PS1-MNIST-Handout

October 3, 2020

1 CS7180 Problem Set 1: Implement a two-layer neural network to recognize hand-written digits (40 points)

Welcome to CS7180!

Before you start, make sure to read the problem description in the handout pdf.

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

# !pip install torch torchvision matplotlib tqdm
```

[2]:
```
# pip install torch
```

[17]:
```
# Setup
import torch
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from tqdm import trange
from torch.autograd import Variable

%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)
```

[17]:
```
<torch._C.Generator at 0x2ddc1192030>
```

1.1 Get MNIST Data

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

[18]:
```python
# Initial transform (convert to PyTorch Tensor only)
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

# 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}')

# Perform normalization of train and test data using calculated training mean␣
 ↪and standard deviation
# This will convert data to be in the range [-1, 1]
#transform = transforms.Compose([
#    transforms.ToTensor(),
#    transforms.Normalize((train_mean, ), (train_std, ))
#])
train_data.transform = transform
test_data.transform = transform

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

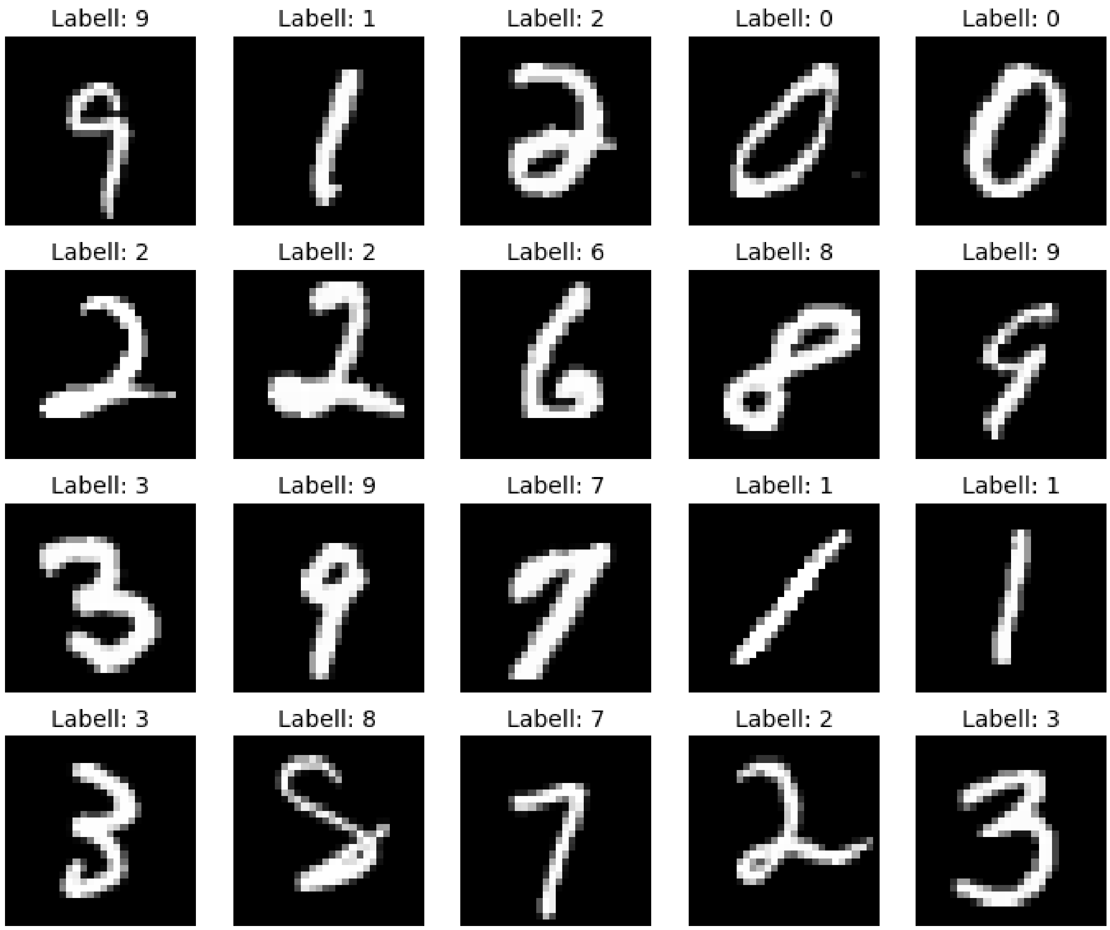
1.2 Part 0: Inspect dataset (0 points)

