q1

April 3, 2020

1 Question 1, PCA ANALYSIS

[40]:
```
url="/home/abhishek/dev/Semester_2/SMAI/Assignments/Assignment_3/dataset"
```

[41]:
```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from skimage import transform,io
import warnings
warnings.filterwarnings('ignore')
from mpl_toolkits.mplot3d import Axes3D
```

[42]:
```
class PCA:

    def read_image(self,url):
        print("Reading Image from directory...")
        dir_list = os.listdir(url)
        faces = []
        image_labels = []
        for image in dir_list:
            labels = image.split("_")
            image_labels.append(labels[0])

        for image in dir_list:
            img = io.imread(url+"/"+image)
            img = img.astype(np.uint8)
            # converting to grayscale
            rgb_weights = [0.2989, 0.5870, 0.1141]
            grayscale_image = np.dot(img[...,:3], rgb_weights)
            #normalizing image
            small_grey = transform.resize(grayscale_image, (64,64),␣
↪mode='symmetric', preserve_range=True)
            reshape_img = small_grey.reshape(1, 4096)
            faces.append(reshape_img[0])
        X_train = np.asarray(faces)
```

```
        # print(self.X_train[0])
        print(X_train.shape)
        print("Reading Image from directory Done...")
        return image_labels,X_train

    def apply_pca(self,X):
        print("Applying PCA.......")
        eig_val, eig_mat = np.linalg.eig(np.cov(X))
        idx = eig_val.argsort()[::-1]
        eig_val = eig_val[idx]
        eig_mat = eig_mat[:,idx]
        print("Eigen Matrix calculation done..")
        # taking principal components
        M = []
        print(eig_mat.shape)
        full_pc = eig_mat.shape[0]
        for numpc in range(0,full_pc,10):
            eigen_coeff = eig_mat[:,range(numpc)]
            score_mat = np.dot(eigen_coeff.T,X)
            final_score = np.dot(eigen_coeff,score_mat).T
            # print(final_score.shape)
            val = np.linalg.norm(X.T-final_score,'fro')
            M.append(val)
            # print(val)

        mse = np.asarray(M)
        print("Applying PCA done..............")
        return mse,eig_mat,eig_val

    def plot_mse_vs_principal_component(self,mse,eig_mat):
        full_pc = eig_mat.shape[1]
        plt.figure()
        plt.plot(range(0,full_pc,10),mse,'r')
        plt.show()

    def no_of_component_such_that_mse_less_than_20(self,eig_mat,X):
#         N = 20 #from observation no of principal compenent = 50
        eigen_coeff = eig_mat[:,range(24)]
        score_mat = np.dot(eigen_coeff.T,X)
        final = np.dot(eigen_coeff,score_mat).T
        final = final*255
        final_score = final.astype(int)
        fig,axes = plt.subplots(4,5,figsize=(9,9),subplot_kw={'xticks':[],␣
↪'yticks':[]},gridspec_kw=dict(hspace=0.01, wspace=0.01))
        for i, ax in enumerate(axes.flat):
            ax.imshow(final_score[i].reshape(64,64),cmap='gray')
```

```
        plt.show()
        return final_score


    def scatter_plot_pca(self,X_new,X):
        #1-D plot
        val=0
        plt.plot(X[:,0],np.zeros_like(X[:,0])+val,'o')
        plt.plot(X_new[:,0],np.zeros_like(X_new[:,0])+val,"x")
        plt.show()

        # 2-D plot
        plt.scatter(X[:,0],X[:,1], alpha=0.2)
        plt.scatter(X_new[:,0],X_new[:,1], alpha=0.8)#=self.image_labels)
        plt.show()

        #3-D plot
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(X[:,0], X[:,1], X[:,2], marker='o')
        ax.scatter(X_new[:,0], X_new[:,1], X_new[:,2], marker='^')
        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_zlabel('Z Label')
        plt.show()
```

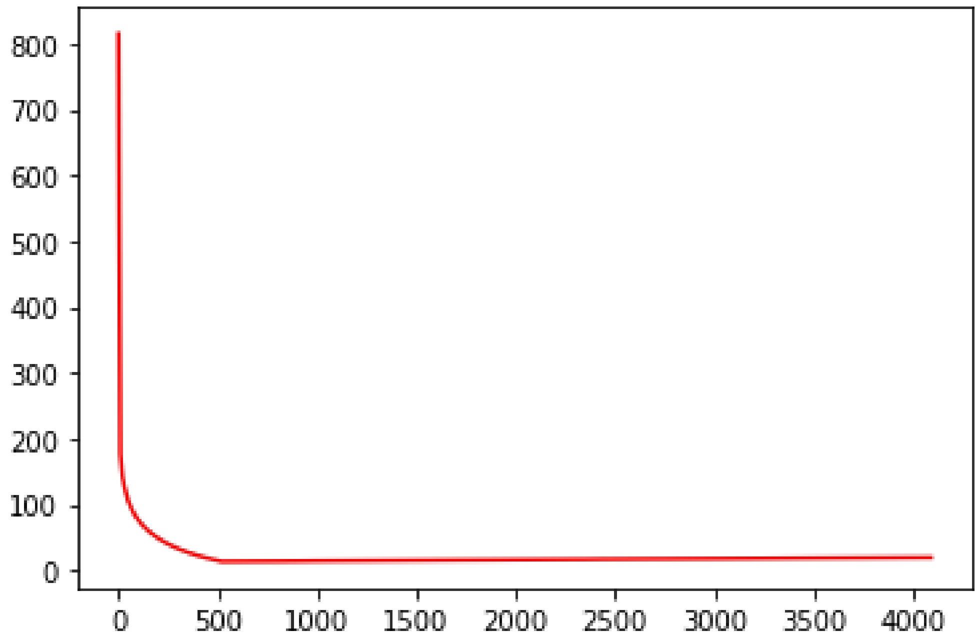
[43]:

```
pca = PCA()
image_label,X_train = pca.read_image(url)
#normalization
X = X_train
X_train = X_train/255
#applying PCA
mse,eig_mat,eig_val = pca.apply_pca(X_train.T)
```

```
Reading Image from directory…
(520, 4096)
Reading Image from directory Done…
Applying PCA…
Eigen Matrix calculation done..
(4096, 4096)
Applying PCA done…
```

[44]:

```
pca.plot_mse_vs_principal_component(mse,eig_mat)
```



[45]:
```
final_score = pca.no_of_component_such_that_mse_less_than_20(eig_mat,X_train.T)
```

[46]:
```
## Check N has Mean Squared error has less than 20%
```

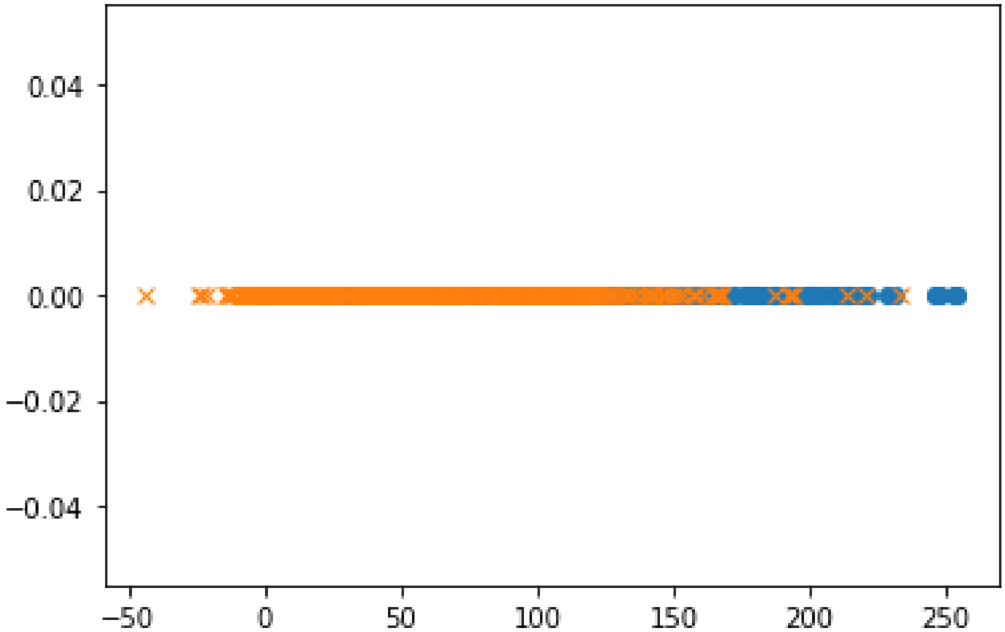
[47]:
```
eigen_val_n = eig_val[0:24]
x = np.sum(eigen_val_n)
y = np.sum(eig_val)
print(x/y)
```

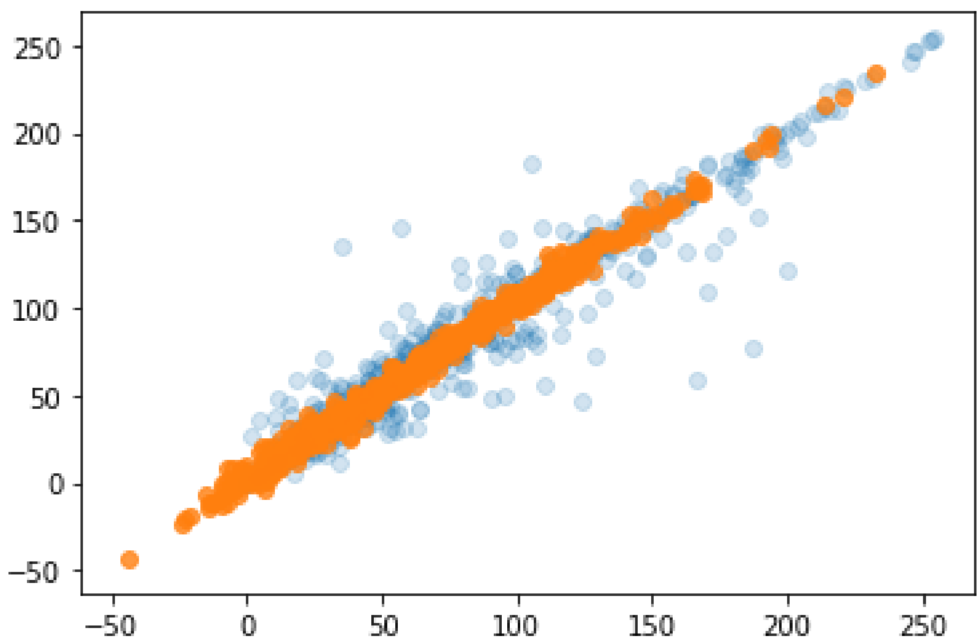
```
(0.804354886673874-5.750783086893538e-33j)
```

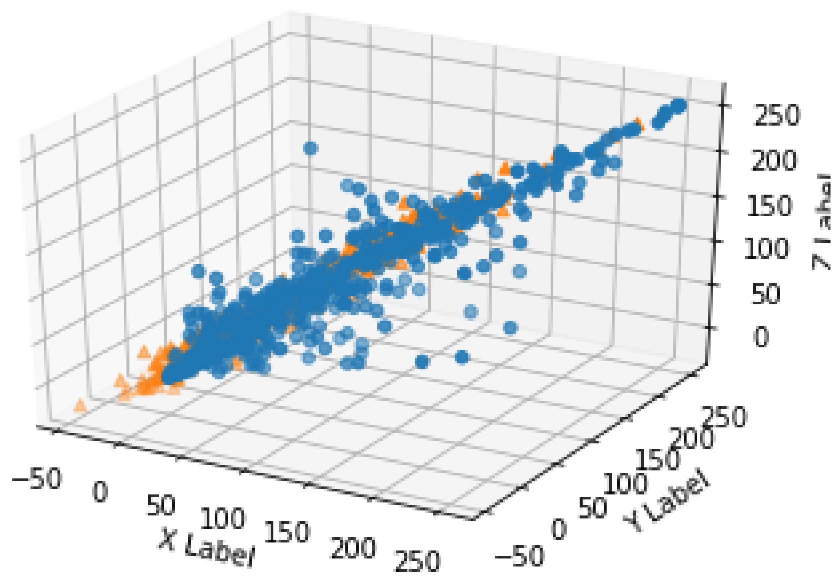
[48]:
```
## for N = 24, mean square error is less than 20%
```

