人工智能期中 Project

18340146 计算机科学与技术 宋渝杰

18340131 计算机科学与技术 缪贝琪

任务一：

实现 CNN 卷积神经网络，完成 cifar-10 图片分类任务

算法原理：

数据读取：

实验过程中突然提到不能直接使用 CIFAR10 函数直接读取数据集并返回，而原数据集又是二进制文件，无法直接打开浏览结构。然而官网提供了以下代码用于返回字典结构的数据集：

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict
```

当然字典结构的数据集无法使用 DataLoader 打包处理，也无法直接喂给神经网络进行训练。网上查询了相关博客，得知需要将其转化为 Datasets 数据集类，转化模版如下：

```
class MyData(torch.utils.data.Dataset): # 需要继承torch.utils.data.Dataset
    def __init__(self):
        # TODO
        # 1.初始化文件路径或文件名列表
        # 也就是在这个模块里，我们所做的工作就是初始化该类的一些基本参数
        # 也可以在此处直接加载好数据集
        pass
    def __getitem__(self, index): # 必须声明
        # TODO
        # 1.从数据集中读取一个数据（例如，使用numpy.fromfile，PIL.Image.open）
        # 2.预处理数据（例如torchvision.transform）
        # 3.返回数据对（例如图像和标签）
        pass
    def __len__(self): # 必须声明
        # 返回数据集的总大小即可
```

在参考了 CIFAR10 的源码后，我的 MyCifar10 代码如下：

```
class MyCifar10(torch.utils.data.Dataset): # MyCifar10

    def __init__(self, root, train = True, transform = None): # 文件目录，训练/验证，预处理
        self.root = os.path.expanduser(root)
        self.transform = transform
        if train: self.lis = ['data_batch_1', 'data_batch_2', 'data_batch_3', 'data_batch_4', 'data_batch_5'] # 训练集
        else: self.lis = ['test_batch'] # 验证集
        self.data, self.targets = [], []
        for name in self.lis:
            path = os.path.join(self.root, name)
            with open(path, 'rb') as f:
                entry = pickle.load(f, encoding = 'latin1') # CIFAR10用的是encoding = 'latin1'
                self.data.append(entry['data']) # 图片数据
                self.targets.extend(entry['labels']) # 标签（注意是extend）
        self.data = np.vstack(self.data).reshape(-1, 3, 32, 32).transpose((0, 2, 3, 1)) # numpy.ndarray高维矩阵的表示:H,W,C

    def __getitem__(self, index):
        img, target = self.data[index], self.targets[index] # 获取数据和标签
        img = Image.fromarray(img)
        if self.transform is not None: img = self.transform(img) # 预处理
        return img, target

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

之后就可以通过和 CIFAR10 几乎一样的模式进行使用：

```
train_dataset = MyCifar10('CNN/data', train = True, transform = transforms.Compose([ # 加载数据
    transforms.RandomCrop(32, padding = 4), # 随机扩展裁剪
    transforms.RandomHorizontalFlip(), # 随机水平翻转
    transforms.ToTensor(), # [0,255]->[0,1]
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), # [0,1]->[-1,1]
]))

test_dataset = MyCifar10('CNN/data', train = False, transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
]))
```

卷积神经网络（CNN）：

一种特殊的神经网络（本质上也是数学模型）。和普通的神经网络一样，分为输入层、隐藏层、输出层。而隐藏层主要为**卷积层**、**池化层**、**激活层**叠加组合形成

卷积层：

这一层是卷积神经网络最重要的一个层次，也是“卷积神经网络”的名字来源。

基本的二维卷积操作：二维卷积操作作用于二维矩阵，可以将一个 m*m 的矩阵与一个 a*a 的卷积核（也是一个矩阵）进行操作得到一个 (m-a+1)*(m-a+1) 的矩阵，卷积的公式如下：

$$y_{i,j} = \sum_{u=0}^{a-1}\sum_{v=0}^{a-1} w_{i+u,j+v} * x_{i+a-1-u,j+a-1-v}$$

多层操作：卷积神经网络中，更多的操作是将 p 个 m*m 的矩阵与 p*q 个 a*a 的卷积核进行卷积操作得到 q 个 (m-a+1)*(m-a+1) 的矩阵，具体的步骤如下：

1. 将这 p 个 m*m 的矩阵与前 p 个 a*a 的卷积核进行卷积操作，得到 p 个 (m-a+1)*(m-a+1) 的矩阵
2. 将这 p 个 (m-a+1)*(m-a+1) 的矩阵相加，得到一个 (m-a+1)*(m-a+1) 的矩阵
3. 对步骤 1、2 重复 q 次，即得到 q 个 (m-a+1)*(m-a+1) 的矩阵

pytorch 二维卷积函数如下：

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
dilation=1, groups=1, bias=True, padding_mode='zeros')
```

常用的参数如下：

in_channels：输入的通道大小，即上文多层操作的 p

out_channels：输出的通道大小，即上文多层操作的 q

kernel_size：卷积核的大小，即上文的 a

stride：卷积核移动长度，默认值（常用值）为 1

padding：矩阵外部扩展大小，即在原 m*m 的矩阵外补充 n 圈，扩展成 (m+2n)*(m+2n) 的矩阵

bias：是否对卷积操作加入偏移量

池化层：

这一层也是十分重要的一个层次，起到数据压缩的作用，可以将 m*m 的矩阵压缩成 (m/k)*(m/k) 的矩阵。

常用的池化层为**最大池化层**和**平均池化层**

最大池化层：对原 m*m 的矩阵每 k*k 的小块计算出最大值，得到 (m/k)*(m/k) 个最大值，组合得到压缩后的 (m/k)*(m/k) 的矩阵

平均池化层：对原 m*m 的矩阵每 k*k 的小块计算出平均值，得到 (m/k)*(m/k) 个平均值，组合得到压缩后的 (m/k)*(m/k) 的矩阵

pytorch 池化层函数如下：

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1,
return_indices=False, ceil_mode=False) # 最大池
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False,
count_include_pad=True) # 平均池
```

常用的参数如下：

kernel_size：窗口大小，即上文的 k

stride：步长，默认值是 kernel_size

激活层：

这一层较为简单，对矩阵的每一个元素进行激活函数处理即可，常用的激活函数为 ReLU：f(x) = max(x,0)

pytorch ReLU 激活层函数如下：

```
torch.nn.ReLU()
```

批规范化（BN）层：

批规范化是指在每次随机梯度下降时，通过 Mini-batch 操作来对相应的激活做规范化操作， 使得结果的均值为 0，方差为 1。

这一层一般都能起到精准度提高的作用，因此本次实验中也配备了 BN 层，放在每个激活层的前面

pytorch BN 层函数如下：

```
torch.nn.BatchNorm2d(self, num_features, eps=1e-5, momentum=0.1, affine=True,
track_running_stats=True)
```

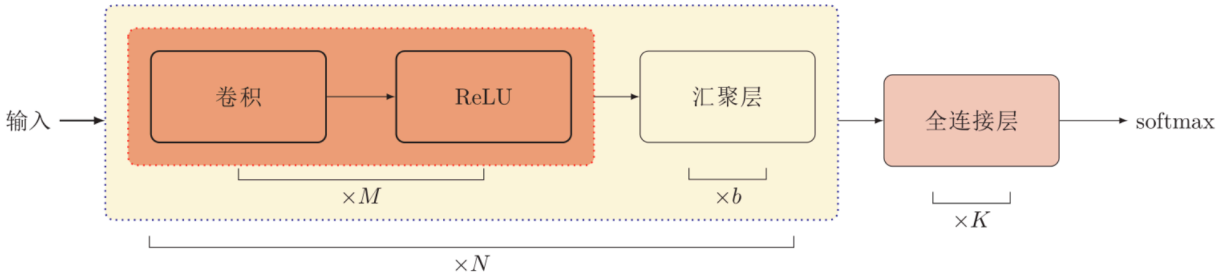
常用的参数如下：

num_features：输入的通道大小

其它参数参照本身的默认值即可

标准 CNN：

标准 CNN 模型由 N 个卷积块（每个卷积块又由 M 个卷积层+激活层、b 个池化层构成）和 K 个全联接层组成，最后输出可以通过 softmax 函数得到图片分别属于每个类别的概率，可以通过最大概率的类别确定该图片预测的类别（其中 M 通常为 2～5，b 为 0 或 1，K 为 0～2，N 可以取 1 到任意大）



