Part II: CNN-LSTM for Sequence Prediction

Introduction:

- In this section, you'll extend the CNN from section (1) to a hybrid CNN-LSTM model to predict the future elements of a sequence.
- Instead of providing a single digit image to predict its category, you will be given a sequence of digit images. These sequences follow a pattern in that each consecutive image pair will be shifted by some constant amount. You will design a CNN-LSTM model to recognize those patterns from raw images and predict the future digits.
- For instance, the input and target of the CNN-LSTM model can be the image squence whose categories are shown as below:

  ```
  Input: 1,3,5,7,9,1   Target: 3,5,7  (shifted by 2)
  Input: 2,6,0,4,8,2   Target: 6,0,4  (shifted by 4)
  ```

  where the input and target consists of 5 elements and 3 elements, respectively.
- The training sequence are generated by varying shifts. We also include some unseen shifts in test sequences to validate the model's generalization ability.

Task:

- You need to design the model and complete the training loop with Pytorch.
- You need to achieve 93% averaged Top1 Acc on test data.
- This experiment shares the same dataset with the first section. Once you prepare the data following the first section, you do not need to download extra content.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
# The arguments of the expeirment
class Args:
    def __init__(self):
        # Based on the availablity of GPU, decide whether to run the experiment on cuc
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        # The random seed for the exp.
        self.seed = 1
        # The mini batch size of training and testing data. If you find you machines r
        # or experinece with OOM issue, you can set a smaller batch size
        self.batch_size = 50
```

```
        # The epochs of the exps. The referenced model achieve over 95% test accuracy
        self.epochs = 1
        # The learning rate of the SGD optimizer
        self.lr = 3e-4 #0.1
        # The momentum of SGD optimizer
        self.momentum = 0.5
        # how many iterations to display the training stats
        self.log_interval = 10
        # The height of input image
        self.img_h = 28
        # The width of the input image
        self.img_w = 28
        # The length of input sequence
        self.input_seq_len = 5
        # The lenght of the sequence to predict
        self.target_seq_len = 2
        # The list to sample shift to generate the training sequence.
        self.train_shift_list = [1,2,4,5]
        # The list to sample shift to generate the testing sequence.
        self.test_shift_list = [1,2,3,4,5]
```

```
# pytorch mnist cnn + lstm
# Load necessary library

from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data.dataset import Dataset
from torch.autograd import Variable
import pandas as pd
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
args = Args()
torch.manual_seed(args.seed)
```

```
<torch._C.Generator at 0x7fb46e5811b0>
```

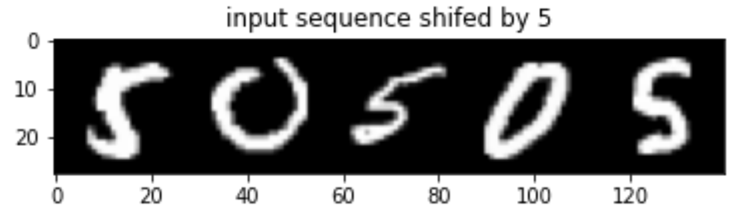
0. The dataloader

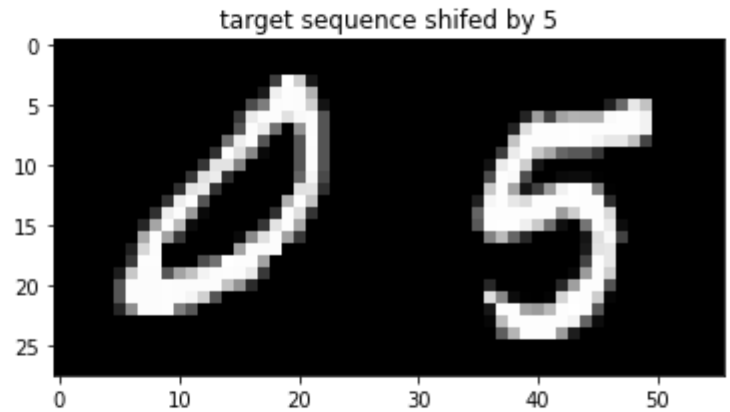
- Load the data from csv file
- Generate the input and target image sequences spaced by constant distance sampled from the predefined shift list

