A3-卷积神经网络实验报告

一、任务目标

1、基本实验要求完成：

a. **实现卷积神经网络分类器**

b. **应用dropout和多种normalization方法，理解它们对模型泛化能力的影响**

c. **理解如何通过交叉验证，为神经网络找到最好的hyperparameters**

2、进阶要求：

a. 在训练网络的过程中，可根据需要自由尝试其它提升性能的方法，例如通过增加模型层数、使用不同的正则化方法、使用模型集成等

二、实验实现过程：

1、实验平台概述：

本次实验采用 Python 完成。

依赖库有：

依赖库	版本
numpy	1.24.3
torchvision	0.14.0+cu116
torch	1.13.0+cu116
sklearn	0.0

2、实验代码框架概述及实现原理：

1. 执行器(Runner)设计：

为方便运行，完成本次实验，先进行设计了一个可通用在各模型上使用的执行器，其包含使用模型的各个重要步骤包含初始化、训练、验证、测试等多个方法。

- `__init__`: `Runner` 类的初始化函数，用于设置模型的训练参数。

```
class Runner:
    def __init__(self, module : nn.Module, batch_size=256, num_workers=8,
                 epochs=15, lr=1e-3, resize=None, device='cuda', set_model_name=
                 datasets='CIFAR-10', weight_decay=0, kf=True,
                 optimizer:torch.optim.Adam = None) -> None:
        self.module = module.to(device)
        self.device = device
        self.batch_size = batch_size
        self.epochs = epochs
        self.Kf=kf
        self.folders=10

        if datasets == 'CIFAR-10':
            self.train_iter, self.test_iter, self.train_set = dataset_make(batch
        else:
            self.train_iter, self.test_iter, self.train_set = minist_dataset_mak

        if self.Kf == False:
            self.val_iter = self.test_iter
        else:
            self.val_iter = None
            .......
```

- `_get_train_result`, `_get_valid_result`, `_get_test_result`:这些函数用于获取训练、验证和测试阶段的结果。
- `outputs_metric`:用于计算模型输出的评价指标。
- `train_steps`, `test_steps`:这些步骤是训练和测试过程的具体执行函数。

```
    def train_steps(self, X:torch.Tensor, Y:torch.Tensor):
        self.optimizer.zero_grad()
        X, Y = X.to(self.device), Y.to(self.device)
        X.requires_grad_()
        Y_c = []
        yy = [0,0,0,0,0,0,0,0,0,0]
        for y in Y:
            yy = [0,0,0,0,0,0,0,0,0,0]
            yy[y] = 1.0
            Y_c.append(yy)
        Y_c = torch.tensor(Y_c, device=self.device,requires_grad=True)
        y_hat = self.module(X)
        loss = self.loss(y_hat, Y_c)
```

```
            loss.backward()
            self.optimizer.step()

            train_score = self.outputs_metric(y_hat, Y)
            return train_score, loss.item()

        @torch.no_grad()
        def test_steps(self, X, Y):
            X, Y = X.to(self.device), Y.to(self.device)
            Y_c = []
            yy = [0,0,0,0,0,0,0,0,0,0]
            for y in Y:
                yy = [0,0,0,0,0,0,0,0,0,0]
                yy[y] = 1.0
                Y_c.append(yy)
            Y_c = torch.tensor(Y_c, device=self.device)
            y_hat = self.module(X)
            loss = self.loss(y_hat, Y_c)
            test_score = self.outputs_metric(y_hat, Y)

            return test_score, loss.item()
```

- `train`, `val`, `test`：这些函数用于启动训练、验证和测试过程。

```
        def train(self):

            # 使用K折交叉验证 - epoch=1
            if self.Kf:
                kf = KFold(n_splits=10, shuffle=True, random_state=0)
                folders = 0
                torch.save(self.module.state_dict(), './lenet_origin.pth')
                for train_index, val_index in kf.split(self.train_set):
                    self.module.load_state_dict(torch.load('./lenet_origin.pth'))
                    self.optimizer = torch.optim.Adam(self.module.parameters(), lr=s

                    train_fold = torch.utils.data.dataset.Subset(self.train_set, tra
                    val_fold = torch.utils.data.dataset.Subset(self.train_set, val_i
                    self.train_iter = DataLoader(dataset=train_fold, batch_size=self
                    self.val_iter = DataLoader(dataset=val_fold, batch_size=128, shu

                    self.module.train()
                    train_loss = 0.0
                    train_acc = 0.0
                    for i in range(self.epochs):
                        with tqdm(total=len(self.train_iter)) as t:
```

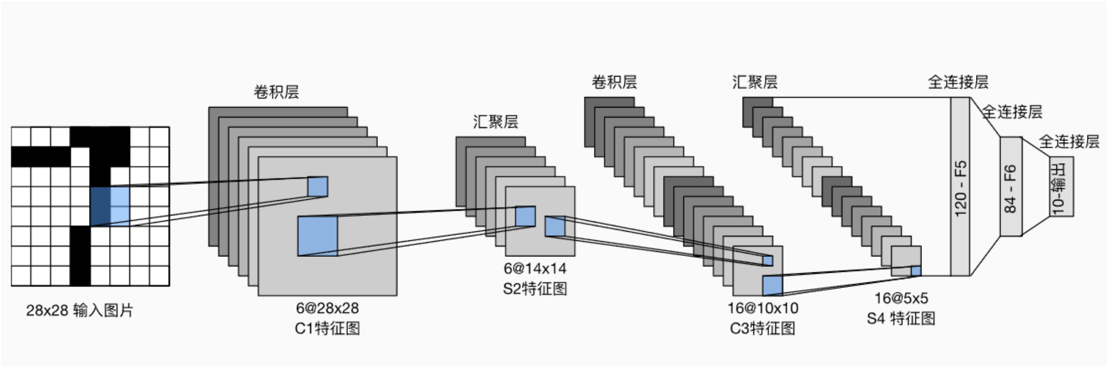
```python
                    for idx, (X, Y) in enumerate(self.train_iter):
                        t.set_description("Fold: %i"%folders+" Epoch: %i"%i)
                        a, b = self.train_steps(X, Y)
                        t.set_postfix(train_loss='%.4f'%b,train_acc='%.4f'%a
                        train_loss += b
                        train_acc += a
                        t.update(1)
                self.train_acc.append(train_acc/len(self.train_iter))
                self.train_losses.append(train_loss/len(self.train_iter))

            self.module.eval()
            self.val()

            folders += 1
    else:
        for i in range(self.epochs):
            self.module.train()
            train_loss = 0.0
            train_acc = 0.0
            with tqdm(total=len(self.train_iter)) as t:
                for idx, (X, Y) in enumerate(self.train_iter):
                    t.set_description("Epoch: %i" %i)
                    a, b = self.train_steps(X, Y)
                    t.set_postfix(train_loss='%.4f'%b,train_acc='%.4f'%a)
                    train_loss += b
                    train_acc += a
                    t.update(1)
            self.train_acc.append(train_acc/len(self.train_iter))
            self.train_losses.append(train_loss/len(self.train_iter))
            self.module.eval()
            self.val()
    self.test()
```

