Load MNIST Data

In [37]:
```
#MNIST dataset downloaded from Kaggle
#https://www.kaggle.com/c/digit-recognizer/data

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


#Read training data from CSV
d0 = pd.read_csv('./mnist_train.csv')

# print first five rows of d0.
#print(d0.head(5))

# save the labels into a variable l.
y = d0['label']

# Drop the label feature and store the pixel data in d.
X_train = d0.drop("label",axis=1)
```

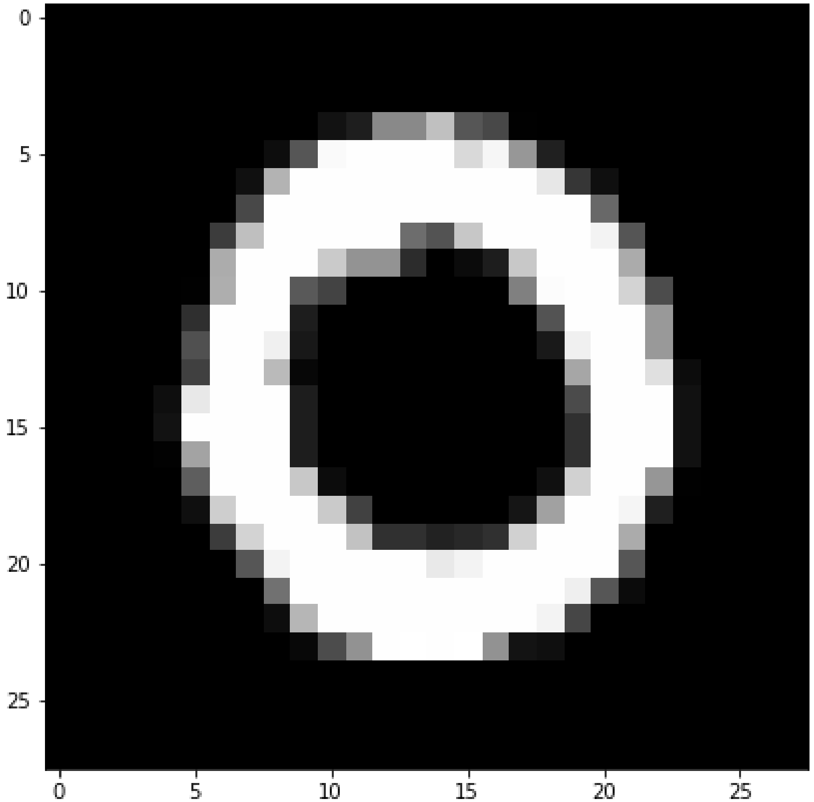
In [38]:
```
print(X_train.shape)
print(y.shape)
```

```
(42000, 784)
(42000,)
```

In [39]:
```
# display or plot a number.
plt.figure(figsize=(7,7))
idx = 1
```

```
grid_data = X_train.iloc[idx].as_matrix().reshape(28,28)  # reshape fro
m 1d to 2d pixel array
plt.imshow(grid_data, interpolation = "none", cmap = "gray")
plt.show()

print(y[idx])
```

```
C:\Users\starlord\Miniconda3\lib\site-packages\ipykernel_launcher.py:5:
FutureWarning: Method .as_matrix will be removed in a future version. U
se .values instead.
  """
```



```
0
```

2D Visualization using PCA

In [40]:

```
# Pick first 15K data-points to work on for time-effeciency.
#Excercise: Perform the same analysis on all of 42K data-points.

labels = y.head(15000)
data = X_train.head(15000)

print("the shape of sample data = ", data.shape)
```

```
the shape of sample data =  (15000, 784)
```

In [41]:

```
# Data-preprocessing: Standardizing the data

from sklearn.preprocessing import StandardScaler
standardized_data = StandardScaler().fit_transform(data)
print(standardized_data.shape)
```

```
C:\Users\starlord\Miniconda3\lib\site-packages\sklearn\preprocessing\da
ta.py:625: DataConversionWarning: Data with input dtype int64 were all
converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
```

```
(15000, 784)
```

```
C:\Users\starlord\Miniconda3\lib\site-packages\sklearn\base.py:462: Dat
aConversionWarning: Data with input dtype int64 were all converted to f
loat64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
```

In [42]:

```
#find the co-variance matrix which is : A^T * A
sample_data = standardized_data

# matrix multiplication using numpy
covar_matrix = np.matmul(sample_data.T , sample_data)

print ( "The shape of variance matrix = ", covar_matrix.shape)
```

```
The shape of variance matrix =  (784, 784)
```