[19]:
```python
test_data
```

[19]:
```
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
           )
```

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

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

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

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

for i in range(20):
    plt.subplot(4, 5, i+1)
    plt.imshow(images[i].squeeze(), cmap='gray', interpolation='none')
    plt.title(f'Labell: {labels[i]}', fontsize=14)
    plt.axis('off')

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

```
Images shape:  torch.Size([64, 1, 28, 28])
Labels shape:  torch.Size([64])
Mean=0.12825286388397217, Std=0.3058689832687378
```



1.3 Part 1: Implement a two-layer 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.

[21]:
```
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)

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

```
        # Linear layer obtaining input to the hidden layer from the input layer
        self.fc1 = torch.nn.Linear(input_size,hidden_size)

        # Applying activation to the output of hidden layer before feeding into␣
↪final layer
        self.act = torch.nn.ReLU()

        # Linear layer -> narrowing to 10 outputs from hidden layer
        self.fc2 = torch.nn.Linear(hidden_size,output_size)

        # Prevent overfitting
        # self.dropout = torch.nn.Dropout(0.2)
        self.log_softmax = torch.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)

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

        x = self.act(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)
```

```
MNISTClassifierMLP(
  (flatten): Flatten()
  (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()
)
```

1.4 Part 2: Implement an optimizer to train the neural net model (10 points)

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

[22]:
```
def train_one_epoch(train_loader, model, device, optimizer, log_interval,␣
↪epoch):
    model.train()
    losses = []
    counter = []

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

#         img, label = Variable(img), Variable(label)
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(img)
        # calculate the loss
        loss = torch.nn.functional.nll_loss(output,label)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        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
```

1.5 Part 3: Run the optimization procedure and test the trained model (10 points)

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.

[23]:
```
def test_one_epoch(test_loader, model, device):
    model.eval()
    test_loss = 0
    num_correct = 0

#     model.eval()
```

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

#              img, label = Variable(img), Variable(label)
            output = model(img)
            pred = output.argmax(dim=1)
            test_loss += torch.nn.functional.
 ↪nll_loss(output,label,reduction='sum').item()
            num_correct += pred.eq(label.view_as(pred)).sum().item()

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

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

[24]:
```
# 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 = []
for epoch in trange(max_epochs, leave=True, desc='Epochs'):
    train_loss, counter = train_one_epoch(train_loader, model, DEVICE,␣
 ↪optimizer, log_interval, epoch)
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