训练：

CNN 的训练和上个实验的全连接神经网络类似：正向传播 -> 输出层计算误差 -> 误差反向传播 -> 迭代足够的次数。CNN 的误差反向传播十分复杂，然而 pytorch 有封装好的函数供我们调用：

```
CNN.train() # nn.Module训练模式
loss_func = nn.CrossEntropyLoss() # 损失函数：交叉熵
optimizer = torch.optim.SGD(CNN.parameters(), lr = 0.01, momentum = 0.9,
weight_decay = 5e-4) # 优化方式：SGDM
out = CNN(data) # 正向传播
loss = loss_func(out, target) # 计算误差
optimizer.zero_grad() # 清空上一步的残余更新参数值
loss.backward() # 误差反向传播
optimizer.step() # 更新参数
```

验证：

由于 CNN 理论上输出每个标签的概率，因此最终标签的判断只需要从所有概率中选择最大值那个即可。得到预测标签之后，和原标签进行对比判断，即可得到预测准确率

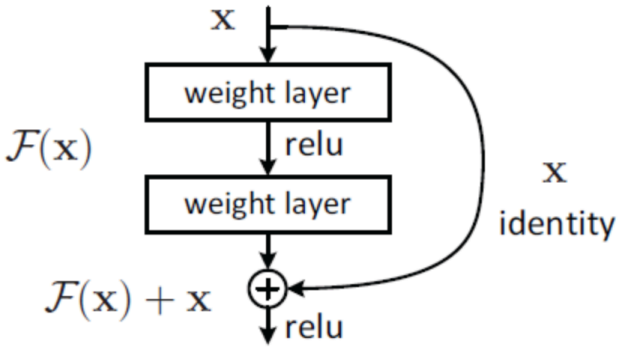
（值得说明的是，源代码中并不通过 softmax 层，因此 CNN 实际输出的值并不是对应标签的概率，但是从选择最大值来看，不通过 softmax 层不会有任何影响，如果选择加入通过 softmax 层的操作反而使得训练和收敛速度均变慢）

```
CNN.eval() # nn.Module验证模式
correct, total = 0, 0
for data, target in test_loader: # 遍历验证集
    data, target = data.to(device), target.to(device) # 数据、标签(GPU)
    out = CNN(data) # 正向传播
    _,predicted = torch.max(out.data, 1) # 返回每一行中的最大值及其索引
    total += target.size(0)
    correct += (predicted == target).sum().item()
print('Test Accuracy: %.2f%%' % (100*correct/total))
```

残差网络（ResNet）：

原本是试图通过修改恒等映射函数 $H(x)=x$ 为残差函数 $F(x)=H(x)-x$ 来解决深层网络退化问题的一种网络，后来发现在图像识别方面效果明显，因此本次实验也采用 4 种典型的残差网络进行对比尝试

残差网络的建立主要依赖于残差块：



可以发现，残差块的输出等价于 $\mathrm{Conv}(\mathrm{Relu}(\mathrm{Conv}(x)))+x$（部分块涉及降维操作）

```
class BasicBlock(nn.Module): # 残差块
    # 此处省略部分代码
    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x))) # 1:卷积->BN->激活
        out = self.bn2(self.conv2(out)) # 2:卷积->BN
        out += self.shortcut(x) # 残差
        out = self.relu(out) # 激活
        return out
```

而残差网络也通过残差层/残差块的叠加来实现，常见的残差网络由 4 层残差层（分别为 64、128、256、512 通道，每层由多个残差块叠加形成）

最后在网络头部加入一个卷积层，网络尾部加入一个平均池化层和一个全连接层形成最终的网络

```
class ResNet(nn.Module):
    # 此处省略部分代码
    def _make_layer(self, num_block, planes, stride): # 新建残差层
        strides, layers = [stride]+[1]*(num_block-1), []
        for stride in strides:
            layers.append(BasicBlock(self.in_planes, planes, stride)) # 每层
num_block个残差块
            self.in_planes = planes
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x))) # 卷积->BN->激活
        out = self.layer1(out) # 4层残差层
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
```

```
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1) # 展开成一维
        out = self.linear(out) # 全连接层
        return out
```

WideResNet（WRN）：

对于普通的 ResNet 来说，一味的增加深度可能精确率只能提高少许，而当深度达到一定阶级时，训练时间过长，且仍可能产生梯度弥散现象。因此 WideResNet 模型探究网络的宽度对网络的影响

可以简单地理解为：**扩展残差层的通道数**而不只是残差块的个数

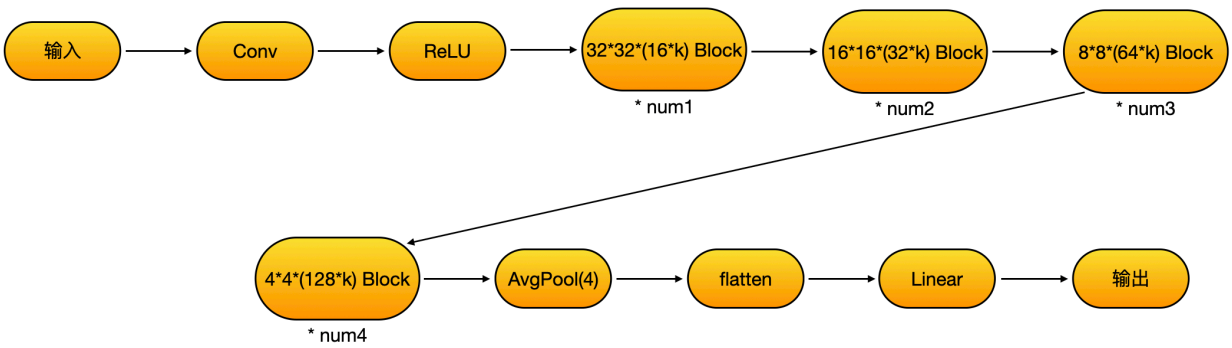
```
class WideResNet(nn.Module):

    def __init__(self, k, num_blocks): # k为宽度扩展倍数
        # 此处省略部分代码
        self.layer1 = self._make_layer(num_blocks[0], 16*k, 1) # 残差层1
        self.layer2 = self._make_layer(num_blocks[1], 32*k, 2) # 残差层2
        self.layer3 = self._make_layer(num_blocks[2], 64*k, 2) # 残差层3
        self.layer4 = self._make_layer(num_blocks[3], 128*k, 2) # 残差层4
        # 此处省略部分代码
```

本次实验的网络结构：

本次实验我采用了 5 种 CNN 网络结构进行尝试，分别是：

- 标准 CNN（其中 M = 1，b = 1，N = 3，K = 3），另外 3 个全连接层之间再加上 ReLU 激活层
- ResNet18（k = 4，4 层分别由 2、2、2、2 个残差块构成）
- ResNet34（k = 4，4 层分别由 3、4、6、3 个残差块构成）
- WideResNet18（k = 10，4 层分别由 2、2、2、2 个残差块构成）
- WideResNet34（k = 10，4 层分别由 3、4、6、3 个残差块构成）

标准 CNN 模型结构图可以参照上文附图；ResNet 和 WideResNet 的网络结构图可以参照下图：



创新点：

本次实验采取的创新点如下：

- 加入 BN 层优化准确率
- 尝试高级网络结构：ResNet 和 WideResNet 等
- 尝试学习率阶段式下降方案
- 尝试加入数据增强：随机扩展裁剪、随机水平翻转、随机垂直翻转等
- 尝试加入随机失活（dropout）

前两点的原理已在前文说明，现在介绍后三点

学习率阶段式下降：当某个学习率在验证正确率趋于稳定时，将学习率除以 10，继续进行学习。测试过程中发现有明显的正确率提升，因此本次试验中均采用该方案进行优化，并初始化为：前 20 次学习率为 0.01，20～30 次学习率为 0.001，30～40 次学习率为 0.0001，共迭代更新 40 次（后续会有增强）

