DS5220 Homework 4

Aditya Singh

TOTAL POINTS

100 / 100

QUESTION 1

10 pts

1.1 2 / 2

✓ **- 0 pts Correct**

- 1 pts partially correct

1.2 3 / 3

✓ **- 0 pts Correct**

- 1 pts did not mention internal co-varient shift

1.3 3 / 3

✓ **- 0 pts Correct**

1.4 2 / 2

✓ **- 0 pts Correct**

- 1 pts vague explaination

QUESTION 2

50 pts

2.1 15 / 15

✓ **- 0 pts Correct**

- 3 pts Lacks explanation about layers and weights.

- 2 pts Should use relu as activation unit.

- 2 pts Activation unit (relu) not mentioned.

- 15 pts Missing solution

2.2 15 / 15

✓ **- 0 pts Correct**

- 2 pts Lacks explanation of the loss by considering different conditions.

- 3 pts Equation not simplified.

- 2 pts Should use relu as activation unit.

- 2 pts Equation not correct.

- 15 pts Missing solution.

2.3 20 / 20

✓ **- 0 pts Correct**

- 2 pts Low accuracy

QUESTION 3

40 pts

3.1 15 / 15

✓ **- 0 pts Correct**

- 3 pts Configurations not mentioned.

- 15 pts Missing solution.

3.2 20 / 20

✓ **- 0 pts Correct**

- 20 pts Not Done

- 5 pts Test or Train accuracy not satisfactory.

3.3 5 / 5

✓ **- 0 pts Correct**

- 5 pts Not Done / Fully Incorrect

- 2 pts Calculation of Parameters not shown

Problem 1 A)

Convolutional Neural Networks capture invariant properties of an image like translational invariance and Rotational due to the presence of Pooling layers in the architecture. Pooling layers pool all pixels in a receptive of a certain size into a single pixel. Recall what's happening in the convolution step. Given an image I of lets say dimensions N×N and a kernel K of lets say M×M, then the convolution of the image with the kernel at position is given by Cij=∑p,qIi−p,j−q * Kp,q. Now lets translate the image lets say k positions down and l positions right. All the pixels that were at position (i,j) now move to (i+k,h+j). Now lets recompute convolution at this new position. Its Ci',j'=∑p,q(Ii+k−p,j+l−q * Kp,q). Now compare this with the notation above. We get i'=i+k and j'=j+l. Hence the convolution for the image at the position (i,j) also shifts by (k,l). The pooling follows which basically pools all the possible translations within a certain receptive field into a single pixel. So basically every translation within this field is mapped to this pixel. This is what introduces translation invariance. This enables the network to learn the object features irrespective of wherever they are. This logic also translate to other invariances.

Problem 1 B)

Batch normalization is a technique for training very deep neural networks that normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of training epochs required to train deep neural networks.

In deep learning, preparing a deep neural network with many layers as they can be delicate to the underlying initial random weights and design of the learning algorithm.

One potential purpose behind this trouble is the distribution of the inputs to layers somewhere down in the network may change after each mini-batch when the weights are refreshed. This can make the learning algorithm always pursue a moving target. This adjustment in the distribution of inputs to layers in the network has alluded to the specialized name internal covariate shift.

The challenge is that the model is refreshed layer-by-layer in reverse from the output to the input utilizing an estimate of error that accept the weights in the layers preceding the current layer are fixed.

Batch normalization gives rich method of parameterizing any deep neural netork. The reparameterization decreases the issue of planning updates across numerous layers.

1.1 **2 / 2**

✓ **- 0 pts** **Correct**

- 1 pts partially correct

Problem 1 A)

Convolutional Neural Networks capture invariant properties of an image like translational invariance and Rotational due to the presence of Pooling layers in the architecture. Pooling layers pool all pixels in a receptive of a certain size into a single pixel. Recall what's happening in the convolution step. Given an image I of lets say dimensions N×N and a kernel K of lets say M×M, then the convolution of the image with the kernel at position is given by Cij=∑p,qIi−p,j−q * Kp,q. Now lets translate the image lets say k positions down and l positions right. All the pixels that were at position (i,j) now move to (i+k,h+j). Now lets recompute convolution at this new position. Its Ci',j'=∑p,q(Ii+k−p,j+l−q * Kp,q). Now compare this with the notation above. We get i'=i+k and j'=j+l. Hence the convolution for the image at the position (i,j) also shifts by (k,l). The pooling follows which basically pools all the possible translations within a certain receptive field into a single pixel. So basically every translation within this field is mapped to this pixel. This is what introduces translation invariance. This enables the network to learn the object features irrespective of wherever they are. This logic also translate to other invariances.

Problem 1 B)

Batch normalization is a technique for training very deep neural networks that normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of training epochs required to train deep neural networks.

In deep learning, preparing a deep neural network with many layers as they can be delicate to the underlying initial random weights and design of the learning algorithm.

One potential purpose behind this trouble is the distribution of the inputs to layers somewhere down in the network may change after each mini-batch when the weights are refreshed. This can make the learning algorithm always pursue a moving target. This adjustment in the distribution of inputs to layers in the network has alluded to the specialized name internal covariate shift.

The challenge is that the model is refreshed layer-by-layer in reverse from the output to the input utilizing an estimate of error that accept the weights in the layers preceding the current layer are fixed.

Batch normalization gives rich method of parameterizing any deep neural netork. The reparameterization decreases the issue of planning updates across numerous layers.

1.2 **3 / 3**

✓ **- 0 pts** **Correct**

- 1 pts did not mention internal co-varient shift

0s completed at 11:49 PM

Problem 1 C)

If the training and testing distributions are different for a neural network, we can expect the Neural Network to achieve a high loss, and thus, a poor performance on the test dataset. This is because when we train a neural network on some training dataset, the neural network learns to generalize to any data-point drawn from the same data distribution as the train dataset, and therefore, can make correct predictions for the data-point . However, if the data-point is sampled from a distribution other than the training data distribution, the neural network will most likely not predict the correct output as the neural network has not been trained using data sampled from this test distribution.

This problem can be mitigated by ensuring that the data drawn from the test distribution is also present in the training, which can then ensure that the neural network can generalize to the test distribution as well.