In [43]:
```
# finding the top two eigen-values and corresponding eigen-vectors
# for projecting onto a 2-Dim space.

from scipy.linalg import eigh

# the parameter 'eigvals' is defined (low value to heigh value)
# eigh function will return the eigen values in asending order
# this code generates only the top 2 (782 and 783) eigenvalues.
values, vectors = eigh(covar_matrix, eigvals=(782,783))

print("Shape of eigen vectors = ",vectors.shape)
# converting the eigen vectors into (2,d) shape for easyness of further
 computations
vectors = vectors.T

print("Updated shape of eigen vectors = ",vectors.shape)
# here the vectors[1] represent the eigen vector corresponding 1st prin
cipal eigen vector
# here the vectors[0] represent the eigen vector corresponding 2nd prin
cipal eigen vector
```

```
Shape of eigen vectors =  (784, 2)
Updated shape of eigen vectors =  (2, 784)
```

In [44]:
```
# projecting the original data sample on the plane
#formed by two principal eigen vectors by vector-vector multiplication.

import matplotlib.pyplot as plt
new_coordinates = np.matmul(vectors, sample_data.T)

print (" resultanat new data points' shape ", vectors.shape, "X", sampl
e_data.T.shape," = ", new_coordinates.shape)
```

```
 resultanat new data points' shape  (2, 784) X (784, 15000)  =  (2, 150
00)
```

In [45]:
```
import pandas as pd
```

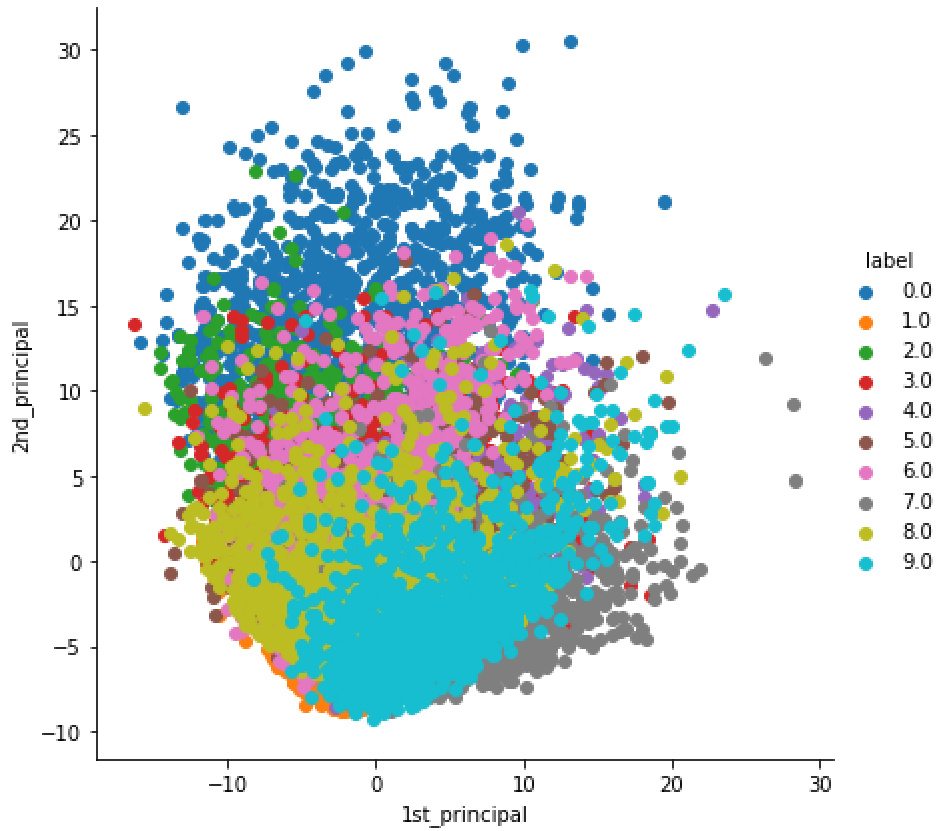
```
# appending label to the 2d projected data
new_coordinates = np.vstack((new_coordinates, labels)).T

# creating a new data frame for ploting the labeled points.
dataframe = pd.DataFrame(data=new_coordinates, columns=("1st_principal"
, "2nd_principal", "label"))
print(dataframe.head())
```

```
   1st_principal  2nd_principal  label
0      -5.558661      -5.043558    1.0
1       6.193635      19.305278    0.0
2      -1.909878      -7.678775    1.0
3       5.525748      -0.464845    4.0
4       6.366527      26.644289    0.0
```

In [46]:

```
# ploting the 2d data points with seaborn
import seaborn as sn
sn.FacetGrid(dataframe, hue="label", size=6).map(plt.scatter, '1st_prin
cipal', '2nd_principal').add_legend()
plt.show()
```

```
C:\Users\starlord\Miniconda3\lib\site-packages\seaborn\axisgrid.py:230:
UserWarning: The `size` paramter has been renamed to `height`; please u
pdate your code.
  warnings.warn(msg, UserWarning)
```



PCA using Scikit-Learn

In [47]:
```
# initializing the pca
from sklearn import decomposition
from sklearn.decomposition import PCA
pca = decomposition.PCA()
```

