Import necessary library

```
import pandas as pd
import numpy as np
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torch import optim

from sklearn.metrics import confusion_matrix, recall_score, precision_score, f1_score, accura
from sklearn.metrics import classification_report, ConfusionMatrixDisplay
```

Dataset preparation

- Acquire the MNIST dataset
- Preprocess the data (normalizaon, reshaping, train/test split)

About Dataset ref: https://www.kaggle.com/datasets/oddrationale/mnist-in-csv

MNIST contains a collection of 70,000, 28 x 28 images of handwritten digits from 0 to 9.

The dataset consists of two files:

- train : 60,000 training examples and labels
- test: 10,000 test examples and labels

each set - label: a number from 0 to 9,image: the pixel values (a number from 0 to 255).

```
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor(),
    download = True,
)
```

```
test_data
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: ToTensor()
```

```
train_data.data[0]
```

```
tensor([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  18,
          18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,   0,   0],
```

```
        [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170, 253,
         253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253, 253,
         253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253, 253,
         198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253, 205,
          11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,  90,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253, 190,
           2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190, 253,
          70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35, 241,
         225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  81,
         240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39, 148,
         229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221, 253,
         253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253, 253,
         253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253, 195,
          80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,  11,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0]],
       dtype=torch.uint8)
```

Normalization

```
#Get mean and std of training data

print('Min Pixel Value: {} \nMax Pixel Value: {}'.format(train_data.data.min(), train_data.data
print('Mean Pixel Value {} \nPixel Values Std: {}'.format(train_data.data.float().mean(), trai
print('Scaled Mean Pixel Value {} \nScaled Pixel Values Std: {}'.format(train_data.data.float(
```

```
Min Pixel Value: 0
Max Pixel Value: 255
Mean Pixel Value 33.31842041015625
Pixel Values Std: 78.56748962402344
Scaled Mean Pixel Value 0.13066047430038452
Scaled Pixel Values Std: 0.30810779333114624
```

```
#Get mean and std of testing data

print('Min Pixel Value: {} \nMax Pixel Value: {}'.format(test_data.data.min(), test_data.data.m
print('Mean Pixel Value {} \nPixel Values Std: {}'.format(test_data.data.float().mean(), test_
print('Scaled Mean Pixel Value {} \nScaled Pixel Values Std: {}'.format(test_data.data.float()
```

```
Min Pixel Value: 0
Max Pixel Value: 255
Mean Pixel Value 33.79122543334961
Pixel Values Std: 79.17247009277344
Scaled Mean Pixel Value 0.1325146108865738
Scaled Pixel Values Std: 0.3104802668094635
```

```
transform_train=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,)) # divides by 255
        ])
transform_test=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1325,), (0.3105,)) # divides by 255
```

```
        transforms.Normalize((0.1325,), (0.3105,)) # divides by 255
        ])
```

```
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = transform_train,
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = transform_test,
    download = True,
)
```

```
1
```

```
(tensor([[[-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.3860, -0.1951,
          -0.1951, -0.1951,  1.1795,  1.3068,  1.8032, -0.0933,  1.6887,
           2.8215,  2.7197,  1.1923, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.0424,  0.0340,  0.7722,  1.5359,  1.7396,  2.7960,
           2.7960,  2.7960,  2.7960,  2.7960,  2.4396,  1.7650,  2.7960,
           2.6560,  2.0578,  0.3904, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
           0.1995,  2.6051,  2.7960,  2.7960,  2.7960,  2.7960,  2.7960,
           2.7960,  2.7960,  2.7960,  2.7706,  0.7595,  0.6195,  0.6195,
           0.2886,  0.0722, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.1951,  2.3633,  2.7960,  2.7960,  2.7960,  2.7960,  2.7960,
           2.0960,  1.8923,  2.7197,  2.6433, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242,  0.5940,  1.5614,  0.9377,  2.7960,  2.7960,  2.1851,
          -0.2842, -0.4242,  0.1231,  1.5359, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.2460, -0.4115,  1.5359,  2.7960,  0.7213,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242,  1.3450,  2.7960,  1.9942,
          -0.3988, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.2842,  1.9942,  2.7960,
           0.4668, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,  0.0213,  2.6433,
           2.4396,  1.6123,  0.9504, -0.4115, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
         [-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,  0.6068,
```

```
test_data.data[1]
```

```
tensor([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
```

```
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 116, 125, 171, 255,
         255, 150,  93,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0, 169, 253, 253, 253, 253,
         253, 253, 218,  30,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 169, 253, 253, 253, 213, 142,
         176, 253, 253, 122,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  52, 250, 253, 210,  32,  12,   0,
           6, 206, 253, 140,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  77, 251, 210,  25,   0,   0,   0,
         122, 248, 253,  65,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,  31,  18,   0,   0,   0,   0,
         209, 253, 253,  65,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 117,
         247, 253, 198,  10,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  76, 247,
         253, 231,  63,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 128, 253,
         253, 144,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 176, 246, 253,
         159,  12,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  25, 234, 253, 233,
          35,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 198, 253, 253, 141,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,  78, 248, 253, 189,  12,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,  19, 200, 253, 253, 141,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 134, 253, 253, 173,  12,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 248, 253, 253,  25,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 248, 253, 253,  43,  20,  20,
          20,  20,   5,   0,   5,  20,  20,  37, 150, 150, 150, 147,  10,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 248, 253, 253, 253, 253, 253,
         253, 253, 168, 143, 166, 253, 253, 253, 253, 253, 253, 253, 123,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0, 174, 253, 253, 253, 253, 253,
         253, 253, 253, 253, 253, 253, 249, 247, 247, 169, 117, 117,  57,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0, 118, 123, 123, 123, 166,
         253, 253, 253, 155, 123, 123,  41,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0]],
       dtype=torch.uint8)
```

Determine the appropriate hyperparameters

```
batch_size = 100
num_workers = 6
n_iters = 5000

#จำนวน epoch = จำนวน iteration/จำนวน batch (training set = 60000)
num_epochs = n_iters / (len(train_data) / batch_size)
num_epochs = int(num_epochs)
print(num_epochs)

learning_rate = 0.01
```

```
8
```

```
random_seed = 1
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)
```

```
<torch._C.Generator at 0x7fc323795a30>
```

