中山大学计算机学院

人工智能

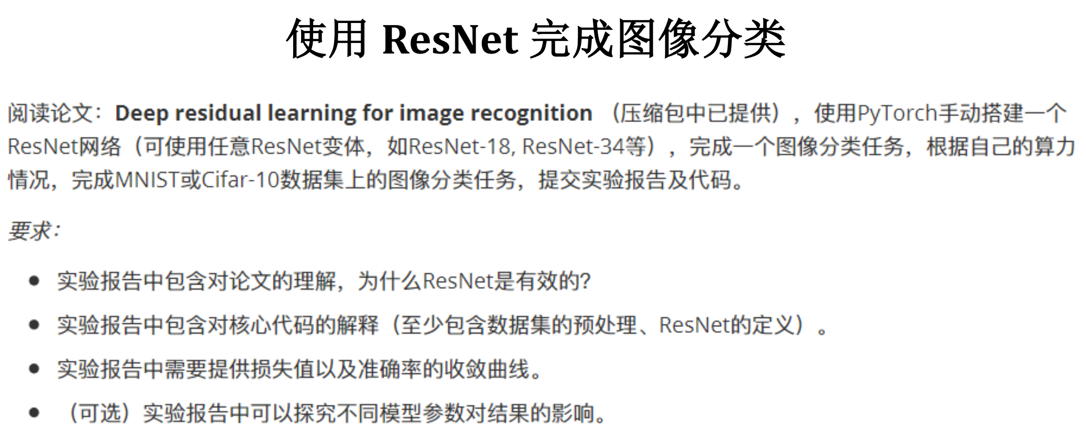
本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	**202320346**	专业（方向）	**计算机科学与技术**
学号	**21312450**	姓名	**林隽哲**

一、 实验题目



使用 **ResNet** 完成图像分类

阅读论文：**Deep residual learning for image recognition** （压缩包中已提供），使用PyTorch手动搭建一个ResNet网络（可使用任意ResNet变体，如ResNet-18, ResNet-34等），完成一个图像分类任务，根据自己的算力情况，完成MNIST或Cifar-10数据集上的图像分类任务，提交实验报告及代码。

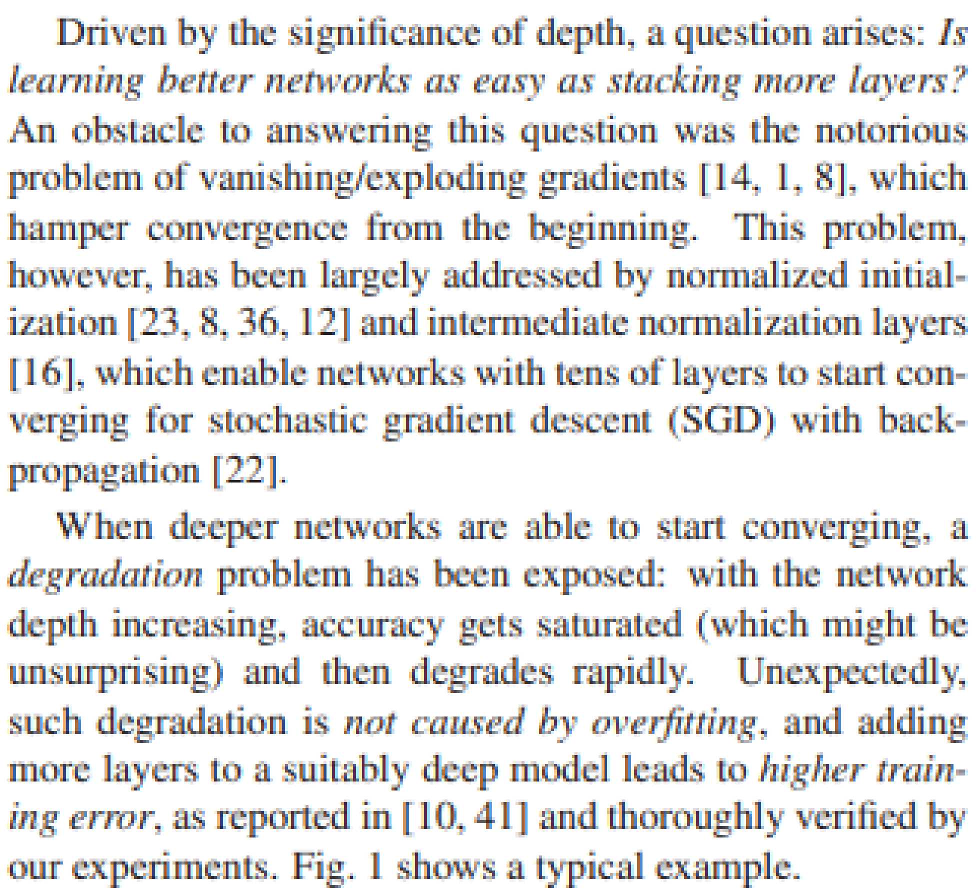
要求：

- 实验报告中包含对论文的理解，为什么ResNet是有效的?
- 实验报告中包含对核心代码的解释（至少包含数据集的预处理、ResNet的定义）。
- 实验报告中需要提供损失值以及准确率的收敛曲线。
- （可选）实验报告中可以探究不同模型参数对结果的影响。

二、 实验内容

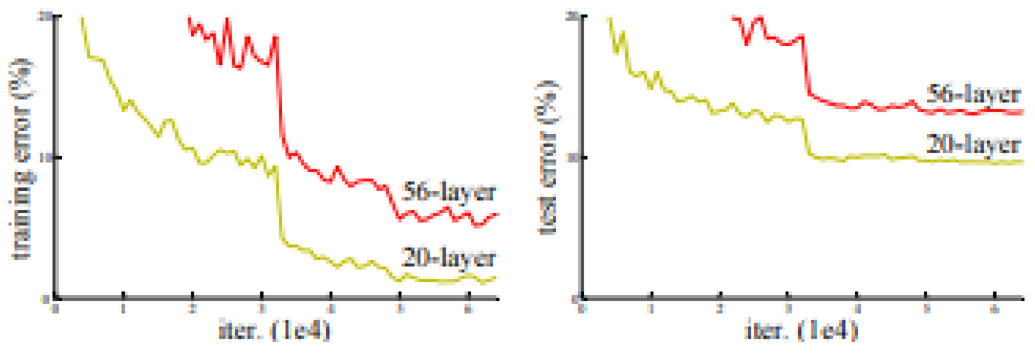
1. 算法原理

首先我们先要明确 ResNet 网络的提出是为了解决什么样的问题：



Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [14, 1, 8], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 8, 36, 12] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with back-propagation [22].

