DAT565 Assignment 7 – Group 79

Student 1 - (Yuchuan Dong 10hrs)
Student 2 - (Ziyuan Wang 10hrs)
Student 3 - (Meixi Lin 10hrs)

October 14, 2023

Problem 1

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

y_train = keras.utils.np_utils.to_categorical(lbl_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(lbl_test, num_classes)
```

Listing 1: Preprocessing code in the notebook

The first two lines change the type of train and test data, transforming from integer type to float type. Because, in the following procedure, the calculation is in float type. Then the following two lines make all the values of pixels range from $[0, 1]$, one of the reasons is that large numerical calculation is complex in computer and it is better to compute easily after normalizing the value. The last two lines are used for converting each class vector (integers) to binary class matrix, which is also called one-hot encoding.

Problem 2

Before starting the Problem 2, we use Pytorch library rather than Keras, because we are more familiar with this library. We analysed the source code given in the Jupyter Notebook and reworked it with Pytorch Library.

The structure of the neural network in Pytorch is shown below:

```python
class dense_model(nn.Module):
    def __init__(self):
        super().__init__()
        self.sequential = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.sequential(x)
        return x
```

Listing 2: Implementation of neural network in Pytorch

Problem 2.1

It is clear that there is one input layer including the process of flattening, two hidden layers and one output layer in total.

Before images enter into the neural network, we first flatten them. So, there will be 784 neurons in the input layer. For two hidden layers, there are both 64 neurons for each.

The activation function is RELU,

$$Relu(x) = max(0, x)$$

which means that values less than zero from certain neuron will become zero, which deactivates this neuron and makes the whole neural network sparse. Sparsity is good property for artificial neural network. The reason is that sparsity reduces the computation in the dense layer and keeps potentially important nodes and discards less important ones. Besides that, RELU solves the gradient vanishment and gradient explosion. Because the gradient of the output from the neuron with RELU activation function is either 0 or 1, which helps us better train the model.

The number of neurons in each layer is shown in the table,

Layer	Output Shape	Number of Parameters
Flatten-1	$batchsize \times 784$	0
Linear-2	$batchsize \times 64$	50240
Relu-3	$batchsize \times 64$	0
Linear-4	$batchsize \times 64$	4160
Relu-5	$batchsize \times 64$	0
Linear-6	$batchsize \times 10$	650

Table 1: Structure of the neural network

There are 55050 trainable parameters in total.

As for the input layer, we transform the image with size of 28×28 into a long vector with the length of 784. So for the input layer, the input size is $batchsize \times 784$.

As for the output layer, the output of the model is the probability of ten classifications. So there will be ten neurons in the output layer.

Problem 2.2

Cross entropy is a common loss function in the classification task. Suppose there are C classes in total.

The form of this loss function is like this,

$$L_i = -log\frac{exp(x_{n,y_n})}{\sum_{c=1}^{C} exp_{x_{n,c}}}$$

$$Loss = \frac{\sum_{i=1}^{batchsize} L_i}{n}$$

where x is from the output layer, with shape of $batchsize \times classes$ and y_n corresponds to its label. L_i is the loss for each output, processed by

$$softmax : x_c = \frac{x_c}{\sum_{c=1}^{C} exp_{x_c}}$$

and

$$-log : x_c = -log(x_c)$$

Then whole process is equivalent to use *torch.nn.LogSoftmax*.

Then sum up all the losses and the average mean of the sum is the entropy loss.

As for the interpretation of this loss function, suppose we have two probability distributions P and Q. Distribution P is based on our training set, and distribution Q is predicted by our model or the output of our model. Then in order to measure how one probability distribution P is different from distribution Q, **KL-Divergence** is introduced:

$$D_{KL}(P||Q) = \sum_{x \in X} P(x)log(\frac{P(x)}{Q(x)}) = \sum_{x \in X} P(x)logP(x) - \sum_{x \in X} P(x)logQ(x)$$

For the first term of this equation, it is a constant and indicates the information content of our training set. The second term is called *CrossEntropy*. If we minimize the value cross entropy, the value of KL-Divergence is minimized as well, which means that we try to make the probability distribution P and Q similar. That is we try to make our output from the model follow the same probability distribution of our training data. This loss function is appropriate for our task, because the output of our model is the probability of ten categories and we want to make prediction right based on the model trained by training data. So, combined with the interpretation of cross entropy, we will train our model correctly with this loss function.

Problem 2.3

We trained our model for ten epochs and the training and validation accuracy is showed below:

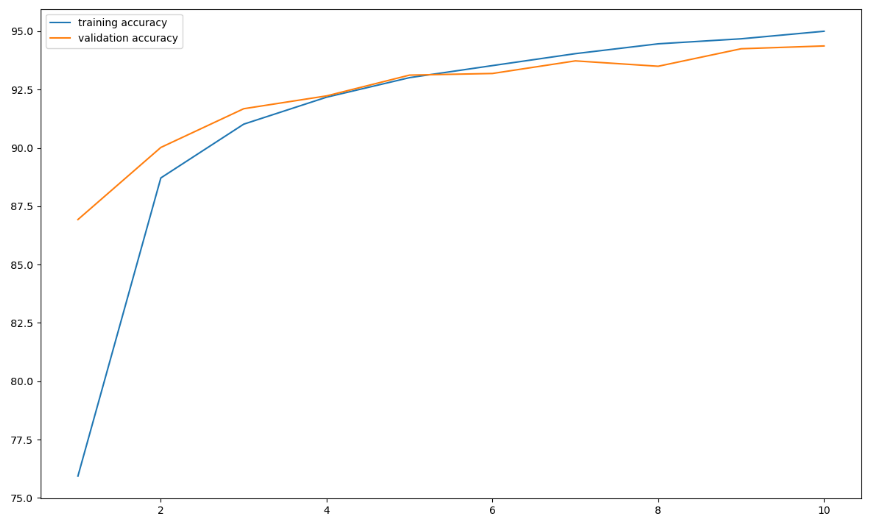

Figure 1: An image of a galaxy

It's worth mentioning that we use the optimizer, SGD to update our parameters. It is noticeable that, both training and validation accuracy increase with epoch increasing. And when it comes to last two epochs, the performance of the model almost stops growing.

Problem 2.4

In this section, we need to change the hidden layers of our model. And when train the model, we have to change the weight decay from 0.001 to 0.000001 and then train the model three times.

