# 机器视觉实验报告（六）

**目录**

[机器视觉实验报告（六）](#_bookmark0) [1](#_bookmark0)

[一． 实验目的](#_bookmark1) [1](#_bookmark1)

[二． 实验原理](#_bookmark2) [1](#_bookmark2)

[三． 实验步骤](#_bookmark3) [4](#_bookmark3)

[四． 数据集](#_bookmark4) [4](#_bookmark4)

[五． 程序代码](#_bookmark5) [4](#_bookmark5)

[六． 实验结果](#_bookmark6) [11](#_bookmark6)

[七． 实验分析与总结](#_bookmark7) [13](#_bookmark7)

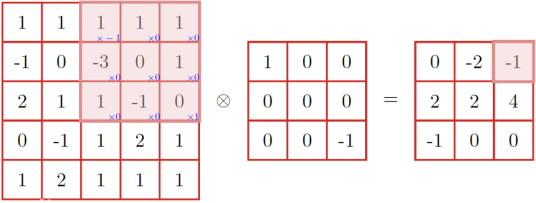
## 一．实验目的

* 熟悉卷积神经网络的基本结构，包括卷积层，池化层，激活函数及最后的全连接层等。
* 学习经典手写数字识别网络 LeNet 及其他经典特征提取网络。
* 熟悉 MindSpore 深度学习框架。
* 在 CIFAR-10 数据集上完成图像分类任务，使训练集或测试集的准确率达到 98%以上。

## 二．实验原理

主要包括卷积层，池化层，全连接层，激活函数，损失函数和优化器 6 个部分。

### 卷积层

卷积核作为窗口在特征图上滑动进行卷积操作。卷积具有平移不变性，能保留输入图像的空间特点，同时相比于全连接层大大减少了参数量，将一个卷积核的参数作用与整个图像。卷积计算示意图如下：

### 池化层

池化层是使用某一位置的相邻输出的总体统计特征来代替网络在该位置的输出。常见的池化方式有最大池化（Max Pooling），平均池化（Average Pooling）等。

当输入图像尺寸为𝑊1 × 𝐻1 × 𝐷1，池化核尺寸为𝐹 × 𝐹，池化核移动步长为 S 时，输出图的尺寸计算公式如下（W,H,D 分别是图像宽度高度和通道数）

𝑊1 − 𝐹

𝑊2 =

𝐻2 =

+ 1

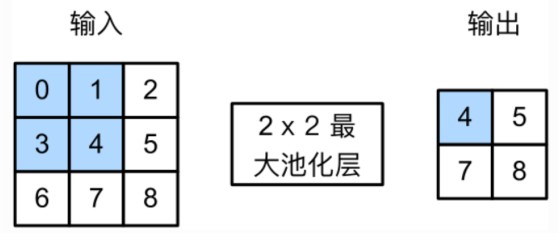
𝑆

𝐻1 − 𝐹

+ 1

𝑆

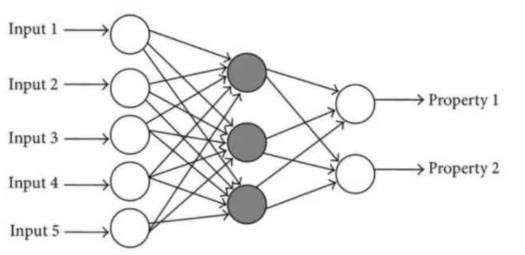
𝐷2 = 𝐷1

示意图如下

在卷积神经网络中，池化层的作用有：

1. 能够使网络自动学得不变性，包括旋转不变性和平移不变性。
2. 能够减小输入规模即参数量，提高统计效率，减少参数的存储需求。

### 全连接层

全连接层相当于矩阵乘法，对输入进行线性变换，从而把有用的信息进行整合。全连接层中间可以添加激活函数（在 2.4 介绍）进行非线性映射进而模拟非线性变换。示意图如下

卷积神经网络中全连接层的作用主要是作为分类器，将学到的特征映射到对应的样本类别。全连接层在实际操作过程中可以使用 1×1 的卷积核实现。使用 nn.linear 时，由于输入输出神经元之间的数量关系确定，所以输入图片必须采用指定的大小，实际应用中会有一定的限制且参数量较大。

### 激活函数

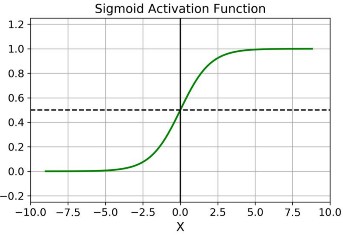
在卷积神经网络中，卷积层，全连接层进行的都是线性变换，必须加入非线性的激活函数才能拟合非线性模型。在神经网络中，激活函数添加在上层节点输出和下层节点的输入之间。