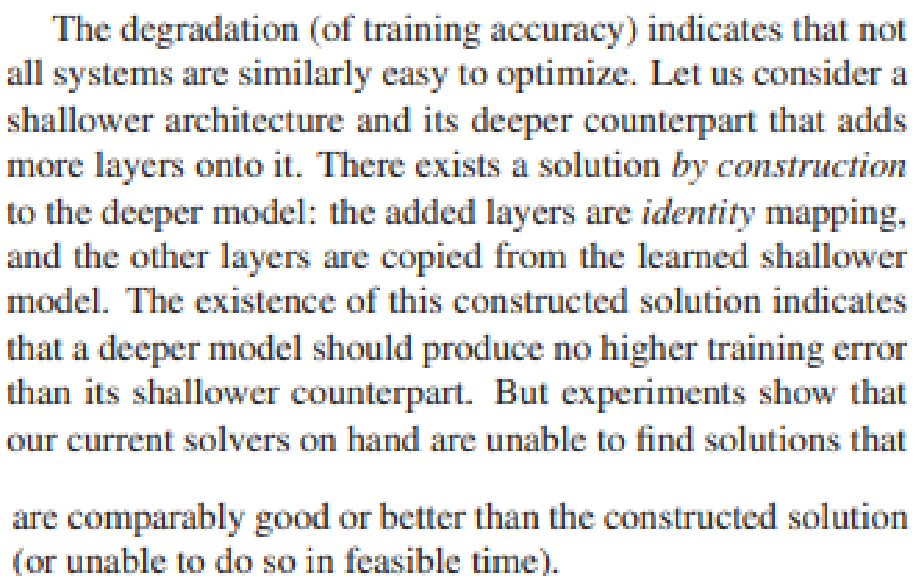
When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is *not caused by overfitting*, and adding more layers to a suitably deep model leads to *higher training error*, as reported in [10, 41] and thoroughly verified by our experiments. Fig. 1 shows a typical example.



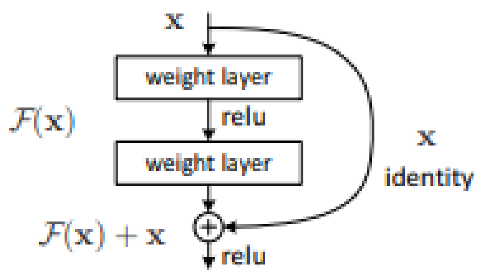
通过实验，ResNet 团队通过实验发现一般的学习网络在随着网络层的不断增加，模型的准确率显示不断地提高，最终会达到一个最大值（准确率饱和），然后随着网络深度的继续增加，模型准确率会毫无征兆的出现大幅度的降低。

