prjtct_1

October 6, 2020

```
In [53]: from IPython.core.display import HTML, display
         #display(HTML("<style>.container{width:100%}; </style>"))
```

```
In [2]: import pandas as pd
        import tensorflow as tf
        import numpy as np
        from sklearn.manifold import TSNE
        from sklearn.model_selection import train_test_split,KFold,validation_curve
        from sklearn.svm import SVC
        from sklearn import metrics
        from scipy import io
        import torchvision.models as models
        import torch.nn as nn
        import torch.nn.functional as F
        import torch
        import matplotlib.pyplot as plt
        from tqdm import tqdm_notebook
        import random
        from itertools import combinations
        from torch.utils.data import DataLoader,Dataset

        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        seed = 1997

        random.seed(seed)
        torch.manual_seed(seed)
        np.random.seed(seed)
```

1 Experiment details

This notebook will compare the feature extracted from pre-trained ResNet18 with the features extracted from Scatter Net. SVM will be used as a classifier with different kernels (e.g., linear, RBF, and Polynomial) on the features extracted from the networks discussed above. Lastly, fine-tunning of the last layer of ResNet18 over MNIST dataset to get 99.8% or above accuracy, to see when the network completely overfits the training data how the extracted features look like.

```
In [3]: #(x_train,y_train),(x_test,y_test) = tf.keras.datasets.mnist.load_data()
        mnist = np.load('./data/mnist.npz',allow_pickle=True)
```

```
x_train = mnist['x_train']
y_train = mnist['y_train']
x_test = mnist['x_test']
y_test = mnist['y_test']

EPOCHS = 25;
N = 500;
batch_size = 128
num_classes = 10;
classes = np.asarray([k for k in range(num_classes)]).reshape(1,-1);
# def func(idx):
#     a, = np.where(y_train==idx)
#     return a[0:N];
#class_indx = np.apply_along_axis(func, 0, classes).reshape(-1)
#class_indx = class_indx[np.random.permutation(np.arange(0,N*num_classes))]
class_indx = np.load('./data/class_indx.npy')

x_train = x_train[class_indx]
y_train = y_train[class_indx]
del class_indx
```

2 Visualizing the data using t-SNE

```
In [4]: ftr = x_train.copy();
        ftr = ftr.reshape(num_classes*N, -1)/255;
        ftr = (ftr - ftr.min())/(ftr.max() - ftr.min())
        tsne = TSNE(n_components=2).fit_transform(ftr)
        tsne = np.apply_along_axis(lambda x: (x - x.min())/(x.max() - x.min()), 0, tsne)
```

```
In [5]: fig = plt.figure(figsize=(20,10))
        ax = fig.add_subplot(111)
        for c in range(num_classes):
            indices = [i for i,l in enumerate(y_train) if l==c]
            tx = np.take(tsne[:,0],indices)
            ty = np.take(tsne[:,1],indices)
            ax.scatter(tx,ty)

        ax.legend([i for i in range(num_classes)])
        fig.text(0.5,0.05,'Figure. 1: t-SNE Visualization of Raw data',ha = 'center',size=14)
        plt.show()
```

