CNN

1. Introduction of CNN

Convolutional neural network (CNN) is a kind of neural network specially used to process data with similar grid structure. Convolutional networks are neural networks that use convolutional operations in place of matrix multiplication in at least one layer of the network.

The basic structure of a CNN usually consists of the following parts: **input layer, convolution layer, pooling layer, activation function layer, full-connection layer and softmax layer,** As shown in Fig.1 below.

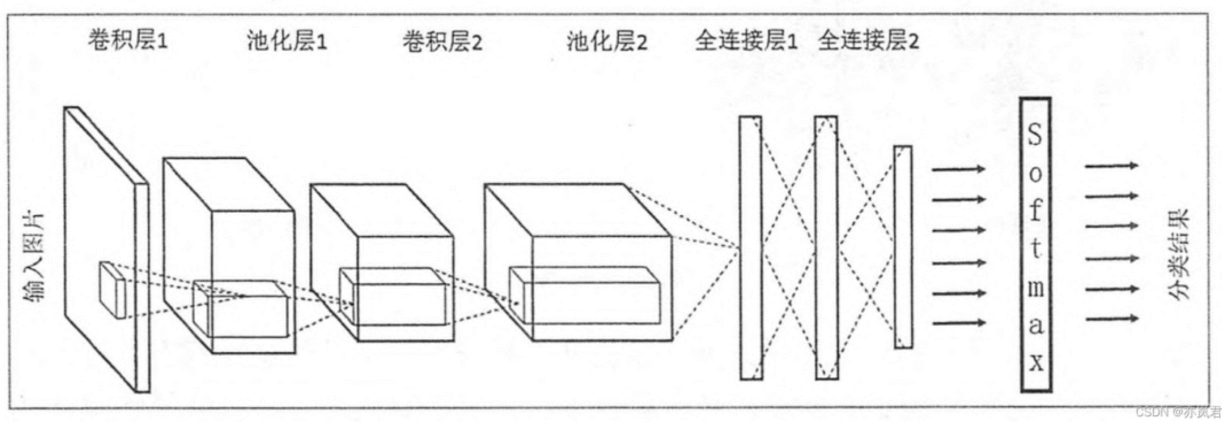

Fig.1 The basic structure of convolutional neural network

The functions of these basic parts are shown below:

- **Input layer:** In image processing CNN, the input layer generally represents the pixel matrix of an image. A picture can be represented by a three-dimensional matrix. The length and width of the 3D matrix represent the size of the image, while the depth of the 3D matrix represents the color channel of the image. For example, a black and white image has a depth of 1, while in RGB color mode, the image has a depth of 3.
- **Convolutional layer:** The core of convolutional neural network is the convolutional layer, and the core part of the convolutional layer is the convolution operation. The operation of inner product (multiplicative and summation of elements one by one) on images (data of different data Windows) and filter matrix (a set of fixed weights: since multiple weights of each neuron are fixed, it can be regarded as a constant filter filter) is the so-called convolution operation, which is also the source of the name of convolutional neural network.
- **Pooling layer:** The core of pooling layer is Pooling operation which uses the overall statistical characteristics of the adjacent area of an input matrix as the output of the location, including Average Pooling, Max Pooling, etc. Pooling simply specifies a value on the region to represent the entire region. Hyperparameters of the pooling layer: pooling window and pooling step. Pooling can also be thought of as a convolution operation. **(My understanding is about the function of the pooling layer is to select some way to reduce dimension compression in order to speed up the computation and retain the typical features in the window, so as to facilitate the next step of convolution/full connection).**

- **Activation function layer:** The activation function here usually refers to the nonlinear activation function, the most important characteristic of activation function is its ability to add nonlinearity into convolutional neural network in order to solve the problems with complex patterns such as computer vision or image processing. The common activation functions include Sigmoid, tanh and Relu. Generally, Relu is used as the activation function of convolutional neural network. The Relu activation function provides a very simple nonlinear transformation method. The function image of Relu is shown below:

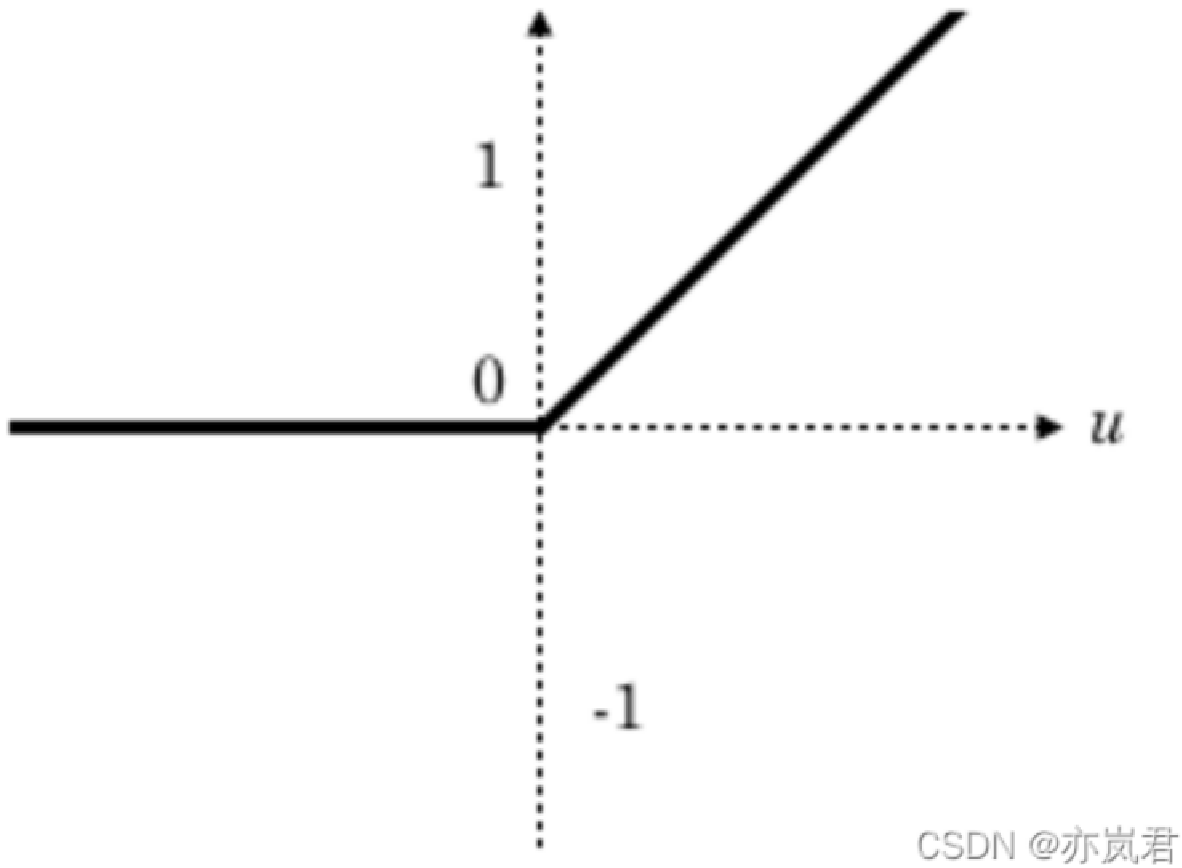

Fig.2 The function image of Relu

- **Full-connection layer:** After the processing of multi-wheel convolution layer and pooling layer, the final classification results are generally given by one or two full-connection layers at the end of CNN. After several rounds of processing of convolution layer and pooling layer, it can be considered that the information in the image has been abstracted into features with higher information content. We can regard the convolution layer and pooling layer as the process of automatic image feature extraction. After the extraction is complete, we still need to use the full-connection layer to complete the sorting task.
- **Softmax layer:** Through the softmax layer, we can get the probability distribution problem that the current sample belongs to different categories. The softmax function will convert the output values of multiple classes into probability distributions in the range of [0, 1]. The function of the softmax layer is shown below:

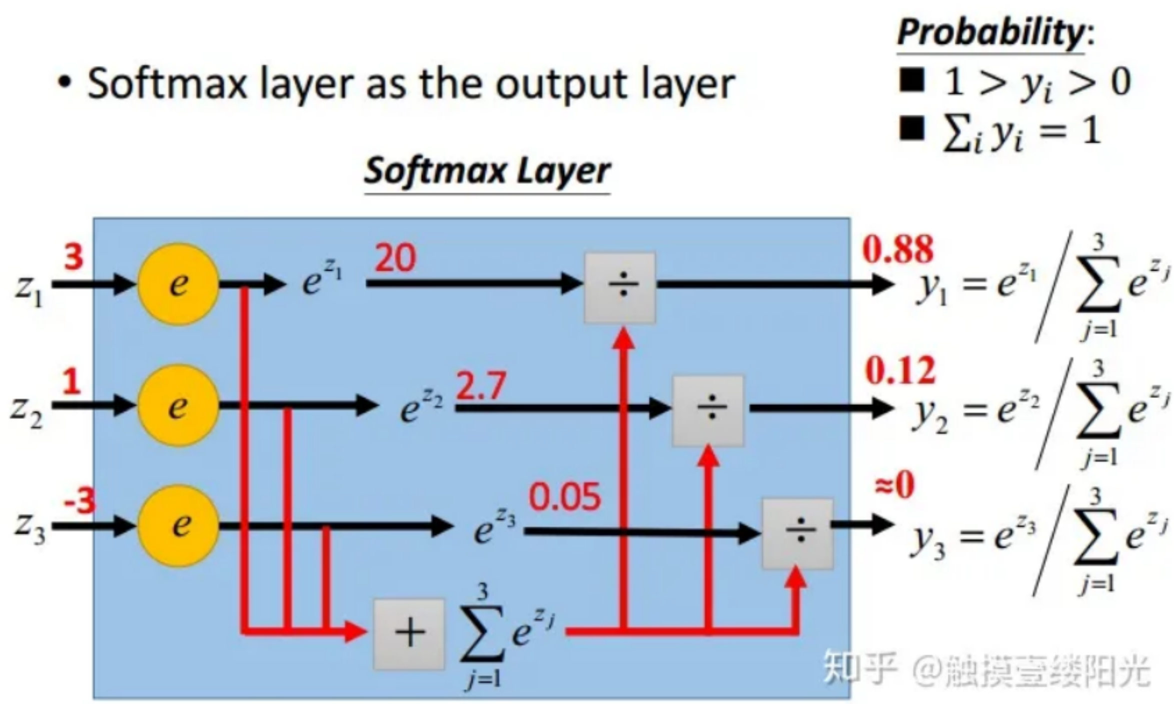

Fig.3 The function of the softmax layer

2. Introduction of MNIST dataset

MNIST dataset is a subset of the dataset in NIST (national institute of standards and technology), available on http://yann.lecun.com/exdb/mnist/ for MNIST dataset, the MNIST dataset mainly includes four files shown below:

文件名称	大小	内容
train-images-idx3-ubyte.gz	9,681 kb	55000张训练集，5000张验证集
train-labels-idx1-ubyte.gz	29 kb	训练集图片对应的标签
t10k-images-idx3-ubyte.gz	1,611 kb	10000张测试集
t10k-labels-idx1-ubyte.gz	5 kb	测试集图片对应的标签

Fig.3 The file included in the MNIST data set

In the above file, the training set contains a total of 60,000 images and labels, while the test set contains a total of 10,000 images and labels. The "idx3" means 3-dimensional, "ubyte" means the image data is stored in the form of bytes, and "t10k" means 10,000 test images. Each picture is a 28*28 pixel handwritten gray matter digital picture of 0 ~ 9, with white characters on black background. The pixel value of the image is 0 ~ 255, the larger pixel value the dot has, the whiter it is. (Its dimension is 1*28*28, and 1 represents the single channel)

3. The codes in hand-written digits recognition by CNN

In this part, I will split the entire program code into 8 sections, as follows:

- **1.MNIST dataset loading**

```python
import torch
import numpy as np
from matplotlib import pyplot as plt
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn.functional as nf
from torch.utils.tensorboard import SummaryWriter
# 对数据进行归一化
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
#导入MNIST数据集，数据集会下载到当前根目录(和本文件同一个目录的文件夹下)的data文件夹下
data_train = MNIST('./data', train = True, download=True, transform = transform)

data_test = MNIST('./data', train=False, download=True, transform= transform)
#分别创建两个DataLoader载入训练集与测试集的数据
# 注意batch-size表示每批样本的大小，一次训练迭代一个batch.因此len(data_train_loader)表示mini-batch的数目
#batch_idx表示batch批的数目下标
data_train_loader = DataLoader(data_train, batch_size=256 ,shuffle= True, num_workers=0)  # 训练集的数据被随机打乱
data_test_loader = DataLoader(data_test, batch_size=1024 , shuffle= False, num_workers=0) # 测试集数据不用做随机排列
```

- **2. Display the MNIST dataset**

This code shows the first 60 images from the last batch of the training set.

```python
figure = plt.figure()
num_of_images = 60

for imgs, targets in data_train_loader:
    break
for index in range(num_of_images):  # 载入训练集index为0-59共60张图片
    plt.subplot(6, 10, index + 1)
    plt.axis("off")
    img = imgs[index, ...]
    plt.imshow(img.numpy().squeeze(), cmap='gray_r')
plt.show()
```

- **3. Construct the CNN model**

This code constructs the convolutional neural network model and instantiates a convolutional neural network.

```python
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
```

```
        self.conv1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 10, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = torch.nn.Sequential(
            torch.nn.Conv2d(10, 20, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(320, 100),
            torch.nn.Linear(100, 50),
            torch.nn.Linear(50, 10),
        )

    def forward(self, x):
        batch_size = x.size(0)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(batch_size, -1)
        # flatten 变成全连接网络需要的输入 (batch, 20,4,4) ==> (batch,320), -1 此处自动算出的是320
        x = self.fc(x)
        return x  # 最后输出的是维度为10的，也就是（对应数学符号的0~9）

model = Net() # 实例化模型
```

- **4. Determine the optimizer and loss function required for the training process**

This code sets up some configurations for neural network training, including the gradient descent optimizer and the learning rate optimizer.

```
model.train()  # 切换模型到训练状态
learning_rate = 0.01
momentum = 0.9
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=momentum, weight_decay = 5e-4)  # lr学习率，momentum冲量
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=6, gamma=0.99, last_epoch=-1)
writer = SummaryWriter()
```

- **5. model training**

This code completes the training process of convolutional neural network through loop iteration.

```
train_loss = 0.0  # 这整个epoch的loss清零
total = 0
correct = 0
epoch = 2
iter_num = 0
```

```python
for i in range(epoch):
    for batch_idx, (inputs, targets) in enumerate(data_train_loader):
        iter_num += 1
        optimizer.zero_grad()
        # forward + backward + update
        outputs = model.forward(inputs)
        loss = nf.cross_entropy(outputs, targets)
        writer.add_scalar("Loss/train", loss, iter_num)
        loss.backward()
        optimizer.step()
        # 把运行中的loss累加起来
        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, dim=1)
        _, predicted = outputs.max(1)
        total += inputs.shape[0]
        correct += predicted.eq(targets).sum().item()

        if batch_idx % 10 == 9:  # 不想要每一次都出loss，浪费时间，选择每xx次出一个平均损失,和准确率
            print('[epoch: %d, batch_idx: %d]: loss: %.3f , acc: %.2f %%'
                  % (i + 1, batch_idx + 1, loss / 100, 100. * correct / total))
            writer.add_scalar('train accuracy per 10 batches', 100. * correct / total, iter_num)
            loss = 0.0
            correct = 0
            total = 0
    scheduler.step()  # 优化并更新学习率
```

