Lab2 实验报告

PB20111689 蓝俊玮

实验步骤

1. 数据生成

本次数据集的获取通过 `torchvision.datasets.CIFAR10` 获取。将获取的数据集保存成训练集 `train_dataset` 和测试集 `test_dataset`。

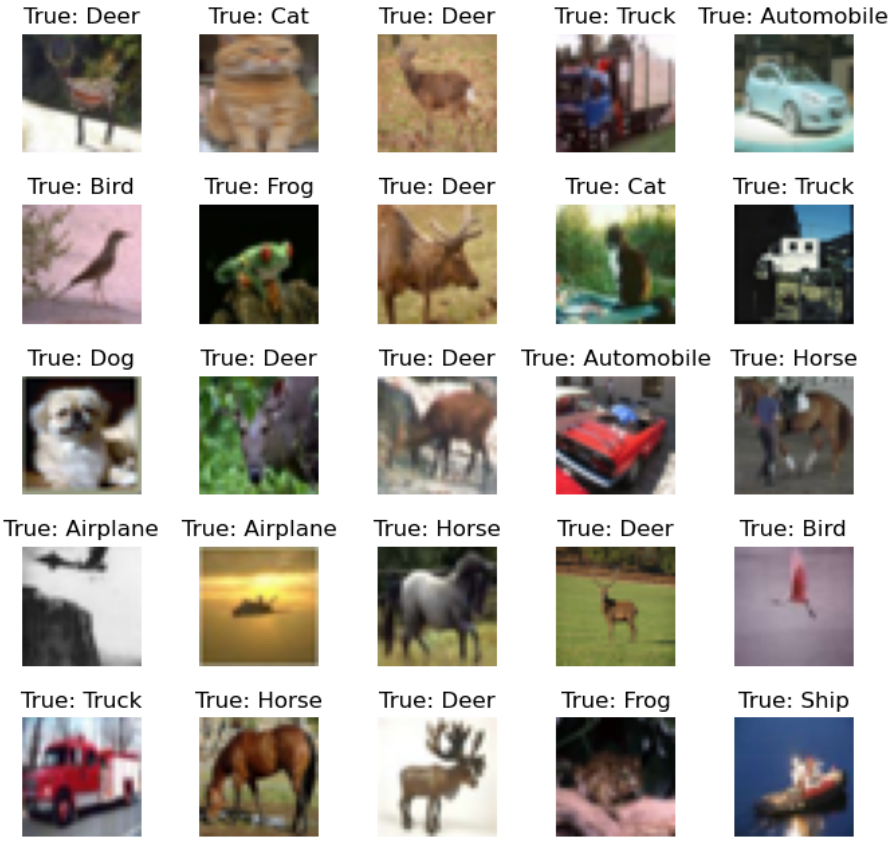
```
data_root_dir = "/amax/home/junwei/deep-learning/CIFAR10"
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((32, 32)),
    torchvision.transforms.ToTensor()
])

train_dataset = torchvision.datasets.CIFAR10(root=data_root_dir, train=True,
transform=transformer, download=True)
test_dataset = torchvision.datasets.CIFAR10(root=data_root_dir, train=False,
transform=transformer, download=True)
```

然后采用留出法，使用 `torch.utils.data.random_split` 将训练集 `train_dataset` 按比例 8 : 2 划分成训练集 `trainset` 和验证集 `validset`，后者用于调参分析时的性能评估。

```
generator = torch.Generator().manual_seed(1689)
trainset, validset = Data.random_split(train_dataset, [0.8, 0.2], generator)
```

