AAI Project Report: Train classifier for modified MNIST with unstable features

Name: 吉辰卿　叶浩腾　　SID: 12332152　12332160

Part 1: Dataset Preprocessing

The goal of this project is to effectively remove unstable features from the training set and test set, so that our model (machine learning or neural network) can effectively learn the correct data features to accurately classify the test set. Before we can accomplish this task, we need to **explore what is a false feature**. We observe that the data format of training set, validation set and test set are all **10*28*28**, which has one more dimension than the data format **28*28** used in our traditional MNIST handwritten digit recognition, and the number of channels in this dimension is 10. Therefore, **the unstable features are likely to be caused by this new dimension --10 channels**. Before building a network model for recognition, we must find a way to remove this unstable feature.

We started it by drawing images of several training sets and find an issue: In most image arrays (.npy) of the training set and validation set, **only one of the 10 channels in the first dimension of each.npy array holds data, and the remaining nine channels are all zeros. The value of the channel where the data is stored is the number corresponding to the image label.** For example, in an image data array with label 2, the number of channel that store the data in the first dimension is probably 2. Later, when we looked at the test set, we noticed a difference: **In the test set, the channel of the image array in the first dimension that stored the data did not corresponding to its own label.** For example, in all array data with the number 6, the channel in which the data is stored in the first dimension is not necessarily 6. This difference leads to the difference in the unstable feature distribution between the training set and the test set, that is, **if we cannot guarantee the number of channel in which the data array that stores the data in the test set is consistent with the number of channel in which the data array that stores the data in the training set of the same label,** the trained model is likely to think that the two data are very different and do not belong to the same category. This means that if we can't effectively remove the unstable features from the training data and the test data, the test accuracy will be very poor. In a conclusion, **the unstable features come from the 10 channels in the first dimension of the data set, and the different distribution of unstable features in the training set and the test set comes from the different channels that store the data in the image array of the training set and the test set.**

Therefore, we first consider the pre-processing of training data, validation data and test data. The data processing methods are as follows:

- **Loop through and merge all data dimensions**

  Firstly, for each data, we traverse the 10 channels of the first dimension. If the mean and variance of this channel are 0, it is proved that there is no data in this channel; Otherwise, it is proved that there is data stored in this channel. We binary the channel that stores the data. Finally, we merge all channels from the first dimension and binary the combined result. To make the code easier to understand, we write the above processing into two functions, `binary_data` and `de_dimension_data`. Of course, we can also not binary the channel containing data, but normalize the data of this channel, as shown in the following function `norm_data`, as shown below:

```python
# Binarization Function
def binary_data(input: np.array):
    mean = np.mean(input)
    std = np.std(input)
    if std == 0 and mean == 0:
        return np.array([-1])
    else:
        input = np.where(input > mean, 1, 0)
        # print(input)
        return input

# Normalization Function
def norm_data(input: np.array):
    mean = np.mean(input)
    std = np.std(input)
    if std == 0:
        if mean == 0:
            return np.array([-1])
        else:
            # print('std==0')
            input = input / 255.0
    else:
        # print(f'std={std}, mean={mean}')
        input = (input - mean) / std
    return input

# Data Dimension Processing
def de_dimension_data(input: np.array):
    de_data = np.zeros((input.shape[1], input.shape[2]))
    num = 0
    for dimension in input:
        # print('***************')
        norm = norm_data(dimension)  // or norm = binary_data(dimension)
        if np.array_equal(norm, np.array([-1])):
            # print("ALL 0 dimension, give it up. ")
            pass
        else:
            # print("Found effective data, keep it. ")
            num += 1
            de_data = de_data + norm
    if num > 1:
        print('multi layers')
        de_data = norm_data(de_data)   // or de_data = binary_data(de_data)
    return de_data
```

For normalization or binarization, we do some ablation experiments later and provide the test results of both schemes.

We can use the above functions to preprocess the training set , validation set in the file path **(The preprocess of the test set is in the testing process, we process the test set by reading the image array through a loop)**, making its size 1*28*28, as follows:

```python
class MNISTDataset(Dataset):
    def __init__(self, set_type='p'):
        self.norm_data_set = np.zeros((10, 28, 28))
        self.label_count_set = np.zeros(10)
        self.data = []
        self.label_set = []
        self.labels = []
        self.type = set_type
        if self.type == 't': ##准备训练集和验证集
            data_dirty = []
            print('Type t, wrap all train and val data. ')
            train_folder = sorted(os.listdir(train_dir))
            val_folder = sorted(os.listdir(val_dir))
            for label, sub_folder in enumerate(train_folder):
                folder_url = os.path.join(train_dir, sub_folder)
                files = sorted(os.listdir(folder_url))
                for file in files:
                    file_url = os.path.join(folder_url, file)
                    this_data = np.load(file_url)
                    # print(this_data[0])
                    # print(this_data.shape)
                    this_data = de_dimension_data(this_data)
                    # print(this_data.shape)
                    # this_data = torch.from_numpy(this_data)
                    # self.data.append(this_data)
                    data_dirty.append(this_data)
                    self.label_set.append(label)
            print(f'train data wrap over, size is {data_dirty.__len__()}')
            for label, sub_folder in enumerate(val_folder):
                folder_url = os.path.join(val_dir, sub_folder)
                files = sorted(os.listdir(folder_url))
                for file in files:
                    file_url = os.path.join(folder_url, file)
                    this_data = np.load(file_url)
                    # print(data.shape)
                    this_data = de_dimension_data(this_data)
                    # print(this_data.shape)
                    # this_data = torch.from_numpy(this_data)
                    # self.data.append(this_data)
                    data_dirty.append(this_data)
                    self.label_set.append(label)
            print(f'train data wrap over, size is {data_dirty.__len__()}')

    def __len__(self):
        return len(self.data)

     def __getitem__(self, idx):
        out_label = torch.tensor(int(self.labels[idx]), device=self.device)
        out_data = torch.tensor(self.data[idx], device=self.device)
        out_data = torch.unsqueeze(out_data, dim=0)
        out_label = torch.squeeze(out_label)
        data_pair = {
            'data': out_data,
            'label': out_label
        }
        return data_pair

all_dataset = MNISTDataset(device=device)
```

We noticed that in the above codes, we merged the training set and the verification set, **because there were only 100 images in the existing verification set and some pictures were labeled incorrectly**. In order to improve the accuracy in the verification process, we **mixed the training set and the verification set and re-divided the training set and the verification set in a certain proportion in the following steps**. Before we redivide the training set and the validation set below, we will refer these two sets as the training set.