这个现象违背了人们最初“越深的网络准确率越高”的直觉。ResNet 团队把这一现象称为“退化（Degradation）”。

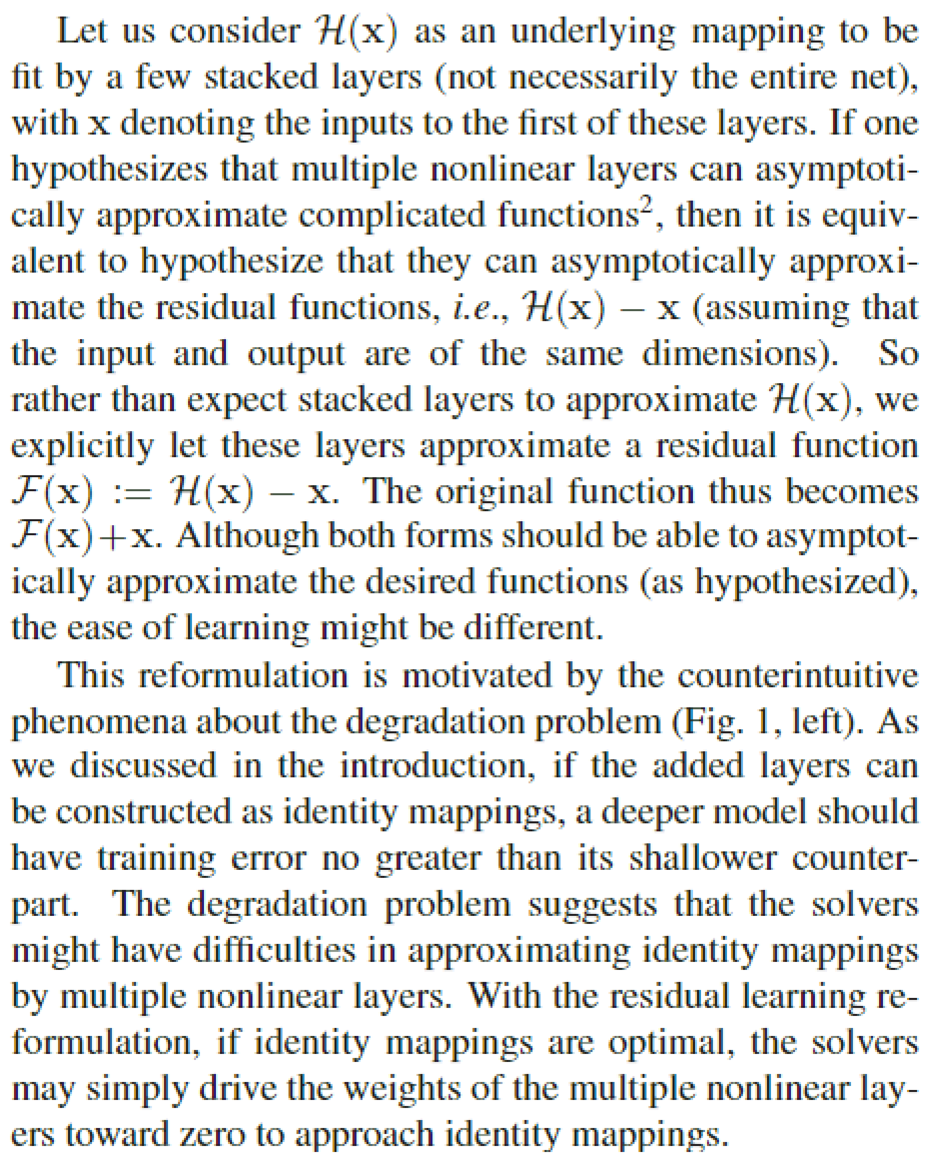
接下来，我们来看一下 ResNet 网络是怎么解决“退化”问题的：


The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution *by construction* to the deeper model: the added layers are *identity* mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. But experiments show that our current solvers on hand are unable to find solutions that are comparably good or better than the constructed solution (or unable to do so in feasible time).

为了解决退化问题，ResNet 团队尝试在网络中引入恒等映射，使得深层模型的一部分来自于浅层模型中的复制，该构造将保证较深的模型不会产生比浅层模型更高的训练误差。为了实现这种构造，ResNet 团队选择通过引入具有残差连接（Shortcut Connection，或者你也可以直接将其翻译为快捷连接）分支的前馈神经网络，如下图所示：


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
Figure 2. Residual learning: a building block.

ResNet 通过引入残差连接，允许某一层的输出直接跳过一个或者多个层，连接到后续层的输入，也就是恒等映射。这样做的好处是，即使某些层不做任何有意义的变换（或者说没有从数据中学习到有用的信息），它们仍可以传递之前层的信息，而不会对梯度产生过多的损失。



Let us consider $\mathcal{H}(\mathrm{x})$ as an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), with x denoting the inputs to the first of these layers. If one hypothesizes that multiple nonlinear layers can asymptotically approximate complicated functions[2], then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, *i.e.*, $\mathcal{H}(\mathrm{x}) - \mathrm{x}$ (assuming that the input and output are of the same dimensions). So rather than expect stacked layers to approximate $\mathcal{H}(\mathrm{x})$, we explicitly let these layers approximate a residual function $\mathcal{F}(\mathrm{x}) := \mathcal{H}(\mathrm{x}) - \mathrm{x}$. The original function thus becomes $\mathcal{F}(\mathrm{x})+\mathrm{x}$. Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), the ease of learning might be different.

This reformulation is motivated by the counterintuitive phenomena about the degradation problem (Fig. 1, left). As we discussed in the introduction, if the added layers can be constructed as identity mappings, a deeper model should have training error no greater than its shallower counterpart. The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers. With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings.

让我们考虑 H(x)作为一个底层映射，它适合于几个对叠层（不一定是整个网络），其中 x 表示这些层中第一个层的输入。如果假设多个非线性层可以渐进逼进复杂函数，则等价于假设它们可以渐进逼近残差函数 F(x)，即$H(x) - x$（假设输入和输出具有相同的量纲）。而不是期望对叠层逼近 H(x)，我们明确地让这些层逼近一个残差函数 $F(x) \coloneqq H(x) - x$。因此，我们将网络转换为对残差函数 F(x)的学习。只要$F(x) = 0$，就构成了一个恒等映射 H(x) = x 。

另外需要提到的是，如果残差映射 F(x)的结果与残差连接 x 的维度不同，那么是无法直接将它们进行相加操作的，必须对 x 进行升维操作，是二者的维度相匹配。在论文中提到了两种 x 的升维方式：