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

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

```
Epochs:
```

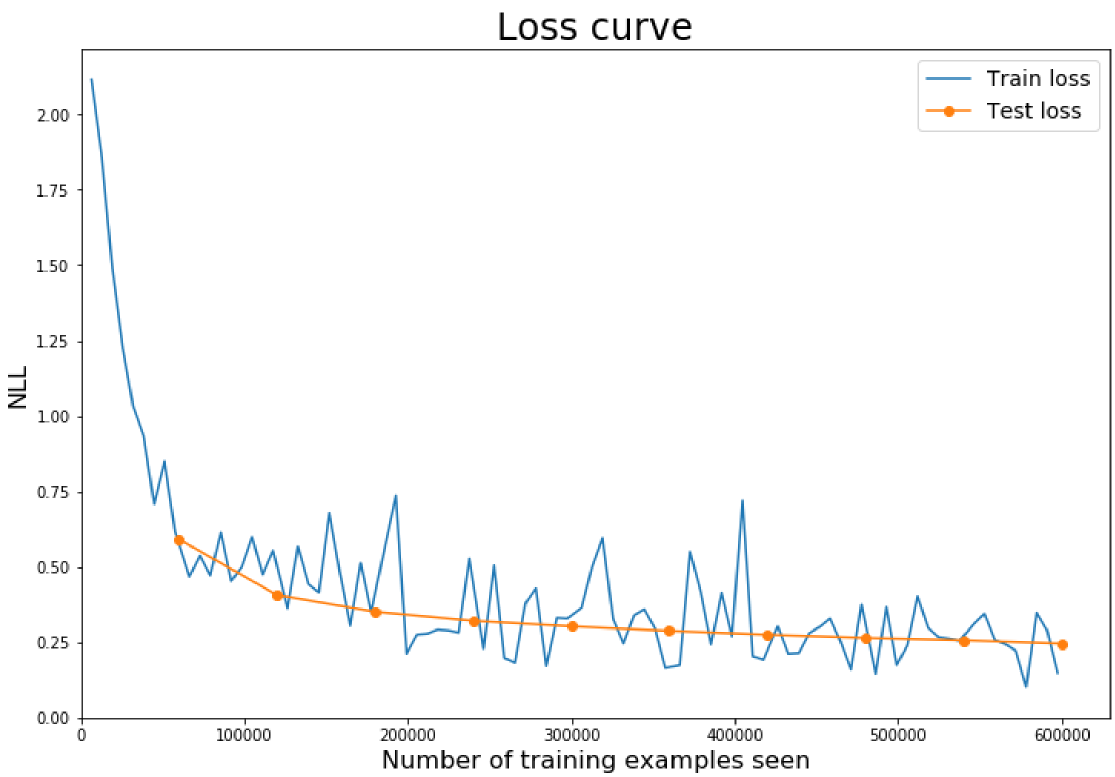
```
100%|                                                                    |
10/10 [01:44<00:00, 10.44s/it]

Test accuracy: 0.9305
```

1.6 Part 4: Ablation studies (10 points)

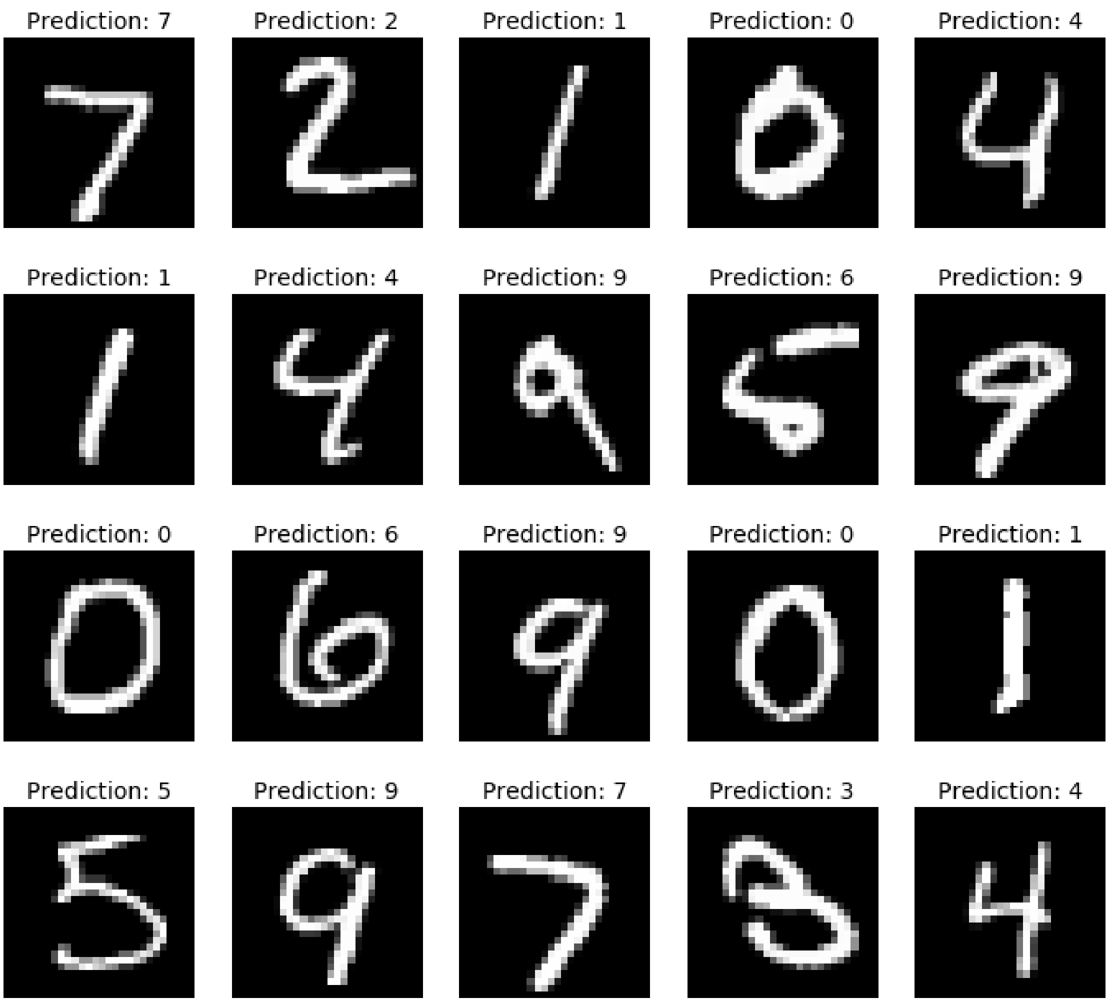
1. Plot the loss curve as the number of epochs increases
2. Show the predictions of the first 20 images of the test set (4 points)
3. Show the first 20 images that the model predicted incorrectly. Discuss about some of the common scenarios that the model predicted incorrectly (4 points)
4. Go back to Part 0, where we created the tranform component to apply on the training and test datasets. Re-run the code after normalizing the training and test dataset to have mean zero and unit variance. Report what you find (2 points)