- **Discard the error data which affect the training of neural network by some threshold**

  After merging the dimensions in the previous step, **we have now logically removed the unstable features caused by the 10 channels in the first dimension.** However, when we scrutinized the training dataset, we found some problems, as shown below:

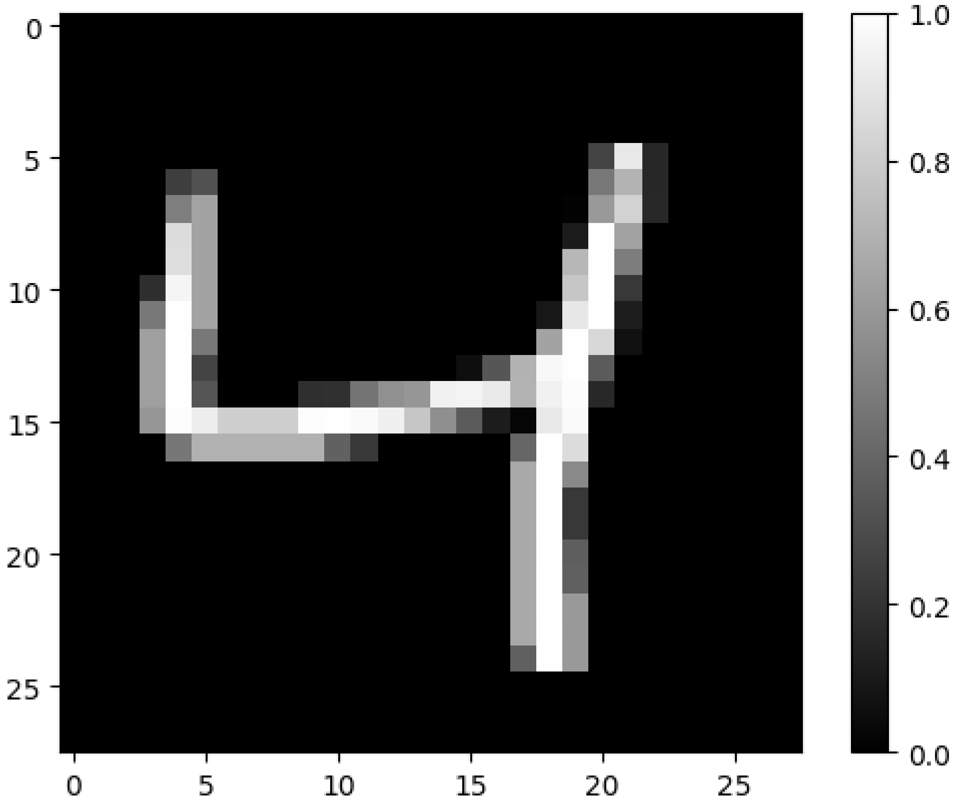


Fig.1 The image of error label in the training and validation set, which corresponds to the label of "8"

Through a large number of drawings, we can find that there are a large number of the false image datas in the training set and verification set, and **their displays of real images are inconsistent with their corresponding labels**. These wrong datas will seriously affect the accuracy of model training. Therefore, **before building the model, we must determine a suitable threshold to filter out these false images.**

Firstly, we sum and average each class of data in the first dimension of the training set obtained above. We choose 0.5 as the threshold, setting the pixel value to 1 if its value is greater than 0.5, and 0 if it is not greater than 0.5. After the above operation, we can **get the "average images" of the 10 kinds of images data array in the training set**, as shown in the figure below:

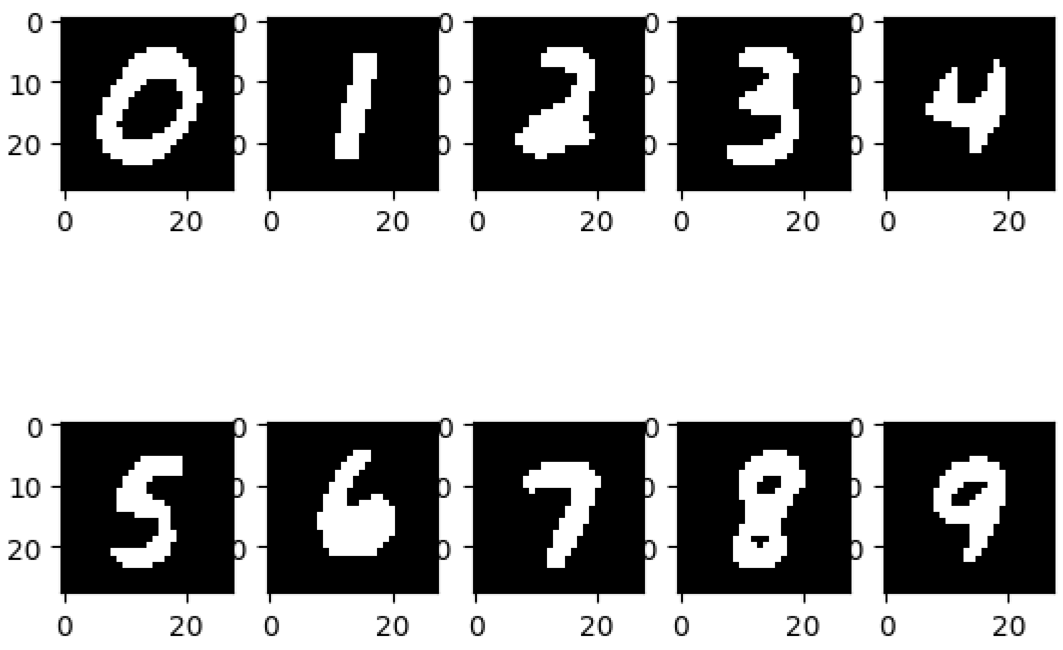


Fig.2 The "average image" from label 0 to 9 (Obtained by the mean and threshold screening of all images in their respective label)

The corresponding codes are shown below:

```python
for (index, datas) in enumerate(data_dirty):
            # print(self.labels[index])
            self.norm_data_set[int(self.label_set[index])] += datas[0]
            self.label_count_set[int(self.label_set[index])] += 1
            print(self.label_count_set)
            for i in range(10):
                self.norm_data_set[i] = self.norm_data_set[i] / self.label_count_set[i]
                self.norm_data_set[i] = np.where(self.norm_data_set[i] > 0.5, 1, 0)
            # print(np.mean(self.norm_data_set[0]), np.std(self.norm_data_set[0]))
            plt.figure()
            idx = 1
            idx_d = 1
            for i in range(1, 11):
                plt.subplot(2, 5, i)
                plt.imshow(self.norm_data_set[i - 1] ,cmap='gray', vmin=0, vmax=1)
```

After that, we still traverse each image in the training dataset and subtract and add it to the "standard" image with the corresponding label. When adding, the result is 1 if the sum of the two is greater than 0, and 0 if not. After that, we sum the results of subtraction and addition in the whole image. At the same time, some evaluation indexes `diff_to_data`, `diff_to_norm` and `diff_to_sum` are set according to the pixel sum of each image and the pixel sum of "standard image". Then, we use their thresholds to determine if the picture is mislabeled. **Obviously, if the pixel sum of the result after subtracting the image from the standard image is too large, it proves that the image probably does not belong to the category displayed by the corresponding label, and we need to discard it**. Finally, through this threshold detection, **we**

discard a few images with high errors in the training set and retain most of the remaining images. The corresponding code is as follows:

```python
 for (index, datas) in enumerate(data_dirty):
    # if int(self.label_set[index]) == 4:
    # if abs((self.mean_set[0] - np.mean(datas)) / self.mean_set[0]) > 20:
    diff_data = abs(self.norm_data_set[int(self.label_set[index])] - datas[0])
    sum_data = np.where((self.norm_data_set[int(self.label_set[index])] + datas[0]) > 0, 1, 0)
    diff_count = diff_data.sum()
    data_count = datas[0].sum()
    norm_count = self.norm_data_set[int(self.label_set[index])].sum()
    sum_count = sum_data.sum()
    diff_to_data = diff_count / data_count
    diff_to_norm = diff_count / norm_count
    diff_to_sum = diff_count / sum_count

    if diff_to_data > 0.9 and diff_to_norm > 0.9 and diff_to_sum > 0.7:
        self.label_count_set[int(self.label_set[index])] -= 1

    else:
        self.data.append(torch.from_numpy(datas))
        self.labels.append(self.label_set[index])
```