- ***全 0 填充***
- ***采用 1*1 卷积***

在本次实验中，我采用的是第二种方法。

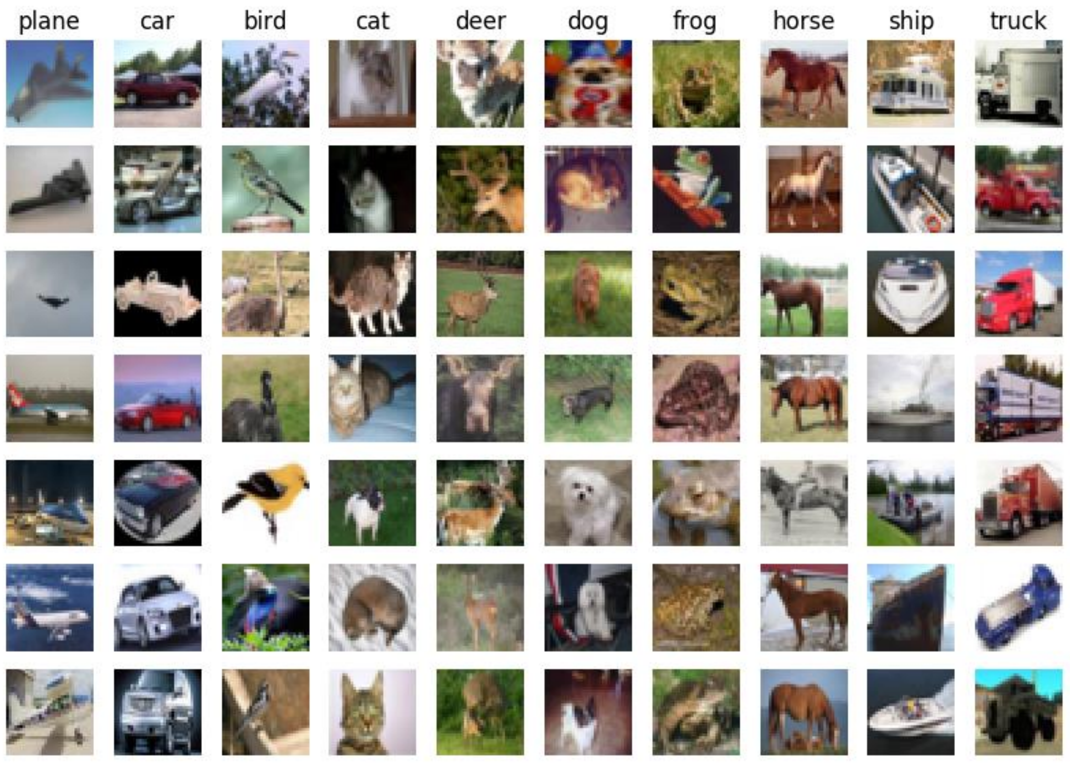
2. 关键代码展示

数据集的预处理

```
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader

# Data preprocessing
transform_train = transforms.Compose([
    # Padding the image by 4 pixels on each side,
    # then take a random crop of size 32x32 pixels.
    transforms.RandomCrop(32, padding=4),
    # Randomly flip the image horizontally with a probability of 0.5
    transforms.RandomHorizontalFlip(),
    # Convert the image to a PyTorch tensor
    transforms.ToTensor(),
    # Normalize the image
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

transform_test = transforms.Compose([
    # Convert the image to a PyTorch tensor
    transforms.ToTensor(),
    # Normalize the image
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

# Load datasets, I use CIFAR10 here
trainset = CIFAR10(root='./data', train=True, download=True, transform=transform_train)
testset = CIFAR10(root='./data', train=False, download=True, transform=transform_test)

trainloader = DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)
testloader = DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)
```

上述代码涉及到训练数据的下载以及预处理。

在数据选取与下载部分，我这次使用的训练数据为 CIFAR-10 训练集。CIFAR-10 是一个更接近普适物体的彩色图像数据集。其中包含 10 个类别的 RGB 彩色图片：airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck。每个类别有 6000 个图像，其中包含 5000 张训练图像以及 1000 张测试图像。数据集中的图片尺寸均为 32*32。



在数据预处理部分，我定义了两个 Compose：transform_train、transform_test，并将它们分别导入到两个 DataLoader：trainloader、testloader 中。在后续的数据使用中，我将会通过两个 DataLoader 进行数据的读取，数据首先会经过 Compose 处理后再供训练与测试使用。对于训练数据，我选择使用随机裁剪以及随机翻转的方式进行数据增强。训练数据与测试数据都将会被转换为 PyTorch 的 tensor 数据格式，并进行一次正则化。

ResNet-18 的定义

```python
import torch.nn as nn

import torch.nn.functional as F


class BasicBlock(nn.Module):

    expansion = 1


    def __init__(self, in_planes, planes, stride=1):

        # Convolutional and BatchNorm layers defined here

        super(BasicBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)

        self.bn1 = nn.BatchNorm2d(planes)

        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)

        self.bn2 = nn.BatchNorm2d(planes)


        self.shortcut = nn.Sequential()

        # If the input number of channels is different from the output,

        # then the shortcut needs to adjust the dimensions
        if stride != 1 or in_planes != self.expansion*planes:

            self.shortcut = nn.Sequential(

                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=1, stride=stride, bias=False),

                nn.BatchNorm2d(self.expansion*planes)

            )


    def forward(self, x):

        # Forward pass logic including shortcut connections

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

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

        out += self.shortcut(x)

        out = F.relu(out)

        return out


class ResNet(nn.Module): # ResNet-18

    def __init__(self, block, num_blocks, num_classes=10):

        # Layers are set up here including intial conv

        super(ResNet, self).__init__()

        self.in_planes = 64

        # Initial the first convolutional layer,

        # 3 input channels for RGB, 64 output channels, kernel size 7x7, stride 2, padding 3

        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)

        # BatchNorm and ReLU layers are added

        self.bn1 = nn.BatchNorm2d(64)

        self.relu1 = nn.ReLU(inplace=True)

        # Maxpool layer is added

        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # 4 blocks are created using the _make_layer helper function

        self.layers = nn.Sequential(

            self._make_layer(block, 64, num_blocks[0], stride=1),

            self._make_layer(block, 128, num_blocks[1], stride=2),

            self._make_layer(block, 256, num_blocks[2], stride=2),

            self._make_layer(block, 512, num_blocks[3], stride=2)

        )

        # AdaptiveAvgPool layer is added

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        # Final linear layer is added

        self.linear = nn.Linear(512*block.expansion, num_classes)


    def _make_layer(self, block, planes, num_blocks, stride):

        # Helper function to create a block of layers

        strides = [stride] + [1]*(num_blocks-1)

        layers = []

        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu1(out)
        out = self.maxpool(out)
        out = self.layers(out)
        out = self.avgpool(out)
        # Reshape the output to fit the linear layer input
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

# ResNet18: 4 layers of BasicBlock, 2 blocks per layer, 2 conv layers per block
net = ResNet(BasicBlock, [2, 2, 2, 2])
```

上述代码涉及到 ResNet-18 模型的定义。

BasicBlock 类中定义了一个 ResNet 中的一个基础的残差块，形式如下：

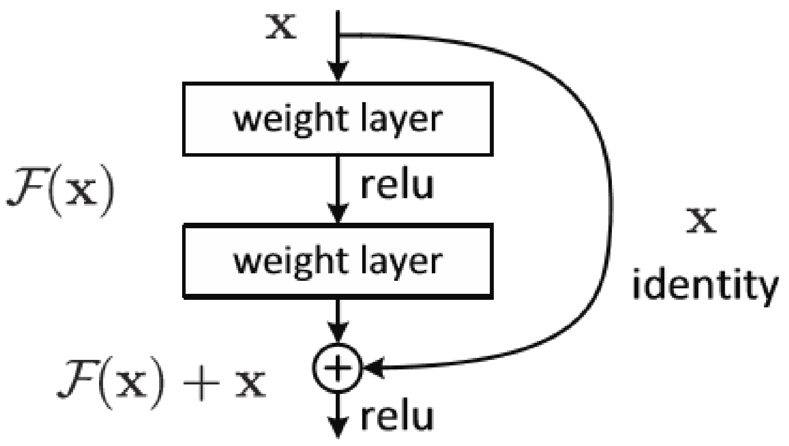

x
weight layer
relu
weight layer
relu
$\mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
x
identity