数据增强：对于图像处理任务，可以通过将图像进行一定的变换操作，改变图像的某些无关特征，增强神经网络对关键特征的识别

常用的图像数据增强方法有随机扩展裁剪、随机水平翻转、随机垂直翻转、随机旋转、缩放等等（当然并不是所有的方式都能对某些特定的数据集或模型起到作用）

而采用这种技术也基于 CNN 对移位、旋转等具有不变性的性质

随机失活（dropout）：随机让某些神经元失活，防止模型依赖某些局部的特性，可以一定程度上避免过拟合，提升模型的泛化能力

但是有研究表明 dropout 层和 BN 层共同作用时效果可能会降低，测试中也是如此，因此最终的结果采用 BN 层，并不采用 dropout 层

实验结果及其分析：

首先是最常见的模型：对其分别加入不同的优化方法并测试验证准确率：

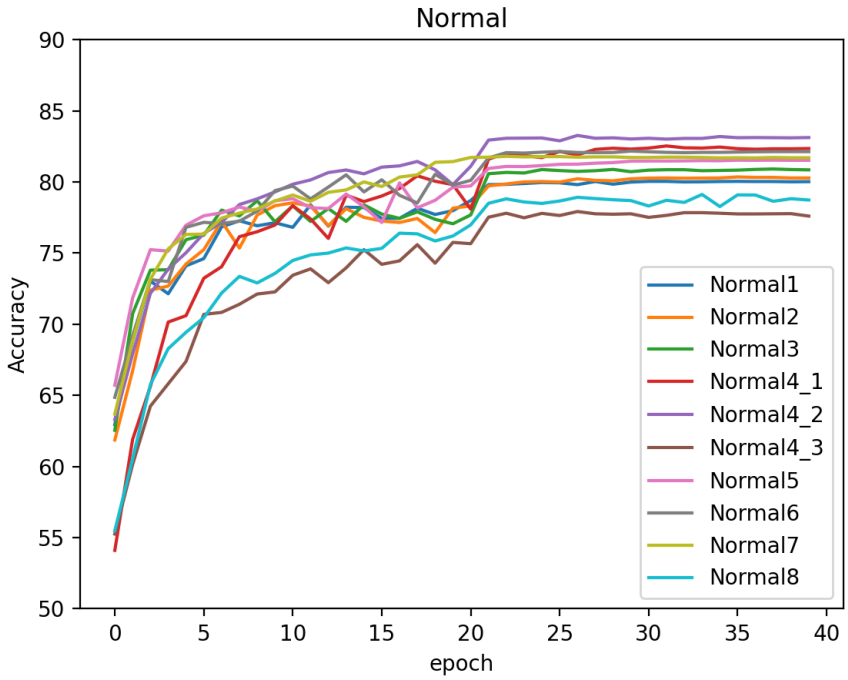
1. 对数据进行标准归一化（基准）
2. 调整为适合 cifar-10 的归一化参数
3. 调整为合适的包装数据集大小
4. 数据增强：随机扩展裁剪、随机水平翻转、随机垂直翻转
5. 额外添加一层卷积层和 BN 层
6. 增大数据通道数
7. 增大全连接层神经元数
8. 增加随机失活层

八种操作的代码体现为：

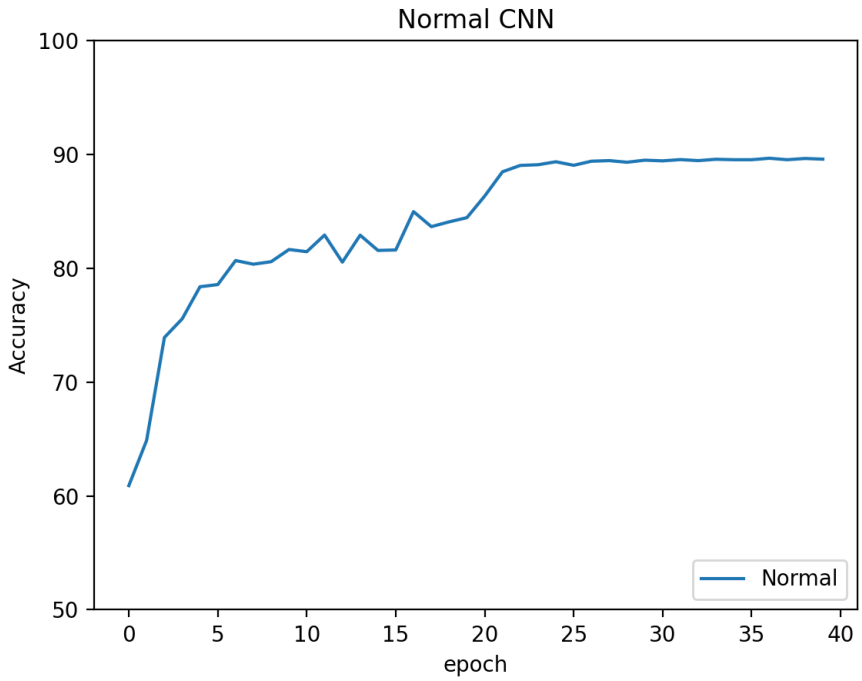
```
# 标准归一化
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
# 适合 cifar-10 的归一化参数
transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
# 调整包装数据集大小
train_loader = torch.utils.data.DataLoader(train_dataset, shuffle = True,
batch_size = 64, num_workers = 2) # batch_size 原本是128
```

```
# 随机扩展裁剪、随机水平翻转、随机垂直翻转
transforms.RandomCrop(32, padding = 4)
transforms.RandomHorizontalFlip()
transforms.RandomVerticalFlip()
# 额外添加一层卷积层和 BN 层
out = self.relu(self.bn4(self.conv4(out))) # 4:卷积->BN->激活
# 增大数据通道数
self.conv1 = nn.Conv2d(3, 64, kernel_size = 3, padding = 1) # 原本是(3, 32)
# 增大全连接层神经元数
self.fc1 = nn.Linear(128*4*4,1024) # 原本是(128*4*4,120)
# 增加随机失活层
self.dropout = nn.Dropout(0.5)
```

测试结果如下图（每次测试只添加上述一种优化方式，第四种又分三次进行测试）：



可以看出，以方法 1（深蓝色线条）为基准，除了优化方法 4_3 和方法 8，即随机垂直翻转和增加随机失活层没有起到优化效果之外，其它方法都或多或少起到了一定的优化。因此将这些起到优化效果的所有方案整合，进行标准 CNN 模型的进一步测试：

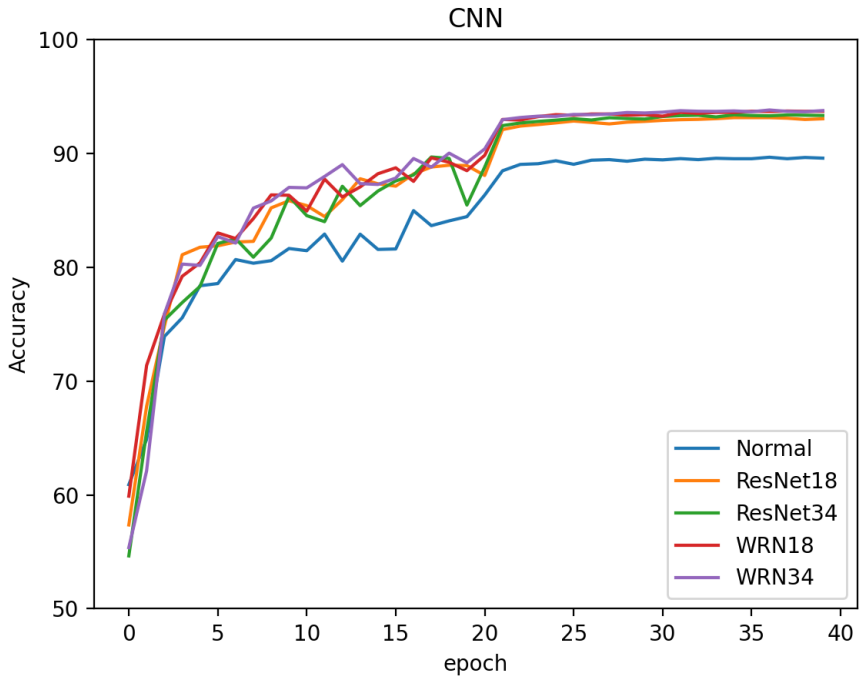


当前标准 CNN 测试准确率为 89.67%。

```
[3]: run CNN/test_Normal.py

Test Accuracy: 89.67%
```

