AG_DLS_HW_1_PT_submission

October 3, 2021

**

Deep Learning Systems (ENGR-E 533) Homework 1 , Fall 2021

**

Name: Anitha Ganapathy Email: aganapa@iu.edu

0.1 Organizing Imports

[148]:
```
# Import Libraries

import torch
import torch.nn as nn

from torch import optim
from torch.utils.data import DataLoader


import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import MNIST

import numpy as np
import pandas as pd

from sklearn.manifold import TSNE
from sklearn.decomposition import PCA

import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

[149]:
```
print("Torch Version     : ",torch.__version__)
print("Torch Version     : ",torchvision.__version__)
print("Is CUDA available :",torch.cuda.is_available() )
```

```
Torch Version     :  1.9.0+cu102
Torch Version     :  0.10.0+cu102
Is CUDA available : True
```

[150]:
```
# !nvidia-smi
```

[151]:
```
print("torch.cuda.current_device()", torch.cuda.current_device())
print("torch.cuda.device_count() : ", torch.cuda.device_count())
print("torch.cuda.memory_allocated() :", torch.cuda.memory_allocated())

#  torch.cuda.memory_cached has been renamed to torch.cuda.memory_reserved
print("torch.cuda.memory_reserved()", torch.cuda.memory_reserved())
print()
```

```
torch.cuda.current_device() 0
torch.cuda.device_count() :  1
torch.cuda.memory_allocated() : 802576384
torch.cuda.memory_reserved() 1228931072
```

0.2 Downloading the MNIST dataset from torchvision using FastMNISt.

0.2.1 Defining the FastMNIST class.

[152]:
```
device = torch.device('cuda')

class FastMNIST(MNIST):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # Scale data to [0,1]
        self.data = self.data.unsqueeze(1).float().div(255)

        # Normalize it with the usual MNIST mean and std
        self.data = self.data.sub_(0.1307).div_(0.3081)

        # Put both data and targets on GPU in advance
        self.data, self.targets = self.data.to(device), self.targets.to(device)

    def __getitem__(self, index):
        """
        Args:
            index (int): Index

        Returns:
            tuple: (image, target) where target is index of the target class.
        """
        img, target = self.data[index], self.targets[index]
```

```
        return img, target
```

[153]:
```
# # Import MNIST dataset
# mnist_train = MNIST('data', train=True, download=True,
#                 transform = torchvision.transforms.Compose([
#                 torchvision.transforms.ToTensor(),
#                 torchvision.transforms.Normalize((0.1307,), (0.3081,))
#                 ]))

# mnist_test = MNIST('data', train=False, download=True,
#                 transform = torchvision.transforms.Compose([
#                 torchvision.transforms.ToTensor(),
#                 torchvision.transforms.Normalize((0.1307,), (0.3081,))
#                 ]))
```

0.3 Defining the Train and Test DataLoader for batch wise data fetch.

[154]:
```
# DataLoader wraps an iterable over our dataset, and supports automatic␣
↪batching, sampling,
# shuffling and multiprocess data loading.

# train_dataloader = DataLoader(mnist_train, batch_size = train_batch_size,
#                               shuffle= False)
# test_dataloader = DataLoader(mnist_test, batch_size = test_batch_size,
#                              shuffle= False)
# num_workers=0 is very important!

def mnist_dataloader(train_batch, test_batch):
  train_dataset = FastMNIST('data/MNIST', train=True, download=True)
  test_dataset = FastMNIST('data/MNIST', train=False, download=True)
  train_batch_size = train_batch
  test_batch_size = test_batch

  train_dataloader = DataLoader(train_dataset, batch_size= train_batch_size,
                              shuffle=True, num_workers=0)
  test_dataloader = DataLoader(test_dataset, batch_size= test_batch_size,
                             shuffle=False, num_workers=0)
  return train_dataloader, test_dataloader
```

[155]:
```
train_dataloader, test_dataloader = mnist_dataloader(64, 1000)

for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    break
```

```
Shape of X [N, C, H, W]:  torch.Size([1000, 1, 28, 28])
```

```
Shape of y:  torch.Size([1000]) torch.int64
```

[156]:
```python
#  Get cpu or gpu device for training.

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Using {} device".format(device))
```

```
Using cuda:0 device
```

0.4 Problem 1: A Detailed View to MNIST Classification [3 points]

0.4.1 1. Train a fully-connected net for MNIST classification.

It shouldbe with 5 hidden layers each of which is with 1024 hidden units. Feel free to use whatever techniques you learned in class. You should be able to get the test accuracy above 98%.

Defining the Neural network Sequential Model

[157]:
```python
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten =nn.Flatten()
        self.hl_0 = nn.Linear(28 * 28, 1024)  # input layer
        self.hl_1 = nn.Linear(1024, 1024)     # hidden 1
        self.hl_2 = nn.Linear(1024, 1024)     # hidden 2
        self.hl_3 = nn.Linear(1024, 1024)     # hidden 3
        self.hl_4 = nn.Linear(1024, 1024)     # hidden 4
        self.hl_5 = nn.Linear(1024, 10)       # hidden 5 / o/p

        for mod in self.modules():
          self.weight_initializer(mod)

    def weight_initializer(self, mod):
      if isinstance(mod , nn.Linear):
        torch.nn.init.xavier_uniform_(mod.weight.data)
        if mod.bias is not None:
          mod.bias.data.fill_(0.0)

    def forward(self, x):
        x = self.flatten(x)
        op_of_0 = torch.relu(self.hl_0(x))
        op_of_1 = torch.relu(self.hl_1(op_of_0))
        op_of_2 = torch.relu(self.hl_2(op_of_1))
        op_of_3 = torch.relu(self.hl_3(op_of_2))
        op_of_4 = torch.relu(self.hl_4(op_of_3))
        op_of_5 = self.hl_5(op_of_4)
        return op_of_0, op_of_1, op_of_2, op_of_3, op_of_4, op_of_5
```

```
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (hl_0): Linear(in_features=784, out_features=1024, bias=True)
  (hl_1): Linear(in_features=1024, out_features=1024, bias=True)
  (hl_2): Linear(in_features=1024, out_features=1024, bias=True)
  (hl_3): Linear(in_features=1024, out_features=1024, bias=True)
  (hl_4): Linear(in_features=1024, out_features=1024, bias=True)
  (hl_5): Linear(in_features=1024, out_features=10, bias=True)
)
```

Optimizing the Model Parameters

[158]:
```
# Defining the optimiser and loss function

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr= 1e-2)
```

In a single training loop, the model makes predictions on the training dataset (fed to it in batches), and backpropagates the prediction error to adjust the model's parameters.

0.4.2 Train and Test for problem 1.

[159]:
```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    train_loss = 0.0

    for i, data in enumerate(dataloader, 0):
        X, y = data[0], data[1]
        # b = X.size(0)
        # X = X.view(b , -1)
        X, y = X.to(device), y.to(device)

        # re initialize the gradients parameters
        optimizer.zero_grad()
        # Compute prediction error
        pred = model(X)[-1]
        # print("Shape pred:", pred.shape)
        # print("Shape y:", y.shape)
        # print("Anitha there")
        loss = loss_fn(pred, y)


        # Backpropagation
        loss.backward()
        optimizer.step()
```

```
        train_loss+= loss.item()
    return train_loss /len(dataloader.dataset)
```

We also check the model's performance against the test dataset to ensure it is learning.

[160]:
```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_acc, correct = 0, 0
    for i, data in enumerate(dataloader, 0):
      X, y = data[0], data[1]
      # b = X.size(0)
      # X = X.view(b , -1)
      X, y = X.to(device), y.to(device)

      model_output = model(X)[-1]
      pred = torch.argmax(torch.softmax(model_output, dim = 1), dim = 1)
      acc = torch.sum(pred == y)
      test_acc += acc.cpu().numpy()

    return test_acc/size * 100
```

Training the model for n epochs

[161]:
```
%%time

avg_test_acc = []
epochs = 300
for t in range(epochs):
    train_loss = 0.0

    # print(f"---------------------------------------------------------")
    train_loss = train(train_dataloader, model, loss_fn, optimizer)
    test_accuracy = test(test_dataloader, model, loss_fn)
    print(f"Epoch : {t}  train loss: {train_loss:>7f} \
     test Accuracy : {test_accuracy:>5f}")
    avg_test_acc.append(test_accuracy)
    if test_accuracy > 98:
      break

print("Average Test Accuracy = ", torch.tensor(avg_test_acc).mean())
print("Done!")
```

```
Epoch : 0  train loss: 0.007003      test Accuracy : 93.930000
Epoch : 1  train loss: 0.002718      test Accuracy : 95.660000
Epoch : 2  train loss: 0.001919      test Accuracy : 95.570000
Epoch : 3  train loss: 0.001455      test Accuracy : 94.230000
Epoch : 4  train loss: 0.001135      test Accuracy : 96.750000
Epoch : 5  train loss: 0.000929      test Accuracy : 97.220000
```

```
Epoch : 6  train loss: 0.000737      test Accuracy : 95.100000
Epoch : 7  train loss: 0.000601      test Accuracy : 97.490000
Epoch : 8  train loss: 0.000478      test Accuracy : 97.330000
Epoch : 9  train loss: 0.000377      test Accuracy : 97.440000
Epoch : 10  train loss: 0.000311      test Accuracy : 97.790000
Epoch : 11  train loss: 0.000237      test Accuracy : 97.850000
Epoch : 12  train loss: 0.000190      test Accuracy : 97.770000
Epoch : 13  train loss: 0.000147      test Accuracy : 97.870000
Epoch : 14  train loss: 0.000115      test Accuracy : 98.010000
Average Test Accuracy =  tensor(96.6673, dtype=torch.float64)
Done!
CPU times: user 34.6 s, sys: 1.32 s, total: 35.9 s
Wall time: 35.4 s
```

0.4.3 Capturing the output of the Softmax layer after feedforward training.

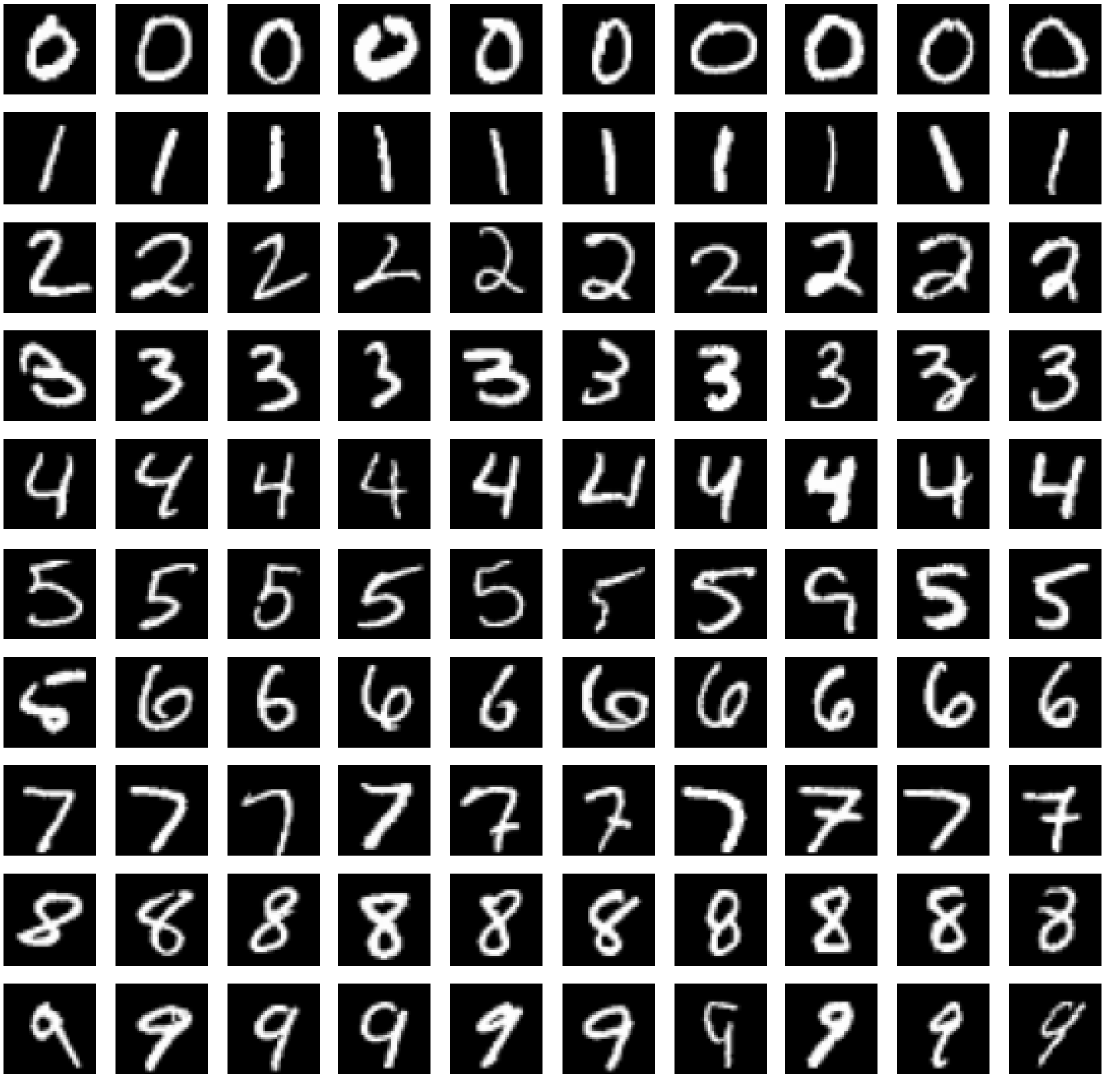
Once you're done with training, as a starter, do a feedforward step on your test samples, a thousand of them. Capture the output of the softmax layer, which will be a 10-dim probability vector per sample.

[162]:
```
def display_mnist_data(id, images, pred_labels):
  plt.figure(id, figsize=(10,10))
  for i in range(10):
    indexes = np.where(pred_labels == i)[0]
    for j in range(10):
      plt.subplot(10, 10, i*10+j+1)
      plt.rcParams['axes.facecolor']=='gray'
      if len(indexes) >j:
        # images = images.reshape((28, 28))
        image = images[indexes[j]].reshape(28,28)
        plt.imshow(image.cpu().numpy(), cmap = 'gray',interpolation = None)
        # print(images[indexes[j]].shape)
      plt.axis('off')
```

Plotting the images of the final layer.

[163]:
```
def fetch_test_batch():
  images, labels = next(iter(test_dataloader))
  return images.to(device), labels

