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：王泳鑫

学号：1911479

年级：2019级

专业：计算机科学与技术

指导教师：侯淇彬

2022 年 5 月 20 日

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

二、 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7所示

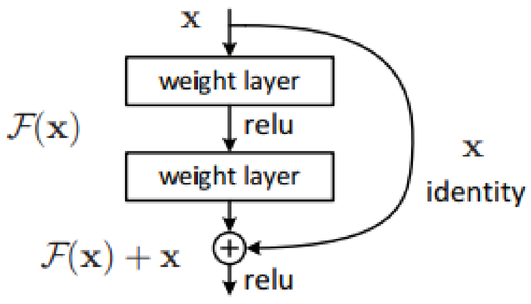

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

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7所示

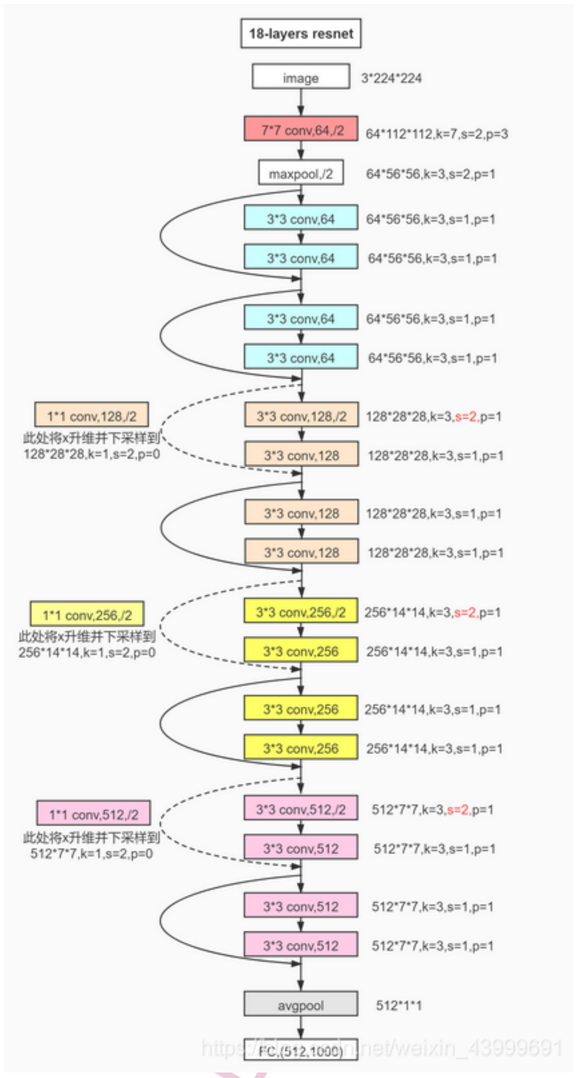


图 2: Caption

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，则使用卷积层来改变维度1x1
        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：

```
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)
        # 经过上述结构后，到这里的cifar10feature map 是size 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):
        # 第一个要进行降采样block
        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7所示

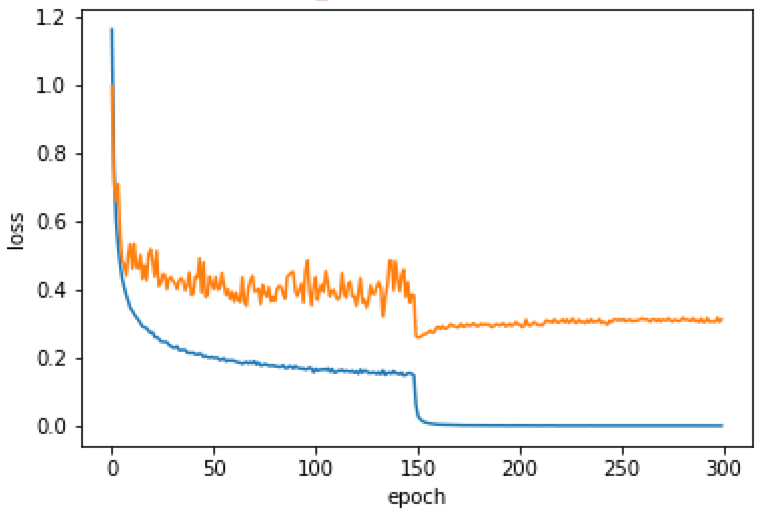
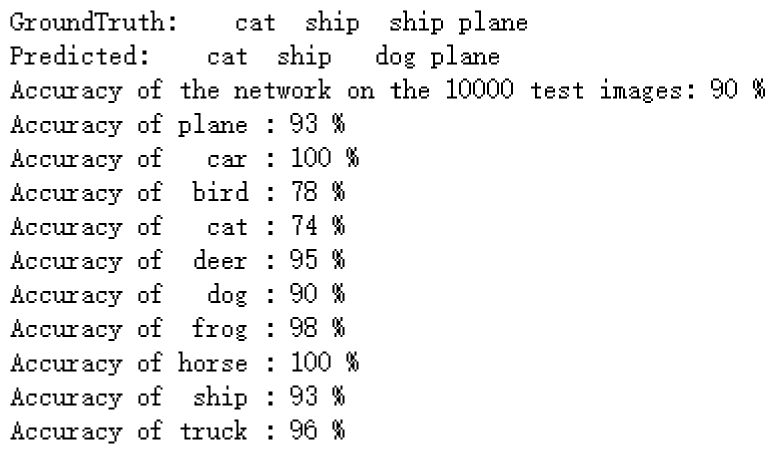


图 3: loss/epochs

分类结果如图7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

三、 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7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。

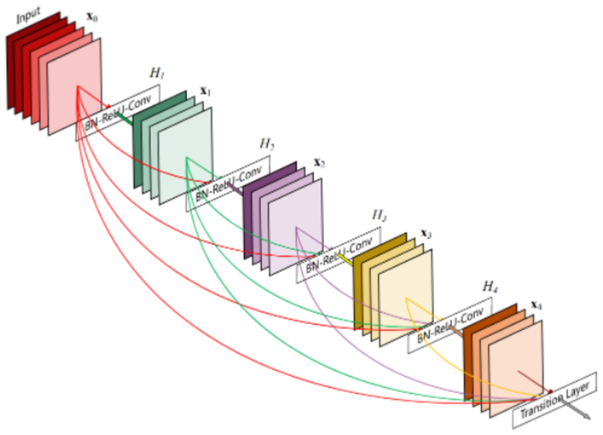


图 5: DenseNet网络结构

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：