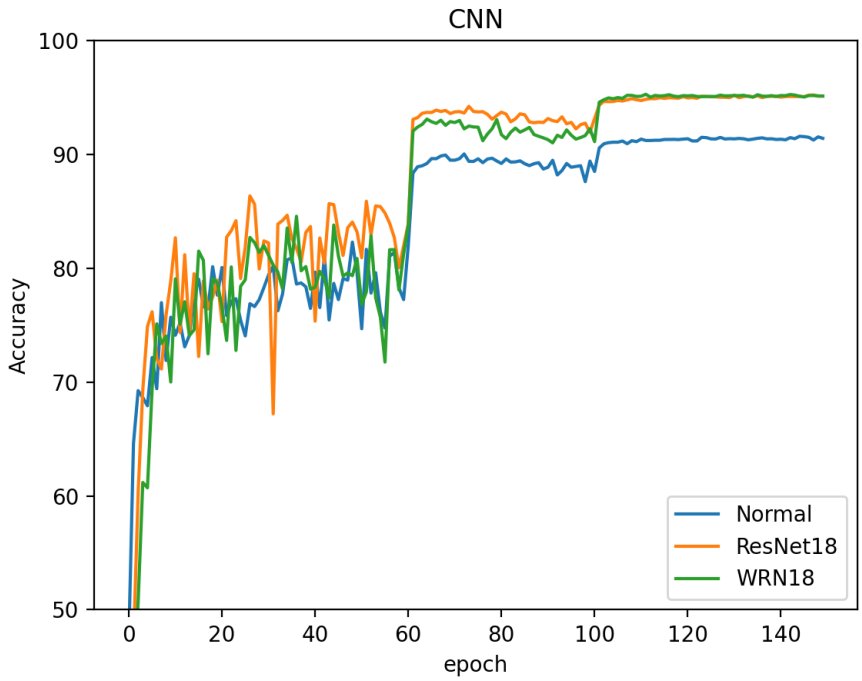
之后是四种高级模型的测试：对 ResNet 来说，可以适用的优化方案为上述方案 2～4（残差网络除残差块仅头部有一层卷积层，尾部有一层全连接层，而测试中发现对该卷积层和全连接层采用上述方案 5～7 并没有取得优化效果），而 ResNet18 为基准模型，ResNet34 是对其的深度优化，WideResNet18 和 WideResNet34 是对前两者的宽度优化

四种高级模型的测试结果如下：



可以看出 WRN34 结果最好（93.82%），深度优化和宽度优化都起到了一定的效果，相比之下宽度优化效果要更明显

继续考虑对标准 CNN、ResNet18 和 WideResNet18 强化“学习率阶段式下降方案”为：前 60 次学习率为 0.1，60～100 次学习率为 0.01，100～135 次学习率为 0.001，135～150 次学习率为 0.0001，共迭代更新 150 次。新的结果如下：



验证准确率得到大幅上升，标准 CNN 验证准确率达到 91.6%，ResNet18 验证准确率达到 95.23%，WRN18 验证准确率达到 95.27%，均比之前的结果高出 2% 左右

由于训练时间过长（GPU 训练一次四个小时以上），不进行 ResNet34 和 WRN34 的进一步训练（预测优化效果不会超过 0.1%），因此**本次试验最终验证准确率为 95.27%**

```
[8]: run CNN/test_WideResNet18.py

     Test Accuracy: 95.27%
```

任务二：

实现 RNN 循环神经网络，完成关键词提取任务（Subtask1: Aspect Term Extraction）

数据集：SemEval-2014，Laptop

算法原理：

问题归约：

本次实验的具体内容为：读取一个句子，提取出里面的“关键词”。表面上使用神经网络的方式实现这个过程似乎比较困难：句子的关键词数量不统一，输出层神经元无法固定数量。但是我们可以采取以下等价的变换，使得实验更加易于实现：

读取一个句子，判断每个词语是否属于关键词

那么这个问题就被归约成为一个**二分类问题**，且神经网络的输出也将会和句子长度等长，符合循环神经网络的基本形式

数据读取：

本次实验数据为 .xml 格式，虽然可能存在更专业的文件读取方式，但是用普通的逐行读取根据结构判断当然也是一种可行的方案。

我们需要读取的有效数据：

1. <text> 与 </text> 围着的“评论”，也就是关键词所在的句子
2. <aspectTerm/> 内的 term 部分，也就是关键词

实际上关键词提取任务用到的就只有这些数据

在数据读取的过程之中，也需要对其进行一些基础的预处理：

1. 将句子分割成单词列表，单词转换为全小写
2. 去除无用字符（实验过程中发现句子中存在 @、*、¥、#、% 等无用字符，这些显然是评论中无意义的部分，且会影响单词的识别匹配）
3. 判断关键词：对单词列表每一个单词进行判断，1 为关键词，0 为非关键词
4. 转化为**词向量**：本次试验采用 glove.6B.50d/glove.6B.300d 两种词向量进行实验

因此数据读取的全过程为：读取句子和关键词 -> 分割句子，去除无用字符 -> 匹配关键词 -> 将单词转换为词向量 -> 返回单词列表和关键词 01 列表

附上部分关键代码（词向量部分）：

```
with open('RNN/glove.6B.50d.txt','r') as f: # 读取词向量形成字典
    dic = {}
    for l in f.readlines():
        s = l.strip('\n').split(' ')
        dic[s[0]] = [float(x) for x in s[1:]] # 形成单词到向量的映射字典
for i in range(len(data)):
    for j in range(len(data[i][0])):
        if data[i][0][j] in dic: data[i][0][j] = dic[data[i][0][j]] # 如果字典有这个单词，则转换成对应的词向量
        else: data[i][0][j] = [0 for x in range(50)] # 没有该单词，则默认为全0
```

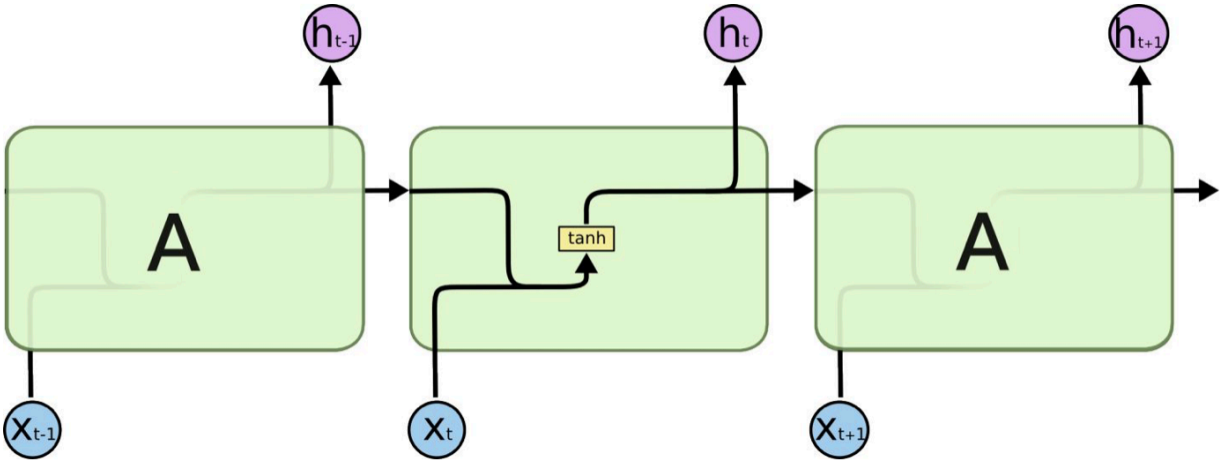
循环神经网络（RNN）：

一种特殊的神经网络（本质上也是数学模型）。和普通的神经网络一样，分为输入层、隐藏层、输出层。而隐藏层比较特殊，将会接受上一次该神经元的输出 h_{t-1} 作为第二个输入，和原本的输入 x_t 共同影响神经元这次的输出 h_t

标准 RNN：

一个标准 RNN 模型由一个输入神经元、一个隐藏神经元、一个输出神经元组成。但由于其循环特性，可以展开为多个输入、隐藏、输出神经元，展开的个数与输入数据的维度（本次实验可以理解为一个句子的长度/单词的个数）相同。

