第二部分：卷积神经网络

计算机科学技术学院　　周训哲 20307110315

2023 年 4 月 13 日

一、CNN 简介

卷积神经网络的基本原理是通过卷积层对输入信号进行特征提取，然后通过池化层对输入进行缩小操作，最后通过线性层和简单的 bp 网络一样将信号进行传输。而训练过程则是使用反向传播算法，目的是最小化损失函数。通过不断调整卷积核的权重和偏置，优化目标识别的效果。

卷积层

卷积核的原理就是使用一个矩阵对矩阵范围内框选到的数据进行加权求和，然后得到一个新的数据，将一系列的数据拼接成一个新的数据作为卷积层的输出，其目的是为了提取区域的特征。卷积层的反向传播原理其实就是利用反向传播算法将输出结果与实际值比较的损失函数反向传播。

正向传播

$$y_k^l = \sigma\left(\sum_{i=1}^{C_{l-1}}\sum_{j=1}^{H_{l-1}}\sum_{k'=1}^{W_{l-1}} w_{i,j,k',k}^l x_{i,j+a,k'+b}^{l-1} + b_k^l\right) \tag{1}$$

反向传播

损失函数对卷积核的梯度：

$$\frac{\partial L}{\partial w_k^l} = \sum_{i=1}^{N} \frac{\partial L}{\partial y_i^l} * x_i^l \tag{2}$$

损失函数对偏置的梯度：

$$\frac{\partial L}{\partial b^l} = \sum_{i=1}^{N} \sum_{j=1}^{H_l} \sum_{k=1}^{W_l} \frac{\partial L}{\partial y_{i,j,k}^l} \tag{3}$$

（其中 L 表示损失函数）

池化层

池化层核的原理就是选取某一区块的特征值进行提取，其目的就是为了缩减输入数据的维度，并且保留原始特征。一般在实际操作过程中，都常用 max_pooling 的方法，即选取某一区块中的最大值作为该区块的特征值进行压缩。而由于池化层操作没有参数，所以不需要进行反向传播的更新操作。

正向传播（以 max_pooling 为例）

$$y_{i,j,k}^l = \max_{p=0}^{P-1} \max_{q=0}^{Q-1} x_{i,jP+p,kQ+q}^{l-1} \tag{4}$$

线性层

线性层一般放在最后，进行对向量的降维和分类。之前卷积层和池化层输入的数据为高维数据，首先要对数据进行展平操作，将数据变成一维的数据，以方便后续线性层的处理。然后线性层的前向和后向传播则是与 bp 神经网络相同，网络中的每个节点包含不同的权值与偏置，通过反向传播算法对权值进行优化以使得最后输出的误差函数尽可能小。

正向传播

$$y_k^L = \sum_{i=1}^{N_{L-1}} w_{i,k}^L x_i^{L-1} + b_k^L \tag{5}$$

反向传播

误差的梯度：

$$\delta_j^L = \frac{\partial L}{\partial y_j^L} \tag{6}$$

权重的梯度：

$$\frac{\partial L}{\partial w_{i,j}^L} = x_i^{L-1} \delta_j^L \tag{7}$$

偏置的梯度：

$$\frac{\partial L}{\partial b_j^L} = \delta_j^L \tag{8}$$

下一层误差的梯度：

$$\delta_i^{l-1} = \sum_{j=1}^{N_l} w_{i,j}^l \delta_j^l \sigma'(z_i^{l-1}) \tag{9}$$

二、代码架构

代码主要分为 model、main、test。其中 model 就是卷积神经网络模型。Main 中包含了数据处理、训练、预测过程。Test 则是利用保存的模型进行预测并输出准确率。