介绍最常用的三种激活函数：Sigmoid，tanh，ReLU。

* Sigmoid

数学形式：𝑓(𝑧) = 1

1+𝑒−𝑧

几何图像：

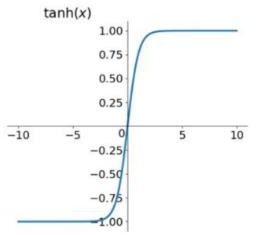
Sigmoid 能够把输入的连续实值变换为 0 和 1 之间的输出，但如果是非常大的负数，那么输出就是 0；如果是非常大的正数，输出就是1。它的缺点是在深度神经网络中梯度反向传递时会导致梯度爆炸和梯度消失，以及因为含有指数运算，求解会耗费更多的时间。

⚫ tanh

数学形式：tanh (𝑥) = 𝑒𝑥−𝑒−𝑥

𝑒𝑥+𝑒−𝑥

几何图像：

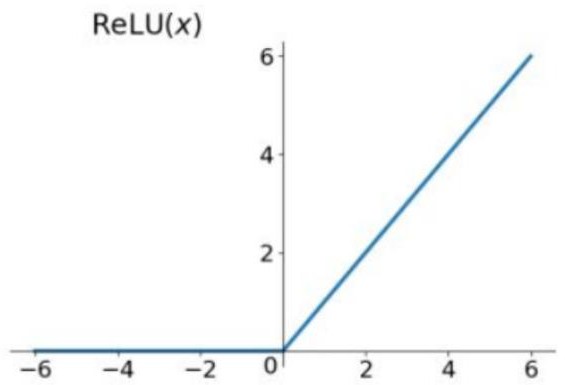


相比于 Sigmoid，tanh 当输入较大或较小时，输出几乎是平滑的并且梯度较小，这不利于权重更新。二者的区别在于输出间隔，tanh 的输出间隔为 1，并且整个函数以 0 为中心，比 sigmoid 函数更好；在tanh 中，负输入将被强映射为负，而零输入被映射为接近零。

⚫ ReLU

数学形式：Relu = 𝑚𝑎𝑥(0, 𝑥)

几何图像：



### 损失函数

ReLU 函数其实就是一个取最大值函数，但并不是全区间可导的，可以通过取 sub-gradient 来解决这个问题， 如上图所示。

ReLU 的优点主要有：

1.解决了正区间的梯度消失问题 2.计算速度快，只需要判断输入是否大于 0 3.收敛速度远快于 sigmoid 和

tanh

ReLU 是目前效果最好，最常用的激活函数，在该实验中， 主要也以 ReLU 作为激活函数。

本实验中解决的是一个多分类问题，选择交叉熵（CrossEntropy）作为损失函数，公式如下

𝑀

1 1

𝐿 = 𝑁 ∑ 𝐿𝑖 = − 𝑁 ∑ ∑ 𝑦𝑖𝑐log (𝑝𝑖𝑐)

