Machine Learning Engineer Nanodegree

Capstone Project

Definition

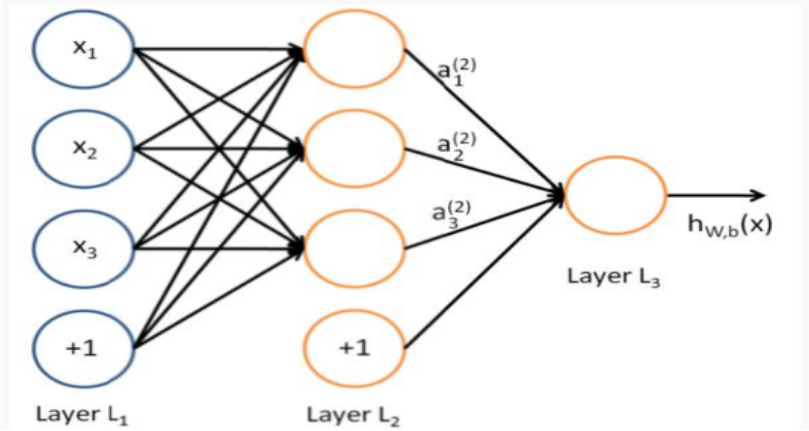
Project Overview

Handwritten Digits Classification Using Neural Networks

Automatic handwriting recognition is of academic and commercial interest. Current algorithms are already pretty good at learning to recognize handwritten digits. Post offices use them to sort letters; banks use them to read personal checks. MNIST [4] is the most widely used benchmark for isolated handwritten digit recognition. More than a decade ago, artificial neural networks called Multilayer Perceptrons

A Multilayer Perceptron (MLP) is a feedforward artificial neural network model that maps sets of input data onto a set of appropriate outputs. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training the network. MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable.

A neural network is put together by hooking together many of our simple “neurons,” so that the output of a neuron can be the input of another.