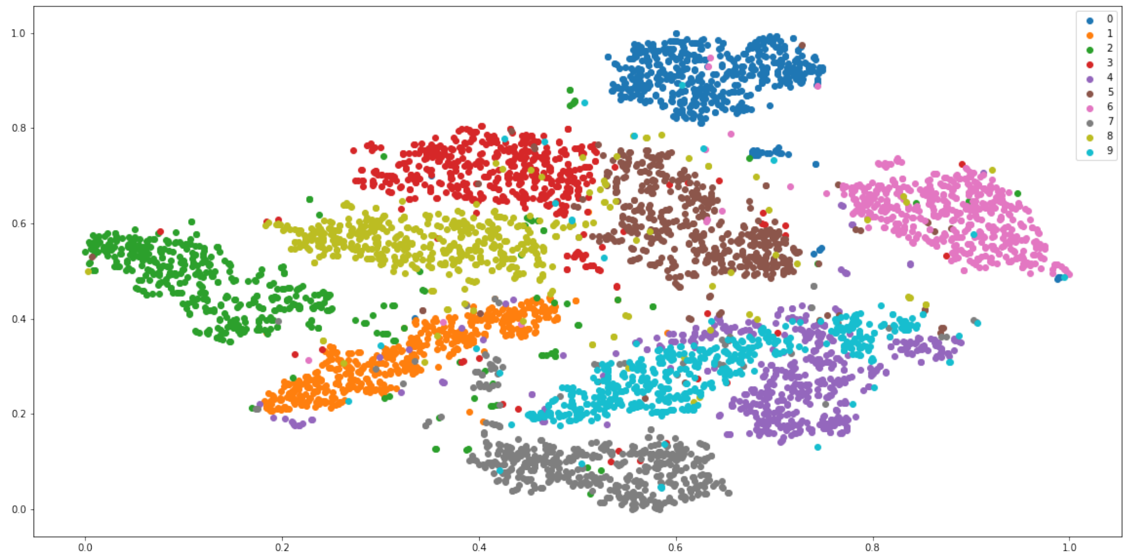

Figure. 1: t-SNE Visualization of Raw data

The above plot shows that different classes are overlapping. This overlapping suggests that a linear classifier may not be of choice. Later, we shall see that features extracted after fine-tuning of the ResNet18 on this data will produce the entirely disjoint features, and we can easily use a linear classifier to classify the data.

3 Feature Extraction Using Pre-trained ResNet18 Network

I am using ResNet18 pre-trained on the ImageNet dataset to extract the features of the MNIST dataset. I will be using t-SNE to visualize the feature. As the images in ImageNet are 3-channels, and we have grayscale images for the MNIST dataset. So, there are two ways to solve this problem 1. Replace the first convolutional layer of the ResNet18 by a custom layer. 2. Replicate the grayscale image to 3 channels to mimic the RGB So, I'll consider option 1 for now. I have made the dataset balanced over all classes. For each class, I am selecting only 500 samples. Because for a larger sample size, I am unable to compute the t-SNE for visualization. As the length of the feature vector of the ResNet18 is 512. We can use different techniques to get this feature vector, but I am using Pytorch 'register_forward_hook' function on the FC layer to get ResNet's features.

```
In [6]: class MNIST_dataset(Dataset):
            def __init__(self, x_train,y_train,x_test,y_test, tp = 'train', rgb=True):

                self.x_train = x_train.copy();
                self.y_train = y_train.copy();
                self.x_test = x_test.copy();
                self.y_test = y_test.copy();
                self.rgb = rgb

                self.type = tp
            def __len__(self):
```

```
            if self.type=='train':
                return len(self.x_train);
            elif self.type=='test':
                return len(self.x_test)
        def __getitem__(self,idx):
            if self.type=='train':
                img = self.x_train[idx]
                y = self.y_train[idx]
            elif self.type=='test':
                img = self.x_test[idx]
                y = self.y_test[idx]
            img = (img - img.mean())/(img.std() + 1e-9)
            if self.rgb:
                img = np.repeat(img[:,:,np.newaxis], 3, axis=2)
                img = img.transpose([2,0,1])
            else:
                img = img.reshape(-1, img.shape[0], img.shape[1])

            img = img.astype('float32')
            y = y.astype('int64')

            return img,y
```

4 Loading Dataset and extracting features

```
In [7]: batch_size = 128
        rgb = False;
        mnist_dataset = MNIST_dataset(x_train, y_train, x_test, y_test, tp = 'train', rgb=rgb)
        mnist_dataloader_train = DataLoader(mnist_dataset, batch_size=batch_size)
        resnet = models.resnet18(pretrained=True)
        # These two lines can be used when we need fine tuning....
        if not rgb:
            resnet.conv1 = nn.Conv2d(1,64,kernel_size=7,stride=1,padding=3,bias=False)
        #resnet.fc = nn.Linear(resnet.fc.in_features, num_classes)
        resnet.eval()
        resnet.to(device)
```

```
Out[7]: ResNet(
          (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3), bias=False
          (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
          (relu): ReLU(inplace=True)
          (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False
          (layer1): Sequential(
            (0): BasicBlock(
              (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
              (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
              (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

```
In [8]: def get_features(self, inpu, output):
            return inpu[0]
        resnet.fc.register_forward_hook(get_features)
        features = None;
        targets = []
        for idx_ , (imgs,labels) in enumerate(mnist_dataloader_train):

            imgs = imgs.to(device)
            cur_feature = resnet(imgs)
            cur_feature = cur_feature.detach().cpu().numpy()
            targets+=list(labels.cpu().numpy())
            if features is not None:
                features = np.concatenate((features, cur_feature))
            else:
                features = cur_feature
```

4.0.1 Visualizing Features using t-SNE

```
In [9]: tsne = TSNE(n_components=2).fit_transform(features)
        tsne = np.apply_along_axis(lambda x: (x - x.min())/(x.max() - x.min()), 0, tsne)
```

