DS 5220 Problem Set 4

Ayush Patel

Contribution: Abhinav Nippani, Adwait Patil

Problem 1

Q1. 1)

Invariance means that you can recognize an object as an object, even when its appearance varies in some way. This is generally a good thing, because it preserves the object's identity, category, (etc) across changes in the specifics of the visual input, like relative positions of the viewer/camera and the object.

- **Translation Invariance -** The object is of the same size as what the model has seen although it will be in a different part of the image. ie- two photos with a cat one with the cat in the middle and on either ends of the image our model should still detect the cat in the image.
- **Rotational/View Invariance -** The object has been rotated by some angle. ie - two photos with a cat one with the cat straight other with the cat rotated 90°.

Convolution layers extract edges, shapes and complex figures of the images. So, even if the figures are translated or rotated it won't change the output as the Convolution operation detects the feature present in the images.

Q1. 2)

- Batch normalization is a technique for training very deep neural networks that normalizes the contribution 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.
- It basically limits the effect to which updating the parameters of early layers can effect the distribution of values that next layers see. So in a way, Batch Normalization reduces the problem of the input values changing. It makes them more stable making the model more general.
- Batch Normalization also behaves as a regularizes by adding noise to each hidden layers within that mini-batch, forcing not to rely too much on any one layer.

Q1. 3)

- Having a different training and testing distributions can lead to a model which has high variance and overfitting issue. This is known as covariance shift.
- To tackle this issue we can use some methods like Subsampling. Subsampling can be done on the training dataset to match the distribution of test data. This can help decrease the high variance of the model.
- In Convolution neural network we can use artificial synthesis. This method can be used if lets say training data has a lot of high resolution image and test data has loss resolution image or viz-a-viz, then artificial synthesis can convert the high resolution image to low resolution matching the testing data.
- Feature dropping is an other approach where we drop the features that are drifting from the accepted level of shift, but this can lead to loss of data so should be done carefully.

Q1. 4)

One reason why ReLU is a good activation function in the hidden layers of deep neural network is that it can produce dead neurons. That means that under certain circumstances, network produce regions in which the weight won't update and the output is always 0(in case of sigmoid).

While if ReLU is used there will be no gradient at all so, it is capable of outputting a true zero value.

Other advantages of ReLU activation function are:

- **Computational simplicity:** The rectifier function fucntion is trivial to implement, requising only a max() function.
- **Linear behaviour:** A neural network is easier to optimize when it's behaviour is linear or close to linear.

Problem 2

Q2. 1)

W_1 and W_2 be the weight matrices of layer 1 and 2 respectively

b_1 and b_2 be the biases matrices of layer 1 and 2 respectively
$relu$ is the activation function

Now,
$x \in \mathbb{R}^{784\times1}$
$y \in$ {0,1,2,3,....9}

Therefore,
$W_1 \in \mathbb{R}^{d_0\times784}$ where d_0 can be any random integer Eg. 100 which represnts number of hidden units
$b_1 \in \mathbb{R}^{d_0}$
$W_2 \in \mathbb{R}^{d_1\times d_0}$ where d_1 is 10 because we have 10 output classes
$b_2 \in \mathbb{R}^{d_1}$

Layer 1

$Z_1 = W_1 \times x + b_1$
$A_1 = relu(Z_1)$ Where A_1 is the activation of layer 1

Layer 2

$Z_2 = W_2 \times A_1 + b_2$

$Z_2 \in \mathbb{R}^{10}$ now we apply softmax to Z_2 to get the output in terms of probabilities of each class.

$A_2 = softmax(Z_2)$ Where A_2 is the activation of layer 2

$A_2 \in \mathbb{R}^{10}$ i.e it is a vector of dimension (10,1)
y_pred =$argmax(A_2)$

Now, y_pred in terms of W_1, b_1, W_2, b_2 is :

y_pred = $argmax(softmax(W_2 \times relu(W_1 \times x + b_1) + b_2))$

Note:
$softmax(z_i) : \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$

Q2. 2)

$(x_1, y_1), (x_2, y_2)....(x_m, y_m)$ be the training set

Now,
$x_i \in \mathbb{R}^{28\times28}$
$y_i \in$ {0,1,2,3,....9}

Loss = CrossEntropy Loss_{Avg}

We already have softmax as the activation for the second layer in the neural network so the Cross-entropy loss for two layers will be:

$l(x, y) = -1/m \sum_1^m y_i * log(ypred_i)$

Here ypred is the softmax probability of label.

