南　开　大　学

计 算 机 学 院

深度学习及应用实验作业

作业二 卷积神经网络实践

姓名：徐宇昂

学号：2011742

年级：2020 级

专业：计算机科学与技术

指导教师：侯淇彬

2023 年 5 月

摘要

本次实验基于老师提供的原始 CNN 网络结构实现 ResNet 网络结构和 DenseNet 网络和 SE-resnet 结构，并实现 Cifar10 数据集分类。

关键字：卷积神经网络，pytorch，CNN

目录

一、 实验要求

- 掌握卷积的基本原理
- 学会使用 PyTorch 搭建简单的 CNN 实现 Cifar10 数据集分类
- 学会使用 PyTorch 搭建简单的 ResNet 实现 Cifar10 数据集分类
- 学会使用 PyTorch 搭建简单的 DenseNet 实现 Cifar10 数据集分类
- 学会使用 PyTorch 搭建简单的 SE-ResNet 实现 Cifar10 数据集分类

二、 原始 CNN 实现

1. CNN 网络结构

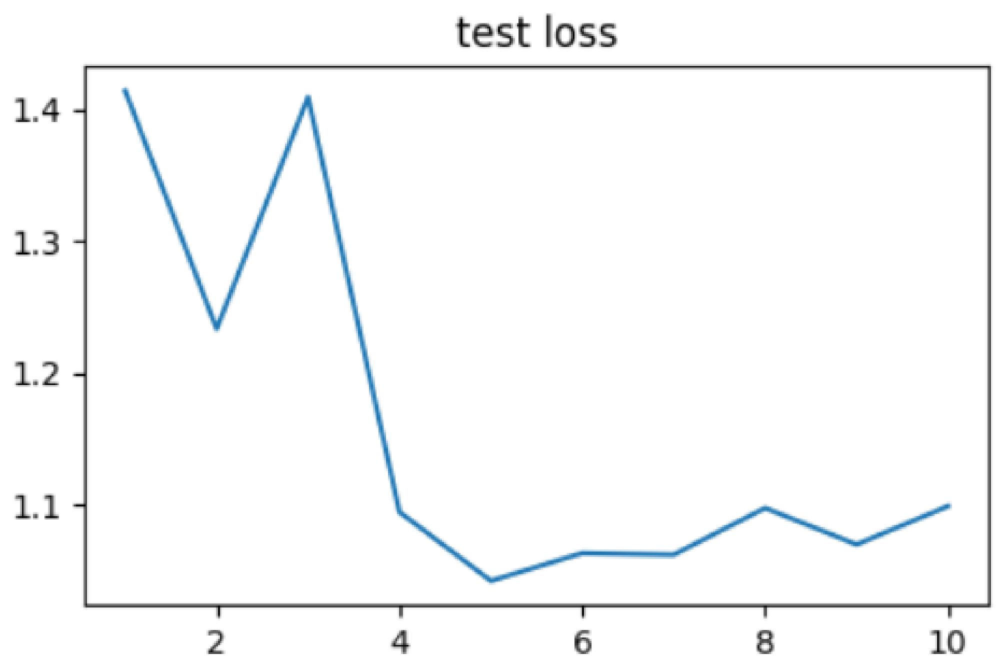
该 CNN 总共有 7 层构成，其中包含两个卷积层、两个最大池化层、三个全连接层，具体网络参数结构如下代码所示：