获取数据集后，我们可以通过查看图像以更好地理解数据集。



2. 模型搭建

本次实验，我将选择 pytorch 作为我的网络框架。下面的模型是我默认的模型，该模型以 3 个卷积块作为基础。其中每个卷积块中包括 2 个卷积层和 1 个池化层。其中使用了 `ReLU` 作为激活函数，并且在助教的提醒下，将 `BatchNorm` 放在了卷积层与激活函数之间（在我重新认真阅读了 PPT 之后，看到的原因为：“由于非线性激活函数，每层输入通常是非高斯的，不适合进行标准化”）。最后连接了 2 个全连接层，并且在全连接层中加入了 `Dropout` 层，在助教的提醒下，将其默认的概率设置为 0.5。

```
class CNN(nn.Module):
    def __init__(self, kernel_size=3, dropout_rate=0.5):
        super(CNN, self).__init__()
        padding_size = int((kernel_size - 1) / 2)

        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.conv3 = nn.Sequential(
```

```
            nn.Conv2d(64, 128, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=kernel_size, padding=padding_size),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )

        self.fc1 = nn.Linear(128*4*4, 512)
        self.dropout = nn.Dropout(p=dropout_rate)
        self.fc2 = nn.Linear(512, 10)
```

基于我们要对 normalization 和网络深度等超参数对分类性能的影响做探究，因此我定义了:

```
class CNN2(nn.Module):
    # 使用 2 个卷积块
class CNN4(nn.Module):
    # 使用 4 个卷积块
class CNN_N(nn.Module):
    # 没有使用 BatchNorm
```

这些都是在默认模型 `CNN` 上做了一些改动得到的。此外，我还设计了 ResNet，用来与普通卷积网络比较性能。（由于 ResNet 不是实验考察点，这里就不展示代码了）

3. 模型训练

模型训练采用的是留出法进行模型的训练和评估。模型训练的轮数 `epoch = 20`，数据集的 `batch_size = 128`，优化器使用的是 `Adam` 优化器，设置其初始学习率为 `5e-3` 且为其正则化设置了 `weight_decay = 1e-5`，学习率调整策略使用的指数衰减。在模型每一轮的训练时，我们会记录其训练集损失值和训练集准确率以及验证集损失值和验证集准确率。

```
def train_valid(estimator, train_set, valid_set, lrd):
    """ 训练并验证
    :param estimator: 需要使用的网络模型
    :param train_set: 训练集
    :param valid_set: 验证集
    :param lrd: 学习率衰减指数
    """
    device = "cuda" if torch.cuda.is_available() else "cpu"
    print("Running on %s" %device)

    model = copy.deepcopy(estimator).to(device)
    model.device = device

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), weight_decay=1e-5, lr=5e-3)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=lrd,
last_epoch=-1)

    n_epochs = 20
    batch_size = 128
    train_loader = Data.DataLoader(train_set, batch_size=batch_size,
shuffle=True)
    valid_loader = Data.DataLoader(valid_set, batch_size=batch_size,
shuffle=False)
```

```
    train_losses = []
    train_accs = []
    valid_losses = []
    valid_accs = []
    lrs = []

    for epoch in range(1, n_epochs + 1):
        # Training Process

        # Validation Process

    return train_losses, train_accs, valid_losses, valid_accs, lrs
```

训练过程如下：

```
model.train()
n_correct = 0
n_total = 0
train_loss = []
for batch in tqdm(train_loader):
    x, true_label = batch
    preds = model(x.to(device))
    loss = criterion(preds, true_label.to(device))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    train_loss.append(loss.item())

    _, pred_label = torch.max(preds, 1) # 选取概率值最大的作为预测标签
    n_correct += (pred_label == true_label.to(device)).sum().item()
    n_total += true_label.shape[0]
lrs.append(optimizer.state_dict()['param_groups'][0]['lr'])

scheduler.step()    # 调整学习率
```

在这里，由于模型最后输出的值实际上是每个标签类别的概率值（即类似经过 softmax 函数），因此我们需要通过 `torch.max(preds, 1)`，将概率值最大的类别作为预测值。同时在所有批次的梯度更新结束之后，我们需要通过 `scheduler.step()` 调整学习率。