Dataloader

```
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
```

```
    'train'  : torch.utils.data.DataLoader(train_data,
                                          batch_size=batch_size,
                                          shuffle=True),
    'test'   : torch.utils.data.DataLoader(test_data,
                                          batch_size=batch_size,
                                          shuffle=True)}
loaders
```

```
{'train': <torch.utils.data.dataloader.DataLoader at 0x7fc27e4fb2b0>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x7fc27e4fb250>}
```

Show sample data

```
examples = enumerate(loaders['test'])
batch_id, (example_data, example_targets) = next(examples)
```

```
example_data.shape
```

```
torch.Size([100, 1, 28, 28])
```

```
fig = plt.figure()

for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
  plt.title("Ground Truth: {}".format(example_targets[i]))
  plt.xticks([])
  plt.yticks([])
fig
```

CNN model design

OUTPUT FORMULA FOR CONVOLUTION

- $O = \frac{W-K+2P}{S} + 1$

OUTPUT FORMULA FOR POOLING

- $O = \frac{W-K}{S} + 1$

- O: output height/length
- W: input height/length
- K: filter size (kernel size)
- P: padding
  - $P = \frac{K-1}{2}$
- S: stride

> Ref: https://www.run.ai/guides/deep-learning-for-computer-vision/pytorch-cnn

ดับเบิลคลิก (หรือกด Enter) เพื่อแก้ไข

Def for calculate next output

```
def cal_out(input,k,s,p):
    out = ((input-k+(2*p))/s)+1
    return out
```

Model

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=100,kernel_size=6,stride=2,padding=
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=4,stride=2)
        self.out1 = nn.Linear(100 * 6 * 6, 10)
        self.relu3 = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # flatten the output of conv2
        x = x.view(x.size(0), -1)
        output = self.out1(x)
        return output
```

```
# class CNN(nn.Module):
#     def __init__(self):
#         super(CNN, self).__init__()
#         self.conv1 = nn.Conv2d(in_channels=1,out_channels=4,kernel_size=7,stride=1,padding=
#         self.relu1 = nn.ReLU()
#         self.pool1 = nn.MaxPool2d(kernel_size=3)
#         self.conv2 = nn.Conv2d(in_channels=4,out_channels=16,kernel_size=2,stride=1,padding
#         self.relu2 = nn.ReLU()
#         self.out1 = nn.Linear(16 * 9 * 9, 10)
#         self.relu3 = nn.ReLU()

#     def forward(self, x):
#         x = self.conv1(x)
#         x = self.relu1(x)
#         x = self.pool1(x)
#         x = self.conv2(x)
#         x = self.relu2(x)
#         # flatten the output of conv2
#         x = x.view(x.size(0), -1)
#         output = self.out1(x)
#         return output
```

```
model = CNN()
print(model)
```

```
CNN(
  (conv1): Conv2d(1, 100, kernel_size=(6, 6), stride=(2, 2), padding=(2, 2))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=4, stride=2, padding=0, dilation=1, ceil_mode=False)
  (out1): Linear(in_features=3600, out_features=10, bias=True)
  (relu3): ReLU()
)
```

Define loss function, opmization algorithm

```
def call_fuction(model):
    loss_func = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr = learning_rate, weight_decay=0.1)
    return loss_func,optimizer
```

```
loss_func,optimizer = call_fuction(model)
```

```
optimizer
```

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.01
    maximize: False
    weight_decay: 0.1
)
```

Train model

```
print('Number of iterations in each epoch in train set =',(len(train_data) / batch_size))
print('Number of iterations in each epoch in test set =',(len(test_data) / batch_size))
print('Number of epochs =',num_epochs)
```

```
Number of iterations in each epoch in train set = 600.0
Number of iterations in each epoch in test set = 100.0
Number of epochs = 8
```

```
def train(model,num_epochs,optimizer,loss_func):
    iter = 0
    correctImages = 0
    totalImages = 0
    trainingLoss = 0
    model.train()
    loss_each = []
    acc_each = []
    for epoch in range(num_epochs):

        for i, (images, labels) in enumerate(loaders['train']):
            images = images.requires_grad_()
            optimizer.zero_grad()
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            loss = loss_func(outputs, labels)
            loss.backward()
            optimizer.step()
            iter += 1

            correctImages += (predicted == labels).sum().item()
            totalImages += labels.size(0)
            trainingLoss += loss.item()
            accumulateAccuracy = round((correctImages/totalImages)*100,4)

            # Number of iterations in each epoch in train set
            if iter % 600 == 0:
                print('Epoch [{}/{}] Iteration: {}'.format(epoch+1, num_epochs, iter))
                print('Training accuracy: {} loss: {}'.format(accumulateAccuracy, round(loss.
                loss_each.append(round(loss.item(),4))
                acc_each.append(accumulateAccuracy)
                pass
        pass
    pass
    return loss_each,acc_each
```

```
trainloss,trainacc = train(model,num_epochs,optimizer,loss_func)
```

```
Epoch [1/8] Iteration: 600
Training accuracy: 89.1883 loss: 0.2394
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-23-17480899683a> in <cell line: 1>()
----> 1 trainloss,trainacc = train(model,num_epochs,optimizer,loss_func)

6 frames
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py in _max_pool2d(input, kernel_size, stride, padding, dilat:
    780     if stride is None:
    781         stride = torch.jit.annotate(List[int], [])
--> 782     return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
    783 
    784 

KeyboardInterrupt: 
```