model.py

在 model 模块，主要包含两种模型。一种是 LeNet-5，复现了其每一层的结构，但是原版的输入维度为 32*32，本实验的手写汉字的输入维度为 28*28，所以对线性层的节点个数有所修改，除此以外，包含通道数、卷积核大小、池化层核大小、步长、padding 均保留为原始结构。

```
class LeNet5(nn.Module):
        def __init__(self):
                super(LeNet5, self).__init__()
                # w' = (w + 2 * p - k) / s + 1
                # w: 28 (conv) 24 (pool) 12 (conv) 8 (pool) 4
                self.conv1 = nn.Conv2d(in_channels=1, out_channels=6,
                        kernel_size=5, stride=1, padding=2)
                self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
                self.conv2 = nn.Conv2d(in_channels=6, out_channels=16,
                        kernel_size=5, stride=1, padding=0)
                self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)
                self.fc2 = nn.Linear(in_features=120, out_features=84)
                self.fc3 = nn.Linear(in_features=84, out_features=12)

        def forward(self, x):
                # view(batch_size, in_channels, height, width)
                x = x.view(x.size(0), 1, x.size(1), x.size(2))
                x = self.conv1(x)
                x = nn.functional.relu(x)
                x = self.pool(x)
                x = self.conv2(x)
                x = nn.functional.relu(x)
                x = self.pool(x)
                # 将数据进行展平操作
                x = x.view(x.size(0), -1)
```

```
                x = self.fc1(x)
                x = nn.functional.relu(x)
                x = self.dropout(x)
                x = self.fc2(x)
                x = nn.functional.relu(x)
                x = self.dropout(x)
                x = self.fc3(x)
                return x
```

另一个结构是自己尝试的 cnn 结构，相比 LeNet-5，MyCNN 只使用了两个卷积层和两个池化层，通道数有所增加，1->16->32。除此以外，卷积层的核大小也从 5 改成 3，并且将 padding 改成 1，以保持特征向量的大小不变。线性层只使用了一层，一方面增加了网络的运行效率，另一方面也可以有效防止线性层过多导致的梯度爆炸问题。

```
class MyCNN(nn.Module):
        def __init__(self):
                super(MyCNN, self).__init__()
                # 卷积核处理之后 size 不变
                self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
                        kernel_size=3, stride=1, padding=1)
                self.bn1 = nn.BatchNorm2d(16)
                self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
                self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
                        kernel_size=3, stride=1, padding=1)
                self.bn2 = nn.BatchNorm2d(32)
                self.fc1 = nn.Linear(in_features=32 * 7 * 7, out_features=120)
                self.fc2 = nn.Linear(in_features=120, out_features=12)
                self.fc = nn.Linear(in_features=32 * 7 * 7, out_features=12)
                self.dropout = nn.Dropout(p=0.3)

```

```
17          def forward(self, x):
18                  x = x.view(x.size(0), 1, x.size(1), x.size(2))
19                  x = self.conv1(x)
20                  x = self.bn1(x)
21                  x = nn.functional.relu(x)
22                  x = self.pool(x)
23                  x = self.conv2(x)
24                  x = self.bn2(x)
25                  x = nn.functional.relu(x)
26                  x = self.pool(x)
27                  x = x.view(-1, 32 * 7 * 7)
28                  # x = self.fc1(x)
29                  # x = nn.functional.relu(x)
30                  # x = self.dropout(x)
31                  # x = self.fc2(x)
32                  x = self.fc(x)
33                  return x
```

在激活函数的选择方面，我比较了 relu 函数和 tanh 在 cnn 中的效果，最终选择了准确率更高和选择的人最多的 relu 作为激活函数（是在已经做好优化之后才进行的激活函数的尝试）。

在后续实验过程中，还加入了批标准化（batch_normalization）和 dropout 的优化方法，所以增加了 bn 层和 dropout 层，但是在实验过程中发现增加了 dropout 层之后的准确率反而下降了，所以具体结构并没有使用 dropout，而是只使用了 bn。

main.py

在主函数中主要就是进行数据的预处理和进行神经网络的训练和预测。

首先进行的是数据的预处理，第一步先使用 cv2 库将数据从.bmp 文件转换为 array，并用文件夹名对汉字的类型进行独热编码，第二步是调用 transform 包进行数据增强（由于增强的方式不能够提高学习率，后续对这