After that, we divide the training set obtained above into the training set and the validation set again, the **partition ratio is 9:1**, as follows:

```python
train_size = int(0.9 * len(all_dataset))
test_size = len(all_dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(all_dataset, [train_size, test_size])
## 划分好了训练集和验证集的DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

Note that: since the test set has no labels, we do not need to prepare another dataloader for the test set. **In the later testing process, we test by reading the image array in the test set through the for-loop**.

Part 2: Model Description

Since MNIST handwritten digits are image datasets, **the processing of image data and classification recognition are generally realized by convolutional neural network (CNN)**. The advantage of CNN in this kind of image recognition tasks is that it can effectively capture the local features in the image through convolutional layer and pooling layer, and realize position invariance and hierarchical feature learning. Parameter sharing and sparse connections make CNN have fewer parameters, improve the computational efficiency, and ensure the generalization ability of the model by countering overfitting. These structures make CNN become an ideal choice for processing large image datasets and achieving good performance, especially for tasks such as image classification, object detection, and face recognition. CNN is neural network 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.3 below.

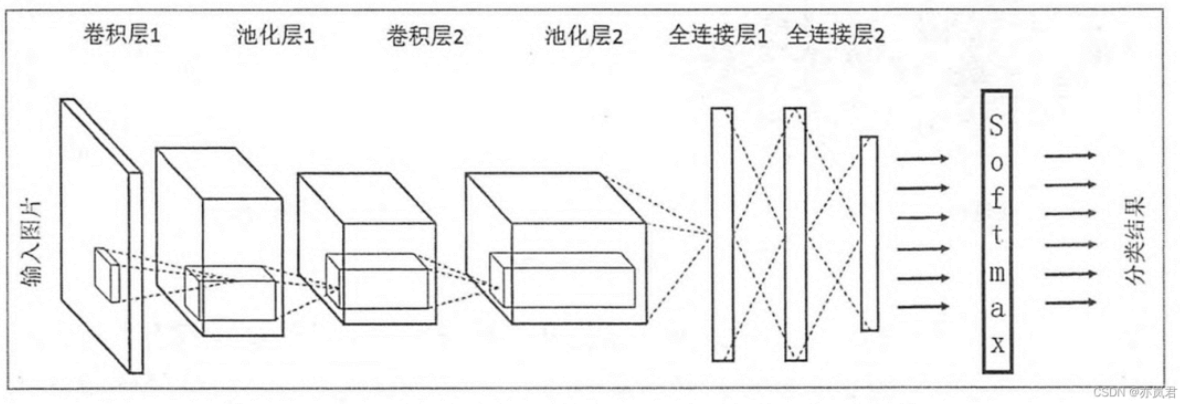


Fig.3 The basic structure of convolutional neural network

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. **(Our 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).**
- **Dropout layer:** The dropout layer in CNN reduces the risk of overfitting during the training phase by randomly inactivating a subset of neurons. This enables the network to be independent of specific neurons, improving model generalization and making it more robust and less sensitive to noise. In the testing phase, the dropout layer needs to be turned off to enable the model to use all neurons comprehensively to make predictions.
- **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:

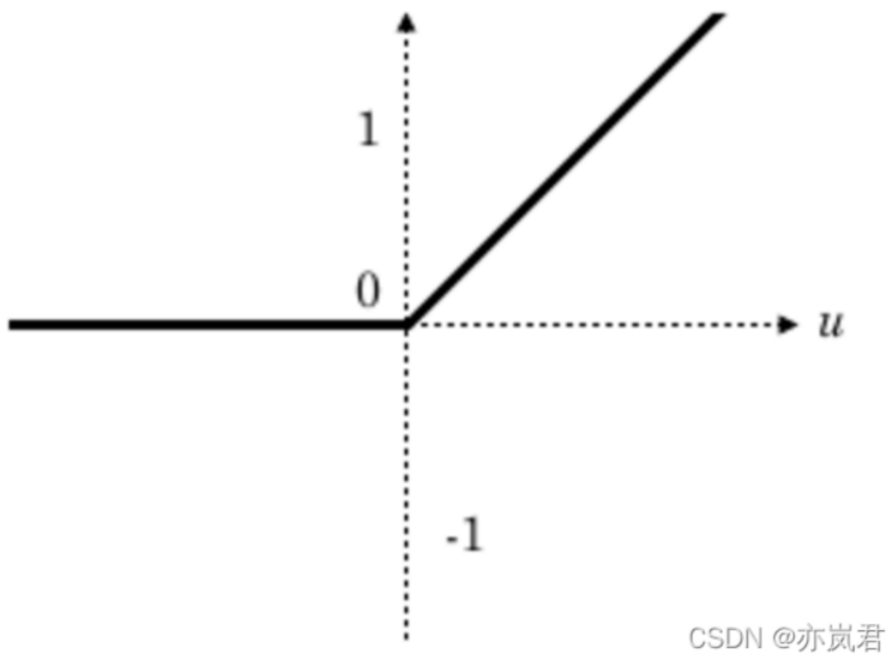


Fig.4 The function image of Relu

- **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:

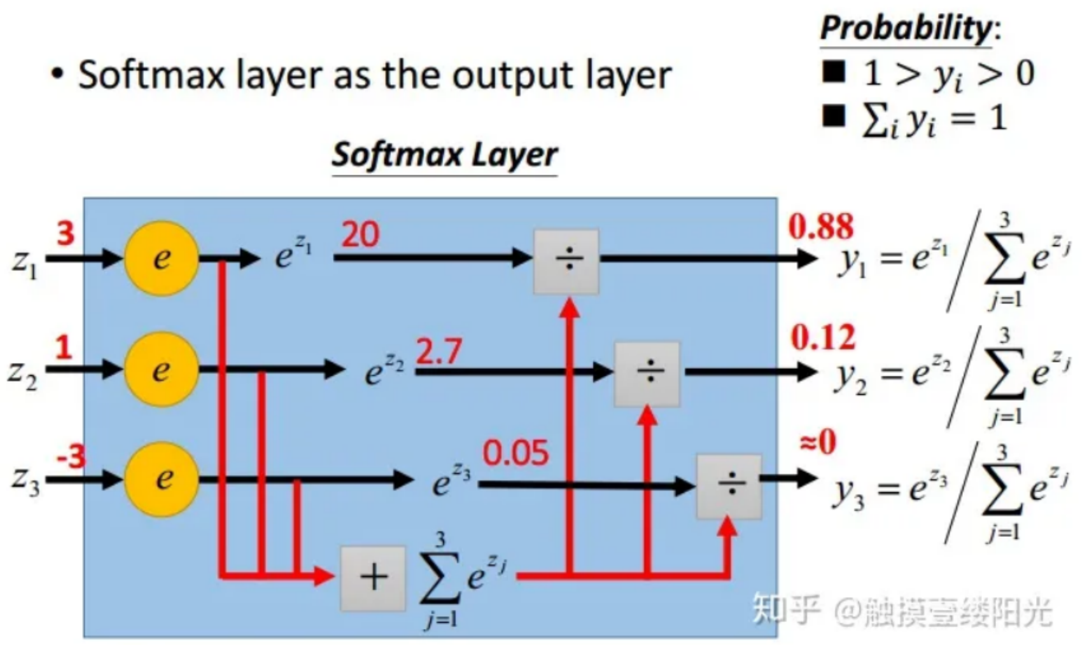

• Softmax layer as the output layer
Probability:
$1 > y_i > 0$
$\sum_i y_i = 1$
Softmax Layer
z_1 3
z_2 1
z_3 -3
e
e^{z_1} 20
e^{z_2} 2.7
e^{z_3} 0.05
+
$\sum_{j=1}^{3} e^{z_j}$
÷
0.88 $y_1 = e^{z_1} / \sum_{j=1}^{3} e^{z_j}$
0.12 $y_2 = e^{z_2} / \sum_{j=1}^{3} e^{z_j}$
≈0 $y_3 = e^{z_3} / \sum_{j=1}^{3} e^{z_j}$
知乎 @触摸壹缕阳光

Fig.5 The function of the softmax layer

In this Project, our CNN model structure is shown below:

```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(2, 2)
        )
        self.conv2 = torch.nn.Sequential(
            torch.nn.Conv2d(10, 20, kernel_size=3),
            torch.nn.ReLU(),

        )
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(2000, 500),
            torch.nn.ReLU(),
            torch.nn.Linear(500, 10)
        )

    def forward(self, x):
        batch_size = x.size(0)
        # x = x.unsqeeze(1)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(batch_size, -1)
        x = self.fc(x)
        x = torch.nn.functional.log_softmax(x, dim=1)
        return x

model = Net().to(device)
```

The visualization structure of the CNN model above is shown in the figure below:

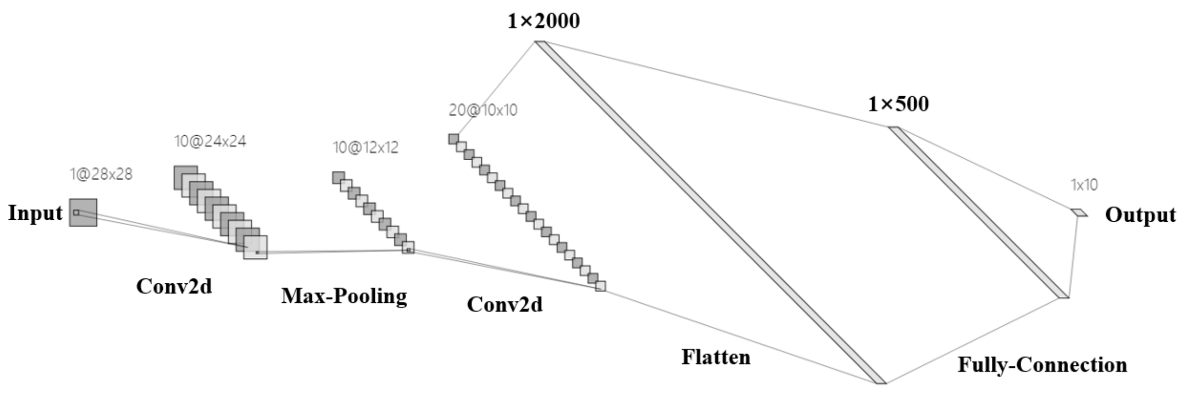


Fig.6 The visualization structure of our CNN model

To get the computation process of our CNN model, we should use the codes below:

```python
from torchviz import make_dot
# 创建虚拟输入，可视化计算图
dummy_input = torch.randn(512, 1, 28, 28)
dot = make_dot(model(dummy_input), params=dict(model.named_parameters()))
dot.render("cnn_computation_graph", format="png", cleanup=True)
```

Then, we can get the computational graph (computation process) of our CNN model:

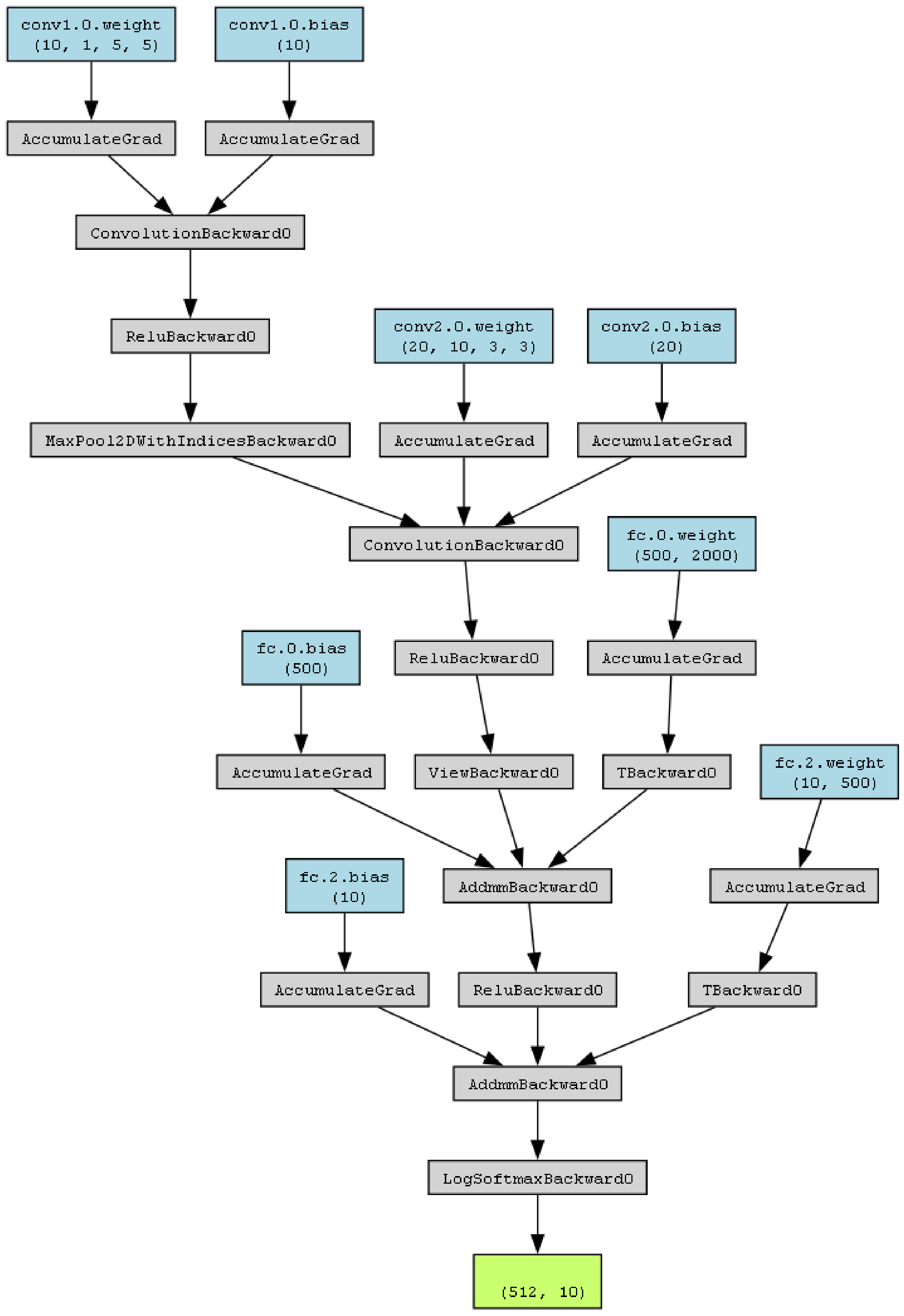


Fig.7 The computational graph (computation process) of our model

Part 3: Experimental Details

- **Hyperparameters in learning rate**

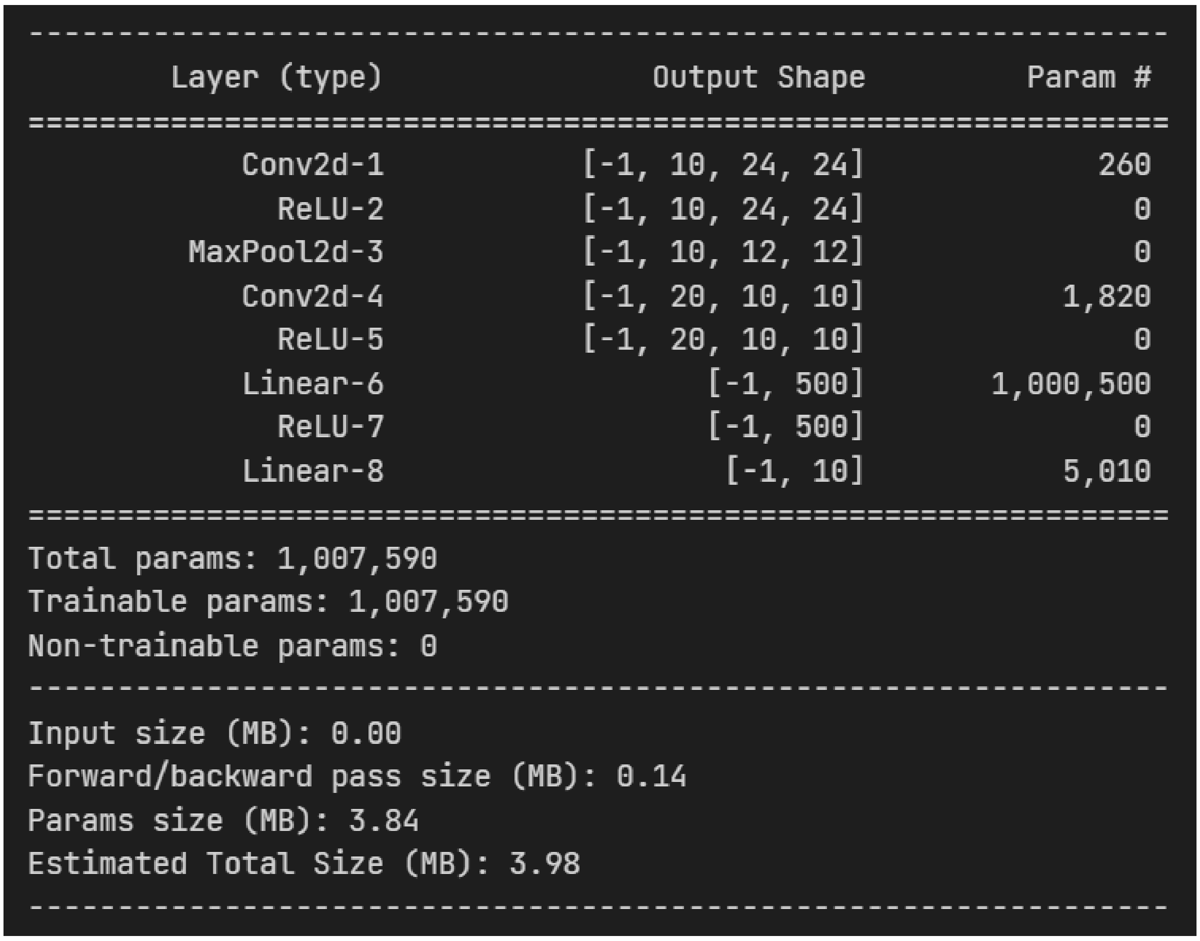
```
batch size (in the training set, validation set and the test set): 512
training epoch: 50
learning_rate: 1e-4
momentum: 0.9
Gradient descent optimizer: torch.optim.SGD (parameters = model.parameters(), lr=learning_rate, momentum=momentum)
Learning rate attenuation optimizer: torch.optim.lr_scheduler.StepLR(optimizer, step_size=6, gamma=0.99, last_epoch=-1)
```

- **Model Size**

We can get the size of our model with the following code and at the same time, we need to input the input_size. Since our previous data preprocessing **added all the first dimensions (10) of this data into one dimension, the input size of a single image is 1*28*28.**

```
## 查看模型的size
from torchsummary import summary
summary(model, input_size=(1, 28, 28))
```

Then, the output of this code then reflects the size of our model, as shown in the figure below:



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 10, 24, 24]             260
              ReLU-2           [-1, 10, 24, 24]               0
         MaxPool2d-3           [-1, 10, 12, 12]               0
            Conv2d-4           [-1, 20, 10, 10]           1,820
              ReLU-5           [-1, 20, 10, 10]               0
            Linear-6                  [-1, 500]       1,000,500
              ReLU-7                  [-1, 500]               0
            Linear-8                   [-1, 10]           5,010
================================================================
Total params: 1,007,590
Trainable params: 1,007,590
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.14
Params size (MB): 3.84
Estimated Total Size (MB): 3.98
----------------------------------------------------------------
```

