中山大学计算机学院人工智能本科生实验报告

课程：Artificial Intelligence

姓名： 学号：

一、实验题目

使用 PyTorch 手动搭建一个 ResNet 网络，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。

二、实验内容

1.算法原理

深度残差网络$(Deep\ residual\ network,\ ResNet)$相比普通$CNN$，最主要的差别在于在网络中插入了旁路。例如在浅层$ResNet$中，就是在每两个卷积层间添加一个旁路。将这样一个具有旁路的单元称为$BasicBlock$(对于浅层$ResNet$)，由于旁路的存在，其输出由：

$$y = \mathcal{F}(x, \{W\}) \xlongequal{2-layers} W_2(\sigma(W_1 x))$$

变为了：

$$y^* = \mathcal{F}(x, \{W\}) + x \xlongequal{2-layers} W_2(\sigma(W_1 x)) + x$$

旁路的添加产生的主要作用在于，使得误差的传递更加直接。在普通神经网络中，误差会因为各种原因，在每层间产生极大的衰减，甚至对于某些激活函数，例如$sigmoid$或者$tanh$，在特殊取值下甚至会使误差无法得到有效的更新。而在加入旁路后，误差则可以无损地传播，普通神经网络中浅层参数难以更新的问题得以解决。

2.伪代码

由于算力受限，选择$ResNet18$构建。

(1)网络定义

首先说明如何搭建网络。注意到宏观来看$ResNet$的基本结构是$BasicBlock$，因此我们首先实现$BasicBlock$。而对于$BasicBlock$却又是由更加基本的{卷积—归一—(激活)}操作序列构成，因此我将类 ResNet18 的结构定义如下：

```
class ResNet18(nn.Module):
    class ConvNormAct(nn.Module):
        """生成Convolution-Normalize-(Activate)序列"""
        def __init__(self, in_channels: int, out_channels: int, kernel_size: int, stride: int, p
                     bias=False, activation=True) -> object:

        def forward(self, x: torch.tensor) -> torch.tensor:

    #-----------------------------------------------------------------#

    class BasicBlock(nn.Module):
        """基于ConvNormAct生成ResNet中的BasicBlock"""
        def __init__(self, in_channels: int, out_channels: int, strides: int) -> None:

        def forward(self, x: torch.tensor) -> torch.tensor:

    #-----------------------------------------------------------------#

    def __init__(self, block: object=BasicBlock, groups: list[int]=[2, 2, 2, 2], num_classes=10)
        """基于BasicBlock生成ResNet"""

    def ConvX(self, channels: int, blocks: int, strides: int, index: int) -> nn.Sequential:
        """生成conv_x层(基于BasicBlock)"""

    def forward(self, x: torch.tensor) -> torch.tensor:
```

我们从底层开始逐层架构。在 ConvNormAct ， 参数 activation 用于决定是否在序列的最后使用激活函数。因为在$BasicBlock$中：

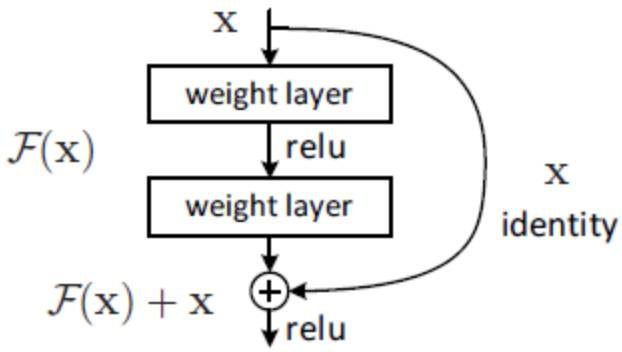


Figure 2. Residual learning: a building block.