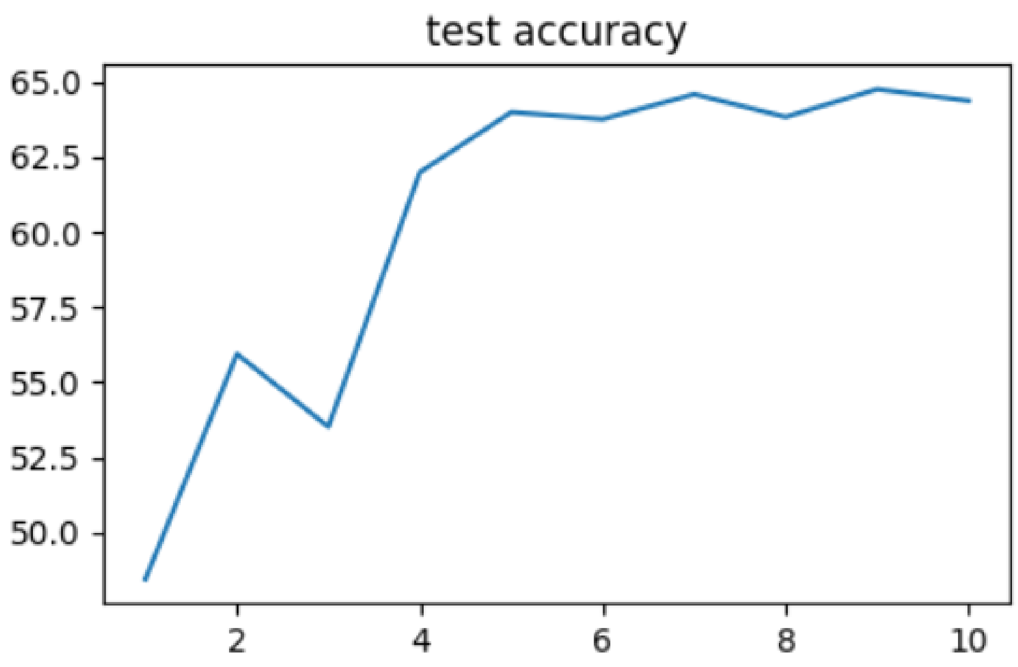
```python
def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)   # kernel_size=5, stride=1
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = self.conv1(x)
        #print(”output shape of conv1:”, x.size())
        x = F.relu(x)
        x = self.pool(x)
        #x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

第一层为 6 个大小 5x5 的卷积核构成的卷积层，步长为 1；经过 Relu 变换后第二层为大小为 2x2，步长为 2 的最大池化层；第三层为 16 个大小 5x5 的卷积核构成的卷积层，步长为 1；经过 relu 变换后第四层为大小为 2x2，步长为 2 的最大池化层；第五、六、七层为全连接层。

2. CNN 训练结果

如下图所示，展现出在测试集上的训练 loss 变化以及正确率曲线：





三、 ResNet **实现**

1. ResNet18 **网络结构**

ResNet18 网络结构是由残差网络构成，残差网络是由 He 等人提出的。2015 年解决图像分类问题。在 ResNets 中，来自初始层的信息通过矩阵加法传递到更深层。此操作没有任何附加参数，因为前一层的输出被添加到前面的层。具有跳跃连接的单个残差块如下所示：

如图所示

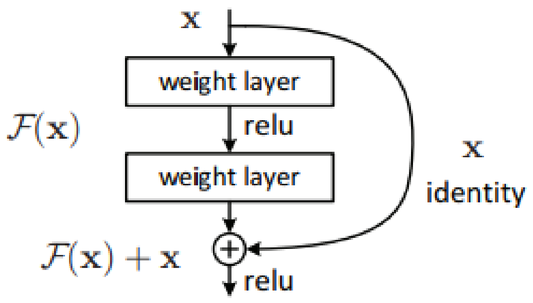

