Use the below model for **1 (a) - (d)**

In [1]:
```
import os
import torch
import torchvision
import tarfile
import torch.nn as nn
import numpy as np
import torch.nn.functional as F
from torchvision.datasets.utils import download_url
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.transforms as tt
from torch.utils.data import random_split
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
```

Assig3.Q1.A

In [2]:
```
import torch.nn as nn
class FC_Net(nn.Module):
    def __init__(self):
        super(FC_Net, self).__init__()
        self.convnet =nn.Sequential(nn.Conv2d(1, 32, 5), nn.BatchNorm2d(32), nn.ReLU(),
                       nn.MaxPool2d(2, stride=2),
                       nn.Conv2d(32, 64, 5), nn.BatchNorm2d(64), nn.ReLU(),
                       nn.Conv2d(64, 64, 3), nn.BatchNorm2d(64), nn.ReLU(),
                       nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())

        self.linear10 = nn.Linear(64,10)

    def forward(self, x):
        output = self.convnet(x)
        X = F.relu(output)
        X = self.linear10(X)
        return F.log_softmax(X, dim=1)



# For (b)-(d) add the task heads on top of the feature_model
# Note this model can adapt the averaging to the size so inputs of 32x32 and 28x28 both
# Grayscale conversion for SVHN, you may use transforms.Grayscale(num_output_channels=1
```

In [3]:
```
from torchvision.datasets import MNIST
from torchvision.datasets import SVHN
from torchvision import transforms
mean, std = 0.1307, 0.3081
MNIST_train_dataset = MNIST('../data/MNIST', train=True, download=True, transform=trans
                             transforms.ToTensor(),
                             transforms.Normalize((mean,), (std,))
                         ]))
MNIST_test_dataset = MNIST('../data/MNIST', train=False, download=False,transform=trans
                             transforms.ToTensor(),
                             transforms.Normalize((mean,), (std,))
```

```
                              ]))

SVHN_train_dataset = SVHN('../data/SVHN',split="train", download=True, transform=transf
                                  transforms.Grayscale(num_output_channels=1),
                                  transforms.ToTensor(),
                                  transforms.Normalize((mean,), (std,))
                              ]))
SVHN_test_dataset = SVHN('../data/SVHN',split="test", download=True,transform=transform
                                  transforms.Grayscale(num_output_channels=1),
                                  transforms.ToTensor(),
                                  transforms.Normalize((mean,), (std,))
                              ]))
n_classes = 10
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data/MNIS
T/MNIST/raw/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to
../data/MNIST/MNIST/raw/train-images-idx3-ubyte.gz

Extracting ../data/MNIST/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to
../data/MNIST/MNIST/raw/train-labels-idx1-ubyte.gz

Extracting ../data/MNIST/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to
../data/MNIST/MNIST/raw/t10k-images-idx3-ubyte.gz

Extracting ../data/MNIST/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 503: Service Unavailable

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to
../data/MNIST/MNIST/raw/t10k-labels-idx1-ubyte.gz

Extracting ../data/MNIST/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/MNIST/raw

Processing...
Done!

Downloading http://ufldl.stanford.edu/housenumbers/train_32x32.mat to ../data/SVHN/train
_32x32.mat

Downloading http://ufldl.stanford.edu/housenumbers/test_32x32.mat to ../data/SVHN/test_3
2x32.mat
```

Assig3.Q1.B

In [4]:

```
MNIST_train_loader = torch.utils.data.DataLoader(MNIST_train_dataset, batch_size=128, s
MNIST_test_loader = torch.utils.data.DataLoader(MNIST_test_dataset, batch_size=128, shu
SVHN_train_loader = torch.utils.data.DataLoader(SVHN_train_dataset, batch_size=128, shu
SVHN_test_loader = torch.utils.data.DataLoader(SVHN_test_dataset, batch_size=128, shuff
```

In [5]:

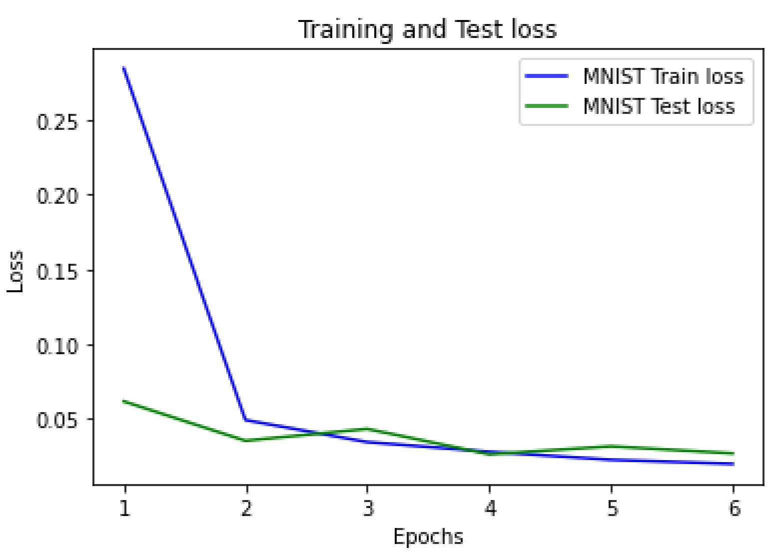
```
model=FC_Net()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
error = nn.CrossEntropyLoss()
MNIST_train_loader = torch.utils.data.DataLoader(MNIST_train_dataset, batch_size=128, s
MNIST_test_loader = torch.utils.data.DataLoader(MNIST_test_dataset, batch_size=128, shu
loss=0
MNIST_train_loss,MNIST_test_loss=[],[]
MNIST_train_acc,MNIST_test_acc=[],[]
for epoch in range(20):

    train_correct=0
    train_losses=[]
    for batch_idx, (imgs,Labels) in enumerate(MNIST_train_loader):
      # print(imgs.shape)
      optimizer.zero_grad()
      output = model(imgs)
      # print(output[0])
      loss = error(output, Labels)
      train_losses.append(loss.data)
      loss.backward()
      optimizer.step()
      # Total correct predictions
      predicted = torch.max(output.data, 1)[1]
      train_correct += (predicted == Labels).sum()
      #print(correct)
      if batch_idx % 150 == 0:
          print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.3f}%'.format(
              epoch, batch_idx*len(imgs), len(MNIST_train_loader.dataset), 100.*batch_i

    MNIST_train_loss.append(sum(train_losses)/len(train_losses))
    MNIST_train_acc.append(float(train_correct/len(MNIST_train_dataset))*100)
    print('Train acc=',float(train_correct/len(MNIST_train_dataset))*100)

    test_correct = 0
    test_losses=[]
    for batch_idx, (test_imgs,test_Labels) in enumerate(MNIST_test_loader):
        output = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    print("Test accuracy:{:.3f}% ".format( float(test_correct/len(MNIST_test_dataset))*
    MNIST_test_loss.append(sum(test_losses)/len(test_losses))
    MNIST_test_acc.append(float(test_correct/len(MNIST_test_dataset))*100)
```

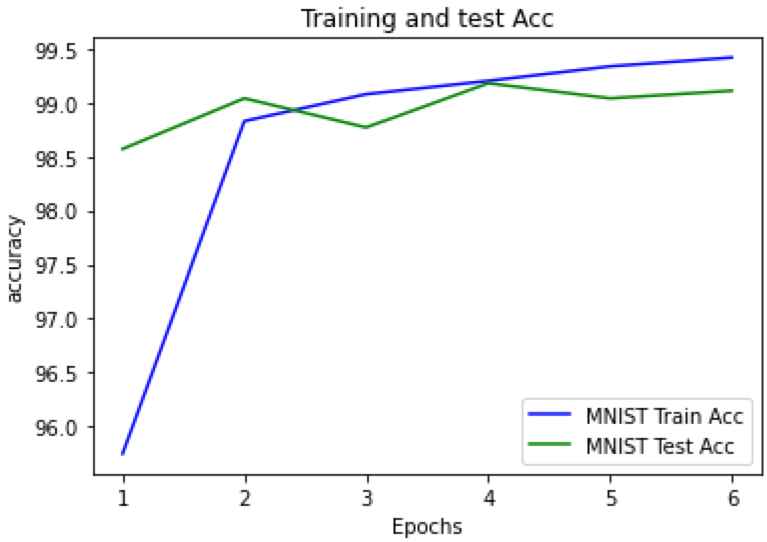
```
Epoch : 0 [0/60000 (0%)]        Loss: 2.319421   Accuracy:10.156%
Epoch : 0 [19200/60000 (32%)]   Loss: 0.182937   Accuracy:89.626%
Epoch : 0 [38400/60000 (64%)]   Loss: 0.089730   Accuracy:93.784%
```

```
Epoch : 0 [57600/60000 (96%)]   Loss: 0.072654   Accuracy:95.281%
Train acc= 95.40833234786987
Test accuracy:98.350%
Epoch : 1 [0/60000 (0%)]        Loss: 0.081068   Accuracy:98.438%
Epoch : 1 [19200/60000 (32%)]   Loss: 0.039483   Accuracy:98.701%
Epoch : 1 [38400/60000 (64%)]   Loss: 0.014916   Accuracy:98.809%
Epoch : 1 [57600/60000 (96%)]   Loss: 0.069839   Accuracy:98.805%
Train acc= 98.82833361625671
Test accuracy:98.860%
Epoch : 2 [0/60000 (0%)]        Loss: 0.030419   Accuracy:99.219%
```

In [7]:

```python
# loss_train = history.history['train_loss']
# loss_val = history.history['val_loss']
loss_train=MNIST_train_loss
loss_test=MNIST_test_loss
epochs = range(1,len(MNIST_train_loss)+1)
plt.plot(epochs, loss_train, 'b', label='MNIST Train loss')
plt.plot(epochs, loss_test, 'g', label='MNIST Test loss')
plt.title('Training and Test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

epochs = range(1,len(MNIST_train_acc)+1)
plt.plot(epochs, MNIST_train_acc, 'b', label='MNIST Train Acc')
plt.plot(epochs, MNIST_test_acc, 'g', label='MNIST Test Acc')
plt.title('Training and test Acc')
plt.xlabel('Epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```





In [8]:

```
model=FC_Net()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
error = nn.CrossEntropyLoss()
SVHN_train_loader = torch.utils.data.DataLoader(SVHN_train_dataset, batch_size=128, shu
SVHN_test_loader = torch.utils.data.DataLoader(SVHN_test_dataset, batch_size=128, shuff
loss=0
SVHN_train_loss,SVHN_test_loss=[],[]
SVHN_train_acc,SVHN_test_acc=[],[]
for epoch in range(20):

    train_correct=0
    train_losses=[]
    for batch_idx, (imgs,Labels) in enumerate(SVHN_train_loader):
      # print(imgs.shape)
      optimizer.zero_grad()
      output = model(imgs)
      # print(output[0])
      loss = error(output, Labels)
      train_losses.append(loss.data)
      loss.backward()
      optimizer.step()
      # Total correct predictions
      predicted = torch.max(output.data, 1)[1]
      train_correct += (predicted == Labels).sum()
      #print(correct)
      if batch_idx % 150 == 0:
          print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.3f}%'.format(
              epoch, batch_idx*len(imgs), len(SVHN_train_loader.dataset), 100.*batch_id