Now, Loss for the neural network in terms of W_1, b_1, W_2, b_2 is :

loss = $-1/m \sum_1^m y_i * log((softmax(W_2 \times A_1 + b_2)))$

Now, $A1 = relu(W_1 \times x + b_1)$

loss = $-1/m \sum_1^m y_i * log(softmax(W_2 \times relu(W_1 \times x + b_1) + b_2))$
Note:
Average Cross Entropy = $-\frac{1}{m} \sum_1^m y_i * log(ypred_i)$

Q2. 3)

In [1]:
```
# Uncomment the below line and run to install required packages if you have not done so

!pip install torch torchvision matplotlib tqdm
```

```
Requirement already satisfied: torch in d:\anaconda-ds\lib\site-packages (1.13.0)
Requirement already satisfied: torchvision in d:\anaconda-ds\lib\site-packages (0.14.0)
Requirement already satisfied: matplotlib in d:\anaconda-ds\lib\site-packages (3.5.2)
Requirement already satisfied: tqdm in d:\anaconda-ds\lib\site-packages (4.64.1)
Requirement already satisfied: typing-extensions in d:\anaconda-ds\lib\site-packages (from torch) (4.3.0)
Requirement already satisfied: requests in d:\anaconda-ds\lib\site-packages (from torchvision) (2.28.1)
Requirement already satisfied: numpy in d:\anaconda-ds\lib\site-packages (from torchvision) (1.23.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in d:\anaconda-ds\lib\site-packages (from torchvision)
(9.2.0)
Requirement already satisfied: cycler>=0.10 in d:\anaconda-ds\lib\site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: pyparsing>=2.2.1 in d:\anaconda-ds\lib\site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: fonttools>=4.22.0 in d:\anaconda-ds\lib\site-packages (from matplotlib) (4.25.
0)
Requirement already satisfied: packaging>=20.0 in d:\anaconda-ds\lib\site-packages (from matplotlib) (21.3)
Requirement already satisfied: python-dateutil>=2.7 in d:\anaconda-ds\lib\site-packages (from matplotlib) (2.
8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in d:\anaconda-ds\lib\site-packages (from matplotlib) (1.4.2)
Requirement already satisfied: colorama in d:\anaconda-ds\lib\site-packages (from tqdm) (0.4.5)
Requirement already satisfied: six>=1.5 in d:\anaconda-ds\lib\site-packages (from python-dateutil>=2.7->matplo
tlib) (1.16.0)
Requirement already satisfied: certifi>=2017.4.17 in d:\anaconda-ds\lib\site-packages (from requests->torchvis
ion) (2022.9.24)
Requirement already satisfied: charset-normalizer<3,>=2 in d:\anaconda-ds\lib\site-packages (from requests->to
rchvision) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in d:\anaconda-ds\lib\site-packages (from requests->torchvision)
(3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in d:\anaconda-ds\lib\site-packages (from requests->torch
vision) (1.26.11)
```

In [2]:
```
# 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)
```

Out[2]:
```
<torch._C.Generator at 0x1b8b07a21b0>
```

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.

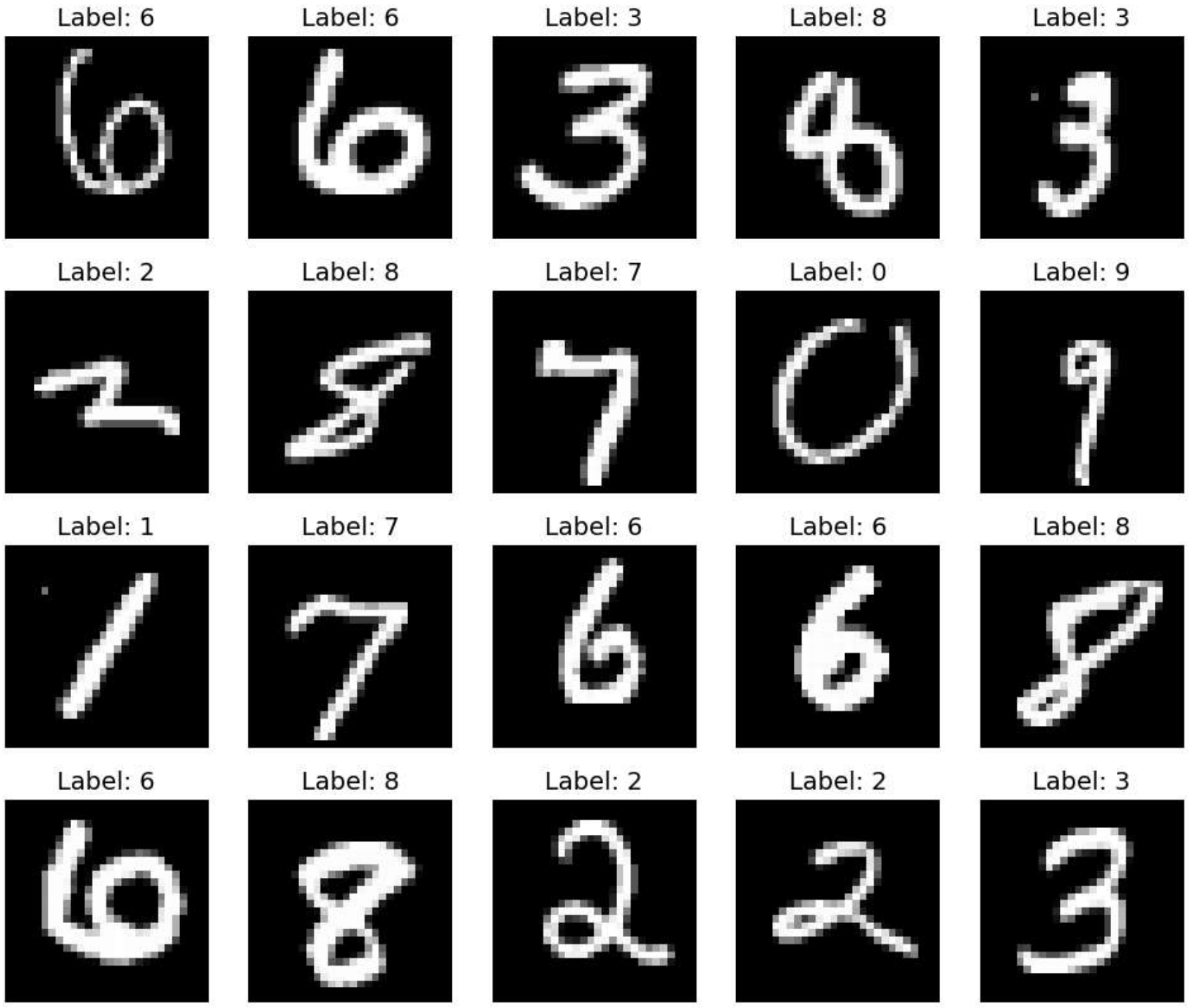
In [3]:
```
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])

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 deviat
# 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=Tru
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=True
```

```
Train Data: Mean=0.1306604762738429, Std=0.3081078071444696
```