[25]:
```
# 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)
```

[25]:
```
<matplotlib.legend.Legend at 0x2ddc419e608>
```

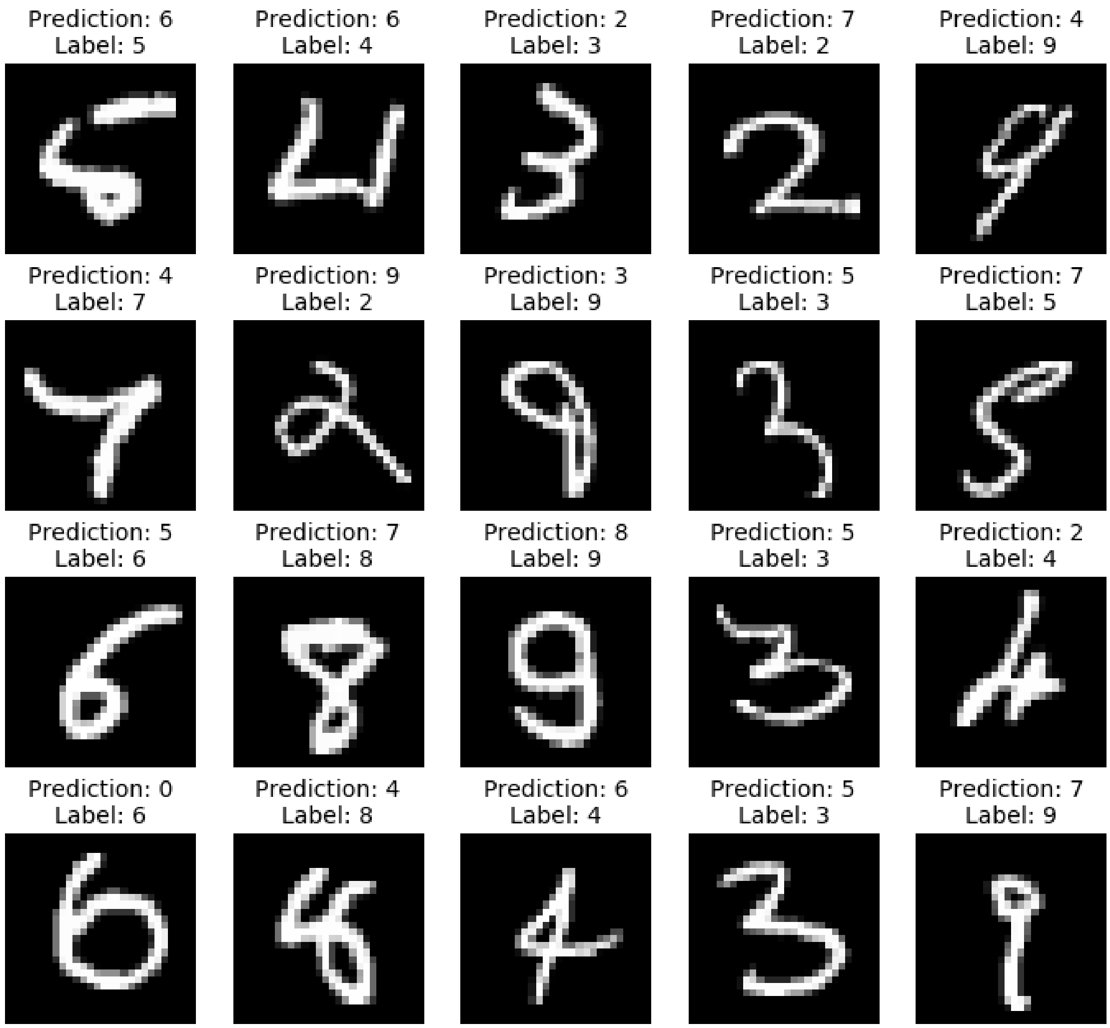


[26]:
```
# 2. Show the predictions of the first 20 images of the test dataset
images, labels = iter(test_loader).next()
images, labels = images.to(DEVICE), labels.to(DEVICE)

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

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

# ------------------
# Write your implementation here. Use the code provided in Part 0 to visualize␣
↪the images.

for i in range(20):
    plt.subplot(4,5,i+1)
    plt.imshow(images[i].squeeze().cpu().numpy(), cmap='gray',␣
↪interpolation='none')
    plt.title(f'Prediction: {pred[i]}',fontsize=14)
    plt.axis('off')

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



[27]:
```
# 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)

# creating lists as we cannot append predictions and labels to tensor objects
incorrect_preds = []
incorrect_labels = []
```

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

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

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

        # Compare prediction and true labels and append the incorrect␣
↪predictions
        # using `torch.cat`.
        for index, i in enumerate(output):
            if pred[index] !=labels[index]:
                incorrect_imgs = torch.cat([incorrect_imgs,␣
↪images[index]],dim=0)
                incorrect_preds.append(pred[index])
                incorrect_labels.append(labels[index])

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

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



Discuss about some of the common scenarios that the model predicted incorrectly (4 points)

The numbers that are predicted incorrectly might be because of training data being too similar to each other, so when deployed on testing data gives poor accuracy.

Also, few digits are hard to recognize for human eyes.

MNSIT data is already somewhat preprocessed. So model might expect to preprocess our test images as well. (like normalizing the training and testing data in the same way). Can also follow PCA or normalizing by Z-score.

Go back to Part 0, where we created the tranform component to apply on the training and test datasets. Re-run the code after normalizing the training and test dataset to have mean zero and unit variance. Report what you find (2 points)

[28]:
```
# 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)

# 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}')

# Perform normalization of train and test data using calculated training mean␣
 ↪and standard deviation
# This will convert data to be in the range [-1, 1]
transform = transforms.Compose([transforms.ToTensor(),transforms.
 ↪Normalize((train_mean, ), (train_std, ))])
train_data.transform = transform
test_data.transform = transform

