模式识别作业实验报告

朱洋洋 2023e8013282066

实验简介

1. 实验目的

本实验旨在使用神经网络模型对CIFAR-10数据集进行分类。CIFAR-10是一个广泛使用的图像数据集，包含10个不同类别的60000张彩色图像。每个类别有6000张图像，其中50000张图像用于训练，10000张图像用于测试。通过本实验，我们将探索如何构建和训练一个神经网络模型来实现对图像的高效分类，并评估模型的性能。

2. 实验步骤

1. **数据预处理**：加载并预处理CIFAR-10数据集，包括数据归一化和数据增强。
2. **模型构建**：构建一个卷积神经网络（CNN）模型，适应图像分类任务。
3. **模型训练**：使用训练数据训练神经网络模型，并在验证集上监控模型的性能。
4. **模型评估**：在测试数据上评估模型的分类准确率，绘制混淆矩阵等。
5. **结果分析**：分析模型的分类结果，讨论模型的优势和不足。

3. 实验工具

- 编程语言：Python
- 深度学习框架：PyTorch
- 硬件环境：
  - CPU：AMD Ryzen 7 5800H with Radeon Graphics
  - GPU：NVIDIA GeForce RTX 3050

算法介绍

1. 神经网络概述

神经网络是一种模拟人脑结构和功能的计算模型，包含多个层次的神经元。神经网络通过学习数据中的模式和特征，能够执行复杂的任务，如图像分类、自然语言处理等。在图像分类任务中，卷积神经网络（CNN）是最常用的一种模型，其通过卷积操作和池化操作提取图像的空间特征。

2. 卷积神经网络（CNN）

卷积神经网络是一种专门用于处理图像数据的深度学习模型。CNN主要由以下几种层组成：

1. **卷积层（Convolutional Layer）**：通过卷积核对输入图像进行卷积操作，提取局部特征。卷积层可以捕捉图像中的边缘、角落等低级特征。
2. **激活层（Activation Layer）**：通常使用ReLU（Rectified Linear Unit）激活函数，引入非线性，使模型能够学习复杂的模式。
3. **池化层（Pooling Layer）**：通过对卷积层输出进行下采样，减少数据维度，提高模型的计算效率，同时保留重要特征。常用的池化方法有最大池化（Max Pooling）和平均池化（Average Pooling）。
4. **全连接层（Fully Connected Layer）**：将卷积层和池化层提取的特征进行展平，连接到输出层，用于最终的分类决策。
5. **输出层（Output Layer）**：使用Softmax函数将模型的输出转换为概率分布，用于分类任务。

3. 模型训练

模型训练过程包括以下几个步骤：

1. **前向传播（Forward Propagation）**：将输入数据通过神经网络各层，计算输出。
2. **损失计算（Loss Calculation）**：使用交叉熵损失函数（Cross-Entropy Loss）计算预测结果与真实标签之间的差异。
3. **反向传播（Backward Propagation）**：通过链式法则计算损失对模型参数的梯度，使用优化算法（如SGD、Adam）更新模型参数。
4. **迭代训练（Training Iterations）**：重复前向传播和反向传播过程，直到损失函数收敛或达到预设的训练轮次。

4. 性能评估

模型训练完成后，需要在测试集上评估其性能。常用的评估指标包括分类准确率、混淆矩阵、精确率、召回率和F1分数等。此外，可以通过绘制学习曲线和验证曲线，分析模型的训练过程和性能变化。

通过本实验，我们将深入理解神经网络的工作原理和实现过程，并掌握如何应用深度学习技术解决实际的图像分类问题。

实验内容

选择设备

本文为了加速模型的训练和推理，可以选择使用gpu硬件：

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

加载训练和测试数据

使用pytorch中的 `DataLoader` 类进行加载训练数据和测试数据。一共有六万张图片，其中训练集包含五万张，测试集包含一万张。数据集中一共有10类，训练集中每类有5000张，测试集中每类有1000张。

先求出训练集的均值和方差，用以对于数据进行初始化，使用 `transfrom` 类型来进行定义：

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
```

把 `transform` 传到数据集和训练集之中：

```
# 加载训练和测试数据
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)
```

简易模型

模型定义

先定义了一个简单的模型，由于简单的模型的训练数据较少，而且训练过程较快，所以可以很快的验证自己的想法，跑通整个流程，简单的模型训练之后，可以迁移到复杂的模型上面。

如图，模型的定义如下所示：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        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.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

模型使用了两层卷积神经网络，并在每层神经网络的计算之后，选择relu作为激活函数，使用了最大池化层；最后使用三个全连接层输出每种类别的概率。

由于一个批次batch是100，所以forward中每次结束后的形状为：