Few ways to handle this issue is by shuffling the data from different distribution into the training and testing (or train test validation)

We can also perform data augmentation on the one of the distribution (one which has very samples) so that the data imbalance is reduced.

Problem 1 D)

ReLU is a better choice as an activation function for hidden layers, when compared to sigmoid function. This is because the sigmoid function tends to saturate i.e the gradient tends to become 0, as the input approaches large positive or large negative values. The small values of gradients due to the saturation of the sigmoid function can result in vanishing gradients during backprop as we multiply gradients from subsequent layers, which would then result in no improvement.

ReLU activation function does not suffer from such a problem as the gradient is always equal 1 when the input is greater or equal to zero, and thus does not saturate when the input is positive.

This is why we use ReLU activation in hidden layer instead of sigmoid activation function in deep neural networks.

1.3 3 / 3

✓ - **0 pts** Correct

0s completed at 11:49 PM

Problem 1 C)

If the training and testing distributions are different for a neural network, we can expect the Neural Network to achieve a high loss, and thus, a poor performance on the test dataset. This is because when we train a neural network on some training dataset, the neural network learns to generalize to any data-point drawn from the same data distribution as the train dataset, and therefore, can make correct predictions for the data-point . However, if the data-point is sampled from a distribution other than the training data distribution, the neural network will most likely not predict the correct output as the neural network has not been trained using data sampled from this test distribution.

This problem can be mitigated by ensuring that the data drawn from the test distribution is also present in the training, which can then ensure that the neural network can generalize to the test distribution as well.

Few ways to handle this issue is by shuffling the data from different distribution into the training and testing (or train test validation)

We can also perform data augmentation on the one of the distribution (one which has very samples) so that the data imbalance is reduced.

Problem 1 D)

ReLU is a better choice as an activation function for hidden layers, when compared to sigmoid function. This is because the sigmoid function tends to saturate i.e the gradient tends to become 0, as the input approaches large positive or large negative values. The small values of gradients due to the saturation of the sigmoid function can result in vanishing gradients during backprop as we multiply gradients from subsequent layers, which would then result in no improvement.

ReLU activation function does not suffer from such a problem as the gradient is always equal 1 when the input is greater or equal to zero, and thus does not saturate when the input is positive.

This is why we use ReLU activation in hidden layer instead of sigmoid activation function in deep neural networks.

1.4 **2 / 2**

✓ **- 0 pts** **Correct**

- 1 pts vague explaination

problem- 2 (A)

w_1 : weight matrix for layer 1.
w_2 : weight matrix for layer 2

let b_1 : bias for the layer 1
b_2 : bias for the layer 2.

$$z_1 = w_1 X + b_1$$

where x is the input matrix of dimension 28×28

$$\therefore z_1 = w_1 x + b_1$$

Now, we will apply the ReLU activation function for non linearity

$$g_1(z) = (0, max(0, z))$$

$$\therefore a_1 = g(z_1)$$

$$\therefore a_1 = g(w_1 x + b_1)$$

this will be the input for the second layer of our neural network.

$$\therefore z_2 = a_1 w_2 + b_2$$

the output of layer 2 will go throught the sigmoid function for final prediction.

$$g_2(z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

$\therefore \quad a_2 (y_{pred}) = g_2(z_2)$

this is our final prediction (a_2)

$$\therefore \boxed{y_{pred} = g_2(w_2 \times g_1(w_1 x + b_1) + b_2)}$$

where g_2 and g_1 are softmax and ReLU activation function respectively

2.1 15 / 15

✓ **- 0 pts** **Correct**

- 3 pts Lacks explanation about layers and weights.

- 2 pts Should use relu as activation unit.

- 2 pts Activation unit (relu) not mentioned.

- 15 pts Missing solution

problem 2 A · B

cross entropy loss function is given as.

$$-\sum_{i=1}^{10} y_i \log(y_{pred})_i$$

∴ we have 10 classes (0, 1, 2 ... 9)

∴ for m = 50,000 samples in training set.

$$\sum_{j=1}^{m} \left[-\sum_{i=1}^{10} y_{(i)}^{<j>} \log(y_{pred(i)})^{<j>} \right]$$

m : 50,000

$$y_{pred} = g_2(w_2 g_1(w_1 x + b_1) + b_2)$$

where 'g_2' is the softmax function applied on the last layer.

'g_1' is the ReLU activation function applied on the output of first layer.

$z_1 = w_1 x + b_1$ [x : i/p matrix]
[w_1 : weight matrix 1]
[b_1 : bias of layer 1]

$a_1 = g_1(z_1)$ g_1

$z_2 = w_2 x a_1 + b_2$ [w_2 : weight of matrix in layer 2
b_2 ↳ bias of layer 2]

$\therefore \quad a_2 = g_2(z_2)$

$y_{pred} = a_2 = g_2(z_2)$

$\therefore \quad L(w_1, b_1, w_2, b_2) =$

$$-\sum_{i=1}^{10} y_i \log\left(g_2(w_2 * g_1(w_1 * x + b_1) + b_2)\right) \quad (i)$$

$\therefore$ for $m = 50,000$ training sample

$$\sum_{j=1}^{m=50000} \left[-\sum_{i=1}^{10} y_i^{\langle j \rangle} \log\left(g_2\left(w_2 g_1(w_1 x + b_1) + b_2\right)\right)_i^{\langle j \rangle} \right]$$

2.2 15 / 15

✓ **- 0 pts** **Correct**

- 2 pts Lacks explanation of the loss by considering different conditions.

- 3 pts Equation not simplified.

- 2 pts Should use relu as activation unit.

- 2 pts Equation not correct.

- 15 pts Missing solution.

