

# 1. 作业说明

---

## LeNet

基于Pytorch实现LeNet-5，并完成CIFAR10识别。

可以尝试使用一些图像预处理技术（去噪，归一化，分割等），再使用神经网络进行特征提取。

同时可以对训练过程进行可视化处理，分析训练趋势。

# 2. 环境说明

---

Windows10 + Anaconda + PyTorch1.12.1(cpuonly)

# 3. 数据集

---

CIFAR10数据集：包含 60000 张 32 X 32 的 RGB 彩色图片，总共 10 个分类。其中，包括 50000 张用于训练集，10000 张用于测试集；

`classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')`

**airplane**



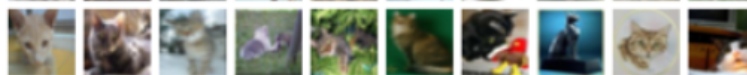
**automobile**



**bird**



**cat**



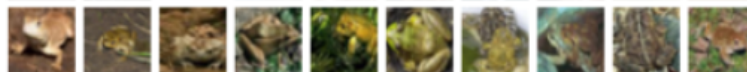
**deer**



**dog**



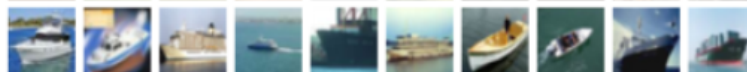
**frog**



**horse**



**ship**



**truck**

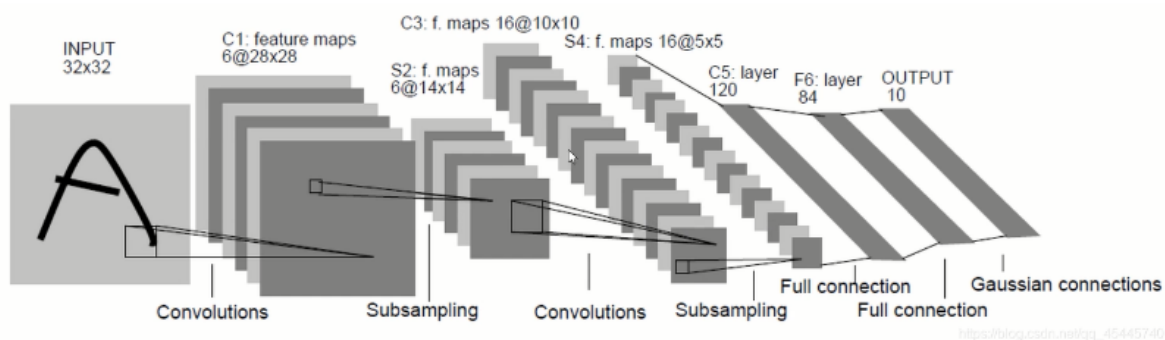


数据集分为五个训练批次和一个测试批次，每个批次有10000个图像。测试批次包含来自每个类的恰好1000个随机选择的图像。

## 4. 理论说明

### LeNet-5模型

LeNet-5模型



LeNet-5模型结构为**输入-卷积层-池化层-卷积层-池化层-全连接层-全连接层-输出**，为串联模式，如上图所示

## 5. 具体实现

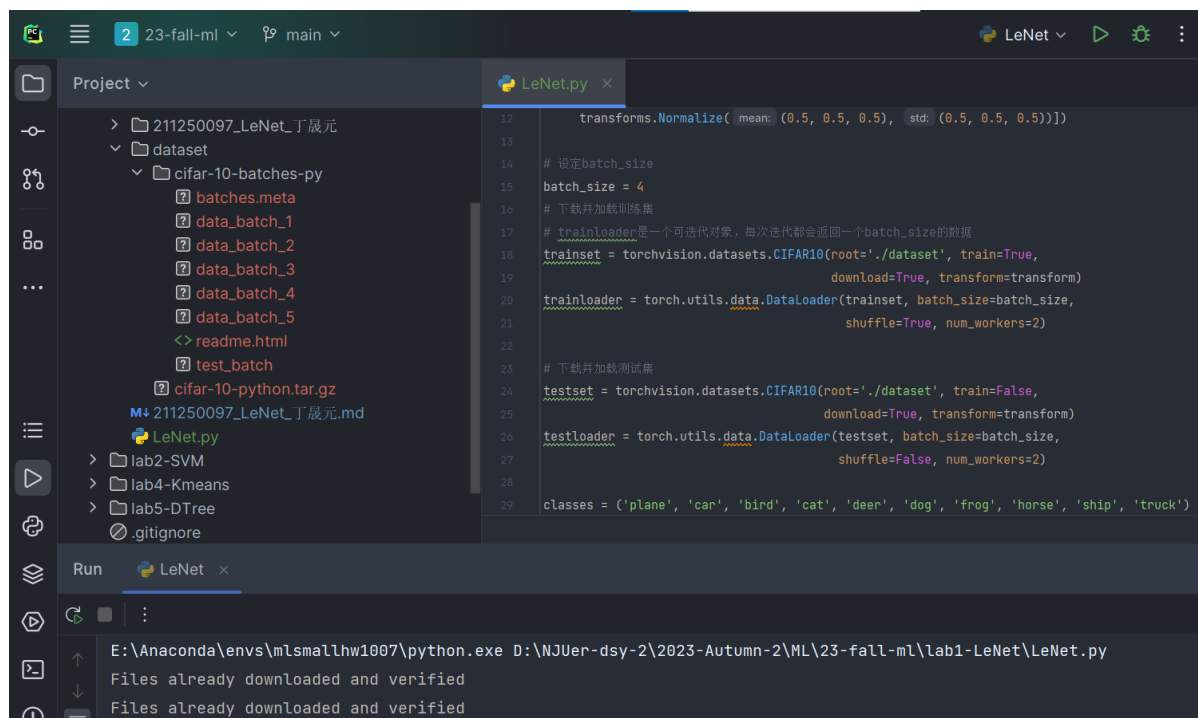
### 5.1. 前置工作

安装pytorch cpu版本

```
1 | conda install pytorch==1.12.1 torchvision==0.13.1 torchaudio==0.12.1 cpuonly -c pytorch
```

urllib3-1.26.16	201 KB	#####	100%
mkL_random-1.2.4	228 KB	#####	100%
giflib-5.2.1	88 KB	#####	100%
win_inet_pton-1.1.0	35 KB	#####	100%
torchvision-0.13.1	6.2 MB	#####	100%
libwebp-1.3.2	73 KB	#####	100%
pytorch-mutex-1.0	3 KB	#####	100%
libtiff-4.5.1	1.1 MB	#####	100%
freetype-2.12.1	490 KB	#####	100%
ca-certificates-2023	123 KB	#####	100%
blas-1.0	6 KB	#####	100%