1.1 pca.scatter_plot_pca(final_score,X)

[49]:
```
pca.scatter_plot_pca(final_score,X)
```






−50
0
50
100
150
200
250
X Label
−50
0
50
100
150
200
250
Y Label
0
50
100
150
200
250
Z Label

```
[ ]:
```

q2

April 3, 2020

1 Q2 - Logistic Regression After applying PCA

[15]:
```python
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from skimage import transform,io
from sklearn.metrics import accuracy_score

url_train_csv = "/home/abhishek/dev/Semester_2/SMAI/Assignments/Assignment_3/q2/
↪sample_train_2.txt"
url_test_csv = "/home/abhishek/dev/Semester_2/SMAI/Assignments/Assignment_3/q2/
↪sample_test_2.txt"
sample_output_csv = "/home/abhishek/dev/Semester_2/SMAI/Assignments/
↪Assignment_3/q2/output_of_sample_test_2.txt"

class LogisticRegression:
    def read_csv_file(self,marker):
        image_files = []
        f = None
        if(marker == "train"):
            f = open(url_train_csv,'r')
        elif(marker == "test"):
            f = open(url_test_csv,'r')
        else:
            f = open(sample_output_csv,'r')

        lines = f.readlines()
        for line in lines:
            image_files.append(line.strip())
        return image_files

    def split_image_dir_and_label(self,images_directory,marker):
        if(marker == "train"):
            # print("splitting directory and labels")
            label = []
```

```
            directory = []
            for l in images_directory:
                x = l.split(" ")
                label.append(x[1])
                directory.append(x[0])
            # print("splitting directory and labels done ....")
            return label, directory
        elif(marker == "test"):
            # print("read lines...")
            lines = []
            for l in images_directory:
                lines.append(l)
            # print("reading lines done...")
            return lines

    def read_image_from_directory_to_grayscale(self,directory):
        # print("reading images from directory and downscaling images")
        faces = []
        for i in directory:
            img = io.imread(i)
            img = img.astype(np.uint8)
            # converting to grayscale
            rgb_weights = [0.2989, 0.5870, 0.1141]
            grayscale_image = np.dot(img[...,:3], rgb_weights)
            small_grey = transform.resize(grayscale_image, (64,64),␣
↪mode='symmetric', preserve_range=True)
            reshape_img = small_grey.reshape(1, 4096)
            faces.append(reshape_img[0])
        X = np.asarray(faces)
        # print("reading images from directory and downscaling images done...")
        return X

    def apply_pca(self,X):
        # print("Applying PCA on the image set")
        eig_val, eig_mat = np.linalg.eig(np.cov(X))
        idx = eig_val.argsort()[::-1]
        eig_val = eig_val[idx]
        eig_mat = eig_mat[:,idx]
        eigen_coeff = eig_mat[:,range(50)]
        X_PCA = np.dot(eigen_coeff.T,X)
        # print("Applying PCA on the image set done .......")
        return X_PCA.T

    def sigmoid(self,z):
        return 1/(1+np.exp(-z))

    def cost(self,w,b,X,y,lmd=10):
```

```
        # print("Calculating Cost")
        m = X.shape[0]
        z = np.matmul(X,w) + b
        hx = self.sigmoid(z)
        J = (-1/m)*( np.sum( y * np.log(hx) + (1. - y) * np.log(1.- hx) ) )
        J += (lmd/(2*m))* np.matmul(w,w)
        # print("Calculating Cost done ..")
        return J

    def gradient_descent(self,w,b,X,y,learning_rate=0.
↪01,lmd=10,no_of_iteration=10000):
        # print("Applying Gradient Descent")
        m = X.shape[0]
        # print("Initial cost: {}".format( self.cost(w,b,X,y) ))
        for i in range(no_of_iteration):
            z = np.matmul(X,w) + b
            hx = self.sigmoid(z)
            dw = (1/m)*np.matmul(X.T,hx-y)
            db = (1/m)*np.sum(hx-y)
            factor = 1-( (learning_rate * lmd)/m)
            w = w*factor - learning_rate*dw
            b = b - learning_rate*db
            # if i % 500 == 0:
                # print("Iteration {} cost :{}".format(i,self.cost(w,b,X,y)))
        # print("Final cost: {}".format( self.cost(w,b,X,y) ))
        # print("Applying Gradient Descent done ....")
        return w,b

    def accuracy(self,w,b,X,y):
        # print("Calculating Accuracy")
        m = X.shape[0]
        z = np.matmul(X,w) + b
        hx = self.sigmoid(z)
        pred = np.round(hx)
        correct_pred = (pred==y)
        total = np.sum(correct_pred)
        # print("Calculating Accurcy done .....")
        return (total*100)/m

    def test(self,w,b,X):
        # m = X.shape[0]
        z = np.matmul(X,w)+b
        hx = self.sigmoid(z)
        pred = np.round(hx)
        return pred

lr = LogisticRegression()
```

[ ]:
```
## Applying PCA on the above dataset
```

[2]:
```
lines = lr.read_csv_file("train")
# print(images_directory)
label, directory = lr.split_image_dir_and_label(lines,"train")
# label = np.asarray(label)
# print(label)
unique_labels = np.unique(np.asarray(label))
# print(unique_labels)
X_train = lr.read_image_from_directory_to_grayscale(directory)
# normalizing
X_train = X_train/255
#applying PCA
X_PCA = lr.apply_pca(X_train.T)
# print(X_PCA.shape)
# print(X_PCA[0])
m = X_train.shape[0]
```

[3]:
```
print(X_PCA.shape)
```

```
(520, 50)
```

[7]:
```
## Training Multi-Class Classifier for each unique class in Tranining set
```

[4]:
```
weight_label_map = {}
i = 0
THETA = []
BIAS = []

for unique in unique_labels:
    y_train = []
    for l in label:
        if(unique == l):
            y_train.append(1)
        else:
            y_train.append(0)

    weight_label_map[i] = unique
    i = i+1
    y_train = np.asarray(y_train)
    # print(y_train)
    w = np.zeros(X_PCA.shape[1],dtype=np.float64)
    b = 0.0
    w,b = lr.gradient_descent(w,b,X_PCA,y_train)
    THETA.append(w)
    BIAS.append(b)
```

[5]:
```python
## ALL Unique Labels in Training set
```

[6]:
```python
print(weight_label_map)
```

```
{0: '000', 1: '001', 2: '002', 3: '003', 4: '004', 5: '005', 6: '006', 7: '007'}
```

[12]:
```python
## Applying PCA on Test Data set
```

[7]:
```python
THETA = np.asarray(THETA)
BIAS = np.asarray(BIAS)

lr1 = LogisticRegression()
lines = lr1.read_csv_file("test")
directory = lr1.split_image_dir_and_label(lines,"test")
X_test = lr1.read_image_from_directory_to_grayscale(directory)
X_test = X_test/255
X_Test_PCA = lr1.apply_pca(X_test.T)
# print(X_Test_PCA.shape)
```

[8]:
```python
## Predicting The labels of Test Data by taking maximum sigmoid value of label␣
↪along all classifier
```

[9]:
```python
y_pred = []
for X in X_Test_PCA:
    i = 0
    max_hx = 0.0
    index = 0
    for w,b in zip(THETA,BIAS):
        pred = lr1.test(w,b,X)
        if(pred > max_hx):
            max_hx = pred
            index = i
        i = i+1
    y_pred.append(weight_label_map[index])

y_pred = np.asarray(y_pred)
for prediction in y_pred:
    print(prediction)
```

```
000
000
000
002
001
002
003
005
```

000
000
000
002
000
002
000
000
007
000
000
002
007
000
000
000
000
000
006
006
001
006
000
000
000
003
006
002
007
000
002
000
000
000
004
000
000
003
003
001
000
006
006
001
005
000
000
001

002
000
000
003
002
001
000
000
000
004
000
000
004
007
006
006
007
007
000
000
000
006
000
006
000
006
004
007
000
007
007
002
000
003
005
002
000
003
000
005
000
003
001
007
000
000
005
000

007
000
000
000
000
000
000
000
000
007
002
003
007
000
000
000
003
007
001
000
000
000
007
004
000
000
002
000
002
006
000
005
000
000
001
000
004
000
005
003
003
000
000
006
007
001
000
000

005
000
000
000
004
000
007
005
000
000
000
002
000
000
004
003
001
000
005
000
004
000
000
000
006
000
000
003
000
000
000
000
000
005
005
000
005
007
000
000
000
002
006
007
000
007
004
003

005
002
000
003
000
000
004
000
003
002
004
004
000
000
006
007
000
004
000
001
003
000
000
000
007
003
002
000
003
000
000
000
000
000
005
002
000
006
000
000
005
000
004
000
000
002
005
002

000
000
000
000
000
000
000
002
000
000
005
004
005
000
000
001
007
007
001
000
000
003
003
000
000
000
004
003
007
003
000
002
007
000
007
005
000
000
000
006
004
000
000
007
000
000
007
000

003
000
001
005
005
000
000
005
004
002
000
000
000
004
006
004
000
005
004
007
002
000
007
000
005
002
000
007
005
000
007
002
007
004
003
000
000
000
003
002
000
000
000
003
000
001
003
000

000
001
000
000
003
000
000
000
007
000
004
007
000
000
004
000
002
001
000
000
000
000
000
000
007
000
000
000
004
000
000
007
000
000
002
000
005
000
000
000
000
000
003
005
000
004
000
002

000
000
000
000
005
004
007
000
002
000
000
004
003
005
007
004
004
001
006
004
004
007
005
007
000
000
003
007
000
004
000
004
003
004
007
002
005
003
006
002
000
000
002
003
005
000
004
000

000
000
003
005
007
007
000
000
000
000
000
003
000
002
002
006
000
003
003
003
001
000
000
000
006
000
003
007
001
000
000
000
004
000
002
000
005
000
000
000
006
002
004
000
000
001
007
005

```
007
000
004
003
006
000
000
000
004
005
001
002
002
007
003
000
004
007
000
000
000
003
004
004
002
000
000
000
002
001
007
003
```

[16]:
```python
labels = lr.read_csv_file("label")
```

[18]:
```python
print(len(labels))
```

```
520
```

[20]:
```python
labels = np.asarray(labels)
```

[21]:
```python
print(labels.shape)
```

```
(520,)
```

[ ]:
```python
## Accuracy Score for predicting the labels
```

[22]:
```python
accuracy_score(labels,y_pred)
```

[22]: 0.5846153846153846

```
[ ]:
```

q3

April 3, 2020

0.1 Question 3 - MNIST Classsification using PyTorch

[0]:
```python
import torch
import numpy as np
```

Load Data Sets

[2]:
```python
from torchvision import datasets
import torchvision.transforms as transforms

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

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

# choose 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)

# 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_size,
    num_workers=num_workers)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
data/MNIST/raw/train-images-idx3-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
data/MNIST/raw/train-labels-idx1-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

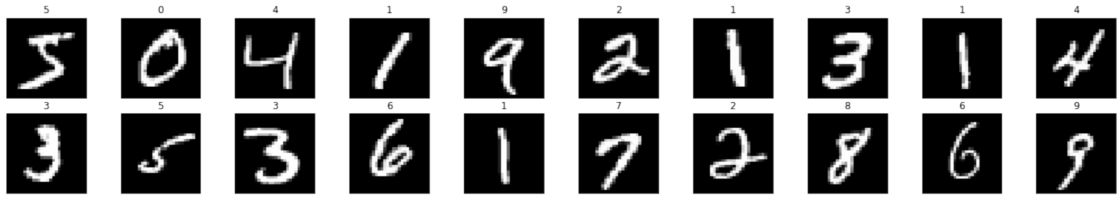
```
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
Processing…
Done!
```

Visualize a batch of Training Set

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

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

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    # print out the correct label for each image
    # .item() gets the value contained in a Tensor
    ax.set_title(str(labels[idx].item()))
```



Define the Network Architecture

[0]:
```
import torch.nn as nn
import torch.nn.functional as F

## Define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 512)
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(512, 512)
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(512, 10)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        return x
```

[5]:
```
# initialize the NN
model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

Specify Loss Function and Optimizer

[0]:
```
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Train the Network