一操作进行了改进，将原始的随机裁剪和水平反转改成随机旋转 15° 以内的角度，结果与不进行数据增强的结果差别不大，所以后续操作中不使用数据增强）。第三步是划分训练集与测试集以及设置相关参数值。

```
images = []
labels = []
for label in range(1, 13):
        folder = os.listdir('train\\' + str(label))
        for file in folder:
                path = os.path.join('train\\' + str(label), file)
                img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
                lbl = np.zeros(12)
                lbl[label-1] = 1
                images.append(img)
                labels.append(lbl)
x = np.array(images)
scale_factor = 1 / 255
x = np.multiply(x, scale_factor)
y = np.array(np.vstack(labels))
x = torch.tensor(x).float()
y = torch.tensor(y).float()
# # 数据增强
# transform = transforms.Compose([
#     # 随机裁剪
#     transforms.RandomCrop(size=28),
#     # 水平翻转
#     transforms.RandomHorizontalFlip(),
#     # 随机旋转 15° 以内的角度
#     transforms.RandomRotation(15)
# ])
# x = transform(x)
```

```
data_size = len(images)
test_size = int(data_size / 10)
train_size = data_size - test_size
permutation = np.random.permutation(x.shape[0])
shuffled_x = x[permutation]
shuffled_y = y[permutation]
x_test = shuffled_x[0: test_size]
y_test = shuffled_y[0: test_size]
x_train = shuffled_x[test_size: data_size]
y_train = shuffled_y[test_size: data_size]
learning_rate = 0.001
batch_size = 40
epochs = 10
model_size = 5
```

主函数中进行的操作主要就是调用模型，然后用训练集进行训练和使用测试集进行预测和计算准确率，并且将训练好的模型存入对应的文件中。

模型的训练：

```
if __name__ == '__main__':
        np.random.seed(0)
        models = []
        accuracy_list = []
        error_count = 0.0
        for i in range(model_size):
                cnn = MyCNN()
                criterion = nn.CrossEntropyLoss()
                optimizer = optim.Adam(cnn.parameters(), lr=learning_rate,
                        weight_decay=0.01)
                scheduler = ReduceLROnPlateau(optimizer, mode='max', patience=3)
                # train
```

```
13                acc_list = []
14                permutation = np.random.permutation(x_train.shape[0])
15                shuffled_x_train = x_train[permutation]
16                shuffled_y_train = y_train[permutation]
17                x_model = shuffled_x_train[0: int(2 / 3 * train_size)]
18                y_model = shuffled_y_train[0: int(2 / 3 * train_size)]
19                # x_model = x_train[i * int(train_size / model_size):
20                #         (i + 1) * int(train_size / model_size)]
21                # y_model = y_train[i * int(train_size / model_size):
22                #         (i + 1) * int(train_size / model_size)]
23                for epoch in range(epochs):
24                        cnn.train()
25                        permutation = np.random.permutation(x_model.shape[0])
26                        shuffled_x = x_model[permutation]
27                        shuffled_y = y_model[permutation]
28                        batches = int(len(shuffled_x) / batch_size) + 1
29                        for j in range(0, batches):
30                                x_ = shuffled_x[j * batch_size:
31                                        (j + 1) * batch_size]
32                                y_ = shuffled_y[j * batch_size:
33                                        (j + 1) * batch_size]
34                                optimizer.zero_grad()
35                                y_hat = cnn(x_)
36                                loss = criterion(y_hat, y_)
37                                loss.backward()
38                                optimizer.step()
39                        with torch.no_grad():
40                                error_count = 0.0
41                                y_hat = cnn(x_test)
42                                for j in range(len(y_hat)):
43                                        index = np.argmax(y_hat[j])
```

```
                                        if y_test[j][index] != 1:
                                                error_count += + 1.0
                                accuracy = 1 - (error_count / len(x_test))
                                acc_list.append(accuracy)
                accuracy_list.append(acc_list)
                models.append(cnn)
```