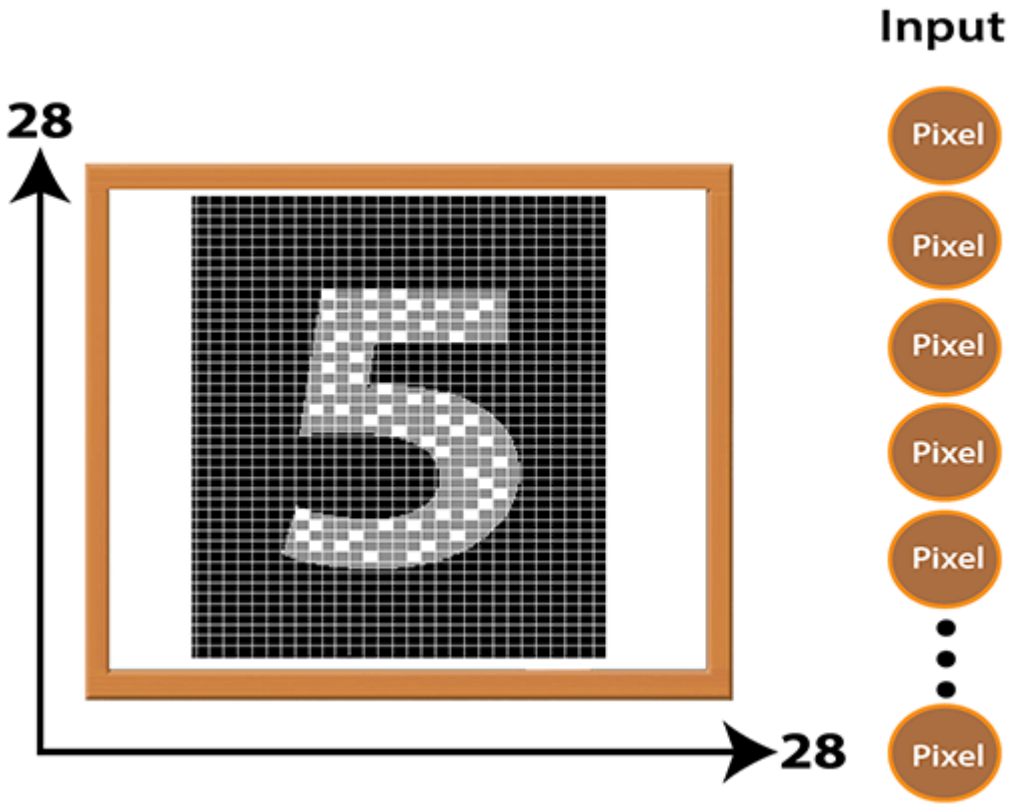


Dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not Ill-suited for machine learning experiments. Furthermore, the black and white images from NIST Ire normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

The digits in the MNIST dataset look like this:

The MNIST dataset is a multiclass dataset which consists of 10 classes into which I can classify numbers from 0 to 9.In MNIST dataset, a single data point comes in the form of an image. These images, contained in MNIST datasets, are typically 28*28 pixels such that 28 pixels traversing the horizontal axis and 28 pixels traversing the vertical axis. It means that a single image from the MNIST database has a total of 784 pixels which must be analyzed. There are 784 nodes in the input layer of my neural network to analyze one of these images.



Preprocessing Data :

Before downloading the data, let me define what are the transformations I want to perform on the data before feeding it into the pipeline. In other words, I can consider it to be some kind of custom edit to are performing to the images so that all the images have the same dimensions and properties. I do it using torchvision.transforms.

```
# convert data to torch.FloatTensor
transform = transforms.ToTensor()
```

transforms.ToTensor() — converts the image into numbers, that are understandable by the system. It separates the image into three color channels (separate images): red, green & blue. Then it converts the pixels of each image to the brightness of their color betIen 0 and 255. These values are then scaled down to a range betIen 0 and 1. The image is now a Torch Tensor.

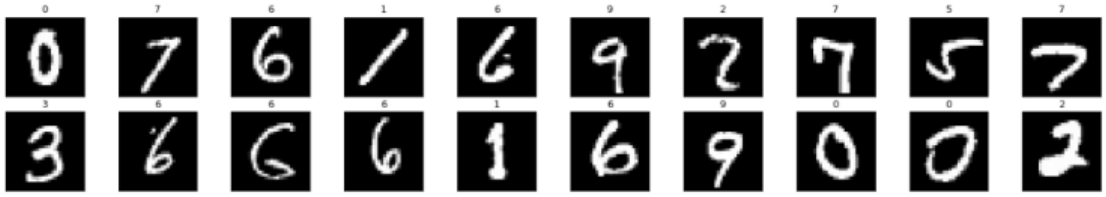
Visualize a Batch of Training Data

The first step in a classification task is to take a look at the data, make sure it is loaded in correctly, then make any initial observations about patterns in that data.

In [3]:

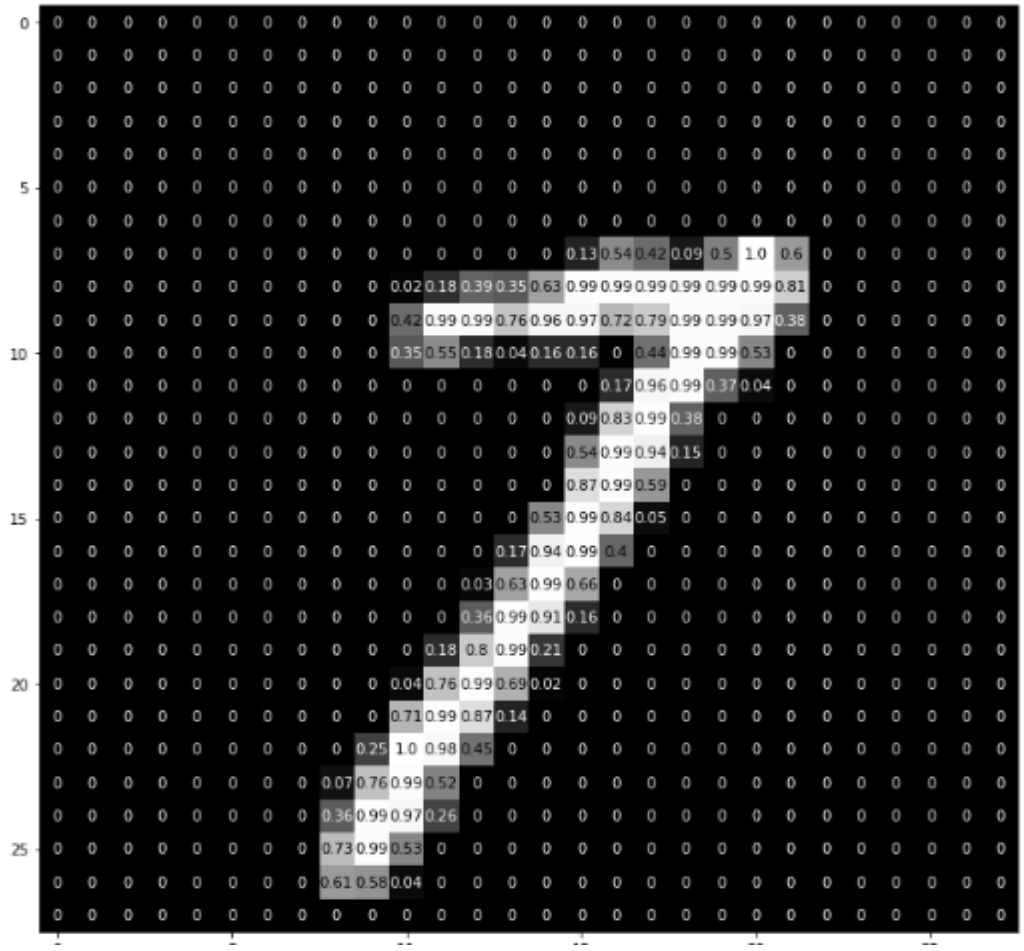
```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    # print out the correct label for each image
    # .item() gets the value contained in a Tensor
    ax.set_title(str(labels[idx].item()))
```



View an Image in More Detail

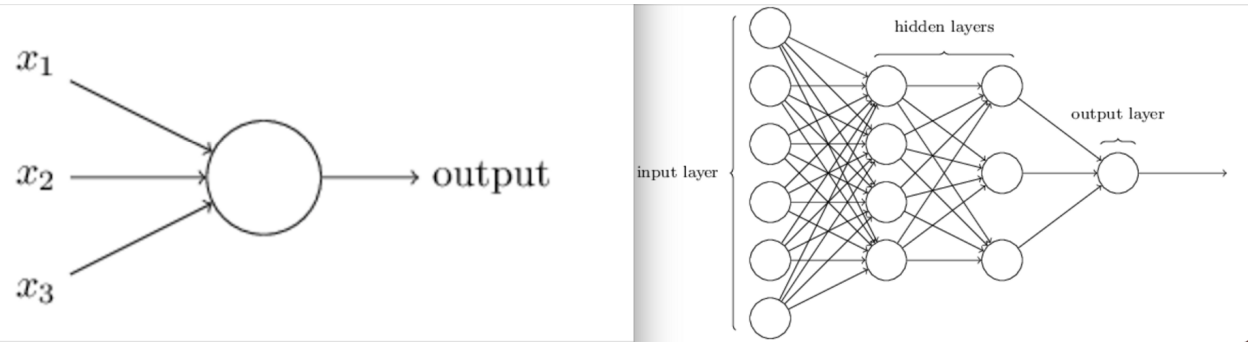
```
img = np.squeeze(images[1])
fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
ax.imshow(img, cmap='gray')
width, height = img.shape
thresh = img.max()/2.5
for x in range(width):
    for y in range(height):
        val = round(img[x][y],2) if img[x][y] !=0 else 0
        ax.annotate(str(val), xy=(y,x),
                    horizontalalignment='center',
                    verticalalignment='center',
                    color='white' if img[x][y]<thresh else 'black')
```