jpeg-9e	320 KB	#####	100%
pytorch-1.12.1	133.6 MB	#####6	95%

下载数据集



## 5.2. 具体步骤分析

### 1. 数据预处理

- 归一化：先使用np的函数计算了mean和std，再使用transform对数据集进行处理  
train\_set的shape是(50000, 32, 32, 3)，分别是sample个数，图像的高，图像的宽，通道数  
处理之后将其载入DataLoader中，每次迭代都会返回一个batch\_size的数据

```
1 # 先下载数据集进行计算std和mean
2 train_set = torchvision.datasets.CIFAR10(root='./dataset', train=True,
3                                       download=True)
4 mean = train_set.data.mean(axis=(0, 1, 2)) / 255
5 std = train_set.data.std(axis=(0, 1, 2)) / 255
6 print("mean:", mean)
7 print("std:", std)
8
9 transform = transforms.Compose(
10     [transforms.ToTensor(),
11      transforms.Normalize(mean, std)])
12
13 # train_loader是一个可迭代对象，每次迭代都会返回一个batch_size的数据
14 # num_workers表示用几个子进程来加载数据
15 train_set = torchvision.datasets.CIFAR10(root='./dataset', train=True,
16                                       download=True, transform=transform)
17 train_loader = torch.utils.data.DataLoader(train_set,
18                                       batch_size=batch_size, shuffle=True, num_workers=2)
19
20 # 下载并加载测试集
21 test_set = torchvision.datasets.CIFAR10(root='./dataset', train=False,
22                                       download=True, transform=transform)
23 test_loader = torch.utils.data.DataLoader(test_set,
24                                       batch_size=batch_size, shuffle=False, num_workers=2)
```