```
torch.Size([100, 3, 32, 32])
torch.Size([100, 6, 14, 14])
torch.Size([100, 16, 5, 5])
torch.Size([100, 400])
torch.Size([100, 120])
torch.Size([100, 84])
torch.Size([100, 10])
```

最后输出的结果为10个类别每个类别的概率，判断的时候选择概率最大的那个。

模型训练

模型训练的代码如下：

```
# 训练模型
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# 记录训练时间
start_train_time = time.time()
for epoch in range(30):  # 迭代次数
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 获取输入
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # 将参数梯度清零
        optimizer.zero_grad()

        # 正向传播 + 反向传播 + 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
            optimizer.step()

            # 打印统计信息
            running_loss += loss.item()
            if i % 200 == 199:    # 每200个小批量打印一次
                print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 200:.3f}')
                running_loss = 0.0

    end_train_time = time.time()
    print(f'Finished Training in {end_train_time - start_train_time:.2f} seconds')
```

在模型训练之中，损失函数使用交叉熵函数`CrossEntropy`，交叉熵函数的数学表达式为：

$$
\ell(x, y) = L = {l_1,\dots,l_N}^\top, \quad
l_n = - w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^C \exp(x_{n,c})}
\cdot \mathbb{1}{y_n \not= \text{ignore_index}}
$$

模型推理

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

模型评估的代码如上所示，使用`torch.no_grad()`，关闭梯度计算，这样可以加快推理的速度。然后每次从testloader中取一小批次进行计算，把概率最大的值作为预测值，统计总数和预测正确的数量。

训练和评估

可以每当训练到一定的次数之后，进行推理，得到训练的效果。下面的代码每进行一个epoch的训练之后，就会进行一次推理，并且计算loss，存入对应的文件之后，以便后续的画图分析。

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# 记录训练时间
# start_train_time = time.time()
for epoch in range(100):  # 迭代次数
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0): # total is 50000, batch is 100, i is 50000/100=500, idx of i is
        # 获取输入
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # 将参数梯度清零
        optimizer.zero_grad()

        # 正向传播 + 反向传播 + 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # 打印统计信息
        # print(f'[{epoch + 1}, {i + 1}] loss: {loss.item():.3f}')
```

```
            running_loss += loss.item()
            # if i % 100 == 99:    # 每100个小批量打印一次
            #     print(f'[{epoch}, {i}] loss: {running_loss / 100:.3f}')
            #     running_loss = 0.0
        avg_loss = running_loss / 500
        print(f'[{epoch}] loss: {avg_loss:.3f}')
        with open('training_loss.txt', 'a') as f_loss:
            f_loss.write(f'{epoch} {avg_loss:.3f}\n')
        # 一轮epoch训练完了，评估模型
        correct = 0
        total = 0
        for data in testloader:
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        accuracy = 100 * correct / total
        print(f'[{epoch}] Accuracy: {accuracy} %')
        with open('training_accuracy.txt', 'a') as f_accuracy:
            f_accuracy.write(f'{epoch} {accuracy:.2f} %\n')
```

复杂模型

由于硬件的资源有限，为了快速的迭代方案，先设计了一个简单的模型，用以快速实现和验证；但是简单模型的潜力非常有限，当方案固定之后，可以设计复杂的模型，从而达到更好的任务效果。

由于硬件的资源仍然有限，所以不敢把网络设计的很大，设计了ResNet9网络用以分类。

网络定义如下：

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.residual_function = 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)
        )
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        return nn.ReLU(inplace=True)(self.residual_function(x) + self.shortcut(x))


class ResNet9(nn.Module):
    def __init__(self):
        super(ResNet9, self).__init__()
        self.prep_layer = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.layer1 = nn.Sequential(BasicBlock(64, 128, stride=2))
        self.layer2 = nn.Sequential(BasicBlock(128, 256, stride=2))
        self.layer3 = nn.Sequential(BasicBlock(256, 512, stride=2))
        self.pool = nn.MaxPool2d(4)
        self.fc = nn.Linear(512, 10)
```

```
    def forward(self, x):
        x = self.prep_layer(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

经过每层之后的形状如下：

```
torch.Size([100, 3, 32, 32])
torch.Size([100, 64, 32, 32])
torch.Size([100, 128, 16, 16])
torch.Size([100, 256, 8, 8])
torch.Size([100, 512, 4, 4])
torch.Size([100, 512, 1, 1])
torch.Size([100, 512])
torch.Size([100, 10])
```