[7]:

```python
n_epochs = 30  # suggest training between 20-50 epochs

model.train() # prep model for training

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

    ###################
    # train the model #
    ###################
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
 ↪parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # print training statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1        Training Loss: 0.804222
Epoch: 2        Training Loss: 0.411707
Epoch: 3        Training Loss: 0.370065
Epoch: 4        Training Loss: 0.348959
Epoch: 5        Training Loss: 0.335502
Epoch: 6        Training Loss: 0.325918
Epoch: 7        Training Loss: 0.318626
Epoch: 8        Training Loss: 0.312826
Epoch: 9        Training Loss: 0.308064
Epoch: 10       Training Loss: 0.304058
Epoch: 11       Training Loss: 0.300624
Epoch: 12       Training Loss: 0.297635
```

```
Epoch: 13       Training Loss: 0.294999
Epoch: 14       Training Loss: 0.292651
Epoch: 15       Training Loss: 0.290539
Epoch: 16       Training Loss: 0.288626
Epoch: 17       Training Loss: 0.286881
Epoch: 18       Training Loss: 0.285279
Epoch: 19       Training Loss: 0.283802
Epoch: 20       Training Loss: 0.282434
Epoch: 21       Training Loss: 0.281160
Epoch: 22       Training Loss: 0.279970
Epoch: 23       Training Loss: 0.278856
Epoch: 24       Training Loss: 0.277808
Epoch: 25       Training Loss: 0.276820
Epoch: 26       Training Loss: 0.275887
Epoch: 27       Training Loss: 0.275002
Epoch: 28       Training Loss: 0.274163
Epoch: 29       Training Loss: 0.273365
Epoch: 30       Training Loss: 0.272604
```

Test the Trained Network¶

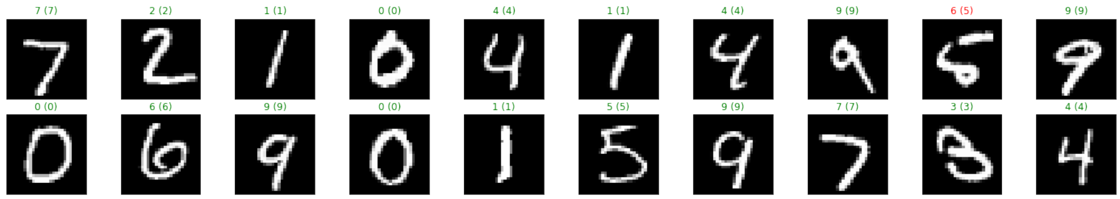
[8]:
```python
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model.eval() # prep model for *evaluation*

for data, target in test_loader:
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct = np.squeeze(pred.eq(target.data.view_as(pred)))
    # calculate test accuracy for each object class
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# calculate and print avg test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))
```

```
for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            str(i), 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 0.274007

Test Accuracy of     0: 98% (962/980)
Test Accuracy of     1: 97% (1109/1135)
Test Accuracy of     2: 88% (918/1032)
Test Accuracy of     3: 90% (914/1010)
Test Accuracy of     4: 92% (909/982)
Test Accuracy of     5: 87% (781/892)
Test Accuracy of     6: 94% (910/958)
Test Accuracy of     7: 92% (946/1028)
Test Accuracy of     8: 88% (864/974)
Test Accuracy of     9: 90% (916/1009)

Test Accuracy (Overall): 92% (9229/10000)
```

Visualize Sample Test Results

[9]:
```
dataiter = iter(test_loader)
images, labels = dataiter.next()

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds = torch.max(output, 1)
# prep images for display
images = images.numpy()

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].
 ↪item())),
                 color=("green" if preds[idx]==labels[idx] else "red"))
```



0.2 MNIST Classification using CNN

[0]:
```python
import numpy as np # to handle matrix and data operation
import pandas as pd # to read csv and handle dataframe

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data
from torch.autograd import Variable

from sklearn.model_selection import train_test_split
```

[0]:
```python
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[0]:
```python
x_train = x_train.reshape(60000,784)
x_test = x_test.reshape(10000,784)
```

[0]:
```python
BATCH_SIZE = 32
```

[0]:
```python
torch_X_train = torch.from_numpy(x_train).type(torch.LongTensor)
torch_y_train = torch.from_numpy(y_train).type(torch.LongTensor)
```

[0]:
```python
torch_X_test = torch.from_numpy(x_test).type(torch.LongTensor)
torch_y_test = torch.from_numpy(y_test).type(torch.LongTensor)
```

[0]:
```python
train = torch.utils.data.TensorDataset(torch_X_train,torch_y_train)
test = torch.utils.data.TensorDataset(torch_X_test,torch_y_test)
```

[0]:
```python
train_loader = torch.utils.data.DataLoader(train, batch_size = BATCH_SIZE, shuffle = False)
test_loader = torch.utils.data.DataLoader(test, batch_size = BATCH_SIZE, shuffle = False)
```

[59]:
```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.linear1 = nn.Linear(784,250)
        self.linear2 = nn.Linear(250,100)
        self.linear3 = nn.Linear(100,10)

    def forward(self,X):
        X = F.relu(self.linear1(X))
        X = F.relu(self.linear2(X))
        X = self.linear3(X)
        return F.log_softmax(X, dim=1)

mlp = MLP()
print(mlp)
```

```
MLP(
  (linear1): Linear(in_features=784, out_features=250, bias=True)
  (linear2): Linear(in_features=250, out_features=100, bias=True)
  (linear3): Linear(in_features=100, out_features=10, bias=True)
)
```

[0]:
```python
def fit(model, train_loader):
    optimizer = torch.optim.Adam(model.parameters())#,lr=0.001, betas=(0.9,0.
↪999))
    error = nn.CrossEntropyLoss()
    EPOCHS = 5
    model.train()
    for epoch in range(EPOCHS):
        correct = 0
        for batch_idx, (X_batch, y_batch) in enumerate(train_loader):
            var_X_batch = Variable(X_batch).float()
            var_y_batch = Variable(y_batch)
            optimizer.zero_grad()
            output = model(var_X_batch)
            loss = error(output, var_y_batch)
            loss.backward()
            optimizer.step()

            # Total correct predictions
            predicted = torch.max(output.data, 1)[1]
            correct += (predicted == var_y_batch).sum()
            #print(correct)
            if batch_idx % 50 == 0:
                print('Epoch : {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}\t Accuracy:{:.
↪3f}%'.format(
```

```
                    epoch, batch_idx*len(X_batch), len(train_loader.dataset),␣
↪100.*batch_idx / len(train_loader), (loss.data), float(correct*100) /␣
↪float(BATCH_SIZE*(batch_idx+1))))
```

[67]:
```
fit(mlp, train_loader)
```

```
Epoch : 0 [0/60000 (0%)]        Loss: 0.028779   Accuracy:100.000%
Epoch : 0 [1600/60000 (3%)]     Loss: 0.135546   Accuracy:96.875%
Epoch : 0 [3200/60000 (5%)]     Loss: 0.161492   Accuracy:97.061%
Epoch : 0 [4800/60000 (8%)]     Loss: 0.029837   Accuracy:97.144%
Epoch : 0 [6400/60000 (11%)]    Loss: 0.251311   Accuracy:97.233%
Epoch : 0 [8000/60000 (13%)]    Loss: 0.288707   Accuracy:96.987%
Epoch : 0 [9600/60000 (16%)]    Loss: 0.001341   Accuracy:96.564%
Epoch : 0 [11200/60000 (19%)]   Loss: 0.147621   Accuracy:96.270%
Epoch : 0 [12800/60000 (21%)]   Loss: 0.162637   Accuracy:96.213%
Epoch : 0 [14400/60000 (24%)]   Loss: 0.040754   Accuracy:96.182%
Epoch : 0 [16000/60000 (27%)]   Loss: 0.053642   Accuracy:96.214%
Epoch : 0 [17600/60000 (29%)]   Loss: 0.021809   Accuracy:96.257%
Epoch : 0 [19200/60000 (32%)]   Loss: 0.007951   Accuracy:96.339%
Epoch : 0 [20800/60000 (35%)]   Loss: 0.117186   Accuracy:96.376%
Epoch : 0 [22400/60000 (37%)]   Loss: 0.001032   Accuracy:96.438%
Epoch : 0 [24000/60000 (40%)]   Loss: 0.240103   Accuracy:96.451%
Epoch : 0 [25600/60000 (43%)]   Loss: 0.097887   Accuracy:96.497%
Epoch : 0 [27200/60000 (45%)]   Loss: 0.023727   Accuracy:96.500%
Epoch : 0 [28800/60000 (48%)]   Loss: 0.015119   Accuracy:96.528%
Epoch : 0 [30400/60000 (51%)]   Loss: 0.070183   Accuracy:96.537%
Epoch : 0 [32000/60000 (53%)]   Loss: 0.012500   Accuracy:96.544%
Epoch : 0 [33600/60000 (56%)]   Loss: 0.110607   Accuracy:96.530%
Epoch : 0 [35200/60000 (59%)]   Loss: 0.357193   Accuracy:96.540%
Epoch : 0 [36800/60000 (61%)]   Loss: 0.010914   Accuracy:96.549%
Epoch : 0 [38400/60000 (64%)]   Loss: 0.176304   Accuracy:96.555%
Epoch : 0 [40000/60000 (67%)]   Loss: 0.098055   Accuracy:96.530%
Epoch : 0 [41600/60000 (69%)]   Loss: 0.044533   Accuracy:96.539%
Epoch : 0 [43200/60000 (72%)]   Loss: 0.099969   Accuracy:96.551%
Epoch : 0 [44800/60000 (75%)]   Loss: 0.024746   Accuracy:96.576%
Epoch : 0 [46400/60000 (77%)]   Loss: 0.333618   Accuracy:96.550%
Epoch : 0 [48000/60000 (80%)]   Loss: 0.118122   Accuracy:96.565%
Epoch : 0 [49600/60000 (83%)]   Loss: 0.100130   Accuracy:96.555%
Epoch : 0 [51200/60000 (85%)]   Loss: 0.101556   Accuracy:96.549%
Epoch : 0 [52800/60000 (88%)]   Loss: 0.163493   Accuracy:96.555%
Epoch : 0 [54400/60000 (91%)]   Loss: 0.001940   Accuracy:96.550%
Epoch : 0 [56000/60000 (93%)]   Loss: 0.151699   Accuracy:96.552%
Epoch : 0 [57600/60000 (96%)]   Loss: 0.085508   Accuracy:96.577%
Epoch : 0 [59200/60000 (99%)]   Loss: 0.006413   Accuracy:96.607%
Epoch : 1 [0/60000 (0%)]        Loss: 0.066838   Accuracy:96.875%
Epoch : 1 [1600/60000 (3%)]     Loss: 0.038393   Accuracy:95.895%
Epoch : 1 [3200/60000 (5%)]     Loss: 0.088629   Accuracy:96.566%
```