In [48]:
```
# configuring the parameteres
# the number of components = 2
pca.n_components = 8
pca_data = pca.fit_transform(sample_data)
```

```
# pca_reduced will contain the 2-d projects of simple data
print("shape of pca_reduced.shape = ", pca_data.shape)
```

```
shape of pca_reduced.shape =  (15000, 8)
```

In [110]:
```
##Eigen Vectors & Values
```

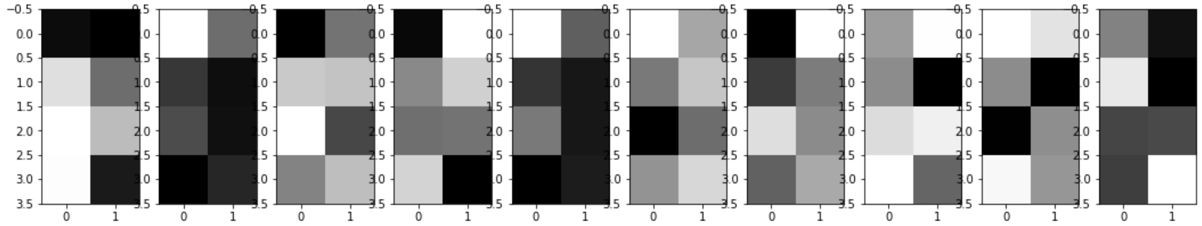
In [49]:
```
print ("Eigen Values =",pca.explained_variance_[0:8])
```

```
Eigen Values = [40.38397838 29.03743968 27.11195251 21.017406   18.0656
3423 15.75990048
 13.68589128 12.70162329]
```

In [50]:
```
print ("Eigen Vectors =",pca.components_[0:8])
```

```
Eigen Vectors = [[ 6.56998190e-18  1.99804625e-18  3.73910215e-19 ... -
0.00000000e+00
  -0.00000000e+00 -0.00000000e+00]
 [ 6.30065305e-17 -5.57161443e-17 -1.15945647e-17 ... -0.00000000e+00
  -0.00000000e+00 -0.00000000e+00]
 [-5.37329591e-17  2.79516539e-17  6.84876996e-18 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 ...
 [-1.22227240e-18  1.44719236e-16 -7.71937851e-17 ... -0.00000000e+00
  -0.00000000e+00 -0.00000000e+00]
 [ 4.87104280e-17  1.79015251e-17  4.35134158e-18 ... -0.00000000e+00
  -0.00000000e+00 -0.00000000e+00]
 [-1.62322198e-17 -5.21530204e-17  7.50229749e-17 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]]
```

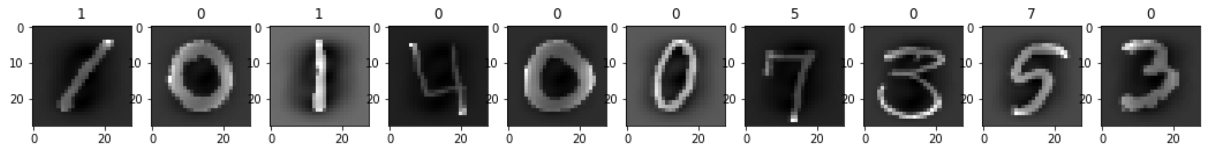
In [54]:
```
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(pca_data[0:10], y[0:10])):
    plt.subplot(1, 11, index + 1)
    plt.imshow(np.reshape(image, (4,2)), cmap=plt.cm.gray)
```



In [92]:
```
approximation = pca.inverse_transform(pca_data)
z=pca_data.shape
z[-1:]
```

Out[92]: (784,)

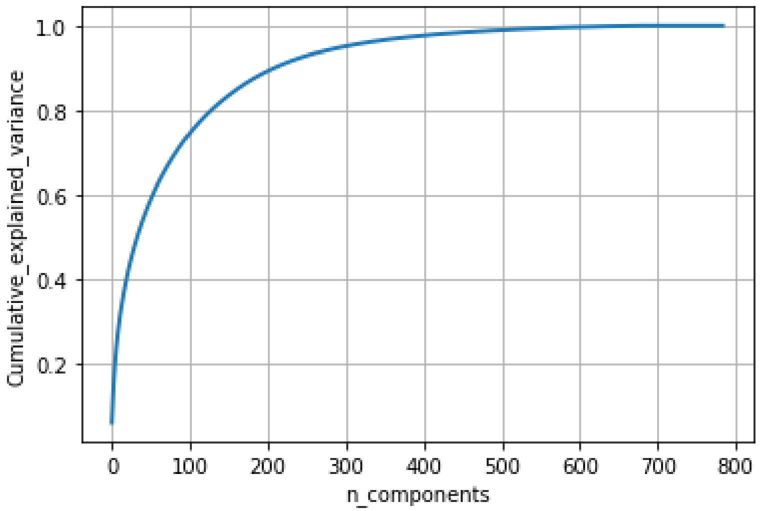
In [99]:
```
y_subset=np.unique(y)
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(approximation[0:10], y[0:10
])):
    plt.subplot(1, 11, index+1)
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.title(y[label+1])
```