预测结果并存储模型：

```
        # test
        accuracy_list = np.array(accuracy_list)
        accuracy_list = accuracy_list.mean(axis=0)
        plt.figure()
        plt.scatter(range(epochs), accuracy_list, label='accuracy')
        plt.legend()
        plt.show()
        output_list = []
        for cnn in models:
                cnn.eval()
                with torch.no_grad():
                        error_count = 0.0
                        y_hat = cnn(x_test)
                        output_list.append(y_hat)
        output_avg = torch.mean(torch.stack(output_list), dim=0)
        for i in range(len(output_avg)):
                index = np.argmax(output_avg[i])
                if y_test[i][index] != 1:
                        error_count += + 1.0
        accuracy = 1 - (error_count / len(x_test))
        for i in range(len(models)):
                torch.save(models[i], "models\\model%s.pth" % i)
```

```
23          print('accuracy: ', accuracy)
```

test.py

在测试文件中主要就是 load 已经训练好的模型，然后使用模型测试准确率。

```
1  if __name__ == '__main__':
2          np.random.seed(0)
3          error_count = 0.0
4          output_list = []
5          for i in range(model_size):
6                  cnn = torch.load("models\\model%s.pth" % i)
7                  cnn.eval()
8                  with torch.no_grad():
9                          error_count = 0.0
10                         y_hat = cnn(x)
11                         output_list.append(y_hat)
12         output_avg = torch.mean(torch.stack(output_list), dim=0)
13         for i in range(len(output_avg)):
14                 index = np.argmax(output_avg[i])
15                 if y[i][index] != 1:
16                         error_count += + 1.0
17         accuracy = 1 - (error_count / len(x))
18         print('accuracy: ', accuracy)
```

三、改进方法及算法原理

LeNet-5

最开始使用的模型就是经典的 LeNet-5 模型，由 Yann LeCun 发明。

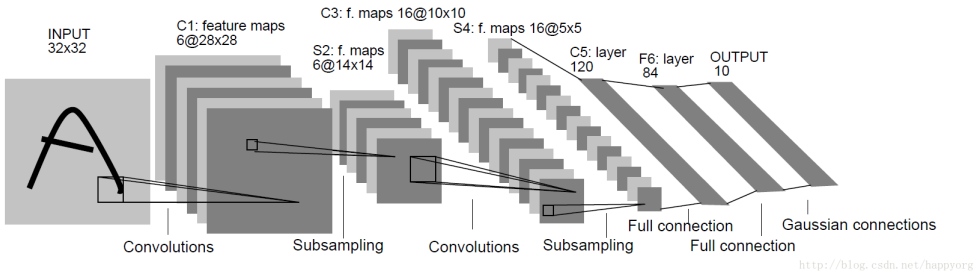

图 1: LeNet-5 的网络结构

按照网络的构思，其输入为 32*32 的图片，但是手写汉字只有 28*28，所以为了保证所有的特征笔画都处于卷积核的中心，可以使用卷积核中的 padding 操作对图像边缘进行填充。

Adam

首先加的第一个优化就是最出名的 adam 优化。其目的是使得梯度优化的过程能够尽可能的快，并且等到数据误差小的的时候会缩小梯度变化的尺度使得尽可能的逼近局部最优，而不至于在局部最优处反复震荡。

其最初是由 SGD（随机梯度下降）演化而来，SGD 梯度更新公式为：

$$w = w - \eta \nabla L(w) \tag{10}$$

由于 SGD 不能很好的优化梯度的变化力度，容易增加梯度下降所需要的时间，之后发展为 Adagrad（自适应梯度算法），可以随着时间变化逐渐缩小梯度的变化力度：

$$w = w - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{11}$$

为了应对 adagrad 算法在更新过程中逐步缩小梯度变化，导致最后难以应对不稳定目标函数的情况，所以 RMS 算法提出了一个遗忘因子 β，能够尽可能的减少最开始的梯度变化带来的影响：

$$w = w - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t \tag{12}$$

而 Adam 优化器则是对 RMS 又进一步进行了优化，对梯度的变化也使用了遗忘因子 β，使得优化器能够应对更加不稳定的目标函数，计算更加高效：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{14}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{15}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{16}$$

$$w = w - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t \tag{17}$$

batch normalization

在之后我还加上了一个批标准化的优化，一方面可以使得每一层输入的值都是有效的值，不会出现巨大的偏差，另一方面可以有效防止梯度消失、过拟合等现象的发生。

其实现原理为在每一层之后都加入一个批标准化层，计算上一层输出的均值和标准差，对输出的数据进行以 batch 为单位的标准化。

正向传播

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{18}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{19}$$