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

图 1: 残差块

由于 ResNet 的更深层表示，因为来自该网络的预训练权重可用于解决多个任务。它不仅限于图像分类，还可以解决图像分割、关键点检测和对象检测方面的广泛问题。而本次实验我所实现的是 ResNet18 的网络结构，18 代表权重层的数量，网络结构如图6所示

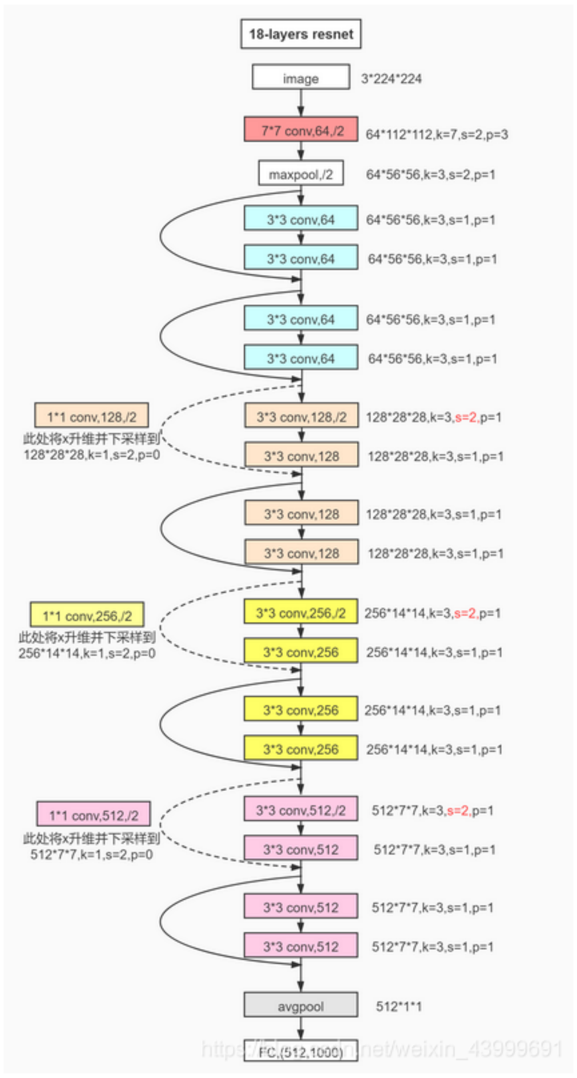


图 2: Caption

2. ResNet18 **代码**

首先是 basicblock 残差块的代码:

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

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )
        # 如果输入输出维度不等，则使用1x1卷积层来改变维度
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != self.expansion * out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, self.expansion * out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * out_channels),
            )

    def forward(self, x):
        out = self.features(x)
        out += self.shortcut(x)
        out = torch.relu(out)
        return out
```