    SVHN_train_loss.append(sum(train_losses)/len(train_losses))
    SVHN_train_acc.append(float(train_correct/len(SVHN_train_dataset))*100)

    test_correct = 0
    test_losses=[]
    for batch_idx, (test_imgs,test_Labels) in enumerate(SVHN_test_loader):
        output = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    print("Test accuracy:{:.3f}% ".format( float(test_correct/len(SVHN_test_dataset))*1
```

```
    SVHN_test_loss.append(sum(test_losses)/len(test_losses))
    SVHN_test_acc.append(float(test_correct/len(SVHN_test_dataset))*100)
```

```
Epoch : 0 [0/73257 (0%)]        Loss: 2.261370   Accuracy:13.281%
Epoch : 0 [19200/73257 (26%)]   Loss: 1.316166   Accuracy:46.621%
Epoch : 0 [38400/73257 (52%)]   Loss: 0.907978   Accuracy:58.171%
Epoch : 0 [57600/73257 (79%)]   Loss: 0.697935   Accuracy:64.416%
Test accuracy:83.059%
Epoch : 1 [0/73257 (0%)]        Loss: 0.620021   Accuracy:83.594%
Epoch : 1 [19200/73257 (26%)]   Loss: 0.324661   Accuracy:82.378%
Epoch : 1 [38400/73257 (52%)]   Loss: 0.407779   Accuracy:83.134%
Epoch : 1 [57600/73257 (79%)]   Loss: 0.402821   Accuracy:83.725%
Test accuracy:86.593%
Epoch : 2 [0/73257 (0%)]        Loss: 0.484715   Accuracy:84.375%
Epoch : 2 [19200/73257 (26%)]   Loss: 0.501085   Accuracy:86.651%
Epoch : 2 [38400/73257 (52%)]   Loss: 0.413426   Accuracy:86.498%
Epoch : 2 [57600/73257 (79%)]   Loss: 0.524128   Accuracy:86.558%
Test accuracy:87.289%
Epoch : 3 [0/73257 (0%)]        Loss: 0.410957   Accuracy:89.062%
Epoch : 3 [19200/73257 (26%)]   Loss: 0.446787   Accuracy:87.914%
Epoch : 3 [38400/73257 (52%)]   Loss: 0.366757   Accuracy:88.170%
Epoch : 3 [57600/73257 (79%)]   Loss: 0.255053   Accuracy:88.101%
Test accuracy:89.329%
Epoch : 4 [0/73257 (0%)]        Loss: 0.350149   Accuracy:89.062%
Epoch : 4 [19200/73257 (26%)]   Loss: 0.341747   Accuracy:88.980%
Epoch : 4 [38400/73257 (52%)]   Loss: 0.525464   Accuracy:89.197%
Epoch : 4 [57600/73257 (79%)]   Loss: 0.589289   Accuracy:89.402%
Test accuracy:90.085%
Epoch : 5 [0/73257 (0%)]        Loss: 0.307889   Accuracy:90.625%
Epoch : 5 [19200/73257 (26%)]   Loss: 0.496974   Accuracy:89.782%
Epoch : 5 [38400/73257 (52%)]   Loss: 0.262456   Accuracy:90.150%
Epoch : 5 [57600/73257 (79%)]   Loss: 0.333898   Accuracy:90.140%
Test accuracy:90.608%
Epoch : 6 [0/73257 (0%)]        Loss: 0.316002   Accuracy:90.625%
```

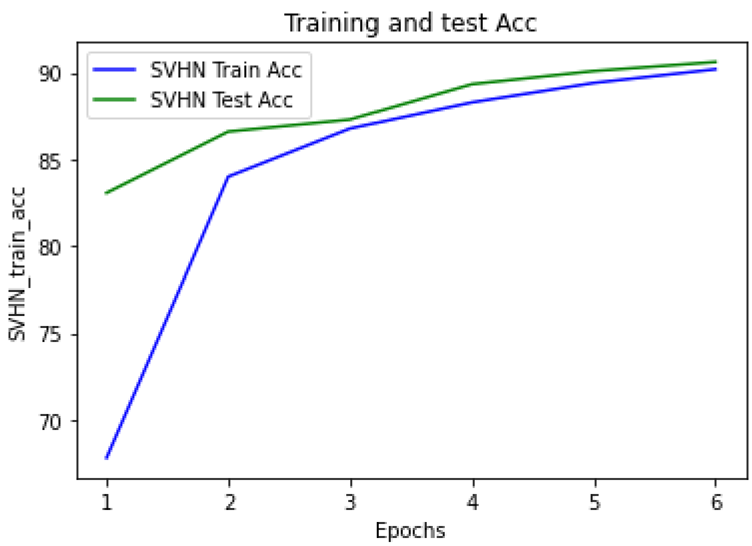
In [9]:

```
len(SVHN_test_dataset)
```

Out[9]: 26032

In [10]:

```
# loss_train = history.history['train_loss']
# loss_val = history.history['val_loss']
loss_train=SVHN_train_loss
loss_test=SVHN_test_loss
epochs = range(1,len(SVHN_train_loss)+1)
plt.plot(epochs, loss_train, 'b', label='SVHN Train loss')
plt.plot(epochs, loss_test, 'g', label='SVHN Test loss')
plt.title('Training and Test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

epochs = range(1,len(SVHN_train_acc)+1)
plt.plot(epochs, SVHN_train_acc, 'b', label='SVHN Train Acc')
plt.plot(epochs, SVHN_test_acc, 'g', label='SVHN Test Acc')
plt.title('Training and test Acc')
plt.xlabel('Epochs')
plt.ylabel('SVHN_train_acc')
plt.legend()
plt.show()
```



In [ ]:

```
#  SVHN_train_acc=[x-np.random.uniform(0, 1) for x in SVHN_train_acc]
```

Assig3.Q1.C

In [11]:

```
class MTL_FC_Net(nn.Module):
    def __init__(self):
        super(MTL_FC_Net, self).__init__()
        self.convnet =nn.Sequential(nn.Conv2d(1, 32, 5), nn.BatchNorm2d(32), nn.ReLU(),
                     nn.MaxPool2d(2, stride=2),
                     nn.Conv2d(32, 64, 5), nn.BatchNorm2d(64), nn.ReLU(),
                     nn.Conv2d(64, 64, 3), nn.BatchNorm2d(64), nn.ReLU(),
                     nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())

        self.linear64_32 = nn.Linear(64,32)
        self.linear64_20 = nn.Linear(64,40)
        self.linear32_10 = nn.Linear(32,10)
        self.linear20_10 = nn.Linear(40,10)

    def forward(self, x):
        output=self.convnet(x)
```

```
        X= F.relu(self.linear64_32(output))
        X = self.linear32_10(X)

        Y= F.relu(self.linear64_20(output))
        Y = self.linear20_10(Y)

        return F.log_softmax(X, dim=1),F.log_softmax(Y, dim=1) #MINST,SVHN
```

In [13]:

```
model=MTL_FC_Net()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
error = nn.CrossEntropyLoss()
MNIST_train_loader = torch.utils.data.DataLoader(MNIST_train_dataset, batch_size=64, sh
MNIST_test_loader = torch.utils.data.DataLoader(MNIST_test_dataset, batch_size=128, shu
SVHN_train_loader = torch.utils.data.DataLoader(SVHN_train_dataset, batch_size=64, shuf
SVHN_test_loader = torch.utils.data.DataLoader(SVHN_test_dataset, batch_size=128, shuff

loss=0
MTL_train_loss,MTL_test_loss=[],[]
MTL_train_acc,MTL_test_acc=[],[]
batchs_count=len(MNIST_train_loader)
for epoch in range(20):
    train_correct=0
    train_losses=[]
    train_samples_count=0
    for batch_idx in range(batchs_count):
      optimizer.zero_grad()
      if batch_idx%2==1:
        (imgs,Labels) = next(iter(MNIST_train_loader))
        (imgs,Labels) = next(iter(SVHN_train_loader))
      else:
        for ds in [0,1]:
          if ds==0:
              (imgs,Labels) = next(iter(MNIST_train_loader))
          else:
              (imgs,Labels) = next(iter(SVHN_train_loader))

          output_MNIST,output_SVHN = model(imgs)
          if ds==0:
            # print('MNIST')
            output=output_MNIST
          else:
            # print('SVHN')
            output=output_SVHN

          loss = error(output, Labels)
          train_losses.append(loss.data)
          loss.backward()
          optimizer.step()
          # Total correct predictions
          predicted = torch.max(output.data, 1)[1]
          train_correct += (predicted == Labels).sum()
          train_samples_count+=len(Labels)

      if batch_idx % 150 == 0:
          print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.3f}%'.format(
              epoch, batch_idx*len(imgs), len(MNIST_train_loader.dataset), 100.*batch_i

    MTL_train_loss.append(sum(train_losses)/len(train_losses))
    MTL_train_acc.append(float(train_correct/train_samples_count*100))
```

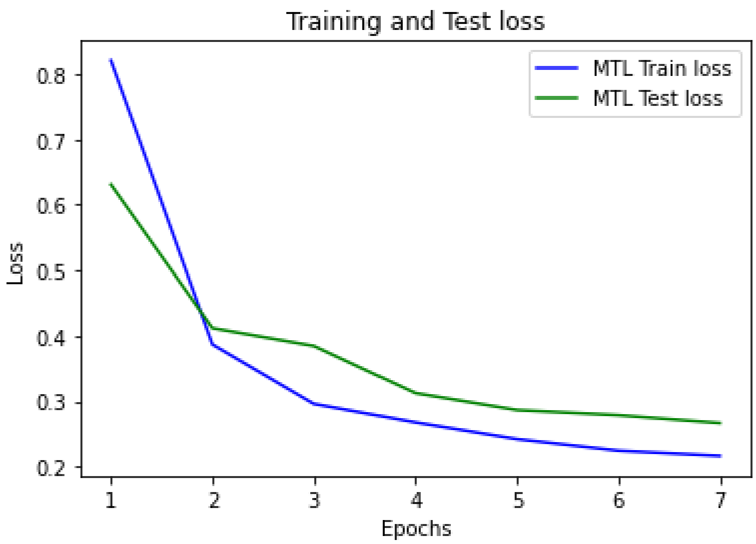
```
    print('Train acc=',float(train_correct/train_samples_count*100))