Algorithm and Techniques

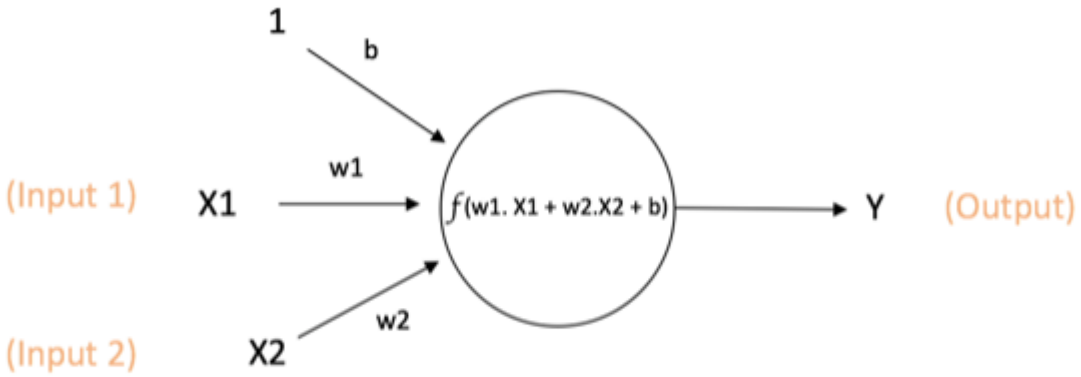
In this assignment I implemented a Multilayer Perceptron Neural Network and evaluated its performance in classifying handwritten digits.

The MLP is essentially a combination of layers of perceptrons. It uses the outputs of the first layer as inputs of the next layer until finally after a particular number of layers, it reaches the output layer. The layers in betIen the input and output layers are called hidden layers.



I will use Multi-Layer Perceptron (MLP) as my neural network model with 784 input neurons. Two hidden layers are used with 512 neurons in hidden layer 1 and 512 neurons in hidden layer 2, folloId by a fully connected layer of 10 neurons for taking the probabilities of all the class labels .

The first part of creating a MLP is developing the feedforward algorithm. Feedforward is essentially the process used to turn the input into an output. Hover, it is not as simple as in the perceptron, but now needs to iterated over the various number of layers. Using matrix operations, this is done with relative ease in python



Output of neuron = Y= f(w1. X1 + w2.X2 + b)

ReLU is used as the activation function for hidden layers and softmax is used as the activation function for output layer, I will use the Softmax activation function in the output layer rather than the sigmoid function. The sigmoid activation function is quite useful for classifying binary datasets, and it was quite effective in arranging probability values betlen 0 and 1. The sigmoid function is not effective for multiclass datasets, and for this purpose, I use the Softmax activation function

Technically, a LogSoftmax function is the logarithm of a **Softmax** function as the name says and it looks like this, as shown below.

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Dropout

Deep learning neural networks are likely to quickly overfit a training dataset with few examples . Ensembles of neural networks with different model configurations are known to reduce overfitting, but require the additional computational expense of training and maintaining multiple models . A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training. This is called dropout

the Network Architecture

```
Net(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

To train a neural network model, I must define a loss function in order to measure the difference betIen my model predictions and the label that I want to predict, I define the **negative log-likelihood loss**. It is useful to train a classification problem with C classes. Together the **LogSoftmax()** and **NLLLoss()** acts as the cross-entropy loss as shown in the network Next, I define the negative log-likelihood loss. It is useful to train a classification problem with C classes. Together the LogSoftmax() and NLLLoss() acts as the cross-entropy loss

Then I will use backpropagation. The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model’s parameters (weights and biases) .In other words, backpropagation aims to minimize the cost function by adjusting network’s weights and biases. The level of adjustment is

determined by the gradients of the cost function with respect to those parameters.

To update parameters, I will use SGD optimizer. It has achieved good accuracy in the model

Implementing:

import libraries

```
# import libraries
import torch
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Dataset split ratio

Some models need substantial data to train upon, so in this case I would optimize for the larger training sets. Models with very few hyperparameters will be easy to validate and tune, so I can probably reduce the size of my validation set, but if my model has many hyperparameters, I would want to have a large validation set as Ill(although you should also consider cross validation). Also, if I happen to have a model with no hyperparameters or ones that cannot be easily tuned, you probably don’t need a validation set too!