𝑖 𝑖

𝑐=1

其中𝑀表示类别的数量，𝑦𝑖𝑐是符号函数，如果分类正确取 1，错误取 0，𝑝𝑖𝑐是观测样本属于对应类别的预测概率。

在网络训练过程中，对损失函数进行梯度下降进行优化，训练网络中的权值。

### 优化器

常用的优化器有 SGD,BGD,Adam，Momentum，RMSprop 等。在本实验中主要使用了 Adam 和

Momentum 优化器，原理如下：

⚫ Adam 优化器

Adam 吸收了 Adagrad（自适应学习率的梯度下降算法）和动量梯度下降算法的优点，既能适应稀疏梯度，又能缓解梯度震荡的问题。公式如下：

首先对梯度，梯度的平方使用指数加权平均𝑚𝑡 = 𝛽1𝑚𝑡−1 + (1 − 𝛽1)𝑔𝑡

𝑣𝑡 = 𝛽2𝑣𝑡−1 + (1 − 𝛽2)𝑔2

𝑡

𝑚̂𝑡

𝑚𝑡

1−𝛽𝑡

=

随后进行偏差修正 𝑣 1

𝑣̂𝑡 =  𝑡

1−𝛽𝑡

2

权重更新公式如下𝜃𝑡

= 𝜃𝑡−1

− 𝜂

√𝑣̂𝑡−1+𝜖

𝑚̂

𝑡−1

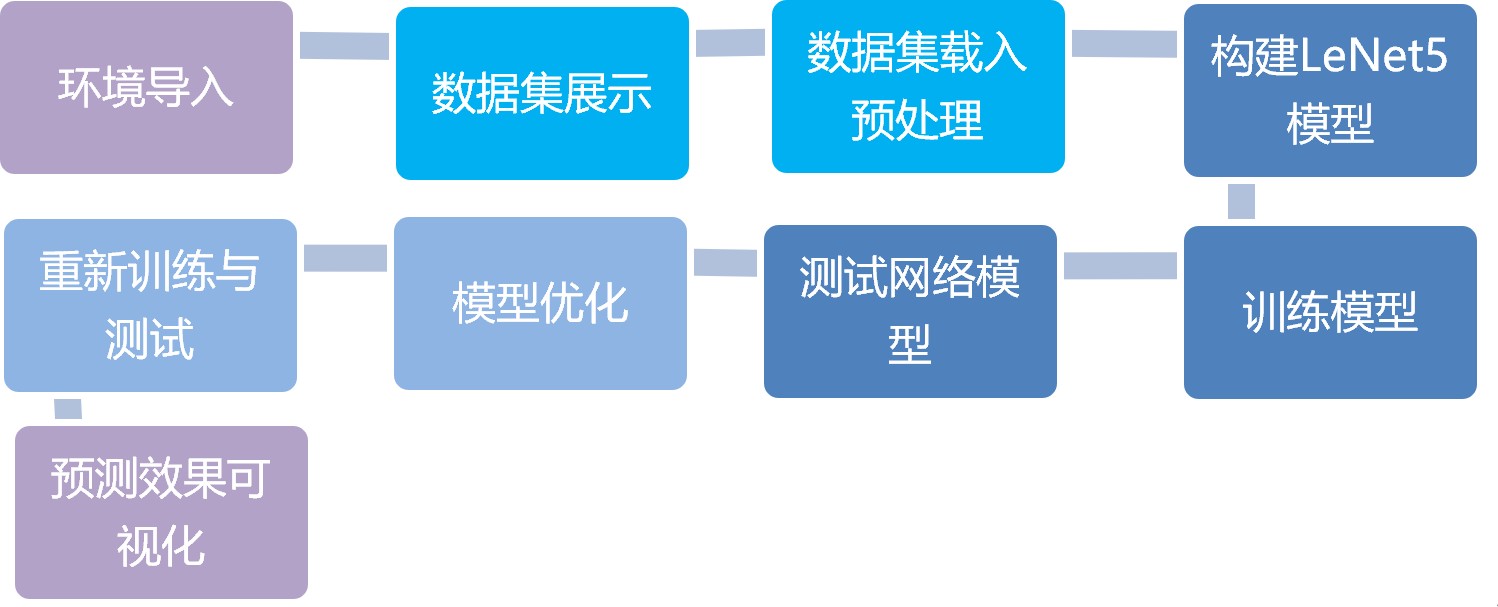
⚫ Momentum 优化器

Momentum 梯度下降算法在与原有梯度下降算法的基础上，引入了动量的概念，使网络参数更新时的方向会受到前一次梯度方向的影响，换句话说，每次梯度更新都会带有前几次梯度方向的惯性，使梯度的变化更加平滑，这一点上类似一阶马尔科夫假设；Momentum 梯度下降算法能够在一定程度上减小权重优化过程中的震荡问题。引入动量的具体方式是： 通过计算梯度的指数加权平均数来积累之前的动量，进而替代真正的梯度。公式如下：

𝑣𝑡 = 𝛾𝑣𝑡−1 + 𝜂∇𝜃𝐽(𝜃𝑡−1)