Part 0: Inspect dataset (0 points)

In [4]:
```
# 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))
# ------------------
# 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
```



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.

In [5]:
```
from scipy.special import softmax
```

In [6]:
```
import torch.nn as nn
import torch.nn.functional as F
import warnings
warnings.filterwarnings('ignore')
```

In [7]:
```
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)
        x = F.log_softmax(x)
        y_output = x

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

model = MNISTClassifierMLP().to(DEVICE)

# sanity check
print(model)
from torchsummary import summary
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): Softmax(dim=1)
)
----------------------------------------------------------------
        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
----------------------------------------------------------------
```

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.

In [8]:
```python
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()
        # Forward + backward + optimize
        output = model(img)
        _,pred_train = output.max(1)
        num_correct += pred_train.eq(label).sum().item()
        loss = torch.nn.functional.nll_loss(output,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
```

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.

In [9]:
```python
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.max(1)  # Get index of largest log-probability and use that as prediction
            num_correct += pred.eq(label).sum().item()
            loss = torch.nn.functional.nll_loss(output,label,size_average=False)
            test_loss += loss.item()

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

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

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

In [10]:
```
# 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, train_num_counter = train_one_epoch(train_loader, model, DEVICE, optimizer, log_in
    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(train_num_counter)


print(f"Train accuracy: {train_correct[-1]/len(train_loader.dataset)}")
print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
print(f"Train loss: {train_losses[-1]}")
print(f"Test loss: {test_losses[-1]}")
```

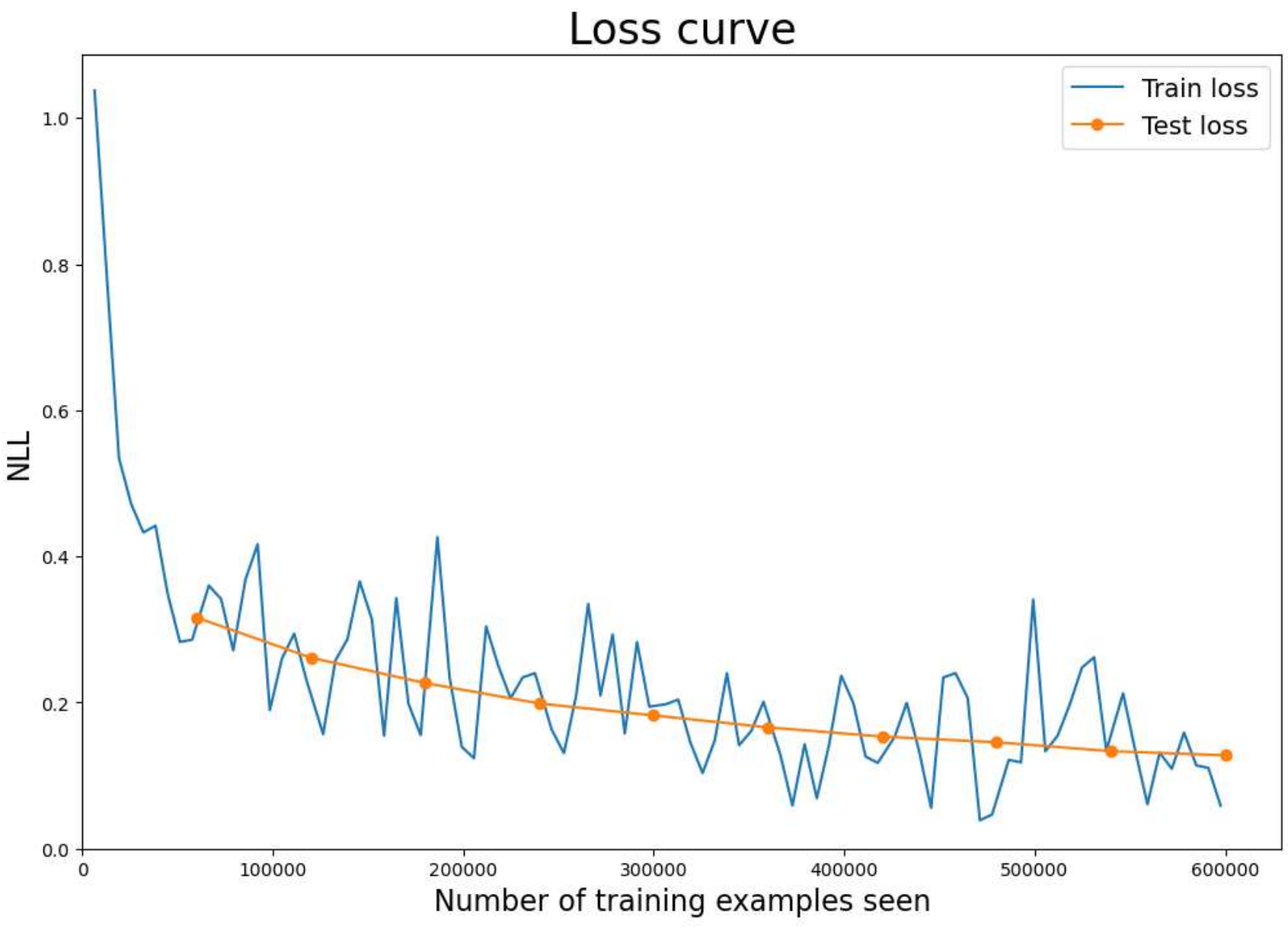
```
Epochs: 100%|██████████| 10/10 [03:50<00:00, 2
3.08s/it]