其中`BasicBlock`的解释如下：

残差函数（Residual Function）：

- 这是一个顺序容器（`nn.Sequential`），包括两个卷积层，每个卷积层后面跟一个批归一化层和ReLU激活函数。
- 第一个卷积层：接受输入通道数`in_channels`和输出通道数`out_channels`，使用3x3的卷积核，步幅为`stride`，填充为1且没有偏置。
- 第二个卷积层：输入通道数和输出通道数都是`out_channels`，使用3x3的卷积核，步幅为1，填充为1且没有偏置。
- 批归一化层（BatchNorm2d）用于加速训练并稳定模型。

捷径连接（Shortcut）：

- 这是另一个顺序容器，只有在输入和输出通道数或步幅不同时才使用，用于匹配残差连接的尺寸。
- 包含一个1x1的卷积层（改变通道数和步幅）和一个批归一化层。

然后在resnet类中定义了使用了三个这样的残差块结构。

实验结果分析

简易模型

参数选择

优化器

优化函数使用了SGD，即随机梯度下降算法，并且使用了一阶动量：
$$
g_t = \nabla f(w_t)
$$

$$
m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t
$$

求出当前下降的梯度，并根据梯度来更新参数：
$$
\eta_t = \alpha m_t \
w_{t+1} = w_t - \eta_t
$$

其中学习率`lr`参数设置为为0.001，动量`momentum`参数设置为0.9。

训练批次大小

统计训练批次大小对于训练时间和GPU使用率的影响，输出结果如下图所示：

```
Finished Training in 33.66 seconds
```

```
zhuyangyang@vm:~$ nvidia-smi
Tue Jun 18 22:44:19 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.146.02             Driver Version: 535.146.02   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3050 ...    Off | 00000000:01:00.0 Off |                  N/A |
| N/A   49C    P0              12W /  35W |    250MiB /  4096MiB |     38%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1427      G   /usr/lib/xorg/Xorg                            4MiB |
|    0   N/A  N/A      1965    C+G   ...libexec/gnome-remote-desktop-daemon       69MiB |
|    0   N/A  N/A    238126      C   ...ngyang/project/test/venv/bin/python      166MiB |
+---------------------------------------------------------------------------------------+
```

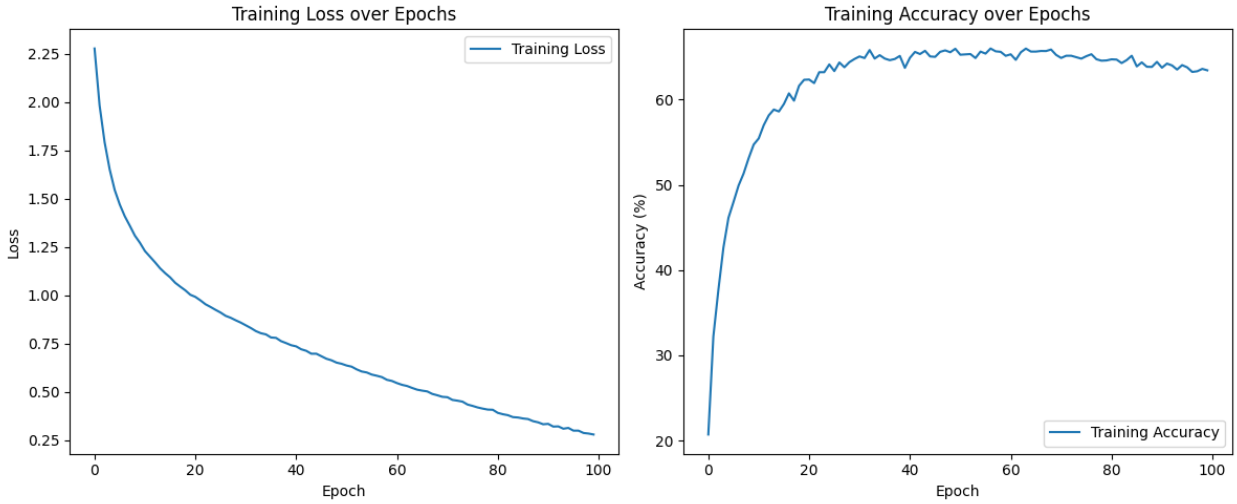
对于训练批次的选择进行了探索。探索的结果做了下图表格的汇总：

训练批次batch大小	训练时间	GPU使用率	10轮准确率
100	33.66s	38%	53.36%
200	33.23s	40%	45.43%
300	33.13s	35%~41%	40.69%