```python
# The dataset to generated training and testing sequence
class MNIST_SEQ_DATASET(Dataset):
    def __init__(self, csv_path, height, width, input_len, output_len, seq_shift, trar
        """
        Custom dataset example for reading data from csv

        Args:
            csv_path (string): path to csv file
            height (int): image height
            width (int): image width
            transform: pytorch transforms for transforms and tensor conversion
        """
        self.data = pd.read_csv(csv_path)
        self.labels = np.asarray(self.data.iloc[:, 0])
        self.height = height
        self.width = width
        self.input_len = input_len
        self.output_len = output_len
        self.transform = transform
        self.seq_shift = seq_shift
        unique_label_array = np.unique(self.labels)
        self.label_data_id_dict = {}
        for unique_label in unique_label_array:
            self.label_data_id_dict[unique_label] = np.where(self.labels == unique_lak

    def get_single_image(self, index):
        single_image_label = self.labels[index]
        # Read each 784 pixels and reshape the 1D array ([784]) to 2D array ([28,28])
        img_as_np = np.asarray(self.data.iloc[index][1:]).reshape(28, 28).astype('uint
        # Convert image from numpy array to PIL image, mode 'L' is for grayscale
        img_as_img = Image.fromarray(img_as_np)
        img_as_img = img_as_img.convert('L')
        # Transform image to tensor
        if self.transform is not None:
            img_as_tensor = self.transform(img_as_img)
        # Return image and the label
        return (img_as_tensor, single_image_label)

    def __getitem__(self, index):
        # Randomly sample the shift from predefined shift list
        seq_shift = np.random.choice(self.seq_shift)
        # Randomly select one category as leading digit
        start_idx = np.random.choice(10)
        # The sequence with following digits
        seq_digit = np.arange(start_idx, start_idx + seq_shift * (self.input_len + sel
        # Modulo opeartion over the digit sequence
        seq_digit = seq_digit % 10
        img_seq = []
        label_seq = []
        # Collect the images for each digit
        for digit in seq_digit:
```

```
            data_id = np.random.choice(self.label_data_id_dict[digit])
            img, label = self.get_single_image(data_id)
            img_seq.append(img)
            label_seq.append(label)
        # Return image and the label
        input_img_seq = img_seq[:self.input_len]
        input_label_seq = label_seq[:self.input_len]
        target_img_seq = img_seq[self.input_len:]
        target_label_seq = label_seq[self.input_len:]
        return torch.stack(input_img_seq), torch.stack(target_img_seq), \
                torch.from_numpy(np.stack(input_label_seq)), \
                torch.from_numpy(np.stack(target_label_seq)), \
                seq_shift

    def __len__(self):
        return len(self.data.index)


# Instantiate the dataset which is then wrapperd by the DataLoader for effective prefe
train_path = '/content/drive/MyDrive/Colab Notebooks/mnist_train.csv'
test_path = '/content/drive/MyDrive/Colab Notebooks/mnist_test.csv'
transformations = transforms.Compose([transforms.ToTensor()])
mnist_train = \
    MNIST_SEQ_DATASET(train_path,
                            args.img_h, args.img_w, args.input_seq_len, args.target_s
                            args.test_shift_list,
                            transformations)

mnist_test = \
    MNIST_SEQ_DATASET(test_path,
                            args.img_h, args.img_w, args.input_seq_len, args.target_s
                            args.test_shift_list,
                            transformations)

mnist_train_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                                       batch_size=args.batch_size,
                                                       shuffle=True)

mnist_test_loader = torch.utils.data.DataLoader(dataset=mnist_test,
                                                       batch_size=args.batch_size,
                                                       shuffle=False)