隐藏神经元的处理与全连接层类似：分别对输入 x 和输入 h_{t-1} 做全连接层形成 $\mathrm{W}_{xh}\mathrm{x}_t$ 和 $\mathrm{W}_{hh}\mathrm{h}_{t-1}$，最后将两者与两个偏置量 b_x 和 b_h 求和后通过激活函数（一般是 tanh）得到输出 h_t



$$\mathrm{h}_t = \tanh(\mathrm{W}_{xh}\mathrm{x}_t + \mathrm{b}_x + \mathrm{W}_{hh}\mathrm{h}_{t-1} + \mathrm{b}_h)$$

输出神经元的处理也是与全连接层类似：对输入 h_t 做全连接层形成 $\mathrm{W}_{hz}\mathrm{h}_t$，最后将其与偏置量 b_z 求和后通过激活函数（一般是 softmax）得到输出 z_t，即该单词为非关键词和关键词的概率

$$\mathrm{z}_t = \mathrm{softmax}(\mathrm{W}_{hz}\mathrm{h}_t + \mathrm{b}_z)$$

Pytorch RNN 类源码如下：

```
torch.nn.RNN(self, *args, **kwargs)
```

常用的参数如下：

input_size：输入的特征数量，此处为词向量大小（50 or 300）

hidden_size：隐藏层的特征数量，一般可以随意设置

num_layers：堆叠数量，默认值为 1

bias：是否加入偏置量，默认值为 True

batch_first：输入方式，如果为 True 则输入方式为（batch，序列长度，输入维度），符合 CNN 的习惯，默认值为 False

dropout：如果非零，除了最后一层外，在每个 RNN 层的输出上引入一个 dropout 层，其 dropout 概率等于参数值，默认值为 0

bidirectional：如果为 True，则成为一个双向 RNN，默认值为 False

训练：

RNN 的训练和上个实验的全连接神经网络类似：正向传播 -> 输出层计算误差 -> 误差反向传播 -> 迭代足够的次数。RNN 的误差反向传播十分复杂，且比 CNN 需要多考虑时序演化过程，然而 pytorch 有封装好的函数供我们调用：

```
RNN.train() # nn.Module训练模式
loss_func = nn.CrossEntropyLoss() # 损失函数：交叉熵
optimizer = torch.optim.SGD(CNN.parameters(), lr = 0.01, momentum = 0.9,
weight_decay = 5e-4) # 优化方式：SGDM
out = RNN(data) # 正向传播
loss = loss_func(out, target) # 计算误差
optimizer.zero_grad() # 清空上一步的残余更新参数值
loss.backward() # 误差反向传播
optimizer.step() # 更新参数
```

如果比较细心的话，你会发现 RNN 与 CNN 原理虽然有很大区别，但是这一段代码和 CNN 几乎是完全一致的

验证：

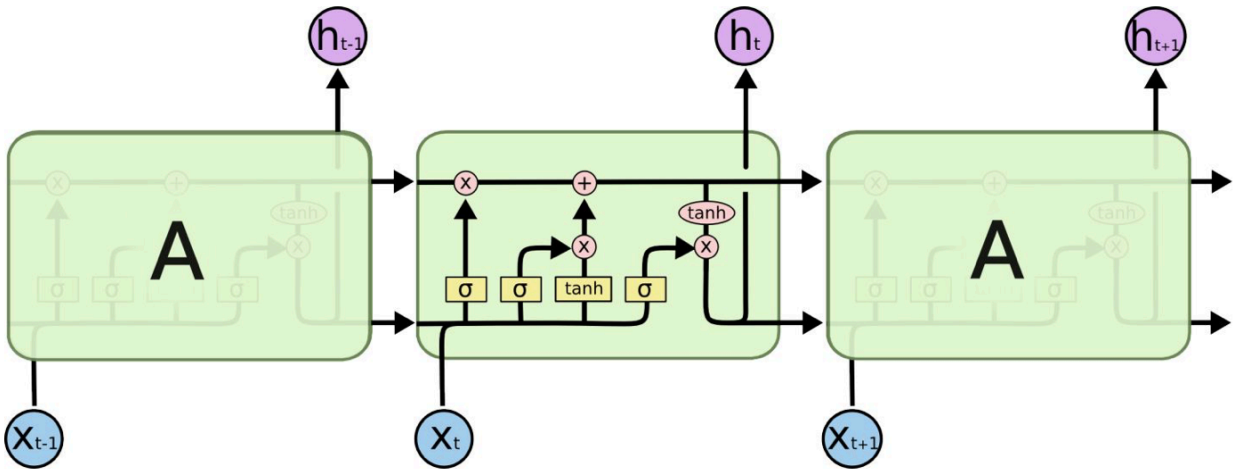
由于 RNN 输出每个单词是否为关键词的对数概率，因此最终是否为关键词的判断只需要从两个概率中选择最大值那个即可。得到预测结果之后，和原结果进行对比判断，即可得到预测准确率

```
RNN.eval() # nn.Module验证模式
correct, total = 0, 0
for data, target in test: # 遍历验证集
    data, target = torch.FloatTensor(data).to(device),
torch.LongTensor(target).to(device) # 数据、标签(GPU)
    out = RNN(data) # 正向传播
    _,predicted = torch.max(out.data, 1) # 返回每一行中的最大值及其索引
    predicted = predicted.cpu().numpy()
    total += len(target)
    for j in range(len(target)):
        if predicted[j] == target[j]: correct += 1
print('Test Accuracy: %.2f%%' % (100*correct/total))
```

长短期记忆网络（LSTM）：

在标准 RNN 中，这个重复的结构模块只有一个非常简单 tanh 层，而这种简单级别的 RNN 只能根据以前比较近的信息进行推断，而如果需要利用比较远的信息，标准的 RNN 就很难起到效果。

长短期记忆网络（LSTM），则是一种为了解决长依赖问题的特殊 RNN 网络。相比于标准 RNN，其隐藏层具有四个网络层：



细胞状态：标准 RNN 除了循环传递 h_{t-1} 外，还传递一个“细胞状态” C_{t-1}，保存序列的历史信息。细胞状态通常改变的很慢，因此也可以做到保存长距离信息的效果

遗忘门：LSTM 的第一层，决定丢弃细胞状态的某些信息。它通过查看 h_{t-1} 和 x_{t-1} 信息来输出一个 0~1 之间的向量 f_t，该向量里面的 0~1 值表示细胞状态 C_{t-1} 中的哪些信息保留或丢弃多少，0 表示不保留，1 表示都保留。

得到该向量之后，将其与细胞状态进行**点乘**操作，即实现信息的遗忘操作：

$$
\begin{aligned}
f_t &= \mathrm{sigmoid}(\mathrm{W}_f[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_f) \\
C'_{t-1} &= C_{t-1} \cdot f_t
\end{aligned}
$$

输入门：LSTM 的第二、三层，决定给细胞状态添加某些新信息。它通过查看 h_{t-1} 和 x_{t-1} 信息来输出一个 0~1 之间的向量 i_t 和新的候选细胞信息 $\tilde{C}_t$，然后通过该向量里面的 0~1 值表示新的候选细胞信息 $\tilde{C}_t$ 中的哪些信息加入多少。

得到加入的信息量之后，将其与细胞状态进行**求和**操作，即实现信息的输入操作：

$$
\begin{aligned}
i_t &= \mathrm{sigmoid}(\mathrm{W}_i[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_i) \\
\tilde{C}_t &= \tanh(\mathrm{W}_c[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_c) \\
C_t &= C'_{t-1} + i_t * \tilde{C}_t
\end{aligned}
$$

在遗忘门和输入门工作结束之后，细胞状态也得到了**更新**

输出门：LSTM 的第四层，输出 h_t。它通过查看 h_{t-1} 和 x_{t-1} 信息来输出一个 0~1 之间的向量 O_t，然后通过该向量里面的 0~1 值表示新的细胞状态 C_t 中的哪些信息输出多少。

$$
\begin{aligned}
O_t &= \mathrm{sigmoid}(\mathrm{W}_o[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_o) \\
\mathrm{h}_t &= O_t * \tanh(C_t)
\end{aligned}
$$