反向传播

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i} \cdot \gamma \tag{20}$$

$$\frac{\partial L}{\partial \sigma_B^2} = \sum_i \frac{\partial L}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot (-\frac{1}{2}(\sigma_B^2 + \epsilon)^{-\frac{3}{2}}) \tag{21}$$

$$\frac{\partial L}{\partial \mu_B} = \frac{\partial L}{\partial \hat{x}_i} \cdot (-\frac{1}{\sqrt{\sigma_B^2 + \epsilon}}) + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{\sum_i -2(x_i - \mu_B)}{N} \tag{22}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{N} + \frac{\partial L}{\partial \mu_B} \cdot \frac{1}{N} \tag{23}$$

$$\frac{\partial L}{\partial \gamma} = \sum_i \frac{\partial L}{\partial y_i} \cdot \hat{x}_i \tag{24}$$

$$\frac{\partial L}{\partial \beta} = \sum_i \frac{\partial L}{\partial y_i} \tag{25}$$

ensemble

集成算法的思想和投票类似，就是通过训练多个弱的训练模型，然后通过一定的策略将这些模型进行组合，得到一个更强的模型。

具体的算法有很多，如最基础的 averaging、voting，到相对进阶的 bagging、boosting、stacking：

Averaging 的思想就是用相同的数据集训练出不同的模型，然后简单的对不同预测的取平均，按理来说，预测概率越大的输出，取平均之后的结果仍然很大，所以这个方法可以有效消除单个模型带来的误差，由于该方法比较容易实现，在最开始使用集成时就使用的该方法。类似 averaging，由于不同模型的准确度有所差异，所以在求值时可能还要加上一定的权值取平均。

Voting 的思想就是比较不同模型对同一个数据的判断，如果输出为同一个结果的比例大于一定阈值，则将输出该值。

Bagging 的想法就是利用一个数据集中的不同子数据集对不同的子模型进行训练，这些数据集可能存在交叉，然后分别训练这些子模型，最终集成得到一个更好的强训练模型。

Boosting 的想法就是通过反复学习得到一系列弱分类器，然后对这些弱分类器进行组合，最后得到一个强分类器。主要思路是，模型的一方面来自于上一个模型，另一方面来自于之前所有模型的加权总和。

Stacking 的想法是首先用一部分数据训练出几个基模型，然后用另一部分数据放入模型当中进行预测，得到预测结果之后又将结果作为训练集对原来总的模型进行训练，这样得到的模型将会是一个整体。

在实现过程中，我一开始使用的是 averaging 方法，在训练集的选择上，一开始使用的是全部的训练集作为输入，这样方便实现，并且在输出结果的时候就是简单的输入为多个模型输出的均值。

在后续改进过程中使用了 bagging 方法，每次随机的 2/3 的训练集作为输入，准确率有所提升。

随后尝试使用了 boosting 方法，每次使用一个 weight 对不同准确率的模型进行加权，如果准确率低，则 loss 的权重更高，输出权重更低。每次训练都继承前一次的模型，不断训练迭代。

之后尝试使用了 staking 集成方法，首先训练一系列模型并得到预测结果，然后将所有模型的预测结果进行整合作为集成模型的训练集进行输入，然后训练一个多层感知机模型作为元模型作为一个整体的模型。

数据增强

数据增强的原理就是对输入数据进行平移、压缩、旋转等方法使得数据更加多样，以增加模型的泛化能力，防止过拟合。在实现过程中可以直接调用 python 中的 transform 的库函数。

```
# 数据增强
transform = transforms.Compose([
        # 随机裁剪
        transforms.RandomCrop(size=28),
        # 水平翻转
        transforms.RandomHorizontalFlip(),
        # 随机旋转 15° 以内的角度
        transforms.RandomRotation(15)
```

```
9 ])
10 x = transform(x)
```

dropout

Dropout 方法的思想类似于集成方法，在训练过程中，随机裁剪掉层内的一些节点，然后每次训练时裁剪掉的节点都不相同（可能会有重合），然后最后在测试时又将所有的节点释放，对数据进行预测。这样做一方面可以防止过拟合，增加模型的泛化能力；另一方面可以通过类似投票的方式提高模型的准确度。