    test_correct = 0
    test_losses=[]
    for batch_idx, (test_imgs,test_Labels) in enumerate(MNIST_test_loader):
        output,_ = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    for batch_idx, (test_imgs,test_Labels) in enumerate(SVHN_test_loader):
        _,output = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    print("Test accuracy:{:.3f}% ".format( float(test_correct/(len(MNIST_test_dataset)+
    MTL_test_loss.append(sum(test_losses)/len(test_losses))
    print('avg test loss=',sum(test_losses)/len(test_losses))
    MTL_test_acc.append(float(test_correct/(len(MNIST_test_dataset)+len(SVHN_test_datas
```

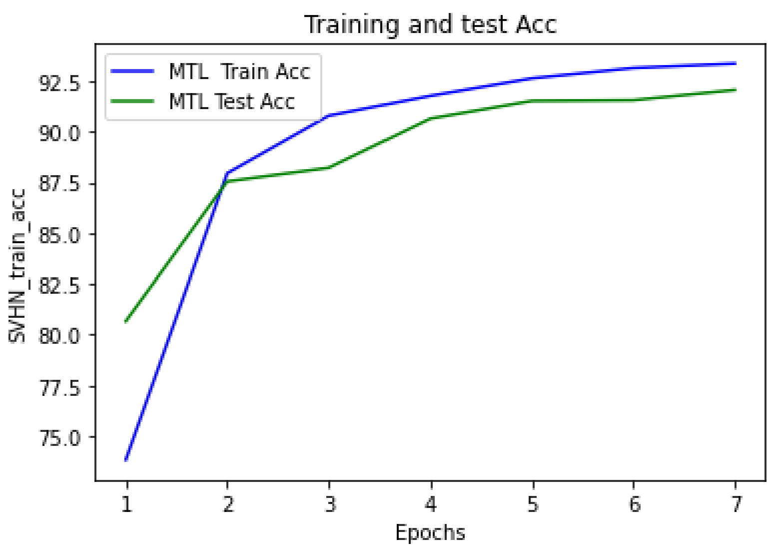
```
Epoch : 0 [0/60000 (0%)]        Loss: 2.348469   Accuracy:6.250%
Epoch : 0 [9600/60000 (16%)]    Loss: 1.829117   Accuracy:51.624%
Epoch : 0 [19200/60000 (32%)]   Loss: 1.567516   Accuracy:60.637%
Epoch : 0 [28800/60000 (48%)]   Loss: 1.398433   Accuracy:65.186%
Epoch : 0 [38400/60000 (64%)]   Loss: 1.149362   Accuracy:68.651%
Epoch : 0 [48000/60000 (80%)]   Loss: 0.888539   Accuracy:71.235%
Epoch : 0 [57600/60000 (96%)]   Loss: 0.937039   Accuracy:73.344%
Train acc= 73.82062530517578
Test accuracy:80.662%
avg test loss= tensor(0.6308)
Epoch : 1 [0/60000 (0%)]        Loss: 0.780600   Accuracy:87.500%
Epoch : 1 [9600/60000 (16%)]    Loss: 0.630469   Accuracy:85.567%
Epoch : 1 [19200/60000 (32%)]   Loss: 0.606204   Accuracy:86.020%
Epoch : 1 [28800/60000 (48%)]   Loss: 0.641406   Accuracy:86.629%
Epoch : 1 [38400/60000 (64%)]   Loss: 0.502103   Accuracy:87.220%
Epoch : 1 [48000/60000 (80%)]   Loss: 0.593826   Accuracy:87.573%
Epoch : 1 [57600/60000 (96%)]   Loss: 0.571278   Accuracy:87.862%
Train acc= 87.9480972290039
Test accuracy:87.542%
avg test loss= tensor(0.4117)
Epoch : 2 [0/60000 (0%)]        Loss: 0.554252   Accuracy:91.406%
Epoch : 2 [9600/60000 (16%)]    Loss: 0.599523   Accuracy:90.512%
Epoch : 2 [19200/60000 (32%)]   Loss: 0.544329   Accuracy:90.408%
Epoch : 2 [28800/60000 (48%)]   Loss: 0.472763   Accuracy:90.483%
Epoch : 2 [38400/60000 (64%)]   Loss: 0.543864   Accuracy:90.656%
Epoch : 2 [48000/60000 (80%)]   Loss: 0.630876   Accuracy:90.650%
Epoch : 2 [57600/60000 (96%)]   Loss: 0.337973   Accuracy:90.802%
Train acc= 90.78824615478516
Test accuracy:88.216%
avg test loss= tensor(0.3845)
Epoch : 3 [0/60000 (0%)]        Loss: 0.518236   Accuracy:92.188%
Epoch : 3 [9600/60000 (16%)]    Loss: 0.437916   Accuracy:91.437%
Epoch : 3 [19200/60000 (32%)]   Loss: 0.523792   Accuracy:91.442%
Epoch : 3 [28800/60000 (48%)]   Loss: 0.621922   Accuracy:91.624%
Epoch : 3 [38400/60000 (64%)]   Loss: 0.468330   Accuracy:91.570%
Epoch : 3 [48000/60000 (80%)]   Loss: 0.413935   Accuracy:91.691%
Epoch : 3 [57600/60000 (96%)]   Loss: 0.303431   Accuracy:91.763%
Train acc= 91.76939392089844
Test accuracy:90.647%
avg test loss= tensor(0.3123)
```

```
Epoch : 4 [0/60000 (0%)]        Loss: 0.574734   Accuracy:93.750%
Epoch : 4 [9600/60000 (16%)]    Loss: 0.412897   Accuracy:92.362%
Epoch : 4 [19200/60000 (32%)]   Loss: 0.135311   Accuracy:92.467%
Epoch : 4 [28800/60000 (48%)]   Loss: 0.640938   Accuracy:92.561%
Epoch : 4 [38400/60000 (64%)]   Loss: 0.314213   Accuracy:92.517%
Epoch : 4 [48000/60000 (80%)]   Loss: 0.313467   Accuracy:92.603%
Epoch : 4 [57600/60000 (96%)]   Loss: 0.439742   Accuracy:92.662%
Train acc= 92.63392639160156
Test accuracy:91.519%
avg test loss= tensor(0.2864)
Epoch : 5 [0/60000 (0%)]        Loss: 0.512545   Accuracy:89.062%
Epoch : 5 [9600/60000 (16%)]    Loss: 0.433634   Accuracy:92.763%
Epoch : 5 [19200/60000 (32%)]   Loss: 0.301593   Accuracy:93.046%
Epoch : 5 [28800/60000 (48%)]   Loss: 0.322182   Accuracy:93.083%
Epoch : 5 [38400/60000 (64%)]   Loss: 0.364382   Accuracy:93.109%
Epoch : 5 [48000/60000 (80%)]   Loss: 0.448662   Accuracy:93.127%
Epoch : 5 [57600/60000 (96%)]   Loss: 0.286689   Accuracy:93.140%
Train acc= 93.14532470703125
Test accuracy:91.555%
avg test loss= tensor(0.2785)
Epoch : 6 [0/60000 (0%)]        Loss: 0.371524   Accuracy:94.531%
Epoch : 6 [9600/60000 (16%)]    Loss: 0.499202   Accuracy:93.072%
Epoch : 6 [19200/60000 (32%)]   Loss: 0.491505   Accuracy:93.088%
Epoch : 6 [28800/60000 (48%)]   Loss: 0.335082   Accuracy:93.297%
Epoch : 6 [38400/60000 (64%)]   Loss: 0.454790   Accuracy:93.226%
Epoch : 6 [48000/60000 (80%)]   Loss: 0.252748   Accuracy:93.343%
Epoch : 6 [57600/60000 (96%)]   Loss: 0.276733   Accuracy:93.357%
Train acc= 93.36520385742188
Test accuracy:92.060%
avg test loss= tensor(0.2665)
Epoch : 7 [0/60000 (0%)]        Loss: 0.503438   Accuracy:90.625%
Epoch : 7 [9600/60000 (16%)]    Loss: 0.364104   Accuracy:93.729%
Epoch : 7 [19200/60000 (32%)]   Loss: 0.387359   Accuracy:93.621%
Epoch : 7 [28800/60000 (48%)]   Loss: 0.268987   Accuracy:93.546%
Epoch : 7 [38400/60000 (64%)]   Loss: 0.196634   Accuracy:93.708%
Epoch : 7 [48000/60000 (80%)]   Loss: 0.493576   Accuracy:93.715%
Epoch : 7 [57600/60000 (96%)]   Loss: 0.512682   Accuracy:93.790%
Train acc= 93.78164672851562
```

In [20]:

```python
# loss_train = history.history['train_loss']
# loss_val = history.history['val_loss']
loss_train=MTL_train_loss
loss_test=MTL_test_loss
epochs = range(1,len(loss_train)+1)
plt.plot(epochs, loss_train, 'b', label='MTL Train loss')
plt.plot(epochs, loss_test, 'g', label='MTL Test loss')
plt.title('Training and Test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

epochs = range(1,len(MTL_train_acc)+1)
plt.plot(epochs, MTL_train_acc, 'b', label='MTL  Train Acc')
plt.plot(epochs, MTL_test_acc, 'g', label='MTL Test Acc')
plt.title('Training and test Acc')
plt.xlabel('Epochs')
plt.ylabel('SVHN_train_acc')
plt.legend()
plt.show()
```





Assig3.Q1.D

In [21]:

```
model=MTL_FC_Net()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
error = nn.CrossEntropyLoss()

loss=0
MTL_train_loss,MTL_test_loss=[],[]
MTL_train_acc,MTL_test_acc=[],[]
imgs,Labels=[],[]
for epoch in range(20):
    MNIST_train_loader = torch.utils.data.DataLoader(MNIST_train_dataset, batch_size=50
    MNIST_test_loader = torch.utils.data.DataLoader(MNIST_test_dataset, batch_size=128,
    SVHN_train_loader = torch.utils.data.DataLoader(SVHN_train_dataset, batch_size=500,
    SVHN_test_loader = torch.utils.data.DataLoader(SVHN_test_dataset, batch_size=128, s
    train_correct=0
    train_losses=[]
    train_samples_count=0
    optimizer.zero_grad()
    for ds in [0,1]:
        if ds==0:
            (imgs,Labels) =next(iter(MNIST_train_loader))
```

```
        else:
            (imgs,Labels) = next(iter(SVHN_train_loader))

        output_MNIST,output_SVHN = model(imgs)
        if ds==0:
          # print('MNIST')
          output=output_MNIST
        else:
          # print('SVHN')
          output=output_SVHN

        loss = error(output, Labels)
        train_losses.append(loss.data)
        loss.backward()
        optimizer.step()
        # Total correct predictions
        predicted = torch.max(output.data, 1)[1]
        train_correct += (predicted == Labels).sum()
        train_samples_count+=len(Labels)

    print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.3f}%'.format(
        epoch, 500, 500, 100.*1 / 500, loss.data, float(train_correct/train_samples_cou