Fig.8 Our model size

Part 4: Training and Validation Records

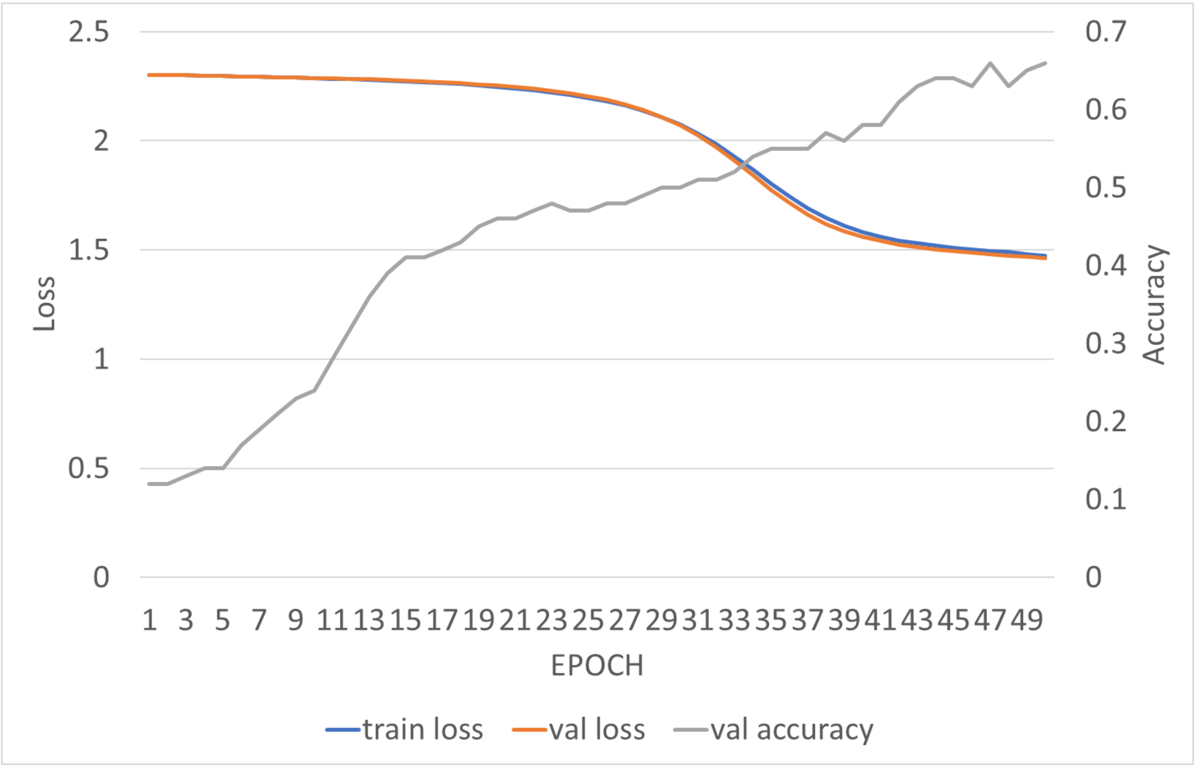
The procedure for training and validating process of the above CNN model is more complex than normal machine learning models by `sklearn`, requiring us to **training and validating iteratively within a set epoch range**. Each iteration involves the following steps: `reading the data and labels in a batch`, `calculating the loss based on the output of the model and the true label`, `backpropagation (loss derivation with the model parameters)`, and `adjusting the model parameters according to the set learning rate and backpropagation results by the optimizer`. Finally, at the end of each epoch, we `adjusted the learning rate of the optimizer using the learning rate attenuation optimizer`. The codes of the training and validating process are shown below:

```python
import torch.nn.functional as nf

train_loss = 0.0  # 这整个epoch的loss清零
total = 0
correct = 0
log = np.zeros([EPOCH, 3])
iter_num = 0
for epoch in range(EPOCH):
    train_loss_list = []
    for batch_index, traindata in enumerate(train_dataloader):
        # for batch_index, (datas, labels) in enumerate(train_dataloader):
        iter_num += 1
        optimizer.zero_grad()
        data, label = traindata['data'].to(device), traindata['label'].to(device)
        data = data.float()
        # data, label = datas.to(device), labels.to(device)
        # print(data.shape)
        # print(data.dtype)
        output = model.forward(data)
        loss = nf.cross_entropy(output, label)
        writer.add_scalar('Loss/train', loss, iter_num)
        loss.backward()
        optimizer.step()
        train_loss_list += [loss.item()]
        train_loss += loss.item()
        _, predicted = torch.max(output.data, dim=1)
        _, predicted = output.max(1)
        total += data.shape[0]
        correct += predicted.eq(label).sum().item()

        if batch_index % 5 == 4:
            print('[epoch: %d, batch_idx: %d]: loss: %.3f , acc: %.2f %%'
                  % (epoch + 1, batch_index + 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()
    log[epoch, 0] = np.mean(train_loss_list)
    correct = 0
    total = 0
    model.eval()
    val_loss = []
    with torch.no_grad():
        for batch_index, valdata in enumerate(val_dataloader):
            data, label = valdata['data'].to(device), valdata['label'].to(device)
            data = data.float()
            # for batch_index, (datas, labels) in enumerate(val_dataloader):
            # data, label = datas.to(device), labels.to(device)
            output = model(data)
            _, predicted = torch.max(output.data, dim=1)
            loss = nf.cross_entropy(output, label)
            val_loss.append(loss.item())
            total += label.size(0)
            correct_batch = predicted.eq(label).sum().item()
            correct += predicted.eq(label).sum().item()
            acc_batch = correct_batch / label.size(0)
            print('[batch_index: %d]: Accuracy on val set: %.1f %% ' % (batch_index, 100 * acc_batch))  # 求测试的准确率，正确数/总数
            predicted_list = predicted.tolist()
            targets_list = label.tolist()
            writer.add_scalar('val accuracy per batch', 100 * acc_batch, batch_index)
    acc = correct / total
    print('Average accuracy on val set: %.1f %% ' % (100. * acc))  # 求测试的准确率，正确数/总数
    log[epoch, 1] = np.mean(val_loss)
    log[epoch, 2] = acc
```

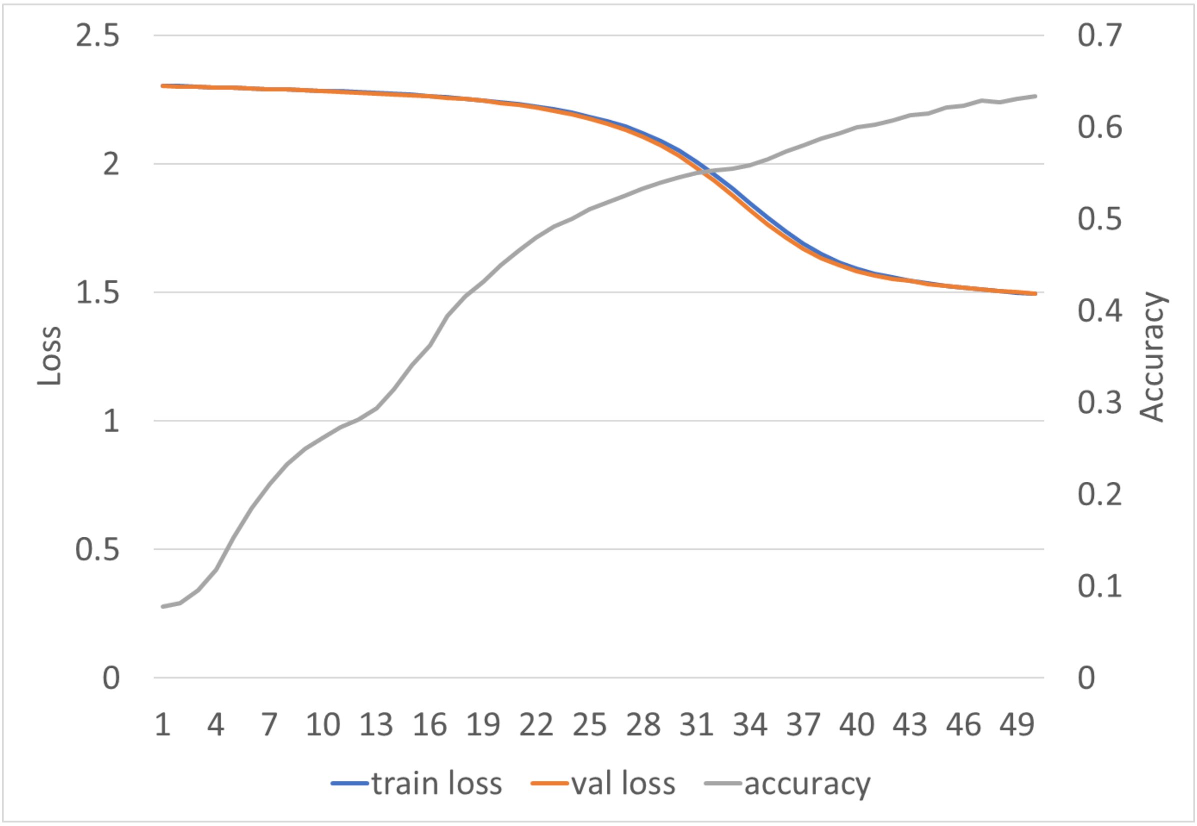
After that, we plot the loss function and validation accuracy in the training and validation process as epochs changes, as follows:

```
from matplotlib import pyplot as plt

figure_x = np.arange(EPOCH)
figure_x += 1
plt.figure(figsize=(5, 3))
plt.plot(figure_x, log[:, 0], linestyle='-', label='train loss', color='r', linewidth=2)
plt.plot(figure_x, log[:, 1], linestyle='-', color='b', label='val loss', linewidth=2)
plt.plot(figure_x, log[:, 2], linestyle='--', color='g', label='val acc', linewidth=2)
plt.title('Training loss and accuracy', fontsize=10)
plt.xlabel('epoch', fontsize=10)
plt.ylabel('value', fontsize=10)
plt.legend(fontsize=10)
plt.ylim(0, 4)
plt.grid()
# plt.savefig('../Results/training_loss_record.png')
plt.show()
with open('../Results/Loss_and_Acc' + version_name + '.csv', 'w') as file:
    # 遍历列表中的元素并将其写入文件
    for i in range(EPOCH):
        file.write(str(i) + ',' + str(log[i, 0]) + ',' + str(log[i, 1]) +
',' + str(log[i, 2]) + '\n')
```

