# Spandan Maaheshwari ps4_2

November 24, 2022

## 1 DS5220 Problem Set 4

## 2 Problem 2

## 3 Implement a two-layer neural network to recognize hand-written digits

```
[ ]: # Uncomment the below line and run to install required packages if you have not␣
     ↪done so
     !pip install torch torchvision matplotlib tqdm
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages
(1.12.1+cu113)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-
packages (0.13.1+cu113)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
packages (3.2.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(4.64.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch) (4.1.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchvision) (1.21.6)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from torchvision) (2.23.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.7/dist-packages (from torchvision) (7.1.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
packages (from matplotlib) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (1.4.4)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (3.0.9)
```

```
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-
packages (from python-dateutil>=2.1->matplotlib) (1.15.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchvision) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (2022.9.24)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (3.0.4)
```

[ ]:
```python
# Setup
import torch
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from tqdm import trange
import torch.nn as nn
import torch.nn.functional as F

%matplotlib inline
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# Set random seed for reproducibility
seed = 1234
# cuDNN uses nondeterministic algorithms, set some options for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.manual_seed(seed)
```

[ ]:
```
<torch._C.Generator at 0x7f5944adc0b0>
```

## 3.1 Get MNIST Data

The `torchvision` package provides a wrapper to download MNIST data. The cell below downloads the training and test datasets and creates dataloaders for each.

[ ]:
```python
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])

train_data = datasets.MNIST('data', train=True, download=True,␣
 ↪transform=transform)
```

```
test_data = datasets.MNIST('data', train=False, download=True,␣
 ↪transform=transform)

## Use the following lines to check the basic statistics of this dataset
# Calculate training data mean and standard deviation to apply normalization to␣
 ↪data
# train_data.data are of type uint8 (range 0,255) so divide by 255.
train_mean = train_data.data.double().mean() / 255.
train_std = train_data.data.double().std() / 255.
print(f'Train Data: Mean={train_mean}, Std={train_std}')

## Optional: Perform normalization of train and test data using calculated␣
 ↪training mean and standard deviation
# This will convert data to be approximately standard normal
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((train_mean, ), (train_std, ))
])

train_data.transform = transform
test_data.transform = transform

batch_size = 64
torch.manual_seed(seed)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=True)
```

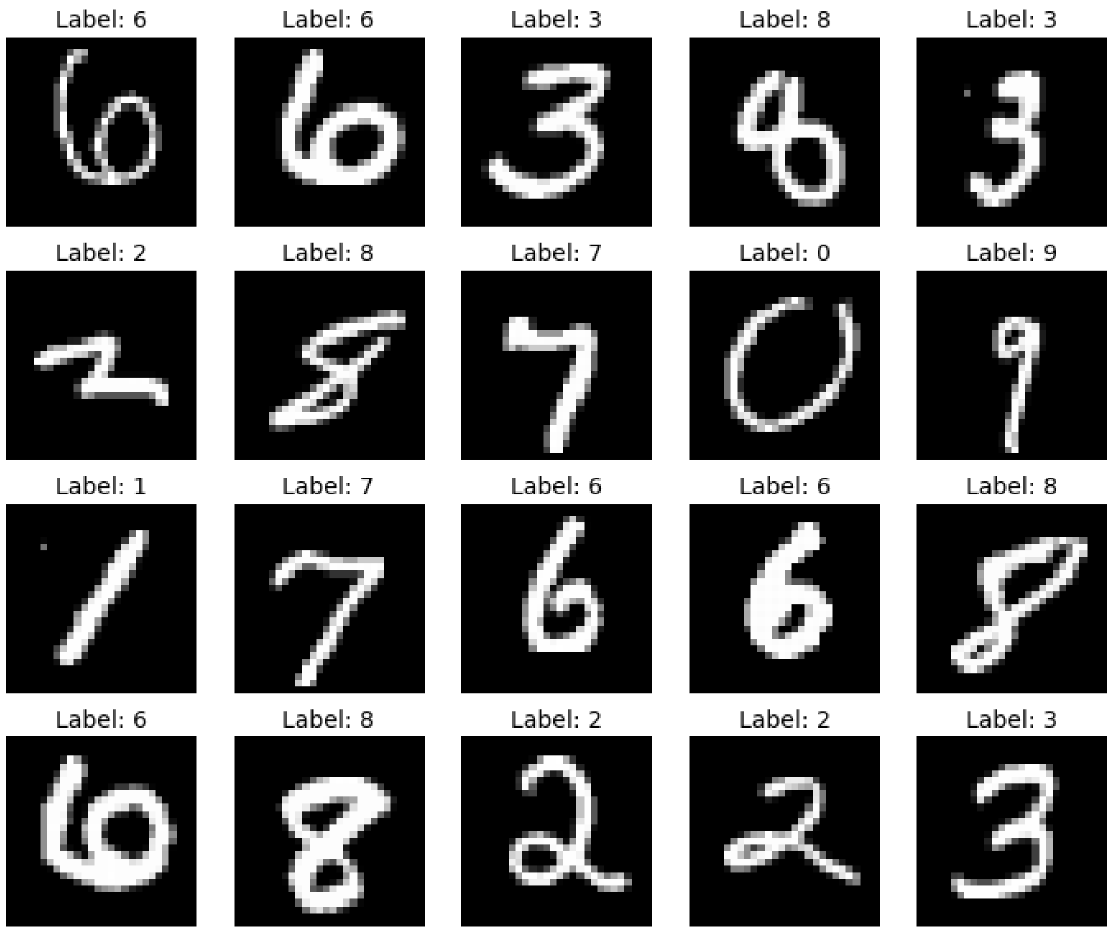
```
Train Data: Mean=0.1306604762738429, Std=0.30810780717887876
```