# Display the input sequence and target sequence
input_img_seq, target_img_seq, input_label_seq, target_label_seq, seq_shift = mnist_tr
img_to_disp = input_img_seq.permute(1,2,0,3).reshape(args.img_h,-1,args.img_w)
input_img_seq = img_to_disp.reshape(args.img_h, -1)
img_to_disp = target_img_seq.permute(1,2,0,3).reshape(args.img_h,-1,args.img_w)
target_img_seq = img_to_disp.reshape(args.img_h, -1)
plt.imshow(input_img_seq,  cmap="gray")
plt.title('input sequence shifed by {}'.format(seq_shift))
```

```
plt.show()
plt.imshow(target_img_seq, cmap="gray")
plt.title('target sequence shifed by {}'.format(seq_shift))
plt.show()
```





1. (TODO) The CNN-LSTM Model [20 points]

- Complete the following section to create a CNN-LSTM model for sequence predicting problem.
- You can borrow the CNN design from previous section as CNN encodes the categorical features of image
- The CNN-LSTM should consist of the three modules:
  - CNN for extracting visual features to a single feature vector
  - LSTM taking as input the sequence of feature vector from CNN and producing the hidden feature to predict the next element
  - A decoder to convert the LSTM prediction to categorical distribution

```
#Attempt 1, please look at both, though I think this one is closer to correct.
class CNN(nn.Module):
    """Custom CNN model to extract visual features from input image"""

    def __init__(self):
        """ Define and instantiate your layers"""
        super(CNN, self).__init__()
        # YOUR CODE HERE
        self.layers = nn.Sequential(
            nn.Conv2d(1, 16, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
```

```
        )

        self.fc = nn.Linear(3136, 32)

    def forward(self, x):
        """
        Run forward pass on input image X

        Args:
            x: torch tensor of input image,
                with shape of [batch_size, 1, img_h, img_w]

        Return:
            out: torch tensor of feature vector computed on input image,
                with shape of [batch_size, latent_dim]

        """
        x = self.layers(x)
        x = x.view(x.size(0), -1)
        output = self.fc(x)

        return output

class CNN_LSTM(nn.Module):
    """ Custom CNN-LSTM model for sequence prediction problem """
    def __init__(self):
        """ Define and instantiate your layers"""
        super(CNN_LSTM, self).__init__()
        # YOUR CODE HERE
        self.hidden_size = 32
        self.num_layers = 1
        self.input_size = 32

        self.cnn = CNN()
        self.cnn = self.cnn.to(args.device)

        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers, batch_
        self.fc = nn.Linear(self.hidden_size, 10)

    def forward(self, x, num_step_to_predict):
        """
        Run forward pass on image squence x and predict the future digitss

        Args:
            x : torch tensor of input image sequence,
                    with shape of [batch_size, input_time_step, 1, img_h, img_w]
            num_step_to_predict: an interger on how many steps to predict.

        Returns:
            output: torch tensor of predicted categorical distribution
                    for the ENTIRE sequence, including input and predicted sequence,
```

```
            with shape of [batch_size, input_time_step + num_step_to_predict,
            Noted the output from i step is the prediction for 1+1 step.

"""
# YOUR CODE HERE
x = x.view(250,1,28,28)
#print(x.shape)
out = self.cnn(x)
new_out = out.view(50,5,32)

#For testing without cnn fc layer (output [250,x,32])...dead end
#test_x = out.view(out.size(0), out.size(1), -1)
#test_x = test_x.permute(0, 2, 1)
#h0 = Variable(torch.zeros(self.num_layers, 250, self.hidden_size)).to(args.de
#c0 = Variable(torch.zeros(self.num_layers, 250, self.hidden_size)).to(args.de

h0 = Variable(torch.zeros(self.num_layers, 50, self.hidden_size)).to(args.devi
c0 = Variable(torch.zeros(self.num_layers, 50, self.hidden_size)).to(args.devi

test_x, (h0, c0) = self.lstm(new_out, (h0, c0))

#For testing without cnn fc layer (output [250,x,32])...dead end
#test_x = test_x[:, -1, :]
#my_tensor = test_x
#test_x = test_x.view(50,5,32)