- `get_model_name`：函数用于生成或获取模型的名称

```python
def get_model_name(self):
    if self.set_name:
        return self.set_name

    model_type = self.module.__class__.__name__
    return model_type
```

2. LeNet:

LeNet初始论文中网络结构如下：

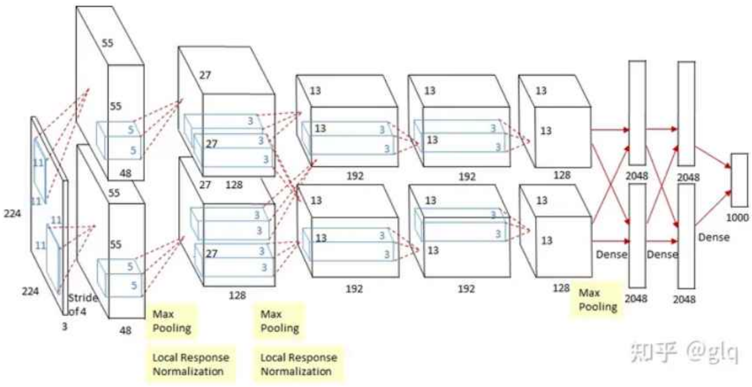


但在实现中，添加了softmax层作为最后的输出层， 进行了微小的改动。

```python
class LeNet(nn.Module):
    def __init__(self, in_dims=3, droup_rate=0.0) -> None:
        super().__init__()
        self.net = nn.Sequential(
        nn.Conv2d(in_dims, 6, kernel_size=5, padding=2), nn.Sigmoid(),
        nn.AvgPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
        nn.AvgPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 6 * 6, 120), nn.Sigmoid(),
        nn.Dropout(p=droup_rate),
        nn.Linear(120, 84), nn.Sigmoid(),
        nn.Linear(84, 10),
        nn.Softmax())

        for layer in self.net:
            if type(layer) == nn.Linear or type(layer) == nn.Conv2d:
                nn.init.xavier_uniform_(layer.weight)

    def __call__(self, X):
        return self.forward(X)


    def forward(self, X):
        return self.net(X)
```

3. AlexNet:

   AlexNet结构如下：



在实现中因为使用的数据集与原论文不同，将最后的分类数更改为了10 (MINIST, CIFAR-10)，并将图片数据填充至224x224大小后进行输入。

```
class AlexNet(nn.Module):
    def __init__(self, in_dims=3,droup_rate=0.5, BN=False) -> None:
        super().__init__()
        if BN:
            self.net = nn.Sequential(
                        nn.Conv2d(in_dims, 96, kernel_size=11, stride=4, padding
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.BatchNo
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.BatchN
                        nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.BatchN
                        nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.BatchN
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Flatten(),
                        #使用dropout层来减轻过拟合
                        nn.Linear(6400, 4096), nn.BatchNorm1d(4096), nn.ReLU(),
                        nn.Dropout(p=droup_rate),
                        nn.Linear(4096, 4096), nn.BatchNorm1d(4096), nn.ReLU(),
                        nn.Dropout(p=droup_rate),
                        nn.Linear(4096, 10))
        else:
            self.net = nn.Sequential(
                        nn.Conv2d(in_dims, 96, kernel_size=11, stride=4, padding
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU()
                        nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU()
                        nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU()
                        nn.MaxPool2d(kernel_size=3, stride=2),
                        nn.Flatten(),
                        #使用dropout层来减轻过拟合
                        nn.Linear(6400, 4096), nn.ReLU(),
```

```
                        nn.Dropout(p=droup_rate),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Dropout(p=droup_rate),
                        nn.Linear(4096, 10))

        for layer in self.net:
            if type(layer) == nn.Linear or type(layer) == nn.Conv2d:
                nn.init.xavier_uniform_(layer.weight)


    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):
        return self.net(X)

```

4. ResNet18 & ResNet50:

对于ResNet网络，首先实现其最基本的残差块作为基础结构。

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

```

```
        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

根据ResNet50所需，实现BottleNeck作为其基础构成部分：

```
class Bottleneck(nn.Module):
    expansion = 4#

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, stride=stride, b
        self.bn1 = nn.BatchNorm2d(planes)  # 归一化处理，使得不会因数据过大而导致网络
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
                               padding=1, bias=False)#block的中间层卷积
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)#bl
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample#判断是否是conv block
        self.stride = stride#不同stage的stride不同，除了stage1的stride为1，其余stag

    def forward(self, x):
        residual = x
        # 卷积操作，就是指的是identity block
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)
        if self.downsample is not None:
            residual = self.downsample(x)
        # 相加
        out += residual
        out = self.relu(out)

```

```python
        return out
```

最后整体实现ResNet类，根据结构需要调用组合：

```python
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10, in_dims=3):  # block即为Bo
        self.inplanes = 64  # 初始输入通道数为64
        super(ResNet, self).__init__()
        # 把stage前面的卷积处理
        self.conv1 = nn.Conv2d(in_dims, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=0, ceil_mod

        # 64, 128, 256, 512是指扩大4倍之前的维度
        # 四层stage,layer表示有几个block块,可见后3个stage的stride全部为2
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)

        ....................
```

ResNet结构根据论文有：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

其中ResNet18,ResNet34由基本的残差块构成， ResNet50等由BottleNeck块构成，故构建其模型可用：

```python
resnet50 = ResNet(Bottleneck, [3, 4, 6, 3])
resnet18 = ResNet(BasicBlock, [2, 2, 2, 2])
```

与上述内容对照即可。

5. Average 模型聚合类实现:

```python
class AGGNet(nn.Module):
    def __init__(self, nets:List[nn.Module]) -> None:
        self.nets = nets
        self.params = [{"params": net.parameters()} for net in nets] # 各模型参数


    def parameters(self, recurse: bool = True):
        return self.params

    def train(self, mode: bool = True):
        for net in self.nets:
            net.train()

    def eval(self):
        for net in self.nets:
            net.eval()

    def to(self, params):
        for net in self.nets:
            net.to(params)
        return self

    def __call__(self, X):
        return self.forward(X)

    def forward(self, X):    # 求各网络结果平均值进行聚合
        outputs = []
        for net in self.nets:
            output = net(X)
            outputs.append(output)

        out = torch.zeros_like(outputs[0], device=X.device, requires_grad=True)
        for output in outputs:
            out = out + output

        out /= len(outputs)

        return out
```