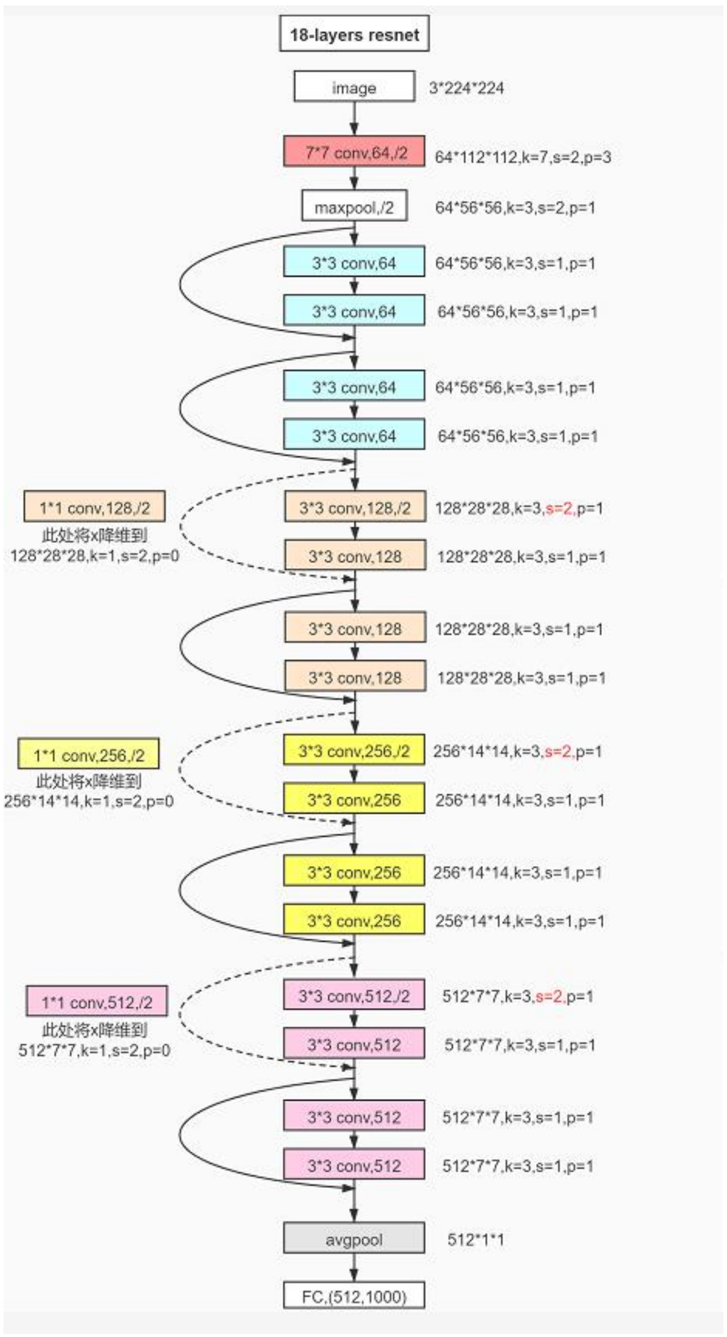
Figure 2. Residual learning: a building block.

每一个基础块（Basic Block）中包含了两个卷积层（Convolutional Layer），并有一个残差连接（Shortcut Connections）将块的输入连接到一个累加运算中，使之与第二个卷积层的输出结果进行相加。注意，若输入的数据在通过卷积运算后改变了维度，则需要将残差连接中也加入一个核大小为 1 的卷积层，使输入经过残差连接后的输出的维度能够与第二个卷积层输出的维度匹配。另外需要提到的是，每个卷积层之后都会跟有一个 BatchNorm2d，用于对卷积后的数据

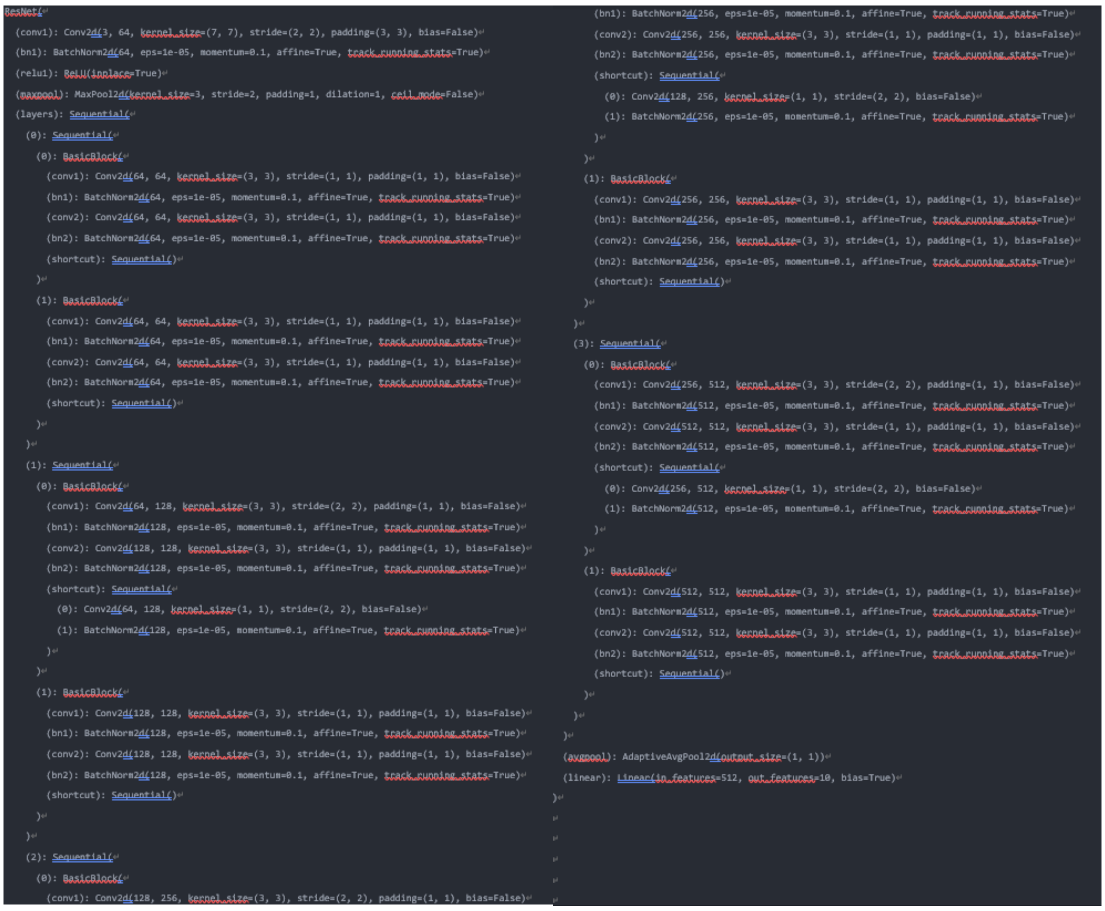
进行归一处理，这使得数据在进行 ReLU 之前不会因为数据过大而导致网络性能的不稳定。BatchNorm2d 的数学原理如下：