#For testing without cnn fc layer (output [250,x,32])...dead end
#h0 = Variable(torch.zeros(self.num_layers, 50, self.hidden_size)).to(args.dev
#c0 = Variable(torch.zeros(self.num_layers, 50, self.hidden_size)).to(args.dev
for i in range(num_step_to_predict):
  test_x, (h0, c0) = self.lstm(test_x, (h0, c0))
  save_x = test_x[:, -1, :]

  #For testing without cnn fc layer (output [250,x,32])...dead end
  #my_tensor = torch.cat([my_tensor, save_x], dim=0)

  out = torch.cat([out, save_x], dim=0)

  #to save out only future sequence elements
  #if i == 0:
  #  my_tensor = save_x
  #else:
  #  my_tensor = torch.cat([my_tensor, save_x], dim=0)

#print(int_tensor.shape)
fc_final_out = self.fc(out)
#print(fc_final_out.shape)
fc_final_out = fc_final_out.view(50,7,10)

return fc_final_out
```

```
#Attempt 2
class CNN(nn.Module):
    """Custom CNN model to extract visual features from input image"""

    def __init__(self):
        """ Define and instantiate your layers"""
        super(CNN, self).__init__()
        # YOUR CODE HERE
        self.layers = nn.Sequential(
            nn.Conv2d(1, 16, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )

        self.fc = nn.Linear(3136, 10)

    def forward(self, x):
        """
        Run forward pass on input image X

        Args:
            x: torch tensor of input image,
                with shape of [batch_size, 1, img_h, img_w]

        Return:
            out: torch tensor of feature vector computed on input image,
                with shape of [batch_size, latent_dim]

        """
        # YOUR CODE HERE
        x = self.layers(x)
        x = x.view(x.size(0), -1)
        output = self.fc(x)

        return output

class CNN_LSTM(nn.Module):
    """ Custom CNN-LSTM model for sequence prediction problem """
    def __init__(self):
        """ Define and instantiate your layers"""
        super(CNN_LSTM, self).__init__()
        # YOUR CODE HERE
        self.hidden_size = 10
        self.num_layers = 1
        self.input_size = 10

        self.cnn = CNN()
        self.cnn = self.cnn.to(args.device)

        self.lstm = nn.LSTM(self.input_size, self.hidden_size, self.num_layers, batch_
        self.fc = nn.Linear(self.hidden_size, 10)
```

```
def forward(self, x, num_step_to_predict):
    """
    Run forward pass on image squence x and predict the future digitss

    Args:
        x : torch tensor of input image sequence,
                with shape of [batch_size, input_time_step, 1, img_h, img_w]
        num_step_to_predict: an interger on how many steps to predict.

    Returns:
        output: torch tensor of predicted categorical distribution
                for the ENTIRE sequence, including input and predicted sequence,
                with shape of [batch_size, input_time_step + num_step_to_predict,
                Noted the output from i step is the prediction for 1+1 step.

    """
    # YOUR CODE HERE
    x = torch.transpose(x, 0, 1)

    for i in range(args.input_seq_len):
      cnn_out = self.cnn(x[i])
      cnn_out = cnn_out.unsqueeze(1)
      if i == 1:
        my_tensor = torch.cat([last_x, cnn_out], dim=1)
      elif i > 1:
        my_tensor = torch.cat([my_tensor, cnn_out], dim=1)
      else:
        last_x = cnn_out

    out = my_tensor
    h0 = torch.zeros(self.num_layers, args.batch_size, self.hidden_size).to(args.c
    c0 = torch.zeros(self.num_layers, args.batch_size, self.hidden_size).to(args.c
    for i in range(num_step_to_predict):
      out, (h0, c0) = self.lstm(out, (h0.detach(), c0.detach()))
      save_out = self.fc(out[:, -1, :])
      save_out = save_out.unsqueeze(1)
      my_tensor = torch.cat([my_tensor, save_out], dim=1)

    return my_tensor
```

2. (TODO) The Training Loop [15 points]

- Instantiate the model and optimizer
- Select proper loss function for this task
- Complete the training loop