DS5220 Problem Set 4

Problem 2

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 dor

#!pip install torch torchvision matplotlib tqdm


# Setup
import torch
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from tqdm import trange
from torch import nn, optim
from torchsummary import summary

%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 0x7f09d8f3a310>
```

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.

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

✓ 0s completed at 10:33 PM

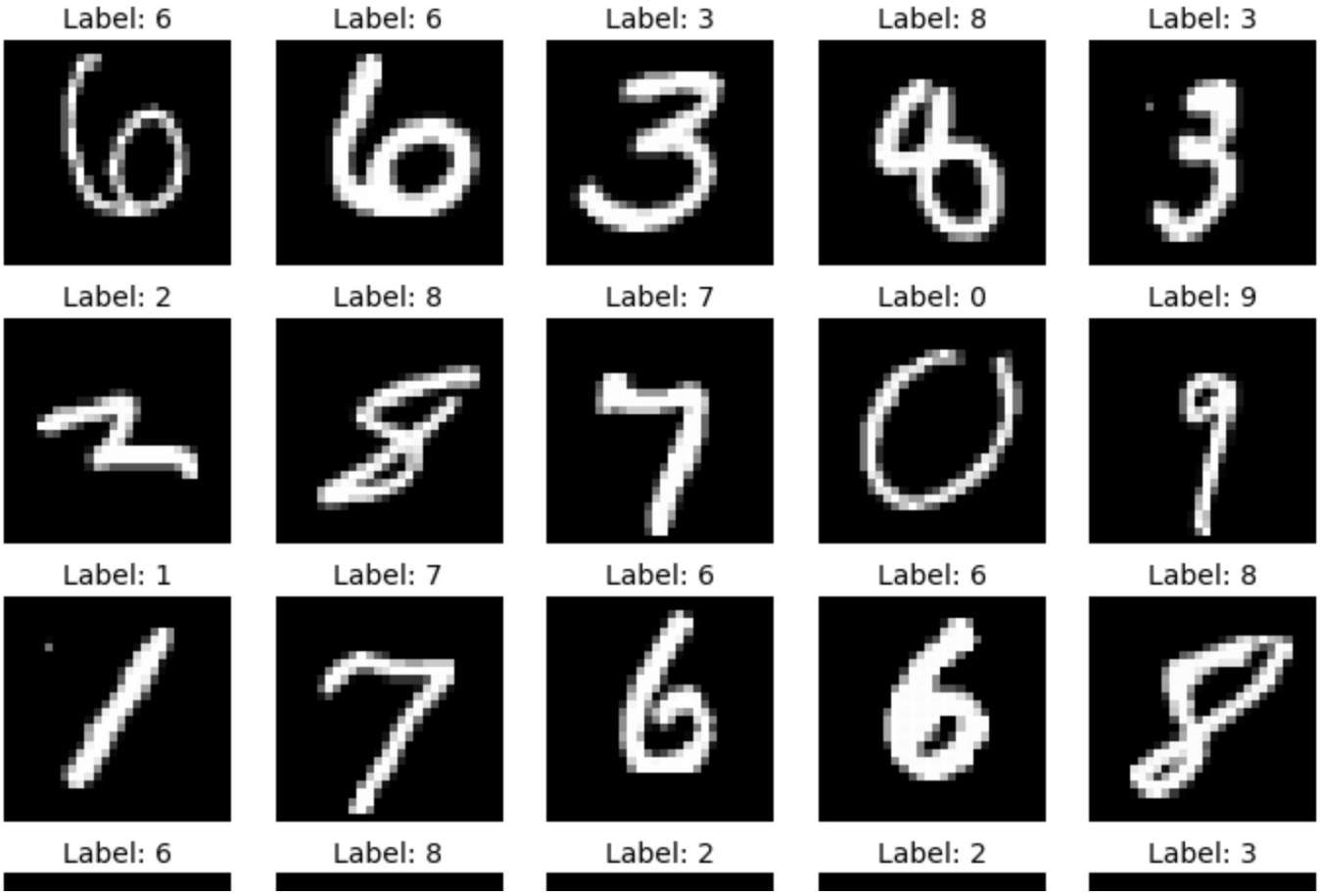
```
test_data = datasets.MNIST('data', train=False, download=True, transform=transform)

## Use the following lines to check the basic statistics of this dataset
# Calculate training data mean and standard deviation to apply normalization to dat
# 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
# 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, shuf
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3
100%                          9912422/9912422 [00:00<00:00,
                              54085973.46it/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/M

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1
100%                            28881/28881 [00:00<00:00,
                                697496.97it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/M

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-
100%                          1648877/1648877 [00:00<00:00,
                              21395332.29it/s]
```

Part 0: Inspect dataset (0 points)

Part b: Inspect dataset (6 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 = next(iter(train_loader))

# 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))

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')
```



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(input_size, hidden_size)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.log_softmax = nn.LogSoftmax(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.flatten(x)
        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        x = self.log_softmax(x)
```

```
        y_output = x

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

model = MNISTClassifierMLP().to(DEVICE)

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

```
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): LogSoftmax(dim=1)
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1                  [-1, 784]               0
           Flatten-2                  [-1, 784]               0
            Linear-3                  [-1, 128]         100,480
              ReLU-4                  [-1, 128]               0
            Linear-5                   [-1, 10]           1,290
        LogSoftmax-6                   [-1, 10]               0
================================================================
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): 0.41
----------------------------------------------------------------
```

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 = []
    criterion = torch.nn.CrossEntropyLoss()
    num_correct_train = 0

    for i, (img, label) in enumerate(train_loader):
```

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

        # ------------------
        # Write your implementation here.
        outputs = model(img)
        loss = criterion(outputs, label)
        pred_train = outputs.argmax(dim=1,keepdim=True) # Get index of largest log-
        num_correct_train += pred_train.eq(label.data.view_as(pred_train)).long().c
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # ------------------

        # Record training loss every log_interval and keep counter of total trainir
        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_train
```

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 = 0.0
    num_correct = 0
    loss_function = torch.nn.CrossEntropyLoss()
    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)
            pred = output.argmax(dim=1,keepdim=True) # Get index of largest log-pro
            num_correct += pred.eq(label.data.view_as(pred)).long().cpu().sum().ite
            test_loss += torch.nn.functional.nll_loss(output, label).item()


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

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

```
    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.Adam(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, D
    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)
    test_correct.append(num_correct)
    train_correct.append(num_correct_train)
print(f"Train accuracy: {train_correct[-1]/len(train_loader.dataset)}")
print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
```