def prediction_last_layer(layer_num = 5):
  model.eval()    # dont calculate the gradient
  images, labels = fetch_test_batch()
  images = images.to(device)
  y_pred = model(images)[layer_num]  # accessing the last layer of the model
  predictions = torch.argmax(torch.softmax(y_pred, dim=1), dim=1).cpu()
  display_mnist_data(0, images, predictions)
  return
```

[164]:
```
# making a prediction for the last layer for the test batch
prediction_last_layer(5)
```



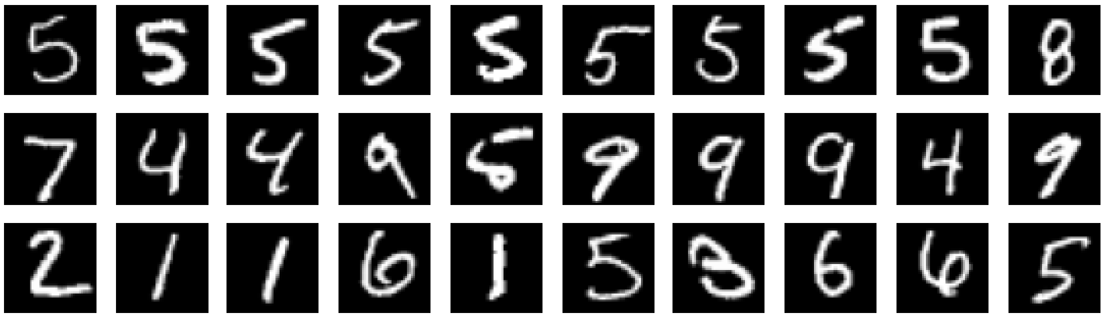
0.4.4 Repeat the procedure in Problem 1.3 for your second to the last layer output.

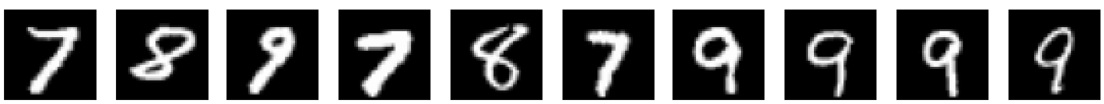
4th Layer

[165]:
```
# function definition for prediction of hidden layers
def predict_hidden_layers(layer_num):
  model.eval()
  random_choices = np.random.randint(1024, size=10)
  images, labels = fetch_test_batch()
  y_pred = model(images.to(device))[layer_num]
  predictions = y_pred[:, random_choices].argmax(axis=1).cpu()
```

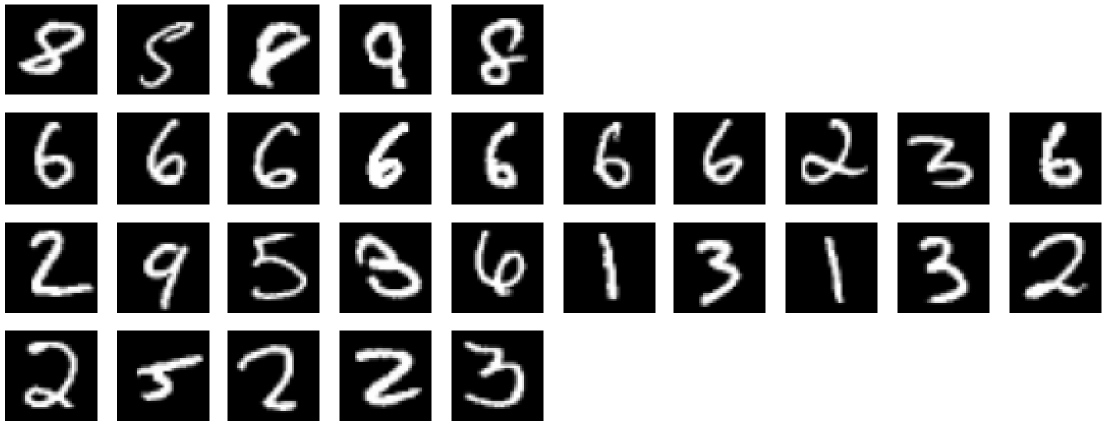
```
  display_mnist_data(0, images, predictions)
```

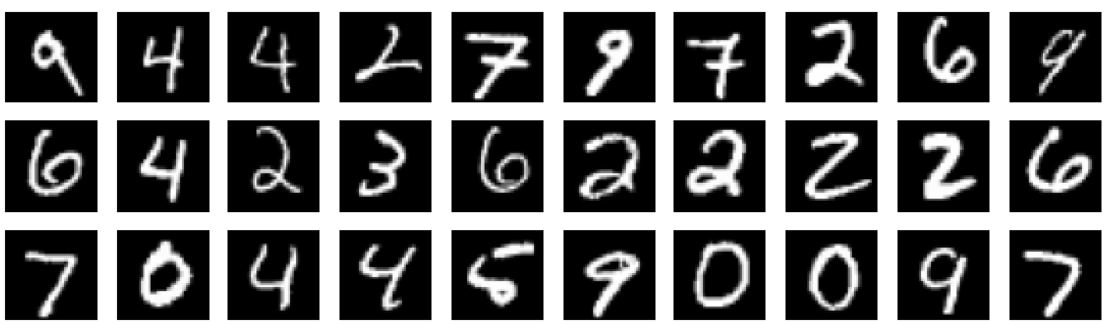
[166]:
```
# predict 4th layer
predict_hidden_layers(4)
```



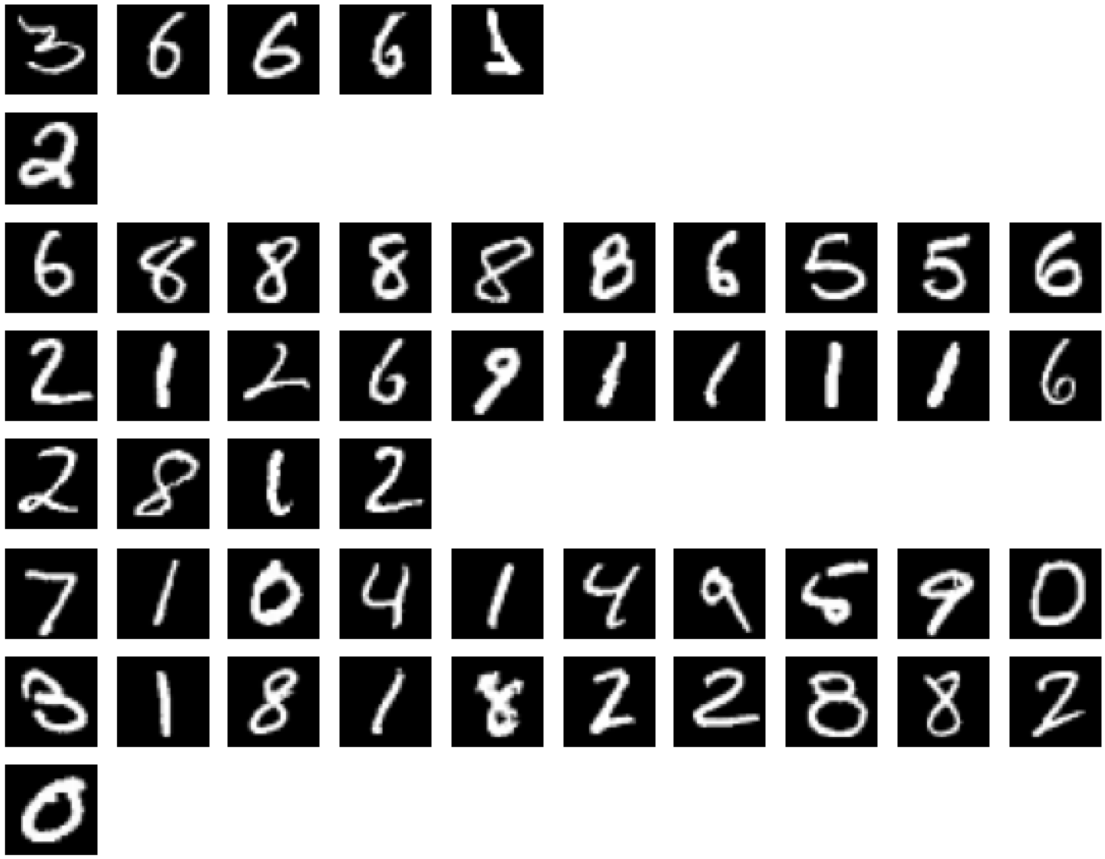


[167]:
```
# predict 3th layer
predict_hidden_layers(3)
```





[168]:
```
# predict 2nd layer
predict_hidden_layers(2)
```



In the plot for 1.3 wedo a feedforward step on our test data of 100 samples. We do a prediction on the softmax layer which is the final output of the Neural Network. We can see that the NN was able to predict the output of the 10 class labels pretty much with ease.

But when we tried to do the prediction of the hidden layers in between the input and output, we can clearly see that the NN is still learning to make predictions and hence few class labels was not even displayed.

My Helper codes

[169]:
```
# 1

# from torchvision import models
# from torchsummary import summary
# summary(model, (28*28, 1024))

# 2
```

```
# for name, param in model.named_parameters():
#   if param.requires_grad:
#     print(name, param.data)