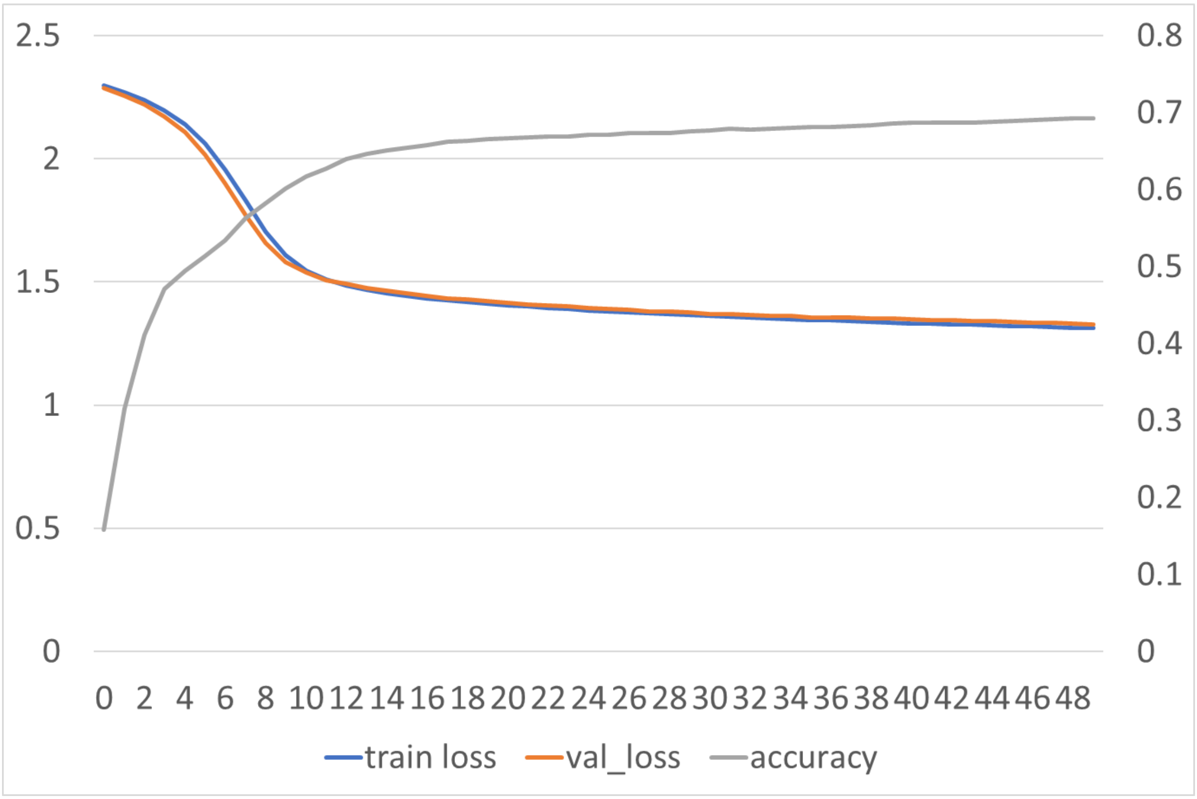
In order to optimize our method, we tried several ways to pre-process the datasets. Since there are ten different layers for one picture, **the first method was to simply sum all ten layers into one, so that the complexity of the data can be decreased.** The training loss, validation loss and average accuracy is shown below.



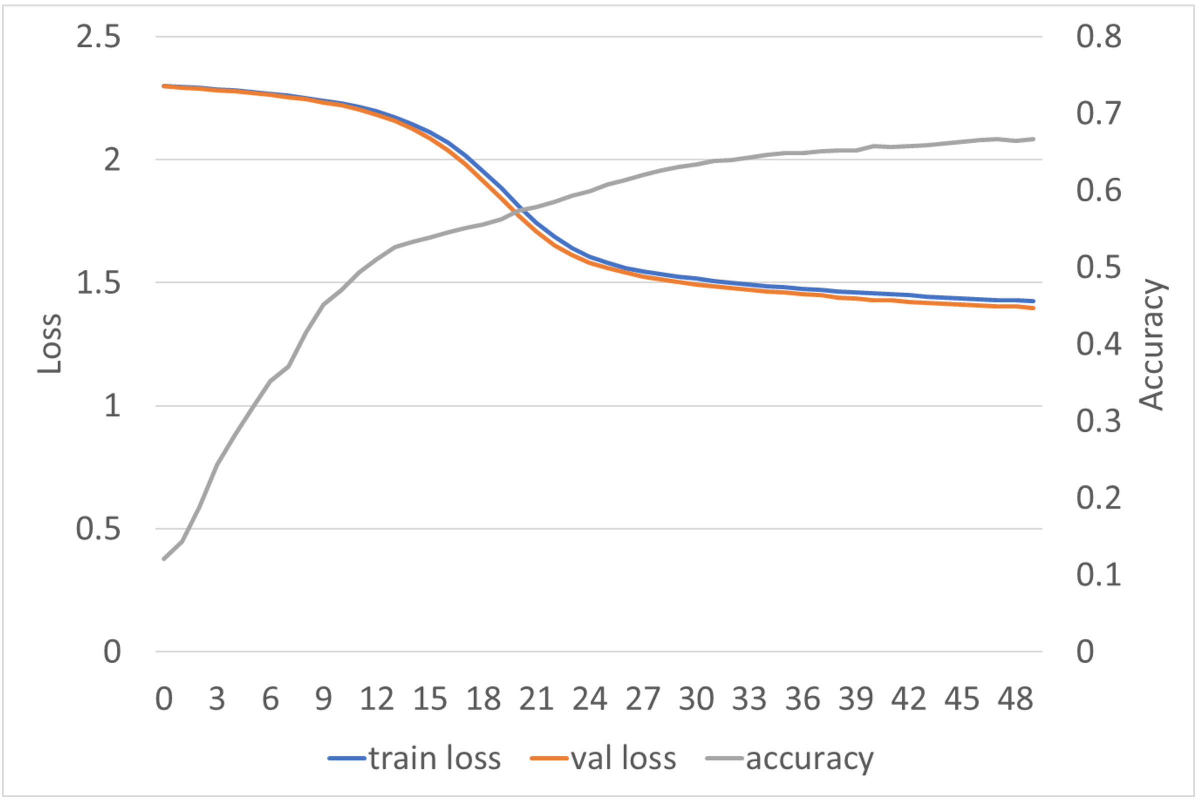
Based on this, since the original size ratio of training set and validation set is 600:1, we tried to mix all data together and separate them into two sets randomly.**The separating ratio will be 9:1, which was more reasonable in learning tasks (This partitioning method was mentioned in Part1).** After re-arranging the datasets, the loss and accuracy status is shown below.