Model evaluation

```
def test(cnn):
    # Test the model
    cnn.eval()
    loss_each = []
    acc_each = []
    y_trues = []
    y_preds = []
    with torch.no_grad():
        iter = 0
        correctImages = 0
        totalImages = 0

        for epoch in range(num_epochs):
            for images, labels in loaders['test']:
                outputs = cnn(images)
                _, predicted = torch.max(outputs, 1)
                loss = loss_func(outputs, labels)
                if epoch == (num_epochs-1):
                    for i in labels.tolist():
                        y_trues.append(i)
                    for i in predicted.tolist():
                        y_preds.append(i)
                correctImages += (predicted == labels).sum().item()
                totalImages += labels.size(0)
                accumulateAccuracy = round((correctImages/totalImages)*100,4)
                iter += 1

                # Number of iterations in each epoch in test set
                if iter % 100 == 0:
                    print('Epoch [{}/{}] Iteration: {}'.format(epoch+1, num_epochs, iter))
                    print('Testing accuracy: {} loss: {}'.format(accumulateAccuracy, round(lo
                    loss_each.append(round(loss.item(),4))
                    acc_each.append(accumulateAccuracy)
                    pass
            pass
        pass
    pass

    # pass
    return y_trues, y_preds,loss_each,acc_each
```

```
y_trues, y_preds, testloss, testacc = test(model)
```

Plot Loss and Accuracy

```
def plot_acc():
    epochs = range(1,num_epochs+1)
    plt.plot(epochs, trainacc, 'ro', label='Accuracy of Train set')
    plt.plot(epochs, testacc, 'bo', label='Accuracy of Test set')
    plt.title('Train Vs Test accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc=0)
    plt.figure()
    plt.show()
```

```
plot_acc()
```

```
def plot_loss():
    epochs = range(1,num_epochs+1)
    plt.plot(epochs, trainloss, 'r', label='Loss of Train set')
    plt.plot(epochs, testloss, 'b--', label='Loss of Test set')
    plt.title('Train Vs Test Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(loc=0)
```

```
    plt.legend(loc=0)
    plt.figure()
    plt.show()
```

```
plot_loss()
```

Confusion Matrix

```
list(range(0,10))
```

```
def disp_report_and_cm():
    print(classification_report(y_trues, y_preds))
    cm = confusion_matrix(y_trues, y_preds)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                  display_labels=list(range(0,10)))
    disp.plot()
    plt.show()
```

```
disp_report_and_cm()
```

Result before Optimize

```
print('Accuracy:',accuracy_score(y_trues, y_preds))
print('Precision:',precision_score(y_trues, y_preds, average='macro'))
print('Recall:',recall_score(y_trues, y_preds, average='macro'))
print('f1-Score:',f1_score(y_trues, y_preds, average='macro'))
```

```
def plot_predict(model):
    with torch.no_grad():
        output = model(example_data)
    fig = plt.figure()
    for i in range(6):
        plt.subplot(2,3,i+1)
        plt.tight_layout()
        plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
        plt.title("Prediction: {}".format(
            output.data.max(1, keepdim=True)[1][i].item()))
        plt.xticks([])
        plt.yticks([])
    return(fig)
```

```
plot_predict(model)
```

Model optimization

For repeatable experiments we have to set random seeds for anything using random number generation - this means numpy and random as well! It's also worth mentioning that cuDNN uses nondeterministic algorithms which can be disabled setting torch.backends.cudnn.enabled = False. (การสุ่มเพื่อเทรน cnn แต่ละครั้งจะ nondeterministic หรือการที่มีความคล้ายกันแค่ไม่เหมือนกัน

```
def before_repeat():
    random_seed = 1
    torch.backends.cudnn.enabled = False
    torch.manual_seed(random_seed)
```

```
before_repeat()
```

> ก่อน train model ใหม่ต้องลบรันทาม และรัน def call_function(),train(),test() ก่อนด้วย

Different CNN architectures

เพิ่ม drop out, fully-connected,ลด padding

```
# cal_out(7,5,1,2)
```

```
batch_size = 100
num_workers = 6
n_iters = 10000

#จำนวน epoch = จำนวน iteration/จำนวน batch (training set = 60000)
num_epochs = n_iters / (len(train_data) / batch_size)
num_epochs = int(num_epochs)
print(num_epochs)

learning_rate = 0.001
```

```
16
```

```
class CNN1(nn.Module):
    def __init__(self):
        super(CNN1, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=16,kernel_size=5,stride=1,padding=0
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(16, 32, 5, 1, 0)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.dropout = nn.Dropout(p=0.3)

        self.out1 = nn.Linear(32 * 4 * 4, 256)
        # self.out1 = nn.Linear(32 * 4 * 4, 10)
        self.relu3 = nn.ReLU()

        #fully connected layer, output 10 classes
        self.out2 = nn.Linear(256, 10)


    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        x = self.dropout(x)

        # # flatten the output of conv2
        x = x.view(x.size(0), -1)
        x = self.out1(x)
        x = self.relu3(x)
        x = self.dropout(x)
        output = self.out2(x)
        return output
```

```
model1 = CNN1()
print(model1)
```

```
CNN1(
  (conv1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.3, inplace=False)
  (out1): Linear(in_features=512, out_features=256, bias=True)
```

```
  (relu3): ReLU()
  (out2): Linear(in_features=256, out_features=10, bias=True)
)
```

```
1 loss_func,optimizer = call_fuction(model1)
2 optimizer
```