PCA for dimensionality redcution (not for visualization)

In [67]:
```
# PCA for dimensionality redcution (non-visualization)

pca.n_components = 784
pca_data = pca.fit_transform(sample_data)
```

```
percentage_var_explained = pca.explained_variance_ / np.sum(pca.explain
ed_variance_);

cum_var_explained = np.cumsum(percentage_var_explained)

# Plot the PCA spectrum
plt.figure(1, figsize=(6, 4))

plt.clf()
plt.plot(cum_var_explained, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()


# If we take 200-dimensions, approx. 90% of variance is expalined.
```



t-SNE using Scikit-Learn

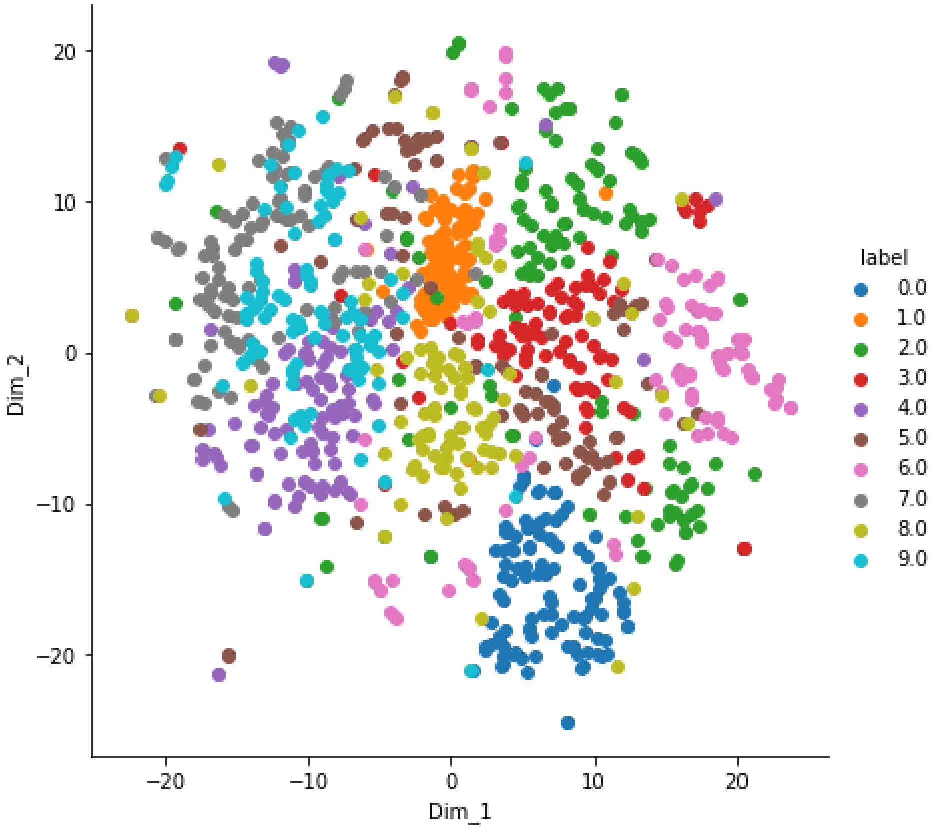
In [77]:
```
# TSNE
#Reference http://colah.github.io/posts/2014-10-Visualizing-MNIST/

from sklearn.manifold import TSNE
import time

# Picking the top 1000 points as TSNE takes a lot of time for 15K point
s
#Preprocessing step
data_1000 = standardized_data[0:1000,:]
labels_1000 = labels[0:1000]


time_start = time.time()
#Parameterization and Model Tranform
# the number of components = 2, default perplexity = 30, default learni
ng rate = 200
# default Maximum number of iterations for the optimization = 1000
model = TSNE(n_components=2, random_state=0, perplexity=100)

tsne_data = model.fit_transform(data_1000)

print (' t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_
start))
```

```
 t-SNE done! Time elapsed: 16.8439781665802 seconds
```

In [78]:
```
# creating a new data frame which help us in ploting the result data
tsne_data = np.vstack((tsne_data.T, labels_1000)).T
tsne_df = pd.DataFrame(data=tsne_data, columns=("Dim_1", "Dim_2", "labe
l"))

# Ploting the result of tsne
sn.FacetGrid(tsne_df, hue="label", size=6).map(plt.scatter, 'Dim_1', 'D
im_2').add_legend()
plt.show()
tsne_data.shape
```

