Machine Learning II
George Washington University
Neeraj Walia, Zhen Zhang
Final Project Report
April 24, 2018

Identify Hand Written Digits

Introduction

In this project, we build a model to correctly identify hand-written digits. We use MNIST dataset as our data. Our model will also be able to identify digits from zero to nine in any handwriting. We chose this problem because of its many real-world applications like toy touchpad that can identify writing digits. We did not spend too much time to do the data preprocessing since the dataset is already cleaned. In this project, we use Pytorch to build deep learning model. Moreover, we did a comparison of these two networks. In this report, we will introduce our data, make a description of the deep learning network and training algorithm that we use, introduce the experimental setup as well as describe the results of your experiments.

Description of the data set

The MNIST database of handwritten digit from "0" to "9" available on kaggle website. All of the digits written by Census Bureau employees and high school students. This data set has 60,000 patterns for training sets and 10,000 patterns for the test set. The size of each image is 28 pixels by 28 pixels. Usually, it be used to judge the accuracy of deep learning. Below is an example to show the digit of MNIST.

Description of the deep learning network and training algorithm

In this project, we built two neural networks, one is multilayer neural network, another is convolution neural network. The convolution neural network can help us extract feature. We use the Mini_ Batch Stochastic Gradient Descent which is a based training algorithm as our training algorithm to help us update the gradient of the parameters in the iterative manner, timprove the speed of training. We also use Batch normalization to deal with the data in the hidden layers.

Experimental setup

- Multilayered Perceptron Network

Multilayered perceptron Network (MLP) is built using two layers network. We initially discussed about adding two to four more hidden layers and tested the concept but the performance did not improve for the model and two layer network was providing best of the performance and accuracy.

Datasets are loaded as trainset and testset from torchvision.datasets.MNIST. Data is transformed using transforms.ToTensor() to the common image.

```
torchvision.datasets.MNIST(root='./mnist', train=True, download=True, transform=transforms.ToTensor())
trainset = torchvision.datasets.MNIST(root='./mnist', train=True, download=True, transform=transforms.ToTensor())
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)
testset = torchvision.datasets.MNIST(root='./mnist', train=False, download=True, transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE, shuffle=True)
```

Multilayered perceptron class is defined with two functions. Initial function defines the two input layers with linear functions and parameters passing to it. We added tanh() function going forward after the first layer to element wise classify the data in groups. For these kind of classification problem with classes LogSoftmax() function and NLLLoss function is used for element-wise classification.

