up the model called **ImageRNN**, which does the following functions: (1) initialization function **__init__** declares a few variables, and then a basic RNN layer **self.basic_rnn** followed by a fully-connected layer **self.FC**; (2) the **init_hidden** function initializes hidden weights with zero values of size num_layers*batch_size*n_neurons; (3) the **forward** function accepts an input of size n_steps*batch_size*n_inputs (here, **permute** function returns a view of the original tensor with its dimensions permuted). The data flows through the RNN layer and then through the fully-connected layer. The output is of size batch_size*n_output.

```
class ImageRNN(nn.Module):
    def __init__(self, batch_size, n_steps, n_inputs, n_neurons, n_outputs):
        super(ImageRNN, self).__init__()

        self.n_neurons = n_neurons
        self.batch_size = batch_size
        self.n_steps = n_steps
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.basic_rnn = nn.RNN(self.n_inputs, self.n_neurons)
        self.FC = nn.Linear(self.n_neurons, self.n_outputs)

    def init_hidden(self,):
        return (torch.zeros(1, self.batch_size, self.n_neurons))

    def forward(self, X):
        X = X.permute(1, 0, 2)
        self.batch_size = X.size(1)
        self.hidden = self.init_hidden()
        lstm_out, self.hidden = self.basic_rnn(X, self.hidden)
        out = self.FC(self.hidden)
        return out.view(-1, self.n_outputs)
```

e. Before training the model, declare a few help functions. The **torch.device** function tells the program that we want to use the GPU if one is available, otherwise the CPU will be the default device.

   Then we create an instance of the model, ImageRNN, with the proper parameters. The **criterion** represents the function we will use to compute the loss of the model, and we use cross-entropy loss. We use Adam optimizer as our optimization function, with a learning rate of 0.001. The **get_accuracy** function simply computes the accuracy of the model given the log probabilities and target values.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = ImageRNN(BATCH_SIZE, N_STEPS, N_INPUTS, N_NEURONS, N_OUTPUTS)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

def get_accuracy(logit, target, batch_size):
    corrects = (torch.max(logit, 1)[1].view(target.size()).data == target.data).sum()
    accuracy = 100.0 * corrects/batch_size
    return accuracy.item()
```

f. Finally, let's train the model. We will loop over the dataset multiple times. In each training round, we will first set the parameter gradients to zeros, reset hidden states, then do forward propagation, backprop, optimization. The **detach** function detaches the output from the computational graph, and then training accuracy will be calculated with the help of **get_accurcy**.

   Again, this step takes time.

```
for epoch in range(N_EPHOCS):
    train_running_loss = 0.0
    train_acc = 0.0
    model.train()

    for i, data in enumerate(trainloader):
        optimizer.zero_grad()
        model.hidden = model.init_hidden()
        inputs, labels = data
        inputs = inputs.view(-1, 28,28)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_running_loss += loss.detach().item()
        train_acc += get_accuracy(outputs, labels, BATCH_SIZE)

    model.eval()
    print('Epoch:  %d | Loss: %.4f | Train Accuracy: %.2f'
          %(epoch, train_running_loss / i, train_acc/i))
```

g. Lastly, we will test the trained model.

```
test_acc = 0.0
for i, data in enumerate(testloader, 0):
    inputs, labels = data
    inputs = inputs.view(-1, 28, 28)
    outputs = model(inputs)
    test_acc += get_accuracy(outputs, labels, BATCH_SIZE)

print('Test Accuracy: %.2f'%( test_acc/i))
```

This is the end of lab 6. There is NO assignment questions.

Submission:
Finish all the steps in the lab, save it to lab6.ipynb file with all the outputs. And submit on Blackboard via submission link.
**Due: Mar 8th 11:30am EST**

Reference:
https://medium.com/dair-ai/building-rnns-is-fun-with-pytorch-and-google-colab-3903ea9a3a79