```
class dense_model(nn.Module):
    def __init__(self, a = 64, b = 64):
        super().__init__()
        self.sequential = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 500),
            nn.ReLU(),
            nn.Linear(500,300),
            nn.ReLU(),
            nn.Linear(300, 10)
        )
        self.softmax = nn.Softmax()

    def forward(self, x):
        x = self.sequential(x)
        return x
```

Listing 3: Modified model

Weight Decay	First	Second	Third	mean accuracy with std
0.001	82.88	82.87	82.63	82.79± 0.11
0.0001	82.59	83.95	83.54	83.36 ± 0.57
0.00001	82.76	82.77	83.31	82.95 ± 0.25
0.000001	82.27	83.56	83.21	83.01 ± 0.54
0.0000001	83.43	83.74	83.66	83.61 ± 0.13

Table 2: Validation accuracy

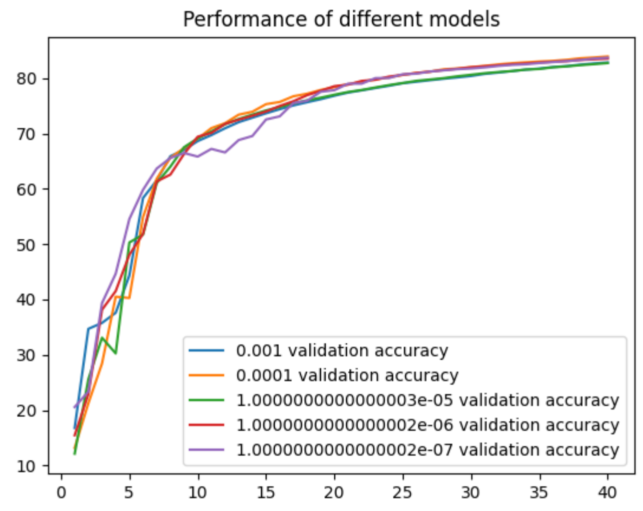

Figure 2: Validation accuracy in the first time training

We can conclude from the figure that when training the model for 40 epochs, models all have accuracy over 80% but there are little differences in accracy between these models.

We do not get the same results as Hinton did. There are few reasons for it. The first one is that we do not know the learning rate when training. We set the learning rate equal to 0.001 in our experiment. Moreover, the optimizer may differ. We use SGD method to update our parameters. There are other methods such as Adam, which are better probably.

Problem 3

In this part, we will use convolutional layers to construct neural network and test its performance on MNIST dataset.

Problem 3.1

We build our convolutional model similar to Lenet,