### 3.2 Part 0: Inspect dataset (0 points)

```
[ ]: # Randomly sample 20 images of the training dataset
     # To visualize the i-th sample, use the following code
     # > plt.subplot(4, 5, i+1)
     # > plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
     # > plt.title(f'Label: {labels[i]}', fontsize=14)
     # > plt.axis('off')

     images, labels = iter(train_loader).next()

     # Print information and statistics of the first batch of images
     print("Images shape: ", images.shape)
     print("Labels shape: ", labels.shape)
     print(f'Mean={images.mean()}, Std={images.std()}')
```

```
fig = plt.figure(figsize=(12, 10))
# ------------------
# Write your implementation here
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
  plt.title(f'Label: {labels[i]}', fontsize=14)
  plt.axis('off')

# ------------------
```

```
Images shape:  torch.Size([64, 1, 28, 28])
Labels shape:  torch.Size([64])
Mean=0.00834457017481327, Std=1.0060752630233765
```



### 3.3 Part 1: Implement a two-layer neural network

Write a class that constructs a two-layer neural network as specified in the handout. The class consists of two methods, an initialization that sets up the architecture of the model, and a forward pass function given an input feature.

[ ]:
```
input_size = 1 * 28 * 28  # input spatial dimension of images
hidden_size = 128         # width of hidden layer
output_size = 10          # number of output neurons

class MNISTClassifierMLP(torch.nn.Module):

    def __init__(self):

        super().__init__()
        self.flatten = torch.nn.Flatten(start_dim=1)
        # ------------------
        # Write your implementation here.

        self.fc1 = nn.Linear(28 * 28, 128)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)
        self.log_softmax = nn.Softmax(dim = 1)

        # ------------------

    def forward(self, x):
        # Input image is of shape [batch_size, 1, 28, 28]
        # Need to flatten to [batch_size, 784] before feeding to fc1
        x = self.flatten(x)

        # ------------------
        # Write your implementation here.

        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        y_output = F.log_softmax(x)

        return y_output
        # ------------------

model = MNISTClassifierMLP().to(DEVICE)

# sanity check
print(model)
```

```
MNISTClassifierMLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (act): ReLU()
  (fc2): Linear(in_features=128, out_features=10, bias=True)
  (log_softmax): Softmax(dim=1)
)
```

### 3.4 Part 2: Implement an optimizer to train the neural net model

Write a method called `train_one_epoch` that runs one step using the optimizer.

```
[ ]: def train_one_epoch(train_loader, model, device, optimizer, log_interval,␣
     ↪epoch):
         model.train()
         losses = []
         counter = []
         num_correct = 0

         for i, (img, label) in enumerate(train_loader):
             img, label = img.to(device), label.to(device)

             # ------------------
             # Write your implementation here.

             optimizer.zero_grad()

             y_pred = model(img)
             _,pred_train = y_pred.max(1)
             num_correct += pred_train.eq(label).sum().item()
             loss = F.nll_loss(y_pred, label)
             loss.backward()
             optimizer.step()

             # ------------------

             # Record training loss every log_interval and keep counter of total␣
     ↪training images seen
             if (i+1) % log_interval == 0:
                 losses.append(loss.item())
                 counter.append(
                     (i * batch_size) + img.size(0) + epoch * len(train_loader.
     ↪dataset))

         return losses, counter, num_correct
```

### 3.5 Part 3: Run the optimization procedure and test the trained model

Write a method called `test_one_epoch` that evalutes the trained model on the test dataset. Return the average test loss and the number of samples that the model predicts correctly.

```
[ ]: def test_one_epoch(test_loader, model, device):
         model.eval()
         test_loss = 0
         num_correct = 0

         with torch.no_grad():
             for i, (img, label) in enumerate(test_loader):
                 img, label = img.to(device), label.to(device)

                 # -------------------
                 # Write your implementation here.

                 output = model(img)
                 test_loss += F.nll_loss(output, label, size_average=False).item()
                 pred = output.data.max(1, keepdim=True)[1]
                 num_correct += pred.eq(label.data.view_as(pred)).sum().item()

                 # -------------------

         test_loss /= len(test_loader.dataset)
         return test_loss, num_correct
```