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

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

[29]:

```
# 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 = []
for epoch in trange(max_epochs, leave=True, desc='Epochs'):
```

```
    train_loss, counter = train_one_epoch(train_loader, model, DEVICE,␣
↪optimizer, log_interval, epoch)
    test_loss, num_correct = test_one_epoch(test_loader, model, DEVICE)

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

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

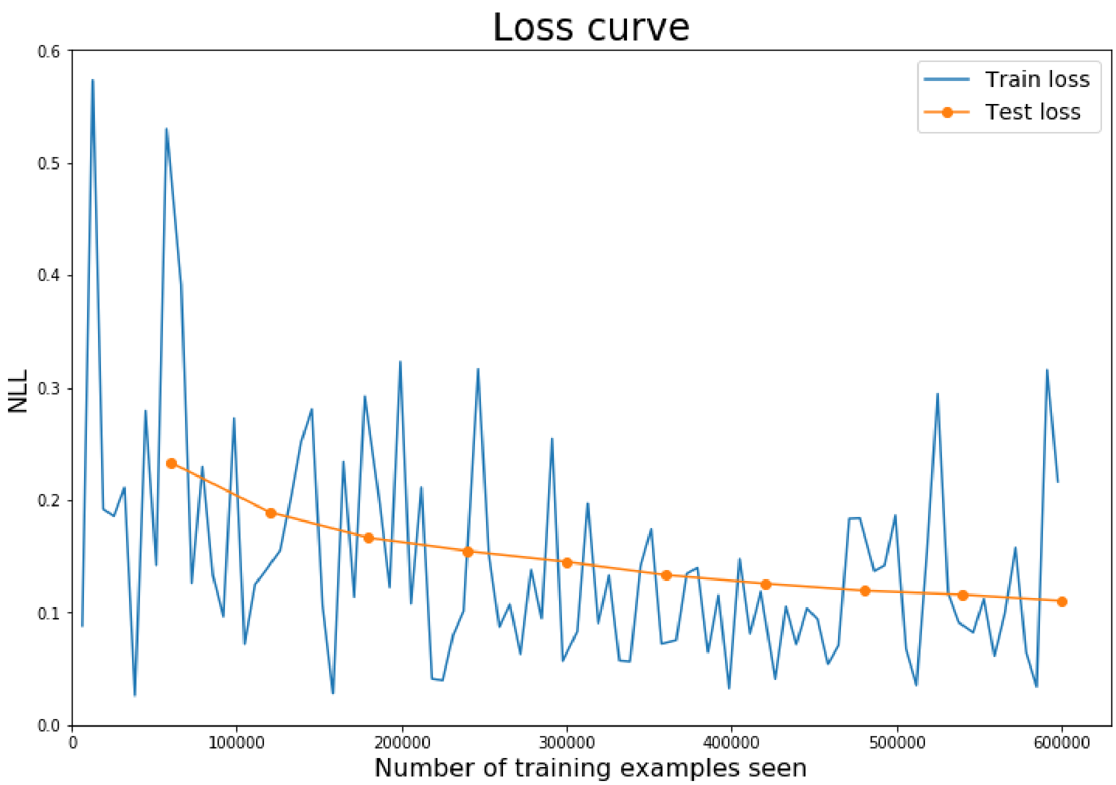
```
Epochs:
100%|                                        |
10/10 [02:38<00:00, 15.80s/it]
```