第一个卷积层需要立即激活，而第二个卷积层则需要加上x之后才激活。

```
class ConvNormAct(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int, stride: int, padd
                 bias=False, activation=True) -> object:
        序列 = [卷积, 归一]
        if (需要激活): 序列.append(激活)

    def forward(self, x: torch.tensor) -> torch.tensor:
        return 序列(x)
```

之后，我们即可基于 `ConvNormAct` 构造 `BasicBlock` 。在这里，第一层卷积的输入通道就是整体的输入通道数，而第二层卷积的输入通道数自然就是第一层的输出通道数了。而对于$ResNet$来说，第二层卷积的输入输出通道数应当相等。

另外，应当注意旁路有实线旁路和虚线旁路之分。其本质的差异在于，旁路数据规模和处理后的数据规模可能会有差别，而这会体现在输入的步长上。当步长为1时不产生差异。否则，需要弥合之间的差异。论文中给出的解决方法是在旁路数据周围空白填充，或者使用1×1的卷积核。我们使用卷积的解决方法。

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, strides: int) -> None:
        卷积层1 = ConvNormAct(in_channels, out_channels, 步长=strides, 需要激活)
        卷积层2 = ConvNormAct(out_channels, out_channels, 步长=1, 不要激活)

        if (strides == 1): 序列 = 空白
        else: 序列 = [卷积, 归一]

    def forward(self, x: torch.tensor) -> torch.tensor:
        out = 卷积层1(x)
        out = 卷积层2(out)
        out = out + 旁路(序列(x))
        return 激活(out)
```

基本部件已经构建完成，接下来即可实现$ResNet18$了。这其中比较重要的在于如何构建论文中的$conv_i_x$层：

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

解决这个问题的函数是 `ConvX` 。初始化函数中的参数 `groups` 说明了每一个$conv_i_x$层中有多少个 `BasicBlock` ， 因此需要利用它来传递参数。

```
class ResNet18(nn.Module):
    def __init__(self, block_type: object=BasicBlock, groups: list[int]=[2, 2, 2, 2], num_classe
        """基于BasicBlock生成ResNet"""
        卷积层1 = 卷积(参数列表)
        卷积层2_x ~ 卷积层5_x = ConvX(参数列表)
        最大池化层 = 最大池化(参数列表)
        平均池化层 = 平均池化(参数列表)
        全连接层 = 全连接(分类数为num_classes)

    def ConvX(self, channels: int, blocks: int, strides: int, index: int) -> nn.Sequential:
        """生成conv_x层(基于BasicBlock)"""
        序列 = []
        for (blocks(表示需要几个BasicBlock)):
            序列.append(BasicBlock(参数列表))

    def forward(self, x: torch.tensor) -> torch.tensor:
        out = 卷积层1(x)
        out = 归一(out)
        out = 激活(out)
        out = 最大池化层(out)
        out = 卷积层2_x ~ 卷积层5_x(out)
        out = 平均池化层(out)
        out = 全连接层(out)
        out = softmax(out)
        return out
```

以上便构建好了$ResNet$的基本架构。虽然将类取名为$ResNet18$，但通过调整内部参数，也能实现构建$ResNet34$。

(2)数据加载

由于$MNIST$原数据集规模太过庞大，受限于算力，重新编写了数据加载。

```
class DataProcess(torch.utils.data.dataset.Dataset):
    """生成数据集"""
    # 为了从大规模数据集中截取一部分
    def __init__(self, data_addr: str, label_addr: str, transformation: transforms.Compose):
        数据 = parseMNIST(数据地址)
        标签 = parseMNIST(标签地址)

    def __getitem__(self, index: int) -> tuple:
        # 由于是以np.ndarray形式存储，因此需要格式转换
        返回数据 = transform(数据[index].toImage())
        返回标签 = 标签[index]
        return (返回数据, 返回标签)

    def __len__(self) -> int:
        return len(标签)

    def parseMNIST(self, file_addr: str) -> np.ndarray:
        """解析MNIST文件"""
        if (是标签文件): 按标签形式解析
        else: 按图片形式解析

        if (是测试数据集): return 数据[: 测试集规模]
        else: return 数据[: 训练集规模]
```

(3)问题任务

针对本次问题，我编写 `Classify` 类来解决。通过调用 `Solve` 方法，即可运行$ResNet$。针对本次任务的类，基本与之前中药识别CNN相同。