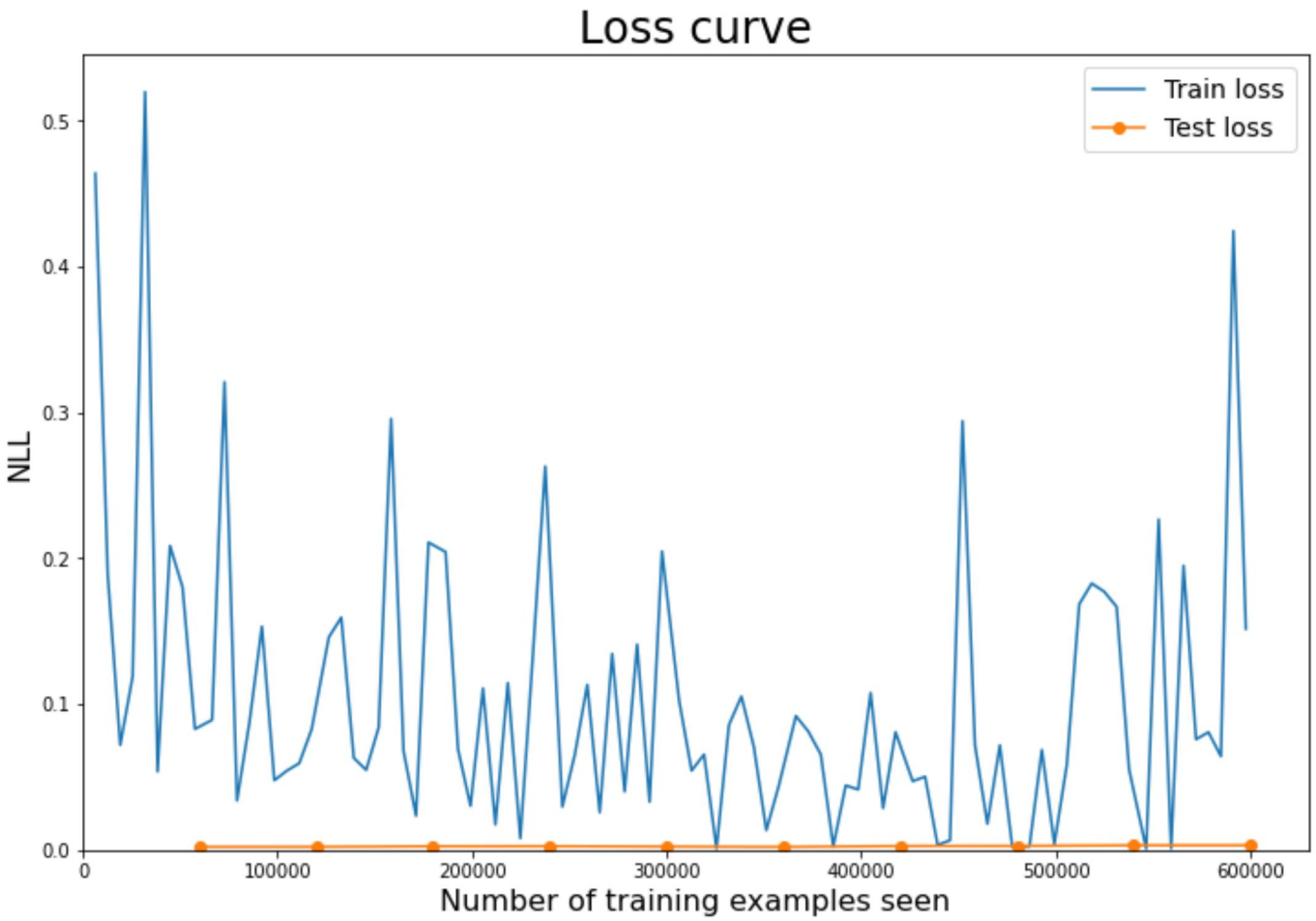
Taining Accuracy is 0.98 and Testing Accuracy is 0.967

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

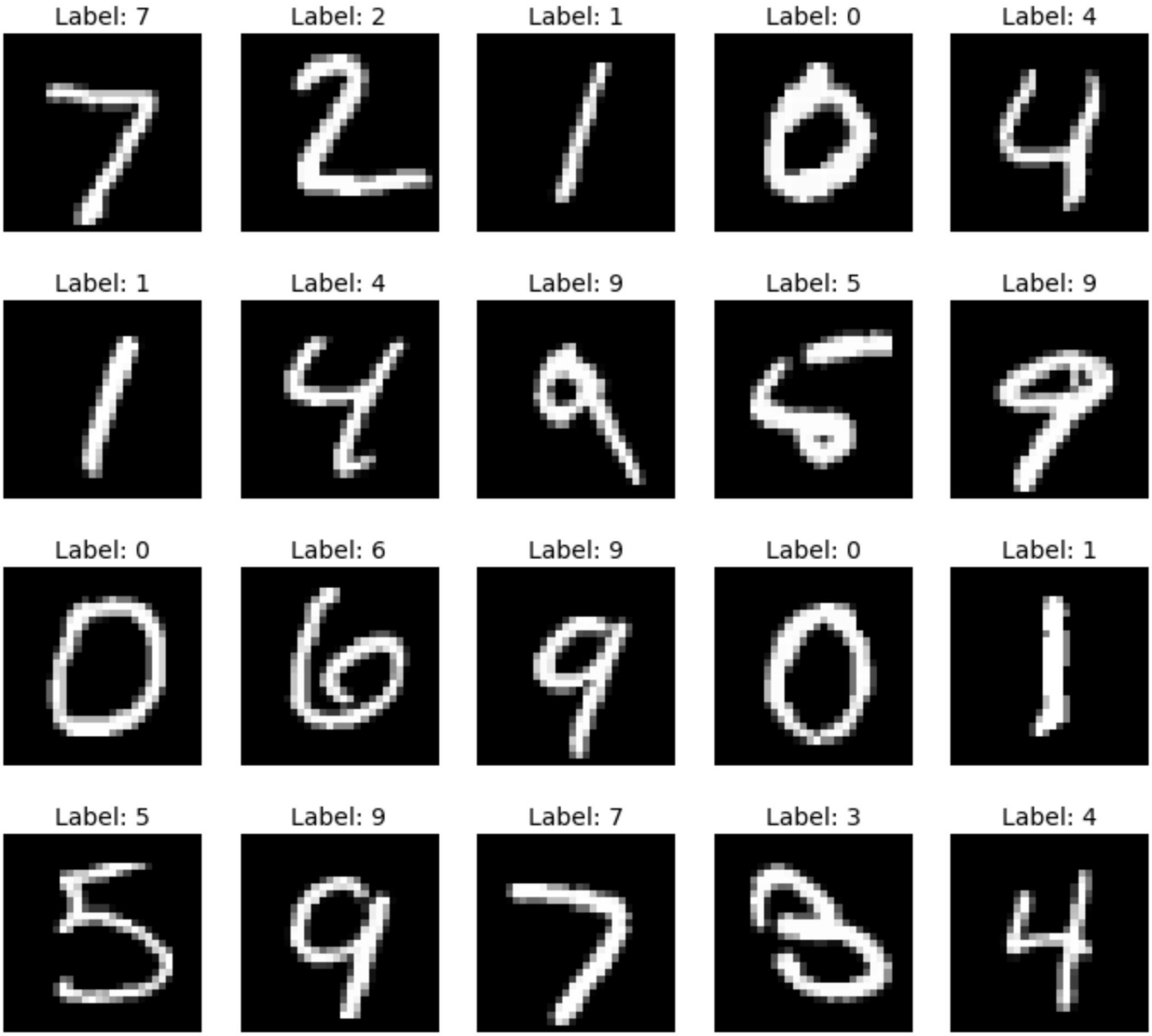
training and test dataset have mean zero and unit variance. Report the result after this normalization step again.

```
# 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)
```