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0.1
)
```

```
1 trainloss,trainacc = train(model1,num_epochs,optimizer,loss_func)
```

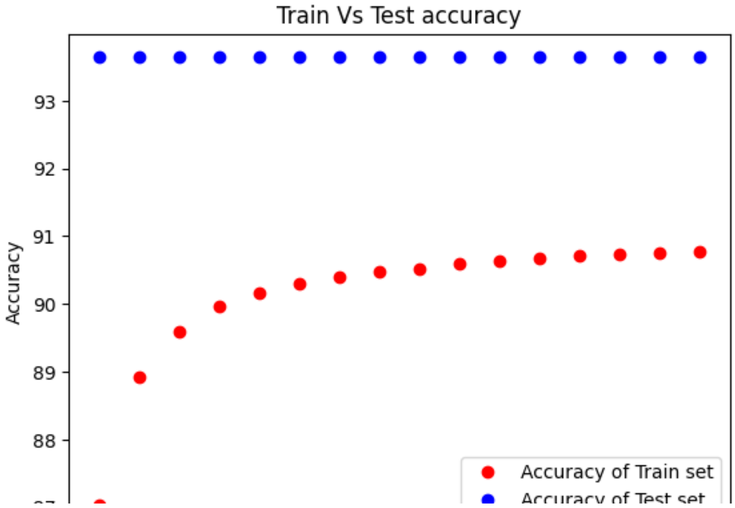
```
Epoch [1/16] Iteration: 600
Training accuracy: 87.0333 loss: 0.2743
Epoch [2/16] Iteration: 1200
Training accuracy: 88.9292 loss: 0.2899
Epoch [3/16] Iteration: 1800
Training accuracy: 89.5906 loss: 0.2048
Epoch [4/16] Iteration: 2400
Training accuracy: 89.9567 loss: 0.3229
Epoch [5/16] Iteration: 3000
Training accuracy: 90.1717 loss: 0.3549
Epoch [6/16] Iteration: 3600
Training accuracy: 90.3083 loss: 0.3259
Epoch [7/16] Iteration: 4200
Training accuracy: 90.3957 loss: 0.3827
Epoch [8/16] Iteration: 4800
Training accuracy: 90.475 loss: 0.4357
Epoch [9/16] Iteration: 5400
Training accuracy: 90.5202 loss: 0.4036
Epoch [10/16] Iteration: 6000
Training accuracy: 90.589 loss: 0.4129
Epoch [11/16] Iteration: 6600
Training accuracy: 90.6445 loss: 0.3454
Epoch [12/16] Iteration: 7200
Training accuracy: 90.6831 loss: 0.3359
Epoch [13/16] Iteration: 7800
Training accuracy: 90.7206 loss: 0.3274
Epoch [14/16] Iteration: 8400
Training accuracy: 90.7415 loss: 0.3308
Epoch [15/16] Iteration: 9000
Training accuracy: 90.7627 loss: 0.2387
Epoch [16/16] Iteration: 9600
Training accuracy: 90.7754 loss: 0.2857
```

```
1 y_trues, y_preds, testloss, testacc = test(model1)
```

```
Epoch [1/16] Iteration: 100
Testing accuracy: 93.65 loss: 0.2445
Epoch [2/16] Iteration: 200
Testing accuracy: 93.65 loss: 0.2651
Epoch [3/16] Iteration: 300
Testing accuracy: 93.65 loss: 0.172
Epoch [4/16] Iteration: 400
Testing accuracy: 93.65 loss: 0.2816
Epoch [5/16] Iteration: 500
Testing accuracy: 93.65 loss: 0.3953
Epoch [6/16] Iteration: 600
Testing accuracy: 93.65 loss: 0.2993
Epoch [7/16] Iteration: 700
Testing accuracy: 93.65 loss: 0.2466
Epoch [8/16] Iteration: 800
Testing accuracy: 93.65 loss: 0.1761
Epoch [9/16] Iteration: 900
Testing accuracy: 93.65 loss: 0.2351
Epoch [10/16] Iteration: 1000
Testing accuracy: 93.65 loss: 0.2252
Epoch [11/16] Iteration: 1100
Testing accuracy: 93.65 loss: 0.2347
Epoch [12/16] Iteration: 1200
Testing accuracy: 93.65 loss: 0.3088
Epoch [13/16] Iteration: 1300
Testing accuracy: 93.65 loss: 0.2522
Epoch [14/16] Iteration: 1400
Testing accuracy: 93.65 loss: 0.2311
```

```
Epoch [15/16] Iteration: 1500
Testing accuracy: 93.65 loss: 0.2822
Epoch [16/16] Iteration: 1600
Testing accuracy: 93.65 loss: 0.2294
```

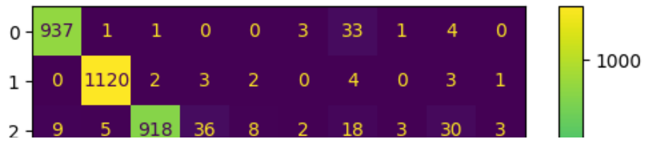
```
1 plot_acc()
```




```
1 plot_loss()
```


```
1 disp_report_and_cm()
```

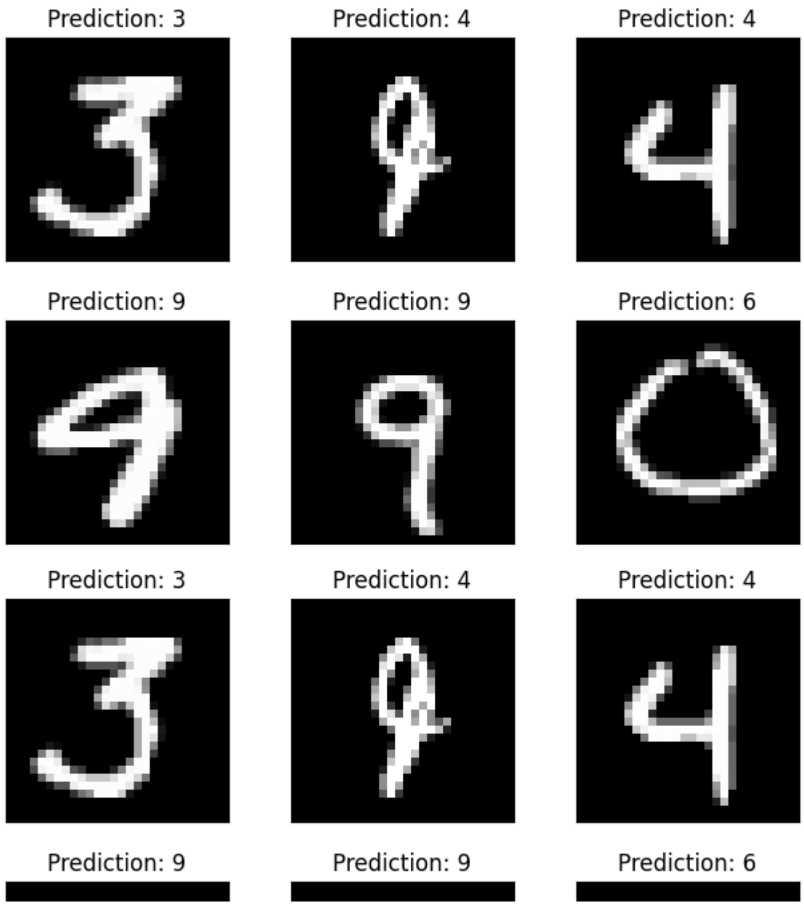
```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       980
           1       0.96      0.99      0.98      1135
           2       0.94      0.89      0.92      1032
           3       0.88      0.97      0.93      1010
           4       0.92      0.97      0.95       982
           5       0.97      0.94      0.96       892
           6       0.91      0.97      0.94       958
           7       0.98      0.87      0.92      1028
           8       0.92      0.91      0.92       974
           9       0.91      0.90      0.90      1009