- **6. Save the state dictionary of the model after training**

This code saves the state information and related parameters of the instantiated neural network after training.

```python
# 等两次完整的迭代进行完毕后，保存训练好的模型及其参数
save_info = {  # 保存的信息：1.迭代步数 2.优化器的状态字典 3.模型的状态字典
    "iter_num": iter_num, "optimizer": optimizer.state_dict(), "model": model.state_dict()
}
save_path = "./model.pth"  # 将模型存储的位置在当前根目录的文件夹中
torch.save(save_info, save_path)
```

- **7. model testing**

This code completes the testing process of the neural network on the testing set through loop iteration and saves the data information in the following confusion matrix drawing through the **multi-layer nested else-if structure**.

```python
correct = 0
total = 0
epoch = 0
```

```
length = 10
model.eval() # 切换模型为测试状态(没加drop_out层，因此这句话可以随便注释掉)
count = 0
zero = [0] * length
one = [0] * length
two = [0] * length
three = [0] * length
four = [0] * length
five = [0] * length
six = [0] * length
seven = [0] * length
eight = [0] * length
nine = [0] * length
ten = [0] * length # 懒得用那个sklearn的库画confusion了，直接手撸一个算咯
with torch.no_grad():  # 测试集不用算梯度
    for batch_idx, (inputs, targets) in enumerate(data_test_loader):
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, dim=1)  # dim = 1 列是第0个维度，行是第1个维度，沿着行(第1个维度)去找1.最大值和2.最大值的下标
        total += targets.size(0)  # 张量之间的比较运算
        correct_batch = predicted.eq(targets).sum().item()
        correct += predicted.eq(targets).sum().item()
        acc_batch = correct_batch / targets.size(0)
        print(targets.size(0))
        print('[batch_index: %d]: Accuracy on test set: %.1f %% ' % (batch_idx, 100 * acc_batch))  # 求测试的准确率，正确数/总数
        predicted_list = predicted.tolist()
        targets_list = targets.tolist()
        for j in range(targets.size(0)):
            if predicted_list[j] == targets_list[j]:
                if predicted_list[j] == 0:
                    zero[0] += 1
                elif predicted_list[j] == 1:
                    one[1] += 1
                elif predicted_list[j] == 2:
                    two[2] += 1
                elif predicted_list[j] == 3:
                    three[3] += 1
                elif predicted_list[j] == 4:
                    four[4] += 1
                elif predicted_list[j] == 5:
                    five[5] += 1
                elif predicted_list[j] == 6:
                    six[6] += 1
                elif predicted_list[j]== 7:
                    seven[7] += 1
                elif predicted_list[j] == 8:
                    eight[8] += 1
                else:
                    nine[9] += 1
            elif predicted_list[j] == 0:
                if targets_list[j] == 1:
```

```
                    zero[1] += 1
                elif targets_list[j] == 2:
                    zero[2] += 1
                elif targets_list[j] == 3:
                    zero[3] += 1
                elif targets_list[j] == 4:
                    zero[4] += 1
                elif targets_list[j] == 5:
                    zero[5] += 1
                elif targets_list[j] == 6:
                    zero[6] += 1
                elif targets_list[j] == 7:
                    zero[7] += 1
                elif targets_list[j] == 8:
                    zero[8] += 1
                else:
                    zero[9] += 1
            elif predicted_list[j] == 1:
                if targets_list[j] == 0:
                    one[0] += 1
                elif targets_list[j] == 2:
                    one[2] += 1
                elif targets_list[j] == 3:
                    one[3] += 1
                elif targets_list[j] == 4:
                    one[4] += 1
                elif targets_list[j] == 5:
                    one[5] += 1
                elif targets_list[j] == 6:
                    one[6] += 1
                elif targets_list[j] == 7:
                    one[7] += 1
                elif targets_list[j] == 8:
                    one[8] += 1
                else:
                    one[9] += 1
            elif predicted_list[j] == 2:
                if targets_list[j] == 0:
                    two[0] += 1
                elif targets_list[j] == 1:
                    two[1] += 1
                elif targets_list[j] == 3:
                    two[3] += 1
                elif targets_list[j] == 4:
                    two[4] += 1
                elif targets_list[j] == 5:
                    two[5] += 1
                elif targets_list[j] == 6:
                    two[6] += 1
                elif targets_list[j] == 7:
                    two[7] += 1
                elif targets_list[j] == 8:
```

```
                    two[8] += 1
                else:
                    two[9] += 1
            elif predicted_list[j] == 3:
                if targets_list[j] == 0:
                    three[0] += 1
                elif targets_list[j] == 1:
                    three[1] += 1
                elif targets_list[j] == 2:
                    three[2] += 1
                elif targets_list[j] == 4:
                    three[4] += 1
                elif targets_list[j] == 5:
                    three[5] += 1
                elif targets_list[j] == 6:
                    three[6] += 1
                elif targets_list[j] == 7:
                    three[7] += 1
                elif targets_list[j] == 8:
                    three[8] += 1
                else:
                    three[9] += 1
            elif predicted_list[j] == 4:
                if targets_list[j] == 0:
                    four[0] += 1
                elif targets_list[j] == 1:
                    four[1] += 1
                elif targets_list[j] == 2:
                    four[2] += 1
                elif targets_list[j] == 3:
                    four[3] += 1
                elif targets_list[j] == 5:
                    four[5] += 1
                elif targets_list[j] == 6:
                    four[6] += 1
                elif targets_list[j] == 7:
                    four[7] += 1
                elif targets_list[j] == 8:
                    four[8] += 1
                else:
                    four[9] += 1
            elif predicted_list[j] == 5:
                if targets_list[j] == 0:
                    five[0] += 1
                elif targets_list[j] == 1:
                    five[1] += 1
                elif targets_list[j] == 2:
                    five[2] += 1
                elif targets_list[j]== 3:
                    five[3] += 1
                elif targets_list[j] == 4:
                    five[4] += 1
```

```
                elif targets_list[j] == 6:
                    five[6] += 1
                elif targets_list[j] == 7:
                    five[7] += 1
                elif targets_list[j] == 8:
                    five[8] += 1
                else:
                    five[9] += 1
            elif predicted_list[j] == 6:
                if targets_list[j] == 0:
                    six[0] += 1
                elif targets_list[j] == 1:
                    six[1] += 1
                elif targets_list[j] == 2:
                    six[2] += 1
                elif targets_list[j] == 3:
                    six[3] += 1
                elif targets_list[j] == 4:
                    six[4] += 1
                elif targets_list[j] == 5:
                    six[5] += 1
                elif targets_list[j] == 7:
                    six[7] += 1
                elif targets_list[j] == 8:
                    six[8] += 1
                else:
                    six[9] += 1
            elif predicted_list[j] == 7:
                if targets_list[j] == 0:
                    seven[0] += 1
                elif targets_list[j] == 1:
                    seven[1] += 1
                elif targets_list[j] == 2:
                    seven[2] += 1
                elif targets_list[j] == 3:
                    seven[3] += 1
                elif targets_list[j] == 4:
                    seven[4] += 1
                elif targets_list[j] == 5:
                    seven[5] += 1
                elif targets_list[j] == 6:
                    seven[6] += 1
                elif targets_list[j] == 8:
                    seven[8] += 1
                else:
                    seven[9] += 1
            elif predicted_list[j] == 8:
                if targets_list[j] == 0:
                    eight[0] += 1
                elif targets_list[j] == 1:
                    eight[1] += 1
                elif targets_list[j] == 2:
```

```
                    eight[2] += 1
                elif targets_list[j] == 3:
                    eight[3] += 1
                elif targets_list[j] == 4:
                    eight[4] += 1
                elif targets_list[j] == 5:
                    eight[5] += 1
                elif targets_list[j] == 6:
                    eight[6] += 1
                elif targets_list[j] == 7:
                    eight[7] += 1
                else:
                    eight[9] += 1
            else:
                if targets_list[j] == 0:
                    nine[0] += 1
                elif targets_list[j] == 1:
                    nine[1] += 1
                elif targets_list[j] == 2:
                    nine[2] += 1
                elif targets_list[j] == 3:
                    nine[3] += 1
                elif targets_list[j] == 4:
                    nine[4] += 1
                elif targets_list[j] == 5:
                    nine[5] += 1
                elif targets_list[j] == 6:
                    nine[6] += 1
                elif targets_list[j] == 7:
                    nine[7] += 1
                else:
                    nine[8] += 1
        count += 1024
        # if count == 10000:
        #     break
        writer.add_scalar('test accuracy per batch', 100 * acc_batch, batch_idx)
acc = correct / total
print('Average accuracy on test set: %.1f %% ' % (100. * acc))  # 求测试的准确率，正确数/总数
```