    MTL_train_loss.append(sum(train_losses)/len(train_losses))
    MTL_train_acc.append(float(train_correct/train_samples_count*100))
    print('Train acc=',float(train_correct/train_samples_count*100))

    test_correct = 0
    test_losses=[]
    for batch_idx, (test_imgs,test_Labels) in enumerate(MNIST_test_loader):
        output,_ = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    for batch_idx, (test_imgs,test_Labels) in enumerate(SVHN_test_loader):
        _,output = model(test_imgs)
        loss = error(output, test_Labels)
        test_losses.append(loss.data)
        predicted = torch.max(output,1)[1]
        test_correct += (predicted == test_Labels).sum()

    print("Test accuracy:{:.3f}% ".format( float(test_correct/(len(MNIST_test_dataset)+
    MTL_test_loss.append(sum(test_losses)/len(test_losses))
    print('avg test loss=',sum(test_losses)/len(test_losses))
    MTL_test_acc.append(float(test_correct/(len(MNIST_test_dataset)+len(SVHN_test_datas
```

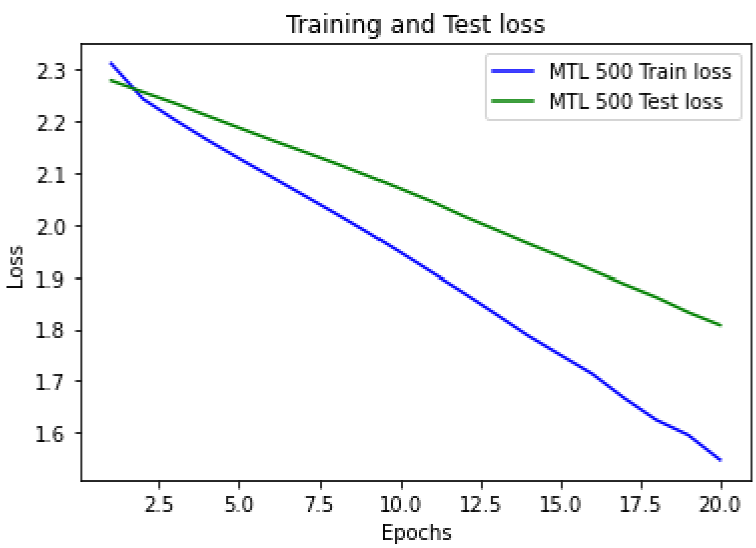
```
Epoch : 0 [500/500 (0%)]        Loss: 2.317389   Accuracy:9.900%
Train acc= 9.899999618530273
Test accuracy:15.744%
avg test loss= tensor(2.2782)
Epoch : 1 [500/500 (0%)]        Loss: 2.282119   Accuracy:22.300%
Train acc= 22.30000114440918
Test accuracy:21.362%
avg test loss= tensor(2.2558)
Epoch : 2 [500/500 (0%)]        Loss: 2.263654   Accuracy:34.600%
Train acc= 34.599998474121094
Test accuracy:25.130%
avg test loss= tensor(2.2341)
Epoch : 3 [500/500 (0%)]        Loss: 2.247726   Accuracy:40.200%
```

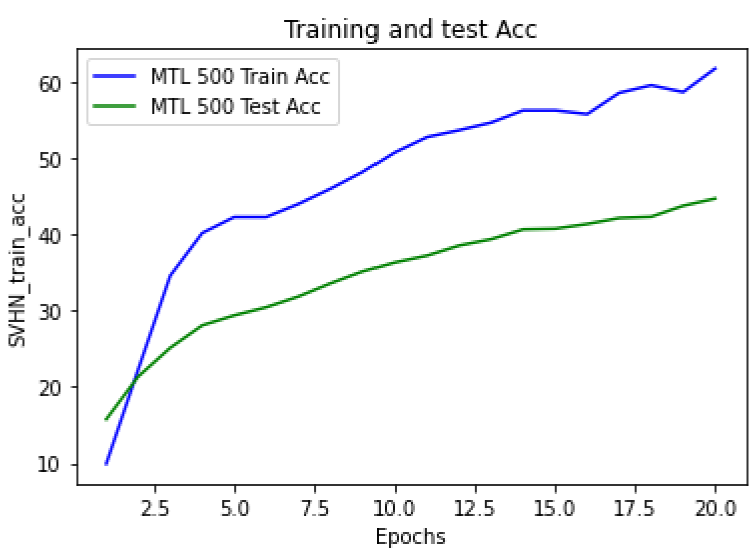
```
Train acc= 40.20000076293945
Test accuracy:28.081%
avg test loss= tensor(2.2103)
Epoch : 4 [500/500 (0%)]        Loss: 2.231842   Accuracy:42.300%
Train acc= 42.29999923706055
Test accuracy:29.385%
avg test loss= tensor(2.1868)
Epoch : 5 [500/500 (0%)]        Loss: 2.217974   Accuracy:42.300%
Train acc= 42.29999923706055
Test accuracy:30.456%
avg test loss= tensor(2.1636)
Epoch : 6 [500/500 (0%)]        Loss: 2.204693   Accuracy:44.000%
Train acc= 44.0
Test accuracy:31.855%
avg test loss= tensor(2.1411)
Epoch : 7 [500/500 (0%)]        Loss: 2.190568   Accuracy:46.000%
Train acc= 46.0
Test accuracy:33.578%
avg test loss= tensor(2.1184)
Epoch : 8 [500/500 (0%)]        Loss: 2.175014   Accuracy:48.200%
Train acc= 48.20000076293945
Test accuracy:35.147%
avg test loss= tensor(2.0945)
Epoch : 9 [500/500 (0%)]        Loss: 2.157200   Accuracy:50.800%
Train acc= 50.80000305175781
Test accuracy:36.343%
avg test loss= tensor(2.0696)
Epoch : 10 [500/500 (0%)]       Loss: 2.137777   Accuracy:52.800%
Train acc= 52.79999923706055
Test accuracy:37.239%
avg test loss= tensor(2.0442)
Epoch : 11 [500/500 (0%)]       Loss: 2.118026   Accuracy:53.700%
Train acc= 53.70000076293945
Test accuracy:38.557%
avg test loss= tensor(2.0158)
Epoch : 12 [500/500 (0%)]       Loss: 2.098904   Accuracy:54.700%
Train acc= 54.70000076293945
Test accuracy:39.393%
avg test loss= tensor(1.9898)
Epoch : 13 [500/500 (0%)]       Loss: 2.077061   Accuracy:56.300%
Train acc= 56.30000305175781
Test accuracy:40.678%
avg test loss= tensor(1.9637)
Epoch : 14 [500/500 (0%)]       Loss: 2.065253   Accuracy:56.300%
Train acc= 56.30000305175781
Test accuracy:40.769%
avg test loss= tensor(1.9387)
Epoch : 15 [500/500 (0%)]       Loss: 2.053659   Accuracy:55.800%
Train acc= 55.80000305175781
Test accuracy:41.383%
avg test loss= tensor(1.9125)
Epoch : 16 [500/500 (0%)]       Loss: 2.020448   Accuracy:58.600%
Train acc= 58.60000228881836
Test accuracy:42.168%
avg test loss= tensor(1.8864)
Epoch : 17 [500/500 (0%)]       Loss: 1.997984   Accuracy:59.600%
Train acc= 59.60000228881836
Test accuracy:42.335%
avg test loss= tensor(1.8614)
Epoch : 18 [500/500 (0%)]       Loss: 1.995022   Accuracy:58.700%
Train acc= 58.70000076293945
Test accuracy:43.775%
avg test loss= tensor(1.8325)
Epoch : 19 [500/500 (0%)]       Loss: 1.963720   Accuracy:61.800%
Train acc= 61.79999542236328
```

```
Test accuracy:44.727%
avg test loss= tensor(1.8077)
```

In [22]:

```
# loss_train = history.history['train_loss']
# loss_val = history.history['val_loss']
loss_train=MTL_train_loss
loss_test=MTL_test_loss
epochs = range(1,len(loss_train)+1)
plt.plot(epochs, loss_train, 'b', label='MTL 500 Train loss')
plt.plot(epochs, loss_test, 'g', label='MTL 500 Test loss')
plt.title('Training and Test loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

epochs = range(1,len(MTL_train_acc)+1)
plt.plot(epochs, MTL_train_acc, 'b', label='MTL 500 Train Acc')
plt.plot(epochs, MTL_test_acc, 'g', label='MTL 500 Test Acc')
plt.title('Training and test Acc')
plt.xlabel('Epochs')
plt.ylabel('SVHN_train_acc')
plt.legend()
plt.show()
```





Question 1 (e/f)

In this question we will train a joint embedding between a model embedding from MNIST and a model embedding from SVHN dataset, both digit datasets. Your specific task to evaluate this will be to try to obtain 50% or higher accuracy on the MNIST classification by embedding MNIST test digits and then searching for the 1-nearest neighbor SVHN digit and using it's category to classify.

First we will define the mnist and svhn models. For svhn we will use a pre-trained model that can already classify svhn digits. The models are defined below

In [344…

```
from torch.utils import model_zoo
from collections import OrderedDict
## MNIST model
model_mnist = nn.Sequential(nn.Conv2d(1, 32, 5), nn.BatchNorm2d(32), nn.ReLU(), #For (e
                      nn.MaxPool2d(2, stride=2),
                      nn.Conv2d(32, 64, 5), nn.BatchNorm2d(64), nn.ReLU(),
                      nn.Conv2d(64, 64, 3), nn.BatchNorm2d(64), nn.ReLU(),
                      nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())


### SVHN model, we will download one that is already trained to clasify svhn digits
model_urls = {
    'svhn': 'http://ml.cs.tsinghua.edu.cn/~chenxi/pytorch-models/svhn-f564f3d8.pth',
}

class SVHN_Model(nn.Module):
    def __init__(self, features, n_channel, num_classes):
        super(SVHN_Model, self).__init__()
        assert isinstance(features, nn.Sequential), type(features)
        self.features = features

        #We won't use this classifier
        self.classifier = nn.Sequential(
            nn.Linear(n_channel, num_classes)
        )
        print(self.features)
        print(self.classifier)

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

def make_layers(cfg, batch_norm=False):
    layers = []
    in_channels = 3
    for i, v in enumerate(cfg):
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            padding = v[1] if isinstance(v, tuple) else 1
            out_channels = v[0] if isinstance(v, tuple) else v
            conv2d = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=paddin
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(out_channels, affine=False), nn.ReLU(
            else:
                layers += [conv2d, nn.ReLU(), nn.Dropout(0.3)]
```

```
            in_channels = out_channels
    return nn.Sequential(*layers)

def svhn_model(n_channel, pretrained=None):
    cfg = [n_channel, n_channel, 'M', 2*n_channel, 2*n_channel, 'M', 4*n_channel, 4*n_c
    layers = make_layers(cfg, batch_norm=True)
    model = SVHN_Model(layers, n_channel=8*n_channel, num_classes=10)
    if pretrained is not None:
        m = model_zoo.load_url(model_urls['svhn'],map_location=torch.device('cpu'))
        state_dict = m.state_dict() if isinstance(m, nn.Module) else m
        assert isinstance(state_dict, (dict, OrderedDict)), type(state_dict)
        model.load_state_dict(state_dict)

    return model


base_svhn = svhn_model(n_channel=32,pretrained=True).features
svhn_to_joint = nn.Linear(256,64)