```
Test accuracy: 0.968
```

The accuracy of the model has however improved from 0.93 to 0.96.

[30]:
```
# 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)
```

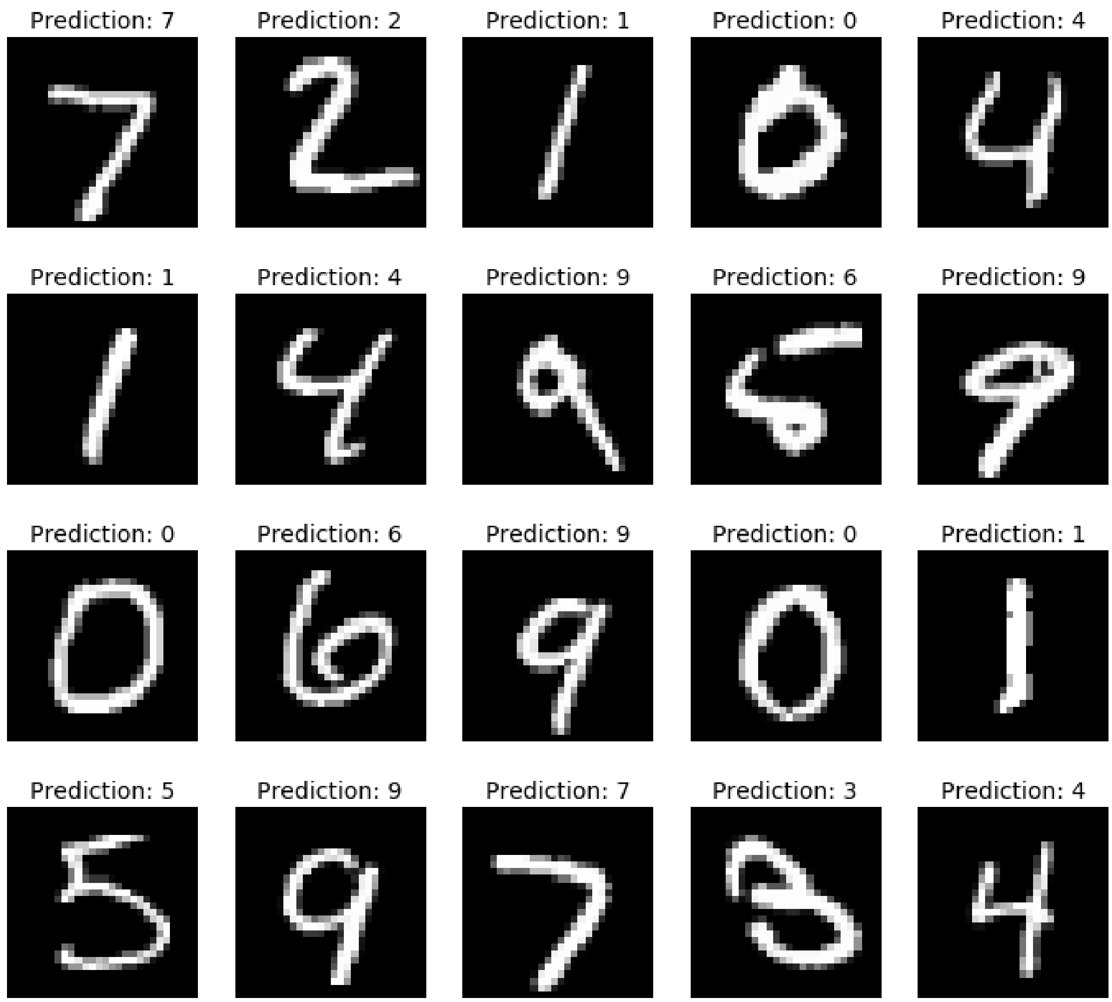
[30]:
```
<matplotlib.legend.Legend at 0x2ddc66a0d08>
```



The curve depicts that the train loss is fluctuating with samples but the test loss is decreasing with increase in samples or examples. The pattern is quite similar