

姓名和学号？	刘瑜，22020006076
本实验属于哪门课程？	中国海洋大学24秋《软件工程原理与实践》
实验名称？	实验2：深度学习基础

一、实验内容

【第一部分：代码练习】

2.1 pytorch基础练习

1.导入torch库，创建张量。


```
7秒  import torch


x = torch.tensor(666)
print(x)

 tensor(666)
```

个人想法和解读：在 PyTorch 中，张量是多维数组的通用名称，可以是标量（0 维）、向量（1 维）、矩阵（2 维）或更高维度的数据结构。666 是一个标量，所以生成的是一个 0 维的张量。


2.创建一个一维张量


```
0秒  # 可以是一维数组（向量）
x = torch.tensor([1, 2, 3, 4, 5, 6])
print(x)


 tensor([1, 2, 3, 4, 5, 6])
```


个人想法和解读：使用 PyTorch 中的 torch.tensor() 函数创建了一个包含 [1, 2, 3, 4, 5, 6] 这些整数的张量。

3.创建任意维度的张量。

```
0秒 [3]  # 可以是二维数组（矩阵）
x = torch.ones(2, 3)
print(x)

 tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
0秒  # 可以是任意维度的数组（张量）
x = torch.ones(2, 3, 4)
print(x)

 tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],
        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]])
```

个人想法和解读：可以使用torch创建任意维度的张量。

4. 创建一个空张量或随机初始化张量

```
✓ 0秒 [5] # 创建一个空张量
      x = torch.empty(5,3)
      print(x)

      tensor([[[-1.6789e-13,  3.1746e-41, -1.8714e-13],
              [ 3.1746e-41,  1.1210e-43,  0.0000e+00],
              [ 8.9683e-44,  0.0000e+00,  2.8869e-11],
              [ 3.1739e-41,  0.0000e+00,  0.0000e+00],
              [ 0.0000e+00,  0.0000e+00,  0.0000e+00]])

✓ 0秒 [6] # 创建一个随机初始化的张量
      x = torch.rand(5,3)
      print(x)

      tensor([[0.3854, 0.2663, 0.2452],
              [0.8825, 0.5796, 0.7036],
              [0.5917, 0.9652, 0.4571],
              [0.8578, 0.0961, 0.1717],
              [0.1667, 0.5117, 0.2876]])
```

5. 定义张量中的数据类型

```
✓ 0秒 # 创建一个全0的张量，里面的数据类型为 long
      x = torch.zeros(5,3, dtype=torch.long)
      print(x)

      tensor([[0, 0, 0],
              [0, 0, 0],
              [0, 0, 0],
              [0, 0, 0],
              [0, 0, 0]])
```

个人想法和解读：添加函数参数dtype可以选择变量的类型

6. 创建新的张量或改变张量的数据类型

```
✓ 0秒 [9] # 基于现有的tensor，创建一个新tensor，
      # 从而可以利用原有的tensor的dtype, device, size之类的属性信息
      y = x.new_ones(5,3) #tensor new_* 方法，利用原来tensor的dtype, device
      print(y)

      tensor([[1, 1, 1],
              [1, 1, 1],
              [1, 1, 1],
              [1, 1, 1],
              [1, 1, 1]])

✓ 0秒 z = torch.randn_like(x, dtype=torch.float) # 利用原来的tensor的大小，但是重新定义了dtype
      print(z)

      tensor([[[-0.8058,  0.6647, -1.8249],
              [-0.2555,  0.7827,  0.6474],
              [-0.0696, -0.5412, -0.6634],
              [ 0.0890, -1.8923, -0.9052],
              [ 0.0800,  0.4625,  0.9035]])
```

个人想法和解读：这段代码展示了 PyTorch 中创建新 tensor 的灵活性，通过 new_* 方法可以方便地继承已有 tensor 的属性，如数据类型和设备，而不需要手动指定。这对保持数据一致性和减少代码重复非常有用，同时还能灵活调整形状或重新定义数据类型，使代码更加简洁、高效。

7.查找张量数组中的数据

```
✓ [13] # 创建一个 2x4 的tensor
0秒 m = torch.Tensor([[2, 5, 3, 7],
                    [4, 2, 1, 9]])

      print(m.size(0), m.size(1), m.size(), sep=' -- ')

      ↻ 2 -- 4 -- torch.Size([2, 4])

✓ [14] # 返回 m 中元素的数量
0秒 print(m.numel())

      ↻ 8

✓ [15] # 返回 第0行, 第2列的数
0秒 print(m[0][2])

      ↻ tensor(3.)

✓ [16] # 返回 第1列的全部元素
0秒 print(m[:, 1])

      ↻ tensor([5., 2.])

✓ [17] # 返回 第0行的全部元素
0秒 print(m[0, :])

      ↻ tensor([2., 5., 3., 7.])
```