可见，训练批次大小的影响不大。反而随着batch的增加，10轮训练准确率有所下降。

训练轮数

在选择sgd优化器的情况下，进行了100轮训练，每次记录训练的准确率和损失，然后进行了绘图，图表如下所示：



如图所示，训练20轮左右以后，准确率达到了60%以上；20轮以后继续进行训练，虽然loss继续下降，但是准确率反而也在下降，应该是产生了过拟合的情况。说明这个简易模型的潜力已经被充分挖掘出来了，识别准确率大概就是65%左右。

绘图代码如下所示：

```python
# read result and plot them
import matplotlib.pyplot as plt

train_loss = '/home/zhuyangyang/project/test/src/language/python/training_loss.txt'
train_accuracy = '/home/zhuyangyang/project/test/src/language/python/training_accuracy.txt'

# Read training loss data
epochs_loss = []
loss_values = []
# with open('training_loss.txt', 'r') as f_loss:
with open(train_loss, 'r') as f_loss:
    for line in f_loss:
        epoch, loss = line.split()
        epochs_loss.append(int(epoch))
        loss_values.append(float(loss))

# Read training accuracy data
epochs_accuracy = []
accuracy_values = []
with open(train_accuracy, 'r') as f_accuracy:
    for line in f_accuracy:
        epoch, accuracy, _ = line.split()
        epochs_accuracy.append(int(epoch))
        accuracy_values.append(float(accuracy))

# Plot training loss
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(epochs_loss, loss_values, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.legend()

# Plot training accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs_accuracy, accuracy_values, label='Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training Accuracy over Epochs')
plt.legend()

# Show the plots
```

```
plt.tight_layout()
plt.show()
```

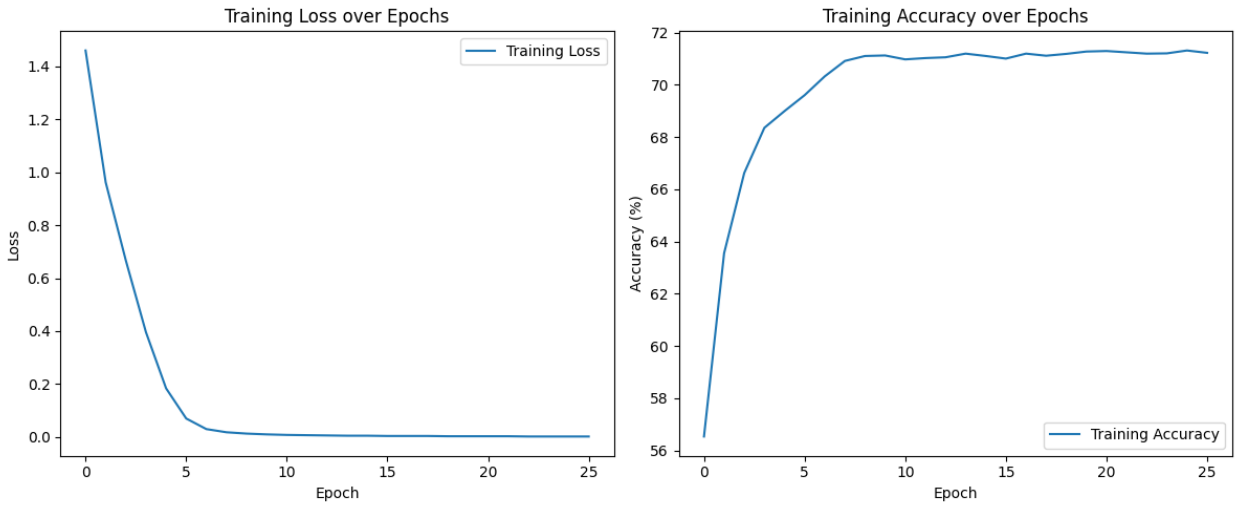
复杂模型

训练过程中，gpu的使用率接近于全满，如下图所示：

```
zhuyangyang@vm:~$ nvidia-smi
Wed Jun 19 16:22:58 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.146.02             Driver Version: 535.146.02   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3050 ...    Off | 00000000:01:00.0 Off |                  N/A |
| N/A   69C    P0              34W /  35W |    730MiB /  4096MiB |     96%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      1402      G   /usr/lib/xorg/Xorg                            4MiB |
|    0   N/A  N/A      1921    C+G   ...libexec/gnome-remote-desktop-daemon       69MiB |
|    0   N/A  N/A     28241      C   ...ngyang/project/test/venv/bin/python      646MiB |
+---------------------------------------------------------------------------------------+
```

由于模型的复杂度较大，训练比较耗费时间，因此之进行了20轮左右的训练，但是准确率已经基本稳定。结果如图所示：



如图所示，在训练6~8轮之后，结果已经基本稳定，resnet9的准确识别率大概在71%左右。