I have a dataset that contains both the training and Test set. I am aware that I can use the SubsetRandomSampler to split the dataset into the training and validation subsets

Load and Visualize the Data

Downloading may take a few moments, and I should see my progress as the data is loading. I may also choose to change the batch_size if I want to load more data at a time.

This cell will create DataLoaders for each of my datasets.

```
: from torchvision import datasets
  import torchvision.transforms as transforms
  from torch.utils.data.sampler import SubsetRandomSampler

  # number of subprocesses
  num_workers = 0
  # how many  batch size to load
  batch_size = 20
  # percentage of training set to use as validation
  valid_size = 0.25

  # convert data to FloatTensor
  transform = transforms.ToTensor()

  # choose the training and test datasets
  train_data = datasets.MNIST(root='data', train=True,
                              download=True, transform=transform)
  test_data = datasets.MNIST(root='data', train=False,
                             download=True, transform=transform)

  # obtain training indices that will be used for validation
  num_train = len(train_data)
  indices = list(range(num_train))
  np.random.shuffle(indices)
  split_num = int(np.floor(valid_size * num_train))
  train_idx, valid_idx = indices[split_num:], indices[:split_num]

  # define samplers for each training and validation batches
  train_sampler = SubsetRandomSampler(train_idx)
  valid_sampler = SubsetRandomSampler(valid_idx)


  # prepare data loaders
  train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
      sampler=train_sampler, num_workers=num_workers)
  valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
      sampler=valid_sampler, num_workers=num_workers)
  test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
      num_workers=num_workers)
```

Model Architecture – MLP :

In this notebook, I will train an MLP to classify images from the MNIST database hand-written digit database.

I will start with constructing a Multilayer Perceptron (MLP) Neural Network using Pytorch. Starting with a class Net(nn.Module) so I can build the model layer by layer. I will begin with a simple model architecture, consisting of fmy layers, an input layer, a two hidden layers and an output layer. layer type that is used in many cases for neural networks.

The first layer will receive the input shape(-1,28*28).

The first two layers will have (784,512) nodes. The activation function I will be using for my first 2layers is the ReLU, or Rectified

Linear Activation. This activation function has been proven to work Ill in neural networks.

The two third layers will have (512,512) nodes. The activation function I will be using is the ReLU, or Rectified Linear Activation.

I will also apply a Dropout value of 20% on my first two layers and my two third layers . This will randomly exclude nodes from each update cycle which in turn results in a network that is capable of better generalisation and is less likely to overfit the training data.my output layer will have 10 nodes (num_labels) which matches the number of possible classifi-ations. The activation is for my output layer is softmax. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities. The model will then make its prediction based on which option has the highest probability.

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

# define the Neural Network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # number of hidden nodes in each layer (512)
        hidden_1 = 512
        hidden_2 = 512
        # linear layer (784 -> hidden_1)
        self.fc1 = nn.Linear(28 * 28, hidden_1)
        # linear layer (n_hidden -> hidden_2)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        # linear layer (n_hidden -> 10)
        self.fc3 = nn.Linear(hidden_2, 10)
        # dropout layer (p=0.2)
        drop_num=0.2
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(drop_num)

    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        x=F.log_softmax(x)
        return x

# initialize the NN
model = Net()
print(model)
```

```
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

Specify Loss Function and Optimizer

Loss function - It's recommended to use negative log-likelihood loss for classification. If I look, I can see that negative log-likelihood loss function applies a softmax funtion to the output layer and then calculates the log loss.