Train the model using the cell below. Hyperparameters are given.

```
[ ]: # Hyperparameters
     lr = 0.01
     max_epochs=10
     gamma = 0.95

     # Recording data
     log_interval = 100

     # Instantiate optimizer (model was created in previous cell)
     optimizer = torch.optim.SGD(model.parameters(), lr=lr)

     train_losses = []
     train_counter = []
     test_losses = []
     test_correct = []
     train_correct = []
     for epoch in trange(max_epochs, leave=True, desc='Epochs'):
         train_loss, counter, num_correct_train = train_one_epoch(train_loader,␣
      ↪model, DEVICE, optimizer, log_interval, epoch)
```

```
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

    # Record results
    train_losses.extend(train_loss)
    train_counter.extend(counter)
    test_losses.append(test_loss)
    train_correct.append(num_correct_train)
    test_correct.append(num_correct)

print(f"Accuracy on the training dataset: {train_correct[-1]/len(train_loader.
 ↪dataset)}")
print(f"Acurracy on the test dataset: {test_correct[-1]/len(test_loader.
 ↪dataset)}")
print(f"Train loss: {train_losses[-1]}")
print(f"Test loss: {test_losses[-1]}")
```

```
Epochs:   0%|          | 0/10 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-
packages/ipykernel_launcher.py:32: UserWarning: Implicit dimension choice for
log_softmax has been deprecated. Change the call to include dim=X as an
argument.
Epochs: 100%|     | 10/10 [03:00<00:00, 18.00s/it]

Accuracy on the training dataset: 0.9640666666666666
Acurracy on the test dataset: 0.9635
Train loss: 0.15572525560855865
Test loss: 0.12718422798663379
```

(c)

Training loss: 0.15572

Test loss: 0.12718

Accuracy on the training dataset: 0.96406

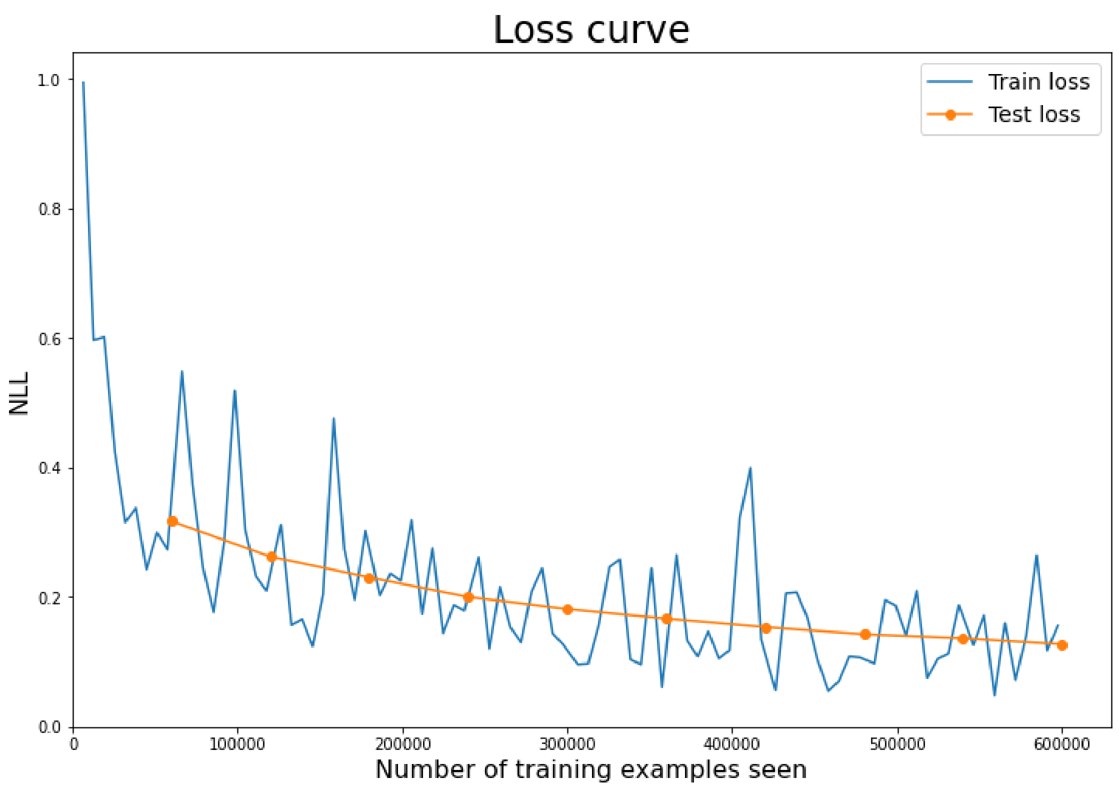
Acurracy on the test dataset: 0.9635

## 3.6 Part 4: Inspection

1. Plot the loss curve as the number of epochs increases.
2. Show the predictions of the first 20 images of the test set.
3. Show the first 20 images that the model predicted incorrectly. Discuss about some of the common scenarios that the model predicted incorrectly.
4. Go back to Part 0, where we created the tranform component to apply on the training and test datasets. Re-run the code by uncommenting the normalization step, so that the training and test dataset have mean zero and unit variance. Report the result after this normalization step again.

```
[ ]: # 1. Draw training loss curve
fig = plt.figure(figsize=(12,8))
plt.plot(train_counter, train_losses, label='Train loss')
plt.plot([i * len(train_loader.dataset) for i in range(1, max_epochs + 1)],
         test_losses, label='Test loss', marker='o')
plt.xlim(left=0)
plt.ylim(bottom=0)
plt.title('Loss curve', fontsize=24)
plt.xlabel('Number of training examples seen', fontsize=16)
plt.ylabel('NLL', fontsize=16)
plt.legend(loc='upper right', fontsize=14)
```