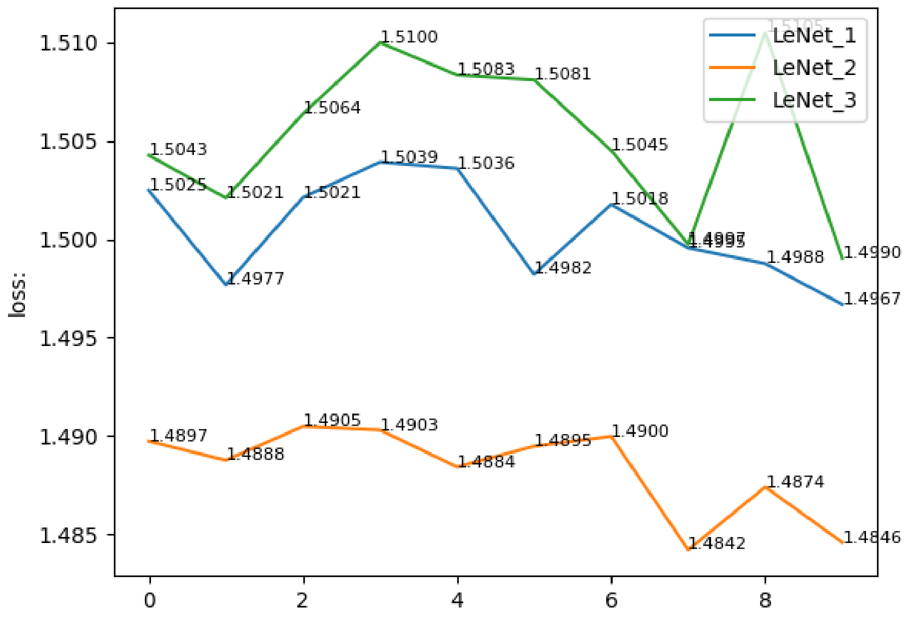
对于该聚合模型类，主要针对于执行器所需接口进行部分重构，并主要对forward()行为进行定义，将结果的平均值作为主要的输出。

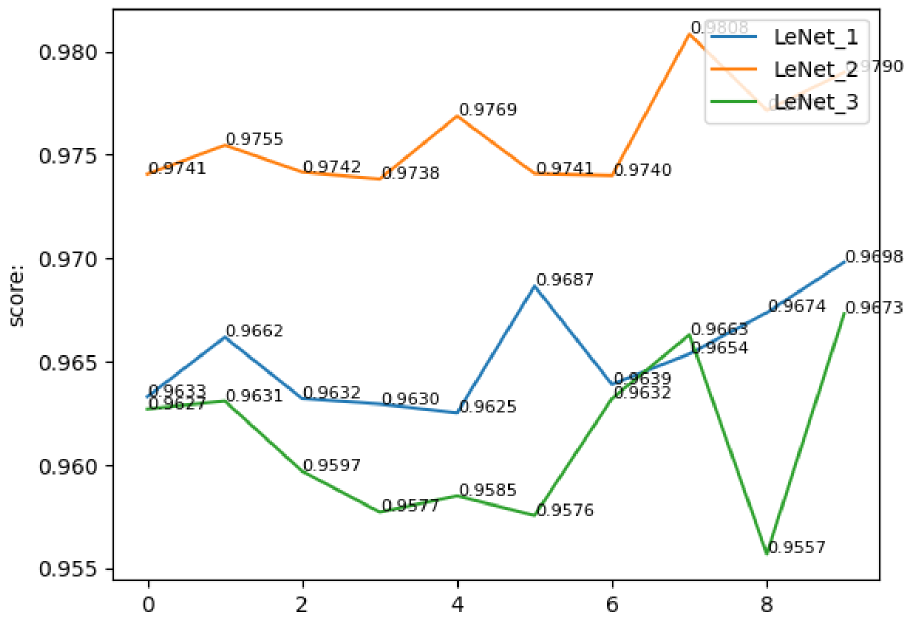
3、具体实验步骤以及步骤结果分析：

① LeNet交叉验证参数调试实验(on MINIST)：

1. batch-size，学习率调试

参数/模型名	LeNet_1	LeNet_2	LeNet_3
epochs	10	10	10
lr	1e-3	1e-3	5e-4
batch_size	256	128	128
droup_rate	0.0	0.0	0.0