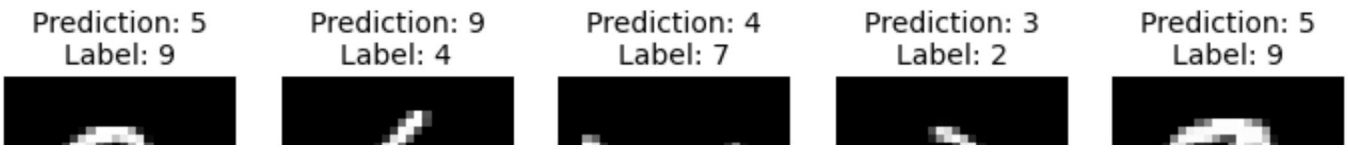

```
# 2. Show the predictions of the first 20 images of the test dataset
images, labels = next(iter(test_loader))
images, labels = images.to(DEVICE), labels.to(DEVICE)

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

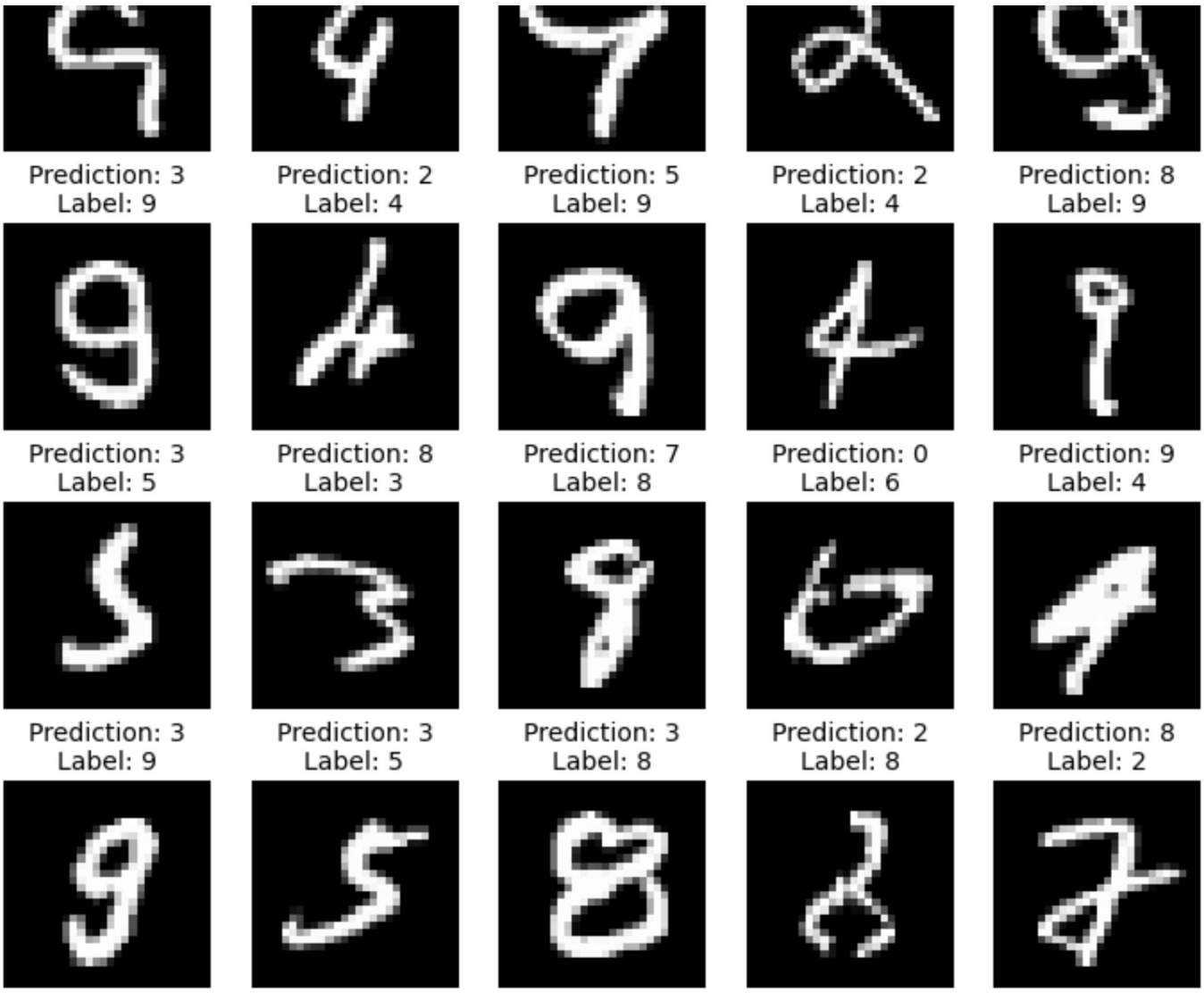
```
fig = plt.figure(figsize=(12, 11))
# ------------------
# Write your implementation here. Use the code provided in Part 0 to visualize the
for i in range(20):
    plt.subplot(4, 5, i+1)
    plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
    plt.title(f'Label: {labels[i]}', fontsize=14)
    plt.axis('off')
# ------------------
```