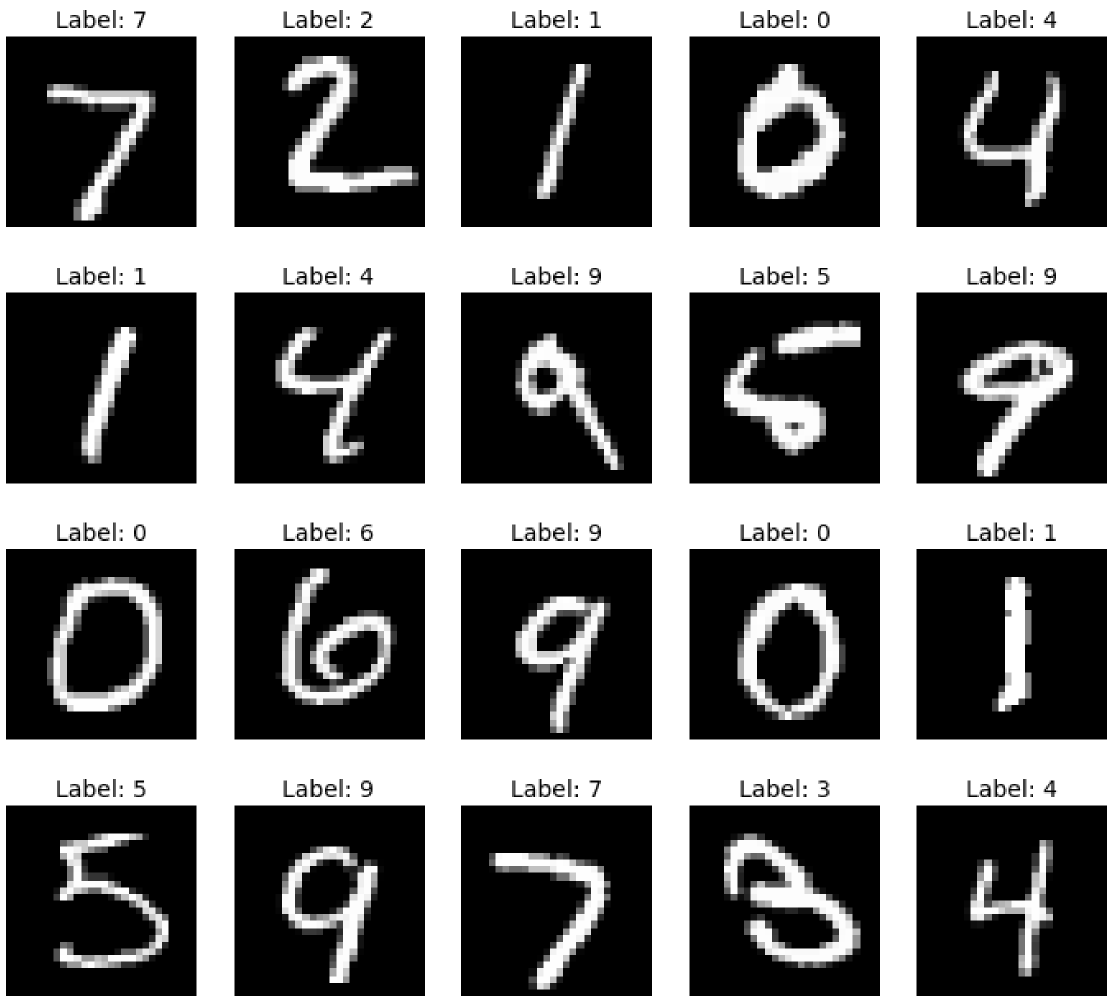
```
[ ]: <matplotlib.legend.Legend at 0x7f593eee7b50>
```