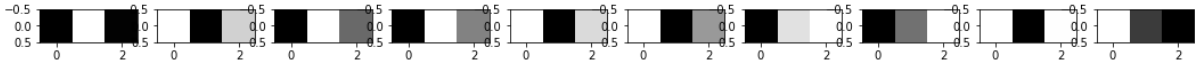
```
C:\Users\starlord\Miniconda3\lib\site-packages\seaborn\axisgrid.py:230:
UserWarning: The `size` paramter has been renamed to `height`; please u
```

```
pdate your code.
  warnings.warn(msg, UserWarning)
```



Out[78]: (1000, 3)

In [79]:
```
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(tsne_data[0:10], y[0:10])):
    plt.subplot(1, 11, index + 1)
    plt.imshow(np.reshape(image, (1,3)), cmap=plt.cm.gray)
```



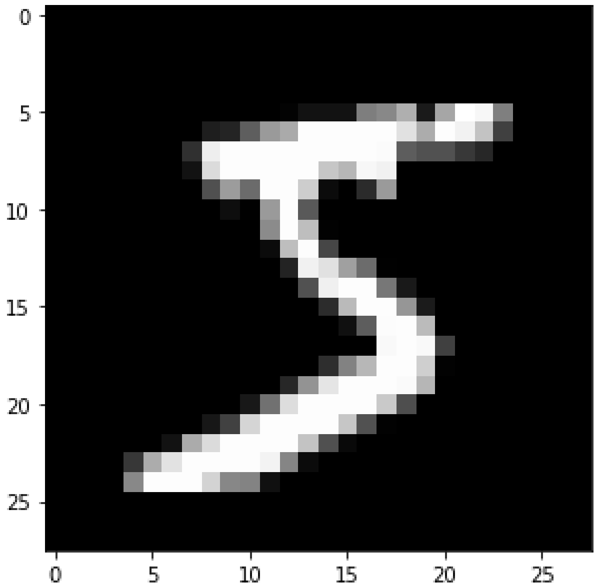
In [92]:
```
#Defining all libraries here for Autoencoder and t-SNE
from torchvision import datasets
import torchvision.transforms as transforms
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torchvision.datasets as dset
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.autograd import Variable
import torch.nn.functional as F

from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
from matplotlib.cbook import get_sample_data
from PIL import ImageFile

# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# load the training and test datasets
train_data = datasets.MNIST(root='data', train=True,
                                   download=True, transform=transform)
test_data = datasets.MNIST(root='data', train=False,
                                  download=True, transform=transform)
```

In [93]:
```
# Create training and test dataloaders

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
```

```
# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch
_size, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_s
ize, num_workers=num_workers)
```

In [81]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()

# get one image from the batch
img = np.squeeze(images[0])

fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(111)
ax.imshow(img, cmap='gray')
```

Out[81]:
```
<matplotlib.image.AxesImage at 0x21b58290cc0>
```



In [82]:
```python
# define the NN architecture
class Autoencoder(nn.Module):
    def __init__(self, encoding_dim):
        super(Autoencoder, self).__init__()
        ## encoder ##
        # linear layer (784 -> encoding_dim)
        self.fc1 = nn.Linear(28 * 28, encoding_dim)

        ## decoder ##
        # linear layer (encoding_dim -> input size)
        self.fc2 = nn.Linear(encoding_dim, 28*28)


    def forward(self, x):
        encoded = self.fc1(x)
        decoded = self.fc2(encoded)
        return decoded
        # add layer, with relu activation function
        #x = F.relu(self.fc1(x))
        # output layer (sigmoid for scaling from 0 to 1)
```

```
        #x = F.sigmoid(self.fc2(x))
        return decoded

# initialize the NN
encoding_dim = 8
model = Autoencoder(encoding_dim)
print(model)
```

```
Autoencoder(
  (fc1): Linear(in_features=784, out_features=8, bias=True)
  (fc2): Linear(in_features=8, out_features=784, bias=True)
)
```

In [83]:

```
# specify loss function
criterion = nn.MSELoss()

# specify loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [84]:

```
# number of epochs to train the model
n_epochs = 10
print('Training Start')
for epoch in range(1, n_epochs+1):
    # monitor training loss
    train_loss = 0.0

    ###################
    # train the model #
    ###################
    for data in train_loader:
        # _ stands in for labels, here
        images, _ = data
        # flatten images
        images = images.view(images.size(0), -1)
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to
 the model
        outputs = model(images)
```