```
# 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 incorrec
```

```
# predictions seen so far
# Ex) incorrect_imgs should be of shape i x C x H x W, where i is the total number
# incorrect images so far.
incorrect_imgs = torch.Tensor().to(DEVICE)
incorrect_preds = torch.IntTensor().to(DEVICE)
incorrect_labels = torch.IntTensor().to(DEVICE)

incorrect_imgs = []
incorrect_preds = []
incorrect_labels = []

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

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

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

        # Compare prediction and true labels and append the incorrect predictions
        # using `torch.cat`.
        for p, l in zip(pred, labels):
            count += 1
            if p != l:
                incorrect_imgs.append(images[count])
                incorrect_preds.append(p)
                incorrect_labels.append(l)

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

# 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', interpolatic
    plt.title(f'Prediction: {incorrect_preds[i].item()}\nLabel: {incorrect_labels[i
    plt.axis('off')
```




Prediction: 3
Label: 9
Prediction: 2
Label: 4
Prediction: 5
Label: 9
Prediction: 2
Label: 4
Prediction: 8
Label: 9
Prediction: 3
Label: 5
Prediction: 8
Label: 3
Prediction: 7
Label: 8
Prediction: 0
Label: 6
Prediction: 9
Label: 4
Prediction: 3
Label: 9
Prediction: 3
Label: 5
Prediction: 3
Label: 8
Prediction: 2
Label: 8
Prediction: 8
Label: 2

Colab paid products - Cancel contracts here

2.3 **20 / 20**

✓ **- 0 pts** Correct

- 2 pts Low accuracy

DS5220 Problem Set 4

Problem 3

Implement a convolutional neural network to recognize hand-written digits

Problem 3 A)

1. Number of Layers: A convolutional Neural Network has three types of layers (not necessarily always)

1. Convolutional Layer
2. Pooling Layer
3. Fully Connected Layer

2. Layer Type:

1. Convolutional layer: The convolutional layer involves the convolution operation which is one of the fundamental operation in the computer vision literature in which a filter (kernel) K of size say MxM is convolved over the Input matrix I of Dimension NxN, then the convolution of the Matrix with the kernel at position (i,j) is given by $C(i,j)=\sum_{p,q}(I_{i-p,j-q} * K_{p,q})$.
2. Fully connected Layer: (flatten) – takes the output of the previous layers, “flattens” them and turns them into a single vector that can be an input for the next stage.

The first fully connected layer – takes the inputs from the feature analysis and applies weights to predict the correct label.

Fully connected output layer – gives the final probabilities for each label.

3. Filter Size and Max Pooling:

1. Filter Size:Filter is a window that scans the image. A filter is used to judge which feature do a pixel belong to. Does it belong to an arc , straight line, diagonal. One filter could be responsible for detecting arcs within an image, another could be responsible for diagonal lines, etc. They are also known as kernels which scan the image region after another. The

characterise its function. A pixel type is judged by convolving the filter -centred on the pixel- with the image region.The weights of the filters are not set by engineers, they take their value by training.

2. Max Pooling Operation: This layer is responsible for reducing dimentionality of the image it does so by going over the grid of the matrix and picking the max of the grid and assigning the max of that grid value to the single pixel in the output, this is called as max pooling.

Number of Hidden Units and Activation Function for Fully Connected Layer:

- Few Configuration to decide the number of hidden units is as follows: The number of hidden neurons shoould be 2/3 the size of the input layer plus the size of output layer.
- The number of hidden neuron should be less than twice of the size of the input layer.
- Setting the number of hidden layers as 1 and the number of neurons in the hidden layer to be the mean of the size of input and output layer.

Activation functions:

The activation function calculates a weighted total and then adds bias to it to decide whether a neuron should be activated or not. The Activation Function's goal is to introduce non-linearity into a neuron's output.

1) Linear Function: –

• Equation: The equation for a linear function is y = ax, which is very much similar to the equation for a straight line.

• -inf to +inf range

• Applications: The linear activation function is only used once, in the output layer

2) The sigmoid function:

• It's a function that is being plotted in the form of 'S' Shape.

• Formula: A = 1/(1 + e-x)

• Non-linear in nature.

3). Tanh Function: Tanh function, also identified as Tangent Hyperbolic function, is an activation that almost always works better than sigmoid function. It's simply a sigmoid function that has been adjusted. Both are related and can be deduced from one another.

• Equation: f(x) = tanh(x) = 2/(1 + e-2x) – 1 OR tanh(x) = 2 * sigmoid(2x) – 1 OR tanh(x) = 2 * sigmoid(2x) – 1

4). RELU (Rectified linear unit) is the fourth letter in the alphabet. It's the most used activation

method. Hidden layers of neural networks are primarily used.

• Formula: A(x) = maximum (0,x). If x is positive, it returns x; else, it returns 0.

• Value Range: (inf, 0)

5). Softmax Function: The softmax function is a type of sigmoid function that comes in handy when dealing with categorization issues.

• Non-linearity in nature

• Uses: Typically utilised when dealing with many classes. The softmax function would divide by the sum of the outputs and squeeze the outputs for each class between 0 and 1.

Common Configurations:

1. Input -> Convolutiion layer Layer -> Activation Layer -> Fully Connected Layer -> O/P
2. Input -> (Convolutiion layer Layer -> Max/Avg Pooling Layer)x2 or more depending on size of input image -> Activation Layer -> Fully Connected Layer -> O/P

Its always adviseable to reduce the dimensions of Image not very drastically so that the model learns the most out of the data (Image).

3. Input -> (Convolutiion layer Layer + Same/Valid Padding -> Max/Avg Pooling Layer)x3 or more depending on size of input image -> Activation Layer -> Fully Connected Layer -> O/P