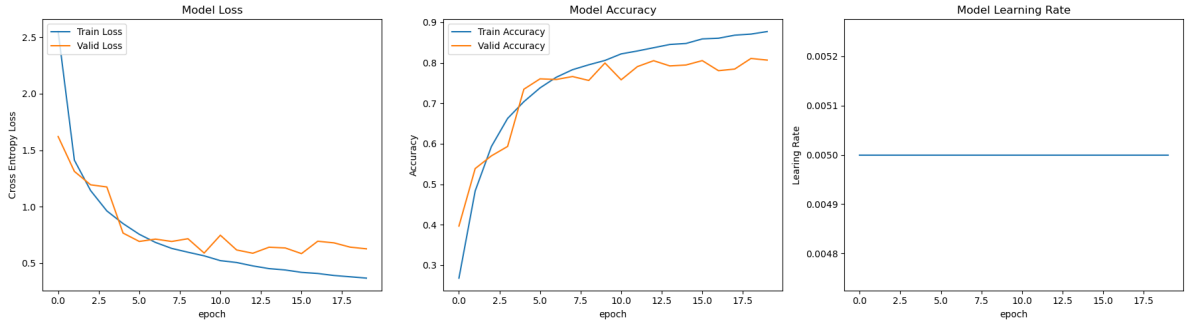
验证过程如下：

```
model.eval()
n_correct = 0
n_total = 0
valid_loss = []
for batch in valid_loader:
    x, true_label = batch
    with torch.no_grad():
        preds = model(x.to(device))
        loss = criterion(preds, true_label.to(device))
        valid_loss.append(loss.item())

    _, pred_label = torch.max(preds, 1)
    n_correct += (pred_label == true_label.to(device)).sum().item()
    n_total += true_label.shape[0]
```