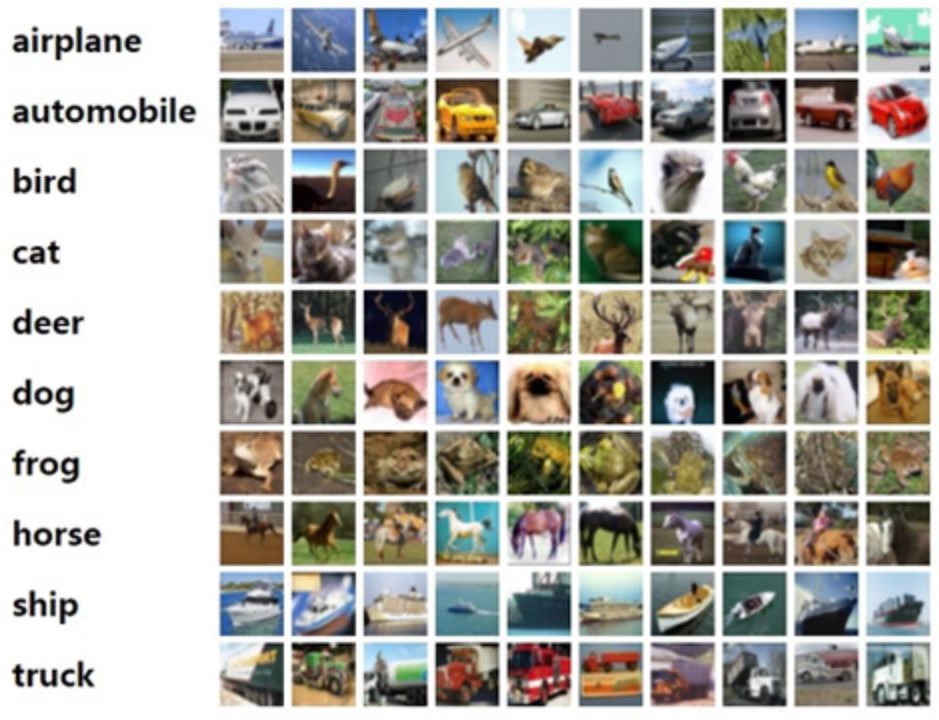
𝜃𝑡 = 𝜃𝑡−1 − 𝑣𝑡

## 三．实验步骤



## 四．数据集

本实验用 CIFAR-10 数据集, 共包含 10 类、共 60000 张图片。其中，每类图片 6000 张，

50000 张是训练集，10000 张是测试集。每张图片大小为 32\*32。部分展示如下：

## 五．程序代码

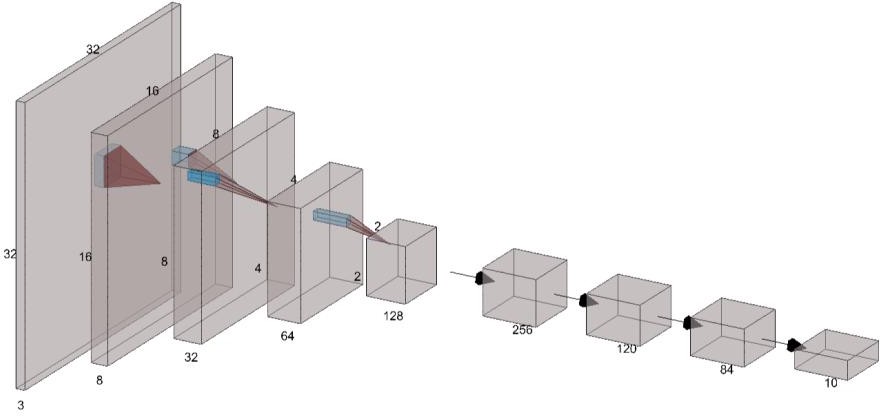
(ps: 第五节只展示代码和编程思路，关于结果的分析在第七节)

由于环境导入，数据集加载等操作手册中已经给出了详细的代码和讲解，这里不再赘述，本章节主要给出修改部分的代码，包括**数据增强，加载训练好的模型，新的网络模型搭建，新的优化器选取四部分**。给出的代码部分有自己的理解和修改，与操作手册中不同，**写在注释中**。

* 数据增强。由于全连接层的限制，输入图像的大小必须为 32\*32.对读入的图片进行随机剪裁，旋转和翻转，随后进行 resize 和 rescale 处理。数据增强能够抑制过拟合，使得图片中的物体翻转，旋转或部分缺失时都能被检测出来。
* 加载 checkpoint。进行测试时，需要读取训练时保存好的.ckpt 文件。在训练中断时也可以通过这种方法继续训练。