$$y = \frac{x - mean(x)}{\sqrt{Var(x)} + eps} * gamma + beta$$

ResNet 类中通过运用 BasicBlock，构建出一个 ResNet-18 模型（其中 18 指 17 个卷积层以及一个线性层（Linear Layer）。当然，模型中还包含了最大池化层（Max Pooling Layer）、平均池化层（Average Polling Layer）等）。下图是 ResNet-18 的一个简要的结构示意图：



对应的实现较为简单，具体实现步骤可以参考代码中的注释。构造出的模型结构如下：



```
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)
  (relu1): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layers): Sequential(
    (0): Sequential(
      (0): BasicBlock(
        (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)
        (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)
        (shortcut): Sequential()
      )
      (1): BasicBlock(
        (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)
        (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)
        (shortcut): Sequential()
      )
    )
    (1): Sequential(
      (0): BasicBlock(
        (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)
        (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)
        (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(
        (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)
        (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)
        (shortcut): Sequential()
      )
    )
    (2): Sequential(
      (0): BasicBlock(
        (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)
        (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)
        (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(
        (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)
        (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)
        (shortcut): Sequential()
      )
    )
    (3): Sequential(
      (0): BasicBlock(
        (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)
        (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)
        (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(
        (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)
        (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)
        (shortcut): Sequential()
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (linear): Linear(in_features=512, out_features=10, bias=True)
)
```

三、 实验结果及分析

选取损失函数与优化函数如下：

```python
# CrossEntropyLoss and SGD optimizer are used
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
```

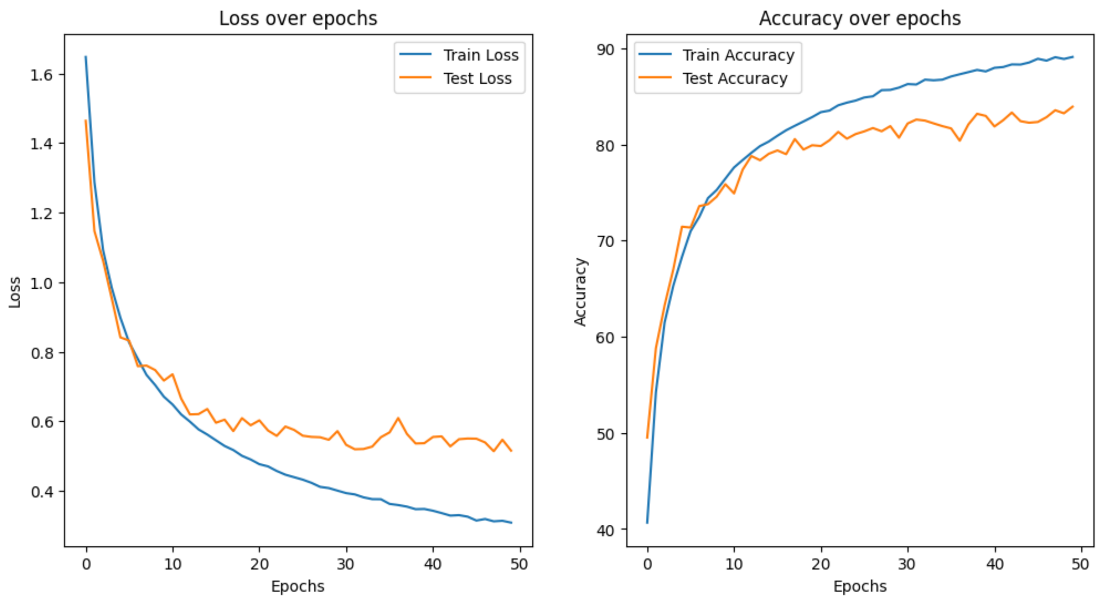
在这里我选择了交叉熵损失函数（Cros Entropy Loss Function）作为损失函数，并使用随机梯度下降（SGD）作为优化函数。（实际上使用 Adam 作为优化函数也是一个不错的选择，但在这里为了方便进行调参，我选择了相对简便的 SGD。在实际的测试中也得出在该实验数据下使用 SGD 与使用 Adam 所得到的最终结果差距并不大。）