```
Epoch : 1 [4800/60000 (8%)]     Loss: 0.073845   Accuracy:96.978%
Epoch : 1 [6400/60000 (11%)]    Loss: 0.323887   Accuracy:97.015%
Epoch : 1 [8000/60000 (13%)]    Loss: 0.080831   Accuracy:97.049%
Epoch : 1 [9600/60000 (16%)]    Loss: 0.028484   Accuracy:97.103%
Epoch : 1 [11200/60000 (19%)]   Loss: 0.217653   Accuracy:97.044%
Epoch : 1 [12800/60000 (21%)]   Loss: 0.015726   Accuracy:97.015%
Epoch : 1 [14400/60000 (24%)]   Loss: 0.031854   Accuracy:96.993%
Epoch : 1 [16000/60000 (27%)]   Loss: 0.102694   Accuracy:97.050%
Epoch : 1 [17600/60000 (29%)]   Loss: 0.314560   Accuracy:97.164%
Epoch : 1 [19200/60000 (32%)]   Loss: 0.043562   Accuracy:97.192%
Epoch : 1 [20800/60000 (35%)]   Loss: 0.078116   Accuracy:97.211%
Epoch : 1 [22400/60000 (37%)]   Loss: 0.010395   Accuracy:97.214%
Epoch : 1 [24000/60000 (40%)]   Loss: 0.008545   Accuracy:97.241%
Epoch : 1 [25600/60000 (43%)]   Loss: 0.087680   Accuracy:97.285%
Epoch : 1 [27200/60000 (45%)]   Loss: 0.016307   Accuracy:97.264%
Epoch : 1 [28800/60000 (48%)]   Loss: 0.070700   Accuracy:97.295%
Epoch : 1 [30400/60000 (51%)]   Loss: 0.283281   Accuracy:97.315%
Epoch : 1 [32000/60000 (53%)]   Loss: 0.124514   Accuracy:97.331%
Epoch : 1 [33600/60000 (56%)]   Loss: 0.077984   Accuracy:97.265%
Epoch : 1 [35200/60000 (59%)]   Loss: 0.018130   Accuracy:97.272%
Epoch : 1 [36800/60000 (61%)]   Loss: 0.063739   Accuracy:97.277%
Epoch : 1 [38400/60000 (64%)]   Loss: 0.262766   Accuracy:97.250%
Epoch : 1 [40000/60000 (67%)]   Loss: 0.040512   Accuracy:97.252%
Epoch : 1 [41600/60000 (69%)]   Loss: 0.088825   Accuracy:97.235%
Epoch : 1 [43200/60000 (72%)]   Loss: 0.161225   Accuracy:97.213%
Epoch : 1 [44800/60000 (75%)]   Loss: 0.050391   Accuracy:97.232%
Epoch : 1 [46400/60000 (77%)]   Loss: 0.577136   Accuracy:97.194%
Epoch : 1 [48000/60000 (80%)]   Loss: 0.154128   Accuracy:97.183%
Epoch : 1 [49600/60000 (83%)]   Loss: 0.199059   Accuracy:97.155%
Epoch : 1 [51200/60000 (85%)]   Loss: 0.163394   Accuracy:97.146%
Epoch : 1 [52800/60000 (88%)]   Loss: 0.163188   Accuracy:97.161%
Epoch : 1 [54400/60000 (91%)]   Loss: 0.007887   Accuracy:97.163%
Epoch : 1 [56000/60000 (93%)]   Loss: 0.123494   Accuracy:97.150%
Epoch : 1 [57600/60000 (96%)]   Loss: 0.084630   Accuracy:97.167%
Epoch : 1 [59200/60000 (99%)]   Loss: 0.041318   Accuracy:97.197%
Epoch : 2 [0/60000 (0%)]        Loss: 0.101924   Accuracy:93.750%
Epoch : 2 [1600/60000 (3%)]     Loss: 0.192111   Accuracy:96.078%
Epoch : 2 [3200/60000 (5%)]     Loss: 0.070167   Accuracy:97.123%
Epoch : 2 [4800/60000 (8%)]     Loss: 0.023145   Accuracy:97.413%
Epoch : 2 [6400/60000 (11%)]    Loss: 0.164375   Accuracy:97.217%
Epoch : 2 [8000/60000 (13%)]    Loss: 0.206731   Accuracy:97.261%
Epoch : 2 [9600/60000 (16%)]    Loss: 0.013814   Accuracy:97.228%
Epoch : 2 [11200/60000 (19%)]   Loss: 0.189980   Accuracy:97.302%
Epoch : 2 [12800/60000 (21%)]   Loss: 0.094958   Accuracy:97.319%
Epoch : 2 [14400/60000 (24%)]   Loss: 0.015213   Accuracy:97.367%
Epoch : 2 [16000/60000 (27%)]   Loss: 0.093988   Accuracy:97.287%
Epoch : 2 [17600/60000 (29%)]   Loss: 0.621408   Accuracy:97.221%
Epoch : 2 [19200/60000 (32%)]   Loss: 0.108640   Accuracy:97.223%
```

```
Epoch : 2 [20800/60000 (35%)]   Loss: 0.021036   Accuracy:97.216%
Epoch : 2 [22400/60000 (37%)]   Loss: 0.001247   Accuracy:97.209%
Epoch : 2 [24000/60000 (40%)]   Loss: 0.015268   Accuracy:97.233%
Epoch : 2 [25600/60000 (43%)]   Loss: 0.101386   Accuracy:97.261%
Epoch : 2 [27200/60000 (45%)]   Loss: 0.131324   Accuracy:97.239%
Epoch : 2 [28800/60000 (48%)]   Loss: 0.069168   Accuracy:97.243%
Epoch : 2 [30400/60000 (51%)]   Loss: 0.025075   Accuracy:97.282%
Epoch : 2 [32000/60000 (53%)]   Loss: 0.016453   Accuracy:97.300%
Epoch : 2 [33600/60000 (56%)]   Loss: 0.020328   Accuracy:97.273%
Epoch : 2 [35200/60000 (59%)]   Loss: 0.872610   Accuracy:97.298%
Epoch : 2 [36800/60000 (61%)]   Loss: 0.101088   Accuracy:97.296%
Epoch : 2 [38400/60000 (64%)]   Loss: 0.088246   Accuracy:97.294%
Epoch : 2 [40000/60000 (67%)]   Loss: 0.119826   Accuracy:97.302%
Epoch : 2 [41600/60000 (69%)]   Loss: 0.052212   Accuracy:97.324%
Epoch : 2 [43200/60000 (72%)]   Loss: 0.325394   Accuracy:97.305%
Epoch : 2 [44800/60000 (75%)]   Loss: 0.062563   Accuracy:97.310%
Epoch : 2 [46400/60000 (77%)]   Loss: 0.133759   Accuracy:97.273%
Epoch : 2 [48000/60000 (80%)]   Loss: 0.219807   Accuracy:97.271%
Epoch : 2 [49600/60000 (83%)]   Loss: 0.019319   Accuracy:97.288%
Epoch : 2 [51200/60000 (85%)]   Loss: 0.045573   Accuracy:97.273%
Epoch : 2 [52800/60000 (88%)]   Loss: 0.129453   Accuracy:97.265%
Epoch : 2 [54400/60000 (91%)]   Loss: 0.037412   Accuracy:97.270%
Epoch : 2 [56000/60000 (93%)]   Loss: 0.061451   Accuracy:97.275%
Epoch : 2 [57600/60000 (96%)]   Loss: 0.268131   Accuracy:97.278%
Epoch : 2 [59200/60000 (99%)]   Loss: 0.029599   Accuracy:97.316%
Epoch : 3 [0/60000 (0%)]        Loss: 0.151179   Accuracy:93.750%
Epoch : 3 [1600/60000 (3%)]     Loss: 0.128003   Accuracy:96.446%
Epoch : 3 [3200/60000 (5%)]     Loss: 0.006657   Accuracy:97.308%
Epoch : 3 [4800/60000 (8%)]     Loss: 0.015271   Accuracy:97.392%
Epoch : 3 [6400/60000 (11%)]    Loss: 0.127014   Accuracy:97.404%
Epoch : 3 [8000/60000 (13%)]    Loss: 0.137492   Accuracy:97.323%
Epoch : 3 [9600/60000 (16%)]    Loss: 0.011746   Accuracy:97.290%
Epoch : 3 [11200/60000 (19%)]   Loss: 0.193034   Accuracy:97.391%
Epoch : 3 [12800/60000 (21%)]   Loss: 0.151988   Accuracy:97.506%
Epoch : 3 [14400/60000 (24%)]   Loss: 0.001826   Accuracy:97.561%
Epoch : 3 [16000/60000 (27%)]   Loss: 0.038850   Accuracy:97.493%
Epoch : 3 [17600/60000 (29%)]   Loss: 0.005036   Accuracy:97.533%
Epoch : 3 [19200/60000 (32%)]   Loss: 0.041748   Accuracy:97.577%
Epoch : 3 [20800/60000 (35%)]   Loss: 0.061989   Accuracy:97.552%
Epoch : 3 [22400/60000 (37%)]   Loss: 0.000083   Accuracy:97.562%
Epoch : 3 [24000/60000 (40%)]   Loss: 0.010684   Accuracy:97.562%
Epoch : 3 [25600/60000 (43%)]   Loss: 0.035633   Accuracy:97.554%
Epoch : 3 [27200/60000 (45%)]   Loss: 0.003687   Accuracy:97.536%
Epoch : 3 [28800/60000 (48%)]   Loss: 0.059841   Accuracy:97.562%
Epoch : 3 [30400/60000 (51%)]   Loss: 0.055981   Accuracy:97.581%
Epoch : 3 [32000/60000 (53%)]   Loss: 0.057697   Accuracy:97.596%
Epoch : 3 [33600/60000 (56%)]   Loss: 0.099345   Accuracy:97.589%
Epoch : 3 [35200/60000 (59%)]   Loss: 0.004503   Accuracy:97.593%
```

```
Epoch : 3 [36800/60000 (61%)]    Loss: 0.132729   Accuracy:97.597%
Epoch : 3 [38400/60000 (64%)]    Loss: 0.078958   Accuracy:97.580%
Epoch : 3 [40000/60000 (67%)]    Loss: 0.049452   Accuracy:97.577%
Epoch : 3 [41600/60000 (69%)]    Loss: 0.001719   Accuracy:97.574%
Epoch : 3 [43200/60000 (72%)]    Loss: 0.175857   Accuracy:97.580%
Epoch : 3 [44800/60000 (75%)]    Loss: 0.034069   Accuracy:97.589%
Epoch : 3 [46400/60000 (77%)]    Loss: 0.021272   Accuracy:97.599%
Epoch : 3 [48000/60000 (80%)]    Loss: 0.064088   Accuracy:97.620%
Epoch : 3 [49600/60000 (83%)]    Loss: 0.003137   Accuracy:97.625%
Epoch : 3 [51200/60000 (85%)]    Loss: 0.014169   Accuracy:97.628%
Epoch : 3 [52800/60000 (88%)]    Loss: 0.221221   Accuracy:97.625%
Epoch : 3 [54400/60000 (91%)]    Loss: 0.001268   Accuracy:97.612%
Epoch : 3 [56000/60000 (93%)]    Loss: 0.063965   Accuracy:97.596%
Epoch : 3 [57600/60000 (96%)]    Loss: 0.254212   Accuracy:97.604%
Epoch : 3 [59200/60000 (99%)]    Loss: 0.000153   Accuracy:97.633%
Epoch : 4 [0/60000 (0%)]         Loss: 0.077204   Accuracy:96.875%
Epoch : 4 [1600/60000 (3%)]      Loss: 0.160978   Accuracy:97.059%
Epoch : 4 [3200/60000 (5%)]      Loss: 0.022128   Accuracy:97.463%
Epoch : 4 [4800/60000 (8%)]      Loss: 0.087849   Accuracy:97.641%
Epoch : 4 [6400/60000 (11%)]     Loss: 0.282174   Accuracy:97.512%
Epoch : 4 [8000/60000 (13%)]     Loss: 0.022924   Accuracy:97.572%
Epoch : 4 [9600/60000 (16%)]     Loss: 0.057789   Accuracy:97.456%
Epoch : 4 [11200/60000 (19%)]    Loss: 0.173873   Accuracy:97.480%
Epoch : 4 [12800/60000 (21%)]    Loss: 0.177891   Accuracy:97.498%
Epoch : 4 [14400/60000 (24%)]    Loss: 0.058999   Accuracy:97.568%
Epoch : 4 [16000/60000 (27%)]    Loss: 0.087360   Accuracy:97.493%
Epoch : 4 [17600/60000 (29%)]    Loss: 0.002993   Accuracy:97.544%
Epoch : 4 [19200/60000 (32%)]    Loss: 0.007460   Accuracy:97.577%
Epoch : 4 [20800/60000 (35%)]    Loss: 0.172938   Accuracy:97.576%
Epoch : 4 [22400/60000 (37%)]    Loss: 0.003647   Accuracy:97.597%
Epoch : 4 [24000/60000 (40%)]    Loss: 0.003558   Accuracy:97.616%
Epoch : 4 [25600/60000 (43%)]    Loss: 0.123521   Accuracy:97.651%
Epoch : 4 [27200/60000 (45%)]    Loss: 0.062108   Accuracy:97.650%
Epoch : 4 [28800/60000 (48%)]    Loss: 0.079932   Accuracy:97.683%
Epoch : 4 [30400/60000 (51%)]    Loss: 0.002716   Accuracy:97.703%
Epoch : 4 [32000/60000 (53%)]    Loss: 0.218044   Accuracy:97.687%
Epoch : 4 [33600/60000 (56%)]    Loss: 0.009926   Accuracy:97.693%
Epoch : 4 [35200/60000 (59%)]    Loss: 0.001960   Accuracy:97.707%
Epoch : 4 [36800/60000 (61%)]    Loss: 0.003491   Accuracy:97.706%
Epoch : 4 [38400/60000 (64%)]    Loss: 0.193213   Accuracy:97.684%
Epoch : 4 [40000/60000 (67%)]    Loss: 0.121122   Accuracy:97.677%
Epoch : 4 [41600/60000 (69%)]    Loss: 0.058635   Accuracy:97.672%
Epoch : 4 [43200/60000 (72%)]    Loss: 0.121910   Accuracy:97.668%
Epoch : 4 [44800/60000 (75%)]    Loss: 0.020887   Accuracy:97.711%
Epoch : 4 [46400/60000 (77%)]    Loss: 0.099990   Accuracy:97.676%
Epoch : 4 [48000/60000 (80%)]    Loss: 0.167490   Accuracy:97.677%
Epoch : 4 [49600/60000 (83%)]    Loss: 0.096079   Accuracy:97.671%
Epoch : 4 [51200/60000 (85%)]    Loss: 0.009618   Accuracy:97.687%
```