# 3
# model.hl_3.weight

# 4
# with torch.no_grad():
#   img = (model(images[9])[0])
#   x = torch.argmax(torch.softmax(img, dim=1), dim=1).cpu().numpy()
#   # plt.imshow(x, cmap = 'gray',interpolation = None)
#   print(x)

# 5
# model.eval()
# img = (model(images[9])[0])
# x = torch.argmax(torch.softmax(img, dim=1), dim=1).cpu().numpy()
# # plt.imshow(x, cmap = 'gray',interpolation = None)
# print(x)
```

0.4.5 t-Stochastic Neighbor Embedding (tSNE) or Principal Component Analysis (PCA)

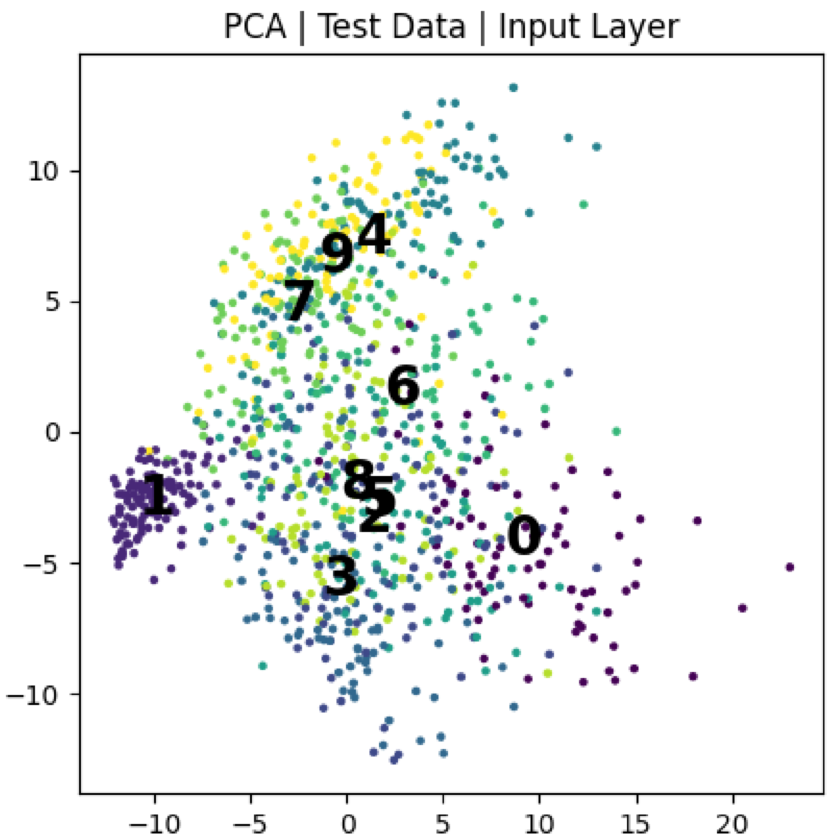
section 1.5 and 1.6

[170]:
```
def scatter_plot(data, class_labels, title):
  plt.style.use('default')
  plt.figure(figsize=(5,5))
  plt.scatter(x=data[:,0], y=data[:,1],c=class_labels, s= 5)

  for i in range(10):
    plt.annotate(str(i),
                 xy=data[np.where(class_labels == i),:].mean(axis=1)[0],
                 horizontalalignment='center',
                 verticalalignment='center',
                 size = 20, weight ='bold', color='black')
  plt.title(title)
```

[171]:
```
model.eval()
images, labels = fetch_test_batch()
labels = labels.detach().cpu().numpy()

# contains the output of 6 layers
# 1 input layer
# 5 hidden layers
predict = model(images.to(device))
```

PCA of the input data section 1.5 and 1.6

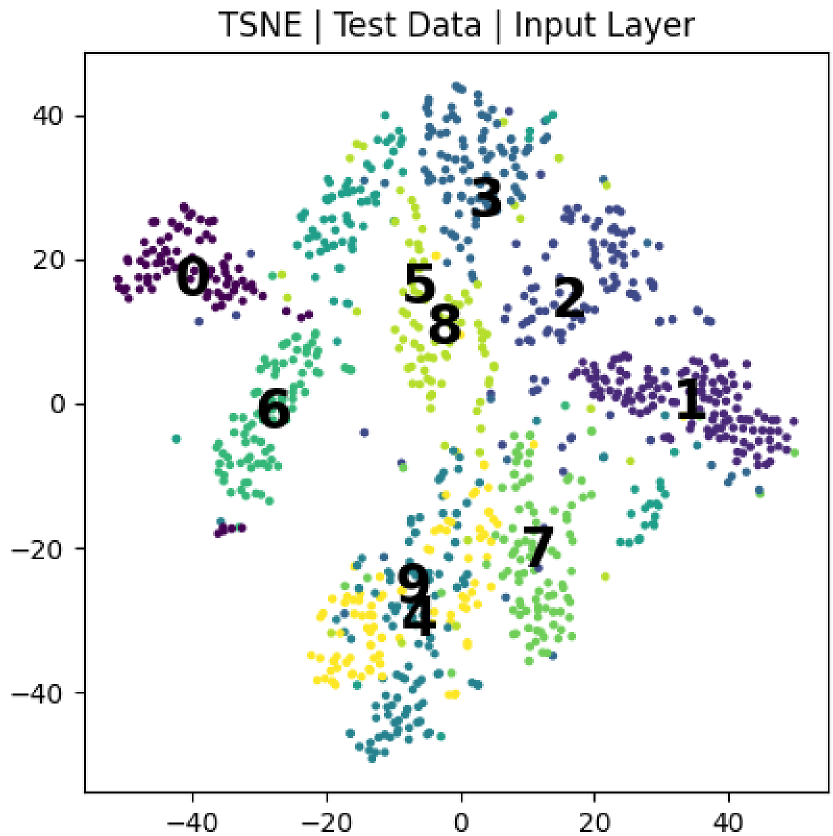
[172]:
```
pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[0].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | Input Layer")
```



TSNE of the input data

[173]:
```
%%time

tsne = TSNE(n_components=2)
tsne_output = tsne.fit_transform(predict[0].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | Input Layer")
```

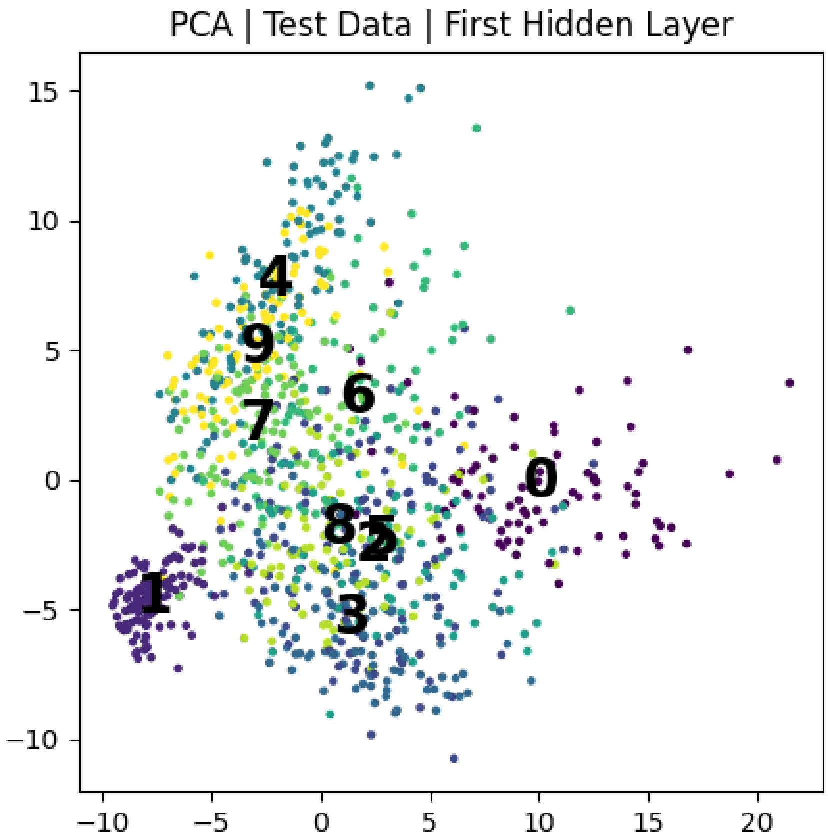
```
CPU times: user 11.7 s, sys: 77.8 ms, total: 11.8 s
Wall time: 6.98 s
```

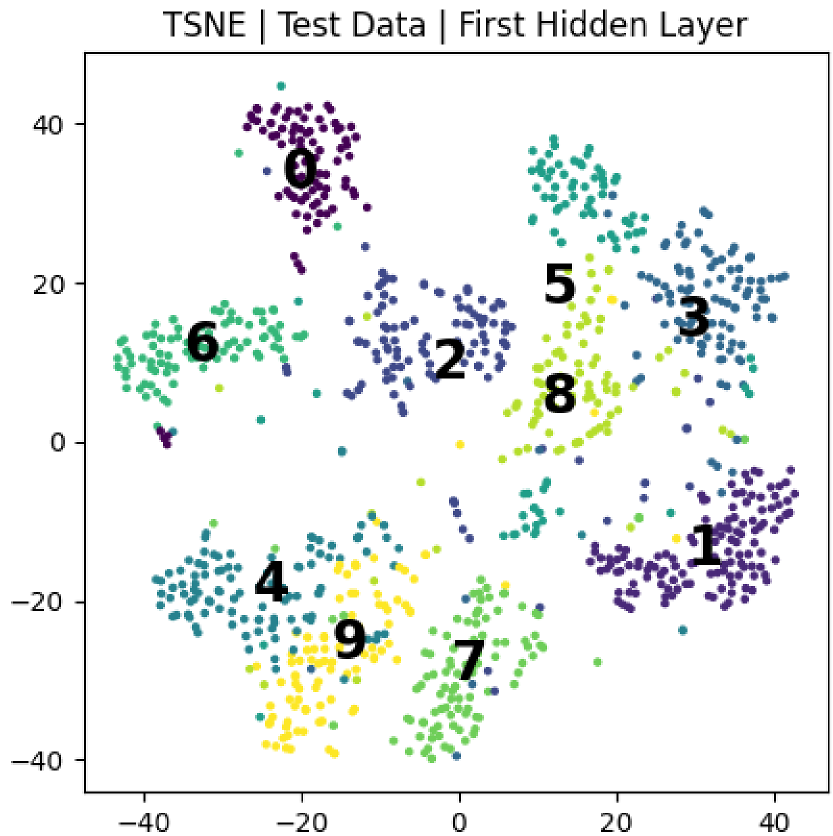


PCA and TSNE on the first hidden layer.

[174]:
```
%%time

pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[1].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | First Hidden Layer")

tsne_output = tsne.fit_transform(predict[1].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | First Hidden Layer")
```