然后是 ResNet 残差网络的代码实现:

```python
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        # cifar10经过上述结构后，到这里的feature map size是 4 x 4 x 512 x expansion
        # 所以这里用了 4 x 4 的平均池化
        self.avg_pool = nn.AvgPool2d(kernel_size=4)
        self.classifer = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        # 第一个block要进行降采样
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.features(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.classifer(out)
        return out
```

网络结构如下所示:

```
  ResNet(
(features): Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
)
(layer1): Sequential(
  (0): BasicBlock(
    (features): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (shortcut): Sequential()
  )
  (1): BasicBlock(
    (features): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (shortcut): Sequential()
  )
)
(layer2): Sequential(
    (0): BasicBlock(
      (features): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (features): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (features): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (features): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (features): Sequential(
        (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (features): Sequential(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (avg_pool): AvgPool2d(kernel_size=4, stride=4, padding=0)
  (classifer): Linear(in_features=512, out_features=10, bias=True)
)
```

3. ResNet18 的训练结果

loss/epochs 如图6所示

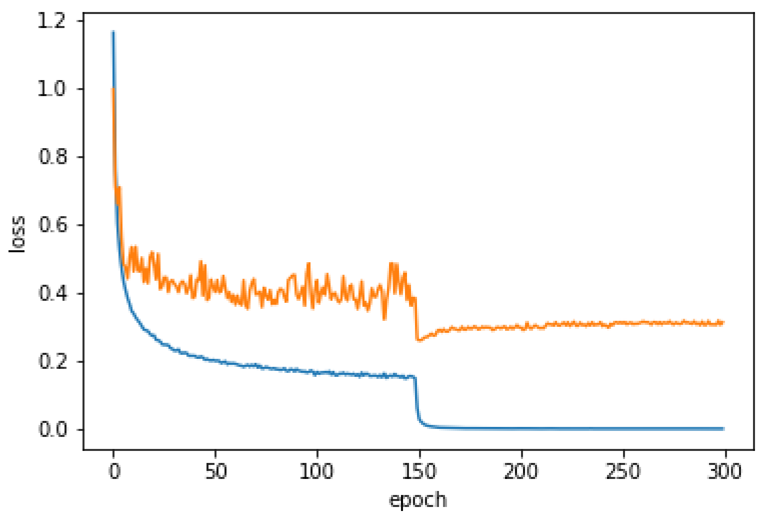
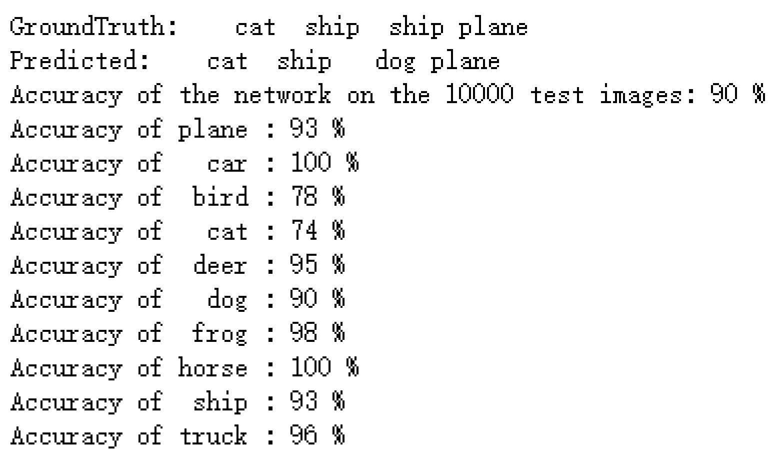


图 3: loss/epochs

分类结果如图6所示



```
GroundTruth:    cat  ship  ship plane
Predicted:    cat  ship   dog plane
Accuracy of the network on the 10000 test images: 90 %
Accuracy of plane : 93 %
Accuracy of   car : 100 %
Accuracy of  bird : 78 %
Accuracy of   cat : 74 %
Accuracy of  deer : 95 %
Accuracy of   dog : 90 %
Accuracy of  frog : 98 %
Accuracy of horse : 100 %
Accuracy of  ship : 93 %
Accuracy of truck : 96 %
```

图 4: 分类结果

四、 DenseNet 实现

DenseNet 模型，它的基本思路与 ResNet 一致，但是它建立的是前面所有层与后面层的密集连接（dense connection），它的名称也是由此而来。DenseNet 的另一大特色是通过特征在 channel 上的连接来实现特征重用（feature reuse）。这些特点让 DenseNet 在参数和计算成本更少的情形下实现比 ResNet 更优的性能，DenseNet 也因此斩获 CVPR 2017 的最佳论文奖。

1. DenseNet 网络结构

相比 ResNet，DenseNet 提出了一个更激进的密集连接机制：即互相连接所有的层，具体来说就是每个层都会接受其前面所有层作为其额外的输入。ResNet 网络的连接机制，作为对比，如图6所示为 DenseNet 的密集连接机制。可以看到，ResNet 是每个层与前面的某层（一般是 2 3 层）短路连接在一起，连接方式是通过元素级相加。而在 DenseNet 中，每个层都会与前面所有

层在 channel 维度上连接 (concat) 在一起 (这里各个层的特征图大小是相同的，后面会有说明)，并作为下一层的输入。对于一个 L 层的网络，DenseNet 共包含 $\frac{L(L+1)}{2}$ 个连接，相比 ResNet，这是一种密集连接。而且 DenseNet 是直接 concat 来自不同层的特征图，这可以实现特征重用，提升效率，这一特点是 DenseNet 与 ResNet 最主要的区别。