```
Epoch : 4 [52800/60000 (88%)]   Loss: 0.322359   Accuracy:97.714%
Epoch : 4 [54400/60000 (91%)]   Loss: 0.001315   Accuracy:97.702%
Epoch : 4 [56000/60000 (93%)]   Loss: 0.065532   Accuracy:97.717%
Epoch : 4 [57600/60000 (96%)]   Loss: 0.120293   Accuracy:97.729%
Epoch : 4 [59200/60000 (99%)]   Loss: 0.000217   Accuracy:97.758%
```

[68]:

```
def evaluate(model):
#model = mlp
    correct = 0
    for test_imgs, test_labels in test_loader:
        #print(test_imgs.shape)
        test_imgs = Variable(test_imgs).float()
        output = model(test_imgs)
        predicted = torch.max(output,1)[1]
        correct += (predicted == test_labels).sum()
    print("Test accuracy:{:.3f}% ".format( float(correct) /
 ↪(len(test_loader)*BATCH_SIZE)))
evaluate(mlp)
```

```
Test accuracy:0.971%
```

##Since a CNN needs a image shape as input let's reshape our flatten images to real image

[69]:

```
torch_X_train = torch_X_train.view(-1, 1,28,28).float()
torch_X_test = torch_X_test.view(-1,1,28,28).float()
print(torch_X_train.shape)
print(torch_X_test.shape)

# Pytorch train and test sets
train = torch.utils.data.TensorDataset(torch_X_train,torch_y_train)
test = torch.utils.data.TensorDataset(torch_X_test,torch_y_test)

# data loader
train_loader = torch.utils.data.DataLoader(train, batch_size = BATCH_SIZE,
 ↪shuffle = False)
test_loader = torch.utils.data.DataLoader(test, batch_size = BATCH_SIZE,
 ↪shuffle = False)MB
```

```
torch.Size([60000, 1, 28, 28])
torch.Size([10000, 1, 28, 28])
```

[70]:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=5)
        self.conv3 = nn.Conv2d(32,64, kernel_size=5)
        self.fc1 = nn.Linear(3*3*64, 256)
```

```
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        #x = F.dropout(x, p=0.5, training=self.training)
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.dropout(x, p=0.5, training=self.training)
        x = F.relu(F.max_pool2d(self.conv3(x),2))
        x = F.dropout(x, p=0.5, training=self.training)
        x = x.view(-1,3*3*64 )
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

cnn = CNN()
print(cnn)

it = iter(train_loader)
X_batch, y_batch = next(it)
print(cnn.forward(X_batch).shape)
```

```
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=10, bias=True)
)
torch.Size([32, 10])
```