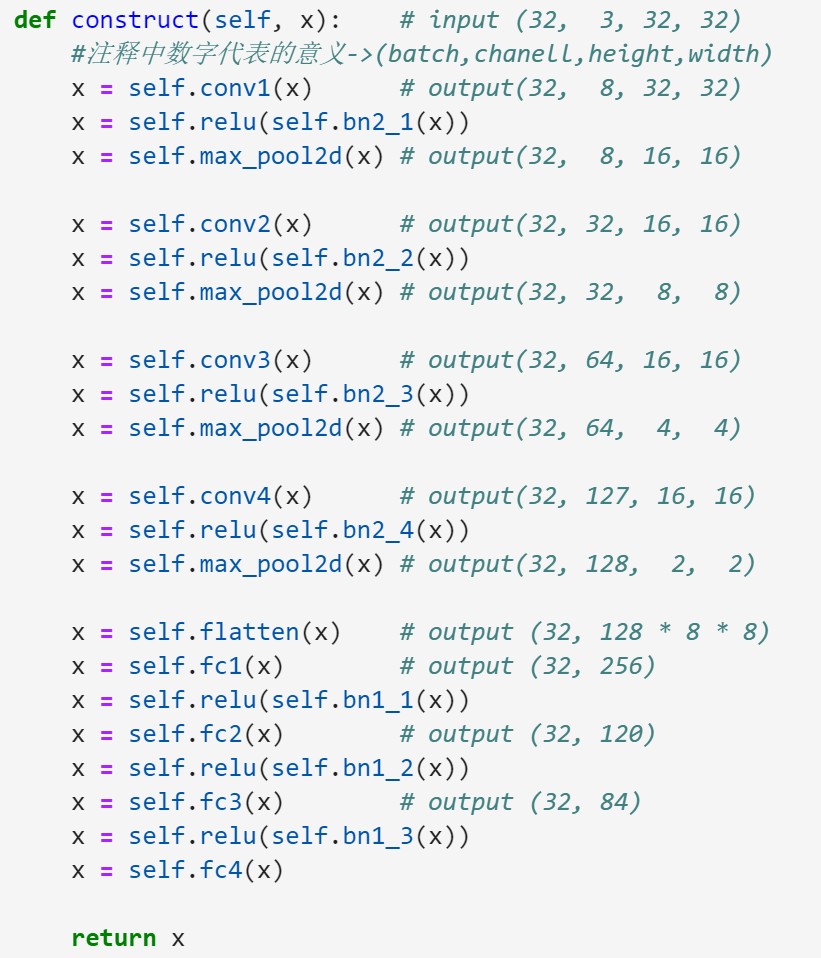


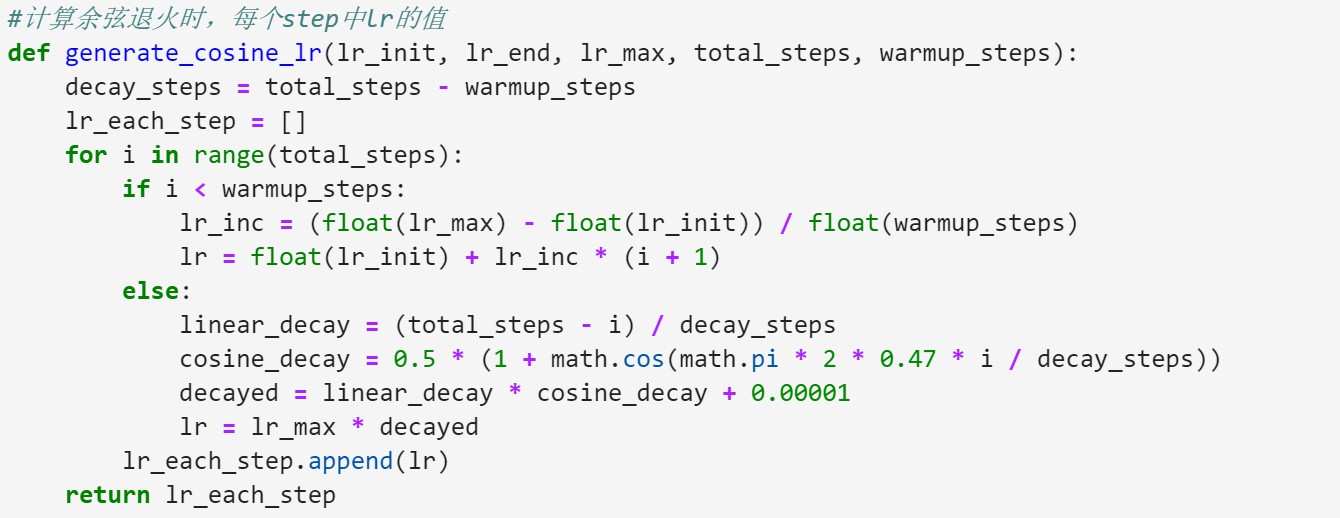
* LeNet 网络结构改进。增加到四个 conv-bn-relu-maxpool 模块，四个 fc-bn-relu模块（最后一次 fc 后不进行 bn 和 relu），另外将卷积模块的卷积核大小都调整为 3\*3.网络结构图如下（在线简易网络绘图工具 [NN SVG (alexlenail.me)](http://alexlenail.me/NN-SVG/AlexNet.html)）图中卷积过程每个方块代表 conv-bn-relu-maxpool，展平后每个方块包括 fc-bn-relu