```
model = CNN_LSTM()
model = model.to(args.device)
optimizer = optim.Adam(model.parameters(), lr=args.lr)
loss_func = nn.NLLLoss()

def train(epoch):
    model.train()
    for batch_idx, (input_img_seq, target_img_seq, input_label_seq, target_label_seq,
        # batch_size * input_seq_len * 1 * img_h * img_w
        input_img_seq = input_img_seq.to(args.device)
        #print(input_img_seq.shape)
        # batch_size * input_seq_len
        input_label_seq = input_label_seq.to(args.device)
        #print(input_label_seq)
        # batch_size * output_seq_len * 1 * img_h * img_w
        target_img_seq = target_img_seq.to(args.device)
        # batch_size * output_seq_len
        target_label_seq = target_label_seq.to(args.device)

        # YOUR CODE HERE
        outputs = model(input_img_seq, args.target_seq_len)

        labels = torch.cat([input_label_seq, target_label_seq], dim=1)

        outputs = torch.transpose(outputs, 2, 1)
        loss = loss_func(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch_idx % 100 == 0:
            correct = 0
            total = 0

            outputs = model(input_img_seq,2)
            predicted = torch.max(outputs.data, 2)

            total += target_label_seq.size(0)

            p = predicted.indices.cpu().detach().numpy()
            l = labels.cpu().detach().numpy()
            for i in range(len(p)):
                #print(p[i],l[i])
                if (p[i] == l[i]).all():
                    correct = correct + 1

            accuracy = 100 * correct / total
            print('batch_idx: {}. accuracy: {}'.format(batch_idx, accuracy))
```

```
for epoch in range(args.epochs):
    train(epoch)
```

```
batch_idx: 0. accuracy: 0.0
batch_idx: 100. accuracy: 0.0
batch_idx: 200. accuracy: 0.0
batch_idx: 300. accuracy: 0.0
batch_idx: 400. accuracy: 0.0
batch_idx: 500. accuracy: 0.0
batch_idx: 600. accuracy: 0.0
batch_idx: 700. accuracy: 0.0
batch_idx: 800. accuracy: 0.0
batch_idx: 900. accuracy: 0.0
batch_idx: 1000. accuracy: 0.0
```

3. Test

- Once your model achieve descent training accuracy, you can run test to validate your model
- You should achieve at least 93% Top1 Acc to get full credit.

```
def test():
    model.eval()
    top1_acc_dict = {test_shift:{'sum_acc':0, 'count':0} for test_shift in args.test_s
    top5_acc_dict = {test_shift:{'sum_acc':0, 'count':0} for test_shift in args.test_s
    for batch_idx, (input_img_seq, target_img_seq, input_label_seq, target_label_seq,
        batch_size = input_img_seq.shape[0]
        # batch_size * input_seq_len * 1 * img_h * img_w
        input_img_seq = input_img_seq.to(args.device)
        # batch_size * input_seq_len
        input_label_seq = input_label_seq.to(args.device)
        # batch_size * output_seq_len * 1 * img_h * img_w
        target_img_seq = target_img_seq.to(args.device)
        # batch_size * output_seq_len
        target_label_seq = target_label_seq.to(args.device)

        total_pred = model(input_img_seq, args.target_seq_len)
        pred = total_pred[:,:-1][:,-1 * args.target_seq_len:].reshape(-1,10)

        _, top_index = pred.topk(5, dim = -1)
        correct_pred = top_index == target_label_seq.reshape(-1)[:,None]
        top1_acc = correct_pred[:,0].float().reshape(batch_size, -1) * 100
        top5_acc = correct_pred[:,:5].sum(dim = -1).float().reshape(batch_size, -1) *
        for seq_shift_ele in torch.unique(seq_shift):
            top1_acc_val = top1_acc[torch.where(seq_shift == seq_shift_ele)[0]].mean(
            top1_acc_count = torch.where(seq_shift == seq_shift_ele)[0].shape[0]
            top1_acc_dict[seq_shift_ele.item()]['sum_acc'] += top1_acc_val.item()
            top1_acc_dict[seq_shift_ele.item()]['count'] += top1_acc_count

            top5_acc_val = top5_acc[torch.where(seq_shift == seq_shift_ele)[0]].mean(
            top5_acc_count = torch.where(seq_shift == seq_shift_ele)[0].shape[0]
            top5_acc_dict[seq_shift_ele.item()]['sum_acc'] += top5_acc_val.item()
```