[71]:
```
fit(cnn,train_loader)
```

```
Epoch : 0 [0/60000 (0%)]        Loss: 24.703955   Accuracy:6.250%
Epoch : 0 [1600/60000 (3%)]     Loss: 1.980315    Accuracy:17.279%
Epoch : 0 [3200/60000 (5%)]     Loss: 1.460487    Accuracy:29.115%
Epoch : 0 [4800/60000 (8%)]     Loss: 0.748759    Accuracy:38.949%
Epoch : 0 [6400/60000 (11%)]    Loss: 1.238455    Accuracy:45.553%
Epoch : 0 [8000/60000 (13%)]    Loss: 0.865389    Accuracy:50.461%
Epoch : 0 [9600/60000 (16%)]    Loss: 0.764462    Accuracy:54.485%
Epoch : 0 [11200/60000 (19%)]   Loss: 0.602340    Accuracy:58.191%
Epoch : 0 [12800/60000 (21%)]   Loss: 0.548422    Accuracy:60.793%
Epoch : 0 [14400/60000 (24%)]   Loss: 0.491586    Accuracy:63.089%
Epoch : 0 [16000/60000 (27%)]   Loss: 0.702923    Accuracy:64.814%
Epoch : 0 [17600/60000 (29%)]   Loss: 0.591373    Accuracy:66.640%
Epoch : 0 [19200/60000 (32%)]   Loss: 0.427941    Accuracy:68.225%
Epoch : 0 [20800/60000 (35%)]   Loss: 0.508253    Accuracy:69.772%
Epoch : 0 [22400/60000 (37%)]   Loss: 0.220711    Accuracy:71.122%
```

```
Epoch : 0 [24000/60000 (40%)]   Loss: 0.182511   Accuracy:72.121%
Epoch : 0 [25600/60000 (43%)]   Loss: 0.578731   Accuracy:73.174%
Epoch : 0 [27200/60000 (45%)]   Loss: 0.405716   Accuracy:74.104%
Epoch : 0 [28800/60000 (48%)]   Loss: 0.335528   Accuracy:75.035%
Epoch : 0 [30400/60000 (51%)]   Loss: 0.524058   Accuracy:75.710%
Epoch : 0 [32000/60000 (53%)]   Loss: 0.522807   Accuracy:76.346%
Epoch : 0 [33600/60000 (56%)]   Loss: 0.353773   Accuracy:77.019%
Epoch : 0 [35200/60000 (59%)]   Loss: 0.348661   Accuracy:77.677%
Epoch : 0 [36800/60000 (61%)]   Loss: 0.335008   Accuracy:78.307%
Epoch : 0 [38400/60000 (64%)]   Loss: 0.295484   Accuracy:78.841%
Epoch : 0 [40000/60000 (67%)]   Loss: 0.330925   Accuracy:79.359%
Epoch : 0 [41600/60000 (69%)]   Loss: 0.761555   Accuracy:79.790%
Epoch : 0 [43200/60000 (72%)]   Loss: 0.308168   Accuracy:80.204%
Epoch : 0 [44800/60000 (75%)]   Loss: 0.039641   Accuracy:80.661%
Epoch : 0 [46400/60000 (77%)]   Loss: 0.347005   Accuracy:81.032%
Epoch : 0 [48000/60000 (80%)]   Loss: 0.695292   Accuracy:81.404%
Epoch : 0 [49600/60000 (83%)]   Loss: 0.664559   Accuracy:81.742%
Epoch : 0 [51200/60000 (85%)]   Loss: 0.267460   Accuracy:82.083%
Epoch : 0 [52800/60000 (88%)]   Loss: 0.457143   Accuracy:82.418%
Epoch : 0 [54400/60000 (91%)]   Loss: 0.123103   Accuracy:82.742%
Epoch : 0 [56000/60000 (93%)]   Loss: 0.171987   Accuracy:83.040%
Epoch : 0 [57600/60000 (96%)]   Loss: 0.293099   Accuracy:83.320%
Epoch : 0 [59200/60000 (99%)]   Loss: 0.139004   Accuracy:83.656%
Epoch : 1 [0/60000 (0%)]        Loss: 0.197437   Accuracy:93.750%
Epoch : 1 [1600/60000 (3%)]     Loss: 0.182057   Accuracy:93.137%
Epoch : 1 [3200/60000 (5%)]     Loss: 0.449420   Accuracy:94.028%
Epoch : 1 [4800/60000 (8%)]     Loss: 0.116307   Accuracy:93.729%
Epoch : 1 [6400/60000 (11%)]    Loss: 0.029951   Accuracy:93.797%
Epoch : 1 [8000/60000 (13%)]    Loss: 0.101050   Accuracy:93.875%
Epoch : 1 [9600/60000 (16%)]    Loss: 0.333615   Accuracy:93.625%
Epoch : 1 [11200/60000 (19%)]   Loss: 0.426801   Accuracy:93.643%
Epoch : 1 [12800/60000 (21%)]   Loss: 0.298322   Accuracy:93.462%
Epoch : 1 [14400/60000 (24%)]   Loss: 0.108020   Accuracy:93.528%
Epoch : 1 [16000/60000 (27%)]   Loss: 0.474114   Accuracy:93.469%
Epoch : 1 [17600/60000 (29%)]   Loss: 0.277469   Accuracy:93.455%
Epoch : 1 [19200/60000 (32%)]   Loss: 0.163615   Accuracy:93.506%
Epoch : 1 [20800/60000 (35%)]   Loss: 0.227395   Accuracy:93.568%
Epoch : 1 [22400/60000 (37%)]   Loss: 0.010190   Accuracy:93.630%
Epoch : 1 [24000/60000 (40%)]   Loss: 0.085058   Accuracy:93.633%
Epoch : 1 [25600/60000 (43%)]   Loss: 0.092341   Accuracy:93.606%
Epoch : 1 [27200/60000 (45%)]   Loss: 0.161767   Accuracy:93.548%
Epoch : 1 [28800/60000 (48%)]   Loss: 0.029576   Accuracy:93.573%
Epoch : 1 [30400/60000 (51%)]   Loss: 0.205116   Accuracy:93.589%
Epoch : 1 [32000/60000 (53%)]   Loss: 0.252389   Accuracy:93.581%
Epoch : 1 [33600/60000 (56%)]   Loss: 0.318719   Accuracy:93.583%
Epoch : 1 [35200/60000 (59%)]   Loss: 0.124187   Accuracy:93.583%
Epoch : 1 [36800/60000 (61%)]   Loss: 0.110199   Accuracy:93.636%
Epoch : 1 [38400/60000 (64%)]   Loss: 0.135114   Accuracy:93.633%
```

```
Epoch : 1 [40000/60000 (67%)]   Loss: 0.337480   Accuracy:93.608%
Epoch : 1 [41600/60000 (69%)]   Loss: 0.151904   Accuracy:93.637%
Epoch : 1 [43200/60000 (72%)]   Loss: 0.201784   Accuracy:93.651%
Epoch : 1 [44800/60000 (75%)]   Loss: 0.120788   Accuracy:93.670%
Epoch : 1 [46400/60000 (77%)]   Loss: 0.631979   Accuracy:93.638%
Epoch : 1 [48000/60000 (80%)]   Loss: 0.330198   Accuracy:93.644%
Epoch : 1 [49600/60000 (83%)]   Loss: 0.092436   Accuracy:93.667%
Epoch : 1 [51200/60000 (85%)]   Loss: 0.046334   Accuracy:93.693%
Epoch : 1 [52800/60000 (88%)]   Loss: 0.316572   Accuracy:93.701%
Epoch : 1 [54400/60000 (91%)]   Loss: 0.114109   Accuracy:93.695%
Epoch : 1 [56000/60000 (93%)]   Loss: 0.394695   Accuracy:93.727%
Epoch : 1 [57600/60000 (96%)]   Loss: 0.083526   Accuracy:93.792%
Epoch : 1 [59200/60000 (99%)]   Loss: 0.015199   Accuracy:93.860%
Epoch : 2 [0/60000 (0%)]        Loss: 0.217455   Accuracy:90.625%
Epoch : 2 [1600/60000 (3%)]     Loss: 0.406698   Accuracy:94.056%
Epoch : 2 [3200/60000 (5%)]     Loss: 0.423212   Accuracy:94.864%
Epoch : 2 [4800/60000 (8%)]     Loss: 0.065729   Accuracy:94.454%
Epoch : 2 [6400/60000 (11%)]    Loss: 0.006236   Accuracy:94.325%
Epoch : 2 [8000/60000 (13%)]    Loss: 0.204239   Accuracy:94.410%
Epoch : 2 [9600/60000 (16%)]    Loss: 0.303900   Accuracy:94.248%
Epoch : 2 [11200/60000 (19%)]   Loss: 0.255757   Accuracy:94.436%
Epoch : 2 [12800/60000 (21%)]   Loss: 0.152741   Accuracy:94.459%
Epoch : 2 [14400/60000 (24%)]   Loss: 0.082564   Accuracy:94.471%
Epoch : 2 [16000/60000 (27%)]   Loss: 0.283941   Accuracy:94.374%
Epoch : 2 [17600/60000 (29%)]   Loss: 0.014378   Accuracy:94.380%
Epoch : 2 [19200/60000 (32%)]   Loss: 0.083600   Accuracy:94.452%
Epoch : 2 [20800/60000 (35%)]   Loss: 0.360928   Accuracy:94.556%
Epoch : 2 [22400/60000 (37%)]   Loss: 0.102351   Accuracy:94.593%
Epoch : 2 [24000/60000 (40%)]   Loss: 0.109202   Accuracy:94.653%
Epoch : 2 [25600/60000 (43%)]   Loss: 0.092279   Accuracy:94.714%
Epoch : 2 [27200/60000 (45%)]   Loss: 0.241476   Accuracy:94.679%
Epoch : 2 [28800/60000 (48%)]   Loss: 0.107590   Accuracy:94.704%
Epoch : 2 [30400/60000 (51%)]   Loss: 0.188392   Accuracy:94.759%
Epoch : 2 [32000/60000 (53%)]   Loss: 0.298663   Accuracy:94.699%
Epoch : 2 [33600/60000 (56%)]   Loss: 0.165397   Accuracy:94.690%
Epoch : 2 [35200/60000 (59%)]   Loss: 0.109414   Accuracy:94.738%
Epoch : 2 [36800/60000 (61%)]   Loss: 0.111880   Accuracy:94.790%
Epoch : 2 [38400/60000 (64%)]   Loss: 0.247883   Accuracy:94.786%
Epoch : 2 [40000/60000 (67%)]   Loss: 0.241609   Accuracy:94.799%
Epoch : 2 [41600/60000 (69%)]   Loss: 0.274725   Accuracy:94.809%
Epoch : 2 [43200/60000 (72%)]   Loss: 0.335197   Accuracy:94.828%
Epoch : 2 [44800/60000 (75%)]   Loss: 0.005174   Accuracy:94.872%
Epoch : 2 [46400/60000 (77%)]   Loss: 1.024677   Accuracy:94.889%
Epoch : 2 [48000/60000 (80%)]   Loss: 0.283337   Accuracy:94.860%
Epoch : 2 [49600/60000 (83%)]   Loss: 0.264172   Accuracy:94.854%
Epoch : 2 [51200/60000 (85%)]   Loss: 0.145659   Accuracy:94.857%
Epoch : 2 [52800/60000 (88%)]   Loss: 0.662844   Accuracy:94.893%
Epoch : 2 [54400/60000 (91%)]   Loss: 0.017660   Accuracy:94.922%
```

```
Epoch : 2 [56000/60000 (93%)]   Loss: 0.118975   Accuracy:94.960%
Epoch : 2 [57600/60000 (96%)]   Loss: 0.390172   Accuracy:94.961%
Epoch : 2 [59200/60000 (99%)]   Loss: 0.000941   Accuracy:95.016%
Epoch : 3 [0/60000 (0%)]        Loss: 0.039304   Accuracy:96.875%
Epoch : 3 [1600/60000 (3%)]     Loss: 0.274808   Accuracy:94.424%
Epoch : 3 [3200/60000 (5%)]     Loss: 0.082404   Accuracy:95.142%
Epoch : 3 [4800/60000 (8%)]     Loss: 0.180459   Accuracy:95.281%
Epoch : 3 [6400/60000 (11%)]    Loss: 0.055648   Accuracy:95.134%
Epoch : 3 [8000/60000 (13%)]    Loss: 0.032606   Accuracy:95.219%
Epoch : 3 [9600/60000 (16%)]    Loss: 0.057986   Accuracy:94.975%
Epoch : 3 [11200/60000 (19%)]   Loss: 0.275520   Accuracy:95.103%
Epoch : 3 [12800/60000 (21%)]   Loss: 0.070716   Accuracy:94.950%
Epoch : 3 [14400/60000 (24%)]   Loss: 0.027030   Accuracy:94.928%
Epoch : 3 [16000/60000 (27%)]   Loss: 0.066866   Accuracy:94.941%
Epoch : 3 [17600/60000 (29%)]   Loss: 0.028506   Accuracy:94.941%
Epoch : 3 [19200/60000 (32%)]   Loss: 0.219603   Accuracy:94.998%
Epoch : 3 [20800/60000 (35%)]   Loss: 0.051435   Accuracy:94.993%
Epoch : 3 [22400/60000 (37%)]   Loss: 0.011864   Accuracy:95.047%
Epoch : 3 [24000/60000 (40%)]   Loss: 0.177396   Accuracy:95.073%
Epoch : 3 [25600/60000 (43%)]   Loss: 0.050125   Accuracy:95.154%
Epoch : 3 [27200/60000 (45%)]   Loss: 0.215653   Accuracy:95.131%
Epoch : 3 [28800/60000 (48%)]   Loss: 0.357761   Accuracy:95.106%
Epoch : 3 [30400/60000 (51%)]   Loss: 0.303260   Accuracy:95.127%
Epoch : 3 [32000/60000 (53%)]   Loss: 0.435719   Accuracy:95.042%
Epoch : 3 [33600/60000 (56%)]   Loss: 0.038899   Accuracy:95.008%
Epoch : 3 [35200/60000 (59%)]   Loss: 0.157955   Accuracy:95.084%
Epoch : 3 [36800/60000 (61%)]   Loss: 0.065332   Accuracy:95.127%
Epoch : 3 [38400/60000 (64%)]   Loss: 0.087521   Accuracy:95.145%
Epoch : 3 [40000/60000 (67%)]   Loss: 0.119639   Accuracy:95.131%
Epoch : 3 [41600/60000 (69%)]   Loss: 0.089203   Accuracy:95.143%
Epoch : 3 [43200/60000 (72%)]   Loss: 0.412446   Accuracy:95.152%
Epoch : 3 [44800/60000 (75%)]   Loss: 0.103714   Accuracy:95.211%
Epoch : 3 [46400/60000 (77%)]   Loss: 0.355738   Accuracy:95.202%
Epoch : 3 [48000/60000 (80%)]   Loss: 0.139295   Accuracy:95.191%
Epoch : 3 [49600/60000 (83%)]   Loss: 0.194295   Accuracy:95.189%
Epoch : 3 [51200/60000 (85%)]   Loss: 0.083338   Accuracy:95.181%
Epoch : 3 [52800/60000 (88%)]   Loss: 0.779976   Accuracy:95.192%
Epoch : 3 [54400/60000 (91%)]   Loss: 0.033422   Accuracy:95.196%
Epoch : 3 [56000/60000 (93%)]   Loss: 0.117889   Accuracy:95.208%
Epoch : 3 [57600/60000 (96%)]   Loss: 0.146757   Accuracy:95.232%
Epoch : 3 [59200/60000 (99%)]   Loss: 0.007108   Accuracy:95.286%
Epoch : 4 [0/60000 (0%)]        Loss: 0.037586   Accuracy:96.875%
Epoch : 4 [1600/60000 (3%)]     Loss: 0.532983   Accuracy:95.282%
Epoch : 4 [3200/60000 (5%)]     Loss: 0.141879   Accuracy:95.575%
Epoch : 4 [4800/60000 (8%)]     Loss: 0.068667   Accuracy:95.882%
Epoch : 4 [6400/60000 (11%)]    Loss: 0.088112   Accuracy:95.740%
Epoch : 4 [8000/60000 (13%)]    Loss: 0.112370   Accuracy:95.754%
Epoch : 4 [9600/60000 (16%)]    Loss: 0.098440   Accuracy:95.650%
```

```
Epoch : 4 [11200/60000 (19%)]    Loss: 0.284854   Accuracy:95.593%
Epoch : 4 [12800/60000 (21%)]    Loss: 0.082128   Accuracy:95.605%
Epoch : 4 [14400/60000 (24%)]    Loss: 0.066085   Accuracy:95.468%
Epoch : 4 [16000/60000 (27%)]    Loss: 0.137398   Accuracy:95.422%
Epoch : 4 [17600/60000 (29%)]    Loss: 0.083179   Accuracy:95.389%
Epoch : 4 [19200/60000 (32%)]    Loss: 0.296649   Accuracy:95.372%
Epoch : 4 [20800/60000 (35%)]    Loss: 0.079538   Accuracy:95.387%
Epoch : 4 [22400/60000 (37%)]    Loss: 0.021219   Accuracy:95.399%
Epoch : 4 [24000/60000 (40%)]    Loss: 0.045635   Accuracy:95.460%
Epoch : 4 [25600/60000 (43%)]    Loss: 0.025185   Accuracy:95.431%
Epoch : 4 [27200/60000 (45%)]    Loss: 0.313549   Accuracy:95.384%
Epoch : 4 [28800/60000 (48%)]    Loss: 0.358900   Accuracy:95.384%
Epoch : 4 [30400/60000 (51%)]    Loss: 0.290802   Accuracy:95.354%
Epoch : 4 [32000/60000 (53%)]    Loss: 0.031017   Accuracy:95.367%
Epoch : 4 [33600/60000 (56%)]    Loss: 0.085758   Accuracy:95.362%
Epoch : 4 [35200/60000 (59%)]    Loss: 0.152539   Accuracy:95.416%
Epoch : 4 [36800/60000 (61%)]    Loss: 0.162986   Accuracy:95.441%
Epoch : 4 [38400/60000 (64%)]    Loss: 0.174990   Accuracy:95.444%
Epoch : 4 [40000/60000 (67%)]    Loss: 0.521610   Accuracy:95.456%
Epoch : 4 [41600/60000 (69%)]    Loss: 0.122210   Accuracy:95.484%
Epoch : 4 [43200/60000 (72%)]    Loss: 0.318961   Accuracy:95.503%
Epoch : 4 [44800/60000 (75%)]    Loss: 0.035795   Accuracy:95.550%
Epoch : 4 [46400/60000 (77%)]    Loss: 0.384703   Accuracy:95.535%
Epoch : 4 [48000/60000 (80%)]    Loss: 0.491851   Accuracy:95.561%
Epoch : 4 [49600/60000 (83%)]    Loss: 0.055732   Accuracy:95.561%
Epoch : 4 [51200/60000 (85%)]    Loss: 0.087599   Accuracy:95.583%
Epoch : 4 [52800/60000 (88%)]    Loss: 0.285666   Accuracy:95.597%
Epoch : 4 [54400/60000 (91%)]    Loss: 0.043655   Accuracy:95.602%
Epoch : 4 [56000/60000 (93%)]    Loss: 0.363469   Accuracy:95.604%
Epoch : 4 [57600/60000 (96%)]    Loss: 0.103343   Accuracy:95.624%
Epoch : 4 [59200/60000 (99%)]    Loss: 0.018067   Accuracy:95.676%
```

0.3 SVM Classifer for MNIST DATA SET

[0]:
```python
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[0]:
```python
x_train = x_train.reshape(60000,784)
x_test = x_test.reshape(10000,784)
```

[0]:
```python
## Applying HOG feature extraction
```

[0]:
```python
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from skimage.feature import hog
from sklearn import preprocessing
```

```
from collections import Counter
```

[38]:
```
list_hog_train = []
for feature in x_train:
    fd = hog(feature.reshape((28,28)), orientations=10,␣
↪pixels_per_cell=(7,7),cells_per_block=(1,1),visualize=False )
    list_hog_train.append(fd)
hog_features = np.array(list_hog_train, 'float64')
preProcess = preprocessing.MaxAbsScaler().fit(hog_features)
hog_features_transformed_train = preProcess.transform(hog_features)
print(hog_features_transformed_train.shape)
```