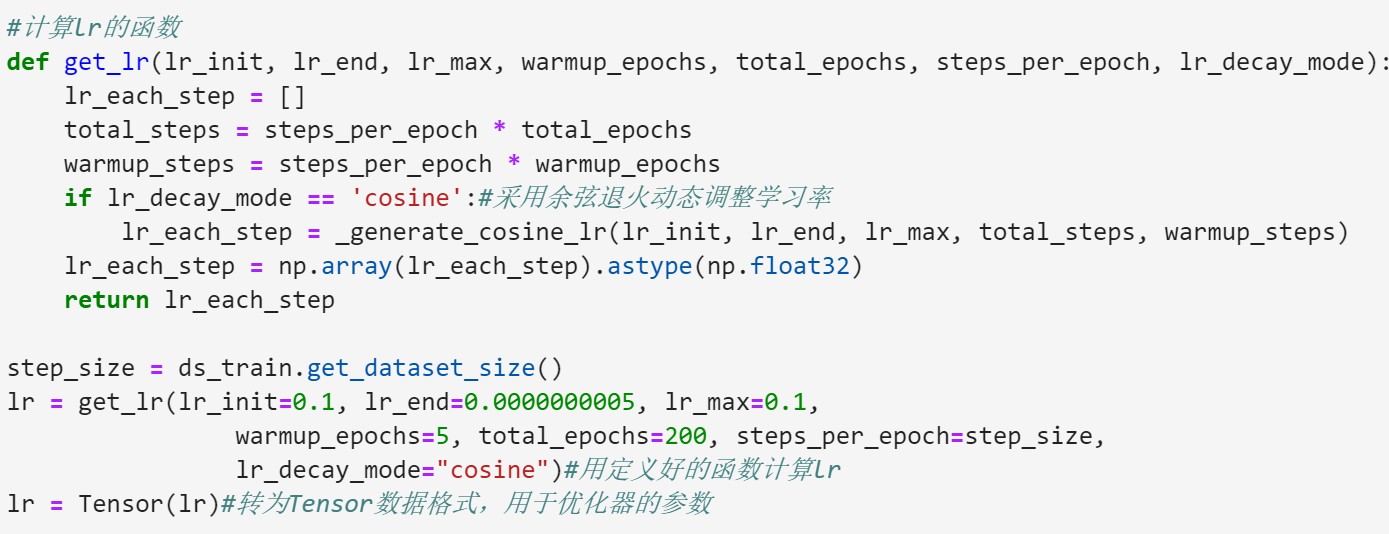


#### 代码如下

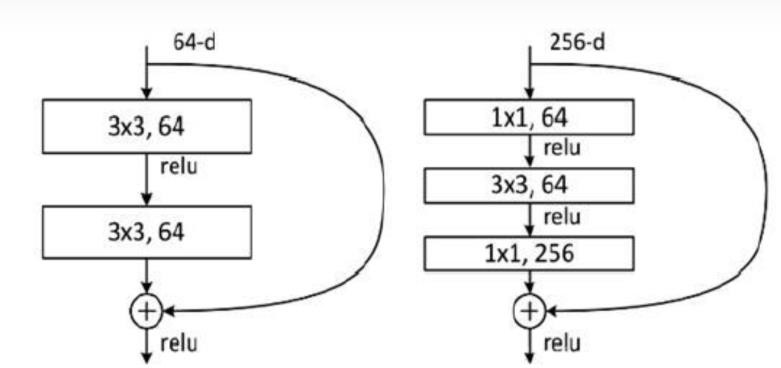




* 优化器改进。原代码中使用 Adam 固定学习率的优化器，使用动态学习率进行试验。学习率动态调整代码和优化器更改实验代码如下



* 加入残差结构，用 MindSpore 搭建 resnet 对网络进一步调优

残差结构：在网络层数加深的时候避免梯度爆炸和消失问题。通过捷径连接，直接将上层特征图 x 作为部分输出的初始结果，输出结果为 H（x） = F（x）+x，当 F（x）=0 时， 即变为恒等映射。残差结构图如下：

搭建网络结构的代码如下：

#####::’’’/.’<. (/############

class PreActBottleNeck(nn.Cell): expansion = 4

def finite(self,

1n\_pI anes , planes, stride=1):

super(PreActBottleNeck, self).finite() self.relu : nn.ReLU()

self.bn1 = nn.BatchNorm2d(in\_planes, eps=1e-5, momentum=0.9) self.conv1 = nn.Conv2d(in\_planes, planes, kernel\_size=1, stride=1) self.bn2 = nn.BatchNorm2d(planes, eps:1e-5, momentum=0.9)

self.conv2 = nn.Conv2d(planes, planes, kernel\_size=3, stride=stride, padding=1, pad\_mode='pad') self.bn3 : nn.BatchNorm2d(planes, eps:1e-5, momentum:0.9)

self.conv3 nn.Conv2d(planes, self.expansion\*planes, kernel\_size=1, stride=1)

self.downtown = False

if stride !: 1 or in planes !: self.expansion\*planes: self.downtown : True

self.shortcut nn.SequentialCell([nn.Conv2d(in\_planes, self.expansion\*planes,

kernel\_size:1, stride:stride)])

self.add = P.TensorAdd()

def

construct(self, x):

out : self.bn1(x) out : self.relu(out) if self.downtown:

identity self.shortcut(out) else:

identity : x

out : self.conv1(out)

out : self.bn2(out) out : self.relu(out) out : self.conv2(out)

out = self.bn3(out) out = self.relu(out) out = self.conv3(out)

out = self.add(out, identity)

return out

class PreActResNet(nn.Cell):

def finite(self,block,num\_blocks,in\_planes,planes, strides,low\_memory,num\_classes=10): super(PreActResNet, self).finite()

self.in\_planes: in\_planes self.low\_memory: low\_memory

self.conv1 = nn.Conv2d(3, self.in\_planes, kernel\_size=7, stride=2)

self.conv2 = nn.Conv2d(3, self.in\_planes, kernel\_size=3, stride=1, pad\_mode='pad', padding=1)

self.layer1 self.\_make\_layer(block,planes[0],num\_blocks[0],strides[0]) self.layer2 self.\_make\_layer(block,planes[1],num\_blocks[1],strides[1]) self.layer3 self.\_make\_layer(block,planes[2],num\_blocks[2],strides[2]) self.layer4 self.\_make\_layer(block,planes[3],num\_blocks[3],strides[3]) self.mean : P.ReduceMean(keep\_dims:True)