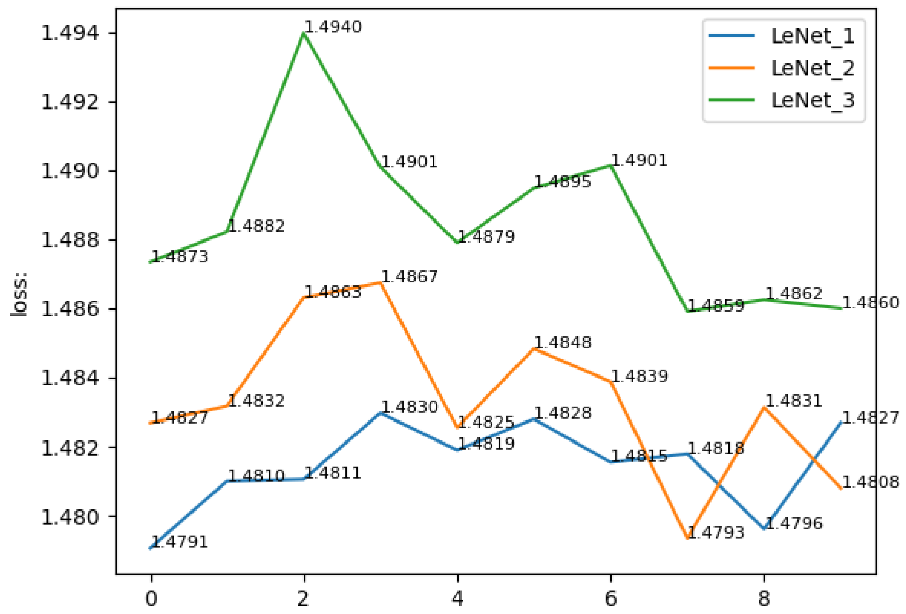


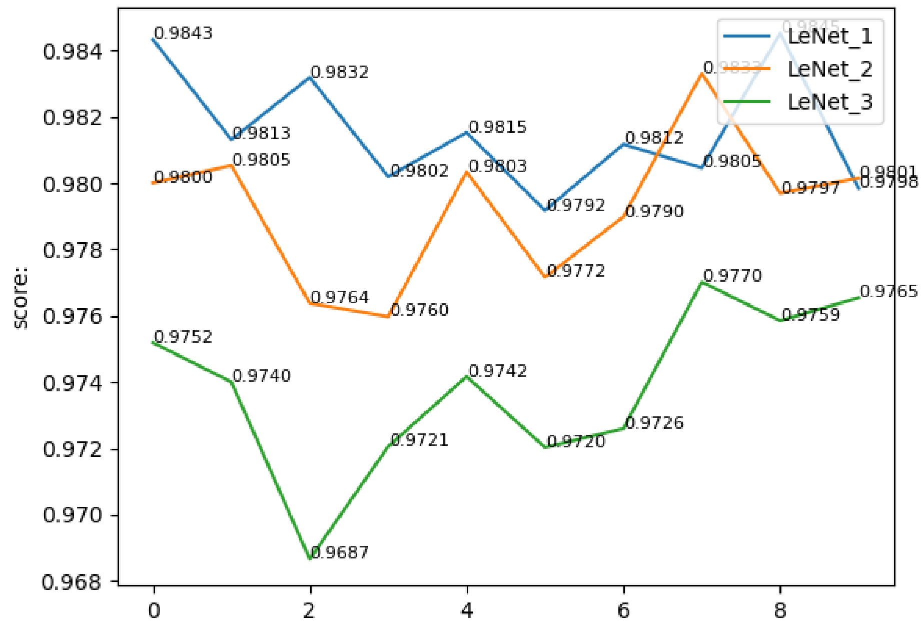


上述图片为10折交叉验证训练下，验证损失图与验证精度图。由图分析容易得知，LeNet_2对应的超参数更适合进行训练得到更佳的测试结果，其精度更高，损失下降更低。

2. Droupout丢弃率调试

参数/模型名	LeNet_1	LeNet_2	LeNet_3
epochs	30	30	30
lr	5e-4	5e-4	5e-4
batch_size	128	128	128
droup_rate	0.0	0.2	0.5





在该次Droupout丢弃率调试中发现，不在全连接层中使用Droupout更能取得稳定，优异的表现，可能是因为LeNet中全连接层神经元个数较少，拟合能力没有过强有关。

② AlexNet网络正则化实验(on MINIST)：

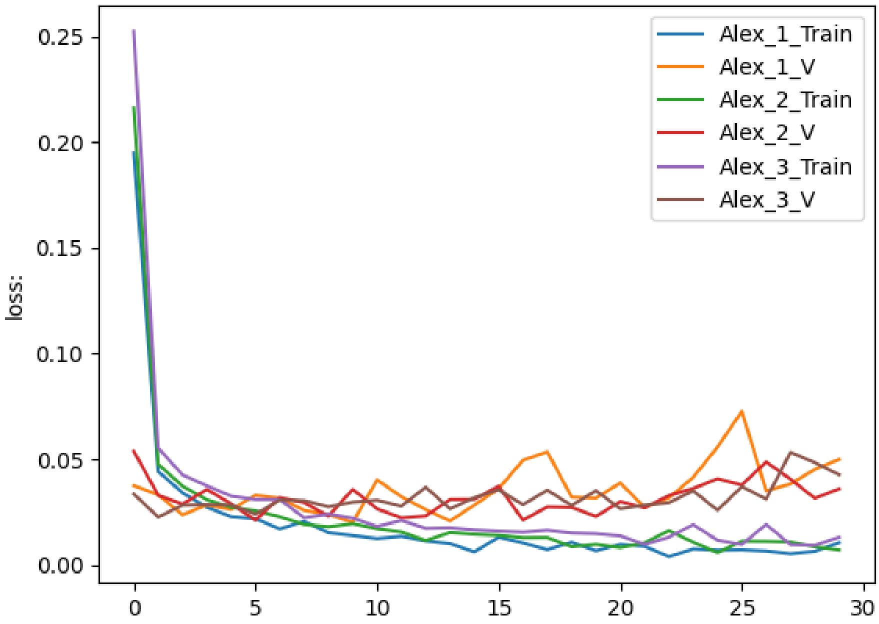
采用alexnet原因为其可学习参数更多，网络学习，拟合能力更强，更易发生过拟合，方便实验对比。