Train accuracy: 0.9646666666666667
Test accuracy: 0.964
Train loss: 0.05892835184931755
Test loss: 0.1277539134129882
```

Part 4: Inspection

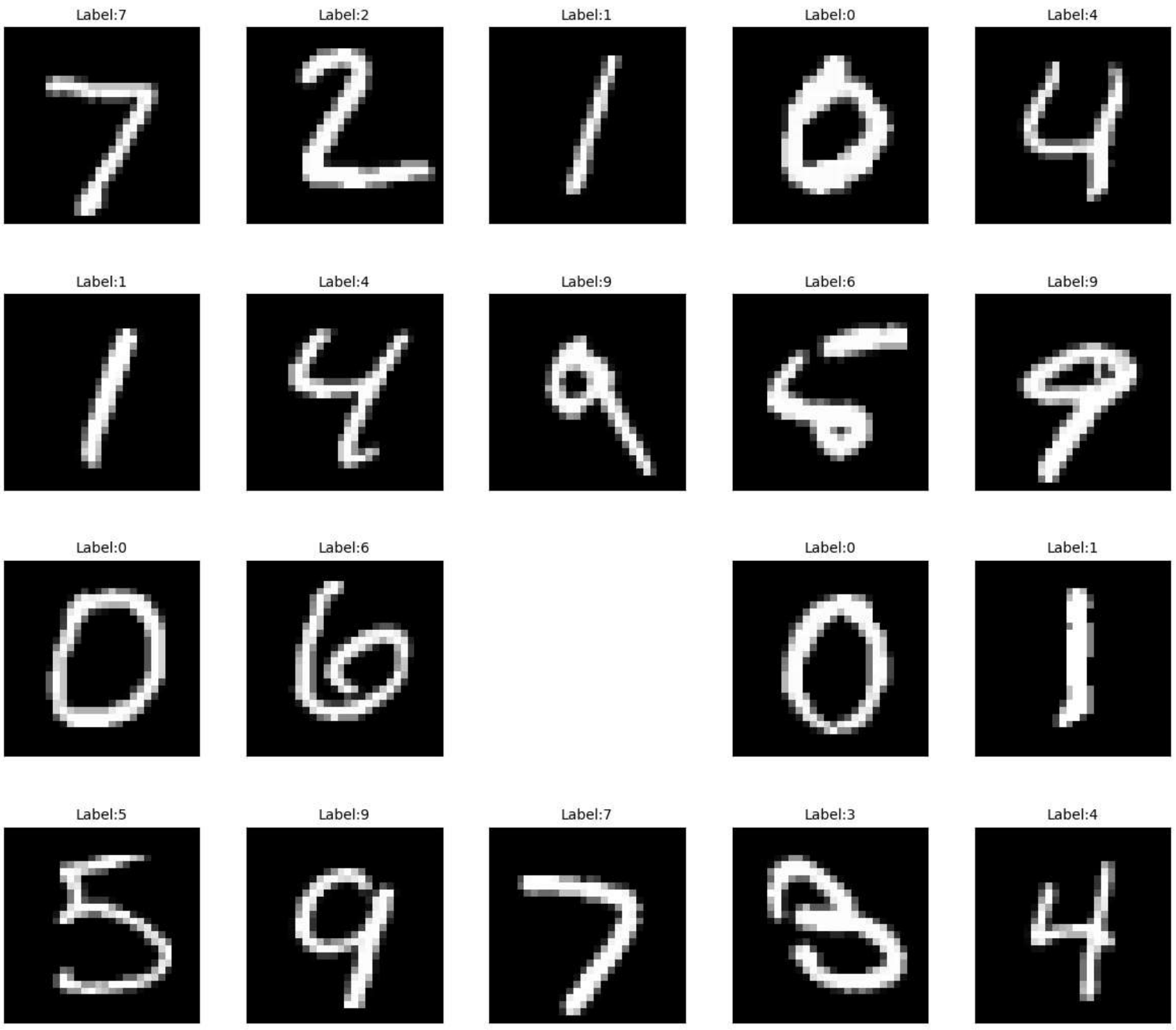
1. Plot the loss curve as the number of epochs increases.
2. Show the predictions of the first 20 images of the test set.
3. Show the first 20 images that the model predicted incorrectly. Discuss about some of the common scenarios that the model predicted incorrectly.