- **8. Plot the confusion matrix results for test accuracy**

This code uses the data saved above to draw the confusion matrix result of the CNN model when testing all the images in the testing set.

```
import matplotlib.pyplot as plt
import numpy as np

confusion_fig = np.array((zero, one, two, three, four, five, six, seven, eight, nine))

```

```
# 热度图，后面是指定的颜色块，可设置其他的不同颜色
plt.imshow(confusion_fig, cmap=plt.cm.Blues)
# ticks 坐标轴的坐标点
# label 坐标轴标签说明
indices = range(len(confusion_fig))
# 第一个是迭代对象，表示坐标的显示顺序，第二个参数是坐标轴显示列表
# plt.xticks(indices, [0, 1, 2])
# plt.yticks(indices, [0, 1, 2])
font2 = {
# 'family' : 'Times New Roman',
'weight' : 'semibold',
'size' : 11.5,
}
plt.xticks(indices, ['0', '1', '2', '3', '4', '5', '6','7', '8', '9'])
plt.yticks(indices, ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])

plt.colorbar()

plt.xlabel('Actual Class', fontdict=font2)
plt.ylabel('Predicted Class', fontdict= font2)
#标题要不要加粗一下子
plt.title('Confusion Matrix',fontsize='13',fontweight='semibold')

# 显示数据
for first_index in range(len(confusion_fig)):  # 第几行
    for second_index in range(len(confusion_fig[first_index])):  # 第几列
        # plt.text(first_index, second_index, confusion_fig8[first_index][second_index])
        if first_index == second_index:
            plt.text(x=first_index, y=second_index, s=confusion_fig[second_index, first_index], color='r',
                     weight='bold', horizontalalignment='center', verticalalignment='center')
        else:
            plt.annotate(confusion_fig[second_index, first_index], xy=(first_index, second_index),
                         horizontalalignment='center', verticalalignment='center')
plt.show()
```