to what we obtained before normalizing our data but the numbers and depth of curves vary slightly. Say the test loss is going below *0.2* in the above plot whereas before normalizing it was between *0.3 to 0.5*

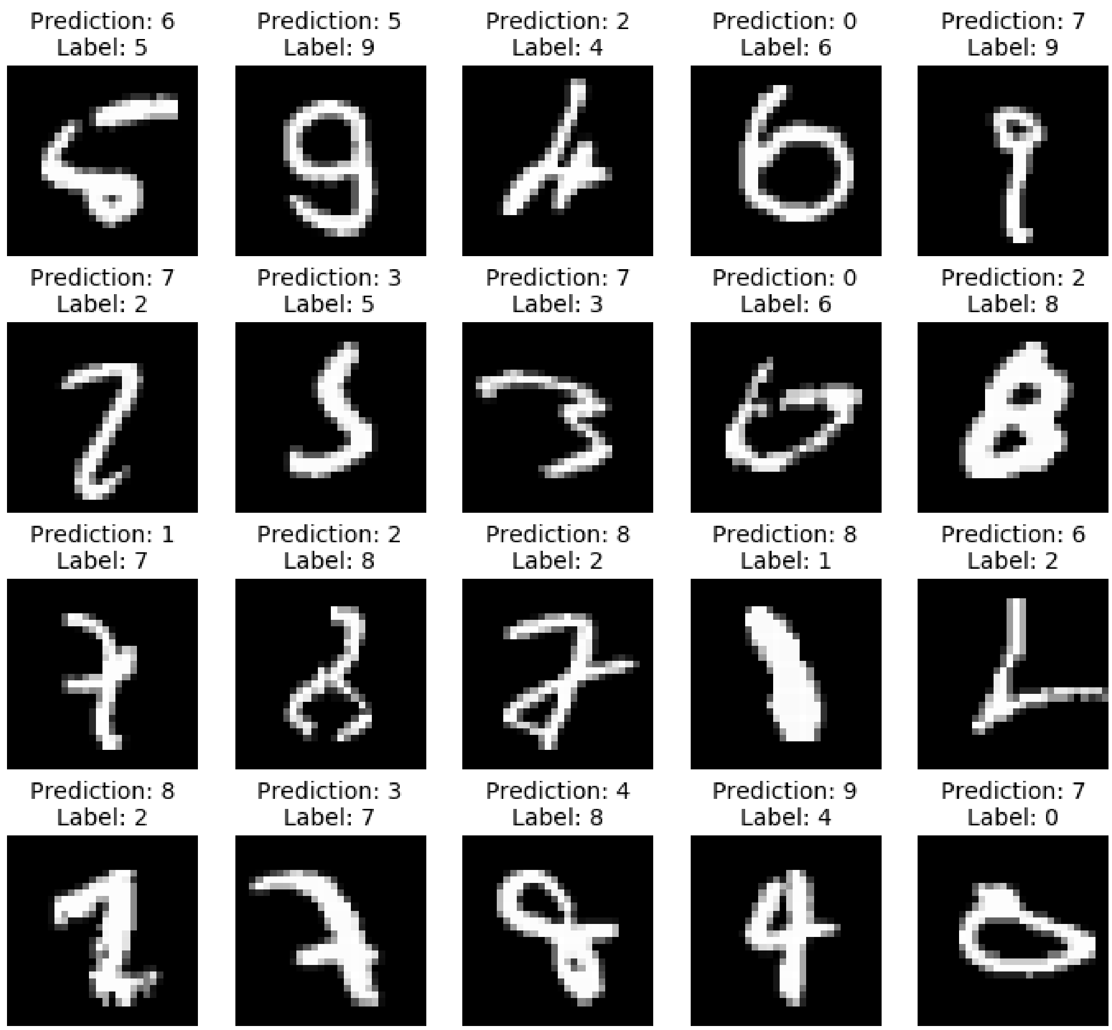
[31]:
```
# 2. Show the predictions of the first 20 images of the test dataset
images, labels = iter(test_loader).next()
images, labels = images.to(DEVICE), labels.to(DEVICE)