```
In [10]: fig = plt.figure(figsize=(20,10))
```

```
ax = fig.add_subplot(111)
for c in range(num_classes):
    indices = [i for i,l in enumerate(targets) if l==c]
    tx = np.take(tsne[:,0],indices)
    ty = np.take(tsne[:,1],indices)
    ax.scatter(tx,ty)

ax.legend([i for i in range(num_classes)])
fig.text(0.5,0.05,'Figure. 2: t-SNE Visualization of features extracted from pretraine
plt.show()
```

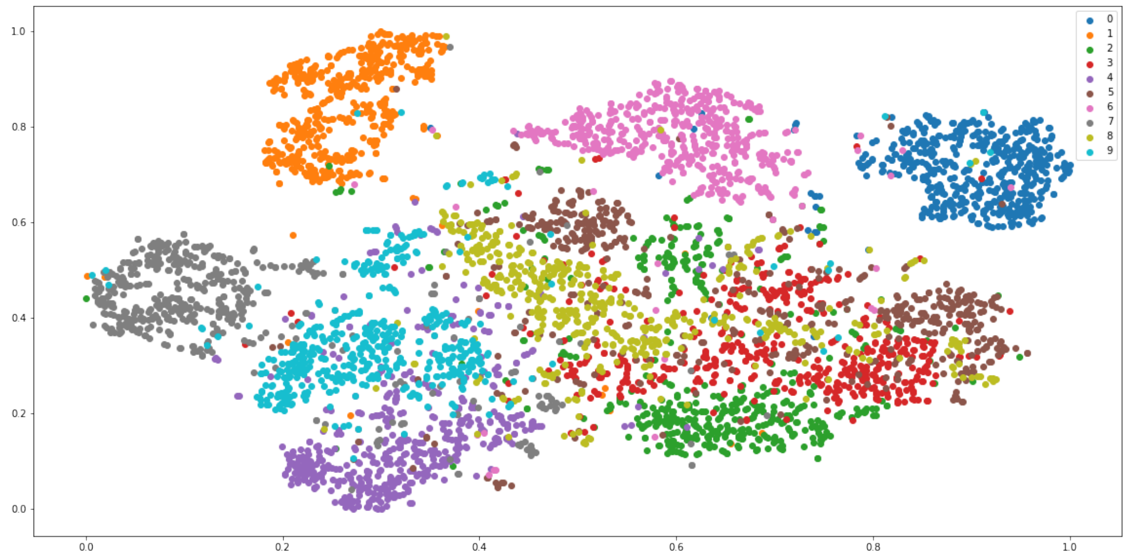

Figure. 2: t-SNE Visualization of features extracted from pretrained ResNet18.

From the above figure, we can see that different classes' features overlap but less overlapped compared to Fig. 1. This overlapping is making classification harder because we have a linear classifier working on these features. Later, we shall see that we had very few overlapping features between classes after fine-tuning the network, suggesting that a linear classifier can do the job.

5 Computing the Global Mean & Class mean

```
In [11]: G_mean = features.mean(axis=0)
         def compute_class_mean(x):
             y, = np.where(np.asarray(targets)==x)
             u = features[y,:]
             return u.mean(axis=0)

         C_mean = np.apply_along_axis(compute_class_mean, 0, classes).T
```

5.1 Computing Total Covariance Matrix

```
In [12]: T_cov = features - G_mean
         T_cov = np.dot(T_cov.T, T_cov)/(N*num_classes)
```

5.2 Computing Between Class Covariance Matrix

```
In [13]: B_cov = C_mean - G_mean
         B_cov = np.dot(B_cov.T,B_cov)/num_classes
```

5.3 Within Class Covariance Matrix

```
In [14]: def compute_class_mean(x):
             y, = np.where(np.asarray(targets)==x)
             u = features[y,:] - C_mean[x]

             return np.dot(u.T,u)

         W_cov = np.apply_along_axis(compute_class_mean, 0, classes)
         W_cov = W_cov.sum(axis=-1)/(N*num_classes)
```

5.4 NC1

```
In [15]: nc1 = np.trace(np.matmul(W_cov,np.linalg.pinv(B_cov)))/num_classes
         print(f"The Contraction within class: {nc1}")
```

```
The Contraction within class: 151223731.2
```

5.5 Equal-Norms of Class Means

```
In [16]: p_resnet_q = []
         norms = np.linalg.norm(C_mean - G_mean,axis=1,ord=2)
         p_resnet_q.append(norms.std()/norms.mean())
         print(f"Closeness to equal-norms of class-means: {p_resnet_q[-1]}")
```

```
Closeness to equal-norms of class-means: 0.4398036003112793
```

5.6 Equal-angularity

```
In [17]: combs = np.asarray(list(combinations(range(10),2)))
         def compute_equal_anularity(x):
             c,cp = x[0],x[1]
             mu_c = C_mean[c,:] - G_mean
             mu_cp = C_mean[cp, :] - G_mean
             f_val = np.dot(mu_c, mu_cp)/(np.linalg.norm(mu_c) * np.linalg.norm(mu_cp))
             return f_val
         equal_angl = np.apply_along_axis(compute_equal_anularity, 1, combs)
```