- 去噪：可以采用图像去噪算法，如高斯滤波器、中值滤波器等，以减少图像中的噪声对模型的干扰。

但该数据集是已经较为干净的数据集，所以不需要进行去噪处理。

## 2. 建立模型LeNet模型

参照上文的模型结构图进行建立

```
1  # Step2 定义一个卷积神经网络
2  class MyNet(nn.Module):
3      def __init__(self):
4          super().__init__()
5          in_channels = 3
6          # 卷积层1
7          # 参数分别是输入通道数，输出通道数，卷积核大小
8          # in_channels指明了输入的通道数，这里是3，因为输入的是RGB图像
9          # out_channels指明了输出的通道数，我们将从中提取18个特征
10         self.conv1 = nn.Conv2d(in_channels, in_channels * 6, 5)
11
12         # 池化层1
13         # 参数分别是池化核大小，步长
14         # 这表示使用最大池化操作，池化核的大小是2x2，步长也是2。这意味着在每次池化操作
15         # 中，特征图被划分为2x2的区域，
16         # 然后在每个区域内选择最大值作为输出。步长为2表示池化操作会每隔2个像素进行一次操作，不会重叠。
17         self.pool1 = nn.MaxPool2d(2, 2)
18
19         # 卷积层2
20         # 这里的输入通道数是18，因为上一层的输出是18个特征图
21         # 输出通道数是18，从中提取48个特征
22         self.conv2 = nn.Conv2d(in_channels * 6, in_channels * 16, 5)
23
24         # 池化层2
25         # 其实这里的池化层和上面的池化层1是一样的
26         # 经过该层后，每张图片对应 16 * 5 * 5 * 3个特征
27         self.pool2 = nn.MaxPool2d(2, 2)
28
29         # 全连接层1
30         # 这里的输入是16*5*5*3
31         self.fc1 = nn.Linear(16 * 5 * 5 * in_channels, 120 * in_channels)
32
33         # 全连接层2
34         # 这里的输入是120，输出是84
35         self.fc2 = nn.Linear(120 * in_channels, 84 * in_channels)
36
37         # 全连接层3
38         # 这里的输入是84，输出是10
39         self.fc3 = nn.Linear(84 * in_channels, 10)
40
41     def forward(self, x):
42         x = self.pool1(F.relu(self.conv1(x)))
43         x = self.pool2(F.relu(self.conv2(x)))
44
```

```

45         # x是一个4维的tensor，第一维是batch_size，第二维是通道数，第三维和第四维是图像
    的高和宽
46         # 经过flatten操作后，x变成了2维的tensor，第一维是batch_size，第二维是通道数*
    图像的高*图像的宽
47         # 因为下一层是全连接层（线性层），所以需要将图像展开成一维的
48         x = torch.flatten(x, 1) # flatten all dimensions except batch
49
50         x = F.relu(self.fc1(x))
51         x = F.relu(self.fc2(x))
52         x = self.fc3(x)
53         return x

```

### 3. 定义损失器和优化函数

```

1 # Step3 定义损失函数和优化器
2 # 交叉熵损失函数
3 criterion = nn.CrossEntropyLoss()
4 # 优化器，使用随机梯度下降算法，学习率为0.001，动量为0.9
5 # optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9)
6 optimizer = optim.Adam(net.parameters(), lr=LR)

```

### 4. 训练及可视化

#### 迭代训练数据集

在每个epoch中，使用 `enumerate(train_loader, 0)` 来获取训练数据集的迭代器，其中 `train_loader` 是已经加载好的训练数据集。然后使用一个循环来遍历每个batch的数据。

#### 前向传播和反向传播

在每个batch中，首先将输入数据 `inputs` 输入到LeNet-5模型中进行前向传播，得到输出 `outputs`。然后，计算模型的损失函数 `loss`，例如交叉熵损失函数。接下来使用反向传播方法计算梯度，并通过调用 `optimizer.step()` 来更新模型的参数。

#### 计算训练损失

在每个epoch的训练过程中，累加每个batch的损失值 `loss.item()` 到 `running_loss` 中，以便后续计算平均训练损失。这个损失值可以用来观察训练过程中的损失下降情况。