```
(60000, 160)
```

[0]:
```
## Extracting hog feature for test data
```

[41]:
```
list_hog_test = []
for feature in x_test:
    fd = hog(feature.reshape((28,28)), orientations=10,␣
↪pixels_per_cell=(7,7),cells_per_block=(1,1),visualize=False )
    list_hog_test.append(fd)
hog_features_test = np.array(list_hog_test, 'float64')
preProcess = preprocessing.MaxAbsScaler().fit(hog_features_test)
hog_features_transformed_test = preProcess.transform(hog_features_test)
print(hog_features_transformed_test.shape)
```

```
(10000, 160)
```

[0]:
```
model = SVC()
model.fit(hog_features_transformed_train,y_train)
y_pred = model.predict(hog_features_transformed_test)
```

Fitting the model with best parameters

[43]:
```
print(accuracy_score(y_test, y_pred))
```

```
0.9713
```

q4

April 3, 2020

1 Question-4 House Electricity Consumption Prediction

[0]:
```
from zipfile import ZipFile
file_name="household_power_consumption.zip"

with ZipFile(file_name,'r') as zip:
  zip.extractall()
  print('Done')

# !unzip /content/household_power_consumption.zip
```

```
Done
```

2 Reading the data file

[0]:
```
%tensorflow_version 1.x
import pandas as pd

headers = ['Date', 'Time', 'Global_active_power', 'Global_reactive_power',
           'Voltage', 'Global_intensity', 'Sub_metering_1', 'Sub_metering_2',
           'Sub_metering_3']

dtypes = {'Date':'str', 'Time':'str', 'Global_active_power':'float',
          'Global_reactive_power': 'float', 'Voltage':'float',
          'Global_intensity':'float', 'Sub_metering_1':'float',
          'Sub_metering_2':'float', 'Sub_metering_3':'float'}

df = pd.read_csv('household_power_consumption.txt', sep=';',
                 dtype=dtypes, na_values=['?'])
df.head()
```

[0]:
```
         Date      Time  …  Sub_metering_2  Sub_metering_3
0  16/12/2006  17:24:00  …             1.0            17.0
1  16/12/2006  17:25:00  …             1.0            16.0
2  16/12/2006  17:26:00  …             2.0            17.0
```

```
3  16/12/2006  17:27:00  …            1.0            17.0
4  16/12/2006  17:28:00  …            1.0            17.0

[5 rows x 9 columns]
```

[0]:
```
import numpy as np
from multiprocessing import cpu_count, Pool

def parallel_map(data, func):
    n_cores = cpu_count()
    data_split = np.array_split(data, n_cores)
    pool = Pool(n_cores)
    data = pd.concat(pool.map(func, data_split))
    pool.close()
    pool.join()
    return data

def parse(row):
    row['DateTime'] = pd.to_datetime(row['DateTime'],
                                     format='%d/%m/%Y %H:%M:%S')
    return row
```

[0]:
```
df['DateTime'] = df['Date'] + ' ' + df['Time']
df = parallel_map(df, parse)
df.dtypes
```

[0]:
```
Date                             object
Time                             object
Global_active_power             float64
Global_reactive_power           float64
Voltage                         float64
Global_intensity                float64
Sub_metering_1                  float64
Sub_metering_2                  float64
Sub_metering_3                  float64
DateTime                 datetime64[ns]
dtype: object
```

[0]:
```
df.drop(['Date', 'Time'], axis=1, inplace=True)
df = df[[df.columns[-1]] + list(df.columns[:-1])]
df.set_index('DateTime', inplace=True)
df.head()
```

[0]:
```
                     Global_active_power  …  Sub_metering_3
DateTime                                  …
2006-12-16 17:24:00                4.216  …            17.0
2006-12-16 17:25:00                5.360  …            16.0
```

```
2006-12-16 17:26:00                5.374  …             17.0
2006-12-16 17:27:00                5.388  …             17.0
2006-12-16 17:28:00                3.666  …             17.0

[5 rows x 7 columns]
```

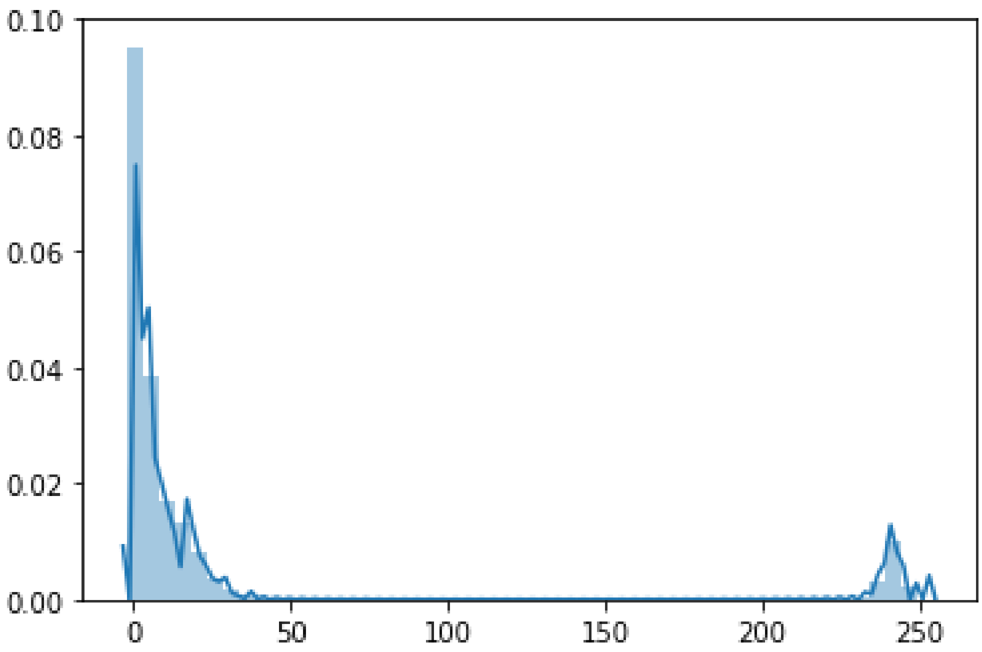
[0]:
```
df['hour'] = df.index.hour
df['day'] = df.index.day
df['month'] = df.index.month
df['day_of_week'] = df.index.dayofweek
```

[0]:
```
df['Rest_active_power'] = df['Global_active_power'] * 1000 / 60 - \
                          df['Sub_metering_1'] - df['Sub_metering_2'] - \
                          df['Sub_metering_3']
```

3 Identifying and handling Missing Data

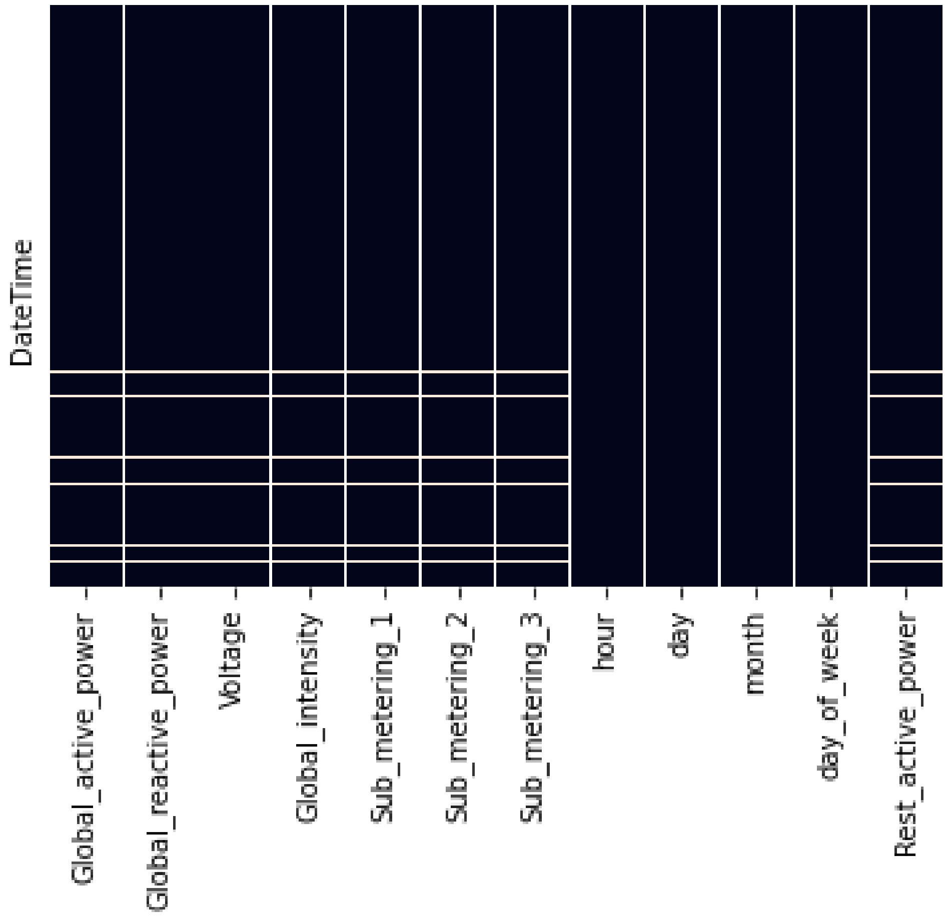
[0]:
```
import seaborn as sns
sns.distplot(df)
```

[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbcef5a8358>



[0]:
```
sns.heatmap(df.isnull(),yticklabels=False,cbar=False)
```

[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fbd67132198>



[0]:
```
number = len(df) - len(df.dropna())
percentage = number * 100 / len(df)
print(f'Number of points with missing values is: {number}')
print(f'Percentage of points with missing values is: {percentage}\n')
print(f'Missing value counts:\n{df.isnull().sum(axis=0)}\n')
```

```
Number of points with missing values is: 25979
Percentage of points with missing values is: 1.2518437457686005

Missing value counts:
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
```

```
Sub_metering_3          25979
hour                        0
day                         0
month                       0
day_of_week                 0
Rest_active_power       25979
dtype: int64
```

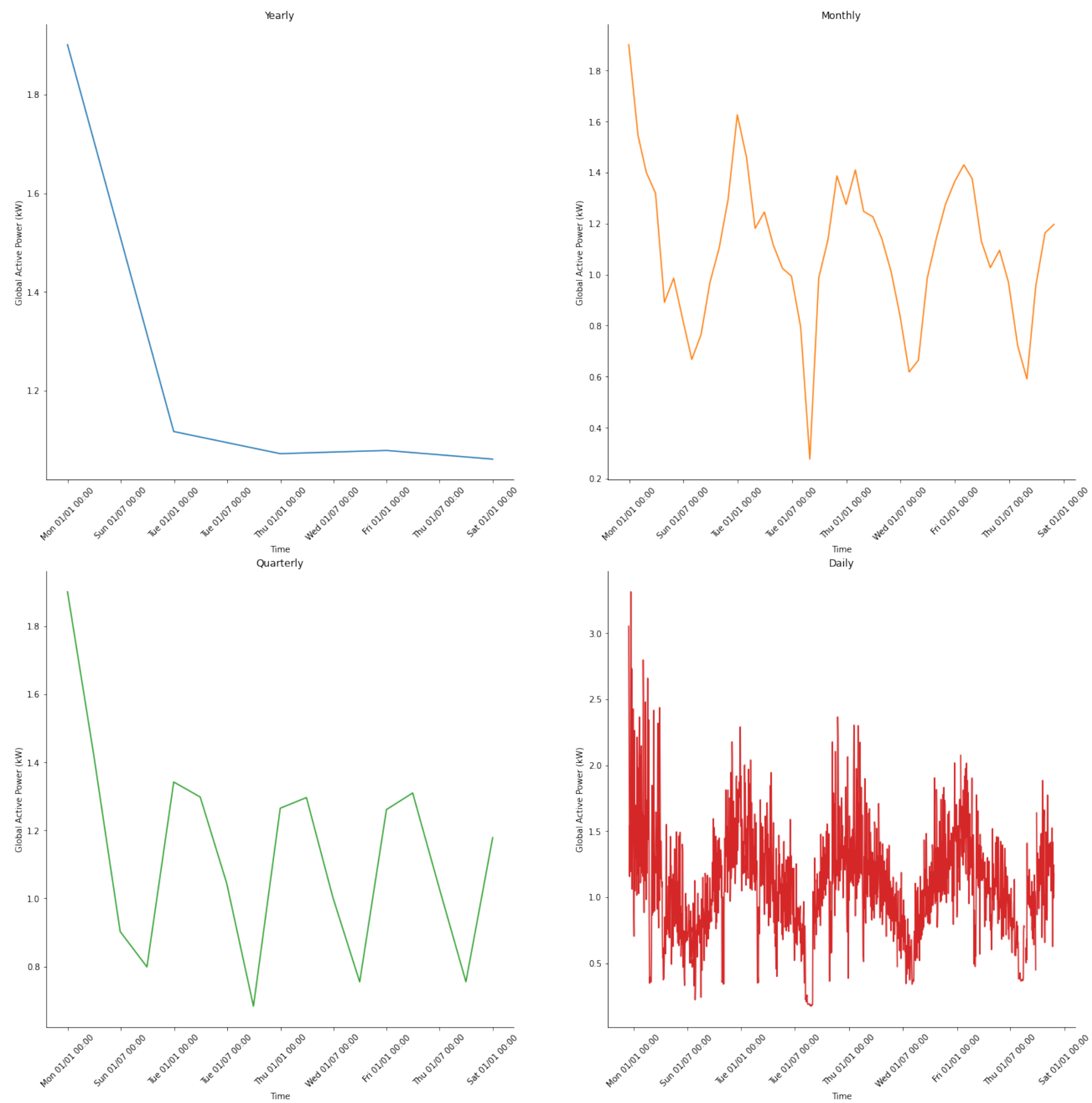
#Power consumption over the whole timespan

Average global active power over resampled data yearly, quarterly, monthly, and daily.

[0]:
```
# Tableau colors
tableau20 = [(31, 119, 180), (174, 199, 232), (255, 127, 14), (255, 187, 120),
             (44, 160, 44), (152, 223, 138), (214, 39, 40), (255, 152, 150),
             (148, 103, 189), (197, 176, 213), (140, 86, 75), (196, 156, 148),
             (227, 119, 194), (247, 182, 210), (127, 127, 127), (199, 199, 199),
             (188, 189, 34), (219, 219, 141), (23, 190, 207), (158, 218, 229)]