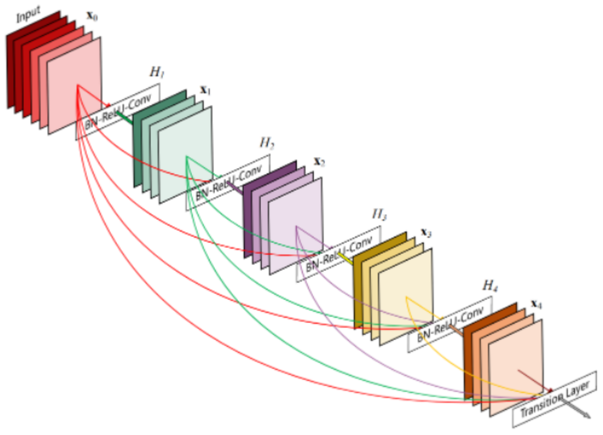


图 5: DenseNet 网络结构

CNN 网络一般要经过 Pooling 或者 stride>1 的 Conv 来降低特征图的大小，而 DenseNet 的密集连接方式需要特征图大小保持一致。为了解决这个问题，DenseNet 网络中使用 DenseBlock+Transition 的结构，其中 DenseBlock 是包含很多层的模块，每个层的特征图大小相同，层与层之间采用密集连接方式。而 Transition 模块是连接两个相邻的 DenseBlock，并且通过 Pooling 使特征图大小降低。

2. DenseNet **代码实现**

Transitionm 模块的实现如下：

```python
class Transition(nn.Module):
    """
    改变维数的Transition层
    先通过1x1的卷积层减少channels，再通过2x2的平均池化层缩小feature-map
    """
    def __init__(self, in_channels, out_channels):
        super(Transition, self).__init__()
        self.features = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(True),
            nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
            nn.AvgPool2d(2)
        )

    def forward(self, x):
        out = self.features(x)
        return out
```

DenseBlock 模块实现如下：

```python
class Bottleneck(nn.Module):
    """
    Dense  Block
    这里的growth_rate=out_channels，就是每个Block自己输出的通道数。
    先通过1x1卷积层，将通道数缩小为4 * growth_rate，然后再通过3x3卷积层降低到growth_rate。
    """
    expansion = 4

    def __init__(self, in_channels, growth_rate):
        super(Bottleneck, self).__init__()
        zip_channels = self.expansion * growth_rate
        self.features = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(True),
            nn.Conv2d(in_channels, zip_channels, kernel_size=1, bias=False),
            nn.BatchNorm2d(zip_channels),
            nn.ReLU(True),
            nn.Conv2d(zip_channels, growth_rate, kernel_size=3, padding=1, bias=False)
        )

    def forward(self, x):
        out = self.features(x)
        out = torch.cat([out, x], 1)
        return out
```

DenseNet 网络结构的实现如下：

```
class DenseNet(nn.Module):
    def __init__(self, num_blocks, growth_rate=12, reduction=0.5, num_classes=10):
        super(DenseNet, self).__init__()
        self.growth_rate = growth_rate
        self.reduction = reduction

        num_channels = 2 * growth_rate

        self.features = nn.Conv2d(3, num_channels, kernel_size=3, padding=1, bias=False)
        self.layer1, num_channels = self._make_dense_layer(num_channels, num_blocks[0])
        self.layer2, num_channels = self._make_dense_layer(num_channels, num_blocks[1])
        self.layer3, num_channels = self._make_dense_layer(num_channels, num_blocks[2])
        self.layer4, num_channels = self._make_dense_layer(num_channels, num_blocks[3], transition=False)
        self.avg_pool = nn.Sequential(
            nn.BatchNorm2d(num_channels),
            nn.ReLU(True),
            nn.AvgPool2d(4),
        )
        self.classifier = nn.Linear(num_channels, num_classes)

        self._initialize_weight()

    def _make_dense_layer(self, in_channels, nblock, transition=True):
        layers = []
        for i in range(nblock):
            layers += [Bottleneck(in_channels, self.growth_rate)]
            in_channels += self.growth_rate
        out_channels = in_channels
        if transition:
            out_channels = int(math.floor(in_channels * self.reduction))
            layers += [Transition(in_channels, out_channels)]
        return nn.Sequential(*layers), out_channels

    def _initialize_weight(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight.data)
                if m.bias is not None:
                    m.bias.data.zero_()

    def forward(self, x):
        out = self.features(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out
```