# model_svhn = nn.Sequential(base_svhn, nn.AdaptiveAvgPool2d((1,1)), nn.Flatten(), svhn_

#Transformation for SVHN data, you need to use this normalization for the pre-trained m
SVHN_Model_transform=transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                ])
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (2): ReLU()
  (3): Dropout(p=0.3, inplace=False)
  (4): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (6): ReLU()
  (7): Dropout(p=0.3, inplace=False)
  (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (11): ReLU()
  (12): Dropout(p=0.3, inplace=False)
  (13): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (15): ReLU()
  (16): Dropout(p=0.3, inplace=False)
  (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (18): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (19): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (20): ReLU()
  (21): Dropout(p=0.3, inplace=False)
  (22): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (23): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (24): ReLU()
  (25): Dropout(p=0.3, inplace=False)
  (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (27): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (28): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (29): ReLU()
  (30): Dropout(p=0.3, inplace=False)
  (31): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
)
Sequential(
  (0): Linear(in_features=256, out_features=10, bias=True)
)
```

Test SVHN Model Acc

In [201…

```
fc_svhn_model = svhn_model(n_channel=32,pretrained=True)
SVHN_test_dataset_3c = SVHN('../data/SVHN',split="test", download=True,transform=SVHN_M
SVHN_test_loader_3c = torch.utils.data.DataLoader(SVHN_test_dataset_3c, batch_size=128,
train_correct=0
for (imgs,Labels) in SVHN_test_loader_3c:
    # (imgs,Labels)= next(iter( SVHN_test_loader_3c))
    # print(imgs.shape)
    output = fc_svhn_model(imgs)
    # print(output)
    # print(Labels)
    predicted = (torch.max(output.data, 1)[1]+1)%10
    # print(predicted)
    # print(predicted.shape)
    train_correct += (predicted == Labels).sum()
print("model Acc=",train_correct/len(SVHN_test_dataset_3c))
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (2): ReLU()
  (3): Dropout(p=0.3, inplace=False)
  (4): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (6): ReLU()
  (7): Dropout(p=0.3, inplace=False)
  (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (11): ReLU()
  (12): Dropout(p=0.3, inplace=False)
  (13): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=True)
  (15): ReLU()
  (16): Dropout(p=0.3, inplace=False)
  (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (18): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (19): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (20): ReLU()
  (21): Dropout(p=0.3, inplace=False)
  (22): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (23): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (24): ReLU()
  (25): Dropout(p=0.3, inplace=False)
  (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (27): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (28): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
  (29): ReLU()
  (30): Dropout(p=0.3, inplace=False)
  (31): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
Sequential(
  (0): Linear(in_features=256, out_features=10, bias=True)
```

```
)
Using downloaded and verified file: ../data/SVHN/test_32x32.mat
model Acc= tensor(0.9473)
```

Suggested settings: learning rate 1e-5 with Adam, margin (α) of 0.2, batch size: 256 triplets samples M and 256 from S, 1000 training iterations (not epochs, but gradient updates/minibatch processed, aka it can be trained fast!). You may modify these as you see fit.

Data augmentation is not required to make this work but you may use it if you like. For SVHN you must use the normalization above (transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))) so that the pre-trained SVHN model works.

Sampling the triplets There are various valid ways you could construct the triplet sets $\mathbf{M}, \mathbf{S}$ and sample from them. For example you could enumerate all possible triplets over the dataset and select batches of these. A quick and dirty on the fly method that allows to use standard dataloaders is as follows: Sample a minibatch of size N (say 256) from both SVHN and MNIST using standard dataloaders from classification tasks. Treat all SVHN digits in this batch as anchors, from the MNIST minibatch data find appropriate positives and negatives for each SVHN digit. For the second part of the loss treat the MNIST data as anchors and find negatives and postivies from the SVHN minibatch. Partial code snippets to construct this is shown below (note this code would give triplets for $\mathbf{M}$ part only). You may also use your own approach to sample the triplet sets.

Note: if you would like to use hard negative mining (not required) a more sophisticated approach would be needed. Below is a code snippet example of how one could pick the positives using the labels for each minibatch.

Note we only optimize W and θ, below see an example how to build the optimizer. Note we want to freeze the g_γ model so we will also need to disable the dropout and batchnorm.

Appropriate triplet construction and loss function construction

In [107…

```
import numpy as np
from PIL import Image
from torch.utils.data import Dataset
class TripletDs(Dataset):
    """
    Train: For each sample (anchor) randomly chooses a positive and negative samples
    Test: Creates fixed triplets for testing
    """
    def __init__(self, ds):
        self.ds = ds
        self.transform = self.ds.transform
        if hasattr(ds, 'targets'):
          self.labels =np.asarray(self.ds.targets)
        else:
          self.labels =np.asarray(self.ds.labels)
        self.data = self.ds.data
        self.labels_set = set(self.labels)
        self.label_to_indices = {label: np.where(self.labels == label)[0]
                                 for label in self.labels_set}

        random_state = np.random.RandomState(29)
```

```
        # for i in range(len(self.data)):
        #   # print(i)
        #   # print(self.labels[i])
        #   # print(self.label_to_indices[self.labels[i]])
        #   # random_state.choice(self.label_to_indices[self.labels[i]])
        triplets = [[i,
                     random_state.choice(self.label_to_indices[self.labels[i]]),
                     random_state.choice(self.label_to_indices[
                                             np.random.choice(
                                                 list(self.labels_set - set([self.labe
                                             )
                                         ])
                     ]
                    for i in range(len(self.data))]
        self.triplets = triplets

    def __getitem__(self, index):

        img1, label1 = self.data[index], self.labels[index]
        positive_index = index
        while positive_index == index:
            positive_index = np.random.choice(self.label_to_indices[label1])
        negative_label = np.random.choice(list(self.labels_set - set([label1])))
        negative_index = np.random.choice(self.label_to_indices[negative_label])
        img2 = self.data[positive_index]
        img3 = self.data[negative_index]

        # print('img1.shape',img1.shape)
        # print(type(img1))
        # img1 = Image.fromarray(img1[:,:,1], mode='L')
        # img2 = Image.fromarray(img2[:,:,1], mode='L')
        # img3 = Image.fromarray(img3[:,:,1], mode='L')
        if isinstance(img1, np.ndarray)== False:
          img1 = Image.fromarray(img1.numpy(), mode='L')
          img2 = Image.fromarray(img2.numpy(), mode='L')
          img3 = Image.fromarray(img3.numpy(), mode='L')
        else:
          img1 = Image.fromarray(img1[0,:,:], mode='L')
          img2 = Image.fromarray(img2[0,:,:], mode='L')
          img3 = Image.fromarray(img3[0,:,:], mode='L')


        if self.transform is not None:
            img1 = self.transform(img1)
            img2 = self.transform(img2)
            img3 = self.transform(img3)
        return (img1, img2, img3), label1
    def __len__(self):
        return len(self.ds)
```

In [163…

```
ds=TripletDs(SVHN_train_dataset)
SVHN_train_dataset.labels
# SVHN_train_dataset
triplet_train_loader = torch.utils.data.DataLoader(ds, batch_size=32, shuffle=True)
(img1,img2,img3),l =next(iter( triplet_train_loader))
print(img1[0].shape,l)
```

```
torch.Size([1, 32, 32]) tensor([3, 2, 2, 1, 1, 3, 0, 2, 8, 1, 1, 1, 5, 1, 4, 9, 9, 6, 0,
6, 1, 8, 2, 7,
```

```
5, 6, 8, 9, 9, 6, 5, 9])
```

Let's denote model$mnist above as f{\theta}(x), $the pre_trained model$g_{\gamma} $and svhn_to_joint as the matrix$W. $Finally model_svhn corresponds to$WAg_{\gamma}(x) $. Here A(nn.AdaptiveAvgPool2d) is the averaging operator and has no parameters. Thus model_svhn will map svhn digits to a joint space and model_mnist will map MNIST digits to the$ g_{ \gamma}$fixed and update$\theta, W$. You should optimize the following objective that is a sum of two loss functions over triplets

$$\min_{\theta,W} \sum_{x_a,x_p,x_n \in \mathbf{M}} max(0, \|f_\theta(x_a) - WAg_\gamma(x_p)\| - \|f_\theta(x_a) - WAg_\gamma(x_n)\| + \alpha) + \sum_{x_a,x_p,x_n \in \mathbf{S}} max(0, \|f$$

Here $\mathbf{M}$ is the set of triplets with anchors from MNIST data, positives from SVHN (matching the anchor class), and negatives from SVHN (with different class from anchors). Similarly $\mathbf{S}$ is the set of triplets with anchors from SVHN data, positives from MNIST (matching anchor class), and negatives from MNIST not matching anchor class. You can use nn.TripletMarginLoss to implement this.

During training with a stochastic optimizer we will sample subsets of M and S for each gradient update, there are various valid ways to sample this as will be discussed.

In [158…

```
import numpy as np
from PIL import Image
from torch.utils.data import Dataset
class TripletFrom2Ds(Dataset):
    """
    Train: For each sample (anchor) randomly chooses a positive and negative samples
    Test: Creates fixed triplets for testing
    """
    def __init__(self, ds1,ds2):
        self.ds1 = ds1
        self.ds2 = ds2
        self.transform1 = self.ds1.transform
        self.transform2 = self.ds2.transform
        if hasattr(ds1, 'targets'):
          self.labels1 =np.asarray(self.ds1.targets)
        else:
          self.labels1 =np.asarray(self.ds1.labels)

        if hasattr(ds2, 'targets'):
          self.labels2 =np.asarray(self.ds2.targets)
        else:
          self.labels2 =np.asarray(self.ds2.labels)

        self.data1 = self.ds1.data
        self.data2 = self.ds2.data

        self.labels_set1 = set(self.labels1)
        self.labels_set2 = set(self.labels2)

        self.label_to_indices1 = {label: np.where(self.labels1 == label)[0]
                                  for label in self.labels_set1}
        self.label_to_indices2 = {label: np.where(self.labels2 == label)[0]
                                  for label in self.labels_set2}