4. Experimental result

- **The part of the images in MNIST dataset**

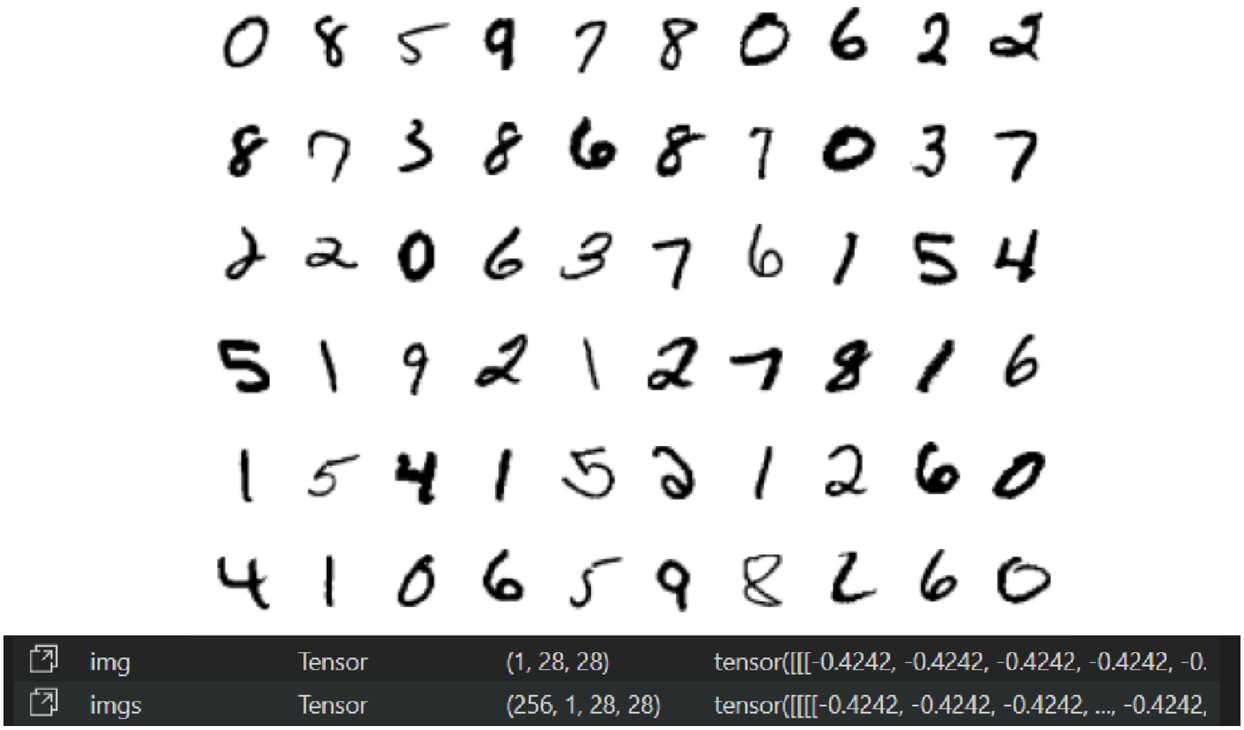

Fig.4 The images displayed in the MNIST dataset

As can be seen from the figure above, the format of each picture in the MNIST data set is 1*28*28. Since the size of a batch in the training set is 256, the size of the tensor "imgs" is 256*1*28*28.

- **The result in the training process (All the figures below is obtained from the tensorboard)**

Since the training set has a total of 60,000 images, each iteration of the loop trains one batch. Each batch contains 256 images, so it takes 235 iterations to train the complete training set. Meanwhile, since there are two epochs in our training process, we need to iterate 470 times in total. The following two figures show the value of loss and accuracy during training process as the number of iterations changes. The horizontal coordinate is the number of loop iterations, and the vertical coordinate is the loss or accuracy during the training process. **(The accuracy rate is calculated per 10 loop iterations)**

Fig.5 The loss of the testing process varies with the number of iterations of the loop