```
class Classify:
    def __init__(self) -> None:

    def Train(self) -> None:
        """训练函数"""
        设置神经网络为训练模式
        for 训练数据集:
            计算预测误差
            反向传播
            优化器更新

    def Test(self) -> None:
        """测试函数"""
        设置神经网络为评估模式
        with torch.no_grad():
            for 测试数据集:
                累加损失
                累加正确分类数
        记录损失
        记录准确率

    def Draw(self) -> None:
        """绘制函数"""

    def Solve(self) -> None:
        """对外的解决方法"""
        for 迭代次数:
            self.Train()
            self.Test()
        self.Draw()
```

3.关键代码展示

(1)数据集预处理

为了实现只使用$MNIST$的子集完成训练，所以使用 `DataProcess` 类来实现自己的数据集加载。

```
class DataProcess(torch.utils.data.dataset.Dataset):
    """生成数据集"""
    # 为了从大规模数据集中截取一部分
    def __init__(self, data_addr: str, label_addr: str, transformation: transforms.Compose):
        self.data = self.parseMNIST(data_addr)
        self.label = self.parseMNIST(label_addr)
        self.transform = transformation

    def __getitem__(self, index: int) -> tuple:
        img, target = self.transform(Image.fromarray(self.data[index])), int(self.label[index])
        return (img, target)

    def __len__(self) -> int:
        return len(self.label)

    def parseMNIST(self, file_addr: str) -> np.ndarray:
        """解析MNIST文件"""
        minst_file_name = os.path.basename(file_addr)  # 根据地址获取MNIST文件名字
        with gzip.open(filename = file_addr, mode = "rb") as minst_file:
            mnist_file_content = minst_file.read()
            if (minst_file_name.find("label") != -1):  # 若传入的为标签二进制编码文件地址
                data = np.frombuffer(buffer = mnist_file_content, dtype = np.uint8, offset = 8)
            else:  # 若传入的为图片二进制编码文件地址
                data = np.frombuffer(buffer = mnist_file_content, dtype = np.uint8, offset = 16)
                data = data.reshape(-1, 28, 28)
        # 截取其中一部分返回
        if (minst_file_name.find("t10k") != -1): return data[: TEST_SCALE]
        else: return data[: DATA_SCALE]
```

在这里， `TEST_SCALE`，`DATA_SCALE` 是预定义的常量，这些常量可在代码文件中的常量定义区中修改。它将整个数据集切片，从而获得适当的数据集规模。

在$MNIST$数据集中，所有文件都以二进制呈现。对于 `label` ，文件前16字节为描述字符；对于 `iamge` ，文件前8字节为描述字符。通过寻找文件中是否存在 `label` 的子字符串，来判断文件是否是标签文件。

在按一维二进制数组读取文件后，如果处理的是 `iamge` 文件，那么需要将其转换为三维数组(以描述图片)。由于$MNIST$中每张图是28×28的，因此最后需要将一维数组 `reshape` 。

而训练数据集和测试数据集的区别在于文件名中是否有 `t10k` 的子字符串。以此作为依据，产生数据集的子集供训练与测试使用。

(2)网络实现

首先给出 BasicBlock 的实现：