在接下来的实验里，若模型名称后缀为Train则代表训练该模型时该epoch的平均值结果。

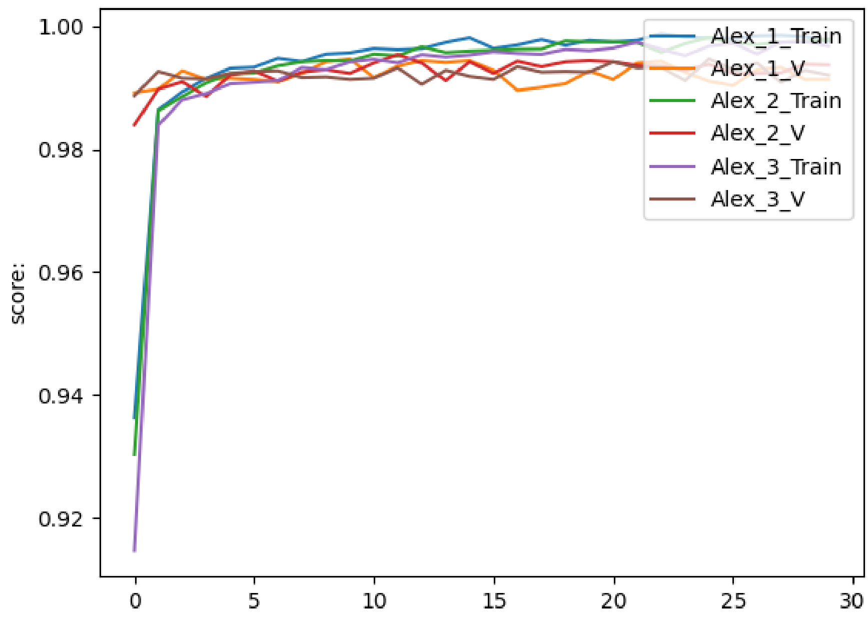
若模型名称后缀为V则代表测试/验证集上该模型时该epoch的平均值结果。

1. Droupout对比

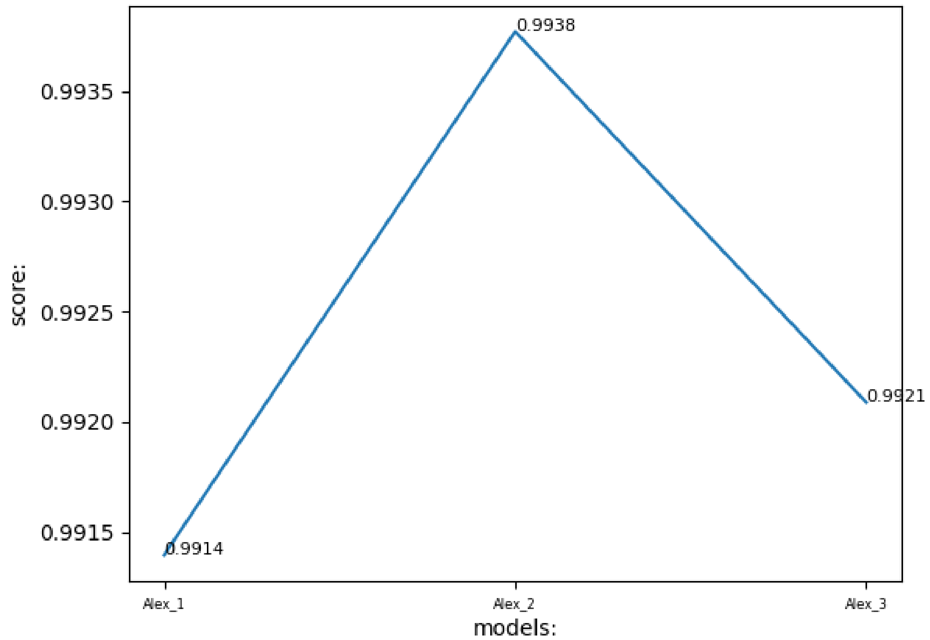
参数/模型名	Alex_1	Alex_2	Alex_3
epochs	30	30	30
lr	5e-4	5e-4	5e-4
batch_size	128	128	128
droup_rate	0.0	0.5	0.7



训练&验证损失图



训练&验证得分图

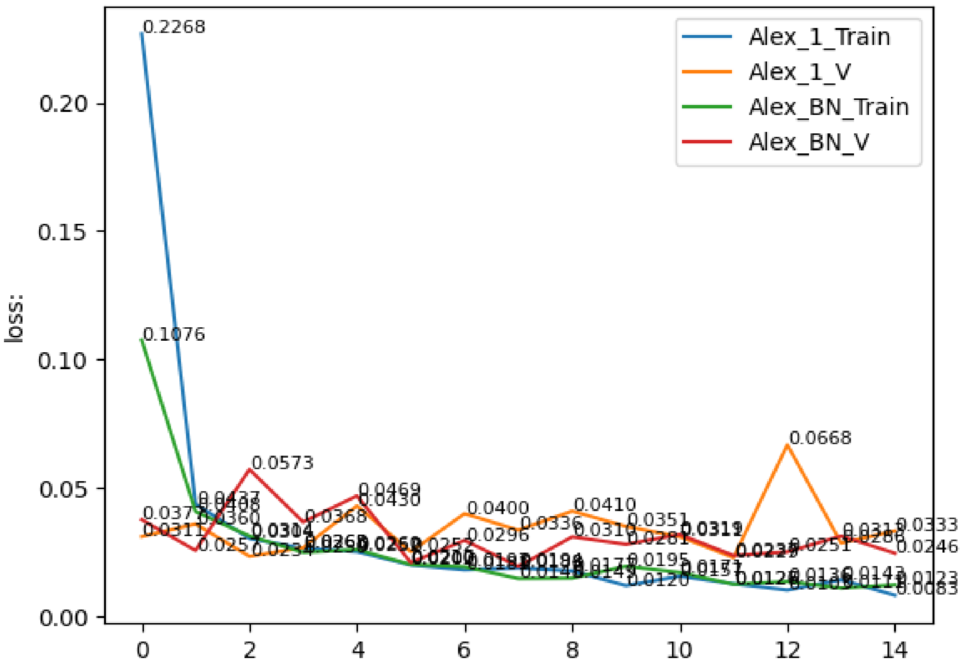


测试得分图

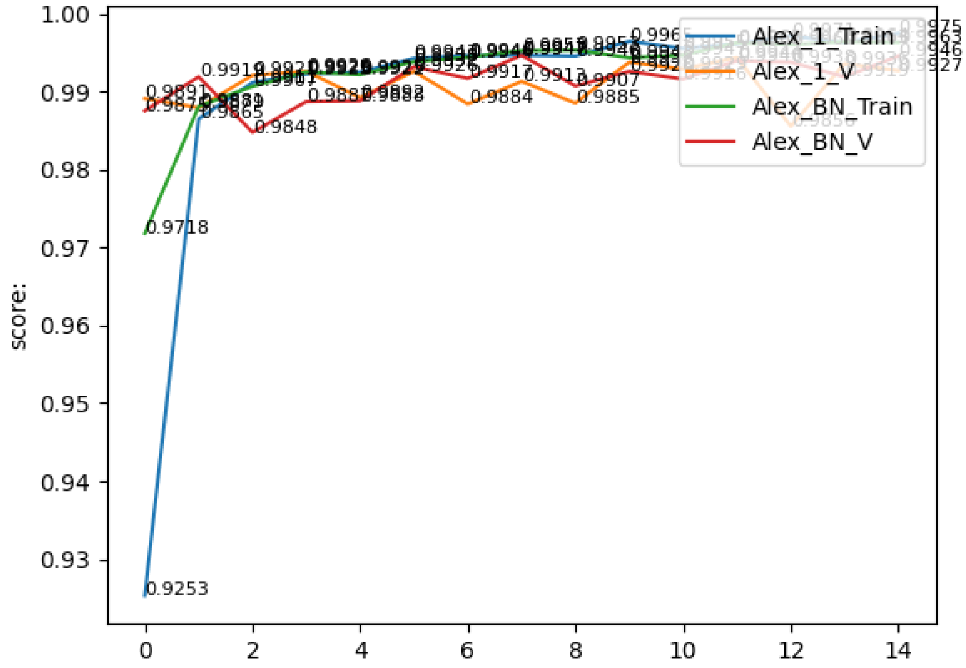
根据以上结果易知，有Droupout网络其验证损失更贴近于较低的训练损失，验证精度也更高。当全连接中丢弃率为0.5时，网络效果最佳，拥有丢弃选择的网络整体能力更强于无丢弃网络。Droupout在一定程度上解决了过拟合问题。

