In [ ]:

```
import torch
import torchvision
from torchvision import utils
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data import Dataset, Subset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import sys
import numpy as np
import os
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
import seaborn as sn
import torch.optim as optim

DEVICE_DEFAULT=torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Utility Functions

Code taken from tutorial

In [ ]:

```
def pbar(p=0, msg="", bar_len=20):
    sys.stdout.write("\033[K")
    sys.stdout.write("\x1b[2K" + "\r")
    block = int(round(bar_len * p))
    text = "Progress: [{}] {}% {}".format(
        "\x1b[32m" + "=" * (block - 1) + ">" + "\033[0m" + "-" * (bar_len - block),
        round(p * 100, 2),
        msg,
    )
    print(text, end="\r")
    if p == 1:
        print()

class AvgMeter:
    def __init__(self):
```

```python
        self.reset()

    def reset(self):
        self.metrics = {}

    def add(self, batch_metrics):
        for key, value in batch_metrics.items():
            if key in self.metrics.items():
                self.metrics[key].append(value)
            else:
                self.metrics[key] = [value]

    def get(self):
        return {key: np.mean(value) for key, value in self.metrics.items()}

    def msg(self):
        avg_metrics = {key: np.mean(value) for key, value in self.metrics.items()}
        return "".join(["[{}] {:.5f} ".format(key, value) for key, value in avg_metrics.items()])

def train(model, optim, lr_sched=None, epochs=200, device=torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    best_acc = 0
    for epoch in range(epochs):
        model.train()
        metric_meter.reset()
        for indx, (img, target) in enumerate(train_loader):
            img = img.to(device)
            target = target.to(device)

            optim.zero_grad()
            out = model.forward(img)
            loss = criterion(out, target)
            loss.backward()
            optim.step()

            metric_meter.add({"train loss": loss.item()})
            pbar(indx / len(train_loader), msg=metric_meter.msg())
        pbar(1, msg=metric_meter.msg())
        train_loss_for_plot.append(metric_meter.get()["train loss"])

        model.eval()
        metric_meter.reset()
        for indx, (img, target) in enumerate(val_loader):
            img = img.to(device)
            target = target.to(device)
            out = model.forward(img)
            loss = criterion(out, target)
            acc = (out.argmax(1) == target).sum().item() * (100 / img.shape[0])

            metric_meter.add({"val loss": loss.item(), "val acc": acc})
            pbar(indx / len(val_loader), msg=metric_meter.msg())
        pbar(1, msg=metric_meter.msg())

        val_metrics = metric_meter.get()
        val_loss_for_plot.append(val_metrics["val loss"])
        val_acc_for_plot.append(max(val_metrics["val acc"], best_acc))
        if val_metrics["val acc"] > best_acc:
            print(
              "\x1b[33m"
              + f"val acc improved from {round(best_acc, 5)} to {round(val_metrics['val acc'], 5)}"
              + "\033[0m"
            )
            best_acc = val_metrics['val acc']
#             torch.save(model.state_dict(), os.path.join(out_dir, "best.ckpt"))
    lr_sched.step()
```

Data Loading

In [ ]:

```
data_train = datasets.MNIST('~/mnist_data', train=True, download=True, transform=transforms.ToTensor())
data_test = datasets.MNIST('~/mnist_data', train=False, download=True, transform=transforms.ToTensor())
```

In [ ]:

```
# Split train data into train(50000) and validation(10000)

train_indices, val_indices, _, _ = train_test_split(
    range(len(data_train)),
    data_train.targets,
    stratify=data_train.targets, # Make sure that the percentage of each class is same in both train & val
    test_size=10000,
)
```

```
train_split = Subset(data_train, train_indices)
val_split = Subset(data_train, val_indices)
```

In [ ]:

```
print(f'Number of training examples: {len(train_split)}')
print(f'Number of validation examples: {len(val_split)}')
print(f'Number of testing examples: {len(data_test)}')
```

In [ ]:

```
BATCH_SIZE = 64
train_loader = DataLoader(train_split, batch_size=BATCH_SIZE)
val_loader = DataLoader(val_split, batch_size=BATCH_SIZE)
test_loader = DataLoader(data_test, batch_size=BATCH_SIZE)
```

Part 1: MNIST Classification using RNN

In [ ]:

```
class RNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, bidirectional, output_dim):

        super().__init__()

        self.rnn = nn.RNN(input_size = input_dim,
                          hidden_size = hidden_dim,
                          num_layers = num_layers,
                          batch_first = True,
                          bidirectional = bidirectional
                         )

        D = (2 if bidirectional else 1)

        self.fc = nn.Linear(D * num_layers * hidden_dim, output_dim)

    def forward(self, batch):

        assert batch.dim() == 4

        output, hidden = self.rnn(batch.squeeze(1))

        # D = 2 if bidirectional, else D = 1
        # output = [batch size, seq length, D * hidden_dim]
        # hidden = [D * num_layers, batch size, hidden_dim]
        flat_hidden = torch.cat([hidden[i,:,:] for i in range(hidden.shape[0])], dim = 1)

        output = self.fc(flat_hidden)

        return output

class LSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, bidirectional, output_dim):

        super().__init__()

        self.lstm = nn.LSTM(input_size = input_dim,
                          hidden_size = hidden_dim,
                          num_layers = num_layers,
                          batch_first = True,
                          bidirectional = bidirectional
                         )