```
CPU times: user 11.1 s, sys: 522 ms, total: 11.6 s
Wall time: 6.7 s
```

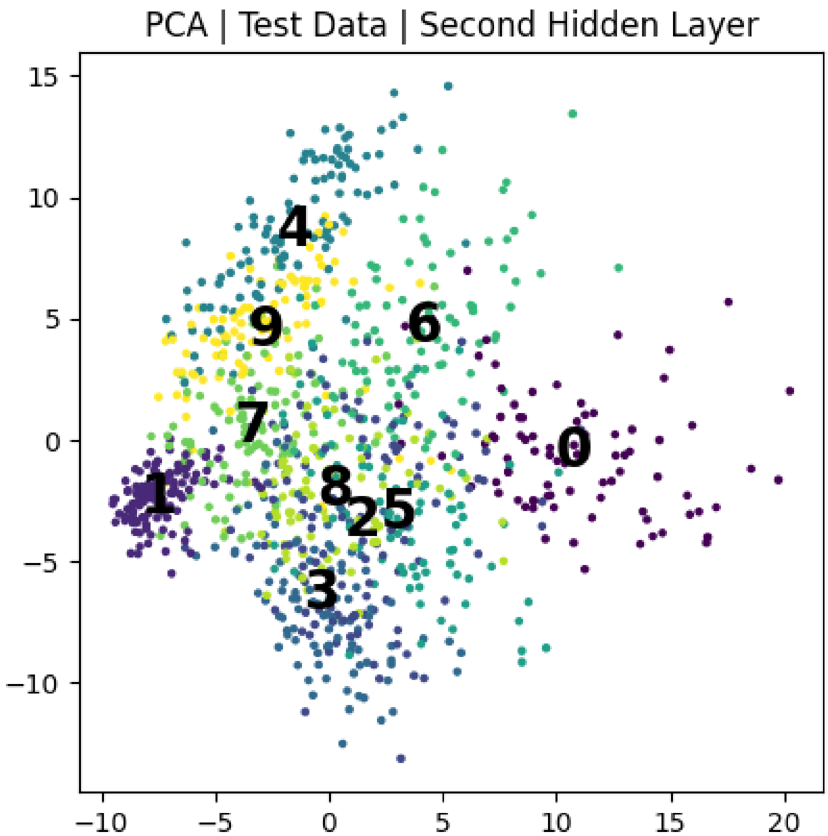

PCA | Test Data | First Hidden Layer
15
10
5
0
−5
−10
−10
−5
0
5
10
15
20
4
9
7
6
0
8
5
2
1
3

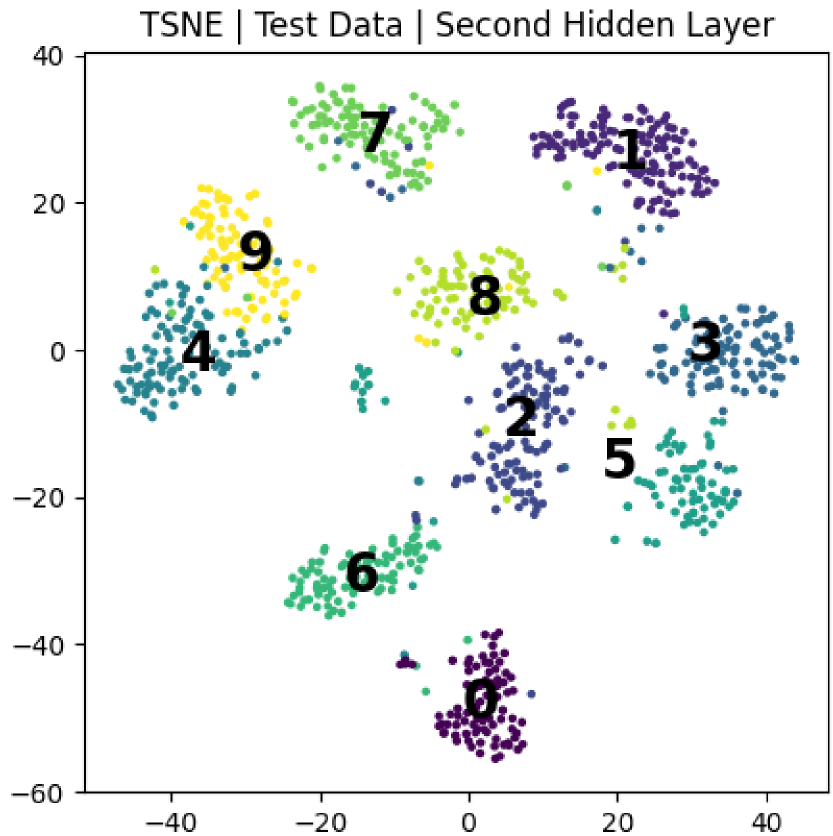


PCA and TSNE on the second hidden layer.

[175]:
```
%%time

pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[2].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | Second Hidden Layer")

tsne_output = tsne.fit_transform(predict[2].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | Second Hidden Layer")
```

```
CPU times: user 11 s, sys: 559 ms, total: 11.6 s
Wall time: 6.71 s
```

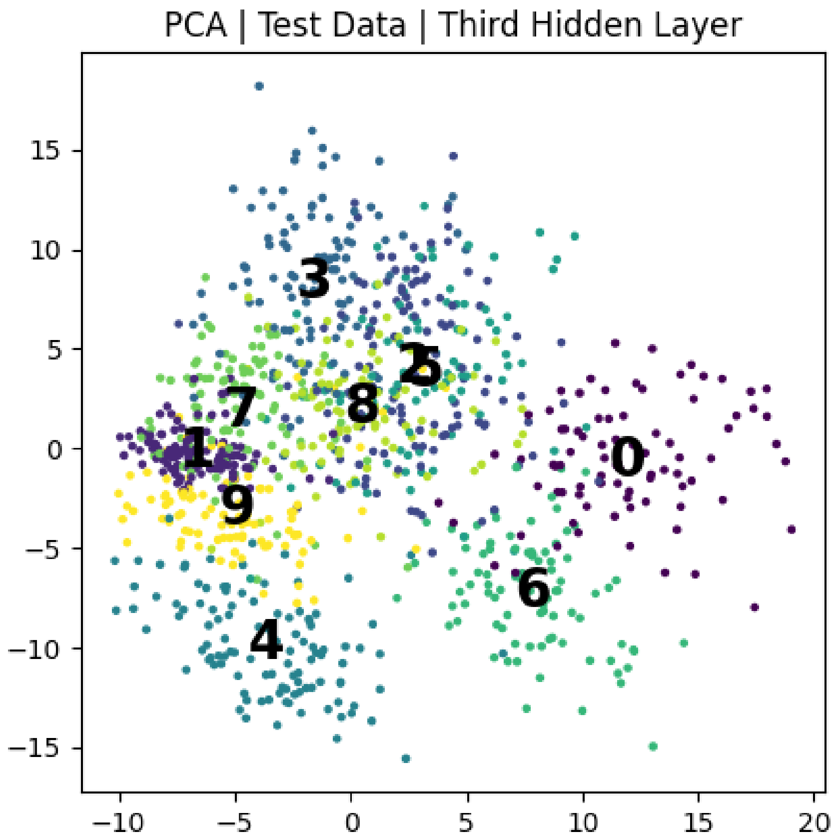

PCA | Test Data | Second Hidden Layer
15
10
5
0
−5
−10
−10
−5
0
5
10
15
20
4
9
6
7
0
1
8
2
5
3

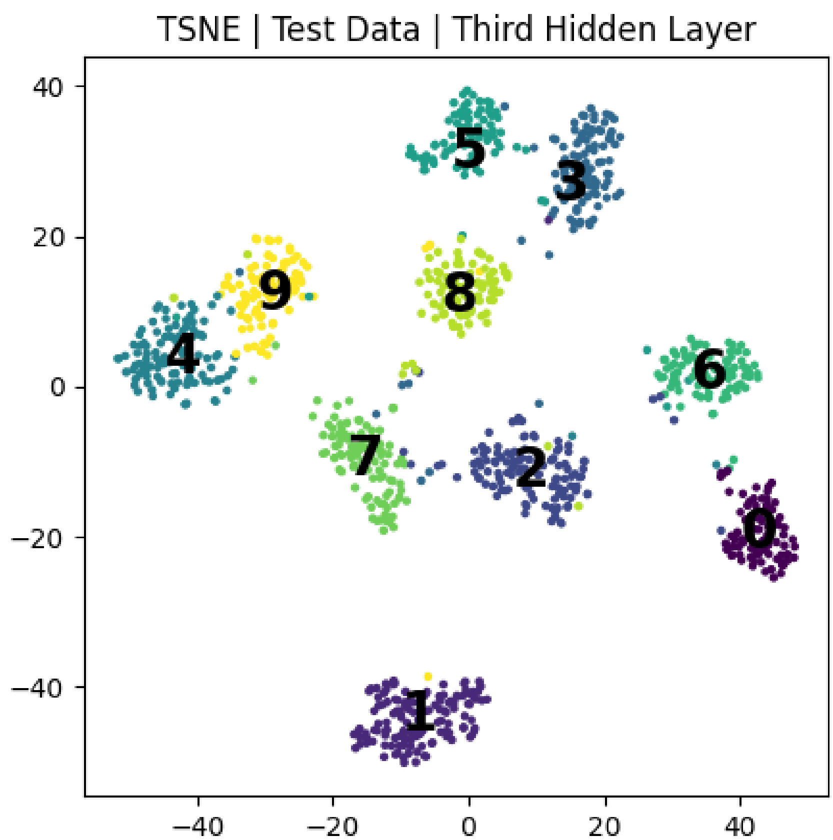


PCA and TSNE on the third hidden layer.

[176]:
```
%%time

pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[3].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | Third Hidden Layer")

tsne_output = tsne.fit_transform(predict[3].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | Third Hidden Layer")
```

```
CPU times: user 10.6 s, sys: 509 ms, total: 11.1 s
Wall time: 6.47 s
```

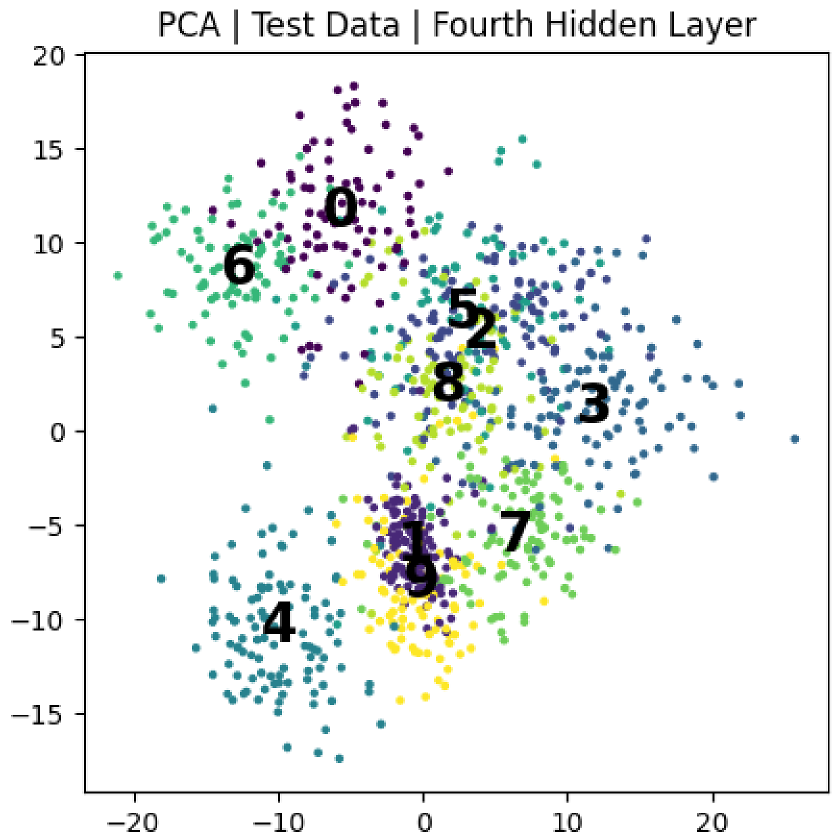

PCA | Test Data | Third Hidden Layer
15
10
5
0
−5
−10
−15
−10
−5
0
5
10
15
20
3
2
5
7
8
1
0
9
6
4

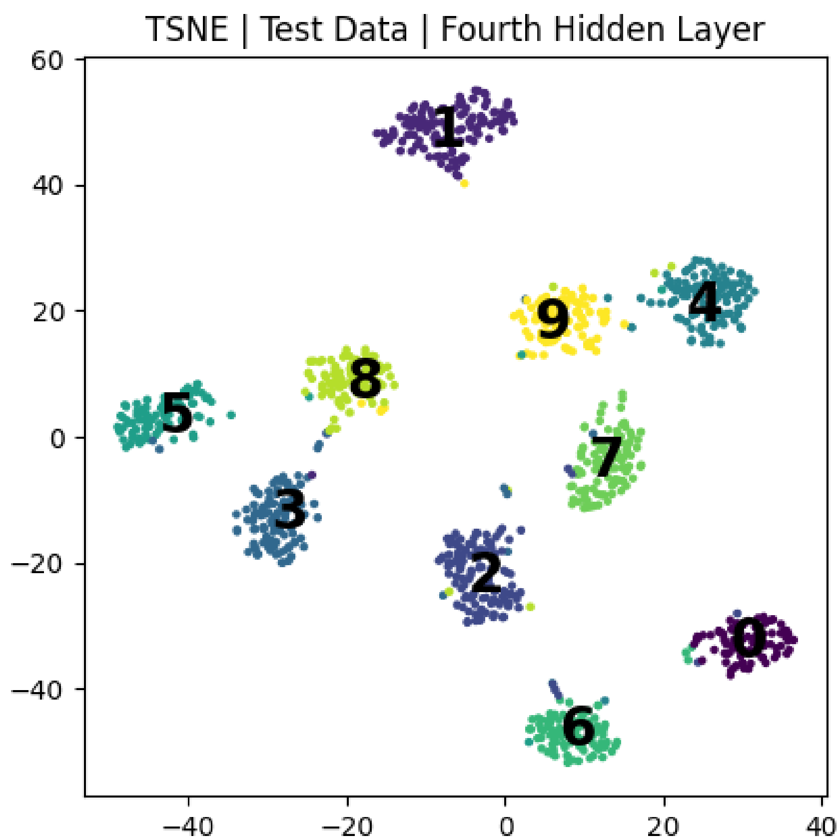


PCA and TSNE on the fourth hidden layer.

[177]:
```
%%time

pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[4].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | Fourth Hidden Layer")

tsne_output = tsne.fit_transform(predict[4].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | Fourth Hidden Layer")
```