接下来进行训练，并将训练过程中得到的损失值与准确率变化曲线绘制如下：

损失值与准确率

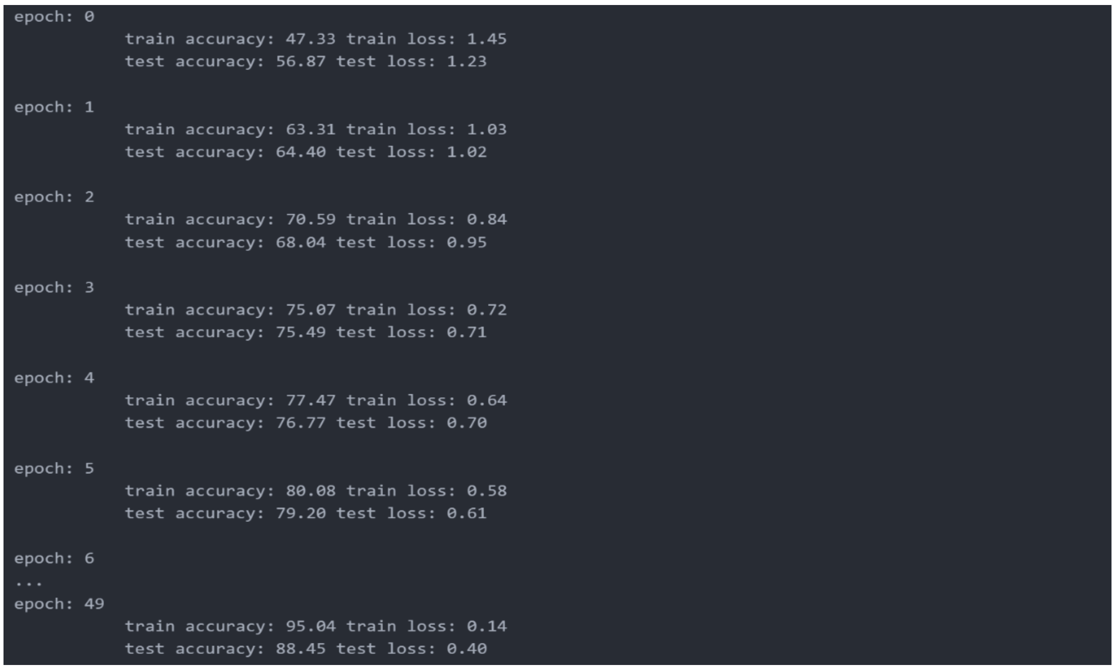
```
epoch: 0
train accuracy: 40.66 train loss: 1.65
test accuracy: 49.51 test loss: 1.46

epoch: 1
train accuracy: 54.21 train loss: 1.29
test accuracy: 58.80 test loss: 1.15

epoch: 2
train accuracy: 61.46 train loss: 1.09
test accuracy: 63.24 test loss: 1.06

epoch: 3
train accuracy: 65.30 train loss: 0.98
test accuracy: 66.99 test loss: 0.95

epoch: 4
train accuracy: 68.29 train loss: 0.90
test accuracy: 71.44 test loss: 0.84

epoch: 5
train accuracy: 70.98 train loss: 0.83
test accuracy: 71.35 test loss: 0.83

epoch: 6
...
epoch: 49
train accuracy: 89.09 train loss: 0.31
test accuracy: 83.92 test loss: 0.52
```

损失值与准确率的收敛曲线



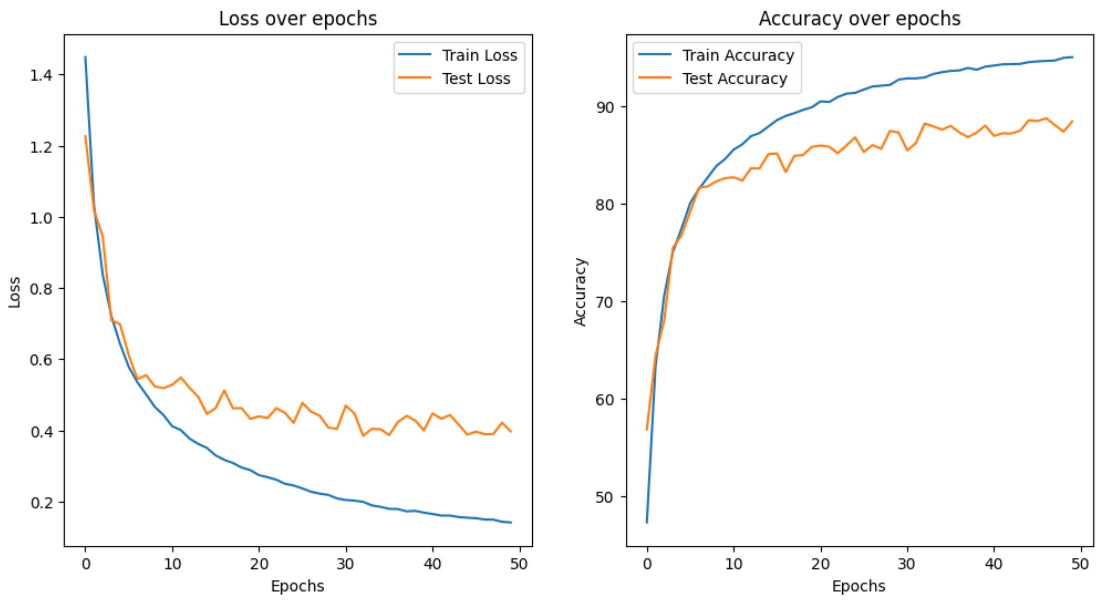
然而，对于一个图像大小均只有 32*32 的训练集，在第一个卷积层中使用一个核大小为 7、步长为 2 的卷积核有点不太合适。因此我尝试将 ResNet 中的第一个卷积层的参数改为 kernel_size 为 2，stride 为 1，padding 为 1，重新对数据集进行训练得到结果如下：