# Scale the RGB values to the [0, 1] range, which is the format matplotlib␣
↪accepts.
for i in range(len(tableau20)):
    r, g, b = tableau20[i]
    tableau20[i] = (r / 255., g / 255., b / 255.)
```

[0]:
```
import matplotlib.pylab as plt
import matplotlib.dates as mdates
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
my_fmt = mdates.DateFormatter('%a %d/%m %H:%M')

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(22, 22))

frequencies = ['1Y', '1M', '1Q', '1D']

dic = {'1Y':'Yearly', '1M':'Monthly', '1Q':'Quarterly', '1D':'Daily'}

i = 0
for row in ax:
    for col in row:
        for tick in col.get_xticklabels():
            tick.set_rotation(45)
        col.plot(df[['Global_active_power']].resample(frequencies[i]).mean(),
                 color=tableau20[i * 2])
        col.set_xlabel('Time')
        col.set_ylabel('Global Active Power (kW)')
        col.set_title(dic[frequencies[i]])
```

```python
        col.xaxis.set_major_formatter(my_fmt)

        # Aesthetics
        col.spines["top"].set_visible(False)
        col.spines["right"].set_visible(False)
        i += 1
```



4 supervised learning

Predict global active power at time $t+h$ given all the variables at times $t, t-1, ..., t-w$, where h is the prediction horizon and w is the window size. Here's a function that takes as input a time series dataframe, a window size, a horizon, a set of variables to be lagged and a set of variables to be forecasted and outputs the dataframe transformed and ready for supervised learing.

[0]:
```
from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, window_size=1, horizon=1, inputs='all',␣
↪targets='all'):
    """
    Frame a time series as a supervised learning dataset.

    Arguments:
        data: A pandas DataFrame containing the time series
        (the index must be a DateTimeIndex).
        window_size: Number of lagged observations as input.
        horizon: Number of steps to forecast ahead.
        inputs: A list of the columns of the dataframe to be lagged.
        targets: A list of the columns of the dataframe to be forecasted.

    Returns:
        Pandas DataFrame of series framed for supervised learning.
    """

    if targets == 'all':
        targets = data.columns

    if inputs == 'all':
        inputs = data.columns


    result = DataFrame(index=df.index)
    names = []

    # input sequence (t-w, ..., t-1)
    for i in range(window_size, 0, -1):
        result = pd.concat([result, data[inputs].shift(i)], axis=1)
        names += [(f'{data[inputs].columns[j]}(t-{i})') for j in␣
↪range(len(inputs))]

    # the input not shifted (t)
    result = pd.concat([result, data.copy()], axis=1)
    names += [(f'{column}(t)') for column in data.columns]

    # forecast (t+h)
    for i in [horizon]:
        result = pd.concat([result, data[targets].shift(-i)], axis=1)
```

```
        names += [(f'{data[targets].columns[j]}(t+{i})') for j in␣
↪range(len(targets))]

    # put it all together
    result.columns = names

    # drop rows with NaN values
    result.dropna(inplace=True)
    return result
```

[0]:
```
inputs = ['Global_active_power', 'Global_reactive_power', 'Voltage',
          'Global_intensity', 'Sub_metering_1', 'Sub_metering_2',
          'Sub_metering_3', 'Rest_active_power']

targets = ['Global_active_power']

df_supervised = series_to_supervised(df, window_size=5, horizon=1,␣
↪inputs=inputs, targets=targets)
df_supervised.head()
```

[0]:
```
                     Global_active_power(t-5)  …  Global_active_power(t+1)
DateTime                                       …
2006-12-16 17:29:00                     4.216  …                     3.702
2006-12-16 17:30:00                     5.360  …                     3.700
2006-12-16 17:31:00                     5.374  …                     3.668
2006-12-16 17:32:00                     5.388  …                     3.662
2006-12-16 17:33:00                     3.666  …                     4.448

[5 rows x 53 columns]
```

[0]:
```
#storing into some file to visualize at different window size
#df_supervised.to_csv('supervised_w10_h1.csv', index=True)
```

5 Supervised Machine Learning

[0]:
```
# import pandas as pd

# df_supervised = pd.read_csv('supervised_w10_h1.csv', parse_dates=['DateTime'])
# df_supervised.set_index('DateTime', inplace=True)
```

6 Splitting data

splitting data into train,validate and test

[0]:
```
def train_validate_test_split(df, train_percent=.6, validate_percent=.2,␣
↪seed=None):
    np.random.seed(seed)
    m = len(df)
    train_end = int(train_percent * m)
    validate_end = int(validate_percent * m) + train_end
    train = df.iloc[:train_end]
    validate = df.iloc[train_end:validate_end]
    test = df.iloc[validate_end:]
    return train, validate, test
```

[0]:
```
train,validate, test = train_validate_test_split(df_supervised)
```

[0]:
```
print(type(train))
train.shape
```

```
<class 'pandas.core.frame.DataFrame'>
```

[0]:
```
(1229311, 53)
```

[0]:
```
X_train = train.values[:, :-1]
y_train = train.values[:, -1]

X_validate = validate.values[:, :-1]
y_validate = validate.values[:, -1]

X_test = test.values[:, :-1]
y_test = test.values[:, -1]
```

[0]:
```
X_test.shape
```

[0]:
```
(409772, 52)
```

[0]:
```
X_train.shape
```

[0]:
```
(1229311, 52)
```

[0]:
```
X_validate.shape
```

[0]:
```
(409770, 52)
```

7 Mean absolute percentage Error

[0]:
```
# #from sklearn.utils import check_arrays
# def mean_absolute_percentage_error(y_true, y_pred):
#     #y_true, y_pred = check_arrays(y_true, y_pred)
#     return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

[0]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt
from sklearn.metrics import accuracy_score
```

[0]:
```
import numpy as np

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

8 Linear Regression

[0]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_validate)
```

[0]:
```
from sklearn.metrics import mean_squared_error

mean_squared_error(predictions, y_validate)
```

[0]: 0.055878991130430315

[0]:
```
mean_absolute_percentage_error(y_validate, predictions)
```

[0]: 10.996608631396205

[0]:
```
rms = sqrt(mean_squared_error(y_validate, predictions))
print(rms)
```

0.23638737515026118

[0]:
```
#accuracy_score(y_validate, predictions, normalize=False)
```

9 Multilayer Perceptron

[0]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_validate = scaler.transform(X_validate)
X_test = scaler.transform(X_test)
```

[0]:
```
from keras.layers import Input, Dense, Dropout, LSTM, Reshape, Flatten
from keras import Sequential

#from tensorflow.keras.optimizers import SGD
from keras.callbacks import EarlyStopping

model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(X_train.shape[1], )))
model.add(Dropout(0.2))
model.add(Dense(1))
```

```
Using TensorFlow backend.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:66: The name tf.get_default_graph
is deprecated. Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:541: The name tf.placeholder is
deprecated. Please use tf.compat.v1.placeholder instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:4432: The name tf.random_uniform is
deprecated. Please use tf.random.uniform instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:148: The name
tf.placeholder_with_default is deprecated. Please use
tf.compat.v1.placeholder_with_default instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:3733: calling dropout (from
tensorflow.python.ops.nn_ops) with keep_prob is deprecated and will be removed
in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 -
keep_prob`.
```

[0]:
```python
from keras.optimizers import Adam
model.compile(loss='mean_squared_error',
              optimizer=Adam(lr=0.001))

history = model.fit(X_train, y_train,batch_size=1024,epochs=100,
                    verbose=1,
                    validation_data=(X_validate, y_validate),
                    callbacks=[EarlyStopping(patience=1)])
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/optimizers.py:793: The name tf.train.Optimizer is deprecated.
Please use tf.compat.v1.train.Optimizer instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:1033: The name tf.assign_add is
deprecated. Please use tf.compat.v1.assign_add instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:1020: The name tf.assign is
deprecated. Please use tf.compat.v1.assign instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:3005: The name tf.Session is
deprecated. Please use tf.compat.v1.Session instead.

Train on 1229311 samples, validate on 409770 samples
Epoch 1/100
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:190: The name
tf.get_default_session is deprecated. Please use
tf.compat.v1.get_default_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:197: The name tf.ConfigProto is
deprecated. Please use tf.compat.v1.ConfigProto instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:207: The name tf.global_variables
is deprecated. Please use tf.compat.v1.global_variables instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:216: The name
tf.is_variable_initialized is deprecated. Please use
tf.compat.v1.is_variable_initialized instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/keras/backend/tensorflow_backend.py:223: The name
```

```
tf.variables_initializer is deprecated. Please use
tf.compat.v1.variables_initializer instead.

1229311/1229311 [==============================] - 7s 6us/step - loss: 0.1475 -
val_loss: 0.0584
Epoch 2/100
1229311/1229311 [==============================] - 7s 5us/step - loss: 0.0833 -
val_loss: 0.0542
Epoch 3/100
1229311/1229311 [==============================] - 6s 5us/step - loss: 0.0781 -
val_loss: 0.0521
Epoch 4/100
1229311/1229311 [==============================] - 7s 5us/step - loss: 0.0756 -
val_loss: 0.0515
Epoch 5/100
1229311/1229311 [==============================] - 7s 5us/step - loss: 0.0743 -
val_loss: 0.0506
Epoch 6/100
1229311/1229311 [==============================] - 7s 5us/step - loss: 0.0737 -
val_loss: 0.0506
Epoch 7/100
1229311/1229311 [==============================] - 7s 5us/step - loss: 0.0731 -
val_loss: 0.0508
```

[0]:
```
predictions = model.predict(X_validate)
```

[0]:
```
mean_squared_error(predictions, y_validate)
```

[0]:
```
0.05082758861318869
```

[0]:
```
#mean_absolute_percentage_error(y_validate, predictions)
```

[0]:
```
rms = sqrt(mean_squared_error(y_validate, predictions))
print(rms)
```

```
0.2254497474232089
```

10 Visualizing the data

Predicting for the test data

[0]:
```
predictions = model.predict(X_test)
```

[0]:
```
df_to_plot = test[['Global_active_power(t+1)']].copy()
df_to_plot['Global_active_power(t+1)_predicted'] = predictions
```

[0]:
```
import matplotlib.pylab as plt

df_to_plot[:1000].plot()
plt.show()
```