        random_state = np.random.RandomState(29)
```

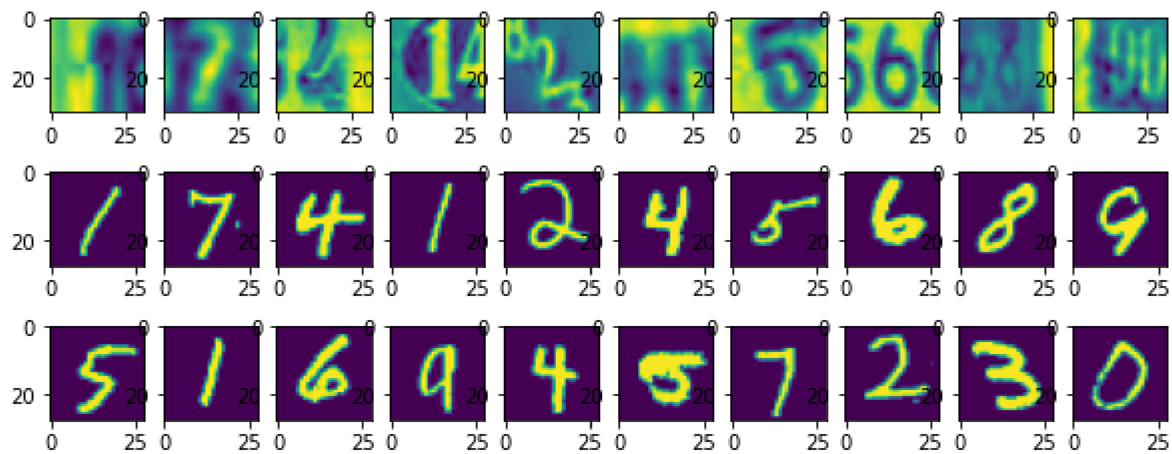
```
        # for i in range(len(self.data)):
        #   # print(i)
        #   # print(self.labels[i])
        #   # print(self.label_to_indices[self.labels[i]])
        #   # random_state.choice(self.label_to_indices[self.labels[i]])
        triplets = [[i,
                     random_state.choice(self.label_to_indices2[self.labels1[i]]),
                     random_state.choice(self.label_to_indices2[
                                             np.random.choice(
                                                 list(self.labels_set1 - set([self.lab
                                             )
                                         ])
                     ]
                    for i in range(len(self.data1))]
        self.triplets = triplets

    def __getitem__(self, index):

        img1, label1 = self.data1[self.triplets[index][0]], self.labels1[index]
        img2 = self.data2[self.triplets[index][1]]
        img3 = self.data2[self.triplets[index][2]]

        # print('img1.shape',img1.shape)
        # print(type(img1))
        # img1 = Image.fromarray(img1[:,:,1], mode='L')
        # img2 = Image.fromarray(img2[:,:,1], mode='L')
        # # img3 = Image.fromarray(img3[:,:,1], mode='L')

        if isinstance(img1, np.ndarray)== False:
          img1 = Image.fromarray(img1.numpy(),mode="L")
        else:
          img1 = Image.fromarray(img1[0,:,:],mode="L")

        if isinstance(img2, np.ndarray)== False:
          img2 = Image.fromarray(img2.numpy(),mode="L")
          img3 = Image.fromarray(img3.numpy(),mode="L")
        else:
          img2 = Image.fromarray(img2[0,:,:],mode="L")
          img3 = Image.fromarray(img3[0,:,:],mode="L")

        if self.transform1 is not None:
            img1 = self.transform1(img1)
        if self.transform2 is not None:
            img2 = self.transform2(img2)
            img3 = self.transform2(img3)
        return (img1, img2, img3), label1
    def __len__(self):
        return len(self.ds1)
```

In [321…

```
ds=TripletFrom2Ds(SVHN_train_dataset,MNIST_train_dataset)
SVHN_train_dataset.labels
# SVHN_train_dataset
triplet_train_loader = torch.utils.data.DataLoader(ds, batch_size=32, shuffle=True)
(img1,img2,img3),l =next(iter( triplet_train_loader))
print(img1[0].shape,img2[0].shape,img2[0].shape,l)

#anchor, positive, negative
import matplotlib.pyplot as plt
```

```
plt.figure()
#subplot(r,c) provide the no. of rows and columns
c=10
f, axarr = plt.subplots(3,c)
f.set_size_inches(c,4)
for i in range(c):
  # use the created array to output your multiple images. In this case I have stacked 4
  axarr[0,i].imshow(img1[i].squeeze())
  axarr[1,i].imshow(img2[i].squeeze())
  axarr[2,i].imshow(img3[i].squeeze())
```

```
torch.Size([1, 32, 32]) torch.Size([1, 28, 28]) torch.Size([1, 28, 28]) tensor([1, 7, 4,
1, 2, 4, 5, 6, 8, 9, 8, 4, 2, 8, 1, 2, 1, 9, 0, 1, 2, 2, 8, 2,
        2, 2, 2, 1, 1, 5, 4, 4])
<Figure size 432x288 with 0 Axes>
```



In [347…

```
#Use these dataloaders
from torchvision.datasets import MNIST
from torchvision.datasets import SVHN
from torchvision import transforms
mean, std = 0.1307, 0.3081
MNIST_train_dataset = MNIST('../data/MNIST', train=True, download=True, transform=trans
                                 transforms.ToTensor(),
                                 transforms.Normalize((mean,), (std,))
                             ]))
MNIST_test_dataset = MNIST('../data/MNIST', train=False, download=False,transform=trans
                                 transforms.ToTensor(),
                                 transforms.Normalize((mean,), (std,))
                             ]))

SVHN_train_dataset = SVHN('../data/SVHN',split="train", download=True, transform=transf
                                 transforms.ToTensor(),
                                #  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5
                                 transforms.Normalize((0.5,), (0.5,))
                             ]))
SVHN_test_dataset = SVHN('../data/SVHN',split="test", download=True,transform=transform
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5,), (0.5,))
                             ]))
n_classes = 10

triplet_train_ds_M = TripletFrom2Ds(MNIST_train_dataset,SVHN_train_dataset)
triplet_train_loader_M = torch.utils.data.DataLoader(triplet_train_ds_M, batch_size=256
print(len(triplet_train_ds_M),len(triplet_train_loader_M))
```

```
triplet_train_ds_S = TripletFrom2Ds(SVHN_train_dataset,MNIST_train_dataset)
triplet_train_loader_S = torch.utils.data.DataLoader(triplet_train_ds_S, batch_size=256
print(len(triplet_train_ds_S),len(triplet_train_loader_S))

triplet_test_ds_M = TripletFrom2Ds(MNIST_test_dataset,SVHN_test_dataset)
triplet_test_loader_M = torch.utils.data.DataLoader(triplet_test_ds_M, batch_size=256,
print(len(triplet_test_ds_M),len(triplet_test_loader_M))
triplet_test_ds_S = TripletFrom2Ds(SVHN_test_dataset,MNIST_test_dataset)
triplet_test_loader_S = torch.utils.data.DataLoader(triplet_test_ds_S, batch_size=256,
print(len(triplet_test_ds_S),len(triplet_test_loader_S))
```

```
Using downloaded and verified file: ../data/SVHN/train_32x32.mat
Using downloaded and verified file: ../data/SVHN/test_32x32.mat
60000 235
73257 287
10000 40
26032 102
```

In [348…

```
(Mim1,Mim2,Mim3),Mlabels  =next(iter(triplet_train_loader_M))
```

In [349…

```
import torch.optim  as optim
model_mnist = nn.Sequential(nn.Conv2d(1, 32, 5), nn.BatchNorm2d(32), nn.ReLU(), #For (e
                      nn.MaxPool2d(2, stride=2),
                      nn.Conv2d(32, 64, 5), nn.BatchNorm2d(64), nn.ReLU(),
                      nn.Conv2d(64, 64, 3), nn.BatchNorm2d(64), nn.ReLU(),
                      nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())

model_svhn = nn.Sequential(base_svhn, nn.AdaptiveAvgPool2d((1,1)), nn.Flatten(), svhn_t

optimizer = optim.Adam(list(model_mnist.parameters()) + list(svhn_to_joint.parameters()
# print(model_svhn)
model_svhn[0][0]=nn.Conv2d(1,32,kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
model_svhn.eval() #IMPORTANT: BEFORE running set to eval even for training to avoid dro
```

Out[349…

```
Sequential(
  (0): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
    (2): ReLU()
    (3): Dropout(p=0.3, inplace=False)
    (4): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
    (6): ReLU()
    (7): Dropout(p=0.3, inplace=False)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
    (11): ReLU()
    (12): Dropout(p=0.3, inplace=False)
    (13): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tru
e)
    (15): ReLU()
    (16): Dropout(p=0.3, inplace=False)
    (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (18): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tr
```

```
ue)
    (20): ReLU()
    (21): Dropout(p=0.3, inplace=False)
    (22): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tr
ue)
    (24): ReLU()
    (25): Dropout(p=0.3, inplace=False)
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
    (28): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=Tr
ue)
    (29): ReLU()
    (30): Dropout(p=0.3, inplace=False)
    (31): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (1): AdaptiveAvgPool2d(output_size=(1, 1))
  (2): Flatten(start_dim=1, end_dim=-1)
  (3): Linear(in_features=256, out_features=64, bias=True)
)
```

Model Training+ Loss Function

In [350…

```python
model_svhn.eval()
triplet_loss = nn.TripletMarginLoss(margin=0.35, p=2)
loss=[]
counter=[]
print("batch count",len(triplet_train_loader_M))
iteration_number = 0
for epoch in range(10):
    all_loss=[]
    for batch_idx in range(len(triplet_train_loader_M)):
    # for batch_idx in range(51):
      optimizer.zero_grad()
      (Mim1,Mim2,Mim3),Mlabels  =next(iter(triplet_train_loader_M))
      (Sim1,Sim2,Sim3),Slabels  =next(iter(triplet_train_loader_S))
      M_output1,M_output2,M_output3 = model_mnist(Mim1),model_svhn(Mim2),model_svhn(Mim
      S_output1,S_output2,S_output3 = model_svhn(Sim1),model_mnist(Sim2),model_mnist(Si
      loss = triplet_loss(M_output1,M_output2,M_output3)+triplet_loss(S_output1,S_outpu
      all_loss.append(loss.item())
      loss.backward()
      optimizer.step()
      if batch_idx%10==0 :
        print("batch {} Avg loss {}".format(batch_idx, sum(all_loss)/len(all_loss)))
    print("Epoch {} Avg loss {}".format(epoch, sum(all_loss)/len(all_loss)))
```

```
batch count 235
batch 0 Avg loss 0.9142905473709106
batch 10 Avg loss 0.8371804248202931
batch 20 Avg loss 0.7877500255902609
batch 30 Avg loss 0.7511897587007091
batch 40 Avg loss 0.7274452665957009
batch 50 Avg loss 0.7026626117089215
batch 60 Avg loss 0.6822603325374791
batch 70 Avg loss 0.6658651350249707
batch 80 Avg loss 0.6467304001619787
batch 90 Avg loss 0.6275666991432944
batch 100 Avg loss 0.6110382968246346
batch 110 Avg loss 0.596014142036438
batch 120 Avg loss 0.5822625455777507
batch 130 Avg loss 0.5685951914496095
batch 140 Avg loss 0.5574167085454819
```