自适应学习率

自适应学习率的方法有点类似于 adam 优化器，即在训练过程中，当模型的准确度不再上升时，说明可能已经达到局部最优，也可能到达欠拟合和过拟合的边界，这时候每训练一次就将学习率降低为原来的 1/10，以逐步逼近局部最优，避免在最有附近震荡。

四、防止过拟合的方法（bonus）

以上方法为实验中尝试了提高准确率的方法。其中能够防止过拟合现象发生的方法包括批标准化、模型集成、数据增强、dropout 方法、都可以通过泛化模型来达到防止过拟合的目的。而自适应学习率则是通过调整训练次数过多以后的学习率，避免训练次数过多而导致梯度下降之后的权重和偏置过于拟合训练数据，从而避免过拟合现象降低准确率。从这个角度出发，adam 优化器也可以通过调整梯度变化的大小一定量的避免过拟合现象的发生。

五、网络改进实验

了解了以上方法之后，便是设计实验论证以上方法的有效性。

第一轮优化

最开始使用的模型就是一个简单的 LeNet-5 模型，在不使用任何优化的情况下，结果表现非常差，只有个位数的准确率。

然后尝试使用了 adam 优化器，效果十分好，在只训练 10 次的情况下直接将准确率提升到 79%，然后在 100 次的情况下，准确率达到 96.6%，在训练 1000 次的情况下，准确率达到 98.3%。

模型	LeNet-5	LeNet-5	LeNet-5	LeNet-5
优化方法（递进）	none	Adam	Adam	Adam
测试轮数	10	10	100	1000
准确率	0.073	0.79	0.966	0.983

在此之下改进了一下网络结构，在之前的基础之上，增加了卷积层的通道数使得特征值更多，并且增加了 padding 使得所有的特征值都聚在卷积核的中间，减小了卷积核的大小，使得经过卷积层之后仍然能保持更多的特征值，最后减小了线性层的数量，避免发生梯度爆炸的现象。改进了网络结构之后，准确率也有所提升，训练 10 次，准确率就达到了 90.4%，训练 100 次就达到了 98.1%，训练 1000 次就达到了 98.7%。

模型	MyCNN	MyCNN	MyCNN
优化方法（递进）	Adam	Adam	Adam
测试轮数	10	100	1000
准确率	0.904	0.981	0.987

说明 为了提高测试的效率，之后改进方法的训练次数都只使用 10 次，并且网络结构确定使用自定义的网络结构。

接下来测试的是批梯度下降的方法，效果明显，在只训练 10 次的情况下就能够达到 95.6% 的准确率，说明每一层之后进行标准化数据确实能够有效提高准确率。

然后使用的就是集成方法。我主要使用的是 bagging 方法，在训练集的选择上，一开始使用的是全部的训练集作为输入，这样方便实现，并且在输出结果的时候就是简单的输入为多个模型输出的均值。这样做使得准确率提高到了 97.3%。

然后尝试使用了数据增强技术。在实现过程中，一开始使用了裁剪和反转的方法，但是以上的方法并不适用于手写汉字识别，因为会破坏汉字原本的结构，准确率没有太大的变化，最高只有 97.2%，会比不用数据增强的效果更差。所以在第一次实验中并没有增加数据增强操作。

然后在网络中尝试 dropout 方法，由于只有一层线性层，所以将 dropout 层放在第二个卷积层之后，结果反而降低了，只有 96.1%。可能是因为 dropout 方法在使用时不能用于卷积层之后，卷积层之间的相邻的像素包含相关的信息，而删除这些节点则可能导致传播过程收到影响。所以在一开始并没有使用 dropout 方法。

最后一个优化是对学习率的优化，使用了 ReduceLROnPlateau 函数能够自适应的调整学习率，设置 3 轮之内准确率没有上升则将学习率降低为原来的 1/10，以防止过拟合的发生，最后的准确率也成功提高到 98.2%。这个也是优化当中最高的准确率。