```
class Transition(nn.Module):
    """改变维数的层
    Transition先通过的卷积层减少，再通过的平均池化层缩小
    1x1channels2x2feature-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：

```
class Bottleneck(nn.Module):
    """
    Dense Block这里的
    growth_rate=out_channels, 就是每个自己输出的通道数。Block先通过卷积层，将通道数缩小为
    1x14 * ，然后再通过卷积层降低到。growth_rate3x3growth_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7所示

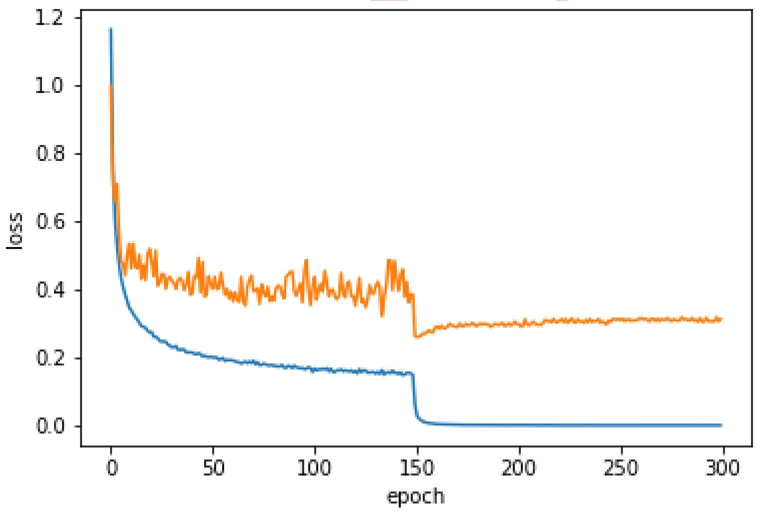
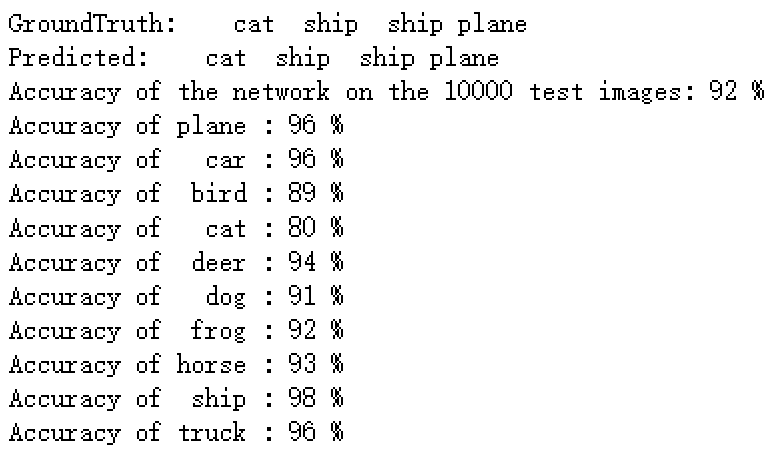


图 6: loss/epochs

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

四、 解答

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。