```
            top5_acc_dict[seq_shift_ele.item()]['count'] += top5_acc_count

        total_top1_acc = np.mean(np.stack([val['sum_acc'] / (val['count'] + 1e-5) for
        total_top5_acc = np.mean(np.stack([val['sum_acc'] / (val['count'] + 1e-5) for

        if batch_idx % args.log_interval == 0:
            print('Test: [{}/{} ({:.0f}%)] Top1 Acc: {:1f}, Top5 Acc: {:1f}'.format(
                batch_idx * input_img_seq.shape[0], len(mnist_test_loader.dataset),
                100. * batch_idx * input_img_seq.shape[0] / len(mnist_test_loader.data

    top1_acc_each_shift = np.stack([val['sum_acc'] / (val['count'] + 1e-5) for key, va
    top5_acc_each_shift = np.stack([val['sum_acc'] / (val['count'] + 1e-5) for key, va
    for idx, (key, _) in enumerate(top1_acc_dict.items()):
        print('Shift {}, Test Top1 Acc: {:1f}, Test Top5 Acc: {:1f}'.format(key, top1_
test()
```

```
Test: [0/10000 (0%)] Top1 Acc: 8.833325, Top5 Acc: 44.999954
Test: [500/10000 (5%)] Top1 Acc: 10.201686, Top5 Acc: 51.382256
Test: [1000/10000 (10%)] Top1 Acc: 10.189020, Top5 Acc: 51.564803
Test: [1500/10000 (15%)] Top1 Acc: 10.461947, Top5 Acc: 51.576704
Test: [2000/10000 (20%)] Top1 Acc: 10.240449, Top5 Acc: 50.720489
Test: [2500/10000 (25%)] Top1 Acc: 10.364292, Top5 Acc: 51.097089
Test: [3000/10000 (30%)] Top1 Acc: 10.351687, Top5 Acc: 50.576751
Test: [3500/10000 (35%)] Top1 Acc: 10.390537, Top5 Acc: 50.290246
Test: [4000/10000 (40%)] Top1 Acc: 10.454577, Top5 Acc: 50.376070
Test: [4500/10000 (45%)] Top1 Acc: 10.383560, Top5 Acc: 50.442450
Test: [5000/10000 (50%)] Top1 Acc: 10.393518, Top5 Acc: 50.530966
Test: [5500/10000 (55%)] Top1 Acc: 10.431199, Top5 Acc: 50.485201
Test: [6000/10000 (60%)] Top1 Acc: 10.347662, Top5 Acc: 50.318645
Test: [6500/10000 (65%)] Top1 Acc: 10.282665, Top5 Acc: 50.233260
Test: [7000/10000 (70%)] Top1 Acc: 10.220241, Top5 Acc: 50.215091
Test: [7500/10000 (75%)] Top1 Acc: 10.195473, Top5 Acc: 50.158270
Test: [8000/10000 (80%)] Top1 Acc: 10.203756, Top5 Acc: 50.204876
Test: [8500/10000 (85%)] Top1 Acc: 10.226572, Top5 Acc: 50.222269
Test: [9000/10000 (90%)] Top1 Acc: 10.220708, Top5 Acc: 50.149175
Test: [9500/10000 (95%)] Top1 Acc: 10.293320, Top5 Acc: 50.142142
Shift 1, Test Top1 Acc: 10.259740, Test Top5 Acc: 10.259740
Shift 2, Test Top1 Acc: 10.040775, Test Top5 Acc: 10.040775
Shift 3, Test Top1 Acc: 10.702179, Test Top5 Acc: 10.702179
Shift 4, Test Top1 Acc: 10.371820, Test Top5 Acc: 10.371820
Shift 5, Test Top1 Acc: 10.079840, Test Top5 Acc: 10.079840
```