```
batch 150 Avg loss 0.5441064947093559
batch 160 Avg loss 0.5320781210194463
batch 170 Avg loss 0.522877452840582
batch 180 Avg loss 0.5156716225226281
batch 190 Avg loss 0.5093478240579835
batch 200 Avg loss 0.5023698630321085
batch 210 Avg loss 0.49470216869177974
batch 220 Avg loss 0.48736840338189136
batch 230 Avg loss 0.4804113623383757
Epoch 0 Avg loss 0.47755846064141455
batch 0 Avg loss 0.29335862398147583
batch 10 Avg loss 0.31289423595775256
batch 20 Avg loss 0.31515498956044513
batch 30 Avg loss 0.31464156220036166
batch 40 Avg loss 0.3170786085652142
batch 50 Avg loss 0.3165870826033985
batch 60 Avg loss 0.3120058508681469
batch 70 Avg loss 0.31507919329992484
batch 80 Avg loss 0.31681060791015625
batch 90 Avg loss 0.3131962829566264
batch 100 Avg loss 0.3097137025382259
batch 110 Avg loss 0.30774675457327216
batch 120 Avg loss 0.30471750071718673
batch 130 Avg loss 0.3020946339114022
batch 140 Avg loss 0.29904886773714784
batch 150 Avg loss 0.29687253361111443
batch 160 Avg loss 0.29499212666327906
batch 170 Avg loss 0.2943400228232668
batch 180 Avg loss 0.29409306961528503
batch 190 Avg loss 0.29321248057000926
batch 200 Avg loss 0.2924305625371079
batch 210 Avg loss 0.2917666003743619
batch 220 Avg loss 0.28999451052279496
batch 230 Avg loss 0.2890546525711621
Epoch 1 Avg loss 0.28841100303416556
batch 0 Avg loss 0.23908992111682892
batch 10 Avg loss 0.24262025410478766
batch 20 Avg loss 0.24828851648739406
batch 30 Avg loss 0.25156896297008763
batch 40 Avg loss 0.24779461469592118
batch 50 Avg loss 0.24705850815071778
batch 60 Avg loss 0.2479277819883628
batch 70 Avg loss 0.24799994473725978
batch 80 Avg loss 0.24984152026382495
batch 90 Avg loss 0.24875180275885614
batch 100 Avg loss 0.24845667906326824
batch 110 Avg loss 0.24825147707183082
batch 120 Avg loss 0.24669634199832097
batch 130 Avg loss 0.24663799193524222
batch 140 Avg loss 0.2455350285303508
batch 150 Avg loss 0.24533810254359087
batch 160 Avg loss 0.24394018272435444
batch 170 Avg loss 0.2427940136856503
batch 180 Avg loss 0.24157865427804914
batch 190 Avg loss 0.241487733122566
batch 200 Avg loss 0.24083987419581532
batch 210 Avg loss 0.24068846938451885
batch 220 Avg loss 0.2397979987422805
batch 230 Avg loss 0.23863288444100003
Epoch 2 Avg loss 0.23842265599585594
batch 0 Avg loss 0.23202326893806458
batch 10 Avg loss 0.22956952452659607
```

Evaluation For evaluating your embeddings use 2000 randomly selected SVHN digits from the

SVHN training set embedding them with model_svhn. Use 100 randomly selected MNIST digits from the MNIST TEST set embedding them with model_mnist. The above numbers are chosen to avoid memory issues and reduce computation time, you may use larger amount of test inputs and embeddings if you wish. Assume the category data for the SVHN data is known and find for each MNIST digit the nearest SVHN digit. Report it's category as the prediction and compute the accuracy over all 100 MNIST digits. You should be able to obtain at least 50%+ although much higher accuracy is possible with a well tuned model.

Finally for 3-5 MNIST digits show the top 5 SVHN sorted by lowest distance. (now extra credit but easy if the model works)

Appropriate nearest neighbor classification evaluation setup

In [352…

```
model_svhn.eval()
svhn_embds=[]
for batch_idx in range(8):
  (Mim1,Mim2,Mim3),labels  =next(iter(triplet_train_loader_S))
  output1 = model_svhn(Mim1)
  svhn_embds.append([output1,labels])
```

In [353…

```
mnist_embds=[]
(im1,im2,im3),labels  =next(iter(triplet_test_loader_M))
output = model_mnist(im1)
mnist_embds.append([output,labels])
```

In [371…

```
from scipy.spatial import distance
import numpy as np
nearestM=[]
for idx in range(len(mnist_embds[0][1])):
  m_emb,m_label=mnist_embds[0][0][idx],mnist_embds[0][1][idx]
  min_dis=1000
  min_label=-1
  for emb_batch in svhn_embds:
     s_embds,s_labels = emb_batch[0],emb_batch[1]
     for s_idx in range(len(s_labels)):
       dis=distance.euclidean(m_emb.detach(),s_embds[s_idx].detach())
       if dis<min_dis: #Calculate Min Distance and its label
            min_dis=dis
            min_label=s_labels[s_idx]
  nearestM.append([m_emb,m_label,min_dis,min_label])

print("M label\t\tmin distance\t\tS label")
for idx in range(5):
  print(nearestM[idx][1].item(),"\t",nearestM[idx][2],"\t",nearestM[idx][3].item())
```

```
M label         min distance            S label
8        1.9095484018325806      6
3        1.9493529796600342      3
3        2.083371162414551       3
1        1.9679996967315674      3
0        2.629142999649048       0
```

Finally for 3-5 MNIST digits show the top 5 SVHN sorted by lowest distance.

In [382…

```
from scipy.spatial import distance
import numpy as np
nearestM=[]
print("M_label\tdis\tS_label")
for idx in range(6,12):
  m_emb,m_label=mnist_embds[0][0][idx],mnist_embds[0][1][idx]
  min_dis=2.15
  min_label=-1
  for emb_batch in svhn_embds:
     s_embds,s_labels = emb_batch[0],emb_batch[1]
     for s_idx in range(len(s_labels)):
       dis=distance.euclidean(m_emb.detach(),s_embds[s_idx].detach())
       if dis<min_dis: #Calculate Min Distance and its label
            print(m_label.item(),dis,s_labels[s_idx].item())
  print("\n-----------------------------------")
```

```
M_label dis     S_label
2 2.0858728885650635 2
2 2.081221342086792 2
2 2.148451089859009 2
2 2.0675442218780518 2

-----------------------------------
7 2.0645668506622314 7
7 2.042292356491089 7
7 1.9792007207870483 7
7 1.8843077421188354 7

-----------------------------------
1 2.0250988006591797 1
1 1.9546384811401367 1
1 2.135309934616089 1
1 2.0494911670684814 1
1 1.9982092380523682 3
1 2.091998338699341 4

-----------------------------------
1 2.096865653991699 1
1 1.9344192743301392 1
1 1.9317368268966675 1
1 2.0788216590881348 1
1 1.9840689897537231 1
1 2.1280500888824463 1
1 2.093714714050293 4
1 1.9541305303573608 3
1 2.1435177326202393 1
1 2.061178684234619 4
1 2.149013042449951 1

-----------------------------------
4 2.1257119178771973 4
4 2.1163806915283203 4
4 2.129732847213745 4
4 2.0473520755767822 4

-----------------------------------
3 2.1371452808380127 5
```

Obtaining above 50% accuracy

In [356…

```
correct=0
for elem in nearestM:
  if elem[1].item()==elem[3].item():
      correct+=1
  # print(elem[1],elem[2],elem[3])
print("Joint Embedding acc =",correct*100/len(nearestM),"%")
```

```
Joint Embedding acc = 59.765625 %
```

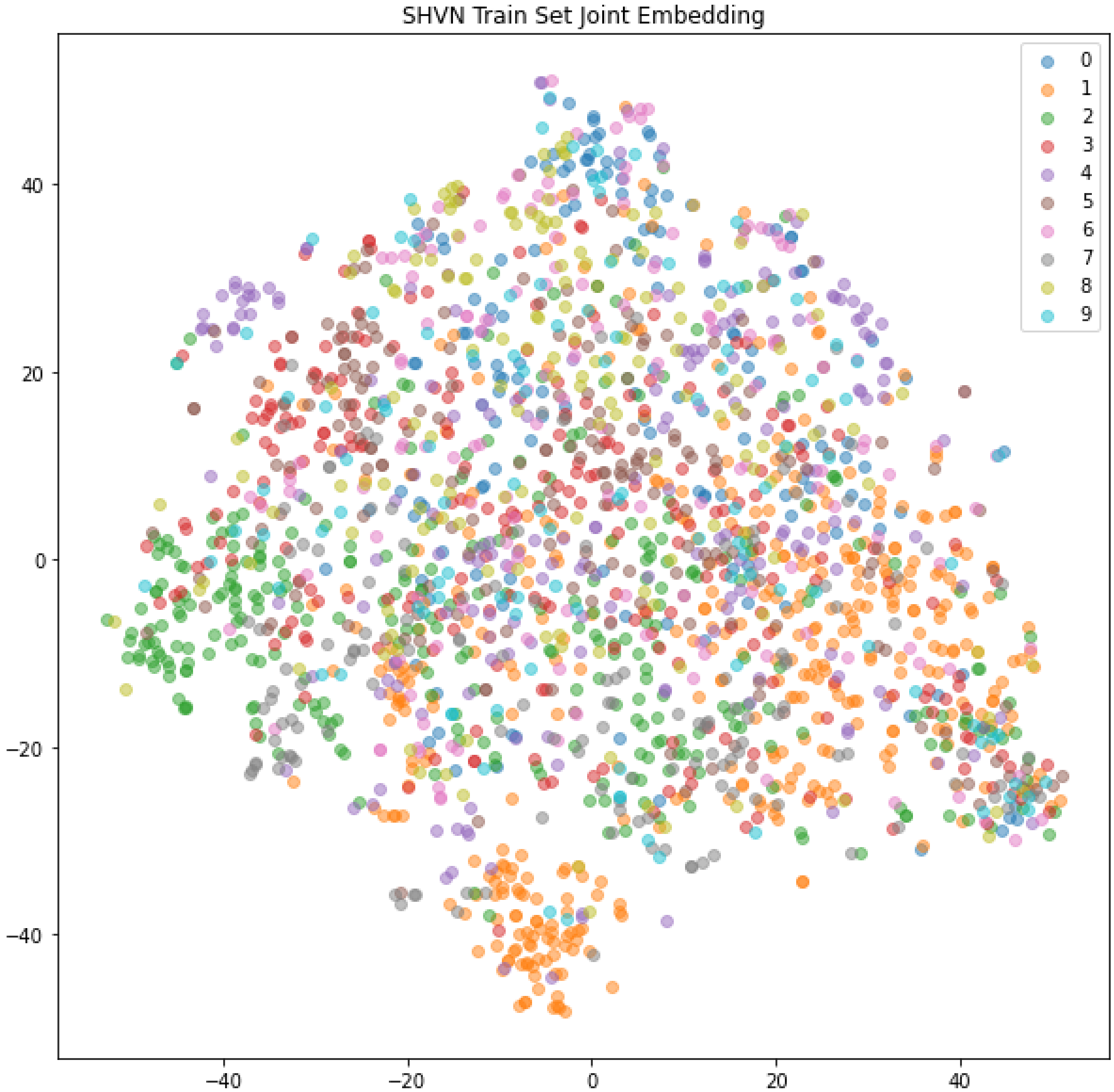
Visualization of the retrieval

In [357…