因此总的来说，LSTM 一个神经元的步骤可以总结为四步：遗忘、输入、更新、输出。

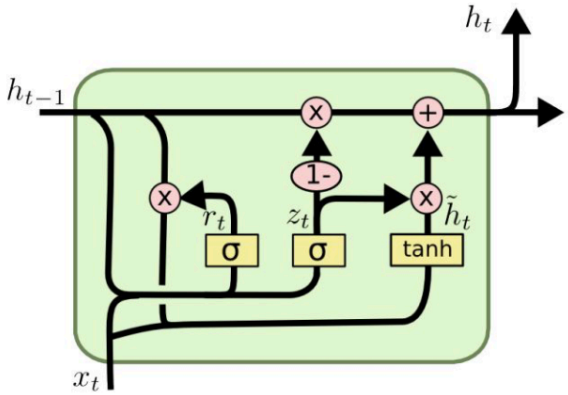
Pytorch LSTM 类源码如下：

```
torch.nn.LSTM(self, *args, **kwargs)
```

常用的参数与 RNN 类相同

循环门单元（GRU）：

GRU 是 LSTM 的一个变种，它将 LSTM 的遗忘门和输入门组合成为一个单独的**更新门**，合并细胞状态 C_{t-1} 和 h_{t-1}，虽然原理和结果都与 LSTM 差别不大，但是模型的参数大大减少，训练起来也更加快速，因此也逐渐开始作为 LSTM 的优化替代品。



$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \\
\tilde{h}_t &= \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
$$

其中 σ 为激活函数 $\mathrm{sigmoid}$，1～3 式省略了偏置量的表述

由于 GRU 与 LSTM 原理类似（只是门操作不同），因此本文不再对其细节操作进行赘述

Pytorch GRU 类源码如下：

```
torch.nn.GRU(self, *args, **kwargs)
```

常用的参数与 RNN 类相同

堆叠/双向 RNN：

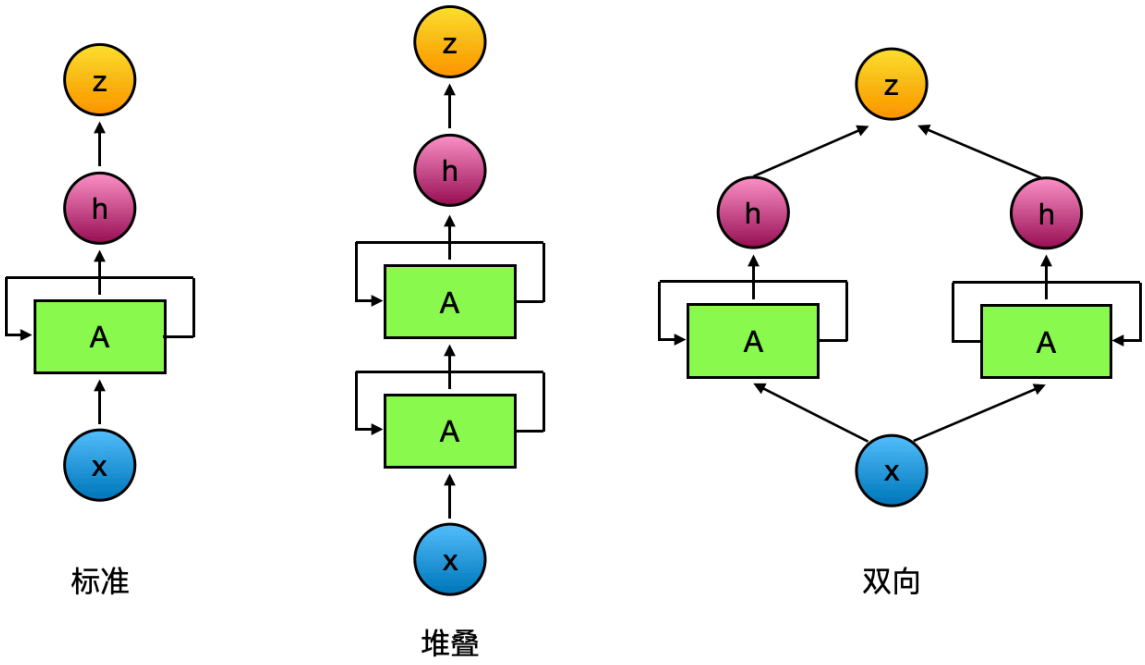
堆叠 RNN：一般的 RNN 为神经元横向连接形成序列形状，而堆叠 RNN 则再在纵向进行多层堆叠。比如说第一层的输出 h_t 将会成为第二层的输入 x_t，以此类推。

双向 RNN：对于本次实验，单词序列关键词的判断不仅与前文有关，而且更大程度上与后文有关（因为情感词语一般在主语后方），因此将单词序列反向经过一遍 RNN 会起到更好的识别效果。双向 RNN 则是将单词序列正向反向均通过一遍 RNN，以求捕捉到双向的特征信息。

本次实验的网络结构：

本次实验我采用了 3 种 RNN 网络结构进行尝试（标准、堆叠、双向），也尝试了 3 种神经元结构（标准 RNN、LSTM、GRU）

3 种网络结构可以参考下图，3 种神经元结构可以参考上文附图



网络将从输入层 x 输入一个单词序列，从输出层 z 输出 01 序列，分别对应单词序列中的每一个单词是否为关键词

创新点：

本次实验采取的创新点如下：

- 尝试高级网络结构：堆叠 RNN 和双向 RNN
- 尝试高级神经元结构：LSTM 和 GRU
- 加入 glove 词向量取代 nn.Embedding 训练层
- 尝试学习率阶段式下降方案
- 尝试加入随机失活（dropout）

前两点的原理已在前文说明，现在介绍后三点

glove 词向量：上文提到需要将单词转换为向量，才能传入神经网络进行训练。虽然 pytorch 中已存在 nn.Embedding 训练层，用于将单词转换为向量，但它在神经网络训练过程中同时进行训练，需要消耗时间。因此直接采用已训练好的 glove 词向量进行转换，不仅可以节省训练词向量的时间，还可以提高精度

学习率阶段式下降：当某个学习率在验证正确率趋于稳定时，将学习率除以 10，继续进行学习。测试过程中发现有明显的正确率提升，因此本次试验中均采用该方案进行优化，并初始化为：前 20 次学习率为 0.01，20～30 次学习率为 0.001，30～40 次学习率为 0.0001，共迭代更新 40 次