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, classes_num):
        super(MLP, self).__init__()
        self.input_layer = nn.Linear(input_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, classes_num)
        self.tanh = nn.Tanh()
        self.reg_layer = nn.LogSoftmax()
    def forward(self, x):
        out = self.input_layer(x)
        out = self.tanh(out)
        out = self.output_layer(out)
        out = self.reg_layer(out)
        return out
```

Input parameters are passes in the first layer and combined with weight and bias and transformed using the linear function. Negative log likelihood loss is used to train the data for classification. Stochastic Gradient descent SGD optimizer is used with learning rate and momentum and input parameters.

To train the dataset, data is read from train dataset and it went through the look of 50 epoch cycles and following steps are performed for training. The data set contains image and labels. Labels are the correct reading of the digits which we will use to train the data. Zero_grad() function is used to clear gradient of all optimization. Loss is calculated after the first pass. Backward propagation is initiated and weights and bias are updated accordingly. The cycle starts again with new weights and goes through the forward pass, calculate the loss and update the new weights and biases.

The trained MLP model is run thru the test data. And the output is predicted. Based on the predicted output, accuracy rate is calculated by predicted output divided by length of test dataset.

- Convolution Neural Network

  Before we begin to build the convolution neural network, we loaded the trainset and testset from torchvision.datasets.MNIST. Then, we also transformed the data by using 'transforms.ToTensor()'. In the convolution neural network, we built two convolutional layers and apply the activation function which is RELU to the output. The convolution layers, pooling layers, and fully connected layer is normal for a convolution neural network. Beside, we add another layer which is Batch normalization layer to normalize the feature of input,

active value, and also keep the same mean and standard deviation for each training process. In this way, the learning efficiency for our network will increase. The pooling algorithm that we used is max pooling which helps us reduce parameter, filter the important information of our layers, lessen calculative burden, and improve the accuracy of our model.

For the training, we defined the loss function which uses to calculate the differences between the predicted value and true value at first. Since our problem is a multiclass classification problem, we used cross entropy as our loss metric. Then, we use the Stochastic Gradient Descent(SGD) as our optimization algorithm to optimize the loss when we train our model. We used SGD methods as optimizer since we want to prevent gradient disappear. We used the loop to over our iterator. In the loop, we get the inputs at first and then change them to variable to prepare for calculating the gradient. Then, we use the 'optimizer.zero_grad()' function to make the gradient to 0 to prevent gradient to increase when we do the backpropagation. Then, we follow the step to do the forward, backward and optimize to update our weight. The updated weight equal to weight subtract learning rate multiply by gradient. Finally, we print the loss results.

After training completed, we begin to test our network to see whether our network has learnt something or not, and the performance of our network on the whole dataset. We choose the evaluation mode when we test our data to turn the Batchnorm off. And our convolution neural network gets 99% accuracy finally. Also, when we test our model, we compare the 'prediction' digit with 'real' digit. And got the result as follows:

When we design our algorithm, we used mini-batch. Since we choose the Stochastic Gradient Descent, we cannot use a big batch size. Therefore, we choose the 200 as our mini_batch size which is small with respect to our dataset. For the learning rate, we choose 0.0001, 0.001, 0.01, and 0.1 to record the loss. When we test the learning rate, we just set the Epoch size to 1 in order to save time.

Initially, we do not consider the problem about overfitting. However, our neural network gets 99% accuracy. And we believed the result is overfitting. Therefore, we try to modify our neural network by using dropout which lets our network randomly delete some nodes of hidden layers in the training process. But the accuracy of our network is still 99%. Thus, we believe our network is not overfitting.

Results

- Multilayered Perceptron Network

  Following parameters lead to the accuracy rate of 98%

  HIDDEN_SIZE = 200

  CLASSES_NUM = 10

  BATCH_SIZE = 200

  IMAGE_DIMEN = 1

IMAGE_SIZE = 28

LEARNING_RATE = 0.01

EPOCH_SIZE = 50

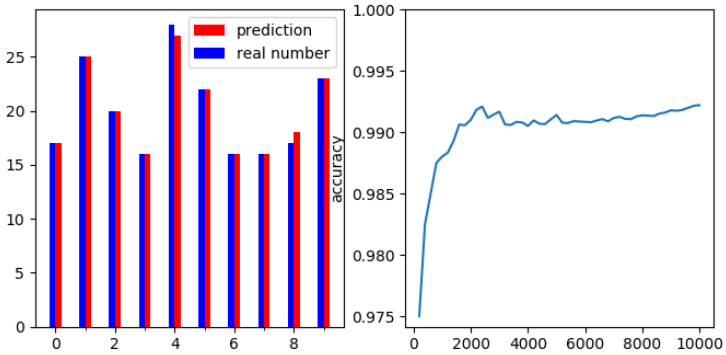
- Convolution Neural Network

  Below is a table to show the loss for varying learning rate. The first column is the difference of learning rate.

	[1,50]	[1,100]	[1,150]	[1,200]	[1,250]	[1,300]
0.0001	2.235	2.049	1.871	1.699	1.542	1.399
0.001	1.733	0.772	0.462	0.346	0.293	0.253
0.01	0.695	0.170	0.125	0.108	0.096	0.092
0.1	0.459	0.125	0.088	0.075	0.069	0.072

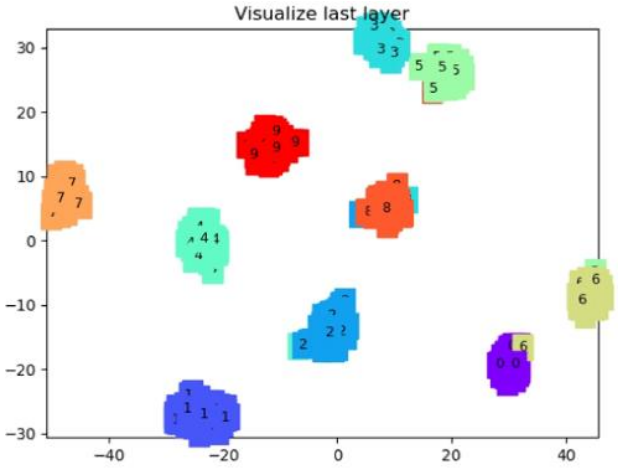
From the table, we can easily find the loss is small when the learning rate equal to 0.1, so we set 0.1 as our final learning rate.

We used the matplotlib module to show the training process, and got the chart as follows:



From the right-side chart, it can easily find almost every predict number match to the real number as the training number. Besides, from the left-side chart, it can see the accuracy is increasing as the training data increasing, and the accuracy above 99% finally.

In the testing time, we got average accuracy 99%. Also, to visualize the last layer of our convolution neural network, except using matplotlib I also used sklearn library. Moreover, to prepare the last layer visualize, I used ‘T-SNE’ to reduce dimensionality of our data. The visualization of last layer as follows:



The visualization clearly shows that almost every digit goes to their own group which also means the performance of our neural network is excellent.

Summary and conclusions

The accuracy of MLP network is rated as 98% and of the convolution network at 99%. This shows that both networks are highly accurate in reading the hand-written digit and with further tweaking they can perform at almost 100% accuracy. So according to our testing both networks are suitable for this project of reading hand written digits.