3. DenseNet 分类结果

loss/epochs 如图6所示

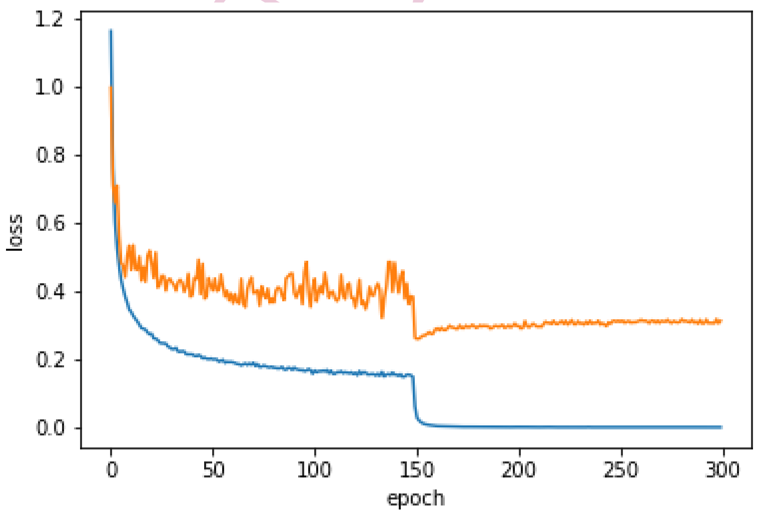
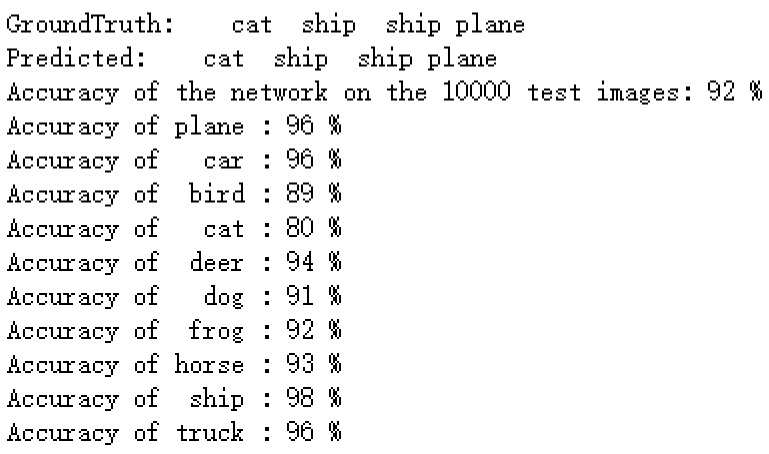


图 6: loss/epochs

分类结果如图7所示



```
GroundTruth:    cat  ship  ship plane
Predicted:    cat  ship  ship plane
Accuracy of the network on the 10000 test images: 92 %
Accuracy of plane : 96 %
Accuracy of   car : 96 %
Accuracy of  bird : 89 %
Accuracy of   cat : 80 %
Accuracy of  deer : 94 %
Accuracy of   dog : 91 %
Accuracy of  frog : 92 %
Accuracy of horse : 93 %
Accuracy of  ship : 98 %
Accuracy of truck : 96 %
```