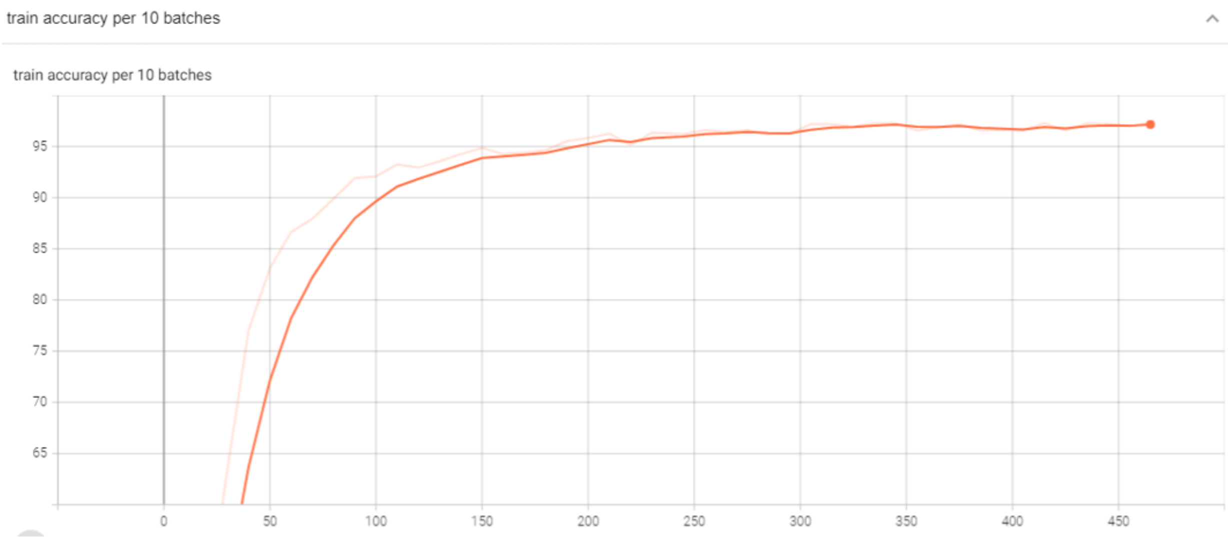

Fig.6 The accuracy of the training process varies with the number of iterations of the loop

As can be seen from the above two figures, with the increase of the number of iterations in the training process, the loss during training is constantly decreasing, while the accuracy is constantly rising. When the training is about to end, the loss and accuracy in the training have basically converged and the accuracy can reach more than 95%. These results are in good agreement with the characteristics of a well-constructing convolutional neural network in training.

- **The result in the testing process (The figure below is obtained from the tensorboard)**

Since the testing set has a total of 10,000 images, each iteration of the loop trains one batch. Each batch contains 1024 images, so it takes 10 iterations to test the complete testing set. The following figure show the value of accuracy during testing process as the number of iterations changes. The horizontal coordinate is the number of loop iterations, and the vertical coordinate is the accuracy during the testing process. **(The accuracy rate is calculated per 10 loop iterations)**

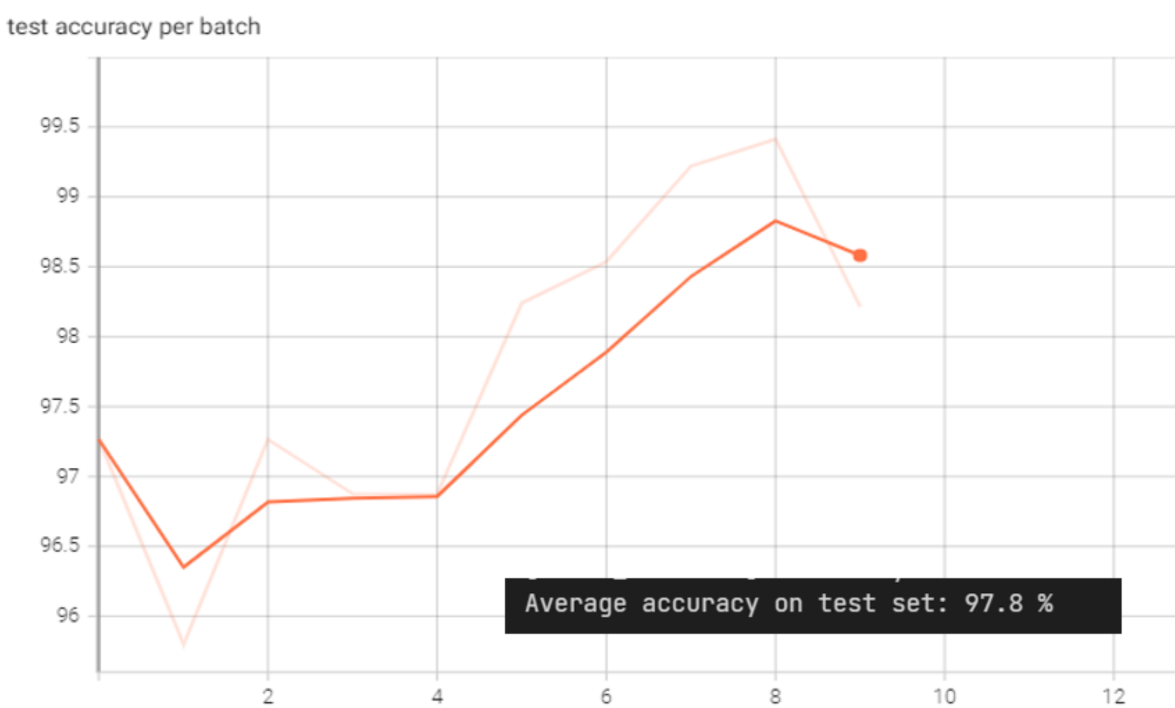

Fig.7 The accuracy of the testing process varies with the number of iterations of the loop

As can be seen from the figure above, the accuracy of our trained neural network in different iterations during the testing process can reach more than 95%, and the average accuracy of these 10 iterations can reach 97.8%. It can be seen that the convolutional neural network can well accomplish the task of handwritten digit recognition.