4. Go back to Part 0, where we created 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.

In [11]:
```
# 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)
```

Out[11]: <matplotlib.legend.Legend at 0x1b8c684bf40>

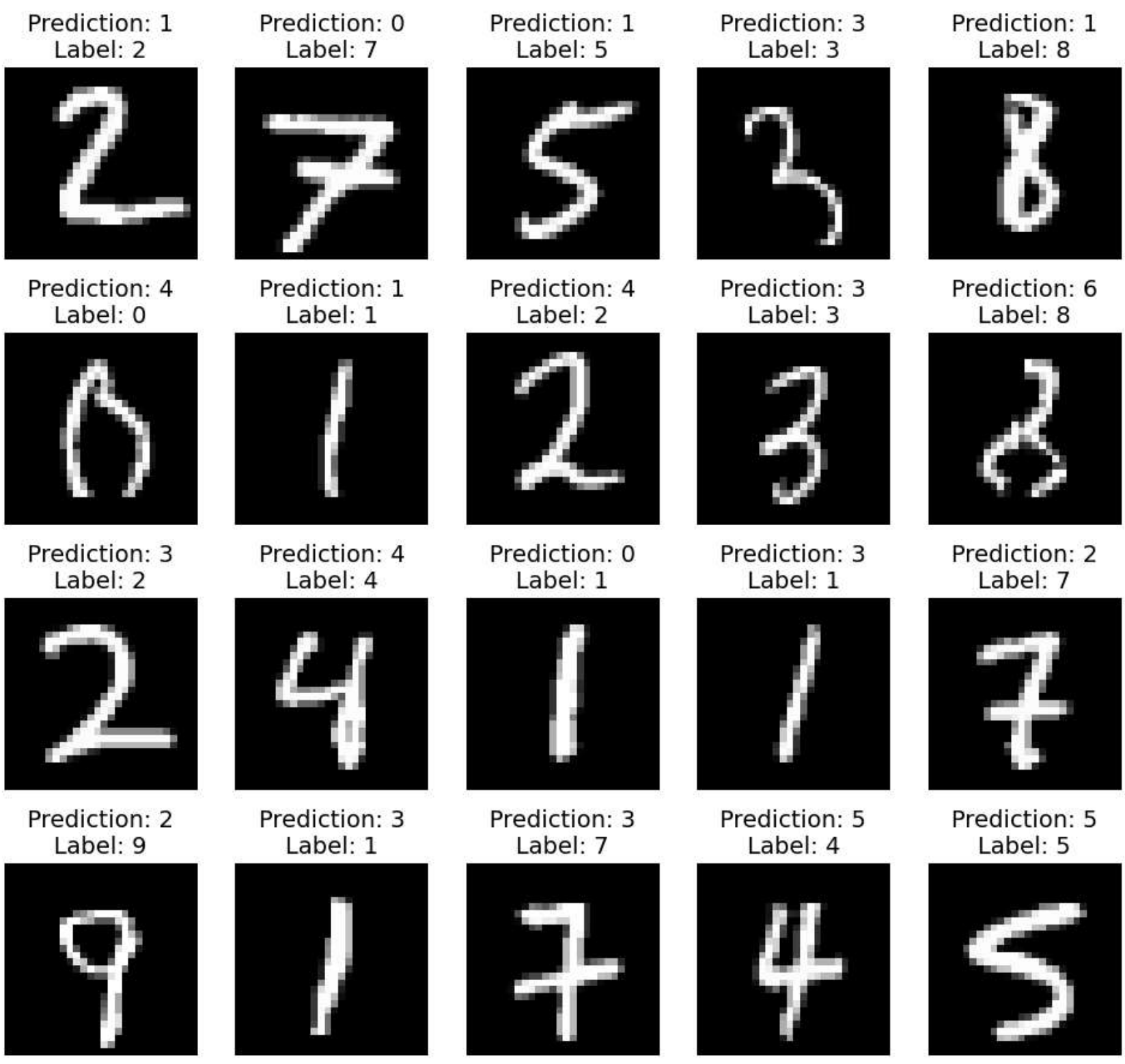


In [12]:
```
# 2. Show the predictions of the first 20 images of the test dataset
dataiter = iter(test_loader)
images, labels = next(dataiter)
images, labels = images.to(DEVICE), labels.to(DEVICE)