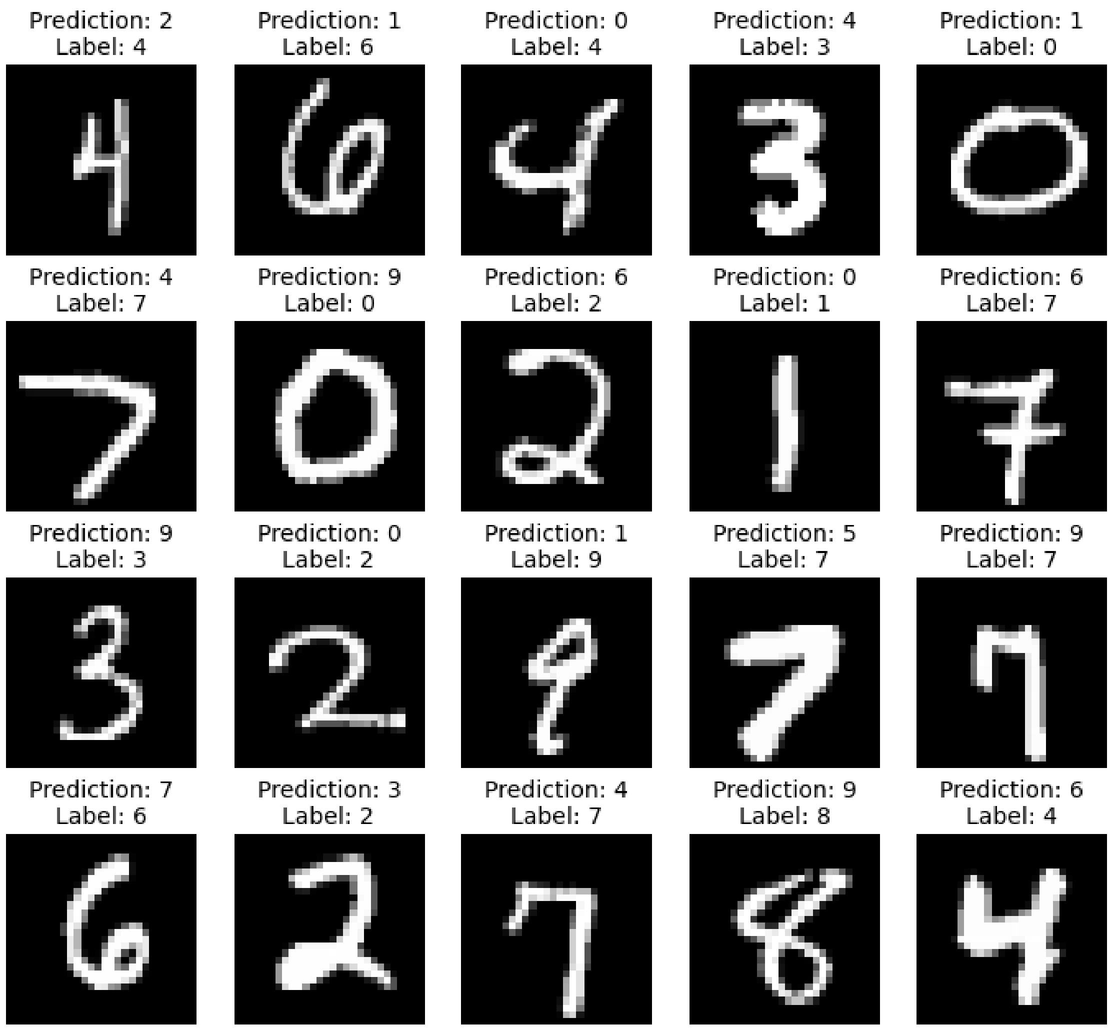
```
[ ]: # 2. Show the predictions of the first 20 images of the test dataset
images, labels = iter(test_loader).next()
images, labels = images.to(DEVICE), labels.to(DEVICE)

output = model(images)
pred = output.argmax(dim=1)

fig = plt.figure(figsize=(12, 11))
```

```
# -------------------
# Write your implementation here. Use the code provided in Part 0 to visualize
 ↪the images.
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
  plt.title(f'Label: {labels[i]}', fontsize=14)
  plt.axis('off')

# -------------------
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:32: UserWarning:
Implicit dimension choice for log_softmax has been deprecated. Change the call
to include dim=X as an argument.
```



```
[ ]: # 3. Get 20 incorrect predictions in test dataset

     # Collect the images, predictions, labels for the first 20 incorrect predictions
     # Initialize empty tensors and then keep appending to the tensor.
     # Make sure that the first dimension of the tensors is the total number of␣
      ↪incorrect
     # predictions seen so far
     # Ex) incorrect_imgs should be of shape i x C x H x W, where i is the total␣
      ↪number of
     # incorrect images so far.
     incorrect_imgs = torch.Tensor().to(DEVICE)
     incorrect_preds = torch.IntTensor().to(DEVICE)
     incorrect_labels = torch.IntTensor().to(DEVICE)

     with torch.no_grad():
         # Test set iterator
         it = iter(test_loader)
         # Loop over the test set batches until incorrect_imgs.size(0) >= 20
         while incorrect_imgs.size(0) < 20:
             images, labels = it.next()
             images, labels = images.to(DEVICE), labels.to(DEVICE)

             # ------------------
             # Write your implementation here.

             pred = output.argmax(dim=1, keepdim=True)
             incorrect = (pred != labels.view_as(pred)).nonzero()[:, 0]

             # Compare prediction and true labels and append the incorrect␣
      ↪predictions
             # using `torch.cat`.

             incorrect_imgs = images[incorrect]
             incorrect_preds =  pred[incorrect]
             incorrect_labels =  labels[incorrect]

             # ------------------

     # Show the first 20 wrong predictions in test set
     fig = plt.figure(figsize=(12, 11))
     for i in range(20):
         plt.subplot(4, 5, i+1)
         plt.imshow(incorrect_imgs[i].squeeze().cpu().numpy(), cmap='gray',␣
      ↪interpolation='none')
         plt.title(f'Prediction: {incorrect_preds[i].item()}\nLabel:␣
      ↪{incorrect_labels[i].item()}', fontsize=14)
         plt.axis('off')
```



[ ]:
```
from torchsummary import summary
summary(model, (1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1                  [-1, 784]               0
            Linear-2                  [-1, 128]         100,480
              ReLU-3                  [-1, 128]               0
            Linear-4                   [-1, 10]           1,290
================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
```

```
Forward/backward pass size (MB): 0.01
Params size (MB): 0.39
Estimated Total Size (MB): 0.40
----------------------------------------------------------------

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:32: UserWarning:
Implicit dimension choice for log_softmax has been deprecated. Change the call
to include dim=X as an argument.
```