```
        p_resnet_q.append(equal_angl.std())
        print(f"Equal-angularity: {p_resnet_q[-1]}")
```

```
Equal-angularity: 0.31044915318489075
```

5.7 Maximal-angle equiangularity

```
In [18]: m_q_angl = np.abs(equal_angl + 1/(num_classes-1)).mean()
         p_resnet_q.append(m_q_angl)
         print(f"Maximal-angle equiangularity: {p_resnet_q[-1]}")
```

```
Maximal-angle equiangularity: 0.25803887844085693
```

6 Image Classification (using SVM) based on the features obtained above

```
In [19]: trainx,testx,trainy,testy = train_test_split(features,np.asarray(targets),test_size=0
```

6.0.1 In following cells we applied SVM with linear kernel to classify.

```
In [20]: model_ = SVC(kernel='linear')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [21]: print(f"Accuracy of SVM using Linear Kernel: {metrics.accuracy_score(testy, predy)*10
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using Linear Kernel: 92.80000000000001
Confusion Matrix:
 [[104   0   0   0   0   0   0   0   0   1]
 [  0 102   2   0   0   0   0   0   0   0]
 [  0   0 102   2   1   2   1   1   0   0]
 [  0   0   7  83   0   5   0   0   1   0]
 [  0   0   1   0  92   0   0   1   3   5]
 [  0   0   0   5   0  70   1   0   1   0]
 [  0   0   0   1   0   1  99   0   0   0]
 [  0   2   0   1   2   0   0  81   0   2]
 [  0   0   4   1   1   6   0   0 101   2]
 [  0   0   1   0   5   1   0   1   1  94]]
```

6.0.2 Now we shall use the non-linear kernel for SVM instead of linear one to see the if we can get the gain in accuracy

```
In [22]: model_ = SVC(kernel='rbf')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [23]: print(f"Accuracy of SVM using RBF kernel: {metrics.accuracy_score(testy, predy)*100}"
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using RBF kernel: 92.9
Confusion Matrix:
 [[104   0   0   0   0   0   0   0   1   0]
 [  0 101   2   0   0   0   0   0   1   0]
 [  0   0  98   5   0   3   2   1   0   0]
 [  0   0   5  86   0   2   0   0   2   1]
 [  0   0   1   0  94   0   0   1   0   6]
 [  0   0   1   7   0  66   1   1   1   0]
 [  0   0   0   0   1   2  98   0   0   0]
 [  0   1   1   0   2   0   0  82   0   2]
 [  0   0   2   1   1   5   0   0 106   0]
 [  0   0   0   1   5   0   0   1   2  94]]
```

```
In [24]: model_ = SVC(kernel='poly')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [25]: print(f"Accuracy of SVM using Polynomial kernel: {metrics.accuracy_score(testy, predy
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using Polynomial kernel: 93.7
Confusion Matrix:
 [[104   0   0   0   0   0   0   0   1   0]
 [  0 102   2   0   0   0   0   0   0   0]
 [  0   0 100   3   1   2   2   1   0   0]
 [  0   0   6  84   0   2   0   0   3   1]
 [  0   0   0   1  95   0   0   1   2   3]
 [  0   0   1   4   0  70   0   1   1   0]
 [  0   0   0   0   1   2  98   0   0   0]
 [  0   1   1   0   3   0   0  82   0   1]
 [  0   0   3   1   1   4   0   0 106   0]
 [  0   0   1   0   5   0   0   1   0  96]]
```

As, we can see that the there no big difference in the accuracy of different kernels.

7 Fine-Tuning of the ResNet18 on MNIST dataset

```
In [26]: resnet = models.resnet18(pretrained=True)
         if not rgb:
             resnet.conv1 = nn.Conv2d(1,64,kernel_size=7,stride=1,padding=3,bias=False)
         resnet.fc = nn.Linear(resnet.fc.in_features, num_classes)
         resnet.to(device)
         optimizer = torch.optim.SGD(resnet.parameters(), lr=5e-4,momentum=0.9)
```

```
In [27]: def compute_accuracy(model, data_loader, device):
             correct_pred, num_examples = 0, 0

             for i, (features, targets) in enumerate(data_loader):

                 features = features.to(device)
                 targets = targets.to(device)

                 probas = model(features)

                 probas = F.softmax(probas)