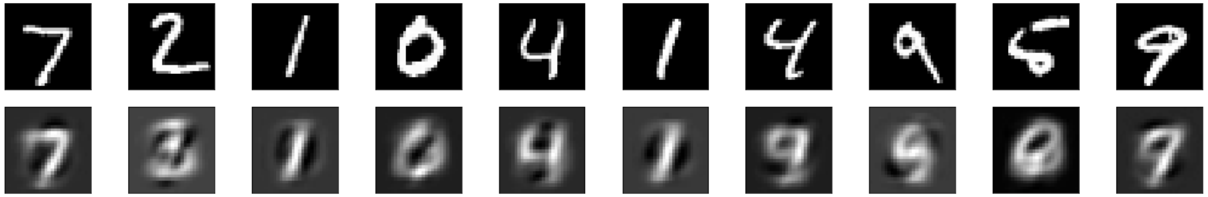
```
        # calculate the loss
        loss = criterion(outputs, images)
        # backward pass: compute gradient of the loss with respect to m
odel parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*images.size(0)

    # print avg training statistics
    train_loss = train_loss/len(train_loader)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch,
        train_loss
        ))
```

```
Epoch: 1 	Training Loss: 1.000614
Epoch: 2 	Training Loss: 0.764980
Epoch: 3 	Training Loss: 0.762036
Epoch: 4 	Training Loss: 0.761769
Epoch: 5 	Training Loss: 0.761126
Epoch: 6 	Training Loss: 0.760979
Epoch: 7 	Training Loss: 0.760897
Epoch: 8 	Training Loss: 0.760748
Epoch: 9 	Training Loss: 0.760657
Epoch: 10 	Training Loss: 0.760587
```

In [85]:

```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()

images_flatten = images.view(images.size(0), -1)
# get sample outputs
output = model(images_flatten)
# prep images for display
images = images.numpy()

# output is resized into a batch of images
```

```
output = output.view(batch_size, 1, 28, 28)
# use detach when it's an output that requires_grad
output = output.detach().numpy()

# plot the first ten input images and then reconstructed images
fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True, f
igsize=(25,4))

# input images on top row, reconstructions on bottom
for images, row in zip([images, output], axes):
    for img, ax in zip(images, row):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```



Simple Linear encoder is similar to a PCA but results are more accurate as compared to a PCA

#Adding Weight initialization

In [86]:
```
# define the NN architecture
class Autoencoder(nn.Module):
    def __init__(self, encoding_dim):
        super(Autoencoder, self).__init__()
        ## encoder ##
        # linear layer (784 -> encoding_dim)
        self.fc1 = nn.Linear(28 * 28, encoding_dim)
        self.fc1.weight.data.normal_(0,0.1)
        self.fc1.bias.data.zero_()
        ## decoder ##
        # linear layer (encoding_dim -> input size)
```

```
        self.fc2 = nn.Linear(encoding_dim, 28*28)
        self.fc1.weight.data.normal_(0,0.1)
        self.fc1.bias.data.zero_()


    def forward(self, x):
        encoded = self.fc1(x)
        decoded = self.fc2(encoded)
        return decoded

# initialize the NN
encoding_dim = 8
model = Autoencoder(encoding_dim)
print(model)
```

```
Autoencoder(
  (fc1): Linear(in_features=784, out_features=8, bias=True)
  (fc2): Linear(in_features=8, out_features=784, bias=True)
)
```

In [87]:
```
# specify loss function
criterion = nn.MSELoss()

# specify loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [88]:
```
# number of epochs to train the model
n_epochs = 10

print('Training Start')
for epoch in range(1, n_epochs+1):
    # monitor training loss
    train_loss = 0.0

    ###################
    # train the model #
    ###################
    for data in train_loader:
        # _ stands in for labels, here
```

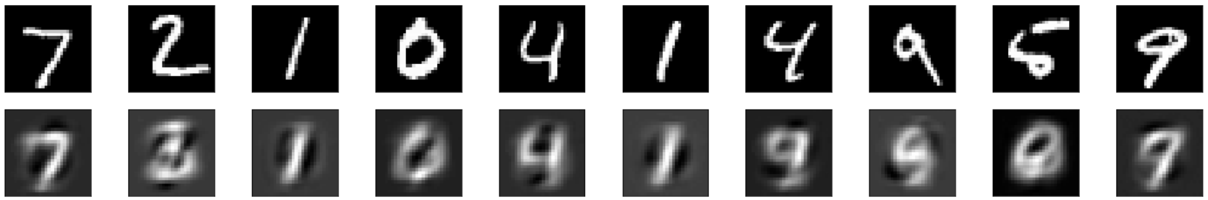
```
        images, _ = data
        # flatten images
        images = images.view(images.size(0), -1)
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to
 the model
        outputs = model(images)
        # calculate the loss
        loss = criterion(outputs, images)
        # backward pass: compute gradient of the loss with respect to m
odel parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*images.size(0)

    # print avg training statistics
    train_loss = train_loss/len(train_loader)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch,
        train_loss
        ))
```

```
Epoch: 1 	Training Loss: 1.090112
Epoch: 2 	Training Loss: 0.770243
Epoch: 3 	Training Loss: 0.762785
Epoch: 4 	Training Loss: 0.761506
Epoch: 5 	Training Loss: 0.761116
Epoch: 6 	Training Loss: 0.760945
Epoch: 7 	Training Loss: 0.760869
Epoch: 8 	Training Loss: 0.760680
Epoch: 9 	Training Loss: 0.760616
Epoch: 10 	Training Loss: 0.760519
```