随机失活（dropout）：随机让某些神经元失活，防止模型依赖某些局部的特性，可以一定程度上避免过拟合，提升模型的泛化能力

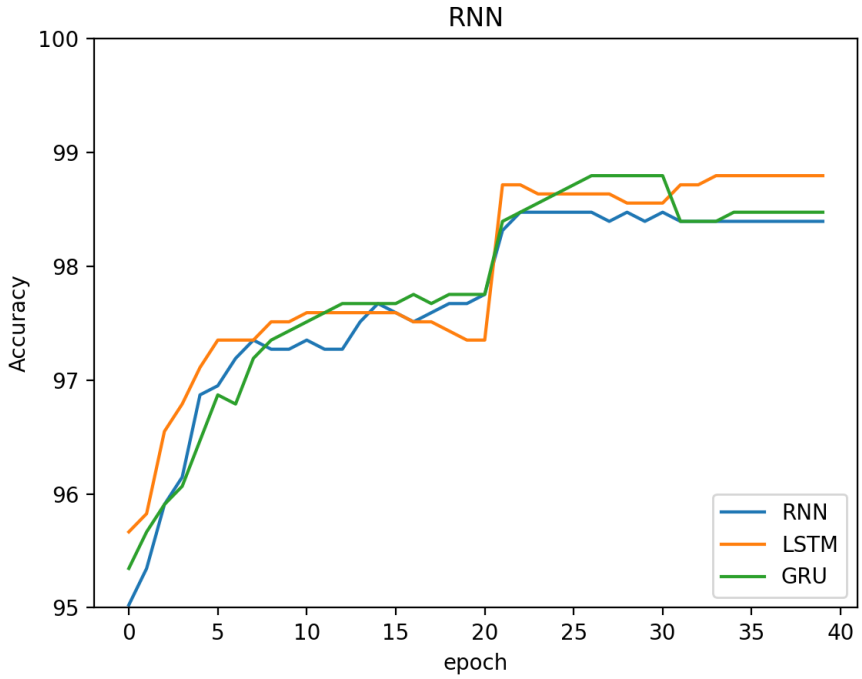
实验结果及其分析：

对于验证文件 Laptops trial data (updated on 17/12/13)：

首先是 3 种神经元结构（标准 RNN，LSTM，GRU），其余的参数均设置如下：

参数	值
词向量大小	50
隐藏层特征数量	128
是否堆叠	False
是否双向	False
是否加入随机失活	False

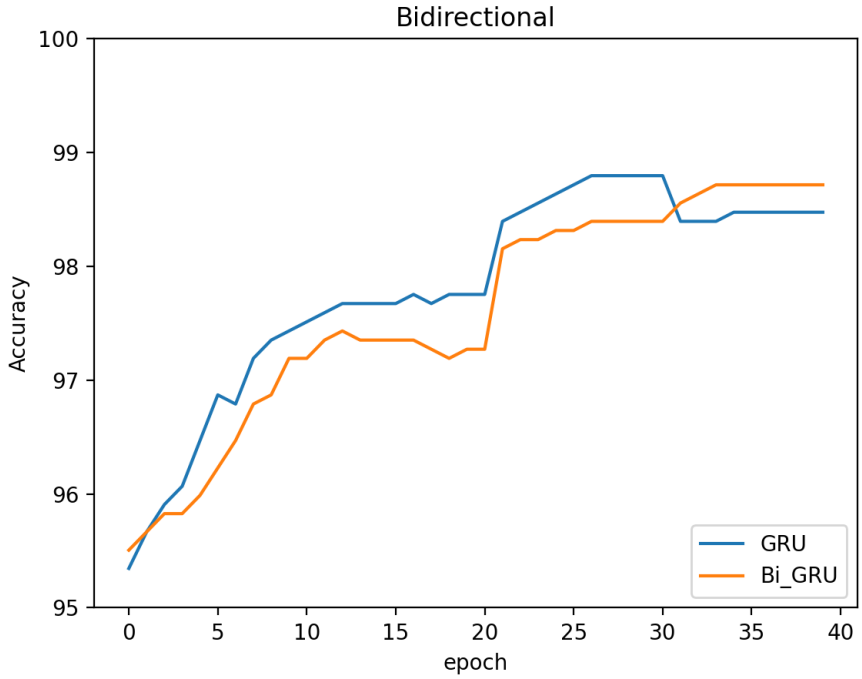
实验结果如下图：



可以看出 LSTM 和 GRU 的准确率都高于标准 RNN，且两者最高的准确率没有区别（均为 98.8%），因此采用参数更少，训练更快的 GRU

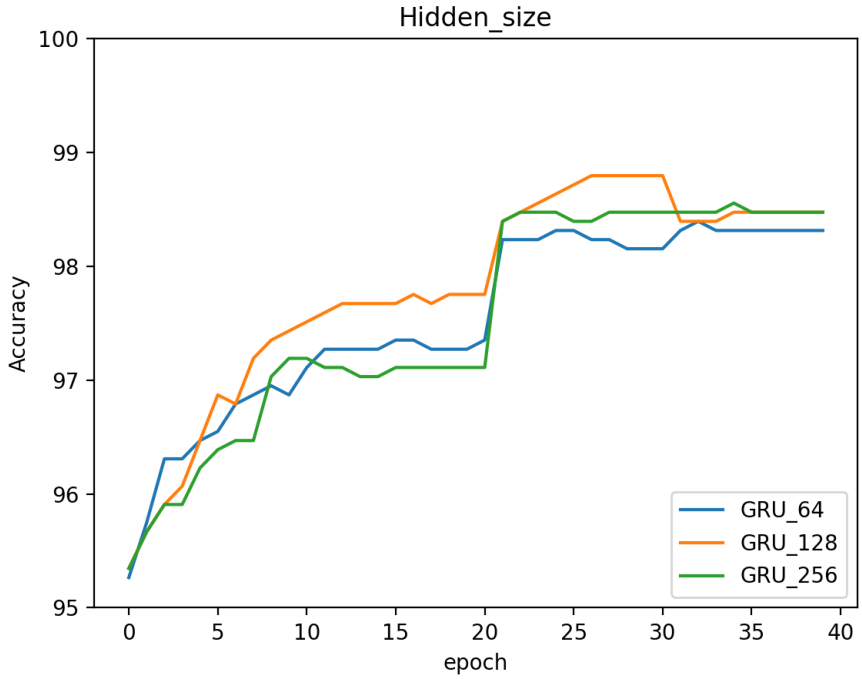
之后对 GRU 进行调参，试图让准确率变得更高：

尝试采用双向 RNN，结果如下：



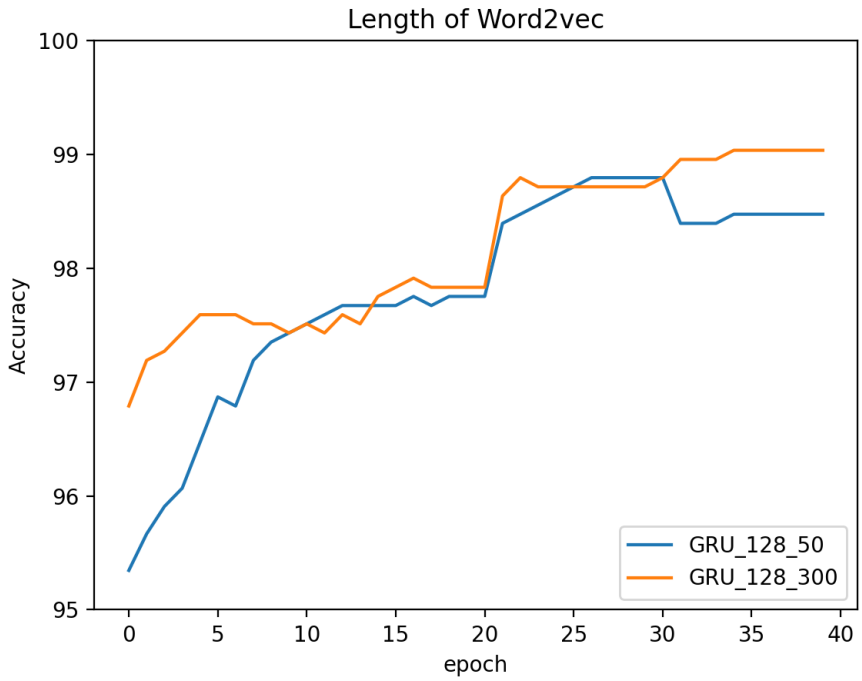
结果不升反降。

在上文基础上，尝试修改隐藏层特征数量，结果如下：



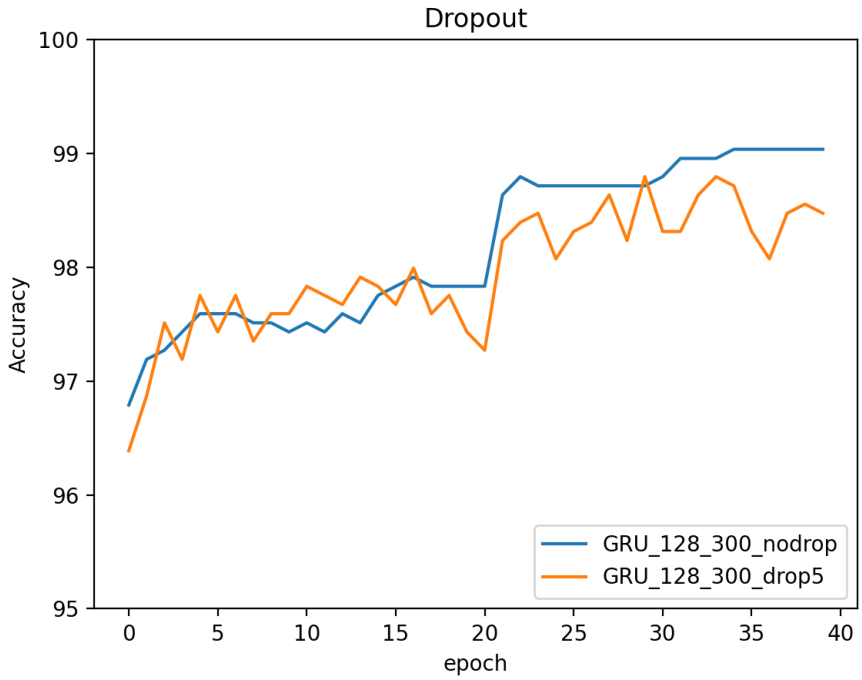
当隐藏层特征数量为 128 时，验证准确率较高（98.8%）

在上文基础上，将词向量大小修改为 300（即采用 glove.6B.300d.txt）优化结果如下：



准确率优化至 99.03%

在上文基础上，尝试加入 dropout 层（dropout 层需要搭配堆叠 RNN，因此此处堆叠 RNN 为 2 层），结果如下：

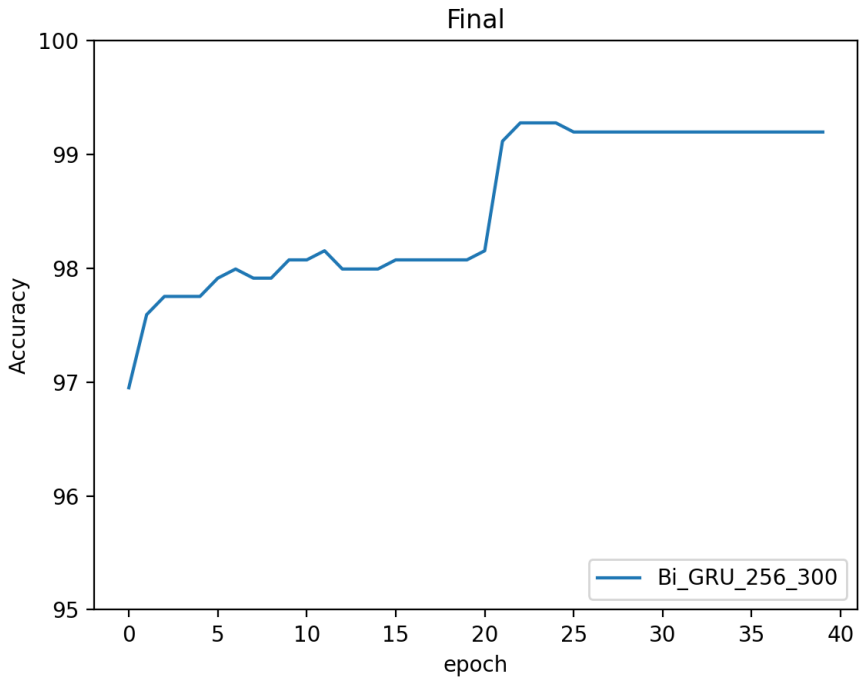


结果不升反降。

在上文基础上，尝试强化“学习率阶段式下降方案”为：前 60 次学习率为 0.1，60～100 次学习率为 0.01，100～135 次学习率为 0.001，135～150 次学习率为 0.0001，共迭代更新 150 次。新的验证结果基本不变，因此 RNN 不采用更长的训练迭代方案

尝试超参组合：虽然单独修改某种参数/单独采用某种方案可能会导致结果下降，然而采用某种组合反而会产生负负得正的效果。尝试组合过程十分繁琐，因此这里只放上最终结果：

参数	值
神经元	GRU
词向量大小	300
隐藏层特征数量	256
是否堆叠	False
是否双向	True
是否加入随机失活	False



最终最高的准确率为 99.28%

```
[4]: run RNN/test_Final.py

     Test Accuracy: 99.28%
```

下图为验证集前 10 条数据预测值和真实值，可以看出仅有一个单词判断失误，其余均判断正确

```
predicted :  [0 0 0 1 1]
target    :  [0 0 0 1 1]
predicted :  [0 0 0 1 0 0]
target    :  [0 0 0 1 0 0]
predicted :  [0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 0 0 0 0 0 0 0 0 0 0]
predicted :  [0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted :  [0 0 0 0 0 0]
target    :  [0 0 0 0 0 0]
predicted :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 0 0 0 0 1 0 0 0 0 0 0 0]
predicted :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
target    :  [0 0 0 0 0 0 0 0 0 0 0 0 0 0]
predicted :  [1 1 0 0 1 0]
target    :  [1 1 0 0 1 0]
```

对于验证文件 Test data: Phase B (released on 27/3/14)

调参过程与上述类似，故不再放上完整过程，最后的超参组合如下：**最终的准确率为 96.32%**

参数	值
神经元	GRU
词向量大小	300
隐藏层特征数量	256
是否堆叠	True（2）
是否双向	True
是否加入随机失活	True（0.5）