        D = (2 if bidirectional else 1)

        self.fc = nn.Linear(D * num_layers * hidden_dim, output_dim)

    def forward(self, batch):

        assert batch.dim() == 4

        output, (hidden, cell) = self.lstm(batch.squeeze(1))

        # D = 2 if bidirectional, else D = 1
        # output = [batch size, seq length, D * hidden_dim]
        # hidden = [D * num_layers, batch size, hidden_dim]

        flat_hidden = torch.cat([hidden[i,:,:] for i in range(hidden.shape[0])], dim = 1)

        output = self.fc(flat_hidden)

        return output
```

In [ ]:

```
INPUT_DIM = 28
HIDDEN_DIM = 256
OUTPUT_DIM = 10
```

```
NUM_LAYERS = 1
BIDIRECTIONAL = False
EPOCHS = 30

model = RNN(INPUT_DIM, HIDDEN_DIM, NUM_LAYERS, BIDIRECTIONAL, OUTPUT_DIM)

# optim = torch.optim.SGD(model.parameters(), lr=10**-3, momentum=0.9, weight_decay=5e-4)
optim = torch.optim.Adam(model.parameters(), lr=10**-4, weight_decay=1e-6)
lr_sched = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=EPOCHS)
criterion = nn.CrossEntropyLoss()
metric_meter = AvgMeter()

out_dir = "Part1"
os.makedirs(out_dir, exist_ok=True)

train_loss_for_plot = []
val_loss_for_plot = []
val_acc_for_plot = []

train(model, optim, lr_sched, epochs=EPOCHS, criterion=criterion, metric_meter=metric_meter, out_dir=out_dir)
# After this the model will be saved in out_dir
```

In [ ]:

```
plt.figure(figsize=(15, 3))
plt.subplot(1, 3, 1)
plt.plot(train_loss_for_plot)
plt.xlabel("Epoch #")
plt.ylabel("Train Loss")
plt.title("Train Loss vs. Epochs")

plt.subplot(1, 3, 2)
plt.plot(val_loss_for_plot)
plt.xlabel("Epoch #")
plt.ylabel("Val Loss")
plt.title("Validation Loss vs. Epochs")

plt.subplot(1, 3, 3)
plt.plot(val_acc_for_plot)
plt.xlabel("Epoch #")
plt.ylabel("Val Accuracy")
plt.ylim([0,105])
plt.title("Validation Accuracy vs. Epochs")
plt.show()
```

In [ ]:

```
model.eval()
metric_meter.reset()
for indx, (img, target) in enumerate(test_loader):
    img = img.to(DEVICE_DEFAULT)
    target = target.to(DEVICE_DEFAULT)
    out = model.forward(img)
    acc = (out.argmax(1) == target).sum().item() * (100 / img.shape[0])
    metric_meter.add({"test acc": acc})

print("Test Accuracy", metric_meter.get()["test acc"])
```

In [ ]:

```
model.eval()
plt.figure(figsize = (15, 7))
for idx in range(10):
    rand_sample = np.random.randint(len(data_test))
    img = data_test[rand_sample][0][0]
    act = str(data_test[rand_sample][1])
    pred = str(model.forward(img.view(1,1,28,28).to(DEVICE_DEFAULT)).argmax(1).item())
    plt.subplot(2, 5, idx+1)
    plt.imshow(img, cmap='gray'); plt.axis('off'); plt.ioff()
    plt.title('True: ' + act + '\nPrediction: ' + pred, fontsize = 20, fontweight='bold', color = 'blue')
plt.show()
```

In [ ]:

```
from PIL import Image
import glob
image_list = []
transform = transforms.Compose([
    transforms.PILToTensor(),
    transforms.Grayscale(1)
])
for filename in glob.glob('Cropped/*.jpeg'): #assuming gif
    im=Image.open(filename)
    image_list.append(transform(im.resize((28,28))).type(torch.float))

model.eval()
plt.clf()
plt.figure(figsize = (6, 6))
for idx in range(len(image_list)):
    img = image_list[idx][0]
    pred = str(model.forward(img.view(1,1,28,28).to(DEVICE_DEFAULT)).argmax(1).item())
    plt.subplot(4, 4, idx+1)
    plt.imshow(img, cmap='gray'); plt.axis('off'); plt.ioff()
    plt.title('Prediction: ' + pred, fontsize = 10, fontweight='bold', color = 'blue')
plt.show()
```

In [ ]:

```
from PIL import Image
import glob
image_list = []
transform = transforms.Compose([
    transforms.PILToTensor(),
    transforms.Grayscale(1)
])
for filename in glob.glob('MyWriting/*.jpg'): #assuming gif
    im=Image.open(filename)
    image_list.append(transform(im.resize((28,28))).type(torch.float))

model.eval()
plt.clf()
plt.figure(figsize = (6, 6))
for idx in range(len(image_list)):
    img = image_list[idx][0]
    pred = str(model.forward(img.view(1,1,28,28).to(DEVICE_DEFAULT)).argmax(1).item())
    plt.subplot(4, 4, idx+1)
    plt.imshow(img, cmap='gray'); plt.axis('off'); plt.ioff()
    plt.title('Prediction: ' + pred, fontsize = 10, fontweight='bold', color = 'blue')
plt.show()
```