#### 测试模型准确率

在每个epoch结束后，使用测试数据集对模型进行评估。通过将测试数据传入已经训练好的LeNet-5模型中，可以获得模型的输出 `outputs`。然后，使用 `torch.max()` 函数找到输出中的最大值，并返回对应的索引，即预测的类别。将预测的类别与真实标签 `labels` 进行比较，并计算正确预测的数量。

```

1 # Step4 训练网络
2 for epoch in range(epochs):
3     loop = tqdm.tqdm(
4         enumerate(train_loader, 0),
5         total=len(train_loader),
6         desc=f'Epoch [{epoch + 1}/{epochs}]',
7         ncols=100,
8     )
9
10    running_loss = 0.0
11    # 获取训练数据
12    for i, data in loop:
13        inputs, labels = data

```

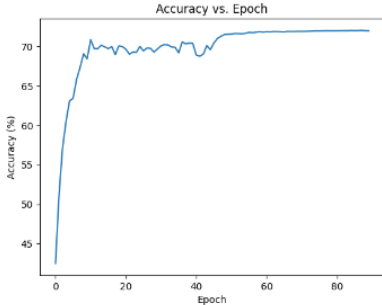
```

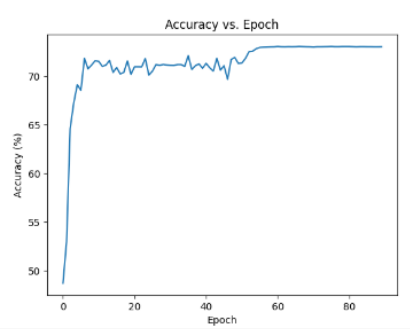
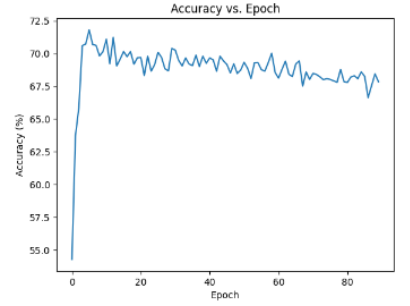
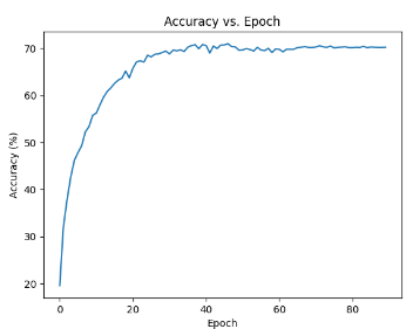
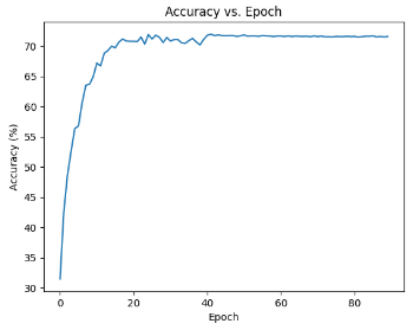
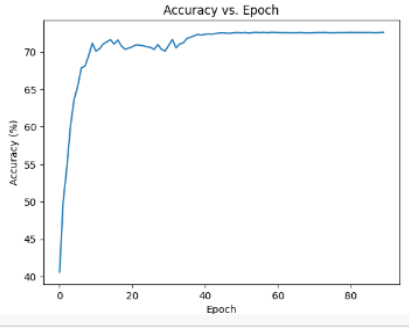
14         # 梯度清零
15         optimizer.zero_grad()
16         # 前向传播
17         outputs = net(inputs)
18         loss = criterion(outputs, labels)
19         # 反向传播, 计算梯度, 更新参数
20         loss.backward()
21         optimizer.step()
22         running_loss += loss.item()
23
24     loop.set_postfix(loss=running_loss / len(train_loader))
25
26     # 每1个epoch记录一次准确率
27     correct = 0
28     total = 0
29     # 因为加载的模型是训练好的, 所以不需要计算梯度
30     with torch.no_grad():
31         for data in test_loader:
32             images, labels = data
33             outputs = net(images)
34             # torch.max()返回两个tensor, 第一个tensor是最大值, 第二个tensor是最大值
的索引
35             # 这里我们只需要索引
36             _, predicted = torch.max(outputs.data, 1)
37             # labels.size(0)是一个batch中label的个数, 也就是4
38             total += labels.size(0)
39             # (predicted == labels)返回一个tensor, tensor中的元素是True或者False
40             # tensor.sum()将True转化为1, False转化为0, 然后求和
41             correct += (predicted == labels).sum().item()
42
43     accuracy = 100 * correct / total # 不需要使用 //, 以保留小数
44     formatted_accuracy = f'{accuracy:.2f}' # 将准确率格式化为带两位小数的字符串
45     print(f'Accuracy at epoch {epoch + 1}: {formatted_accuracy} %')
46     accuracy_values.append(formatted_accuracy) # 保存带两位小数的准确率
47     epoch_numbers.append(epoch + 1)

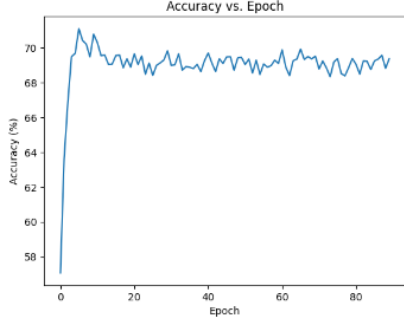
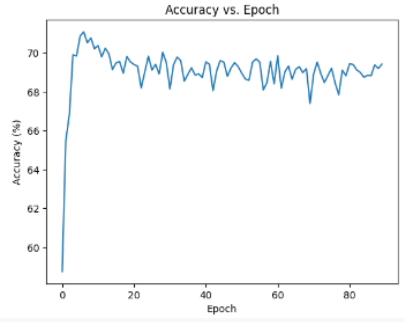
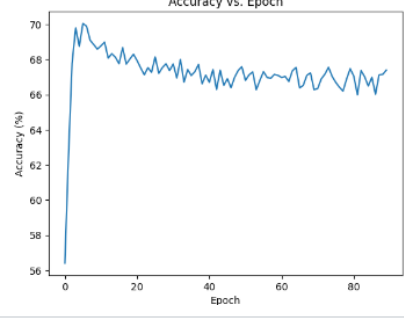
```