```
[1]:  run RNN/test_GRU_bid_256_drop5.py

      Test Accuracy: 96.32%
```

项目心得：

本次项目的题目看上去十分的简洁精炼：CNN 实现图片分类，RNN 实现关键词提取。但是整个过程下来，可谓是事情又多又难又容易跑偏

1. 神经网络框架的选择：CNN 和 RNN 那根本讲不清楚的 BP 算法，基本上直接断送不用框架实现的想法。那么对于现有的常见框架 pytorch（更简单，更适合科研）和 tensorflow (keras)（更通用，更适合商业），如果选了 tensorflow 据说要难入门很多

2. 框架模型的学习：CNN 要学 CNN 类、forward 函数、CNN 的搭建、Dataset 的构造（突然说不能直接用 CIFAR10 函数读取数据），RNN 要学 RNN 类，数据的传入（如何从 .xml 文件的数据转成能喂给神经网络的格式），总体来说 RNN 更难
3. 参考资料：CNN 的问题是常见的问题，网上已有不少的博客可供参考（包括理论和实际代码），RNN 则几乎没有参考资料，只能转换思路找类似的问题和文章参考思路（比如说参考“通过 RNN 实现词性判断”）
4. 训练时间过长：首先本课程是没有提供 GPU or CPU 环境的（幸运的是“高性能计算程序设计基础”课程给学生提供了 GPU 环境），然而在实验中训练一次 WideResNet18，150 轮就花费了 Tesla V100 近四个小时，如果只有自己电脑的 CPU 再慢至少一百倍，而且电脑还不一定能活得下来...

   因此 CNN 也放弃了对 ResNet34 和 WRN34 进行强化“学习率阶段式下降方案”的进一步训练（猜测不会优化超过 0.1%）

   RNN 还好，训练一遍基本只需要 GPU 20min
5. 为了使得 CNN 和 RNN 两部分更像是一个整体（更像是一个项目），我们虽然基本是各负责一部分，但也将代码和报告均整合为类似的格式，使得 CNN 和 RNN 代码能相似的部分（训练、验证过程、BPNN 三部曲等等）都将格式修改统一，报告的样式和模版看下来也感觉像是无缝切换的舒适～

小组分工：

18340146 宋渝杰：负责 CNN 部分，负责代码格式的整合统一

18340131 缪贝琪：负责 RNN 部分，负责报告格式的整合统一

参考资料：

CNN：

https://blog.csdn.net/sinat_42239797/article/details/90641659 定义自己的数据集及加载训练

https://blog.csdn.net/yanxueotft/article/details/97977754 Pytorch搭建CIFAR10的CNN卷积神经网络

https://blog.csdn.net/jining11/article/details/89114502 使用Pytorch搭建CNN

https://zhuanlan.zhihu.com/p/30117574 一文搞定Pytorch+CNN讲解

https://zhuanlan.zhihu.com/p/91477545 pytorch图像数据增强7大技巧

https://blog.csdn.net/winycg/article/details/86709991 pytorch实现ResNet

https://blog.csdn.net/jiangpeng59/article/details/79609392 pytorch笔记：04)resnet网络&解决输入图像大小问题

https://blog.csdn.net/sunqiande88/article/details/80100891 Pytorch实战2：ResNet-18实现Cifar-10图像分类（测试集分类准确率95.170%）

https://zhuanlan.zhihu.com/p/41679153 数据增强(Data Augmentation)

RNN：

https://www.cnblogs.com/picassooo/p/12544439.html PyTorch LSTM的一个简单例子：实现单词词性判断

https://www.jianshu.com/p/95d5c461924c 【译】理解LSTM（通俗易懂版）

https://zhuanlan.zhihu.com/p/34203833 深入理解lstm及其变种gru

https://pytorch.org/docs/stable/nn.html pytorch torch.nn 官方文档