2. BatchNorm对比

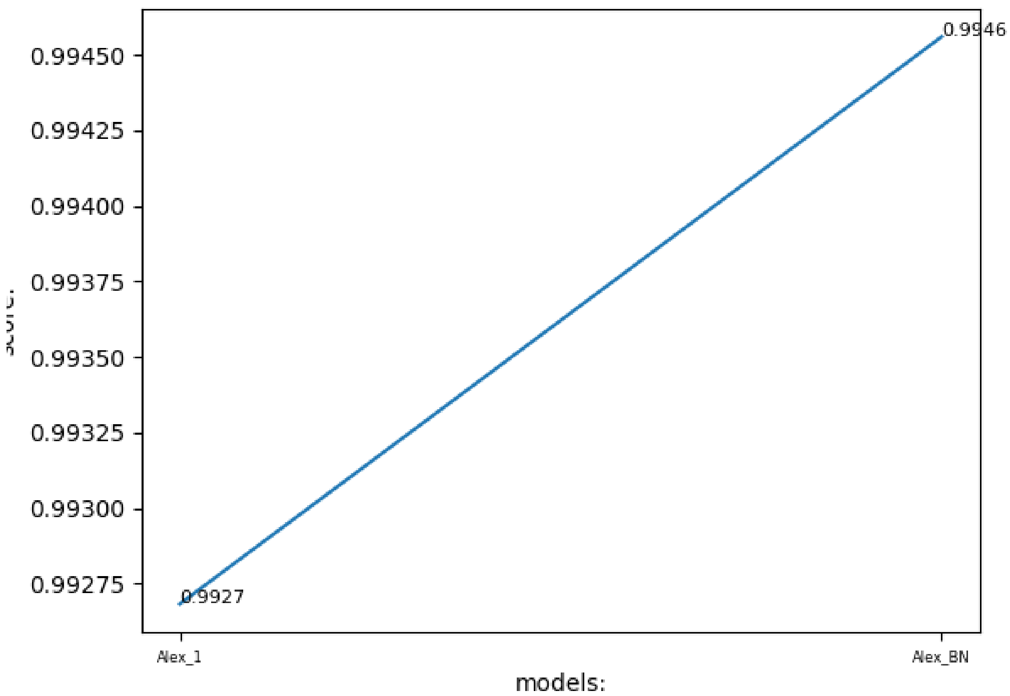
参数/模型名	Alex_1	Alex_BN
epochs	15	15
lr	5e-4	5e-4
batch_size	128	128
droup_rate	0.0	0.0
是否在卷积，全连接层后添加了BatchNorm层	否	是



训练&验证损失图



训练&验证得分图



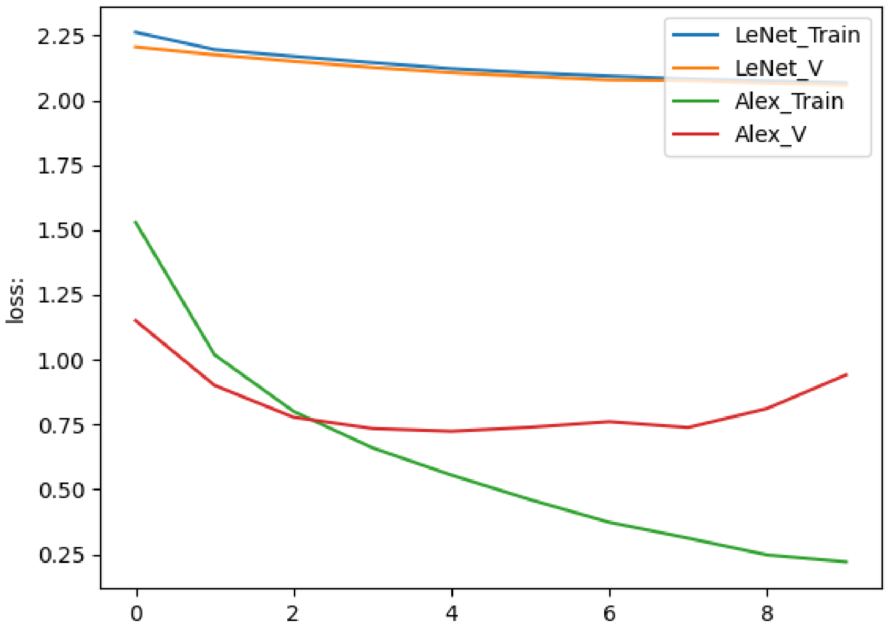
测试得分图

以上结果可分析知：在卷积层，全连接层插入BatchNorm层进行批量归一化后，其损失下降更快，训练得分与测试得分更加贴近，其结果泛化能力显然更强。BatchNorm对于解决过拟合，使网络更易训练这方面有一定作用。