Optimizer - here I will use SGD which is a generally good optimizer for many use cases.

```
: # specify loss function (that negative log-likelihood loss function )
criterion = nn.NLLLoss()
# specify optimizer (SGD optimizer) and learning rate = 0.05
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
```

Train the Network

Here I train the model.

I will start with 35 epochs which is the number of times the model will cycle through the data.

The model will improve on each cycle until it reaches a certain point.

I will also start with a low batch size, as having a large batch size can reduce the generalization ability of the model.

The steps for training/learning from a batch of data are described in the comments below:

1- Clear the gradients of all optimized variables

2- Forward pass: compute predicted outputs by passing inputs to the model

3- Calculate the loss

4- Backward pass: compute gradient of the loss with respect to model parameters

5- Perform a single optimization step (parameter update)

6- Update average training loss

The following loop trains for 35 epochs; take a look at how the values for the training loss decrease over time. I want it to decrease while also avoiding overfitting the training data.

```python
# number of epochs
n_epochs=35
train_losses=[]
valid_losses=[]
class_correct_valid = list(0. for i in range(10))
class_total_valid = list(0. for i in range(10))
class_correct_train = list(0. for i in range(10))
class_total_train = list(0. for i in range(10))

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

# check if CUDA is available
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')
    for epoch in range(n_epochs):
    # monitor training loss
    ###################
        train_loss=0

    # train the model #
    ###################
        model.train() # prep model for training
        for data, target_train in train_loader:
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target_train)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update running training loss
            train_loss += loss.item()
        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct = np.squeeze(pred.eq(target_train.data.view_as(pred)))
        # calculate training accuracy for each object class
        for i in range(len(target_train)):
            label = target_train.data[i]
            class_correct_train[label] += correct[i].item()
            class_total_train[label] += 1
```

Here I will review the accuracy of the model on the training data sets.

```
Epoch: 35       Training Loss: 0.009401

Train Accuracy (Overall):   98% ( 689/ 700)
```

Metrics

I will use the accuracy metric which will allow me to view the accuracy score on the validation data when I train the model.

```
    ######################
    # validate the model #
    ######################
        model.eval() # prep model for evaluation
        valid_loss=0
        for data, target_valid in valid_loader:
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target_valid)
            # update running validation loss
            valid_loss += loss.item()

        # print training/validation statistics
        # calculate average loss over an epoch
        train_loss = train_loss/len(train_loader)
        valid_loss = valid_loss/len(valid_loader)

        train_losses.append(train_loss)
        valid_losses.append(valid_loss)

        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct = np.squeeze(pred.eq(target_valid.data.view_as(pred)))
        # calculate validation accuracy for each object class
        for i in range(len(target_valid)):
            label = target_valid.data[i]
            class_correct_valid[label] += correct[i].item()
            class_total_valid[label] += 1


        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch+1,
            train_loss,
            valid_loss
            ))

        # save model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
            valid_loss_min,
            valid_loss))
            torch.save(model.state_dict(), 'model.pt')
            valid_loss_min = valid_loss
        print('\nTrain Accuracy (Overall): %4d%% (%4d/%4d)' % (
        100. * np.sum(class_correct_train) / np.sum(class_total_train),
        np.sum(class_correct_train), np.sum(class_total_train)))

        print('\nValidation Accuracy (Overall): %4d%% (%4d/%4d)' % (
        100. * np.sum(class_correct_valid) / np.sum(class_total_valid),
        np.sum(class_correct_valid), np.sum(class_total_valid)))
```

Here I will review the accuracy of the model on validaition data sets.

```
Epoch: 35       Training Loss: 0.009401         Validation Loss: 0.071400
```