```
class Model_LeNet(nn.Module):
    def __init__(self, in_channels = 1, num_classes=10):
        super(Model_LeNet, self).__init__()
        self.sequential1 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=4, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=1),
            nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(2, 2), stride=1),
            nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(2, 2), stride=2),
            nn.Flatten()
        )

        self.sequential2 = nn.Sequential(
            nn.Linear(400, 84),
            nn.ReLU(),
            nn.Linear(84, num_classes)
        )

    def forward(self, x):
        x = self.sequential1(x)
        # print(x.shape)
        x = self.sequential2(x)
        return x
```

Listing 4: CNN model

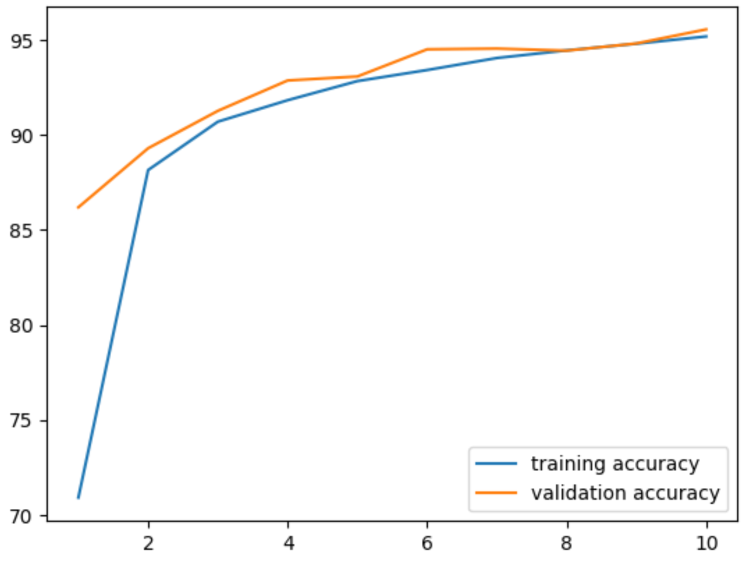

Figure 3: Caption

From what we got above, the accuracy of this model is 95.56%, which is quite close to the performance of model in Problem 2.3.

To introduce the structure of this model, there are three convolutional layers. Convolutional layers can extract adjacent pixel features and have less parameters to train compared with dense layer. Besides convolutional layer, there are also pooling layer, also called sub-sampling layer, helping reduce the spatial dimensions of the images. After the pooling layer, we flatten the image and then send it into the fully-connected layer.

Problem 3.2

There are numerous benefits of convolutional layers. First and foremost, the parameter of the kernel in convolutoinal layer is shared-weight, which means that there are less parameters to train. Secondly, it can extract neighbor features automatically, sliding the kernel from the first pixel to the last pixel. Last but not least, we can increase the channels of the image and decrease the size of image at the same time, which make it possible for the model to learn from high dimensions.

References

[1] Donald E. Knuth (1986) *The TEX Book*, Addison-Wesley Professional.

[2] Leslie Lamport (1994) *LATEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.

Appendix

train.py

```
import torch
import torch.nn as nn
from torchsummary import summary
from torch.utils.data import Dataset, DataLoader
from torchvision.datasets import MNIST
import torchvision.transforms as transform
import os
import sys
from PIL import Image
import matplotlib.pyplot as plt
from letnet import Model_LeNet

os.chdir(sys.path[0])

class dense_model(nn.Module):
    def __init__(self, a = 64, b = 64):
        super().__init__()
        self.sequential = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, a),
            nn.ReLU(),
            nn.Linear(a,b),
            nn.ReLU(),
            nn.Linear(b, 10)
        )
        self.softmax = nn.Softmax()

    def forward(self, x):
        x = self.sequential(x)
        return x

class mnist_dataset(Dataset):
      def __init__(self) -> None:
           super().__init__()
           self.transform = transform.Compose([
               transform.PILToTensor()
           ])
           self.train = MNIST(root = "./data", train=True, download=False, transform=
    self.transform)
```

```
            self.test = MNIST(root = "./data", train=False, download=False, transform=
    self.transform)

# def train_script(model, dataset):



# model = dense_model()
# summary(model, (1, 1, 28, 28))

def show_image(dataset):
    pic = dataset.train[0][0].permute(1,2,0).squeeze().numpy()
    print(pic.shape)
    # print(type(pic))
    pic = Image.fromarray(pic)
    pic.save('./pic.png')


def train_script(model:nn.Module, epochs = 10, learning_rate = 0.001):