```
class ResNet18(nn.Module):
    class BasicBlock(nn.Module):
        """基于ConvNormAct生成ResNet中的BasicBlock"""
        def __init__(self, in_channels: int, out_channels: int, strides: int) -> None:
            super(ResNet18.BasicBlock, self).__init__()
            self.conv1 = ResNet18.ConvNormAct(in_channels, out_channels, 3, stride=strides, pad
            self.conv2 = ResNet18.ConvNormAct(out_channels, out_channels, 3, stride=1, padding=

            self.short_cut = nn.Sequential()
            if (strides != 1):
                self.short_cut = nn.Sequential(
                    nn.Conv2d(in_channels, out_channels, 1, stride=strides, padding=0, bias=Fal
                    nn.BatchNorm2d(out_channels)
                )

        def forward(self, x: torch.tensor) -> torch.tensor:
            out = self.conv1(x)
            out = self.conv2(out)
            out = out + self.short_cut(x)
            return F.relu(out)
```

BasicBlock 基于之前定义的 ConvNormAct 构建。并通过对输入参数 strides 的判断，判定此次构建的 BasicBlock 使用实线旁路还是虚线旁路。如果是虚线旁路，使用卷积生成相应规模的张量，从而运行相加。

之后是 ResNet 的具体实现：

```
def __init__(self, block_type: object=BasicBlock, groups: list[int]=[2, 2, 2, 2], num_classes=1(
        """基于BasicBlock生成ResNet"""
        super(ResNet18, self).__init__()
        self.channels = 64  # 这个参数会在创建convi_x时更新，用来记录下一次BasicBlock的输入通道数
        self.block_type = block_type

        self.conv1 = nn.Conv2d(1, self.channels, kernel_size=7, stride=2, padding=3, bias=False)
        self.batch_norm = nn.BatchNorm2d(self.channels)
        self.max_pool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2_x = self.ConvX(channels=64, blocks=groups[0], strides=1, index=2)
        self.conv3_x = self.ConvX(channels=128, blocks=groups[1], strides=2, index=3)
        self.conv4_x = self.ConvX(channels=256, blocks=groups[2], strides=2, index=4)
        self.conv5_x = self.ConvX(channels=512, blocks=groups[3], strides=2, index=5)
        self.average_pool = nn.AvgPool2d(7)
        self.fc = nn.Linear(512, num_classes) # 分类数为num_classes

    def ConvX(self, channels: int, blocks: int, strides: int, index: int) -> nn.Sequential:
        """生成conv_x层(基于BasicBlock)"""
        list_strides = [strides] + [1] * (blocks - 1)  # 对于一个conv_x，第一个卷积的stride为2，其
        conv_x = nn.Sequential()
        for i in range(len(list_strides)):
            layer_name = str("block_%d_%d" % (index, i))  # add_module要求名字不同
            conv_x.add_module(layer_name, self.block_type(self.channels, channels, list_strides[
            self.channels = channels  # 更新为下一BasicBlock的输入通道数
        return conv_x