                 _, predicted_labels = torch.max(probas, 1)
                 num_examples += targets.size(0)
                 correct_pred += (predicted_labels == targets).sum()


             return correct_pred.float()/num_examples * 100

In [28]: for epc in range(EPOCHS):

             resnet.train();
             for a,b in  mnist_dataloader_train:
                 a = a.to(device)
                 out = resnet(a)
                 b = b.to(device)
                 loss = F.cross_entropy(out, b)

                 optimizer.zero_grad();

                 loss.backward();

                 optimizer.step();

             resnet.eval();
             with torch.set_grad_enabled(False):
                 acc = compute_accuracy(resnet,mnist_dataloader_train, device)

                 print(f"Epoch: {epc}, Training Acc: {acc}")
```

```
/home/murad/.local/lib/python3.6/site-packages/ipykernel_launcher.py:11: UserWarning: Implicit
  # This is added back by InteractiveShellApp.init_path()


Epoch: 0, Training Acc: 80.53999328613281
Epoch: 1, Training Acc: 92.5999984741211
```

```
Epoch: 2, Training Acc: 96.08000183105469
Epoch: 3, Training Acc: 97.63999938964844
Epoch: 4, Training Acc: 98.55999755859375
Epoch: 5, Training Acc: 98.95999908447266
Epoch: 6, Training Acc: 99.25999450683594
Epoch: 7, Training Acc: 99.55999755859375
Epoch: 8, Training Acc: 99.7199935913086
Epoch: 9, Training Acc: 99.79999542236328
Epoch: 10, Training Acc: 99.85999298095703
Epoch: 11, Training Acc: 99.87999725341797
Epoch: 12, Training Acc: 99.89999389648438
Epoch: 13, Training Acc: 99.91999816894531
Epoch: 14, Training Acc: 99.93999481201172
Epoch: 15, Training Acc: 99.93999481201172
Epoch: 16, Training Acc: 99.93999481201172
Epoch: 17, Training Acc: 99.95999908447266
Epoch: 18, Training Acc: 99.95999908447266
Epoch: 19, Training Acc: 99.95999908447266
Epoch: 20, Training Acc: 99.97999572753906
Epoch: 21, Training Acc: 99.97999572753906
Epoch: 22, Training Acc: 99.97999572753906
Epoch: 23, Training Acc: 99.97999572753906
Epoch: 24, Training Acc: 99.97999572753906
```

7.1 Visualizing Features using t-SNE after fine-tuning with training error zero

```
In [29]: def get_features(self, inpu, output):
             return inpu[0]
         resnet.fc.register_forward_hook(get_features)
         features = None;
         targets = []
         for idx_ , (imgs,labels) in enumerate(mnist_dataloader_train):

             imgs = imgs.to(device)
             cur_feature = resnet(imgs)
             cur_feature = cur_feature.detach().cpu().numpy()
             targets+=list(labels.cpu().numpy())
             if features is not None:
                 features = np.concatenate((features, cur_feature))
             else:
                 features = cur_feature
```

```
In [30]: tsne = TSNE(n_components=2).fit_transform(features)
         tsne = np.apply_along_axis(lambda x: (x - x.min())/(x.max() - x.min()), 0, tsne)
```

```
In [31]: fig = plt.figure(figsize=(20,10))
         ax = fig.add_subplot(111)
```

```
for c in range(num_classes):
    indices = [i for i,l in enumerate(targets) if l==c]
    tx = np.take(tsne[:,0],indices)
    ty = np.take(tsne[:,1],indices)
    ax.scatter(tx,ty)
ax.legend([i for i in range(num_classes)])
fig.text(0.5,0.05,'Figure. 3: t-SNE Visualization of features extracted after fine-tu
plt.show()
```

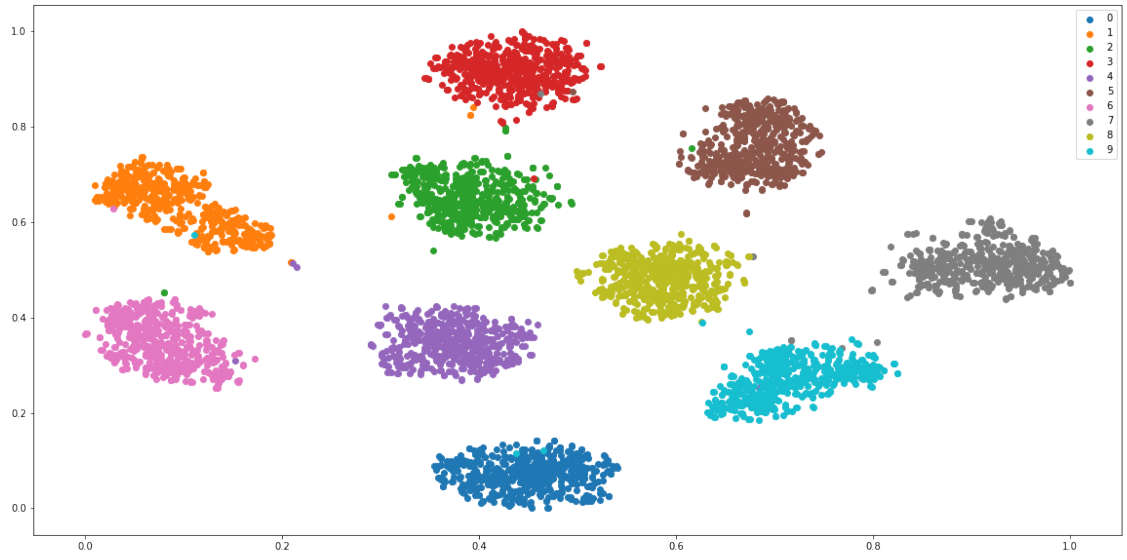

Figure. 3: t-SNE Visualization of features extracted after fine-tuning of MNIST data.

8 Computing Class means after fine tuning