```
CPU times: user 10.6 s, sys: 625 ms, total: 11.2 s
Wall time: 6.48 s
```


PCA | Test Data | Fourth Hidden Layer
20
15
10
5
0
−5
−10
−15
−20
−10
0
10
20
0
6
5
2
8
3
7
1
9
4



PCA and TSNE on the fifth hidden or final layer.

[ ]:
```
%%time

pca = PCA(n_components=2)
pca_output = pca.fit_transform(predict[-1].detach().cpu().numpy())
scatter_plot(pca_output, labels, "PCA | Test Data | 5th and Final Layer")

tsne_output = tsne.fit_transform(predict[-1].detach().cpu().numpy())
scatter_plot(tsne_output, labels, "TSNE | Test Data | 5th and Final Layer")
```

From the plots we can say that the TSNE dimension reduction performs much better than the PCA for our classification problem.

We observed that the TSNE output classification slowly convereges t achive the higher accuracy of classifying the class labels into the correct cluster of its class.

0.5 Problem 2: Adult Optimization [4 points]

0.5.1 Replicate the Figures in M03 Adult Optimization, slide 33 and 34.

[32]:
```
# Define model
class Optimizer_Network(nn.Module):
    def __init__(self, hidden_units, activation_fun):
        super(Optimizer_Network, self).__init__()
        self.flatten = nn.Flatten()
        self.hidden_units = hidden_units
        self.activation_fun = activation_fun
        self.hl_0 = nn.Linear(28 * 28, self.hidden_units)  # input layer
        self.hl_1 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
↪hidden 1
        self.hl_2 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
↪hidden 2
        self.hl_3 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
↪hidden 3
        self.hl_4 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
↪hidden 4
        self.hl_5 = nn.Linear(self.hidden_units, 10)       # hidden 5 / o/p


    def forward(self, x):
        x = self.flatten(x)
        out = self.activation_fun(self.hl_0(x))
        out = self.activation_fun(self.hl_1(out))
        out = self.activation_fun(self.hl_2(out))
        out = self.activation_fun(self.hl_3(out))
        out = self.activation_fun(self.hl_4(out))
        out = self.hl_5(out)
        return out
```

0.5.2 Multiple initializer for different models.

[33]:
```
def normal_initializer(mod):
  if isinstance(mod , nn.Linear):
    nn.init.normal_(mod.weight.data, mean = 0.0, std= 0.01)

def xavier_initializer(mod):
  if isinstance(mod , nn.Linear):
    nn.init.xavier_normal_(mod.weight.data)

def kaiming_he_initializer(mod):
  if isinstance(mod , nn.Linear):
    nn.init.kaiming_normal_(mod.weight.data, nonlinearity='relu')
```

0.5.3 Creating five different networks that share the same architecture.

1. Model: LSNI() Activation function: logistic sigmoid function initialization: normal distribution (mean = 0, std = 0:01)

[34]:
```
LSNI = Optimizer_Network(512, torch.nn.Sigmoid()).to(device)
LSNI.apply(normal_initializer)
LSNI.name = 'LSNI'
```

2. Model: LSXI() Activation function: logistic sigmoid function initialization: Xavier initializer

[35]:
```
LSXI = Optimizer_Network(512, torch.nn.Sigmoid()).to(device)
LSXI.apply(xavier_initializer)
LSXI.name = 'LSXI'
```

3. Model: RLNI() Activation function: ReLu initialization: normal distribution (mean = 0, std = 0:01)

[36]:
```
RLNI = Optimizer_Network(512, torch.nn.ReLU()).to(device)
RLNI.apply(normal_initializer)
RLNI.name = 'RLNI'
```

4. Model: RLXI() Activation function: ReLu initialization: Xavier Initialixer

[37]:
```
RLXI = Optimizer_Network(512, torch.nn.ReLU()).to(device)
RLXI.apply(xavier_initializer)
RLXI.name = 'RLXI'
```

5. Model: RLKHeI() Activation function: ReLu initialization: Kaiming He's initializer.

[38]:
```
RLKHeI = Optimizer_Network(512, torch.nn.ReLU()).to(device)
RLKHeI.apply(kaiming_he_initializer)
RLKHeI.name = 'RLKHeI'
```

0.5.4 Training the model for n epochs

[39]:
```
%%time

# Defining the optimiser and loss function

def train_2(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    train_loss = 0.0

    for i, data in enumerate(dataloader, 0):
        X, y = data[0], data[1]
        # b = X.size(0)
```

```
        # X = X.view(b , -1)
        X, y = X.to(device), y.to(device)

        # re initialize the gradients parameters
        optimizer.zero_grad()
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        train_loss+= loss.item()
    return train_loss /len(dataloader.dataset)

def test_2(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_acc, correct = 0, 0
    for i, data in enumerate(dataloader, 0):
      X, y = data[0], data[1]
      # b = X.size(0)
      # X = X.view(b , -1)
      X, y = X.to(device), y.to(device)

      model_output = model(X)
      pred = torch.argmax(torch.softmax(model_output, dim = 1), dim = 1)
      acc = torch.sum(pred == y)
      test_acc += acc.cpu().numpy()

    return test_acc/size * 100


def evaluate_models(model, chosen_optimizer, lrate = 1e-2):
  loss_fn = nn.CrossEntropyLoss()

  if chosen_optimizer == 'SGD':
    optimizer = optim.SGD(model.parameters(), lr= lrate)
  elif chosen_optimizer == "ADAM":
    optimizer = optim.Adam(model.parameters(), lr= lrate)

  epochs = 201
  test_accuracy_list = []

  print(f'\n Mod : {model.name}   Optimizer: {chosen_optimizer} \n ')
  for t in range(epochs):
```

```
      train_loss = 0.0

      train_loss = train_2(train_dataloader, model, loss_fn, optimizer)
      test_accuracy = test_2(test_dataloader, model, loss_fn)
      if t % 50 == 0:
        print(f"Epoch : {t}  train loss: {train_loss:>7f} \
        test Accuracy : {test_accuracy:>5f}")
      test_accuracy_list.append(test_accuracy)

  print("Done!")
  return test_accuracy_list
```

```
CPU times: user 5 ţs, sys: 0 ns, total: 5 ţs
Wall time: 8.58 ţs
```

0.5.5 Slide 33 & 34 with FIXED LR = 0.01.

[50]:
```
%%time
sgd = "SGD"
adam = "ADAM"

train_dataloader, test_dataloader = mnist_dataloader(512, 500)

# Chosen Optimizer = SIGMOID
LSNI.apply(normal_initializer)
test_data_LSNI_sgd = evaluate_models(model= LSNI, chosen_optimizer= sgd,
                                     lrate = 1e-2)

LSXI.apply(xavier_initializer)
test_data_LSXI_sgd = evaluate_models(LSXI, sgd, 1e-2)

RLNI.apply(normal_initializer)
test_data_RLNI_sgd = evaluate_models(RLNI, sgd, 1e-2)

RLXI.apply(xavier_initializer)
test_data_RLXI_sgd = evaluate_models(RLXI, sgd, 1e-2)

RLKHeI.apply(kaiming_he_initializer)
test_data_RLKHeI_sgd = evaluate_models(RLKHeI, sgd, 1e-2)

# Chosen Optimizer = ADAM

train_dataloader, test_dataloader = mnist_dataloader(512, 500)
LSNI.apply(normal_initializer)
test_data_LSNI_adam = evaluate_models(LSNI, adam, 1e-2)

LSXI.apply(xavier_initializer)
```

```
test_data_LSXI_adam = evaluate_models(LSXI, adam, 1e-2)

RLNI.apply(normal_initializer)
test_data_RLNI_adam = evaluate_models(RLNI, adam, 1e-2)

RLXI.apply(xavier_initializer)
test_data_RLXI_adam = evaluate_models(RLXI, adam, 1e-2)

RLKHeI.apply(kaiming_he_initializer)
test_data_RLKHeI_adam = evaluate_models(RLKHeI, adam, 1e-2)
```

```
 Mod : LSNI   Optimizer: SGD

Epoch : 0  train loss: 0.004527         test Accuracy : 11.350000
Epoch : 50  train loss: 0.004527         test Accuracy : 10.280000
Epoch : 100  train loss: 0.004527         test Accuracy : 11.350000
Epoch : 150  train loss: 0.004527         test Accuracy : 11.350000
Epoch : 200  train loss: 0.004527         test Accuracy : 11.350000
Done!

 Mod : LSXI   Optimizer: SGD

Epoch : 0  train loss: 0.004541         test Accuracy : 11.350000
Epoch : 50  train loss: 0.004523         test Accuracy : 11.350000
Epoch : 100  train loss: 0.004514         test Accuracy : 10.090000
Epoch : 150  train loss: 0.004481         test Accuracy : 21.090000
Epoch : 200  train loss: 0.003270         test Accuracy : 44.430000
Done!

 Mod : RLNI   Optimizer: SGD

Epoch : 0  train loss: 0.004528         test Accuracy : 10.320000
Epoch : 50  train loss: 0.004528         test Accuracy : 9.580000
Epoch : 100  train loss: 0.004528         test Accuracy : 11.350000
Epoch : 150  train loss: 0.004528         test Accuracy : 11.350000
Epoch : 200  train loss: 0.004528         test Accuracy : 11.350000
Done!