    def forward(self, x: torch.tensor) -> torch.tensor:
        out = self.conv1(x)
        out = F.relu(self.batch_norm(out))
        out = self.max_pool(out)
        out = self.conv2_x(out)
        out = self.conv3_x(out)
        out = self.conv4_x(out)
        out = self.conv5_x(out)
        out = self.average_pool(out)
        out = out.view(out.size(0), -1)
        out = F.softmax(self.fc(out), dim=1)
        return out
```

分别根据论文中的表格，创建 conv2_x~conv5_x 。并按顺序连接以构成前向传播链。

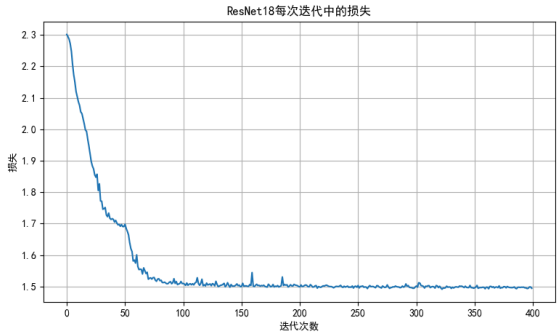
4.创新优化

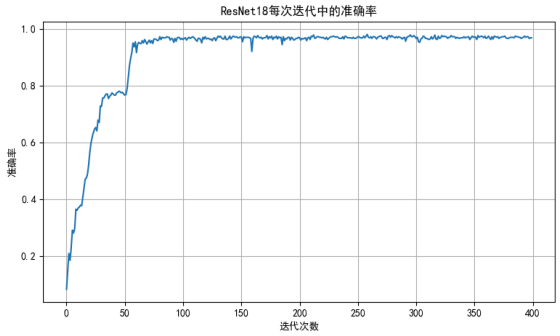
通过修改 ResNet 中初始化函数的参数，也可以构建$ResNet34$。

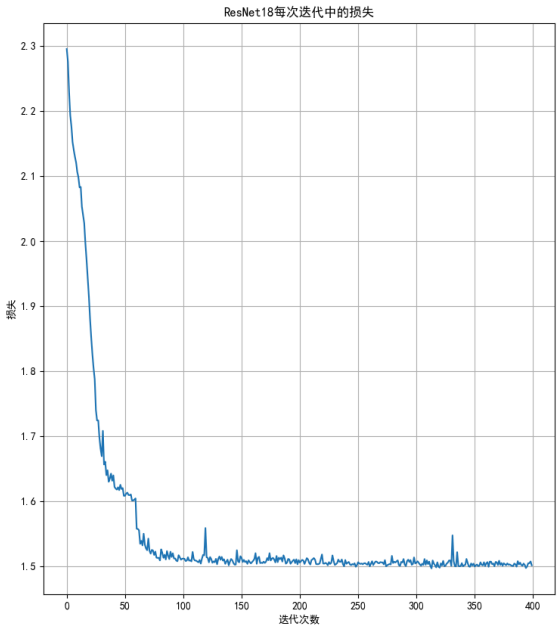
三、实验结果分析

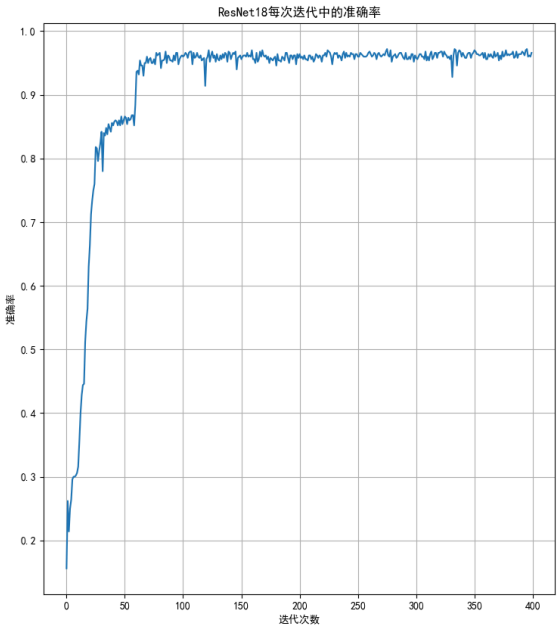
1.实验结果展示

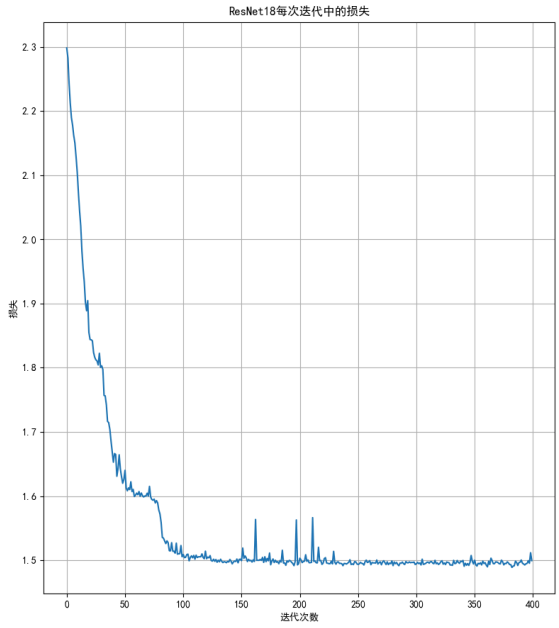
选择$MNIST$数据集，并选择训练数据集中的前1000项作为训练集，测试数据集中的前500项作为测试集。在400次迭代下，运行三次，$ResNet18$的损失与准确率分别如图所示：

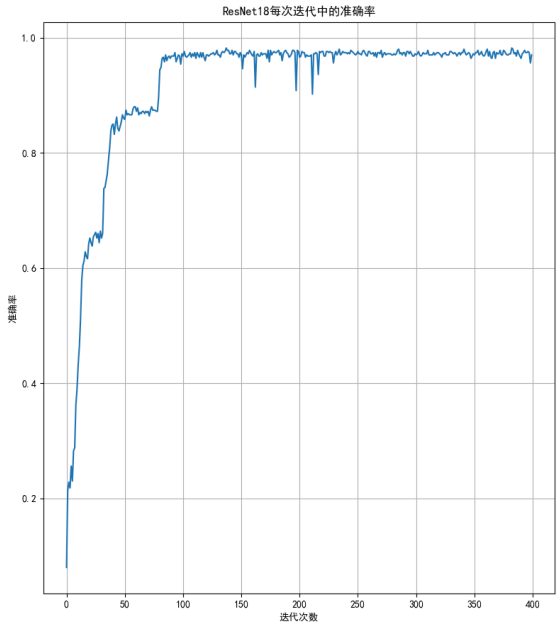












可以看到，一百次迭代基本就能达到最优，最后损失基本收敛在1.5，准确率基本收敛在96%。

2.评测指标展示分析

在上述条件下，使用$Intel$十一代$i7$训练用时十一小时。

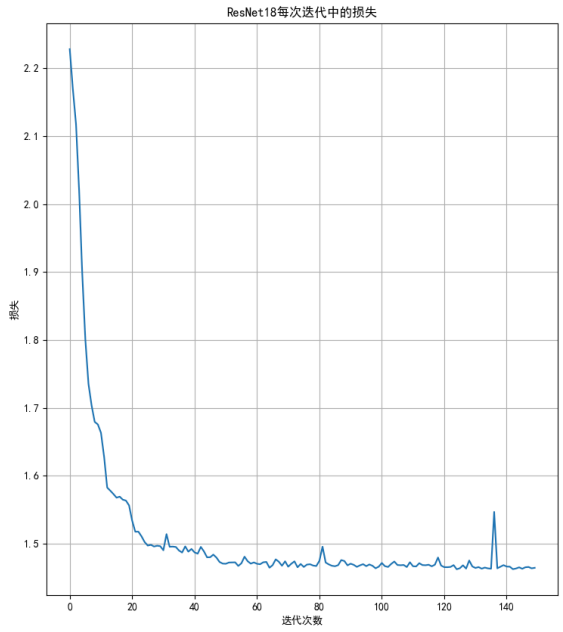
使用AMD的$Ryzen9$用时六小时。