# Spandan Maaheshwari ps4_3

November 24, 2022

## 1 DS5220 Problem Set 4

## 2 Problem 3

## 3 Implement a convolutional neural network to recognize handwritten digits

```
[ ]: # Uncomment the below line and run to install required packages if you have not␣
      ↪done so
     !pip install torch torchvision matplotlib tqdm
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages
(1.12.1+cu113)
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-
packages (0.13.1+cu113)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-
packages (3.2.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(4.64.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch) (4.1.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.7/dist-packages (from torchvision) (7.1.2)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from torchvision) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchvision) (1.21.6)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (3.0.9)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib) (1.4.4)
```

```
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
packages (from matplotlib) (0.11.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-
packages (from python-dateutil>=2.1->matplotlib) (1.15.0)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (2022.9.24)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchvision) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchvision) (2.10)
```

[ ]:
```python
# Setup
import torch
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from tqdm import trange

%matplotlib inline
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# Set random seed for reproducibility
seed = 1234
# cuDNN uses nondeterministic algorithms, set some options for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.manual_seed(seed)
```

[ ]:
```
<torch._C.Generator at 0x7f246c68b050>
```

### 3.1 Get MNIST Data

The `torchvision` package provides a wrapper to download MNIST data. The cell below downloads the training and test datasets and creates dataloaders for each.

[ ]:
```python
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

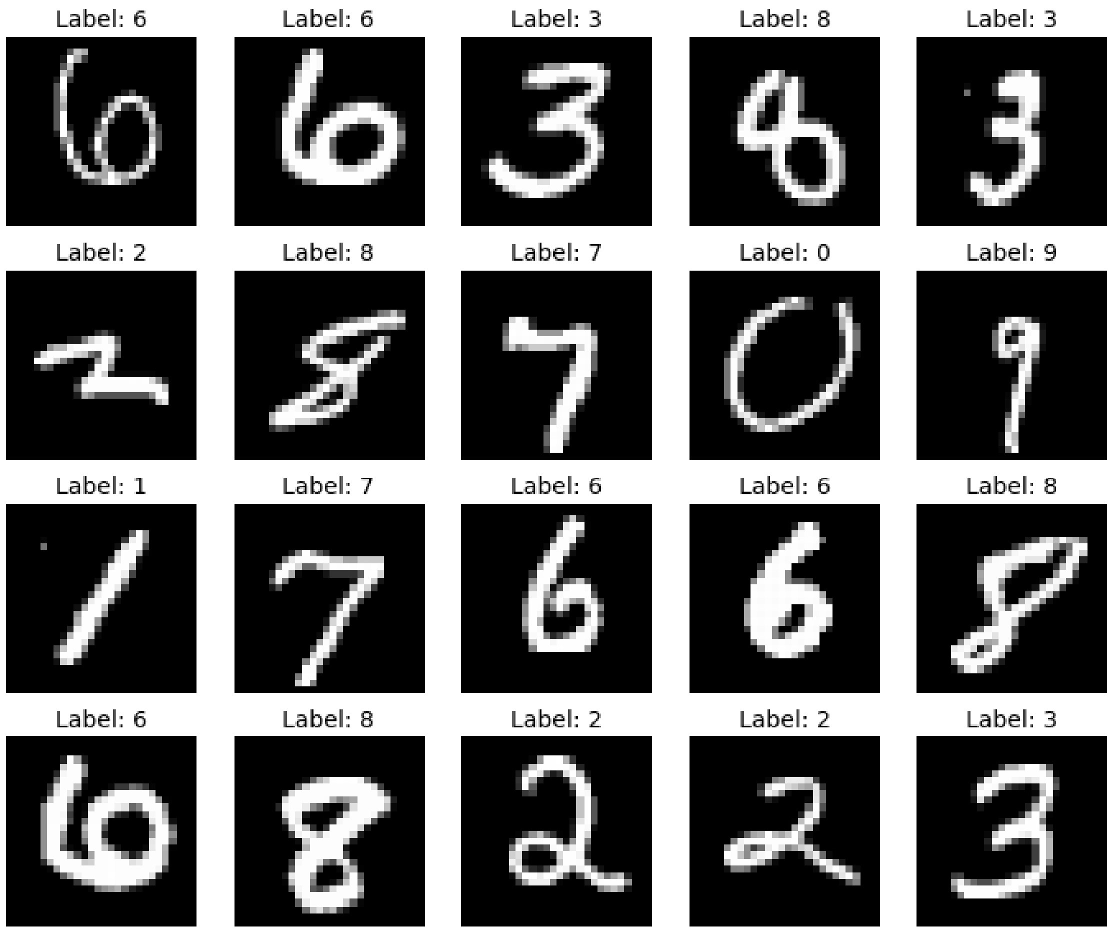
```
## Use the following lines to check the basic statistics of this dataset
# Calculate training data mean and standard deviation to apply normalization to␣
 ↪data
# train_data.data are of type uint8 (range 0,255) so divide by 255.
train_mean = train_data.data.double().mean() / 255.
train_std = train_data.data.double().std() / 255.
print(f'Train Data: Mean={train_mean}, Std={train_std}')

## Optional: Perform normalization of train and test data using calculated␣
 ↪training mean and standard deviation
# This will convert data to be approximately standard normal
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((train_mean, ), (train_std, ))
])

train_data.transform = transform
test_data.transform = transform

batch_size = 64
torch.manual_seed(seed)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=True)
```