 Mod : RLXI   Optimizer: SGD

Epoch : 0  train loss: 0.005340         test Accuracy : 22.510000
Epoch : 50  train loss: 0.000150         test Accuracy : 96.820000
Epoch : 100  train loss: 0.000043         test Accuracy : 97.420000
Epoch : 150  train loss: 0.000012         test Accuracy : 97.530000
Epoch : 200  train loss: 0.000005         test Accuracy : 97.540000
Done!
```

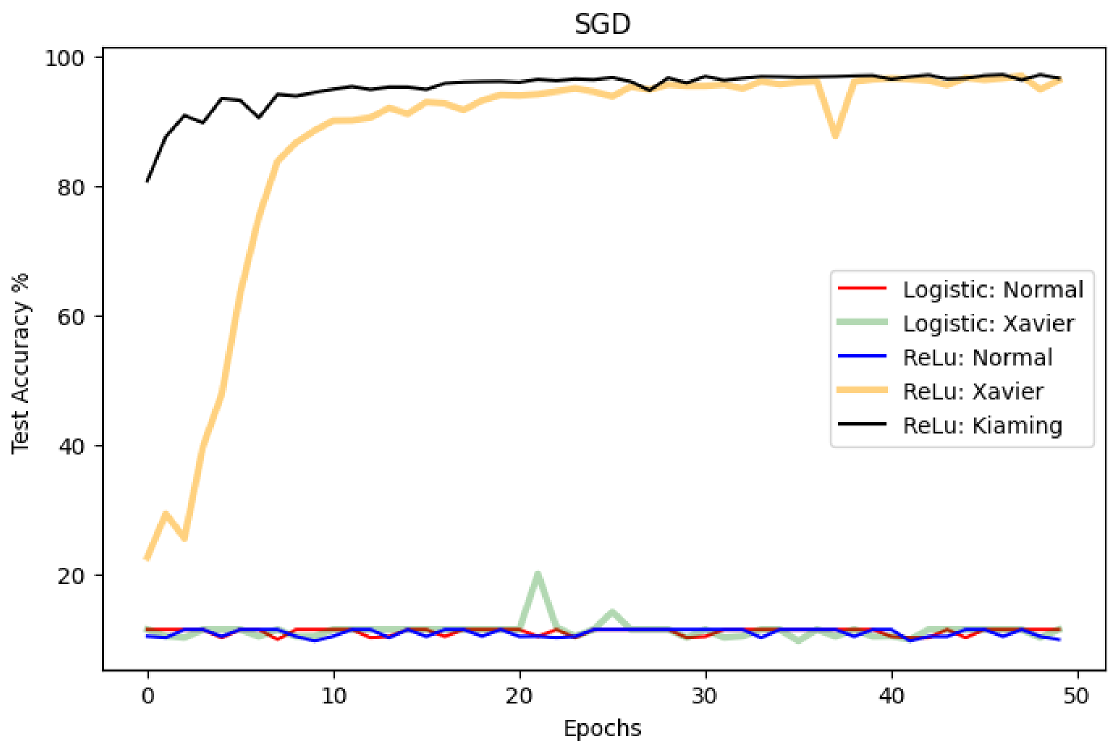
```
 Mod : RLKHeI   Optimizer: SGD

Epoch : 0  train loss: 0.002697         test Accuracy : 80.640000
Epoch : 50  train loss: 0.000055         test Accuracy : 97.180000
Epoch : 100  train loss: 0.000012         test Accuracy : 97.470000
Epoch : 150  train loss: 0.000005         test Accuracy : 97.480000
Epoch : 200  train loss: 0.000003         test Accuracy : 97.550000
Done!

 Mod : LSNI   Optimizer: ADAM

Epoch : 0  train loss: 0.004563         test Accuracy : 11.350000
Epoch : 50  train loss: 0.004526         test Accuracy : 11.350000
Epoch : 100  train loss: 0.004526         test Accuracy : 11.350000
Epoch : 150  train loss: 0.004526         test Accuracy : 11.350000
Epoch : 200  train loss: 0.004526         test Accuracy : 11.350000
Done!

 Mod : LSXI   Optimizer: ADAM

Epoch : 0  train loss: 0.004826         test Accuracy : 9.800000
Epoch : 50  train loss: 0.002686         test Accuracy : 29.870000
Epoch : 100  train loss: 0.002408         test Accuracy : 37.860000
Epoch : 150  train loss: 0.002305         test Accuracy : 37.660000
Epoch : 200  train loss: 0.002296         test Accuracy : 37.410000
Done!

 Mod : RLNI   Optimizer: ADAM

Epoch : 0  train loss: 0.002869         test Accuracy : 83.240000
Epoch : 50  train loss: 0.000050         test Accuracy : 97.630000
Epoch : 100  train loss: 0.000055         test Accuracy : 97.380000
Epoch : 150  train loss: 0.000045         test Accuracy : 97.280000
Epoch : 200  train loss: 0.000211         test Accuracy : 97.000000
Done!

 Mod : RLXI   Optimizer: ADAM

Epoch : 0  train loss: 0.004035         test Accuracy : 60.810000
Epoch : 50  train loss: 0.000525         test Accuracy : 90.030000
Epoch : 100  train loss: 0.000502         test Accuracy : 90.740000
Epoch : 150  train loss: 0.000505         test Accuracy : 90.490000
Epoch : 200  train loss: 0.000497         test Accuracy : 90.800000
Done!

 Mod : RLKHeI   Optimizer: ADAM
```

```
Epoch : 0  train loss: 0.002865         test Accuracy : 92.020000
Epoch : 50  train loss: 0.000142         test Accuracy : 96.140000
Epoch : 100  train loss: 0.000158         test Accuracy : 96.050000
Epoch : 150  train loss: 0.000134         test Accuracy : 96.280000
Epoch : 200  train loss: 0.000127         test Accuracy : 95.920000
Done!
CPU times: user 24min 39s, sys: 25.2 s, total: 25min 4s
Wall time: 24min 48s
```

Adam - On deeper networks with no pretraining For a 512X5 network for MNIST

Replicating slide - 33

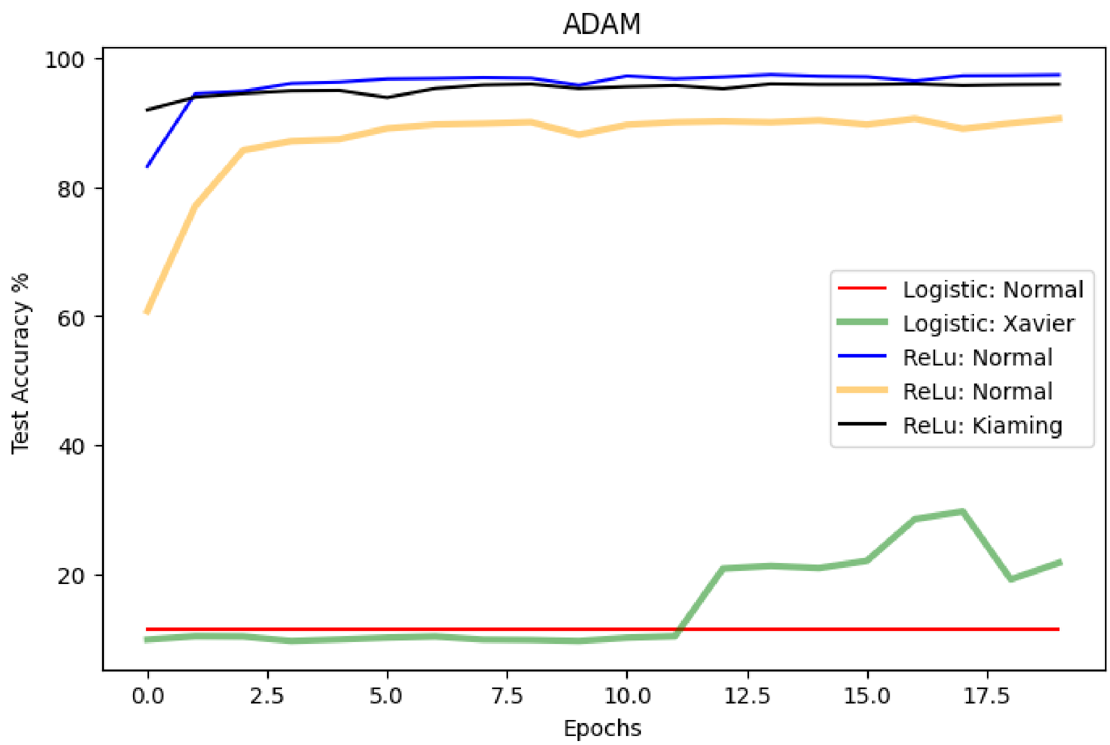
[51]:
```
%%time

# Plotting the test accuracy results for SGD Optimizer
epochs = range(0,50)
plt.figure(figsize=(8,5))
plt.plot(epochs, test_data_LSNI_sgd[:50], color='red',
         label='Logistic: Normal')
plt.plot(epochs, test_data_LSXI_sgd[:50], color='green', linewidth= 3,
         alpha=0.3,label='Logistic: Xavier')
plt.plot(epochs, test_data_RLNI_sgd[:50], color='blue', label='ReLu: Normal')
plt.plot(epochs, test_data_RLXI_sgd[:50], color='orange',linewidth= 3,
         alpha=0.5, label='ReLu: Xavier')
plt.plot(epochs, test_data_RLKHeI_sgd[:50], color='black',  alpha=1,
         label='ReLu: Kiaming')
plt.title('SGD')
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy % ')
plt.legend()
plt.show()
```




```
CPU times: user 230 ms, sys: 3.02 ms, total: 233 ms
Wall time: 229 ms
```

[52]:

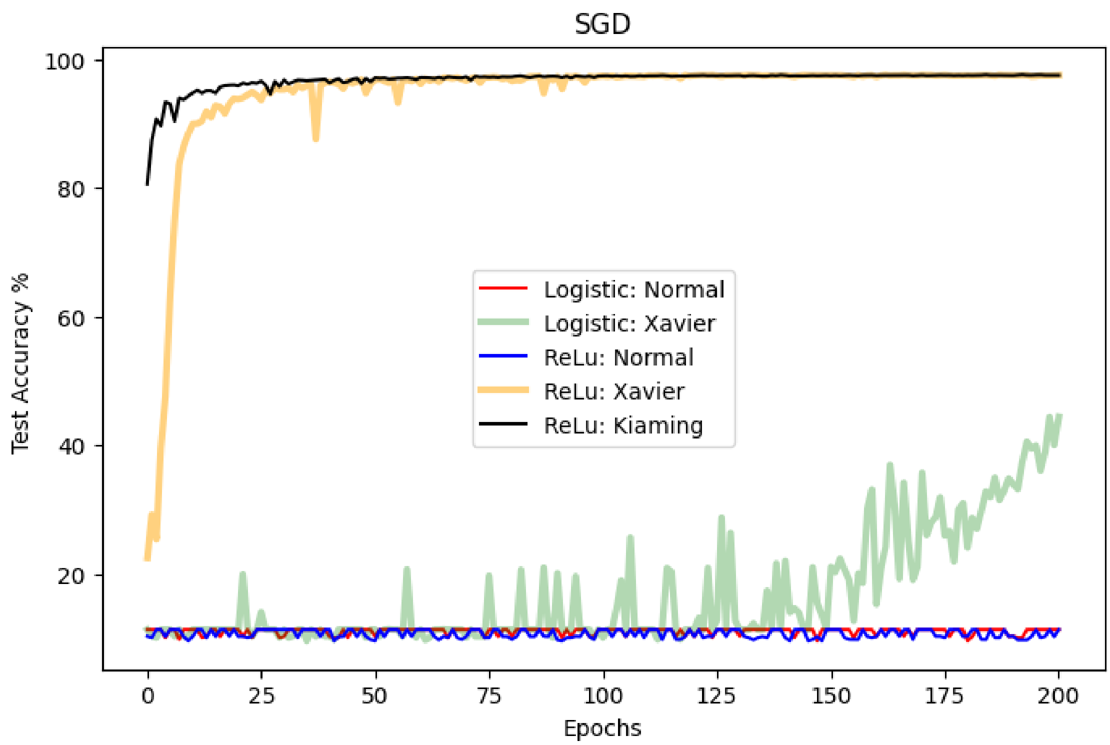
```
# Plotting the test accuracy results for ADAM Optimizer
epochs = range(0,20)
plt.figure(figsize=(8,5))
plt.plot(epochs, test_data_LSNI_adam[:20], color='red', label='Logistic:␣
↪Normal')
plt.plot(epochs, test_data_LSXI_adam[:20], color='green',
         linewidth= 3, alpha=0.5, label='Logistic: Xavier')
plt.plot(epochs, test_data_RLNI_adam[:20], color='blue', label='ReLu: Normal')
plt.plot(epochs, test_data_RLXI_adam[:20], color='orange',
         linewidth= 3, alpha=0.5, label='ReLu: Normal')
plt.plot(epochs, test_data_RLKHeI_adam[:20], color='black', label='ReLu:␣
↪Kiaming')
plt.title('ADAM')
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy % ')
plt.legend()
plt.show()
```



Adam - On deeper networks with no pretraining

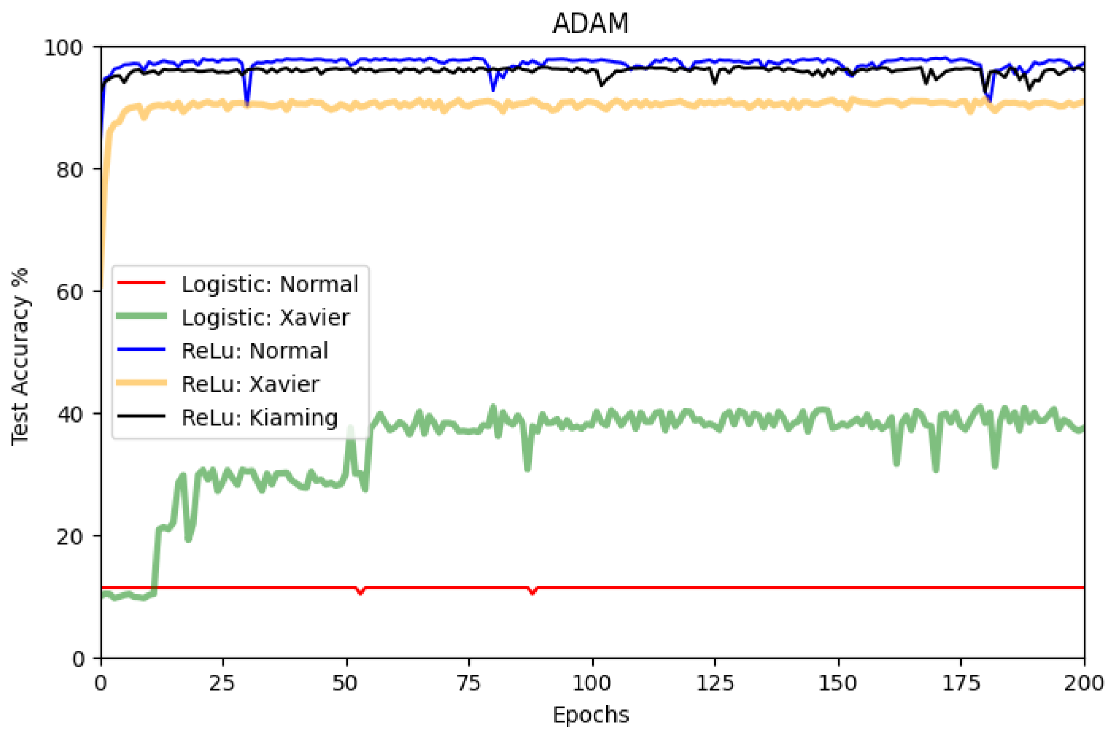
Replicating slide - 34

[53]:
```
%%time