self.flatten : nn.Flatten()

self.linear nn.Dense(planes[3]\*block.expansion, num\_classes)

def \_make\_layer(self, block, planes, num\_blocks, stride): layers []

strides = [stride] + [1]\*(num\_blocks-1)

for s in strides: layers.append(block(self.in\_planes, planes, s)) self.in\_planes: planes \* block.expansion



return nn.SequentialCell(layers)

def construct(self, x): if self.low\_memory:

out = self.conv1(x) else:

out = self.conv2(x) out: self.layer1(out) out: self.layer2(out) out = self.layer3(out) out = self.layer4(out)

out: self.mean(out, (2, 3))

out: out.view(out.shape[0], -1) out: self.linear(out)

return out

#############/@ *resnetSB######¥##¥#############*

def ResNet50 (c1ass\_num=10, 1ow\_meoory=Fa1se) :

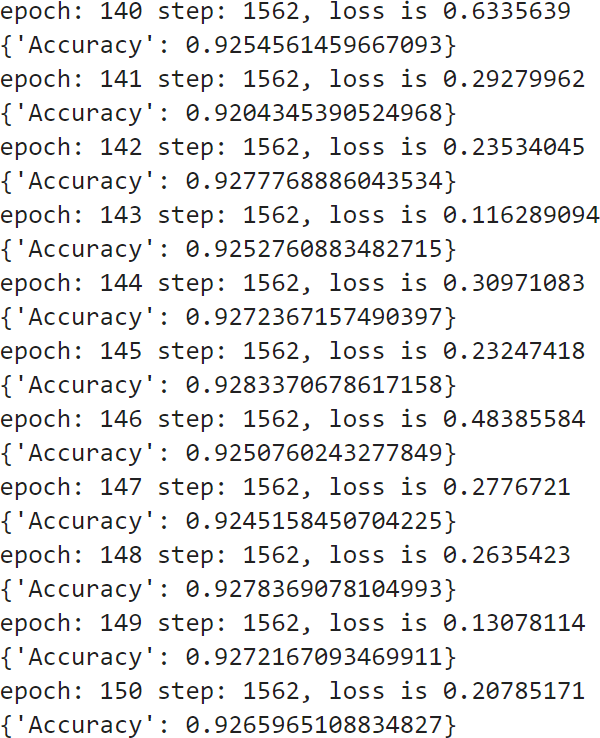
return PreActResNet(PreActBottleNeck,

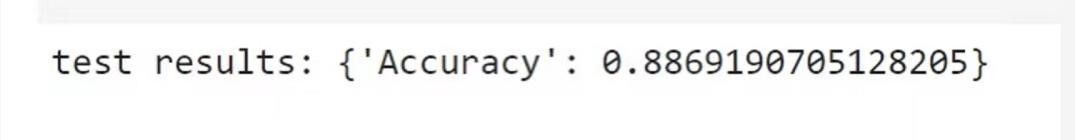
num\_b1ocks= [3, 4, 6, 3] , # esnetS08g \* 1n\_p1anes=64,

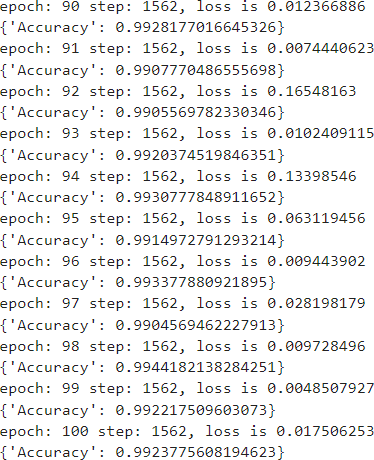
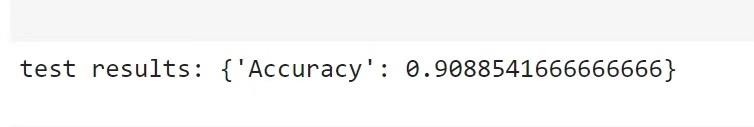
planes=[64, 12g, 256, 512],

strides=[1, 2, 2, 2], low\_memory=low\_memory, num\_classes:class\_num)

## 六．实验结果

* 在给出的 LeNet 改进网络上，训练 150 个 epoch 得到训练准确率：（并未达到 98%，可能是因为 maxpool 太多，导致最后特征丢失过多，因此后续又搭建了 resnet 进行实验）
* LeNet 改进网络测试集准确率：