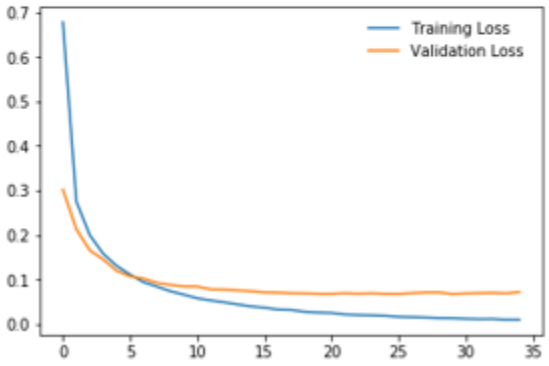
```
Validation Accuracy (Overall):   97% ( 684/ 700)
```

Here I will drow the loss function of the model on both training datasets and validaition data sets .

```
In [8]: %matplotlib inline

        plt.plot(train_losses,label="Training Loss")
        plt.plot(valid_losses,label="Validation Loss ")
        plt.legend(frameon=False)
```

```
Out[8]: <matplotlib.legend.Legend at 0x1d27bbb8e88>
```



Results :

Finally, I test my model on previously unseen test data and evaluate it's performance. Testing on unseen data is a good way to check that my model generalizes Ill. It may also be useful to be granular in this analysis and take a look at how this model performs on each class as Ill as looking at its overall loss and accuracy

The following observations are based on the results of the test:

• The classifier performs Ill with new data.

• Misclassification does occur but seems to be betIen classes that are relatively similar such as Drilling and Jackhammer.

```
# initialize lists to monitor test loss and accuracy
test_loss =0

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model.eval() # prep model for evaluation

for data, target in test_loader:
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct = np.squeeze(pred.eq(target.data.view_as(pred)))
    # calculate test accuracy for each object class
    for i in range(len(target)):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# calculate and print avg test loss
test_loss = test_loss/len(test_loader.sampler)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            str(i), 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

Accuracy

```
Test Loss: 0.068537

Test Accuracy of     0: 99% (973/980)
Test Accuracy of     1: 98% (1123/1135)
Test Accuracy of     2: 98% (1012/1032)
Test Accuracy of     3: 98% (993/1010)
Test Accuracy of     4: 97% (954/982)
Test Accuracy of     5: 97% (867/892)
Test Accuracy of     6: 98% (943/958)
Test Accuracy of     7: 97% (999/1028)
Test Accuracy of     8: 96% (944/974)
Test Accuracy of     9: 98% (990/1009)

Test Accuracy (Overall): 97% (9798/10000)
```

Visualize Sample Test Results

This cell displays test images and their labels in this format: predicted (ground-truth). The text will be green for accurately classified examples and red for incorrect predictions.

```
:
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds = torch.max(output, 1)
# prep images for display
images = images.numpy()

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].item())),
                 color=("green" if preds[idx]==labels[idx] else "red"))
```

```
C:\Users\lenovo\Anaconda3\lib\site-packages\ipykernel_launcher.py:35: UserWarning: Implicit dimension choice for log_softmax ha
s been deprecated. Change the call to include dim=X as an argument.
```

Conculison

In this work, I conduct an experiment implementing Back-propagation Neural Network to achieve the classification of the MNIST handwritten digit database. In the experimental model, 28 * 28 = 784 pixels are regarded as input and 10 different classes of digits from 0 to 9 as output. Besides, I use classification accuracy and loss plot to determine the performance of the neural network. After testing on various parameters of the model, I set the system parameters as follow: number_of_epochs = 35, learning rate = 0.05 and batch_size = 20 The experimental results show that to a certain extent, the back-propagation neural network can be used to solve classification problem in the real world.

Besides, the neural network (the multilayer perceptron and back propagation) is proposed for the classification of the standard base MNIST isolated digit. An extraction technique is used in the phase of extraction of characteristics before implementing the classification of the digits. The recognition rate is 97.00% with a Test database. The method of extraction shows enough good results.

References:

Deep Learning Book” by Goodfellow, Bengio, and Courville.

http://neuralnetworksanddeeplearning.com/

https://en.wikipedia.org/wiki/Multilayer_perceptron

https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7