```
Train Data: Mean=0.1306604762738429, Std=0.30810780717887876
```

## 3.2 Part 0: Inspect dataset (0 points)

[ ]:
```
# Randomly sample 20 images of the training dataset
# To visualize the i-th sample, use the following code
# > plt.subplot(4, 5, i+1)
# > plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
# > plt.title(f'Label: {labels[i]}', fontsize=14)
# > plt.axis('off')

images, labels = iter(train_loader).next()

# Print information and statistics of the first batch of images
print("Images shape: ", images.shape)
print("Labels shape: ", labels.shape)
print(f'Mean={images.mean()}, Std={images.std()}')

fig = plt.figure(figsize=(12, 10))
# ------------------
# Copy the implementation from Problem 4 here
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
  plt.title(f'Label: {labels[i]}', fontsize=14)
  plt.axis('off')

# ------------------
```

```
Images shape:  torch.Size([64, 1, 28, 28])
Labels shape:  torch.Size([64])
Mean=0.00834457017481327, Std=1.0060752630233765
```



### 3.3 Implement a convolutional neural network (10 points)

Write a class that constructs a two-layer neural network as specified in the handout. The class consists of two methods, an initialization that sets up the architecture of the model, and a forward pass function given an input feature.

```
[ ]: input_size = 1 * 28 * 28  # input spatial dimension of images
     hidden_size = 128          # width of hidden layer
     output_size = 10           # number of output neurons

     import torch.nn as nn
     import torch.nn.functional as F

     class CNN(torch.nn.Module):
       def __init__(self):

           super(CNN, self).__init__()

           self.conv1 = nn.Conv2d(1, 10, kernel_size=5,stride=1,padding=0)

           self.conv2 = nn.Conv2d(10, 20, kernel_size=5,stride=1,padding=0)

           self.conv2_drop = nn.Dropout2d()
           self.fc1 = nn.Linear(320, 128)
           self.fc2 = nn.Linear(128, 10)

       def forward(self, x):
           x = F.relu(F.max_pool2d(self.conv1(x), 2))
           x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
           x = x.view(-1, 320)
           x = F.relu(self.fc1(x))
           x = F.dropout(x, training=self.training)
           x = self.fc2(x)
           return F.log_softmax(x, dim=1)

     model = CNN().to(DEVICE)

     # sanity check
     print(model)
     from torchsummary import summary
     summary(model, (1,28,28))
```

```
CNN(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 10, 24, 24]             260
            Conv2d-2             [-1, 20, 8, 8]           5,020
         Dropout2d-3             [-1, 20, 8, 8]               0
            Linear-4                  [-1, 128]          41,088
            Linear-5                   [-1, 10]           1,290
================================================================
Total params: 47,658
Trainable params: 47,658
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.18
Estimated Total Size (MB): 0.25
----------------------------------------------------------------
```

Write a method called `train_one_epoch` that runs one step using the optimizer.

[ ]:
```
def train_one_epoch(train_loader, model, device, optimizer, log_interval,␣
 ↪epoch):
    model.train()
    losses = []
    counter = []
    num_correct = 0

    for i, (img, label) in enumerate(train_loader):
        img, label = img.to(device), label.to(device)

        # ------------------
        # Copy the implementation from Problem 2 here

        optimizer.zero_grad()

        y_pred = model(img)
        _,pred_train = y_pred.max(1)
        num_correct += pred_train.eq(label).sum().item()
        loss = F.nll_loss(y_pred, label)
        loss.backward()
        optimizer.step()

        # ------------------

        # Record training loss every log_interval and keep counter of total␣
 ↪training images seen
        if (i+1) % log_interval == 0:
            losses.append(loss.item())
            counter.append(
                 (i * batch_size) + img.size(0) + epoch * len(train_loader.
↪dataset))

    return losses, counter, num_correct
```

Write a method called `test_one_epoch` that evalutes the trained model on the test dataset. Return the average test loss and the number of samples that the model predicts correctly.

[ ]:
```
def test_one_epoch(test_loader, model, device):
    model.eval()
    test_loss = 0
    num_correct = 0

    with torch.no_grad():
        for i, (img, label) in enumerate(test_loader):
            img, label = img.to(device), label.to(device)

            # ------------------
            # Copy the implementation from Problem 2 here

            output = model(img)
            test_loss += F.nll_loss(output, label, reduction='sum').item()
            pred = output.data.max(1, keepdim=True)[1]
            num_correct += pred.eq(label.data.view_as(pred)).sum().item()

            # ------------------

    test_loss /= len(test_loader.dataset)
    return test_loss, num_correct
```

Train the model using the cell below. Hyperparameters are given.

[ ]:
```
# Hyperparameters
lr = 0.01
max_epochs=10
gamma = 0.95

# Recording data
log_interval = 100

# Instantiate optimizer (model was created in previous cell)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
train_counter = []
test_losses = []
test_correct = []
```

```
train_correct = []
for epoch in trange(max_epochs, leave=True, desc='Epochs'):
    train_loss, counter, num_correct_train = train_one_epoch(train_loader,␣
 ↪model, DEVICE, optimizer, log_interval, epoch)
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

    # Record results
    train_losses.extend(train_loss)
    train_counter.extend(counter)
    test_losses.append(test_loss)
    train_correct.append(num_correct_train)
    test_correct.append(num_correct)

print(f"Accuracy on the training dataset: {train_correct[-1]/len(train_loader.
 ↪dataset)}")
print(f"Acurracy on the test dataset: {test_correct[-1]/len(test_loader.
 ↪dataset)}")
print(f"Train loss: {train_losses[-1]}")
print(f"Test loss: {test_losses[-1]}")
```