    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    dataset = mnist_dataset()
    model = model.to(device)
    train_dataloader = DataLoader(dataset=dataset.train, batch_size=256, shuffle=True)
    test_dataloader = DataLoader(dataset=dataset.test, batch_size=256, shuffle=True)

    criterion = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr = learning_rate, weight_decay=0.001)

    train_acc, val_acc = [], []
    for epoch in range(epochs):
        number = 0
        for x, y in train_dataloader:
            _x, _y = x.float().to(device), y.to(device)
            pred_y = model(_x)
            opt.zero_grad()
            loss = criterion(pred_y, _y)
            loss.backward()
            opt.step()
            pred_y = torch.argmax(pred_y, dim=1)
            number += (pred_y == _y).sum()
            # print(loss)
        training_accuracy = float(number / len(dataset.train) * 100)
        # print(training_accuracy)
        # print(number)
        # print(loss)
        train_acc.append(training_accuracy)
        model.eval()
        number = 0
        with torch.no_grad():
            for x, y in test_dataloader:
                _x, _y = x.float().to(device), y.to(device)
                pred_y = model(_x)
                pred_y = torch.argmax(pred_y, dim=1)
                number += (pred_y == _y).sum()
        val_accuracy = float(number / len(dataset.test) * 100)
        val_acc.append(val_accuracy)
        # break
    x_axis = [i+1 for i in range(epochs)]
    torch.save(model.state_dict(), f'./conv_model_{learning_rate}')
    plt.plot(x_axis, train_acc, label='training accuracy')
    plt.plot(x_axis, val_acc, label='validation accuracy')
    plt.legend()
    plt.show()
```

```
def weight_decay_train(model:nn.Module, epochs = 40, learning_rate = 0.001, decay_item
 = 0.001):
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    dataset = mnist_dataset()
    model = model.to(device)
    train_dataloader = DataLoader(dataset=dataset.train, batch_size=256, shuffle=True)
    test_dataloader = DataLoader(dataset=dataset.test, batch_size=256, shuffle=True)


    criterion = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr = learning_rate, weight_decay =
    decay_item)

    train_acc, val_acc = [], []
    for epoch in range(epochs):
        number = 0
        for x, y in train_dataloader:
            _x, _y = x.float().to(device), y.to(device)
            pred_y = model(_x)
            opt.zero_grad()
            loss = criterion(pred_y, _y)
            loss.backward()
            opt.step()
            pred_y = torch.argmax(pred_y, dim=1)
            number += (pred_y == _y).sum()
            # print(loss)
        training_accuracy = float(number / len(dataset.train) * 100)
        # print(training_accuracy)
        # print(number)
        # print(loss)
        train_acc.append(training_accuracy)
        model.eval()
        number = 0
        with torch.no_grad():
            for x, y in test_dataloader:
                _x, _y = x.float().to(device), y.to(device)
                pred_y = model(_x)
                pred_y = torch.argmax(pred_y, dim=1)
                number += (pred_y == _y).sum()
        val_accuracy = float(number / len(dataset.test) * 100)
        val_acc.append(val_accuracy)
    # break
    x_axis = [i+1 for i in range(epochs)]
    torch.save(model.state_dict(), f'./dense_model_{learning_rate}_{decay_item}')
    plt.plot(x_axis, val_acc, label=f'{decay_item} validation accuracy')


def weight_decay_train_script(model):
    decay = [0.001 * (0.1 ** i) for i in range(5)]
    for decay_item in decay:
        print(f"start training {decay_item}")
        weight_decay_train(model=model, decay_item=decay_item)
    plt.title('Performance of different models')
    plt.legend()
    plt.show()

def validate_performance(model:nn.Module):
    model.load_state_dict(torch.load("./cnn_model_0.001"))
    dataset = mnist_dataset()
    test_dataloader = DataLoader(dataset=dataset.test, batch_size=256, shuffle=True)
    device = "cpu"
    model.eval()
    number = 0
    with torch.no_grad():
        for x, y in test_dataloader:
            _x, _y = x.float().to(device), y.to(device)
            pred_y = model(_x)
            pred_y = torch.argmax(pred_y, dim=1)
            number += (pred_y == _y).sum()
    val_accuracy = float(number / len(dataset.test) * 100)
    print(val_accuracy)

if __name__ == '__main__':

    # model = dense_model(a=500, b=300)
    model = Model_LeNet(1)
    # weight_decay_train(model)
    # summary(model, (1,1,28,28))
    # print(model(torch.rand([256,28,28])))
    # weight_decay_train_script(model)
    validate_performance(model)
    # train_script(model,20)
```

Listing 5: CNN model

letnet.py

```
class Model_LeNet(nn.Module):
    def __init__(self, in_channels = 1, num_classes=10):
        super(Model_LeNet, self).__init__()
        self.sequential1 = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=4, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=1),
            nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(2, 2), stride=1),
            nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=(2, 2), stride=2),
            nn.Flatten()
        )

        self.sequential2 = nn.Sequential(
            nn.Linear(400, 84),
            nn.ReLU(),
            nn.Linear(84, num_classes)
        )

    def forward(self, x):
        x = self.sequential1(x)
        # print(x.shape)
        x = self.sequential2(x)
        return x
```

Listing 6: CNN model