图 7: 分类结果

五、 SE-ResNet 实现

1. SE-ResNet 网络结构

这里我们沿用 ResNet18 的残差网络结构，将基本块网络中加入 SE 单元，SE 单元包含一个平均池化层、两个全连接层、和一个相乘单元，将该 SE 单元和残差网络大致结构相结合，即加入两个 3x3 卷积层，网络结构为：

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): SEBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=64, out_features=4, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=4, out_features=64, bias=False)
          (3): Sigmoid()
        )
      )
    )
    (1): SEBasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=64, out_features=4, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=4, out_features=64, bias=False)
          (3): Sigmoid()
        )
      )
    )
  )
  (layer2): Sequential(
    (0): SEBasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=128, out_features=8, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=8, out_features=128, bias=False)
          (3): Sigmoid()
        )
      )
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): SEBasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=128, out_features=8, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=8, out_features=128, bias=False)
          (3): Sigmoid()
        )
      )
    )
  )
  (layer3): Sequential(
    (0): SEBasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=256, out_features=16, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=16, out_features=256, bias=False)
          (3): Sigmoid()
        )
      )
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): SEBasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=256, out_features=16, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=16, out_features=256, bias=False)
          (3): Sigmoid()
        )
      )
    )
  )
  (layer4): Sequential(
    (0): SEBasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
                                                    bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
                                                    track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
                                                    bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
                                                    track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=512, out_features=32, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=32, out_features=512, bias=False)
          (3): Sigmoid()
        )
      )
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
                                                      track_running_stats=True)
      )
    )
    (1): SEBasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
                                                    bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
                                                    track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
                                                    bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
                                                    track_running_stats=True)
      (se): SELayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (fc): Sequential(
          (0): Linear(in_features=512, out_features=32, bias=False)
          (1): ReLU(inplace=True)
          (2): Linear(in_features=32, out_features=512, bias=False)
          (3): Sigmoid()
        )
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=1)
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

图状形式呈现为如下图：

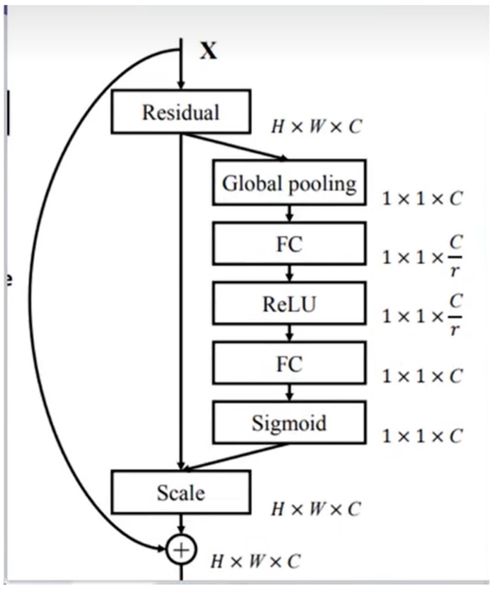


图 8: SE-resnet 结构图

值得注意的是，在平均池化后两个全连接层是必要的，其中第一个全连接层为了减少运算，第二个全连接层则是为了把通道数还原以便于和原数据分通道相乘。

2. SE-ResNet **代码**

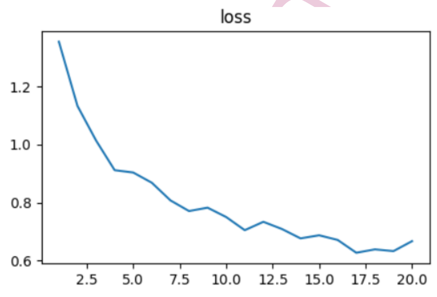
如以下代码所示，我们只需要重新定义经典 Resnet 残差块（3x3 卷积层）和 SE 模块组合成一个新的模块，将其嵌入到 ResNet18 网络结构中，就能得到带 SE 模块的 ResNet 网络模型：