```
Epochs: 100%|     | 10/10 [05:15<00:00, 31.58s/it]

Accuracy on the training dataset: 0.9541666666666667
Acurracy on the test dataset: 0.9815
Train loss: 0.18401074409484863
Test loss: 0.061273519840463996
```

(b)

Training loss: 0.1840

Test loss: 0.06127

Accuracy on the training dataset: 0.95416

Acurracy on the test dataset: 0.9815

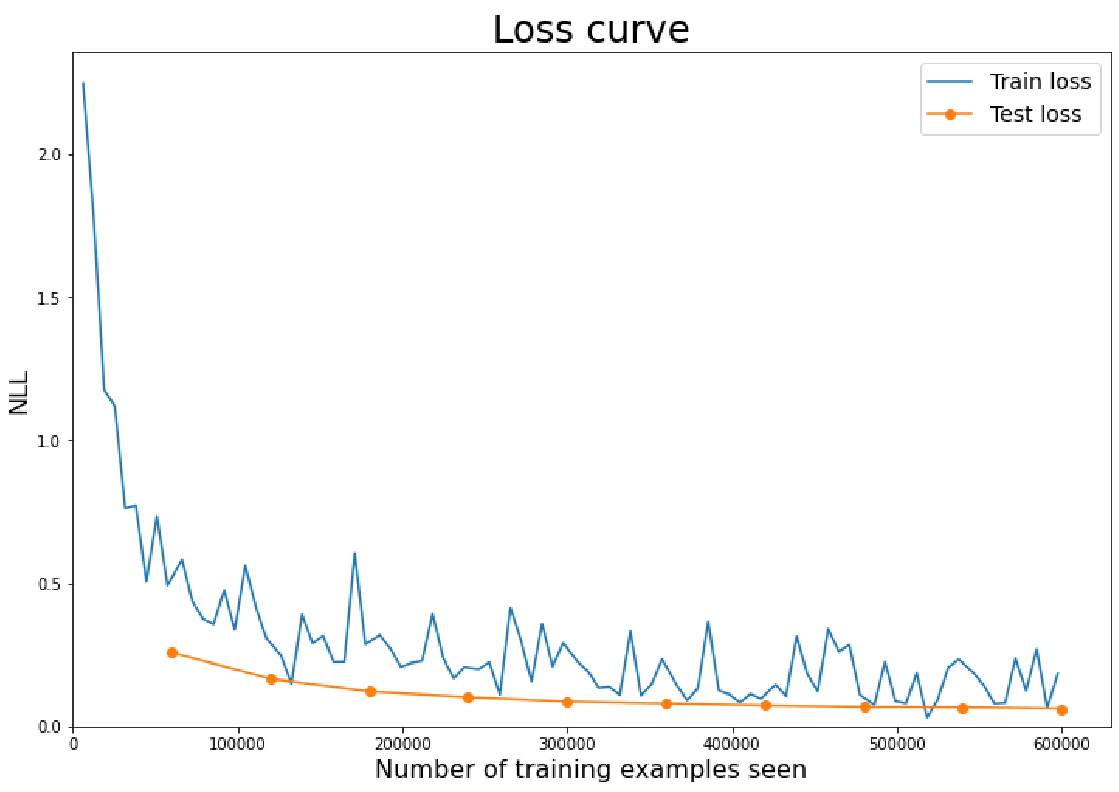
Convolutional Neural Networks outperforms the Feed forward Neural Networks in terms of accuracy obtained on the test dataset since:

Test accuracy of CNN model: 98.5%

Test accuracy of Feed forward NN: 96.35%

```
[ ]: # 1. Drawing training & testing loss curve
fig = plt.figure(figsize=(12,8))
plt.plot(train_counter, train_losses, label='Train loss')
plt.plot([i * len(train_loader.dataset) for i in range(1, max_epochs + 1)],
         test_losses, label='Test loss', marker='o')
plt.xlim(left=0)
plt.ylim(bottom=0)
plt.title('Loss curve', fontsize=24)
plt.xlabel('Number of training examples seen', fontsize=16)
plt.ylabel('NLL', fontsize=16)
plt.legend(loc='upper right', fontsize=14)
```

```
[ ]: <matplotlib.legend.Legend at 0x7f2465d34e90>
```



```
[ ]: from torchsummary import summary
summary(model, (1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 10, 24, 24]             260
            Conv2d-2             [-1, 20, 8, 8]           5,020
         Dropout2d-3             [-1, 20, 8, 8]               0
            Linear-4                  [-1, 128]          41,088
            Linear-5                   [-1, 10]           1,290
================================================================
Total params: 47,658
Trainable params: 47,658
```

```
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.18
Estimated Total Size (MB): 0.25
----------------------------------------------------------------
```

(c)

Number of parameters used in feed forward neural networks from problem 2 manually and through model summary: 101,770

Number of parameters used in CNN from problem 3 manually and through model summary: 47,658

CNNs are more parameter-efficient than feed-forward neural networks because they share parameter properties such as dimensionality and feature sharing. As a result, the number of parameters in a CNN is reduced, and the computations are also reduced.