In [89]:

```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()
```

```python
images_flatten = images.view(images.size(0), -1)
# get sample outputs
output = model(images_flatten)
# prep images for display
images = images.numpy()

# output is resized into a batch of images
output = output.view(batch_size, 1, 28, 28)
# use detach when it's an output that requires_grad
output = output.detach().numpy()

# plot the first ten input images and then reconstructed images
fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True, f
igsize=(25,4))

# input images on top row, reconstructions on bottom
for images, row in zip([images, output], axes):
    for img, ax in zip(images, row):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```



In [90]:
```python
##Results aren't better with weight initialization but may be with diff
erent weight iterations different results might have been received
```

In [ ]:
```python
#With t-SNE for MNIST visulatilzation with a linear Autoencoder
```

In [95]:
```python
mnist_train = dset.MNIST("./", train=True, transform=transforms.ToTenso
r(), target_transform=None, download=True)
mnist_test = dset.MNIST("./", train=False, transform=transforms.ToTenso
```

```
r(),target_transform=None, download=True)

train_loader = torch.utils.data.DataLoader(mnist_train,batch_size=batch
_size, shuffle=True,num_workers=2,drop_last=True)
test_loader = torch.utils.data.DataLoader(mnist_test,batch_size=batch_s
ize, shuffle=False,num_workers=2,drop_last=True)

class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder,self).__init__()
        self.encoder = nn.Linear(28*28,8)
        self.decoder = nn.Linear(8,28*28)

    def forward(self,x):
        x = x.view(batch_size,-1)
        encoded = self.encoder(x)
        out = self.decoder(encoded).view(batch_size,1,28,28)

        return encoded,out

model = Autoencoder().cuda()

loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

print("Training Start")

for i in range(n_epochs):
    for j,[image,label] in enumerate(train_loader):
        x = Variable(image).cuda()

        optimizer.zero_grad()
        _,output = model.forward(x)
        loss = loss_func(output,x)
        loss.backward()
        optimizer.step()

    if j % 1000 == 0:
        print(loss)
```

```python
print("Test data encoding")

total_arr = []
for i in range(1):
    for j,[image,label] in enumerate(test_loader):
        x = Variable(image).cuda()

        optimizer.zero_grad()
        encoded,output = model.forward(x)
        for k in range(batch_size):
            total_arr.append(encoded[k].view(-1).cpu().data.numpy())

        if j >125:
            break

print(len(total_arr))
```

```
Training Start
Test data encoding
2540

------Starting to plot------
```

In [96]:

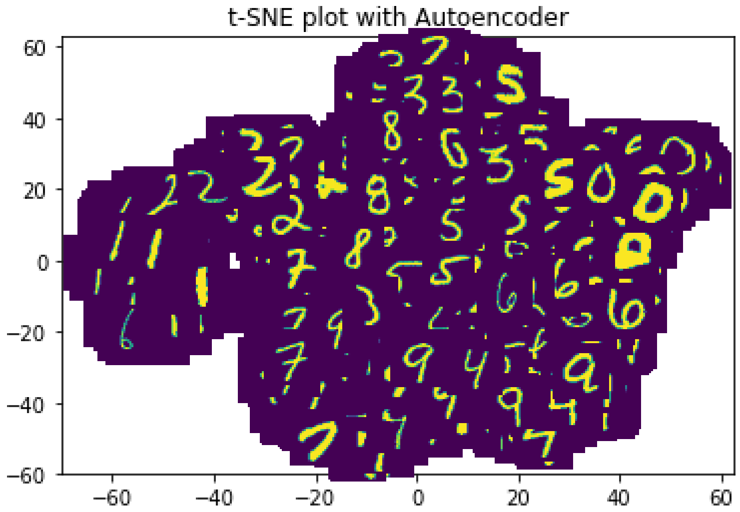
```python
print("\n------Starting TSNE------\n")

tsne_model = TSNE(n_components=2, init='pca',random_state=0)
result = tsne_model.fit_transform(total_arr)

print("\n------TSNE Done------\n")

def imscatter(x, y, image, ax=None, zoom=1):
    if ax is None:
        ax = plt.gca()
    try:
        image = image
    except TypeError:
        # Likely already an array...
```

```
            pass
        im = OffsetImage(image)
        x, y = np.atleast_1d(x, y)
        artists = []
        for x0, y0 in zip(x, y):
            ab = AnnotationBbox(im, (x0, y0), xycoords='data', frameon=Fals
e)
            artists.append(ax.add_artist(ab))
        ax.update_datalim(np.column_stack([x, y]))
        ax.autoscale()
        return artists
```

```
------Starting TSNE------


------TSNE Done------
```

In [98]:

```
mnist_test = dset.MNIST("./", train=False,target_transform=None, downlo
ad=True)

for i in range(len(result)):
    #print("{}/{}".format(i,len(result)))
    image = mnist_test[i][0]
    imscatter(result[i,0],result[i,1], image=image ,zoom=0.2)
    plt.title('t-SNE plot with Autoencoder')
plt.show()
```



In [73]:
```
##Non-linear Autoencoder by adding activation function
```

In [16]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Autoencoder(nn.Module):
    def __init__(self, encoding_dim):
        super(Autoencoder, self).__init__()
        ## encoder ##
        # linear layer (784 -> encoding_dim)
        self.fc1 = nn.Linear(28 * 28, encoding_dim)

        ## decoder ##
        # linear layer (encoding_dim -> input size)
        self.fc2 = nn.Linear(encoding_dim, 28*28)


    def forward(self, x):
        #x = self.fc1(x)
        #x = self.fc2(x)
        #return decoded
```

```
        # add layer, with relu activation function
        x = F.relu(self.fc1(x))
        # output layer (sigmoid for scaling from 0 to 1)
        x = F.sigmoid(self.fc2(x))
        return x

# initialize the NN
encoding_dim = 8
model = Autoencoder(encoding_dim)
print(model)
```

```
Autoencoder(
  (fc1): Linear(in_features=784, out_features=8, bias=True)
  (fc2): Linear(in_features=8, out_features=784, bias=True)
)
```

In [17]:

```
# specify loss function
criterion = nn.MSELoss()

# specify loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [18]:

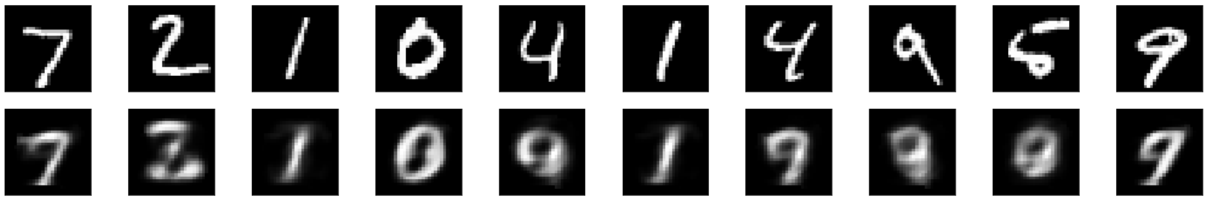
```
# number of epochs to train the model
n_epochs = 20

for epoch in range(1, n_epochs+1):
    # monitor training loss
    train_loss = 0.0

    ###################
    # train the model #
    ###################
    for data in train_loader:
        # _ stands in for labels, here
        images, _ = data
        # flatten images
        images = images.view(images.size(0), -1)
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
```

```
        # forward pass: compute predicted outputs by passing inputs to
 the model
        outputs = model(images)
        # calculate the loss
        loss = criterion(outputs, images)
        # backward pass: compute gradient of the loss with respect to m
odel parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*images.size(0)

    # print avg training statistics
    train_loss = train_loss/len(train_loader)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch,
        train_loss
        ))
```

```
Epoch: 1 	Training Loss: 1.240061
Epoch: 2 	Training Loss: 0.911659
Epoch: 3 	Training Loss: 0.875029
Epoch: 4 	Training Loss: 0.864565
Epoch: 5 	Training Loss: 0.860102
Epoch: 6 	Training Loss: 0.857605
Epoch: 7 	Training Loss: 0.855962
Epoch: 8 	Training Loss: 0.854788
Epoch: 9 	Training Loss: 0.853897
Epoch: 10 	Training Loss: 0.853194
Epoch: 11 	Training Loss: 0.852653
Epoch: 12 	Training Loss: 0.852237
Epoch: 13 	Training Loss: 0.851889
Epoch: 14 	Training Loss: 0.851597
Epoch: 15 	Training Loss: 0.851351
Epoch: 16 	Training Loss: 0.851141
Epoch: 17 	Training Loss: 0.850957
Epoch: 18 	Training Loss: 0.850798
Epoch: 19 	Training Loss: 0.850660
Epoch: 20 	Training Loss: 0.850539
```

In [19]:

```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()

images_flatten = images.view(images.size(0), -1)
# get sample outputs
output = model(images_flatten)
# prep images for display
images = images.numpy()

# output is resized into a batch of images
output = output.view(batch_size, 1, 28, 28)
# use detach when it's an output that requires_grad
output = output.detach().numpy()

# plot the first ten input images and then reconstructed images
fig, axes = plt.subplots(nrows=2, ncols=10, sharex=True, sharey=True, f
igsize=(25,4))

# input images on top row, reconstructions on bottom
for images, row in zip([images, output], axes):
    for img, ax in zip(images, row):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```



In [99]:

```
#Results seem worse than the linear encoder as compared to non-linear a
utoencoder. PCA results were similar to the linear encoder
```