```
# Uncomment the below line and run to install required packages if you have not dor
import numpy as np
!pip install torch torchvision matplotlib tqdm
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/d
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /us
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/d
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/
```

3.1 **15 / 15**

✓ **- 0 pts** **Correct**

- 3 pts Configurations not mentioned.

- 15 pts Missing solution.

method. Hidden layers of neural networks are primarily used.

• Formula: A(x) = maximum (0,x). If x is positive, it returns x; else, it returns 0.

• Value Range: (inf, 0)

5). Softmax Function: The softmax function is a type of sigmoid function that comes in handy when dealing with categorization issues.

• Non-linearity in nature

• Uses: Typically utilised when dealing with many classes. The softmax function would divide by the sum of the outputs and squeeze the outputs for each class between 0 and 1.

Common Configurations:

1. Input -> Convolutiion layer Layer -> Activation Layer -> Fully Connected Layer -> O/P
2. Input -> (Convolutiion layer Layer -> Max/Avg Pooling Layer)x2 or more depending on size of input image -> Activation Layer -> Fully Connected Layer -> O/P

Its always adviseable to reduce the dimensions of Image not very drastically so that the model learns the most out of the data (Image).

3. Input -> (Convolutiion layer Layer + Same/Valid Padding -> Max/Avg Pooling Layer)x3 or more depending on size of input image -> Activation Layer -> Fully Connected Layer -> O/P

```
# Uncomment the below line and run to install required packages if you have not dor
import numpy as np
!pip install torch torchvision matplotlib tqdm
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: torchvision in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/d
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /us
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/d
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/
```

```
# 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 0x7f0ed4276fd0>
```

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.

```
# 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 dat
# 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
# 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
```

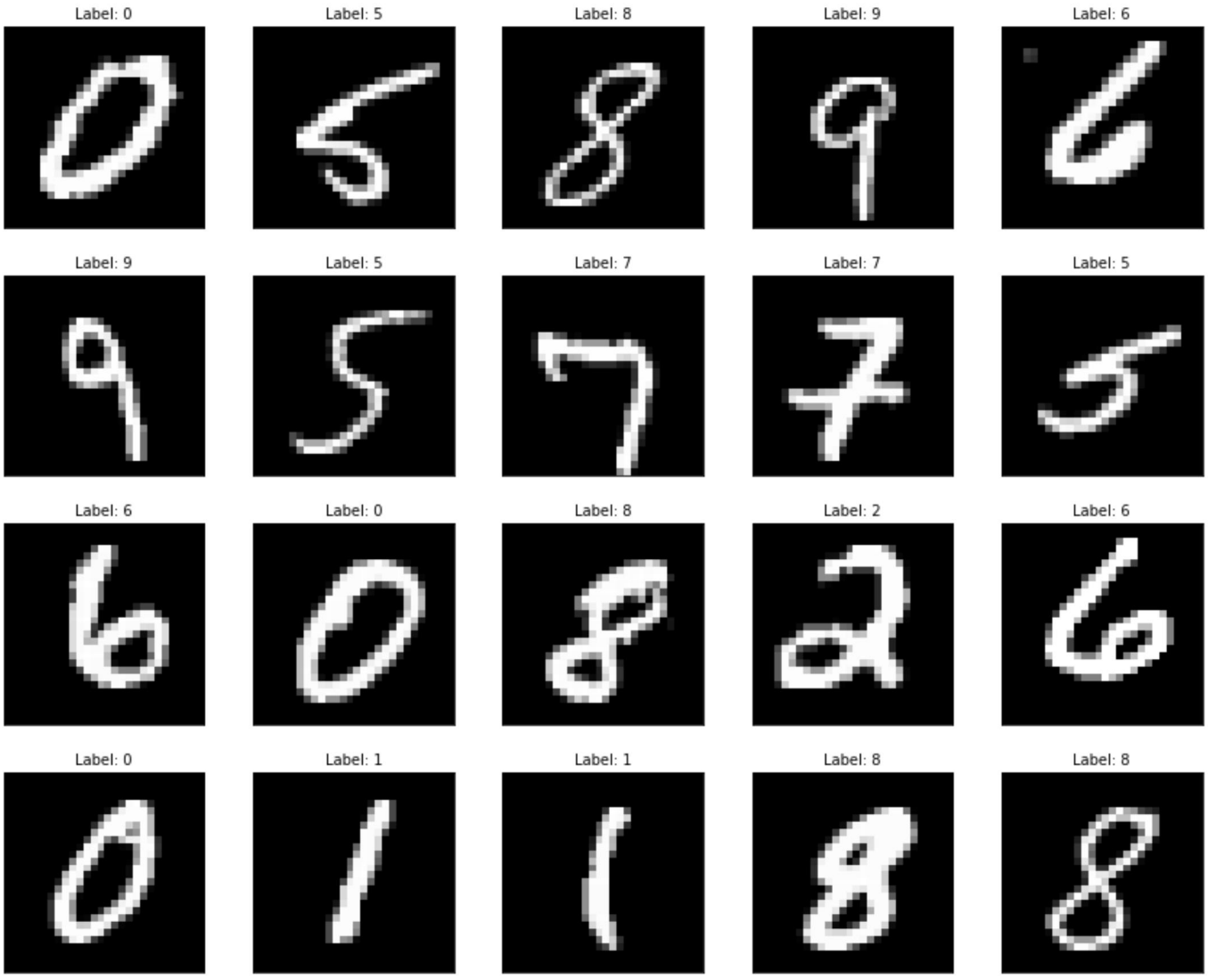
```
batch_size = 64
torch.manual_seed(seed)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuf
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3
100%                          9912422/9912422 [00:00<00:00,
                              37441441.19it/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/M

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1
100%                            28881/28881 [00:00<00:00,
                                510266.32it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/M