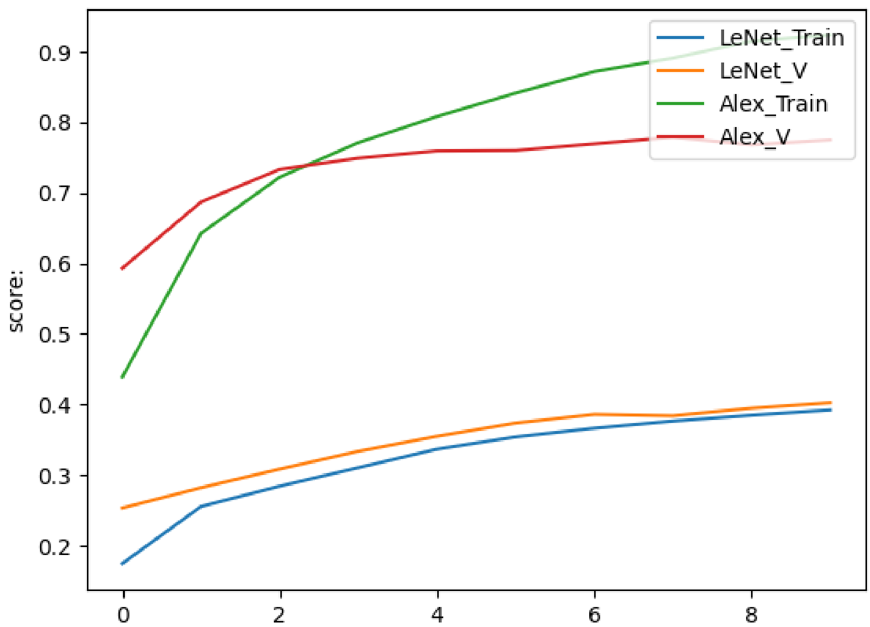
③ 网络性能提升实验 (on CIFAR-10)：

该类实验中，我们对所有网络训练10个epoch得到实验结果。

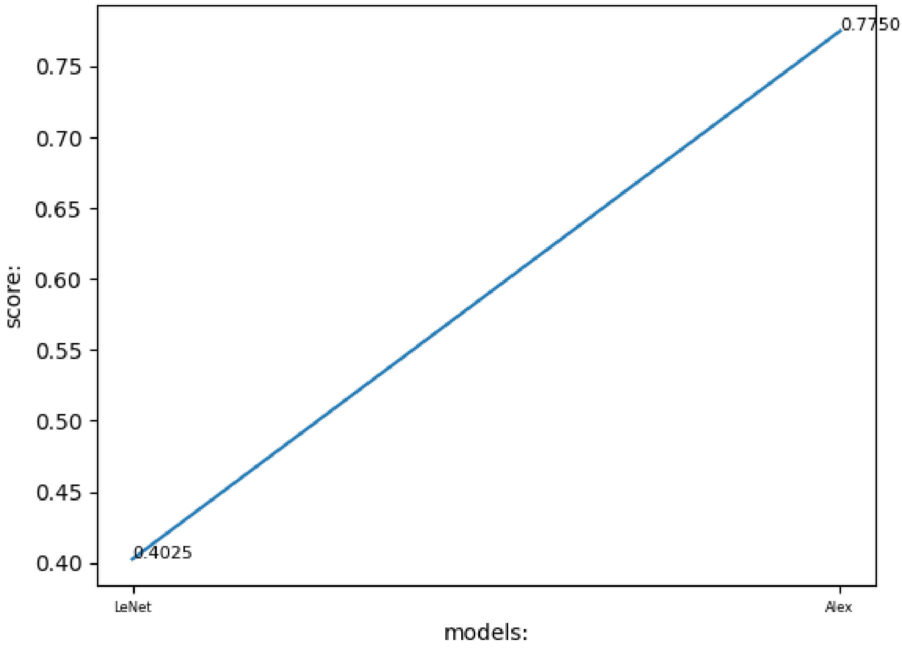
1. 网络层数（LeNet, AlexNet）：



训练&验证损失图



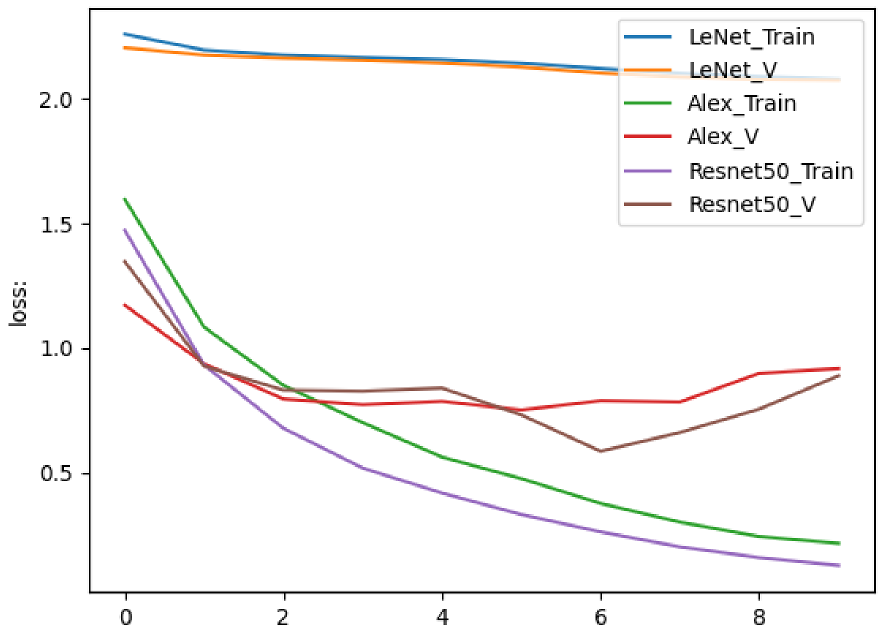
训练&验证得分图



测试得分图

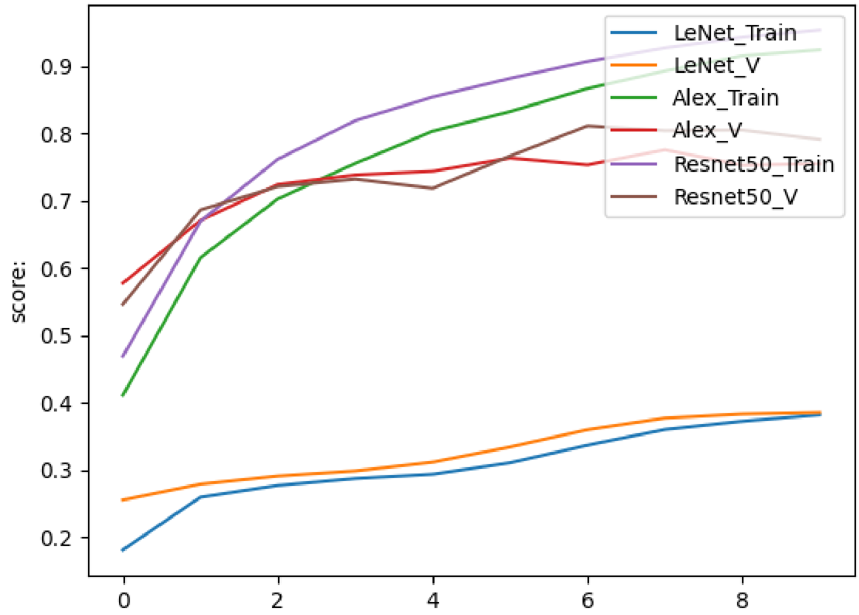
由以上结果和AlexNet与LeNet的结构对比可知，拥有更深层的网络其特征提取能力确实更强，在3通道RGB数据上，LeNet明显遭遇瓶颈，训练较慢且精度较低，AlexNet拥有显著优势。