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000
```




```
print('Accuracy:',accuracy_score(y_trues, y_preds))
print('Precision:',precision_score(y_trues, y_preds, average='macro'))
print('Recall:',recall_score(y_trues, y_preds, average='macro'))
print('f1-Score:',f1_score(y_trues, y_preds, average='macro'))
```

```
Accuracy: 0.9365
Precision: 0.9376093337555854
Recall: 0.9364077468081048
f1-Score: 0.9361741060721295
```

```
plot_predict(model1)
```



Data augmentation

```
transform_train=transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,)) # divides by 255
        ])
transform_test=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1325,), (0.3105,)) # divides by 255
        ])
```

```
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = transform_train,
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = transform_test,
    download = True,
)
```

```
batch_size = 100
num_workers = 6
n_iters = 10000

#จำนวน epoch = จำนวน iteration/จำนวน batch (training set = 60000)
num_epochs = n_iters / (len(train_data) / batch_size)
num_epochs = int(num_epochs)
print(num_epochs)

learning_rate = 0.001
```

```
16
```

```
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=batch_size,
                                          shuffle=True),
    'test'  : torch.utils.data.DataLoader(test_data,
                                          batch_size=batch_size,
                                          shuffle=True)}
loaders
```

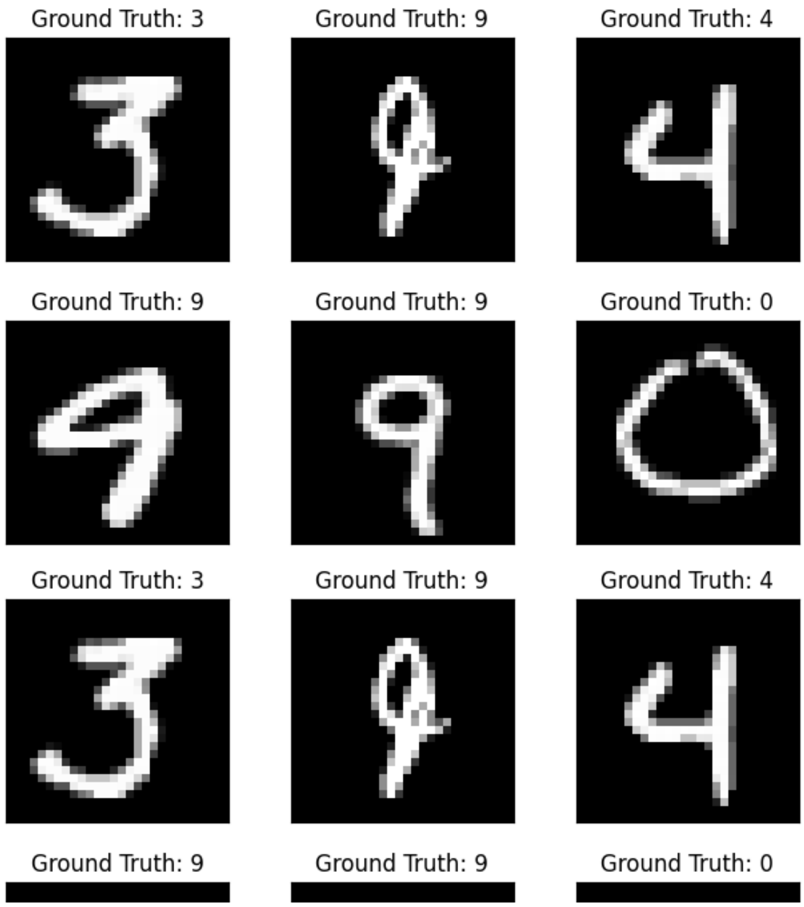
```
{'train': <torch.utils.data.dataloader.DataLoader at 0x7fdb7e23fe80>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x7fdb7e23ef20>}
```

```
examples = enumerate(loaders['test'])
batch_id, (example_data, example_targets) = next(examples)
```

```
example_data.shape
```

```
torch.Size([100, 1, 28, 28])
```

```
fig = plt.figure()

for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
  plt.title("Ground Truth: {}".format(example_targets[i]))
  plt.xticks([])
  plt.yticks([])
fig
```




```
loss_func,optimizer = call_fuction(model1)
optimizer
```

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0.1
)
```

```
trainloss,trainacc = train(model1,num_epochs,optimizer,loss_func)
```