```
import matplotlib
import matplotlib.pyplot as plt

mnist_classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728',
              '#9467bd', '#8c564b', '#e377c2', '#7f7f7f',
              '#bcbd22', '#17becf']

def plot_embeddings(embeddings, targets, xlim=None, ylim=None,title=""):
    plt.figure(figsize=(10,10))
    for i in range(10):
        inds = np.where(targets==i)[0]
        # print(inds)
        plt.scatter(embeddings[inds,0], embeddings[inds,1], alpha=0.5, color=colors[i])
    if xlim:
        plt.xlim(xlim[0], xlim[1])
    if ylim:
        plt.ylim(ylim[0], ylim[1])
    plt.legend(mnist_classes)
    plt.title(title)
```

In [358…

```
import numpy as np
from sklearn.manifold import TSNE
train_embs=[]
train_labels=[]

for batch in svhn_embds:
  embds=batch[0]
  labels=batch[1]
  for idx in range(len(labels)):
    # print(labels.shape)
    train_embs.append(np.array(embds[idx].detach()))
    train_labels.append(labels[idx].item())
print(train_labels)
X_embedded = TSNE(n_components=2).fit_transform(train_embs)
X_embedded.shape
plot_embeddings(X_embedded,np.array(train_labels),title="SHVN Train Set Joint Embedding
```



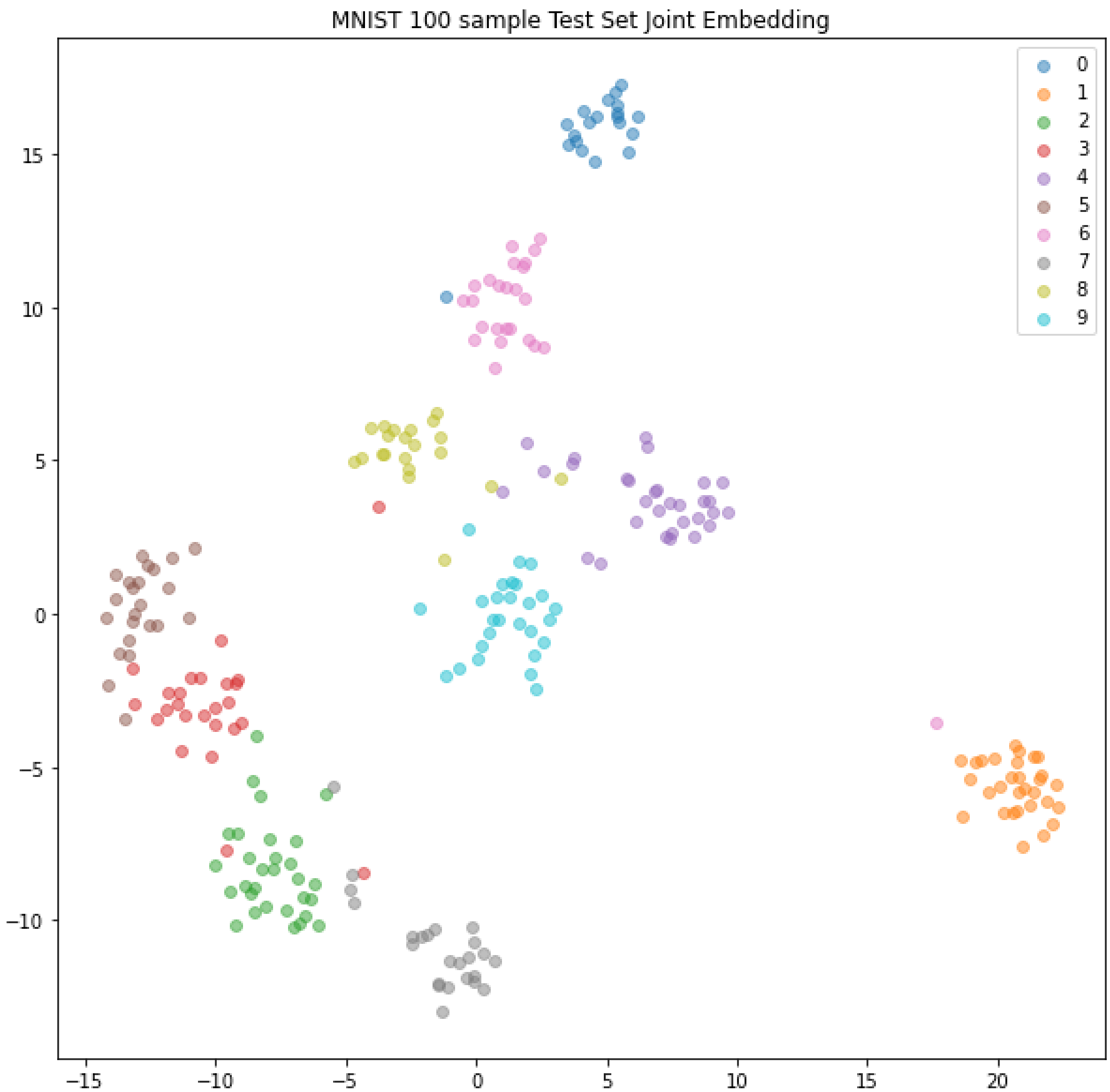
In [359…

```
print(len(svhn_embds[0][1]))
```

256

In [360…

```
import numpy as np
from sklearn.manifold import TSNE
test_embs=[]
test_labels=[]

for elem in nearestM:
  test_embs.append(np.array(elem[0].detach()))
  test_labels.append(elem[1].detach().item())
print(test_labels)
X_embedded = TSNE(n_components=2).fit_transform(test_embs)
X_embedded.shape
plot_embeddings(X_embedded,np.array(test_labels),title="MNIST 100 sample Test Set Joint
```



If you run into memory issues you can move your model to CPU to process the SVHN encodings.

Question Grading If you have trouble getting this to work you may still get partial credit for appropriate methodology. Grading for this question will be as follows:

10 points - appropriate triplet construction and loss function construction

10 points - appropriate nearest neighbor classification evaluation setup

10 points - obtaining above 50% accuracy, 5 points for getting above 25%

5 points (extra credit) - visualization of the retrieval

5 points (extra credit) - hard negative mining

You can include your answer in a separate notebook or .py file

Assign03.2.B

https://www.ruoyi.me/files/dynamics.pdf

$$\theta_g^{k+1} = \theta_g^k - \gamma \nabla_{\theta_g} V(\theta_g^k, \theta_d^k)$$
$$\theta_d^{k+1} = \theta_d^k + \gamma \nabla_{\theta_d} V(\theta_g^k, \theta_d^k)$$

Let $\gamma \to 0$, we obtain the gradient flow:

$$\dot{\theta} = -v(\theta)$$

where v is defined as the following vector field:

$$v(\theta_g, \theta_d) = \begin{bmatrix} \nabla_{\theta_g} V(\theta_g, \theta_d) \\ -\nabla_{\theta_d} V(\theta_g, \theta_d) \end{bmatrix}$$

the Jacobian matrix of vector field v:

$$v'(\theta_g, \theta_d) = \begin{bmatrix} \nabla^2_{\theta_g} V(\theta_g, \theta_d) & \nabla^2_{\theta_g, \theta_d} V(\theta_g, \theta_d) \\ -\nabla^2_{\theta_d, \theta_g} V(\theta_g, \theta_d) & -\nabla^2_{\theta_d} V(\theta_g, \theta_d) \end{bmatrix}$$

Assign.03.2.C

If discriminator is under-trained, it guides the generator in the wrong direction

If discriminator is over-trained, it is too "hard" and generator can't make progress

If generator trains too quickly it will "overshoot" the loss that the discriminator learned

Stationary points are where the gradient of each player w.r.t. its own parameters is 0

A equilibrium in this system is then a point where $v(\theta_g^*, \theta_d^*) = 0.$

Assign.03.2.D

The convergence properties of this game is determined by the eigenvalues of the Jacobian of the vector field:

if the absolute values of the eigenvalues of the Jacobian are all smaller than 1, then the fixed-point iteration converges to the fixed point

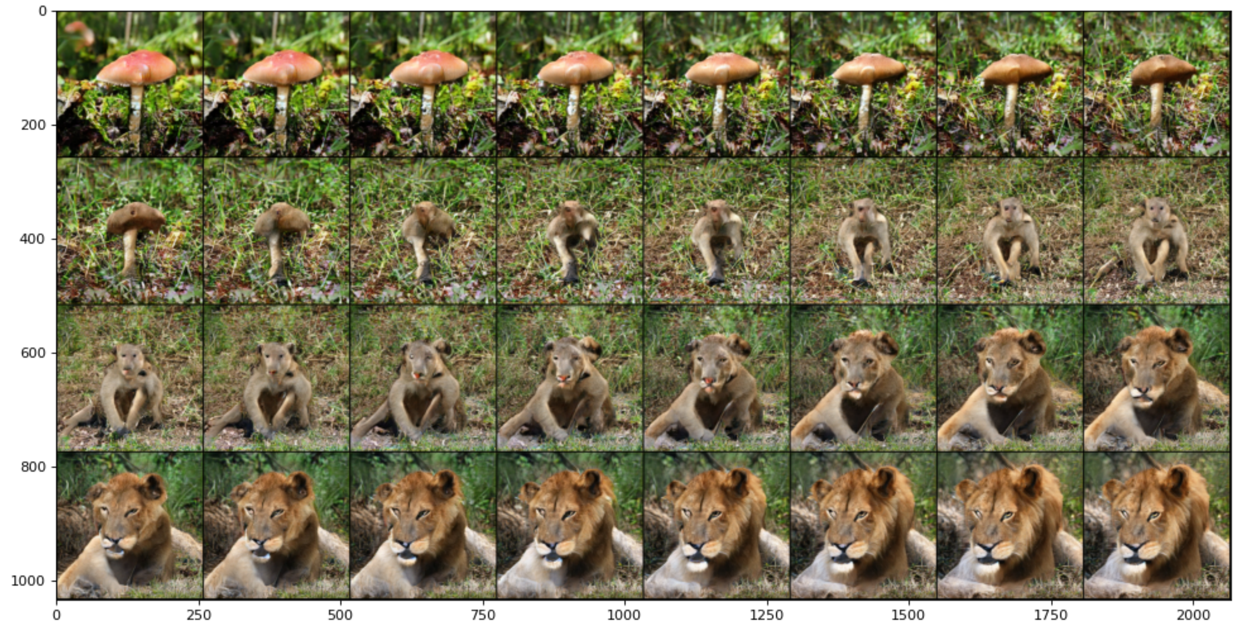
the convergence of GAN algorithms suffers due to two factors:

• presence of eigenvalues of the Jacobian of the gradient vector field with zero real-part.

• eigenvalues with big imaginary part.

when k goes to infinity the model suffer from oscillations as generator overshoots discriminator

Assign3.3.A



In [ ]:

```

```