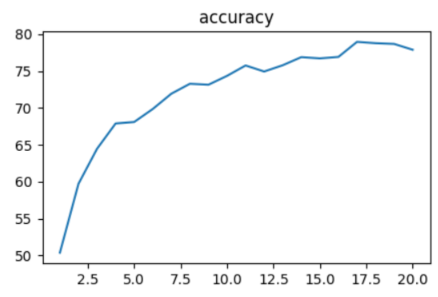
```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )
    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
    ......
    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

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

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

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

        return out
def se_resnet18(num_classes=10):
    model = ResNet(SEBasicBlock, [2, 2, 2, 2], num_classes=num_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
    return model
```

注意到在 Basicblock 里有一个 residual 模块，这里便是将残差模块的"跳绳连接"体现出来。

3. SE-ResNet 训练结果

该网络在 Cifar10 验证集上的训练 loss 曲线、准确度曲线图如下所示：





六、 对模型训练结果的解释

深度神经网络的美妙之处在于它们可以比浅层神经网络更有效地学习复杂的功能。在训练深度神经网络时，模型的性能随着架构深度的增加而下降。这被称为退化问题。但是，随着网络深度的增加，模型的性能下降的原因可能是什么？让我们尝试了解退化问题的原因。可能的原因之一是过度拟合。随着深度的增加，模型往往会过度拟合，但这里的情况并非如此。

ResNet 的作者（He 等人）认为，使用批量归一化和通过归一化正确初始化权重可确保梯度具有合适的标准。实验证明，与浅层网络相比，深层网络会产生较高的训练误差。这表明更深层无法学习甚至恒等映射。

主要原因之一是权重的随机初始化，均值在零、L1 和 L2 正则化附近。结果，模型中的权重总是在零左右，因此更深的层也无法学习恒等映射。这里出现了跳跃连接的概念，它使我们能够

训练非常深的神经网络。

跳跃连接，会跳跃神经网络中的某些层，并将一层的输出作为下一层的输入。引入跳跃连接是为了解决不同架构中的不同问题。在 ResNets 的情况下，跳跃连接解决了我们之前解决的退化问题，而在 DenseNets 的情况下，它确保了特征的可重用性，在 SE-ResNet 网络中跳跃连接更是丰富了信息的深度，从而使模型得到更多的信息。

因此，训练过程产生的不同主要由是否引入残差模块而决定的，我们将 ResNet、DenseNet 和 SE-ResNet 网络均看作引入残差模块的模型，而平凡 CNN 网络看作没有引入残差模块的模型，我们可以发现引入残差模块的模型的准确率要远远高于不引人残差模块的模型，并且发现平凡 CNN 在训练过程中容易出现过拟合现象，这样的原因能够被我们上述的理念所解释：残差模块中的跳跃连接能有效地避免神经网络训练过程中的“退化现象”。

接下来对比三种带有残差模块的模型训练过程中的不同：resnet18 和 densenet 模型准确度的区别不大，但是 densenet 训练速度非常慢，而且并不比 resnet18 有较为明显的准确度的提升，这里的原因同样很简单：在于 DenseNet 借鉴了 ResNet 残差模块的思想，两者均使用“过去的”节点参与训练，但是在此基础上 DenseNet 网络频繁利用先前的节点形成所谓的密集连接，是一个非常耗时的训练过程，同时 DenseBlock 在按通道拼接特征的时候，需要耗费大量的显存，达到平方级别，需要设计一块缓存供拼接层共享使用，但值得一提的是它的参数量要比 resnet 少很多。相反，SE-ResNet 为我们展现出了不俗的表现力，它能够在较少 epochs 的情况下就达到很高的正确率，这是因为该模型是在 resnet18 基础上添加 SE 模块改进而来，SE 模块能够通过每层信息不同的权重比而着重提取哪些特征，使得图片的信息更加丰富，是在计算机视觉大赛中图像分类方面表现不俗的种子选手。