个人想法和解读：这段代码展示了如何创建和操作一个 2x4 的 Tensor。在创建后，使用 size() 方法获取维度信息，numel() 方法返回元素总数，通过索引访问特定元素和切片获取行或列的数据。

8.标量乘法运算

```
✓ [19] # Create tensor of numbers from 1 to 5
0秒 # 注意这里结果是1到4, 没有5
      v = torch.arange(1, 5)
      print(v)

      ↻ tensor([1, 2, 3, 4])

✓ [20] # Scalar product
0秒 m @ v.float()

      ↻ tensor([49., 47.])

✓ [21] # Calculated by 1*2 + 2*5 + 3*3 + 4*7
0秒 m[[0], :] @ v.float()

      ↻ tensor([49.])

[ ] # Add a random tensor of size 2x4 to m
      m + torch.rand(2, 4)

      ↻ tensor([[2.4351, 5.9721, 3.5852, 7.3047],
              [4.0053, 2.4374, 1.8735, 9.4239]])
```

个人想法和解读：通过 torch.arange(1, 5) 创建了一个包含数字 1 到 4 的一维 tensor。接着使用 @ 运算符执行矩阵乘法，计算了 m 和 v 的点积，得到了标量结果。第二个乘法示例通过选择 m 的第一行，展示了如何仅对特定行进行操作。

9.转置和生成等间隔张量

```
[23] # * 转置, 由 * 2x4 * 变为 * 4x2 *
print(m.t())

# * 使用 * transpose * 也可以达到相同的效果, 具体使用方法可以百度
print(m.transpose(0, -1))
```

```
tensor([[2., 4.],
        [5., 2.],
        [3., 1.],
        [7., 9.]])
tensor([[2., 4.],
        [5., 2.],
        [3., 1.],
        [7., 9.]])
```

```
# * returns * a * 1D * tensor * of * steps * equally * spaced * points * between * start=3, * end=8 * and * steps=20
torch.linspace(3, 8, 20)
```

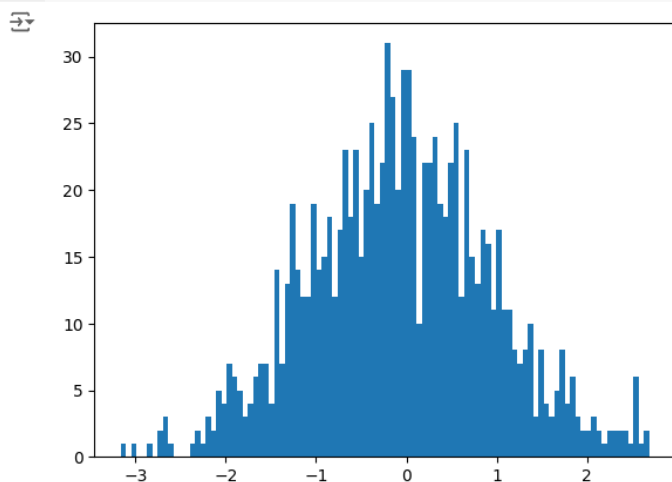
```
tensor([3.0000, 3.2632, 3.5263, 3.7895, 4.0526, 4.3158, 4.5789, 4.8421, 5.1053,
        5.3684, 5.6316, 5.8947, 6.1579, 6.4211, 6.6842, 6.9474, 7.2105, 7.4737,
        7.7368, 8.0000])
```

个人想法和解读：转置一个 2x4 的 Tensor m，将其变为 4x2 的形状，使用 m.t() 或 m.transpose(0, 1) 都可以实现这一点；torch.linspace(3, 8, 20) 用于生成从 3 到 8 的一维 Tensor，其中包含 20 个等间隔的数值。