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

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

# ------------------
# Write your implementation here. Use the code provided in Part 0 to visualize␣
↪the images.

for i in range(20):
    plt.subplot(4,5,i+1)
    plt.imshow(images[i].squeeze().cpu().numpy(), cmap='gray',␣
↪interpolation='none')
```

```
    plt.title(f'Prediction: {pred[i]}',fontsize=14)
    plt.axis('off')

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



[32]:
```
# 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)
```

```python
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 incorrect_imgs.size(0) < 20:
        images, labels = it.next()
        images, labels = images.to(DEVICE), labels.to(DEVICE)

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

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

        # Compare prediction and true labels and append the incorrect␣
↪predictions
        # using `torch.cat`.
        for index, i in enumerate(output):
            if pred[index] !=labels[index]:
                incorrect_imgs = torch.cat([incorrect_imgs,␣
↪images[index]],dim=0)
                incorrect_preds.append(pred[index])
                incorrect_labels.append(labels[index])

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

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



In terms of Predictions, the model has improved after normalizing the data. The incorrect predictions above are genuinely hard for humans too recognize at times. *Prediction: 6, Label:2* is hard to predict and there are chances for predicting it as *4* or *6*. Similarly all the above incorrect predictions are genuine and the model performance ahs improved considerally after data normalization.