接下来以默认的参数进行训练和验证，可以得到结果（最后一轮）如下：

训练损失值	验证损失值	训练准确率	验证准确率
0.3666	0.6262	0.8770	0.8066



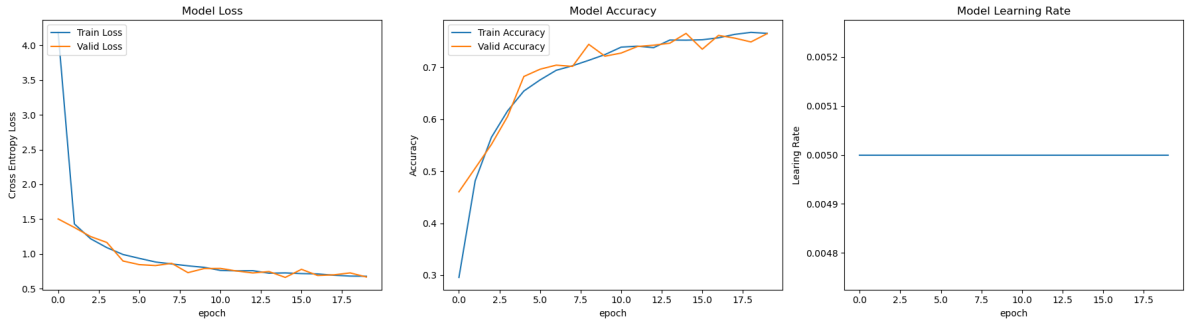
不难看出，模型训练基本上已经收敛，且在验证集上的准确率也是比较稳定。

4. 模型调参

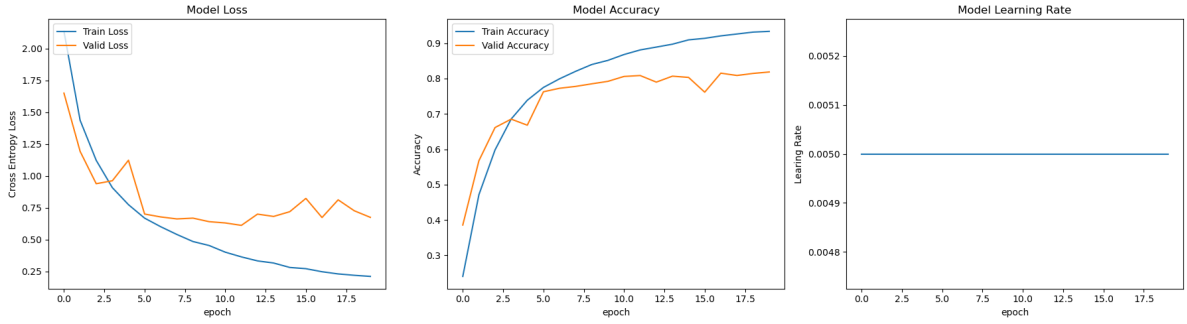
4.1 Network Depth

在上面模型搭建中提到过，在本次实验中，我对网络深度进行探究的方式是通过探究不同卷积块的个数而非隐藏层的个数。原因是我觉得在卷积网络中，卷积块扮演着十分重要的角色。卷积网络的功能以卷积块为基础，以卷积块为探究基础的意义比隐藏层的要大很多。单单增加一两层的隐藏层对模型的影响可能不是很大，而且这些隐藏层之间的可能也没有功能联系，而卷积块将这些隐藏层集成在一起，其意义是 1 + 1 > 2 的。

对 CNN2 模型进行训练：



对 CNN4 模型进行训练：

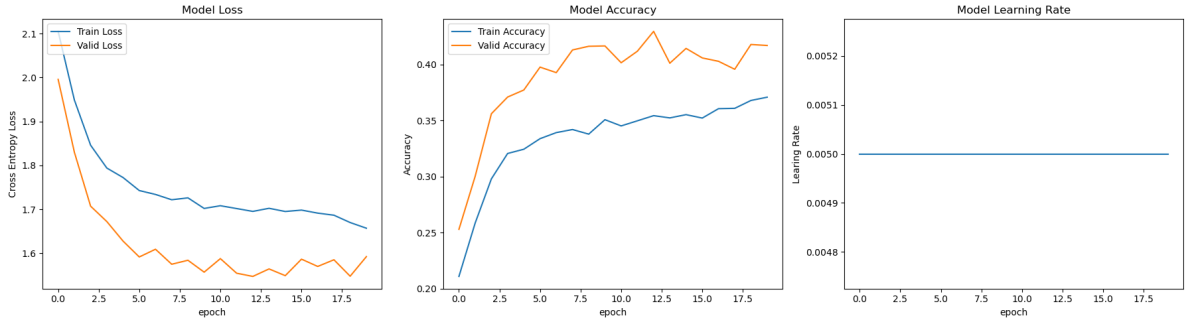


Network Depth	训练损失值	验证损失值	训练准确率	验证准确率
CNN2	0.6792	0.6688	0.7653	0.7648
CNN3	0.3666	0.6262	0.8770	0.8066
CNN4	0.2121	0.6753	0.9335	0.8186

如果仔细观察上面的曲线图，我们可以发现 CNN2 网络训练损失值收敛在 0.75 左右，我们默认的 CNN3 网络模型训练损失值收敛时未到达 0.9，而 CNN4 网络模型训练损失值收敛时超过了 0.9。很明显 CNN2 网络的网络深度不足，导致其模型训练即使已经收敛了，仍然学习到的内容不足，表现出其在验证集的准确率不佳，体现出该模型的学习能力不足，表达能力不够，在该数据集下的性能表现不够好。而随着网络深度的增加，可以看到其训练损失值和训练准确率都在变好，可以体现出网络深度的重要。当然，并不是网络深度越深越好，也可以看出在模型网络深度增加时，模型训练收敛的速度越来越慢，且其波动性也是比较明显。因为网络深度的增加必然会引入更多的参数，其训练难度会增加，使模型难以收敛。