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.tight_layout()
    plt.imshow(images[i].cpu().squeeze(),cmap='gray',interpolation='none')
    plt.title(f'Label:{pred[i]}', fontsize=10)
    plt.xticks([])
    plt.yticks([])
plt.show()
```



In [13]:

```
# 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)
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)

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

        output = model(images)
        pred = output.argmax(dim=1)
        incorrect = pred.not_equal(labels).sum().item()
        # Compare prediction and true labels and append the incorrect predictions
        # using `torch.cat`.
        incorrect_imgs.append(images[incorrect])
        incorrect_preds.append(incorrect)
        incorrect_labels.append(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]}\nLabel: {incorrect_labels[i].item()}', fontsize=14)
    plt.axis('off')
plt.show()

```



Conclusion:
1.From the above plot we can observe that the model is predicting wrong for numbers which are incomplete or tilted in an angle that makes it appear similar to another number (Eg. 9 when titled resembles 1).
2.Second scenario where we observe wrong predictions is blurred images. For example in the first row we observe that the 3 when blurred resembles 8.
3.Finally another scenario that we observe is when the numbers are clipped from either top or bottom. Eg we observe 8 in last row when clipped resembles 6.

Problem 3

Q3. 1)

1. Number of Layers:

Every network has a single input layer and a single output layer.

The number of neurons in the input layer equals the number of input variables in the data being processed. The number of neurons in the output layer equals the number of outputs associated with each input.

The layers between the input and output layers are the hidden layers. The number of layers and the number of nodes in each hidden layer are model hyperparameters that you must specify and learn.

The number of hyperparameters (weights, bias) increases with the increase in number of layers.

Determining the Number of layers:

a. Number of Layers = 0:

Only capable of representing linear separable functions or decisions

b. Number of Layers = 1:

Can approximate any function that contains a contains a continuous mapping from one finite space to another

c. Number of Layers > 1:

Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy

2. Layer Type (Convolution and Fully Connected)

Convolution Layer:

Convolution Layer is used to extract features from images.

The weights in this layer are represented by a kernel/filter which slides over the image and extracts the features through a dot product between the kernel values and the pixel values of the image. The values in the kernel are updated everytime backpropagation is implemented.

Let the height, width and number of channels in the image be represented by (h, w, c) respectively. Let the height and width of the kernel be represented by (f_h, f_w) respectively and the number of output channels be c_o. This means that we need to use c_o filters each of size (f_h, f_w). The more the number of filters, the more the number of features learned by the model.

Therefore, the size of the input layer would be (h, w, c) and the size of the output layer would be $((h - f_h + 2 * p)/s + 1, (w - f_w + 2 * p)/s + 1, c_o))$. Here, p is the padding size that needs to be added to the image to prevent loss of information and s is the strides which need to be taken while sliding over the image after each convolution.

Common Convolutional Layers:

a. 1-D Conv Layer:

It moves along a single axis, so it is generally applied to sequential data like text or signals

b. 2-D Conv Layer:

Generally used within deep neural networks (ResNet, AlexNet, etc) for image processing and computer vision tasks.

c. 3-D Conv Layer:

Conv3D is usually used for videos where you have a frame for each time span. These layers usually have more parameters to be learnt than the previous layers.

Fully Connected Layer:

Fully Connected layers in a neural networks are those layers where all the inputs from one layer are connected to every activation unit of the next layer.

Fully Connected Layers compute the linear combination of input layer 'm' number of times where 'm' is the number of output nodes. After the linear combination, it applies an activation function over the linear combination output.

In most popular machine learning models, the last few layers are full connected layers which compiles the data extracted by previous layers to form the final output. It is the second most time consuming layer second to Convolution Layer.

Models where Fully Connected Layers are commonly used:

a. After Convolutional layers in Image Processing Tasks (ResNet, AlexNet, etc) as a cheap way of learning non-linear combinations

b. To approximate any function that contains a continuous mapping from one finite space to another

c. Can be used to reduce the dimensionality

3. Filter Size for Convolution Layer and Max Pooling

Filter Size:

Filter Size refers to the dimensions of the filter/kernel (f_h, f_w) used during the convolutional layer as explained above. It determines the size of the receptive field where the information is extracted.