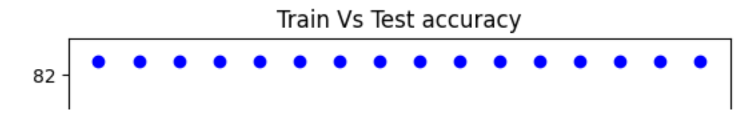
```
Epoch [1/16] Iteration: 600
Training accuracy: 68.955 loss: 0.6889
Epoch [2/16] Iteration: 1200
Training accuracy: 71.1675 loss: 0.6685
Epoch [3/16] Iteration: 1800
```

```
Training accuracy: 72.2644 loss: 0.8567
Epoch [4/16] Iteration: 2400
Training accuracy: 73.0175 loss: 0.7257
Epoch [5/16] Iteration: 3000
Training accuracy: 73.4433 loss: 0.6579
Epoch [6/16] Iteration: 3600
Training accuracy: 73.8172 loss: 0.7397
Epoch [7/16] Iteration: 4200
Training accuracy: 74.061 loss: 0.9463
Epoch [8/16] Iteration: 4800
Training accuracy: 74.2517 loss: 0.9383
Epoch [9/16] Iteration: 5400
Training accuracy: 74.3863 loss: 0.9362
Epoch [10/16] Iteration: 6000
Training accuracy: 74.542 loss: 0.6689
Epoch [11/16] Iteration: 6600
Training accuracy: 74.6724 loss: 0.7209
Epoch [12/16] Iteration: 7200
Training accuracy: 74.7761 loss: 0.7705
Epoch [13/16] Iteration: 7800
Training accuracy: 74.884 loss: 0.8183
Epoch [14/16] Iteration: 8400
Training accuracy: 74.9385 loss: 0.7529
Epoch [15/16] Iteration: 9000
Training accuracy: 75.0258 loss: 0.6349
Epoch [16/16] Iteration: 9600
Training accuracy: 75.0896 loss: 0.9992
```

```
y_trues, y_preds, testloss, testacc = test(model1)
```

```
Epoch [1/16] Iteration: 100
Testing accuracy: 82.4 loss: 0.6508
Epoch [2/16] Iteration: 200
Testing accuracy: 82.4 loss: 0.713
Epoch [3/16] Iteration: 300
Testing accuracy: 82.4 loss: 0.6943
Epoch [4/16] Iteration: 400
Testing accuracy: 82.4 loss: 0.7386
Epoch [5/16] Iteration: 500
Testing accuracy: 82.4 loss: 0.5758
Epoch [6/16] Iteration: 600
Testing accuracy: 82.4 loss: 0.6414
Epoch [7/16] Iteration: 700
Testing accuracy: 82.4 loss: 0.5768
Epoch [8/16] Iteration: 800
Testing accuracy: 82.4 loss: 0.6378
Epoch [9/16] Iteration: 900
Testing accuracy: 82.4 loss: 0.6242
Epoch [10/16] Iteration: 1000
Testing accuracy: 82.4 loss: 0.5927
Epoch [11/16] Iteration: 1100
Testing accuracy: 82.4 loss: 0.6709
Epoch [12/16] Iteration: 1200
Testing accuracy: 82.4 loss: 0.5382
Epoch [13/16] Iteration: 1300
Testing accuracy: 82.4 loss: 0.672
Epoch [14/16] Iteration: 1400
Testing accuracy: 82.4 loss: 0.7175
Epoch [15/16] Iteration: 1500
Testing accuracy: 82.4 loss: 0.6818
Epoch [16/16] Iteration: 1600
Testing accuracy: 82.4 loss: 0.6097
```

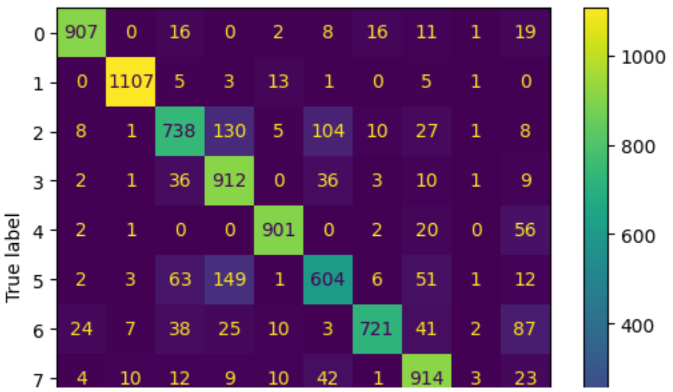
```
plot_acc()
```




```
plot_loss()
```


```
disp_report_and_cm()
```

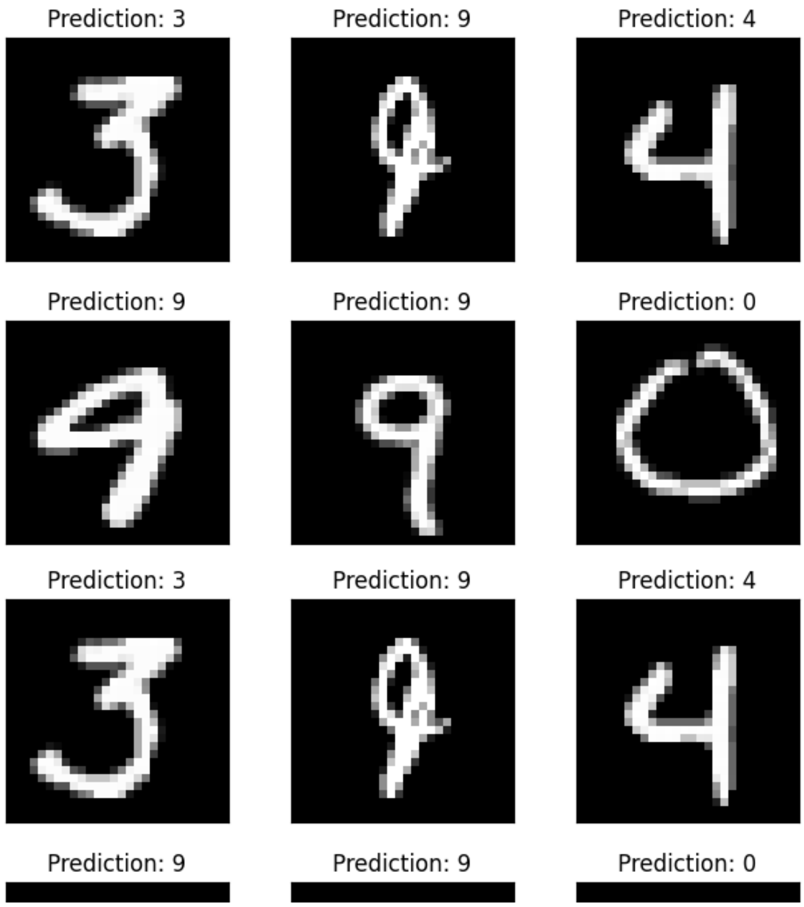
```
              precision    recall  f1-score   support

           0       0.94      0.93      0.93       980
           1       0.97      0.98      0.97      1135
           2       0.80      0.72      0.75      1032
           3       0.62      0.90      0.74      1010
           4       0.93      0.92      0.92       982
           5       0.72      0.68      0.70       892
           6       0.90      0.75      0.82       958
           7       0.80      0.89      0.84      1028
           8       0.98      0.58      0.73       974
           9       0.74      0.86      0.80      1009

    accuracy                           0.82     10000
   macro avg       0.84      0.82      0.82     10000
weighted avg       0.84      0.82      0.82     10000
```




```
print('Accuracy:',accuracy_score(y_trues, y_preds))
print('Precision:',precision_score(y_trues, y_preds, average='macro'))
print('Recall:',recall_score(y_trues, y_preds, average='macro'))
print('f1-Score:',f1_score(y_trues, y_preds, average='macro'))
```

```
Accuracy: 0.824
Precision: 0.839832031379148
Recall: 0.8198635829151002
f1-Score: 0.82055084290603
```

```
plot_predict(model1)
```