# Plotting the test accuracy results for SGD Optimizer
epochs = range(201)
plt.figure(figsize=(8,5))
plt.plot(epochs, test_data_LSNI_sgd, color='red', label='Logistic: Normal')
plt.plot(epochs, test_data_LSXI_sgd, color='green', linewidth= 3, alpha=0.
 ↪3,label='Logistic: Xavier')
plt.plot(epochs, test_data_RLNI_sgd, color='blue', label='ReLu: Normal')
plt.plot(epochs, test_data_RLXI_sgd, color='orange',linewidth= 3, alpha=0.5,␣
 ↪label='ReLu: Xavier')
plt.plot(epochs, test_data_RLKHeI_sgd, color='black',  alpha=1,label='ReLu:␣
 ↪Kiaming')
plt.title('SGD')
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy % ')
plt.legend()
plt.show()
```




```
CPU times: user 263 ms, sys: 5.09 ms, total: 268 ms
Wall time: 262 ms
```

[54]:
```
%%time

# Plotting the test accuracy results for ADAM Optimizer
epochs = range(0,201)
plt.figure(figsize=(8,5))
plt.ylim(0,100 )
plt.xlim(0,200)
plt.plot(epochs, test_data_LSNI_adam, color='red', label='Logistic: Normal')
plt.plot(epochs, test_data_LSXI_adam, color='green',
         linewidth= 3, alpha=0.5, label='Logistic: Xavier')
plt.plot(epochs, test_data_RLNI_adam, color='blue', label='ReLu: Normal')
plt.plot(epochs, test_data_RLXI_adam, color='orange',
         linewidth= 3, alpha=0.5, label='ReLu: Xavier')
plt.plot(epochs, test_data_RLKHeI_adam, color='black', label='ReLu: Kiaming')
plt.title('ADAM')
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy % ')
plt.legend()
plt.show()
```




```
CPU times: user 248 ms, sys: 7.02 ms, total: 255 ms
Wall time: 253 ms
```

0.6 Problem 3: Dropout [3 points]

Replicate the figures in M03 Adult Optimization, slide 40.

0.6.1 Dropout - Structural noise injection

[129]:
```
# Define model
class NoDropuout_Network(nn.Module):
    def __init__(self, hidden_units, activation_fun):
        super(NoDropuout_Network, self).__init__()
        self.flatten = nn.Flatten()
        self.hidden_units = hidden_units
        self.activation_fun = activation_fun
        self.hl_0 = nn.Linear(28 * 28, self.hidden_units)  # input layer
        self.hl_1 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 1
        self.hl_2 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 2
        self.hl_3 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 3
```

```
        self.hl_4 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 4
        self.hl_5 = nn.Linear(self.hidden_units, 10)       # hidden 5 / o/p


    def forward(self, x):
        x = self.flatten(x)
        out = self.activation_fun(self.hl_0(x))
        out = self.activation_fun(self.hl_1(out))
        out = self.activation_fun(self.hl_2(out))
        out = self.activation_fun(self.hl_3(out))
        out = self.activation_fun(self.hl_4(out))
        out = self.hl_5(out)
        return out
```

[130]:
```
# Define model
class Dropuout_Network(nn.Module):
    def __init__(self, hidden_units, activation_fun):
        super(Dropuout_Network, self).__init__()
        self.flatten = nn.Flatten()
        self.hidden_units = hidden_units
        self.activation_fun = activation_fun
        self.hl_0 = nn.Linear(28 * 28, self.hidden_units)  # input layer
        self.hl_1 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 1
        self.hl_2 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 2
        self.hl_3 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 3
        self.hl_4 = nn.Linear(self.hidden_units, self.hidden_units)     #␣
 ↪hidden 4
        self.hl_5 = nn.Linear(self.hidden_units, 10)       # hidden 5 / o/p

        self.dropout_layer1 = nn.Dropout(p=0.2)
        self.dropout_layerx = nn.Dropout(p=0.5)


    def forward(self, x):
        x = self.flatten(x)
        x = self.dropout_layer1(x)
        x = self.activation_fun(self.hl_0(x))
        x = self.dropout_layerx(x)
        x = self.activation_fun(self.hl_1(x))
        x = self.dropout_layerx(x)
        x = self.activation_fun(self.hl_2(x))
        x = self.dropout_layerx(x)
        x = self.activation_fun(self.hl_3(x))
```

```
        x = self.dropout_layerx(x)
        x = self.activation_fun(self.hl_4(x))
        x = self.dropout_layerx(x)
        x = self.hl_5(x)
        return x
```

0.6.2 Creating Four different networks to verify dropout.

1. Model: LSXI_NDrp() Activation function: logistic sigmoid function Initialization : Xavier Initializer Droupout : NO

[131]:
```
LSXI_NDrp = NoDropuout_Network(1024, torch.nn.Sigmoid()).to(device)
LSXI_NDrp.apply(xavier_initializer)
LSXI_NDrp.name = 'LSXI_NDrp'
```

2. Model: LSXI_YDrp() Activation function: logistic sigmoid function Initialization : Xavier Initializer Droupout : Yes

[132]:
```
LSXI_YDrp = Dropuout_Network(1024, torch.nn.Sigmoid()).to(device)
LSXI_YDrp.apply(xavier_initializer)
LSXI_YDrp.name = 'LSXI_YDrp'
```

3. Model: RLKHeI_NDrp() Activation function: ReLu function Initialization : Kaiming He's Initializer Droupout : NO

[133]:
```
RLKHeI_NDrp = NoDropuout_Network(1024, torch.nn.ReLU()).to(device)
RLKHeI_NDrp.apply(kaiming_he_initializer)
RLKHeI_NDrp.name = 'RLKHeI_NDrp'
```

4. Model: RLKHeI_YDrp() Activation function: ReLu function Initialization : Kaiming He's Initializer Droupout : YES

[134]:
```
RLKHeI_YDrp = Dropuout_Network(1024, torch.nn.ReLU()).to(device)
RLKHeI_YDrp.apply(kaiming_he_initializer)
RLKHeI_YDrp.name = 'RLKHeI_YDrp'
```

[135]:
```
def train_dropout(dataloader, model, loss_fn, optimizer):
  model.train()
  train_loss = 0.0
  loss_fn = nn.CrossEntropyLoss()
  for i, data in enumerate(train_dataloader, 0):
      X, y = data[0], data[1]
      # b = X.size(0)
      # X = X.view(b , -1)
      X, y = X.to(device), y.to(device)

      # re initialize the gradients parameters
      optimizer.zero_grad()
```

```
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        loss.backward()  # Backpropagation
        optimizer.step()
        train_loss+= loss.item()

    return train_loss /len(dataloader.dataset)

def test_dropout(dataloader, model, loss_fn):
      model.eval()
      test_loss = 0.0
      test_acc = 0.0
      for i, data in enumerate(dataloader, 0):
        X, y = data[0], data[1]
        X, y = X.to(device), y.to(device)
        model_output = model(X)

        pred = torch.argmax(torch.softmax(model_output, dim = 1), dim = 1)
        acc = torch.sum(pred == y)
        loss = loss_fn(model_output, y)
        test_loss += loss.item()

        test_acc += acc.cpu().numpy()
        test_acc = test_acc/len(dataloader.dataset)
        test_loss = test_loss/len(dataloader.dataset)
      return test_loss, test_acc
```

[136]:
```
%%time

# Defining the optimiser and loss function

def evaluate_dropout_models(model, chosen_optimizer, lrate, epochs ):
  loss_fn = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(), lr= lrate)

  train_loss_list = []
  test_loss_list = []

  print(f'\n Mod : {model.name}   Optimizer: {chosen_optimizer} \n \n ')
  for t in range(epochs):
      train_loss = 0.0
      train_loss = train_dropout(train_dataloader, model, loss_fn, optimizer)
      train_loss_list.append(train_loss)
      test_loss, test_acc = test_dropout(test_dataloader, model, loss_fn)
      # print(f"Epoch : {t}  train loss: {train_loss:>7f} \
```

```
        # test loss : {test_loss:>5f}")
        # test_loss_list.append(test_loss)
        if t % 100 == 0 :
          print(f"Epoch : {t}  train loss: {train_loss:>7f} test loss :␣
↪{test_loss:>5f}  test acc : {test_acc:>5f}")
        test_loss_list.append(test_loss)

    print("Done!")
    return train_loss_list ,  test_loss_list
```

```
CPU times: user 4 ţs, sys: 0 ns, total: 4 ţs
Wall time: 7.15 ţs
```

0.6.3 Evaluate the Models for Dropout

[137]:
```
%%time
adam = "ADAM"