By convention, the filter size is usually odd in computer vision tasks. One of the reasons for this is that we get a central position during the convolutional process. Another reason is that if the filter size is even , we might need asymmetrical padding which is not desirable.

Max Pooling:

Max pooling reduces the dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer. While sliding the kernel/filter over the image, instead of convolution, max pooling outputs the maximum pixel value of the region covered by the filter.

Since max pooling reduces the resolution of the image, it reduces the number of parameters and consequently the computational load. Max Pooling also helps in reducing overfitting since we are taking the higher activated pixels and discarding the lower valued pixels that are not as activated. Therefore, max pooling preserves the most important characteristics while reducing dimensionality.

4. Number of Hidden Units and Activation Function for Fully Connected Layer

Number of Hidden Units:

Number of Hidden Units are the number of neurons in the hidden layer. This is the number of times linear combination would be done on the input layer. The more the number of hidden units, more the features extracted from the input layer.

Generally, the number of hidden units is the two-thirds the number of input units, plus the number of output units

Activation Function:

Activation Functions are used to prevent linearity or introduce non linearity in the neural network. This is necessary because, complex problems cannot be solved by linear equations. If activation functions are not used, the neural network is essentially a linear regression model which is not desirable.

Activation Functions being non linear and differentiable, are very useful in backpropagation for updating the weights.

Activation Functions being non-linear and differentiable, are very useful in backpropagation for updating the weights.
Common Activation Functions and their usecases:

a. Sigmoid and Softmax:

Sigmoid is used for binary classification methods where we only have 2 classes, while SoftMax applies to multiclass problems

b. Relu (Rectified Linear Unit):

It trains faster as it overcomes the problem of vanishing gradients. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.

c. Tanh:

Usually used in hidden layers of a neural network as its values lie between -1 to 1 therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

Some common configurations of CNNs:

LeNet:

Used for hand-written digit recognition (MNIST) and other image classification tasks

LeNet is a 7 layer architecture that excludes the input layer. Consisting of 3 convolution layers, a fully connected layer (FC), and an output layer with learnable weights. 32×32 pixels image size as the input. In the first convolution, a 5×5 kernel that outputs in 6 28×28 feature maps. Subsample using 2×2 stride 2 filter that outputs in 6 14×14 feature maps. Convolution 16 5×5 kernel that outputs in 16 10×10 feature maps. Subsample using 2×2 stride 2 filter that outputs in 16 5×5 feature maps. Fully connected dense layers of 120,84,10 layers

AlexNet:

was the first architecture that lowered down the top-5% error from 26% to 15.3% in ImageNet Classification.

Comprises 5 convolutional layers and 3 dense layers. There were two news concepts with these 8 layers that were introduced that are MaxPooling and the use of the ReLu Activation function.

AlexNet can be used for Natural Language Processing, Medical Image Analysis and Audio Classification tasks.

VGG16:

has 13 convolutional and 3 dense layers, picking up the ReLu tradition from AlexNet. The architecture was developed by the Visual Geometry Group (VGG).

Kernel size = 3×3, stride = 1×1, padding = same, the configuration of each convolutional layer. The only changes that were present were the number of filters. To retain the feature maps across blocks used Padding permitted an increase in layers. Windows size = 2×2 and stride = 2×2, the max pool layer configuration. Therefore the size of the image is halved at every pool layer. RGB image as an input image of 224× 224 pixels. Therefore, input size = 224×224×3. The training was done in stages so as to overcome the problem of vanishing/exploding gradients.

VGG16 is generally used for object detection tasks.

Q3. 2)

In [14]:
```
# Uncomment the below line and run to install required packages if you have not done so
#!pip install torch torchvision matplotlib tqdm
#! pip install torchsummary
```

In [15]:
```
# 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)
```

Out[15]:
```
<torch._C.Generator at 0x1b8b07a21b0>
```

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.

In [16]:
```
# 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 deviat
# 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=Tru
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=True
```

Part 0: Inspect dataset (0 points)

In [17]:
```
# 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()
dataiter = iter(train_loader)
images , labels = next(dataiter)

# 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.tight_layout()
#     plt.imshow(images[i].squeeze(),cmap='gray',interpolation='none')
#     plt.title(f'Label:{labels[i]}', fontsize=10)
#     plt.xticks([])
#     plt.yticks([])
# plt.imshow(images[1].squeeze())
```

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