4.2 Normalization

对没有使用 BatchNorm 的网络模型 CNN_N 进行训练：

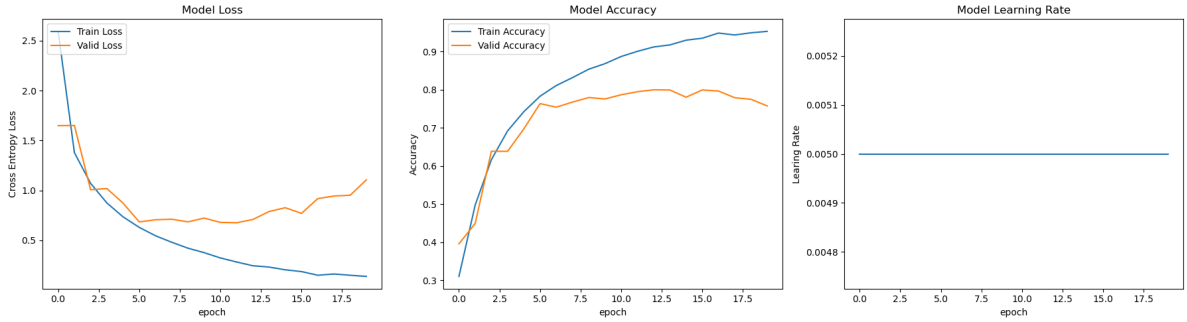


Normalization	训练损失值	验证损失值	训练准确率	验证准确率
有	0.3666	0.6262	0.8770	0.8066
无	1.6750	1.5923	0.3708	0.4169

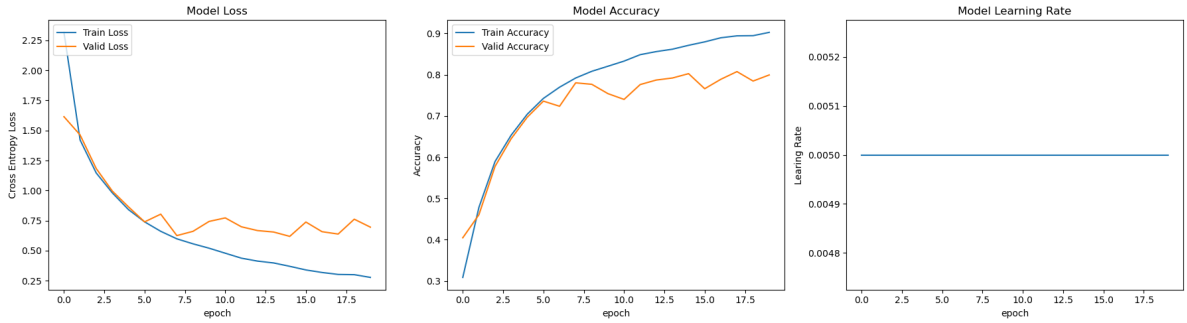
从这个结果可以看出，BatchNorm 是十分重要的，它极大的影响了网络模型的性能表现。这是因为 Batch Norm 可以改善梯度的传播。可以看到在上面没有了 Batch Norm 之后，模型的训练效果十分差，甚至其在训练集上的性能表现远不如在验证集上的性能表现，充分体现出模型学习的效果差，由此说明 BatchNorm 在梯度传播中的重要作用。此外，Batch Norm 还有允许比较大的学习率的作用，它也是一种隐形的正则化方法。它可以改善标准归一化对网络模型的负面影响，提高神经单元的表达能力。

4.3 Dropout

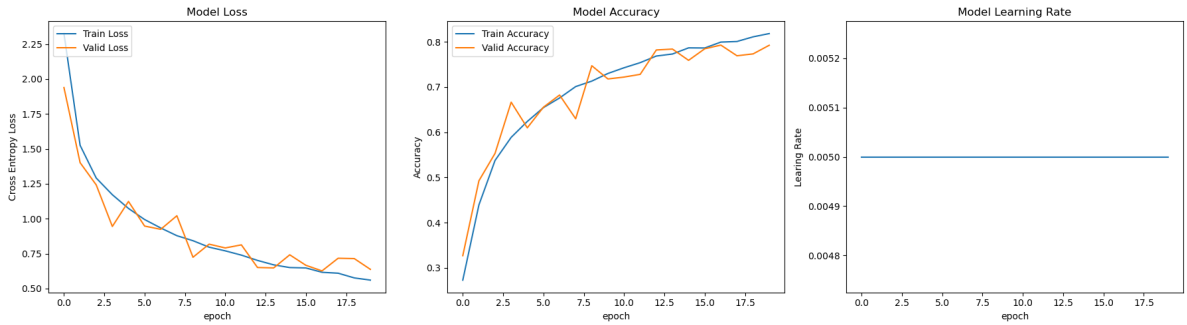
当 `dropout_rate = 0` 的时候，即此时没有被 dropout 的神经元，此时模型相当于没有进行 dropout 操作，其训练结果为：