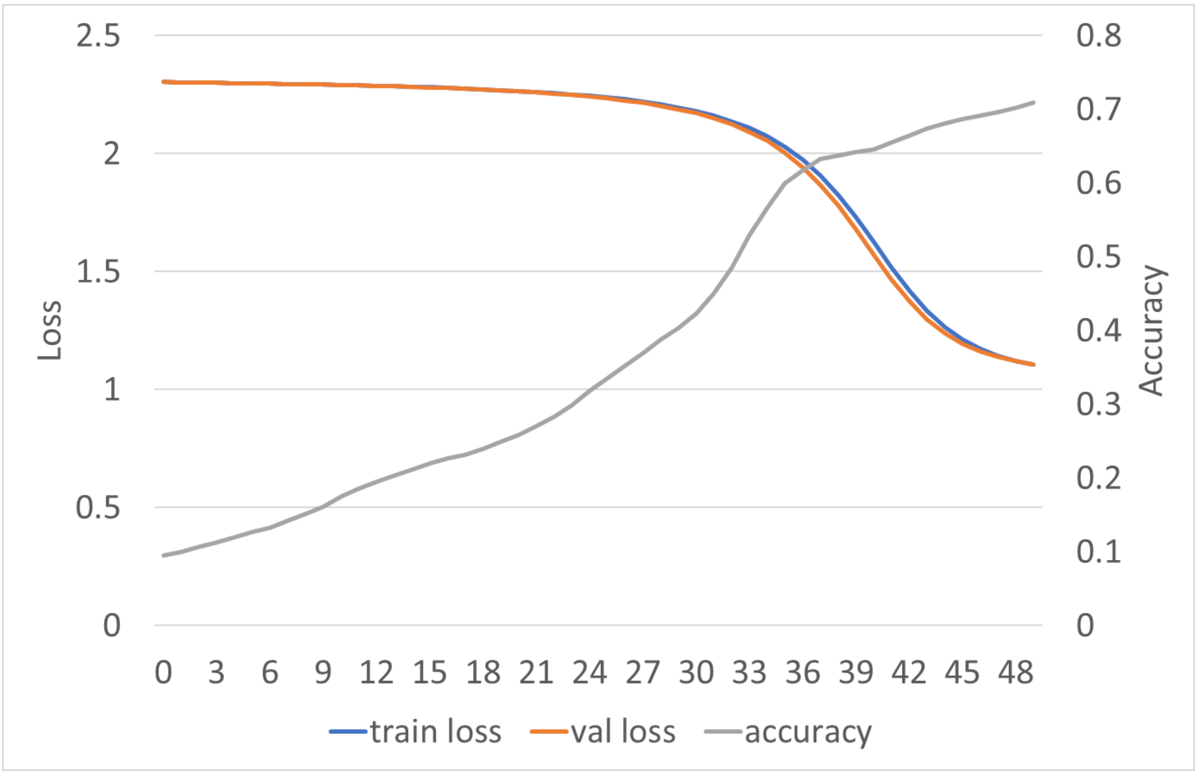
The accuracy seemed not changing much, so we tried further processing. **We calculate each layer's data's average value and standard deviation value, in order to find out all-0 layers and delete them**. For those with actual data, we add them together to form a data with only one layer. Besides, we used mean and std values to normalize all data so that they can share a similar distribution. After that, the the loss and accuracy status is shown below.



The accuracy was increased with this process. **We also tried to binarize data instead of normalize them. The the loss and accuracy status is shown below.**



However, the accuracy is slightly decreased, and the loss is higher. **But still, for research we tried to find out if we can optimize it based on binarization method. We set up a few threshold values in order to identify data that are clearly mis-tagged, so that ideally, the accuracy may be increased (The details of the method for setting the threshold are covered in detail in Part 1).** The results are shown below.



Part 5: Testing Records

After the training and validation process, we **manually label the `.npy` array of the first 512 images in the test set and test the trained neural network by reading these images in a loop**:

```python
import math
# print(data_test_loader.batch_size)
predicted_list = []
count_test = 0
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)
test_files = os.listdir(test_dir)
test_files.sort(key=lambda x: int(x.split('.')[0]))
for file in test_files:
    file_url = os.path.join(test_dir, file)
    # print(file_url)
    this_data = np.load(file_url)
    # 沿着第一个维度相加
    # this_data = np.sum(this_data, axis=0)
    this_data = de_dimension_data(this_data)
    # this_data = np.multiply(this_data, 255.0)
    this_data = torch.tensor(this_data, device=device)
    this_data = torch.unsqueeze(this_data, dim=0)
    this_data = this_data.float()
    this_data = this_data.to(device)
    out = model(Variable(this_data))
    problt = torch.nn.functional.softmax(out, dim=1)
    problt = Variable(problt)
    problt = problt.cpu().numpy()
    pred = np.argmax(problt)
    predicted_list.append(pred.item())
```

During test process, **the accuracy is higher, which means that these method (several ways to pre-process the datasets) mentioned in Part 4 may be working**. In order to compare all these methods, we manually give 512 data in prediction sets their tags, so that we can test if all these methods are really working. The accuracy comparison is shown below.

Several ways to pre-process the datasets	**Sum**	**9:1 Seperate**	**Normalization**	**Binarization**	**Threshold**
Accuracy in 512 examples	0.822266	0.824219	0.898438	0.84375	0.777344

Results shows that although the threshold method shown better performance in training, it can identify fewer labels in random test. The normalization method is the best among all of them, and it can be well explained theoretically. For the binarization and threshold method, the binary distribution may cause the reduction of some features, and the threshold although successfully abandon some mistake data, but it will also abandon some correct data with complexity. In this situation, the model will receive too few complex samples for it to learn, and it will eventually be difficult to identify data with unclear features.

Part 6: Member Contribution

Task	吉辰卿	叶浩腾
Dataset Preprocessing	✗	✓
Model Construction	✓	✗
Training and Testing	✓	✓
Debug	✓	✓
Report Writing	✓	✓
Overall Contribution Percentage	50 %	50 %

Part 7: Conclusion

In this project, we first found the root of the instability by looking at the structure of the data set and visualizing part of the data of the training set and the test set. For the unstable features, we sampled the sum of dimensions to remove the unstable features in the training set and the test set, and **adopted a more complex threshold selection scheme to effectively screen out the image data with wrong labels in the training set, which will improve the stability of model training**. Finally, by manually labeling the test data set, we **verify the influence of different data preprocessing components on the test accuracy using a process similar to the ablation experiment. That is: Re-arraging Datasets, Normalization, Binarization and Thresold filter methods.** By comparing the accuracy of test samples, **normalizing the data in datasets that are rearranged with the ratio 9:1 is the best method we found. In 512 samples, it can correctly predict 89% of them.**

Since both of us are not CS majors and do not have any CV or DL background, more complex model architectures cannot be mastered in the short term. Therefore, this project can be realize the further optimization by transfer learning from training set to test set through a more complex **Transformer structure**.