## 6. 结果

准确率来看均在70%以上, 最好的达到73%

batch_size	优化器	LR	图	最好 epoch	准确率
128	SGD	0.005		87	72.05

batch_size	优化器	LR	图	最好 epoch	准确率
64	SGD	0.005		60, 66, 75	<b>73.08</b>
32	SGD	0.005		5	71.81
128	SGD	0.001		46	70.98
64	SGD	0.001		41	71.97
32	SGD	0.001		89	72.63

batch_size	优化器	LR	图	最好 epoch	准确率
128	Adam	0.001		5	71.1
64	Adam	0.001		6	71.08
32	Adam	0.001		5	70.05

## 分析

从准确率变化曲线来看，明显SGD优化器的效果要好于Adam

- SGD（随机梯度下降）优化器：  
SGD是一种基本的优化算法，在每个batch中根据当前样本计算梯度并更新模型参数。SGD的特点是简单直接，每次迭代只使用一个样本进行梯度计算和参数更新，因此计算开销较小。然而，这种随机性也可能导致优化过程中出现一些波动，尤其是在训练数据集较小或噪声较多的情况下。但是，通过适当的学习率调整和较多的迭代次数，SGD可以在合理的时间内收敛到较好的解。
- Adam优化器：  
Adam是一种自适应学习率的优化算法，结合了动量法和自适应学习率机制。它可以根据梯度的一阶矩估计（均值）和二阶矩估计（方差）自适应地调整学习率，从而更好地适应不同参数的变化情况。Adam在很多情况下表现良好，并且具有较快的收敛速度。然而，对于某些数据集和模型结构，Adam可能会在训练过程中过早收敛或陷入局部最优解，尤其是在学习率设置不当时。这可能导致准确率变化曲线的性能较差。

## 7. 参考链接

官方demo：

[Training a Classifier — PyTorch Tutorials 2.1.0+cu121 documentation](#)



查找PyTorch API:

<https://pytorch.org/docs/stable/index.html>