优化方法（递进）	批梯度下降	集成	数据增强	dropout	自适应学习率
准确率	0.956	0.973	0.972	0.961	0.982

后续所有工作将在之前的优化当中又进行优化。

第二轮优化

首先对网络结构进行了尝试优化，由于之前在 bp 网络中使用的是 tanh 函数，所以 cnn 也尝试使用，结果明显不如 relu 的表现，为 95.4%，可能是因为对于卷积核，使用更大的梯度可以使其更好的收敛。所以之后的网络仍然沿用 relu 作为激活函数。

接着对数据增强进行了改进，将原来的方法改成了使用旋转的方法。随机旋转 15 度以内的角度以增加数据的多样性，结果的效果和原始比较差别不大，甚至有所下降，为 97%。

然后对集成方法进行了改进，首先使用的方法是将数据分割成很多块，然后分别对不同块的数据进行学习，得到若干个模型，最后将训练的结果加和，这样的准确率可以达到 98.3%。从原来初始的使用整个训练集训练改成了使用随机的 2/3 的训练集作为输入，准确率提升到了 99.3%。

优化方法（递进）	tanh 激活函数	数据增强（只使用旋转）	集成（用数据分开训练）
准确率	0.954	0.97	0.983

为了使用 dropout 操作，并且同时为了避免单层网络降维过快导致的参数爆炸，将线性层进行了扩展，使得原来的单个线性层变成了两个线性层，从原来的 32*7*7->12 中间加了一层含 120 个节点的线性层。结果可以达到 92%，较之前有所提升，在使用 dropout 操作后，准确率可以达到 98.9%。

优化方法（递进）	集成（随机采用总数据 2/3）	增加线性层	增加 dropout 层
准确率	0.993	0.92	0.989

说明 后续经过测试，当使用随机采样集成方法时，随着训练次数的增加，其准确率反而降低，说明训练次数过少很可能出现采样的随机性，导致准确率很高或很低（上面训练 10 次的准确率取的是最高的值），但是随着次数增加，模型泛化能力增强，准确率会回归到一个合理的区间。

补充 在实验的最后，我最后尝试了其他几种集成的方法，包括调用随机森林的 bagging 方法，具体来说就是利用不同的子训练集训练多个模型，然后将多个模型的训练集预测输出作为输入放入到随机森林中进行再一次的集成训练，主要实现类似 stacking 方法将多个模型整合为一个模型的目的。这个模型可以很快的收敛，效果也很好，但是验收时的表现却不是很好，推测可能是训练次数和参数过多导致出现过拟合的问题。然后自己手动实现了 boosting 的集成方法，思路就是利用每次的训练结果的误差可以计算得到各个模型的训练和输出权重，误差越大，下一次训练的权重，也即力度也越大，可以加速收敛过程，同时输出的权重也越小，保证成功率。同时还加入记忆系数，保证对之前的训练误差导致的权重变化有所记忆。这

种集成方法的表现不算优异，但是还是表现良好。最后没有调用集成模型，只是单独使用线性模型作为 metamodel 实现了 stacking 集成方法，由于线性层过于简单，导致会出现梯度爆炸问题，但是同时使用的网络层数较浅，很难进行改进，所以得出结论，对于本模型，使用简单的线性 metamodel 很难得到较为理想的结果。

在实现这一部分过程时，主要时间花在研究模型构造以及方法的具体实现上，虽然结果甚至没有之前的实验效果好，但是还是通过这样的实践学习到了很多深度学习中的集成方法。

六、对网络设计的理解

相比于简单的 bp 全连接神经网络，卷积神经网络更加适合于图像识别，因为其算法中的卷积核天然自带多维向量的特征提取的功能，所以很快就能够达到比简单的全连接神经网络更高的准确率。但是如何进一步提高 cnn 的准确率则需要诸多的优化方法。

在优化过程中，我深刻体会到优化方法对网络设计的重要性。一个好的优化方法可以极大提高网络的准确性和泛化性，同时也能够提升训练和学习的效率。在设计过程中，我尝试了很多方法，其中有的方法能够使准确率有很大提升，有的方法则比较不适用于当前的模型，同时又有的方法之间能够相互作用共同提高识别的准确度，这些都需要通过实验进行验证。网络设计不仅是对方法的掌握，还是一门实验科学，需要不断的尝试，结合实际的模型进行验证才能够得到一个最优的算法模型。