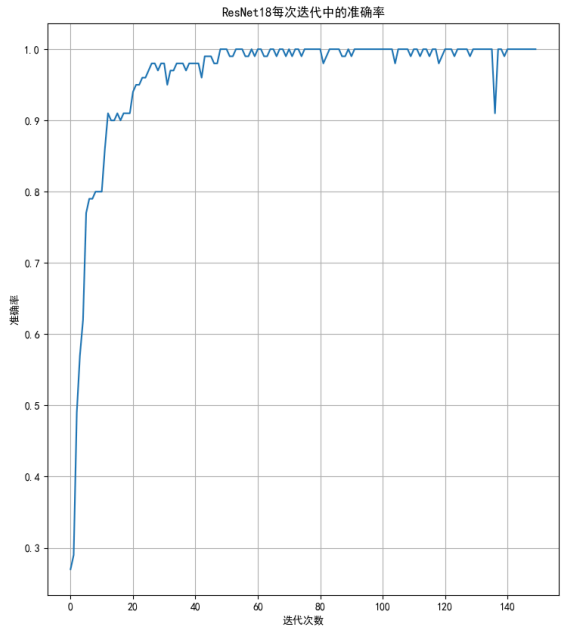
使用$Nvidia4060$用时24分钟。

从本例来看，可见硬件配置对于神经网络的运行速度有极大的影响。由于神经网络多为并行运算，基本不用考虑分支情况，张量在GPU上运算具有极大的优势。

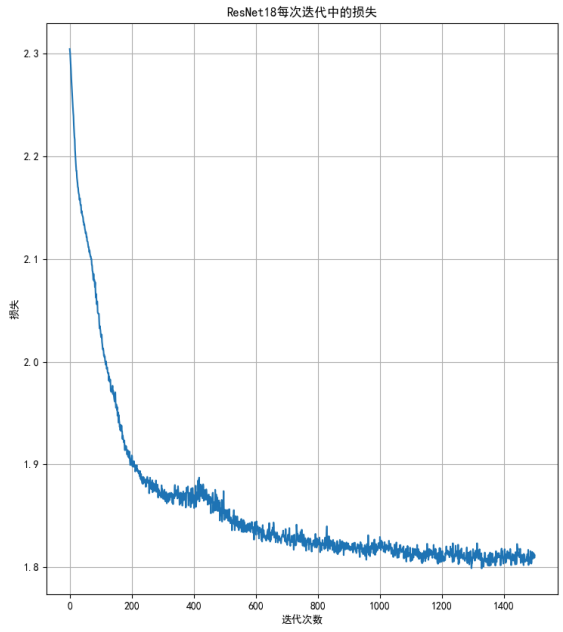
3.附录

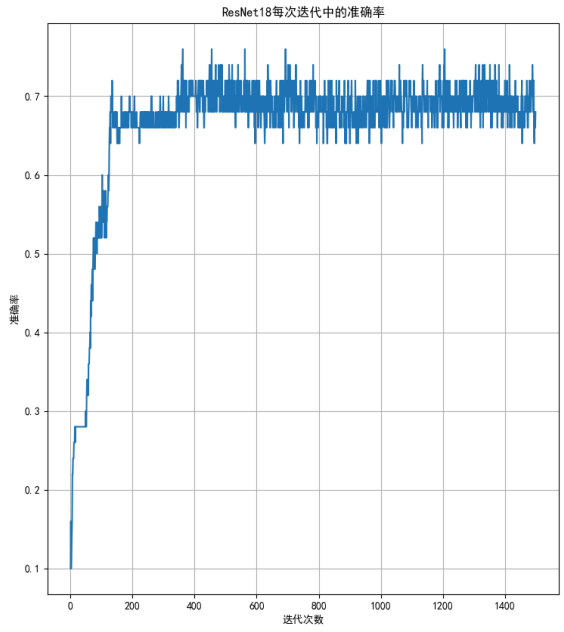
在训练集为1000，测试集为500，迭代次数400的情况下，我们发现基本上100次迭代即可达到最优效果。现在，将参数设置为训练集3000，测试集500，迭代次数150，得到的效果如下：





50次迭代便基本收敛，并且损失的收敛值更小，准确率也基本稳定在百分之百。可见影响神经网络准确率的重要参数是训练集的规模，而迭代次数并非越多越有效。例如，在训练集100，测试集50，迭代次数1500的情况下：





准确率只能收敛在70%左右，后续的迭代并不能提高准确率。

四、思考题

无。

五、参考资料

PyTorch实现ResNet亲身实践 - Yichao Cai的文章 - 知乎