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-
100%                          1648877/1648877 [00:00<00:00,
```

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 = next(iter(train_loader))

# 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
dataiter = iter(train_loader)
images, labels = next(dataiter)
```

```
print("Images shape: ", images.shape)
print("Labels shape: ", labels.shape)
print(f'Mean={images.mean()}, Std={images.std()}')

fig = plt.figure(figsize=(12,10))

for i in range(20):
    plt.subplot(4,5,i+1)
    plt.tight_layout()
    plt.imshow(images[i].squeeze(),cmap='gray',interpolation='none')
    plt.title(f'Label: {labels[i]}', fontsize=10)
    plt.xticks([])
    plt.yticks([])
# ------------------
```



Implement a convolutional neural network (10 points)

Write a class that constructs a two-layer neural network as specified in the handout. The class

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 CNN(torch.nn.Module):

    def __init__(self):

        super().__init__()
        self.flatten = torch.nn.Flatten(start_dim=1)
        # ------------------
        # Write your implementation here.
        self.conv1 = torch.nn.Conv2d(1, 10, kernel_size = 5) # first convolutional
        self.conv2 = torch.nn.Conv2d(10,20, kernel_size = 5) # second convolutional
        self.fc = torch.nn.Linear(320,10) # third fully-connected layer
        # ------------------

    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

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

        x = self.conv1(x)
        x = torch.nn.functional.max_pool2d(x,2)
        x = torch.nn.functional.relu(x)

        x = self.conv2(x)
        x = torch.nn.functional.dropout2d(x)
        x = torch.nn.functional.max_pool2d(x,2)
        x = torch.nn.functional.relu(x)

        x = self.flatten(x)
        x = self.fc(x)

        x = torch.nn.functional.relu(x)
        x = torch.nn.functional.log_softmax(x)

        y_output = x

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

model = CNN().to(DEVICE)
```

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

```
CNN(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (fc): Linear(in_features=320, out_features=10, bias=True)
)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:38: UserWarning:
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 10, 24, 24]             260
            Conv2d-2             [-1, 20, 8, 8]           5,020
           Flatten-3                  [-1, 320]               0
            Linear-4                   [-1, 10]           3,210
================================================================
Total params: 8,490
Trainable params: 8,490
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.03
Estimated Total Size (MB): 0.09
----------------------------------------------------------------
```

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 = []
    y_pred = []
    num_correct_train = 0
    for i, (img, label) in enumerate(train_loader):
        img, label = img.to(device), label.to(device)

        # ------------------
        # Copy the implementation from Problem 4 here

        optimizer.zero_grad()

        output = model(img)

        loss = torch.nn.functional.nll_loss(output,label)
        pred_train = output.argmax(dim=1,keepdim=True)
        num_correct_train += pred_train.eq(label.data.view_as(pred_train)).long().c
```

```
        num_correct_train += pred_train.eq(label.data.view_as(pred_train)).long().
        loss.backward()

        optimizer.step()


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

        # Record training loss every log_interval and keep counter of total trainir
        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_train
```

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)
            pred = output.argmax(dim=1,keepdim=True) # Get index of largest log-pro
            num_correct += pred.eq(label.data.view_as(pred)).long().cpu().sum().ite
            test_loss += torch.nn.functional.nll_loss(output, label).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, correct_train = train_one_epoch(train_loader, model, DEVI(
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

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

print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
```

```
Epochs:   0%|          | 0/10 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-pa
Epochs: 100%|██████████| 10/10 [03:11<00:00, 19.18s/it]Test accuracy: 0.954
```

```
print(f"Train accuracy: {np.mean(test_correct)/len(test_loader.dataset)}")
print(f"Test accuracy: {np.mean(train_correct)/len(train_loader.dataset)}")
```

The Train accuracy is 0.96 and the test accuracy is 0.957

```
print(f"Test Loss: {np.mean(train_losses)}")
print(f"Train Loss: {np.mean(test_losses)}")
```

TRAIN LOSS IS 0.0015 AND TEST LOSS IS 0.122

When Compared to the Feed Forward Neural Network from previous model the test accuracy is only slightly better since the dataset is very simple and not complex and the number of channels is also only 1, (CNN shine in RGB images) though the number of parameters which translate to computation power requied for CNN is significantly less as we can see below

Problem 3- C)

3.2 **20 / 20**

✓ **- 0 pts** **Correct**

- 20 pts Not Done

- 5 pts Test or Train accuracy not satisfactory.

Problem 3 - c)

Parameters for the Convolutional Neural Network Architecture

Dimensions of weight matrix 1 (w1): 5 x 5 x 1 x 10, Number of params: 250

Dimensions of Bias matrix 1 (b1) : 1 x 1 x 1 x 10, Number of params: 10

Dimensions of weight matrix 2 (w2): 5 x 5 x 10 x 20, Number of params: 5000

Dimensions of Bias matrix 2 (b2) : 1 x 1 x 1 x 20, Number of params: 20

Dimensions of weight matrix 3 Fully connected layer (w3) : 10 x 320 : 3200

Dimensions of bias matrix 3 Fully connected layer (b3) : 10 x 1 : 10

Total number of Parameters for CNN architecture will be = **8,490**

Parameters for Feed Forward Neural Network (FNN) Architecture

Dimensions of weight matrix 1 (w1): 128 x 784, Number of params: 100,352

Dimensions of Bias matrix 1 (b1) : 128 x 1, Number of params: 128

Dimensions of weight matrix 2 (w2): 10 x 128, Number of params: 1280

Dimensions of Bias matrix 2 (b2) : 10 x 1, Number of params: 10

Total number of Parameters for CNN architecture will be = **101,770**

As we can see the Number of Parameters for CNN (8480) is significantly less than that of FNN architecture (100,770) this is due to the fact that in CNN there is parameter sharing involve when perfoming convolution operation also the dimensions of input images are reduced post convolution operation and max pooling operation.

Colab paid products - Cancel contracts here

3.3 5 / 5

✓ **- 0 pts** **Correct**

- 5 pts Not Done / Fully Incorrect

- 2 pts Calculation of Parameters not shown