当 `dropout_rate = 0.25` 的时候，其训练结果为：



当 `dropout_rate = 0.75` 的时候，其训练结果为：



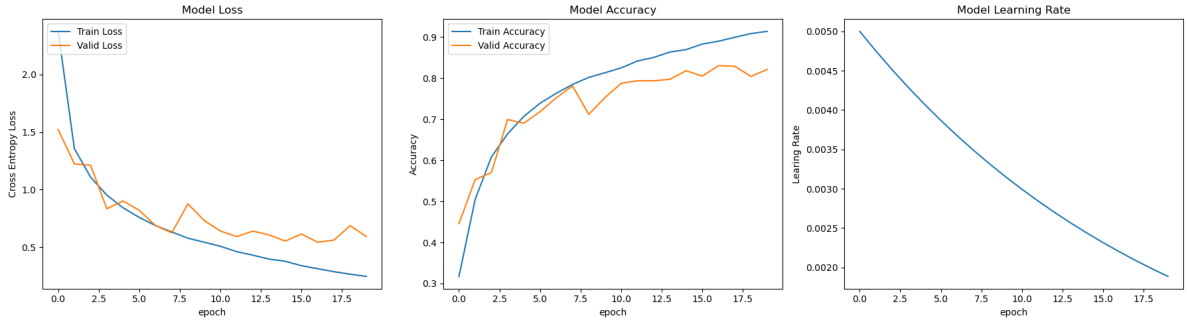
Dropout	训练损失值	验证损失值	训练准确率	验证准确率
0	0.1383	1.1060	0.9527	0.7574
0.25	0.2773	0.6955	0.9027	0.7995
0.50	0.3666	0.6262	0.8770	0.8066
0.75	0.5603	0.6372	0.8181	0.7922

当没有 Dropout 层的时候，可以看出模型训练会产生过拟合的结果。模型在验证集上的表现随着训练的过拟合而变差，其损失率和准确率都在变差。而与默认的 Dropout Rate 为 0.5 相比，0.25 在验证集上的性能表现有不断的波动，不够稳定，感觉还是有轻微的过拟合现象。而当 Dropout Rate 为 0.75 的时候，处于抑制状态的神经元过多，模型训练是肯定不会过拟合的。但是却表现出其训练学习的不足，由于太多神经元被抑制了，也导致了网络模型的学习能力不足，在学习中丢失了不少重要的特征信息，所以其性能也是稍显不足。

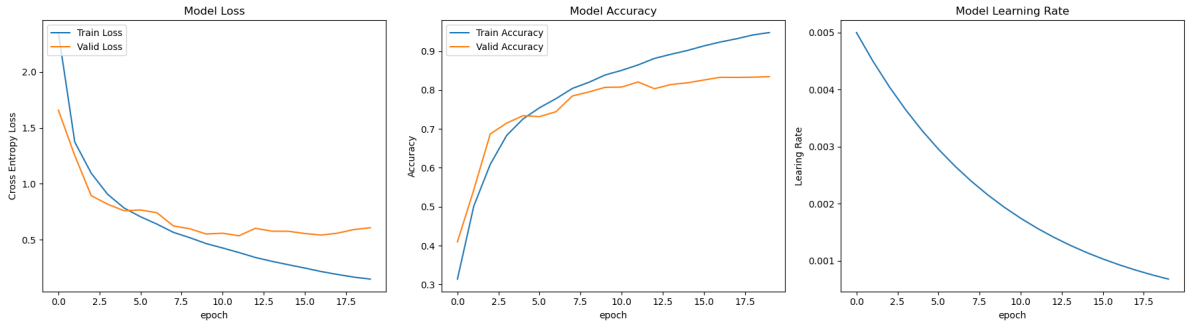
4.4 Learning Rate Decay

由于我采用的是指数衰减，当其衰减指数设置的太小时，学习率会过早的到 0，因此对学习率衰减的探究我设置的衰减指数为 0.95，0.90 和 0.80.