损失值与准确率



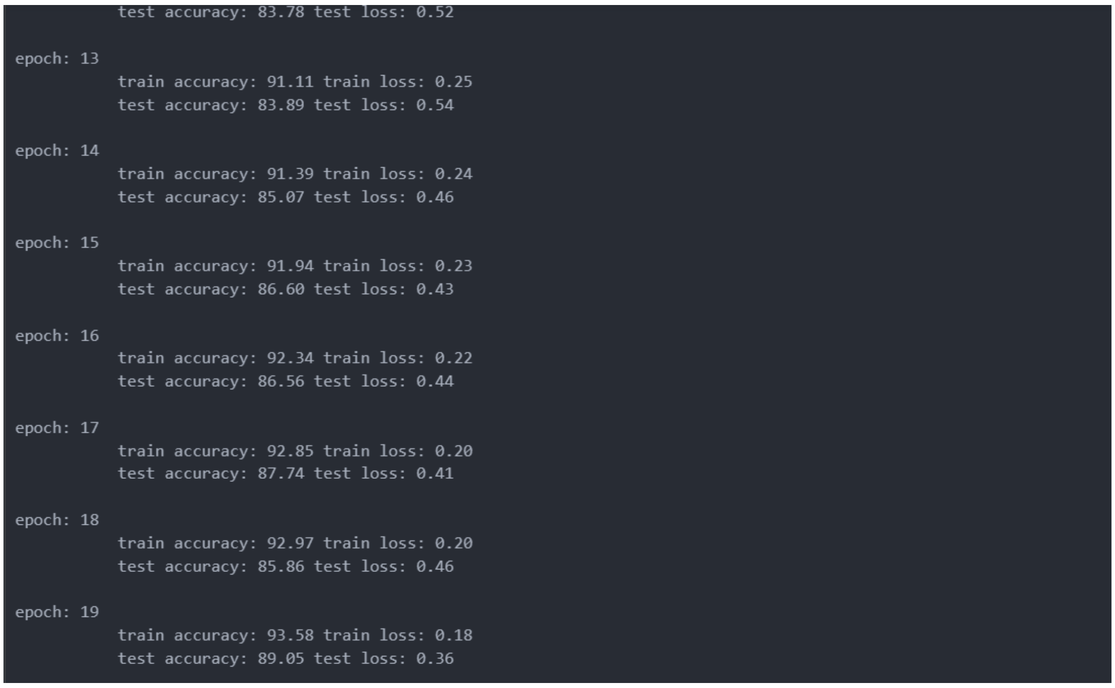
```
epoch: 0
        train accuracy: 47.33 train loss: 1.45
        test accuracy: 56.87 test loss: 1.23

epoch: 1
        train accuracy: 63.31 train loss: 1.03
        test accuracy: 64.40 test loss: 1.02

epoch: 2
        train accuracy: 70.59 train loss: 0.84
        test accuracy: 68.04 test loss: 0.95

epoch: 3
        train accuracy: 75.07 train loss: 0.72
        test accuracy: 75.49 test loss: 0.71

epoch: 4
        train accuracy: 77.47 train loss: 0.64
        test accuracy: 76.77 test loss: 0.70

epoch: 5
        train accuracy: 80.08 train loss: 0.58
        test accuracy: 79.20 test loss: 0.61

epoch: 6
...
epoch: 49
        train accuracy: 95.04 train loss: 0.14
        test accuracy: 88.45 test loss: 0.40
```

损失值与准确率的收敛曲线



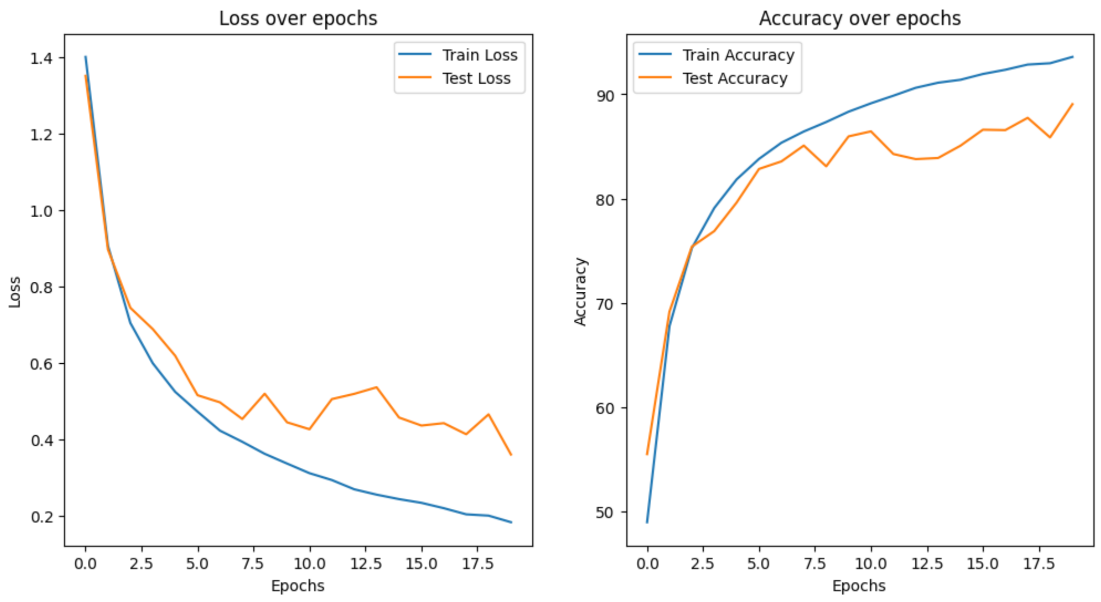
从上面结果可以看出，相比其原来的结构，修改后的网络在准确率上有所提升，并且最终的分类效果还是不错的。接下来，我们尝试去掉网络中最初的最大池化层，重新对训练集进行训练得到结果如下：

损失值与准确率



```
        test accuracy: 83.78 test loss: 0.52

epoch: 13
        train accuracy: 91.11 train loss: 0.25
        test accuracy: 83.89 test loss: 0.54

epoch: 14
        train accuracy: 91.39 train loss: 0.24
        test accuracy: 85.07 test loss: 0.46

epoch: 15
        train accuracy: 91.94 train loss: 0.23
        test accuracy: 86.60 test loss: 0.43

epoch: 16
        train accuracy: 92.34 train loss: 0.22
        test accuracy: 86.56 test loss: 0.44

epoch: 17
        train accuracy: 92.85 train loss: 0.20
        test accuracy: 87.74 test loss: 0.41

epoch: 18
        train accuracy: 92.97 train loss: 0.20
        test accuracy: 85.86 test loss: 0.46

epoch: 19
        train accuracy: 93.58 train loss: 0.18
        test accuracy: 89.05 test loss: 0.36
```

损失值与准确率的收敛曲线



可以看出虽然最终训练得到的准确率相对更高，但是其损失曲线与准确率曲线的稳定性都有所下降。

四、 参考资料

• 《Deep residual learning for image recognition》