```
<Figure size 1200x1000 with 0 Axes>
```

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.

In [18]:

```
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(in_channels=1,out_channels=10,kernel_size=(5,5),stride=1,padding=0) # j
        self.act1 = torch.nn.ReLU() #Relu activation
        self.maxpool1 = torch.nn.MaxPool2d(kernel_size=(2,2))
        self.conv2 = torch.nn.Conv2d(in_channels=10,out_channels=20,kernel_size=(5,5),stride=1,padding=0) #
        self.act2 = torch.nn.ReLU()
        self.maxpool2 = torch.nn.MaxPool2d(kernel_size=(2,2))
        self.fc1 = torch.nn.Linear(in_features=320,out_features=128) # third fully-connected layer
        self.fc2 = torch.nn.Linear(in_features=128,out_features=10)
        #self.act3 = torch.nn.Softmax()
        # ------------------

    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.conv1(x)
        x = self.act1(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.act2(x)
        x = self.maxpool2(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        x = F.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))
  (act1): ReLU()
  (maxpool1): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (act2): ReLU()
  (maxpool2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=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
              ReLU-2          [-1, 10, 24, 24]               0
         MaxPool2d-3          [-1, 10, 12, 12]               0
            Conv2d-4            [-1, 20, 8, 8]           5,020
              ReLU-5            [-1, 20, 8, 8]               0
         MaxPool2d-6            [-1, 20, 4, 4]               0
           Flatten-7                 [-1, 320]               0
            Linear-8                 [-1, 128]          41,088
            Linear-9                  [-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): 0.12
Params size (MB): 0.18
Estimated Total Size (MB): 0.31
----------------------------------------------------------------
```

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

In [19]:
```python
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 4 here
        optimizer.zero_grad()
        # Forward + backward + optimize
        output = model(img)
        _,pred_train = output.max(1)
        num_correct += pred_train.eq(label).sum().item()
        loss = torch.nn.functional.nll_loss(output,label)
        loss.backward()
        optimizer.step()

        # Appending losses

        # 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.

In [20]:
```python
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.max(1)  # Get index of largest log-probability and use that as prediction
            num_correct += pred.eq(label).sum().item()
            loss = torch.nn.functional.nll_loss(output,label)
            test_loss += loss.item()

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

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

In [21]:
```
# 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, train_num_correct = train_one_epoch(train_loader, model, DEVICE, optimizer, log_in
    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(train_num_correct)

print(f"Test accuracy: {test_correct[-1]/len(test_loader.dataset)}")
print(f"Train accuracy: {train_correct[-1]/len(train_loader.dataset)}")
print(f"Test loss: {test_losses[-1]}")
print(f"Train loss: {train_losses[-1]}")
```

```
Epochs: 100%|██████████| 10/10 [04:22<00:00, 2
6.30s/it]

Test accuracy: 0.9846
Train accuracy: 0.98445
Test loss: 0.0007151925156169454
Train loss: 0.022231418639421463
```

Comparision of the test accuracy of the Feed Forward and Convolution Neural Network

We can see that test accuracy of the Covolution Neural Network (98.08%) is better than Feed Forward Neural network (96.4%). Since, the dataset is based on images, we can conclude that CNN is better as it takes into consideration of spatial features.

Q3. 3)

The number of parameters in neural network of **Problem 2** are:

- n_1 : Number of neurons in input layer
- n_2 Number of neurons in hidden layer
- n_3 : Number of neurons in output layer

In the ANN model we had,

- n_1 :784
- n_2 128
- n_3 : 10

Therefore the total number of parameters is:
number of connection between the input layer and second layer + number of connection between bias and second layer + number of connections between the secondand the third layer + number of connections between the bias ofsecond layer and output layer.
Therefore, Parameters = $(784 * 128 + 128) + (128 * 10 + 10) = 101770$

For Problem 3:
In Problem 3 we used Convolution Neural Network.
Let us assume we have taken filter of (n,n) size and the input feature map be 'a' and output feature map be 'b'.
Therefore, parameter would be $(n * n * +1) * b$
In our model there are 2 Convolution layer,
First convolution layer: $(5 * 5 * 1 + 1) * 10 = 260$
Second convolution layer: $(5 * 5 * 10 + 1) * 20 = 5020$
Number of parameters of first fully connected layer: $320 * 128 + 128 = 41088$
Number of parameters of second fully connected layer: $128 * 10 + 10 = 1290$
Total = $260 + 5020 + 41088 + 1290 = 47658$

Conclusion:

- Since, there are less parameters in Convolutional Neural Network as compared to Feed Forward Neural Network. ConvolutionvNeural Network is more parameter efficient.

- From this we can conclude that Convolutional Neural Network outperform Feed Forward Neural Network both in terms of accuracy and computation time for image classification.