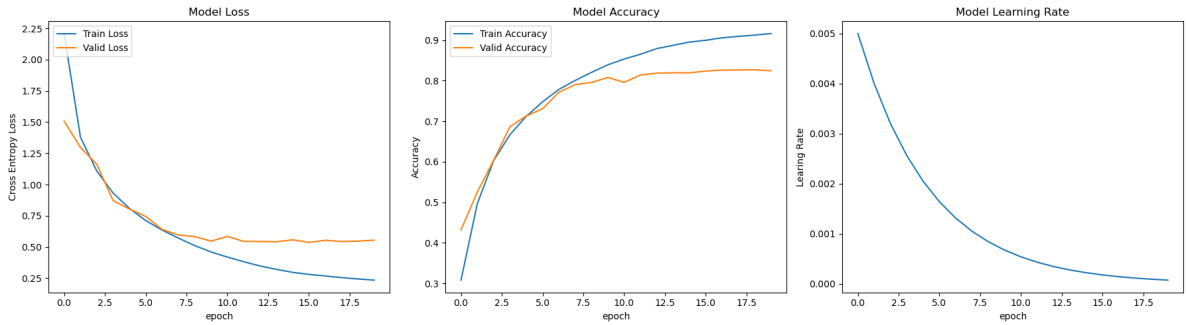
当 `Learning Rate Decay = 0.95` 的时候，其训练结果为：



当 `Learning Rate Decay = 0.90` 的时候，其训练结果为：



当 `Learning Rate Decay = 0.80` 的时候，其训练结果为：

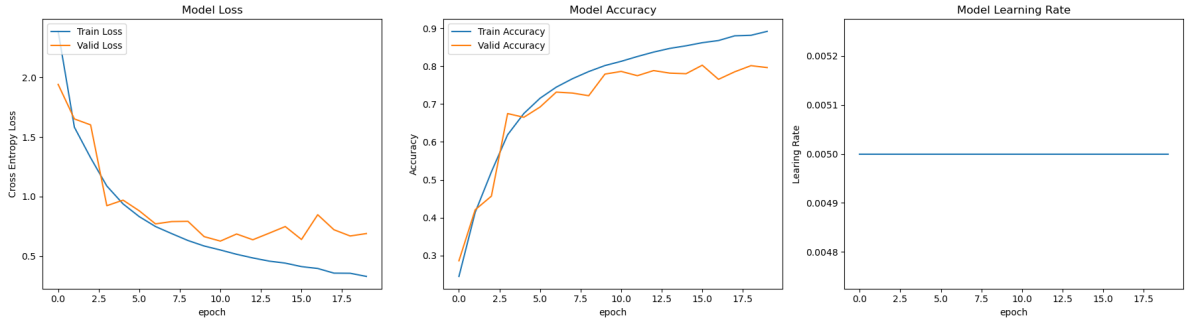


Learning Rate Decay	训练损失值	验证损失值	训练准确率	验证准确率
1.00	0.3666	0.6262	0.8770	0.8066
0.95	0.2466	0.5922	0.9137	0.8211
0.90	0.1478	0.6070	0.9477	0.8344
0.80	0.2353	0.5551	0.9164	0.8247