# download MNIST dataand set up the dataloader
train_dataloader, test_dataloader = mnist_dataloader(200, 1000)

LSXI_NDrp.apply(xavier_initializer)
train_LSXI_NDrp, test_LSXI_NDrp = evaluate_dropout_models(LSXI_NDrp,
                                                          adam, 0.0001, 501)
LSXI_YDrp.apply(xavier_initializer)
train_LSXI_YDrp, test_LSXI_YDrp = evaluate_dropout_models(LSXI_YDrp,
                                                          adam, 0.0001, 501)
```

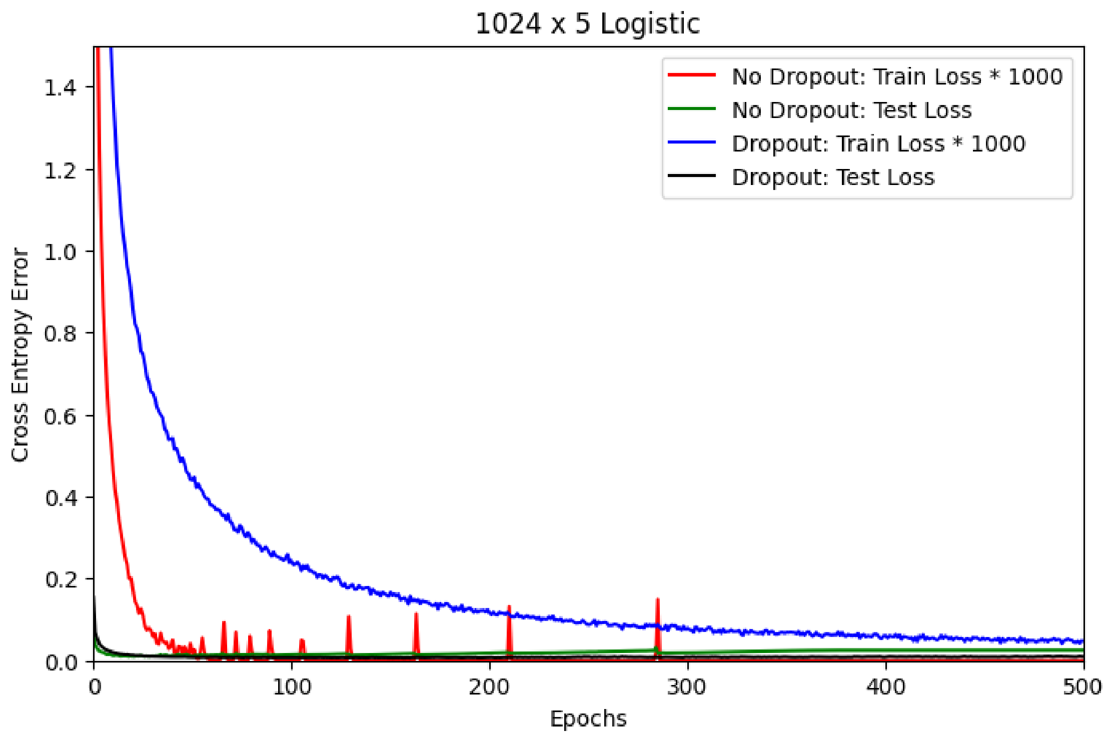
```
 Mod : LSXI_NDrp   Optimizer: ADAM


Epoch : 0  train loss: 0.007104 test loss : 0.000064  test acc : 0.082509
Epoch : 100  train loss: 0.000000 test loss : 0.000015  test acc : 0.098410
Epoch : 200  train loss: 0.000000 test loss : 0.000018  test acc : 0.098410
Epoch : 300  train loss: 0.000000 test loss : 0.000019  test acc : 0.098210
Epoch : 400  train loss: 0.000000 test loss : 0.000025  test acc : 0.098210
Epoch : 500  train loss: 0.000000 test loss : 0.000025  test acc : 0.098210
Done!

 Mod : LSXI_YDrp   Optimizer: ADAM


Epoch : 0  train loss: 0.011656 test loss : 0.000154  test acc : 0.041304
Epoch : 100  train loss: 0.000243 test loss : 0.000007  test acc : 0.098510
Epoch : 200  train loss: 0.000116 test loss : 0.000007  test acc : 0.098510
Epoch : 300  train loss: 0.000079 test loss : 0.000008  test acc : 0.098310
```

```
Epoch : 400  train loss: 0.000060 test loss : 0.000008  test acc : 0.098510
Epoch : 500  train loss: 0.000043 test loss : 0.000010  test acc : 0.098410
Done!
CPU times: user 23min 17s, sys: 32.6 s, total: 23min 50s
Wall time: 23min 36s
```

Replicating slide - 40

[138]:
```
train_LSXI_NDrp_x = list(np.asarray(train_LSXI_NDrp)* 1000)
test_LSXI_NDrp_x  = list(np.asarray(test_LSXI_NDrp) * 1000)
train_LSXI_YDrp_x = list(np.asarray(train_LSXI_YDrp)* 1000)
test_LSXI_YDrp_x  = list(np.asarray(test_LSXI_YDrp) * 1000)
```

[146]:
```
%%time

x_axis_limit = 500

# Plotting the test accuracy results for Logistic sigmoid with and with dropout
epochs = range(0,x_axis_limit)
plt.figure(figsize=(8,5))
plt.ylim(0.0,1.5)
plt.xlim(0,x_axis_limit)
plt.plot(epochs, train_LSXI_NDrp_x[:x_axis_limit], color='red',
         label='No Dropout: Train Loss * 1000')
plt.plot(epochs, test_LSXI_NDrp_x[:x_axis_limit] , color='green',
         label='No Dropout: Test Loss')
plt.plot(epochs, train_LSXI_YDrp_x[:x_axis_limit] , color='blue',
         label='Dropout: Train Loss * 1000')
plt.plot(epochs, test_LSXI_YDrp_x[:x_axis_limit], color='black',
         label='Dropout: Test Loss')

plt.title('1024 x 5 Logistic')
plt.xlabel('Epochs')
plt.ylabel('Cross Entropy Error')
plt.legend()
plt.show()
```




```
CPU times: user 236 ms, sys: 3.97 ms, total: 240 ms
Wall time: 238 ms
```

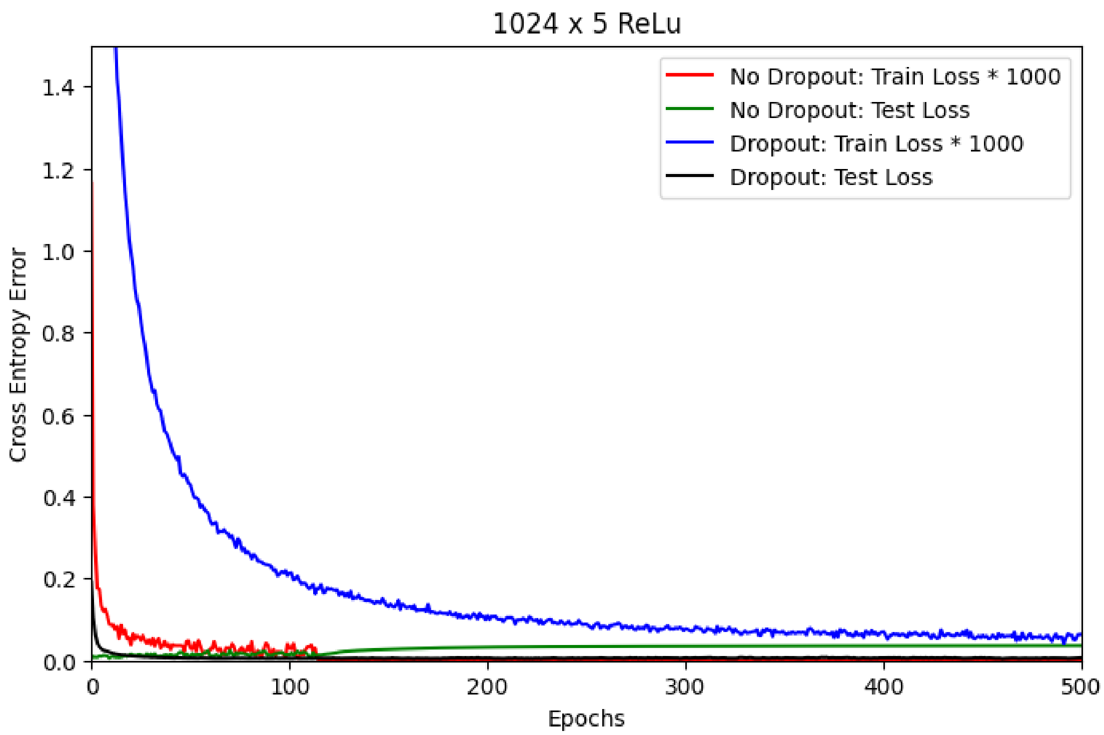
[140]:
```
# download MNIST dataand set up the dataloader
train_dataloader, test_dataloader = mnist_dataloader(200, 1000)

RLKHeI_NDrp.apply(kaiming_he_initializer)
train_RLKHeI_NDrp, test_RLKHeI_NDrp = evaluate_dropout_models(RLKHeI_NDrp,␣
↪adam, 0.0005, 501)
RLKHeI_YDrp.apply(kaiming_he_initializer)
train_RLKHeI_YDrp, test_RLKHeI_YDrp = evaluate_dropout_models(RLKHeI_YDrp,␣
↪adam, 0.0001, 501)
```

```
 Mod : RLKHeI_NDrp   Optimizer: ADAM


Epoch : 0  train loss: 0.001165 test loss : 0.000013  test acc : 0.095610
Epoch : 100  train loss: 0.000031 test loss : 0.000012  test acc : 0.098110
Epoch : 200  train loss: 0.000000 test loss : 0.000032  test acc : 0.098810
Epoch : 300  train loss: 0.000000 test loss : 0.000035  test acc : 0.098810
Epoch : 400  train loss: 0.000000 test loss : 0.000036  test acc : 0.098810
Epoch : 500  train loss: 0.000000 test loss : 0.000036  test acc : 0.098810
```

```
Done!

 Mod : RLKHeI_YDrp   Optimizer: ADAM


Epoch : 0  train loss: 0.018491 test loss : 0.000191  test acc : 0.062007
Epoch : 100  train loss: 0.000213 test loss : 0.000005  test acc : 0.098810
Epoch : 200  train loss: 0.000109 test loss : 0.000006  test acc : 0.098810
Epoch : 300  train loss: 0.000078 test loss : 0.000007  test acc : 0.098410
Epoch : 400  train loss: 0.000070 test loss : 0.000007  test acc : 0.098810
Epoch : 500  train loss: 0.000053 test loss : 0.000007  test acc : 0.098610
Done!
```

[141]:
```python
train_RLKHeI_NDrp_x = list(np.asarray(train_RLKHeI_NDrp) * 1000)
test_RLKHeI_NDrp_x  = list(np.asarray(test_RLKHeI_NDrp) * 1000)
train_RLKHeI_YDrp_x = list(np.asarray(train_RLKHeI_YDrp) * 1000)
test_RLKHeI_YDrp_x  = list(np.asarray(test_RLKHeI_YDrp) * 1000)
```

[147]:
```python
%%time

# Plotting the test accuracy results for Relu with dropoup

x_axis_limit = 500
epochs = range(0,x_axis_limit)
plt.figure(figsize=(8,5))
plt.ylim(0.0,1.5)
plt.xlim(0,x_axis_limit)

plt.plot(epochs, train_RLKHeI_NDrp_x[:x_axis_limit], color='red',
         label='No Dropout: Train Loss * 1000')
plt.plot(epochs, test_RLKHeI_NDrp_x[:x_axis_limit] , color='green',
         label='No Dropout: Test Loss')
plt.plot(epochs, train_RLKHeI_YDrp_x[:x_axis_limit] , color='blue',
         label='Dropout: Train Loss * 1000')
plt.plot(epochs, test_RLKHeI_YDrp_x[:x_axis_limit], color='black',
          label='Dropout: Test Loss')

plt.title('1024 x 5 ReLu')
plt.xlabel('Epochs')
plt.ylabel('Cross Entropy Error')
plt.legend()
plt.show()
```




```
CPU times: user 247 ms, sys: 12.2 ms, total: 259 ms
Wall time: 252 ms
```

0.7 Convert the notebook to HTML

[143]:
```
# %%shell
# jupyter nbconvert --to html /content/DLS_HW_1_PT.ipynb
```

1 The End.