```
In [32]: G_mean = features.mean(axis=0)
         def compute_class_mean(x):
             y, = np.where(np.asarray(targets)==x)
             u = features[y,:]
             return u.mean(axis=0)

         C_mean = np.apply_along_axis(compute_class_mean, 0, classes).T
         T_cov = features - G_mean
         T_cov = np.dot(T_cov.T, T_cov)/(N*num_classes)
         B_cov = C_mean - G_mean
         B_cov = np.dot(B_cov.T,B_cov)/num_classes
         def compute_class_mean(x):
             y, = np.where(np.asarray(targets)==x)
             u = features[y,:] - C_mean[x]

             return np.dot(u.T,u)
```

```
        W_cov = np.apply_along_axis(compute_class_mean, 0, classes)
        W_cov = W_cov.sum(axis=-1)/(N*num_classes)

        nc1 = np.trace(np.matmul(W_cov,np.linalg.pinv(B_cov)))/num_classes
        print(f"The Contraction within class: {nc1}")
        f_resnet_q = []
        norms = np.linalg.norm(C_mean - G_mean,axis=1,ord=2)
        f_resnet_q.append(norms.std()/norms.mean())
        print(f"Closeness to equal-norms of class-means: {f_resnet_q[-1]}")
        combs = np.asarray(list(combinations(range(10),2)))
        def compute_equal_anularity(x):
            c,cp = x[0],x[1]
            mu_c = C_mean[c,:] - G_mean
            mu_cp = C_mean[cp, :] - G_mean
            f_val = np.dot(mu_c, mu_cp)/(np.linalg.norm(mu_c) * np.linalg.norm(mu_cp))
            return f_val
        equal_angl = np.apply_along_axis(compute_equal_anularity, 1, combs)
        f_resnet_q.append(equal_angl.std())
        print(f"Equal-angularity: {f_resnet_q[-1]}")
        m_q_angl = np.abs(equal_angl + 1/(num_classes-1)).mean()
        f_resnet_q.append(m_q_angl)
        print(f"Maximal-angle equiangularity: {f_resnet_q[-1]}")
```

```
The Contraction within class: -93557068.8
Closeness to equal-norms of class-means: 0.039259172976017
Equal-angularity: 0.08577156066894531
Maximal-angle equiangularity: 0.07403488457202911
```

9 The following section is based on the features extracted from Scatter Net.

The Matlab toolbox of scattering net was used to extract the same samples' features, which we have analyzed under ResNet18.

```
In [33]: features = io.loadmat('./data/scat_net_features.mat')
         features = features['fetr']
```

```
In [34]: tsne = TSNE(n_components=2).fit_transform(features)
         tsne = np.apply_along_axis(lambda x: (x - x.min())/(x.max() - x.min()), 0, tsne)
```

```
In [35]: fig = plt.figure(figsize=(20,10))
         ax = fig.add_subplot(111)
         for c in range(num_classes):
             indices = [i for i,l in enumerate(y_train) if l==c]
             tx = np.take(tsne[:,0],indices)
             ty = np.take(tsne[:,1],indices)
```