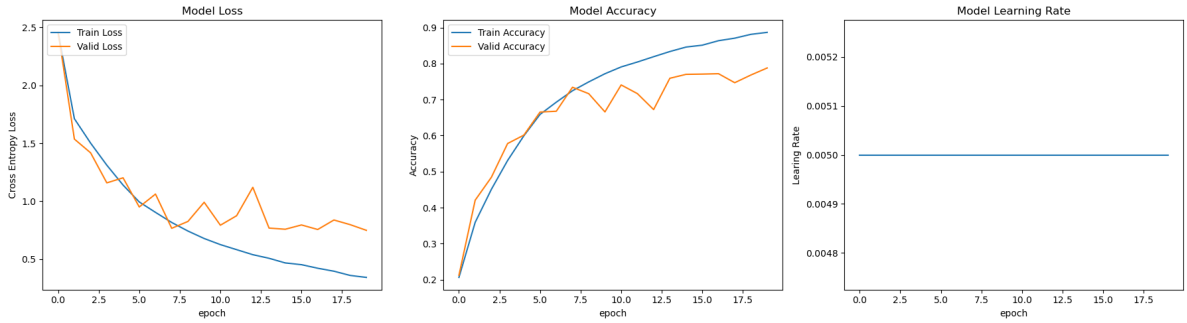
在默认的参数设置下，是没有设置学习率衰减的，其整体表现还算不错。但是在加上学习率衰减之后，发现其性能又有了进一步的提升。当学习率衰减指数为 0.90 的时候，其表现性能是最好的。而当学习率衰减指数为 0.95 的时候，可以看出在训练的过程中，模型在验证集上的表现波动较大，而当学习率衰减指数为 0.80 的时候，可以看出学习率在第 10 轮的时候，比学习率衰减指数为 0.90 时第 20 轮的学习率还要低，因此其模型的学习在第 10 轮就开始缓慢进行，所以在后面的 10 轮训练中，学习到的特征就微乎其微，因此在验证集上表现出没有变化，很平缓。从总的表现来看，还是学习率衰减指数为 0.90 的时候更好，它让学习率可以更好的随着训练过程而变化。

4.5 Kernel Size

当 `Kernel Size = 5` 的时候，其训练结果为：



当 `Kernel Size = 7` 的时候，其训练结果为：



Kernel Size	训练损失值	验证损失值	训练准确率	验证准确率
3	0.3666	0.6262	0.8770	0.8066
5	0.3302	0.6899	0.8921	0.7964
7	0.3431	0.7493	0.8868	0.7879

当卷积核大小从 3 变到 5 和 7之后，第一感觉就是其训练时在验证集上的表现波动很大。从数据的体现上就是其效果变差。卷积核大小变大后可以增大神经元的感受野，但是感觉在这个数据上增加感受野效果并不是很好。cifar10 数据集的图片大小是 32x32，从之前的数据集图片展示可以看到，有些要检测的目标并不是占据整个图片，其中不少待分类的目标十分小。这样的话，增加卷积核的大小，即增加了感受野的大小，在处理这些图片的时候，就会被更多非目标的环境信息所干扰。而且，cifar10 的数据集看起来本身就很模糊，其边缘信息本就不明显，如果选取了更多领域的信息，那么在这个领域内的处理必然会钝化边缘特征。所以感觉卷积核大小为 3x3 确实是更合适的选择，至少我觉得它更容易提取出边缘信息，而识别这些物体最重要的特征之一就是边缘特征。

5. ResNet

除了上述普通的卷积网络，我还尝试了一下残差网络，并且尝试利用 “bottleneck” 层改进其效率。

Model	训练损失值	验证损失值	训练准确率	验证准确率
CNN	0.3666	0.6262	0.8770	0.8066
ResNet	0.0204	0.6773	0.9934	0.8677
bottleneck	0.1056	0.4997	0.9655	0.8558

还是可以明显的看出 ResNet 的性能提升。

6. 性能测试

最后选择 4 个卷积块作为基础，带有 Normalization，Dropout 为 0.5，Learning Rate Decay 为 0.90，Kernel Size 为 3x3 为参数，在整个训练集上进行训练，最后在测试集的预测结果如下：

Model	训练损失值	测试损失值	训练准确率	测试准确率
CNN4	0.0357	1.0339	0.9879	0.8388

绘制出其混淆矩阵和部分预测图：

同时我也尝试了 ResNet 模型为基础，带有 Normalization，Dropout 为 0.5，Learning Rate Decay 为 0.90，Kernel Size 为 3x3 为参数，其结果如下：

Model	训练损失值	测试损失值	训练准确率	测试准确率
ResNet	0.0156	0.6233	0.9950	0.8818