- **Detailed evaluation in the testing dataset**

In order to further evaluate the performance of the model in the testing set, we extracted the first 16 images of the last batch of the testing set and did the recognition test. Then, we compared the difference between the real value and the predicted value of the pictures, as shown in the figure below:

Fig.8 The test results of the first 16 digital images in the testing set
(The first column of the predicted value and the real value represents the image index, the second column represents the category label to which the image belongs)

From the figure above, we can see that for the 16 extracted images, the convolutional neural network classified them into the correct categories completely, and the predicted value and the true value were exactly the same at each image index.

After that, we drew the confusion matrix of the recognition result for all images in the testing set, as shown in the figure below:

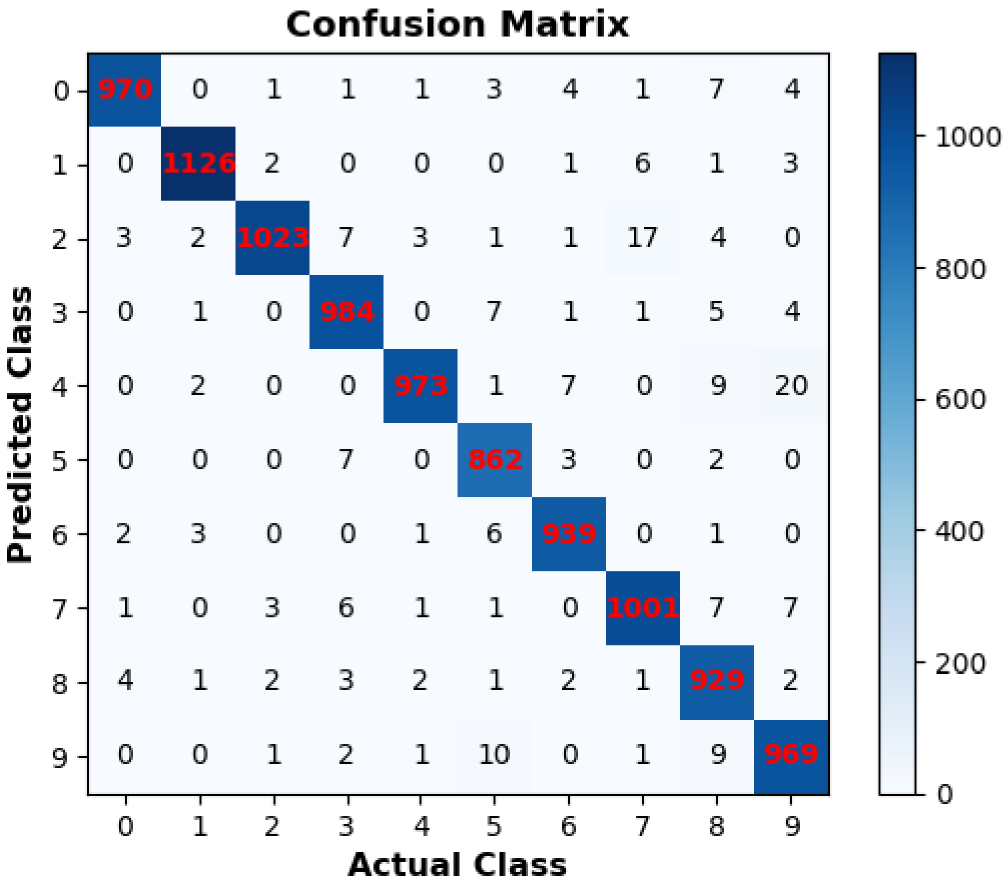

Fig.9 The Confusion matrix for testing set

As can be seen from the figure above, the convolutional neural network can be used to classify most of the 10,000 pictures of different categories in the testing set into the correct category, which reflects that the convolutional neural network constructed by us can perform handwritten digit recognition task well.

5. Conclusion and Prospect

- **Conclusion:**
  - After two iterations by using our CNN model, the average accuracy of the model on the test set reached 97.8%.
  - All the results of evaluating the testing set by our model fully show that the convolutional neural network can well accomplish the task of handwritten digit recognition.
  - CNN is mainly suitable for image classification and recognition tasks, because the original design of CNN is actually to carry out convolution operation of the images.
- **Prospect:**
  - The structure of the neural network can continue to be optimized. (Such as adding some Drop_out layers to prevent overfitting and Batch Normalization layers to prevent the gradient disappearing)
  - Change the structure of the CNN model, such as using the residual neural network (ResNet) or the classic feature extraction network GoogLeNet.

- Try to use few-shot or zero-shot Learning model to verify whether a certain kind of digital images can be accurately classified according to effective features when the size of the training dataset is very small.