Test by testing dataset (50 images)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
transform_test=transforms.Compose([
        transforms.Grayscale(),
        transforms.ToTensor(),
        transforms.Normalize((0.1325,), (0.3105,)) # divides by 255
        ])
```

```
from torchvision.datasets import ImageFolder
test_data_new = ImageFolder('/content/drive/MyDrive/Senior_Project/50images/testing/', transf
```

```
print('New test images :', len(test_data_new))
```

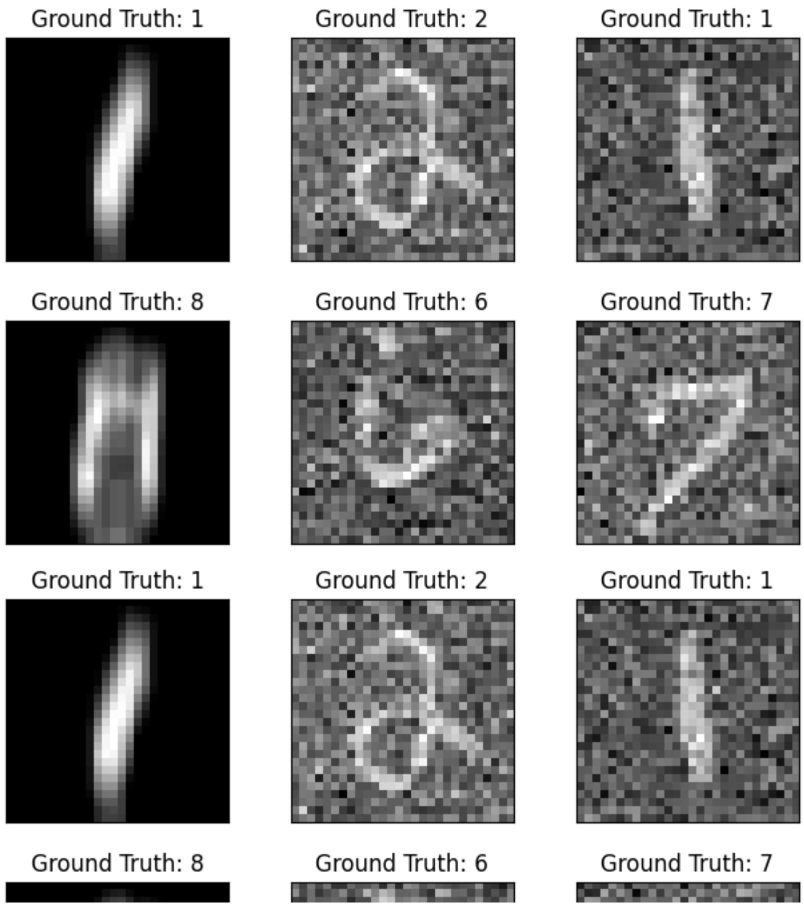
```
loaders['test_new'] = torch.utils.data.DataLoader(test_data_new,
                                          batch_size=batch_size,
                                          shuffle=True)
```

```
examples = enumerate(loaders['test_new'])
batch_id, (example_data, example_targets) = next(examples)
```

```
example_data.shape
```

```
torch.Size([100, 1, 28, 28])
```

```
fig = plt.figure()

for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
  plt.title("Ground Truth: {}".format(example_targets[i]))
  plt.xticks([])
  plt.yticks([])
fig
```




```
print('Number of iterations in each epoch in test set =',(len(test_data_new) / batch_size))
```

```
Number of iterations in each epoch in test set = 5.0
```

```
def test2(cnn):
    # Test the model
    cnn.eval()
    loss_each = []
    acc_each = []
    y_trues = []
    y_preds = []
    with torch.no_grad():
        iter = 0
        correctImages = 0
        totalImages = 0
        for epoch in range(num_epochs):
            for images, labels in loaders['test_new']:
                outputs = cnn(images)
                _, predicted = torch.max(outputs, 1)
                loss = loss_func(outputs, labels)
                if epoch == (num_epochs-1):
                    for i in labels.tolist():
                        y_trues.append(i)
                    for i in predicted.tolist():
                        y_preds.append(i)
                correctImages += (predicted == labels).sum().item()
                totalImages += labels.size(0)
                accumulateAccuracy = round((correctImages/totalImages)*100,4)
                iter += 1

                # Number of iterations in each epoch in test set
                if iter % 5 == 0:
                    print('Epoch [{}/{}] Iteration: {}'.format(epoch+1, num_epochs, iter))
                    print('Testing accuracy: {} loss: {}'.format(accumulateAccuracy, round(lo
                    loss_each.append(round(loss.item(),4))
                    acc_each.append(accumulateAccuracy)
                    pass
            pass
        pass
    pass
            # print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)

    # pass
    return y_trues, y_preds,loss_each,acc_each
```

Base Model

```
y_trues, y_preds, testloss, testacc = test2(model)
```

```
plot_acc()
```

```
plot_loss()
```

```
disp_report_and_cm()
```

```
print('Accuracy:',accuracy_score(y_trues, y_preds))
print('Precision:',precision_score(y_trues, y_preds, average='macro'))
print('Recall:',recall_score(y_trues, y_preds, average='macro'))
print('f1-Score:',f1_score(y_trues, y_preds, average='macro'))
```