```
        ax.scatter(tx,ty)
    ax.legend([i for i in range(num_classes)])
    fig.text(0.5,0.05,'Figure. 4: t-SNE Visualization of features extracted using Scatter
    plt.show()
```

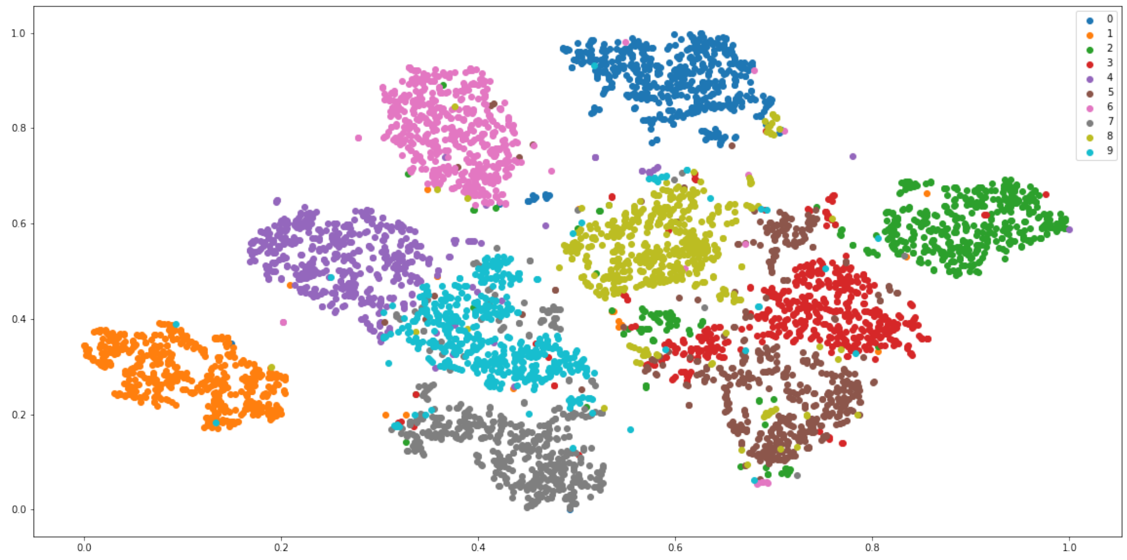

Figure. 4: t-SNE Visualization of features extracted using Scatter Net.

9.1 Computing the Global & Class Means

```
In [36]: G_mean = features.mean(axis=0)
         def compute_class_mean(x):
             y, = np.where(np.asarray(y_train)==x)
             u = features[y,:]
             return u.mean(axis=0)

         C_mean = np.apply_along_axis(compute_class_mean, 0, classes).T
```

9.2 Computing Total Covariance

```
In [37]: T_cov = features - G_mean
         T_cov = np.dot(T_cov.T, T_cov)/(N*num_classes)
```

9.3 Computing Between Covariance

```
In [38]: B_cov = C_mean - G_mean
         B_cov = np.dot(B_cov.T,B_cov)/num_classes
```

9.4 Computing Within class covariance Matrix

```
In [39]: def compute_class_mean(x):
             y, = np.where(np.asarray(y_train)==x)
```

```
            u = features[y,:] - C_mean[x]

            return np.dot(u.T,u)

        W_cov = np.apply_along_axis(compute_class_mean, 0, classes)
        W_cov = W_cov.sum(axis=-1)/(N*num_classes)
```

9.5 NC1

```
In [40]: nc1 = np.trace(np.matmul(W_cov,np.linalg.pinv(B_cov)))/num_classes
         print(f"The Contraction within class: {nc1}")
```

```
The Contraction within class: -108916006.4
```

9.6 Equal Norms of Class Means

```
In [41]: norms = np.linalg.norm(C_mean - G_mean,axis=1,ord=2)
         p_snet_q = list();
         p_snet_q.append(norms.std()/norms.mean())
         print(f"Closeness to equal-norms of class-means: {p_snet_q[-1]}")
```

```
Closeness to equal-norms of class-means: 0.34719786047935486
```

9.7 Equal angularity

```
In [42]: combs = np.asarray(list(combinations(range(10),2)))
         def compute_equal_anularity(x):
             c,cp = x[0],x[1]
             mu_c = C_mean[c,:] - G_mean
             mu_cp = C_mean[cp, :] - G_mean
             f_val = np.dot(mu_c, mu_cp)/(np.linalg.norm(mu_c) * np.linalg.norm(mu_cp))
             return f_val
         equal_angl = np.apply_along_axis(compute_equal_anularity, 1, combs)
         p_snet_q.append(equal_angl.std())
         print(f"Equal-angularity: {p_snet_q[-1]}")
```

```
Equal-angularity: 0.36474093794822693
```

9.8 Maximal angle equiangularity

```
In [43]: m_q_angl = np.abs(equal_angl + 1/(num_classes-1)).mean()
         p_snet_q.append(m_q_angl)
         print(f"Maximal-angle equiangularity: {p_snet_q[-1]}")
```

```
Maximal-angle equiangularity: 0.3195512890815735
```

10 Training SVM on Features extracted using Scatter Net

```
In [44]: trainx,testx,trainy,testy = train_test_split(features,np.asarray(y_train),test_size=0
```