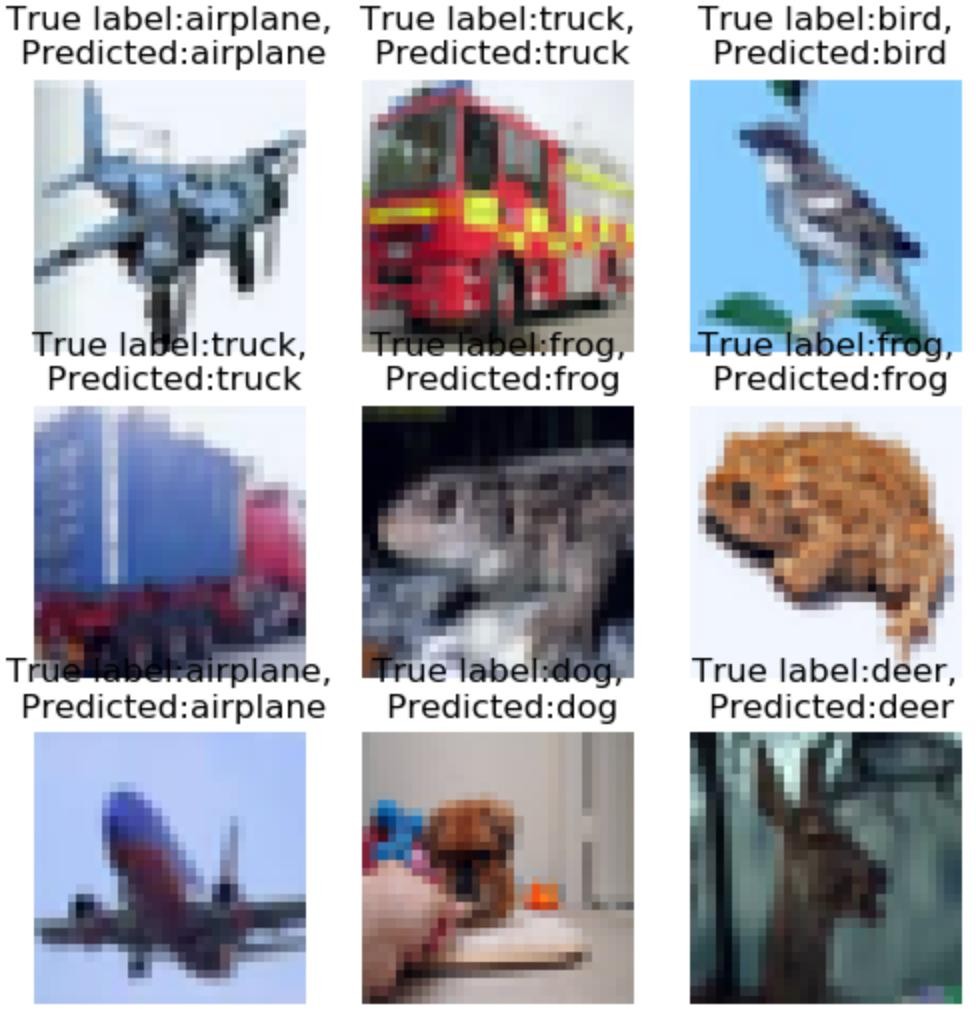


* 使用残差网络 resnet50，训练 100 个 epoch 得到的训练准确率：（达到 99%）
* 残差网络测试集准确率：

这一部分并没用相同的权重反复跑测试数据…因为jupyter 会出bug，每次都要重启notebook

才能正常运行。所以 test 的结果可能不是最优的。

* 预测图



## 七．实验分析与总结

* 关于优化器：

使用 Adam 优化器进行训练的效果较为稳定，网络收敛速度相对较快且准确率会有持续的上升；使用 Momentum 优化器进行训练，效果略差于 Adam。

* 关于学习率：

使用固定学习率训练（本实验尝试 0.0001，0.0003），准确率会稳定上升，但最终收敛的准确率相对较低。同时如果学习率大，网络收敛会比较快，准确率会比较低；学习率小， 收敛慢，但比较精细，准确率会高。

使用动态学习率训练（本实验采用余弦退火方式动态调整），网络收敛速度相对较慢且准确率是螺旋上升的，但由于使用了动态学习率，网络收敛后能达到相对较好的效果（好于固定学习率），但不太稳定（多次训练准确率相差比较大）

* 关于正则化和过拟合：

在优化器中加入 weight\_decay 进行正则化抑制过拟合取值 1e-4，在 resnet50 上进行实验，训练过程中的准确率会相对下降，但在测试集上准确率会提升 1%左右（91%~92%）。忘记截图了…QAQ

在整个实验中，过拟合现象都比较严重，表现在测试集准确率严重低于训练集。应当使用

dropout 等方法进一步抑制过拟合。

⚫ 关于网络模型：

在 ResNet50 中，使用了残差结构抑制了梯度爆炸和梯度消失，准确率在训练和测试上均有提升。分析认为与网络的加深和结构的设计有关。

自行改进的 LeNet 中，相比于原模型未进行很大的结构变动。通过添加 conv，bn 层，修改卷积核大小，也起到了提高准确率的效果。

谢谢老师(●’◡’●)