```
plot_predict(model)
```

New Model

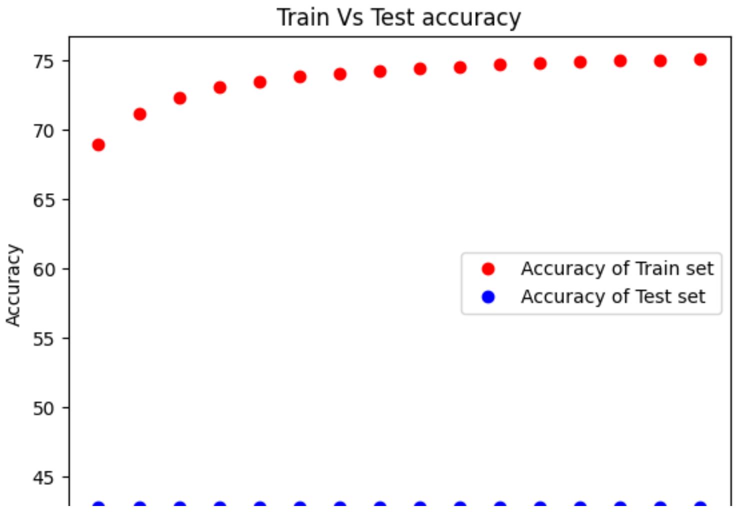
[ ] ↳ ซ่อน 6 เซลล์

New Model with augmentation

```
1 y_trues, y_preds, testloss, testacc = test2(model1)
```

```
Epoch [1/16] Iteration: 5
Testing accuracy: 42.8 loss: 1.8082
Epoch [2/16] Iteration: 10
Testing accuracy: 42.8 loss: 1.6302
Epoch [3/16] Iteration: 15
Testing accuracy: 42.8 loss: 1.7968
Epoch [4/16] Iteration: 20
Testing accuracy: 42.8 loss: 1.7691
Epoch [5/16] Iteration: 25
Testing accuracy: 42.8 loss: 1.6936
Epoch [6/16] Iteration: 30
Testing accuracy: 42.8 loss: 1.6883
Epoch [7/16] Iteration: 35
Testing accuracy: 42.8 loss: 1.6485
Epoch [8/16] Iteration: 40
Testing accuracy: 42.8 loss: 1.8405
Epoch [9/16] Iteration: 45
Testing accuracy: 42.8 loss: 1.5212
Epoch [10/16] Iteration: 50
Testing accuracy: 42.8 loss: 1.8063
Epoch [11/16] Iteration: 55
Testing accuracy: 42.8 loss: 1.7826
Epoch [12/16] Iteration: 60
Testing accuracy: 42.8 loss: 1.8067
Epoch [13/16] Iteration: 65
Testing accuracy: 42.8 loss: 1.6807
Epoch [14/16] Iteration: 70
Testing accuracy: 42.8 loss: 1.6512
Epoch [15/16] Iteration: 75
Testing accuracy: 42.8 loss: 1.757
Epoch [16/16] Iteration: 80
Testing accuracy: 42.8 loss: 1.6635
```

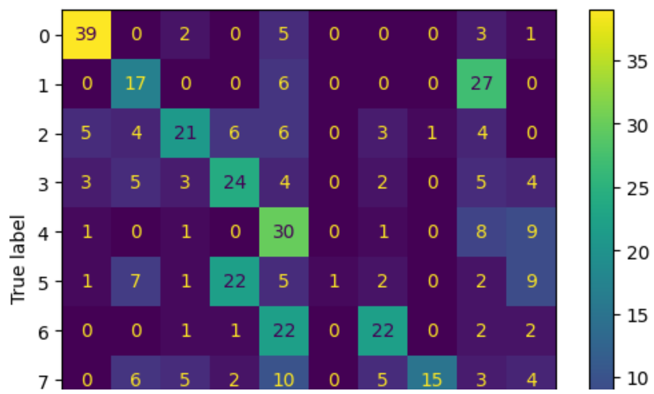
```
1 plot_acc()
```




```
1 plot_loss()
```


```
1 disp_report_and_cm()
```

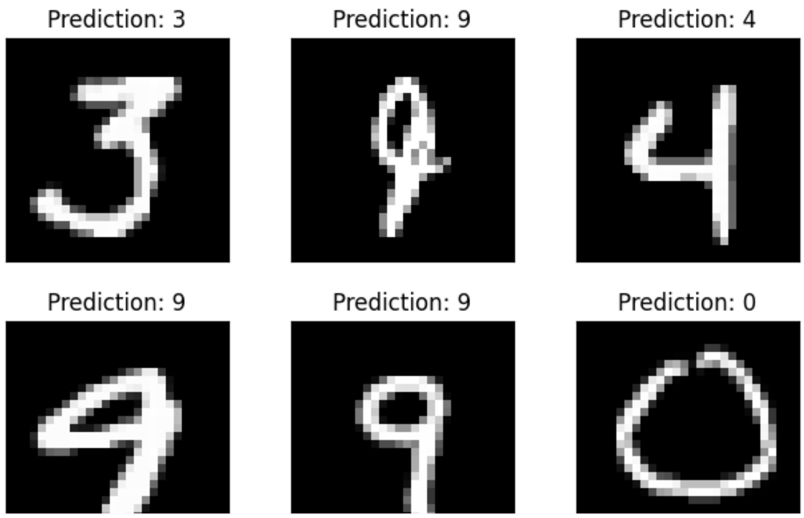
```
              precision    recall  f1-score   support

           0       0.76      0.78      0.77        50
           1       0.36      0.34      0.35        50
           2       0.62      0.42      0.50        50
           3       0.40      0.48      0.44        50
           4       0.26      0.60      0.36        50
           5       1.00      0.02      0.04        50
           6       0.61      0.44      0.51        50
           7       0.94      0.30      0.45        50
           8       0.31      0.54      0.39        50
           9       0.36      0.36      0.36        50

    accuracy                           0.43       500
   macro avg       0.56      0.43      0.42       500
weighted avg       0.56      0.43      0.42       500
```




```
1 print('Accuracy:',accuracy_score(y_trues, y_preds))
2 print('Precision:',precision_score(y_trues, y_preds, average='macro'))
3 print('Recall:',recall_score(y_trues, y_preds, average='macro'))
4 print('f1-Score:',f1_score(y_trues, y_preds, average='macro'))
```

```
Accuracy: 0.428
Precision: 0.5615894618175594
Recall: 0.42800000000000005
f1-Score: 0.4175131160752478
```

```
1 plot_predict(model1)
```


Prediction: 3
Prediction: 9
Prediction: 4
Prediction: 9
Prediction: 9
Prediction: 0