```
In [45]: model_ = SVC(kernel='linear')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [46]: print(f"Accuracy of SVM using linear kernel: {metrics.accuracy_score(testy, predy)*10
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using linear kernel: 97.89999999999999
Confusion Matrix:
 [[107   0   0   0   0   0   0   0   0   0]
 [  0 103   0   0   0   0   0   1   0   0]
 [  1   0  90   1   0   0   0   0   0   0]
 [  0   0   0  98   0   0   0   0   0   0]
 [  0   0   0   0  87   0   0   0   0   0]
 [  0   0   0   0   0  95   0   0   1   0]
 [  1   0   1   0   1   1  92   0   0   0]
 [  0   1   0   0   0   0   0  95   0   0]
 [  1   2   0   1   0   0   0   1 100   0]
 [  0   1   0   0   1   1   1   2   1 112]]
```

```
In [47]: model_ = SVC(kernel='rbf')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [48]: print(f"Accuracy of SVM using RBF kernel: {metrics.accuracy_score(testy, predy)*100}"
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using RBF kernel: 97.39999999999999
Confusion Matrix:
 [[106   0   0   0   0   0   1   0   0   0]
 [  0 103   0   0   0   0   0   1   0   0]
 [  0   0  92   0   0   0   0   0   0   0]
 [  0   0   0  96   0   0   0   1   0   1]
 [  0   0   0   0  87   0   0   0   0   0]
 [  0   0   0   0   0  93   1   0   2   0]
 [  1   0   0   0   2   1  91   0   1   0]
 [  0   0   1   0   0   0   0  93   0   2]
 [  0   0   0   1   1   0   0   0 103   0]
 [  0   1   0   2   1   0   1   2   2 110]]
```

```
In [49]: model_ = SVC(kernel='poly')
         model_.fit(trainx,trainy)
         predy = model_.predict(testx)
```

```
In [50]: print(f"Accuracy of SVM using Polynomial kernel: {metrics.accuracy_score(testy, predy
         print(f"Confusion Matrix:\n {metrics.confusion_matrix(testy,predy)}")
```

```
Accuracy of SVM using Polynomial kernel: 96.39999999999999
Confusion Matrix:
 [[105   0   0   0   0   1   1   0   0   0]
 [  0 103   0   0   0   0   0   1   0   0]
 [  0   1  88   0   0   2   0   1   0   0]
 [  0   1   0  95   0   0   0   1   0   1]
 [  0   0   0   0  87   0   0   0   0   0]
 [  0   0   0   0   0  93   1   0   2   0]
 [  1   4   0   0   2   1  87   0   1   0]
 [  0   0   1   0   0   0   0  93   0   2]
 [  0   1   0   2   1   0   0   0 101   0]
 [  0   1   0   0   2   0   0   2   2 112]]
```

11 Analysis

Comparison of Fine Tunned & Pre-trained ResNet18 Comparing Fig. 1 & Fig. 2 with Fig. 3 (after the fine-tuning), we can see that class has a noticeable distinction in different classes' features. While in Fig. 2, we can see that there are many classes whose features overlap with other classes. However, here the features are very much apart from each other.

Comparison of Scatter Net & ResNet18 SVM is performing better on features extracted from Scatter Net than the pre-trained ResNet18. The t-SNE visualization of features Fig. 2 & Fig. 4 can explain SVM's performance difference. There are very few classes that overlap in Fig. 4 (using Scatter Net) compared to Fig. 3 (using pre-trained ResNet18). We can conclude that scattering net gives better features on this subset of the MNIST dataset than pre-trained ResNet18.

```
In [51]: data_ = pd.DataFrame(np.vstack([p_resnet_q,f_resnet_q,p_snet_q]).T)
         data_.columns = ['Pre-trained ResNet', 'Fine-Tunned ResNet','Scattering Net']
         data_.index = ['eq-norms of class means', 'eq-angularity','max-angle equiangularity']

In [52]: data_
```

```
Out[52]:                           Pre-trained ResNet  Fine-Tunned ResNet  \
         eq-norms of class means             0.439804            0.039259
         eq-angularity                       0.310449            0.085772
         max-angle equiangularity            0.258039            0.074035

                                   Scattering Net
         eq-norms of class means         0.347198
         eq-angularity                   0.364741
         max-angle equiangularity        0.319551
```

12 Explaining Convergence of cosine

By looking at the above table, we can see that for Fine-tuned ResNet, when training error goes zero, all three quantities go to zero. While in the other two cases, these quantities are significantly away from zero. The convergence of maximal-angle equiangularity to zero shows that cosine similarity converges to -1/(num_classes -1). Fig. 3 also confirms that the classes are maximally apart after fine-tuning as training error goes to zero.

```
In [ ]:
```