2. 添加残差块，训练更深层网络(ResNet50, AlexNet, LeNet)：

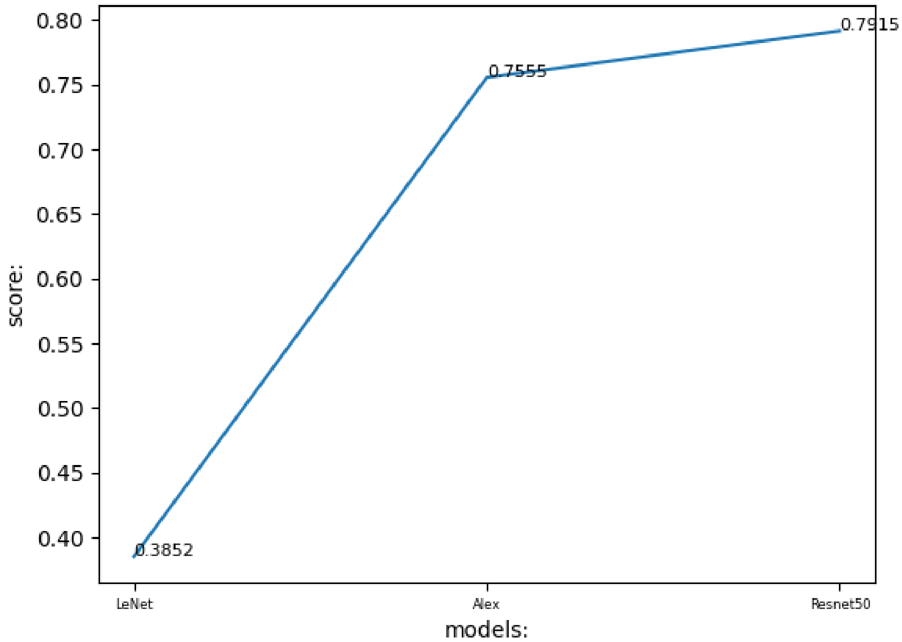


训练&验证损失图

由以上结果易得：ResNet50由于其更深层的网络结构以及利用了残差的原理，使训练深层网络变得更加容易，取得了更佳的损失下降，精度结果。



训练&验证得分图

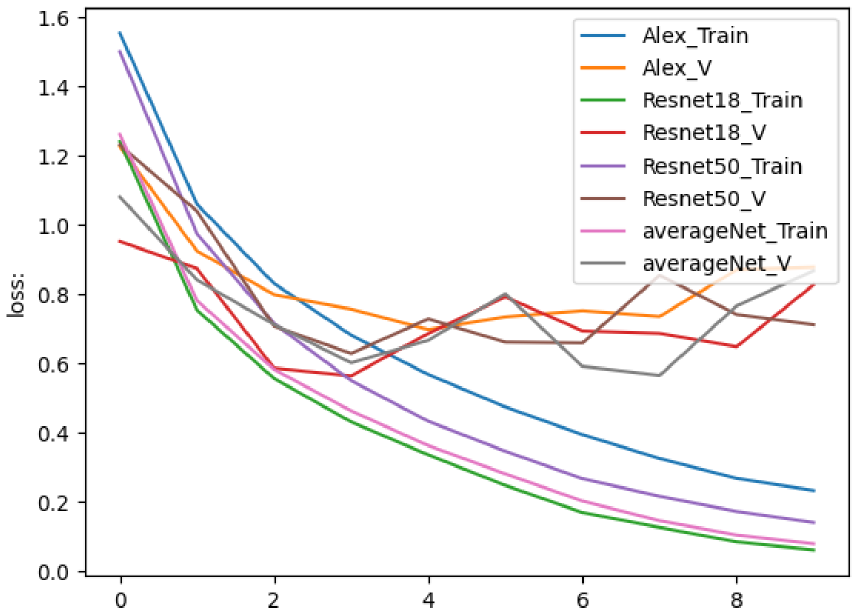


测试得分图

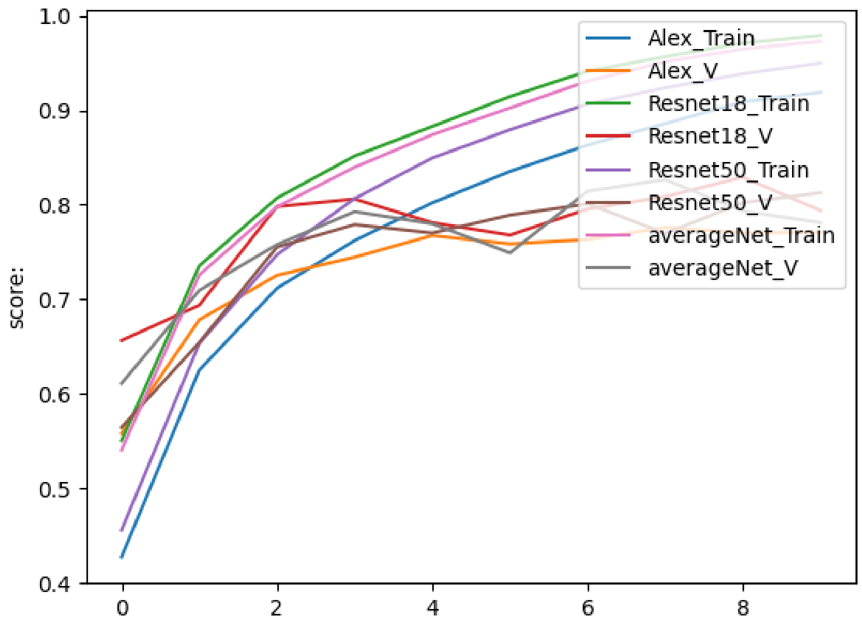
由以上结果易得：ResNet50由于其更深层的网络结构以及利用了残差的原理，使训练深层网络变得更加容易，取得了更佳的损失下降，精度结果。

3. 使用模型集成方法（Resnet18 & AlexNet）：

averageNet由模型集成方法中的Average方法得到。利用AlexNet与ResNet18得到了该集成神经网络。



训练&验证损失图



训练&验证得分图

观察以上结果，ResNet18与AverageNet取得了更佳的结果，二者能力较近。观察验证集得分上，集成神经网络也能多次得到较佳测试结果。模型集成方法对于模型性能提升拥有一定作用。

三、结语：

🎉 在本次实验中，先尝试对早期较为基础的卷积神经网络LeNet进行了实现，并在LeNet网络上理解交叉验证调试相关超参数。

之后利用AlexNet在MINIST数据集上尝试正则化方法对模型泛化能力的提升。

在模型的性能提升方法尝试中：

先替换训练数据集为CIFAR-10，以提高模型训练难度。

尝试了更深层的AlexNet与LeNet在更高难度数据集上训练结果对比，更深层神经网络显然拥有更大优势。

后又尝试使用ResNet50，利用残差神经网络更易训练深层次神经网络的特性，得到了更好的结果。

最后尝试使用 Average 这一模型集成方法，将AlexNet与ResNet18进行了聚合，提升了模型性能。

在这次实验中对于卷积神经网络的使用及实现，模型结构设计和正则化方法上有了更深入的理解。