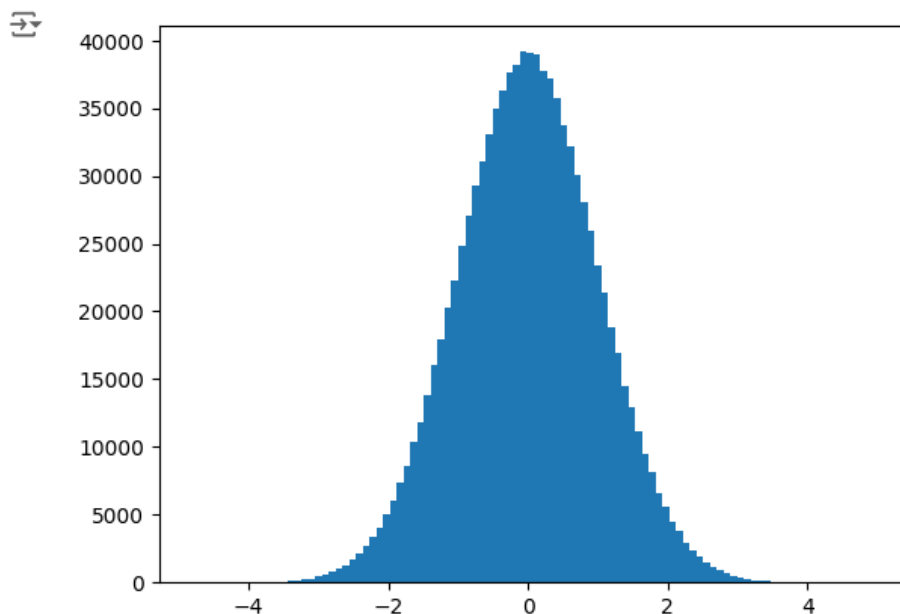
10.生成正态分布

```
from matplotlib import pyplot as plt

# matplotlib 只能显示numpy类型的数据, 下面展示了转换数据类型, 然后显示
# 注意 randn 是生成均值为 0, 方差为 1 的随机数
# 下面是生成 1000 个随机数, 并按照 100 个 bin 统计直方图
plt.hist(torch.randn(1000).numpy(), 100):
```



0秒 # 当数据非常非常多的时候，正态分布会体现的非常明显
`plt.hist(torch.randn(10**6).numpy(), 100);`



11. 拼接张量

0秒 [27] # 创建两个 1x4 的tensor
`a = torch.Tensor([[1, 2, 3, 4]])`
`b = torch.Tensor([[5, 6, 7, 8]])`

在 0 方向拼接 (即在 Y 方各上拼接)，会得到 2x4 的矩阵
`print(torch.cat((a,b), 0))`

`tensor([[1., 2., 3., 4.],
[5., 6., 7., 8.]])`

0秒 # 在 1 方向拼接 (即在 X 方各上拼接)，会得到 1x8 的矩阵
`print(torch.cat((a,b), 1))`

`tensor([[1., 2., 3., 4., 5., 6., 7., 8.]])`

个人想法和解读：拼接张量，第二个参数选择拼接的维度，0为行，1为列。

2.2 pytorch基础练习

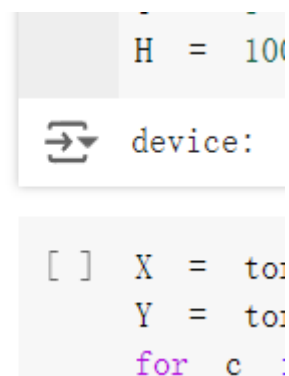
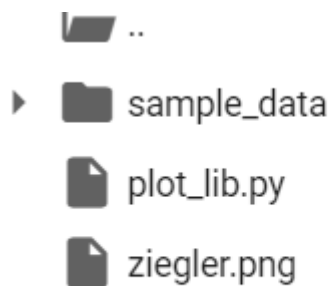
1. 加载plot_lib.py和图片

0秒 `!wget https://raw.githubusercontent.com/Atcold/pytorch-Deep-Learning/master/res/plot_lib.py`

--2024-09-27 11:17:27-- https://raw.githubusercontent.com/Atcold/pytorch-Deep-Learning/master/res/plot_lib.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4796 (4.7K) [text/plain]
Saving to: 'plot_lib.py'

plot_lib.py 100%[=====>] 4.68K --KB/s in 0s

2024-09-27 11:17:27 (46.6 MB/s) - 'plot_lib.py' saved [4796/4796]



2.准备工作

```
import random
import torch
from torch import nn, optim
import math
from IPython import display
from plot_lib import plot_data, plot_model, set_default

# 因为colab是支持GPU的, torch 将在 GPU 上运行
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('device: ', device)

# 初始化随机数种子。神经网络的参数都是随机初始化的,
# 不同的初始化参数往往会导致不同的结果, 当得到比较好的结果时我们通常希望这个结果是可以复现的,
# 因此, 在pytorch中, 通过设置随机数种子也可以达到这个目的
seed = 12345
random.seed(seed)
torch.manual_seed(seed)

N = 1000 # 每类样本的数量
D = 2 # 每个样本的特征维度
C = 3 # 样本的类别
H = 100 # 神经网络里隐层单元的数量

device: cpu
```

个人想法和解读：首先检测 GPU 是否可用，并将设备设为 GPU 或 CPU。同时，通过设定随机数种子确保实验结果可复现。定义了关键参数，包括每类样本数量 N、每个样本的特征维度 D、样本类别数 C，以及神经网络隐藏层的单元数 H，为后续的神经网络模型训练做好准备。

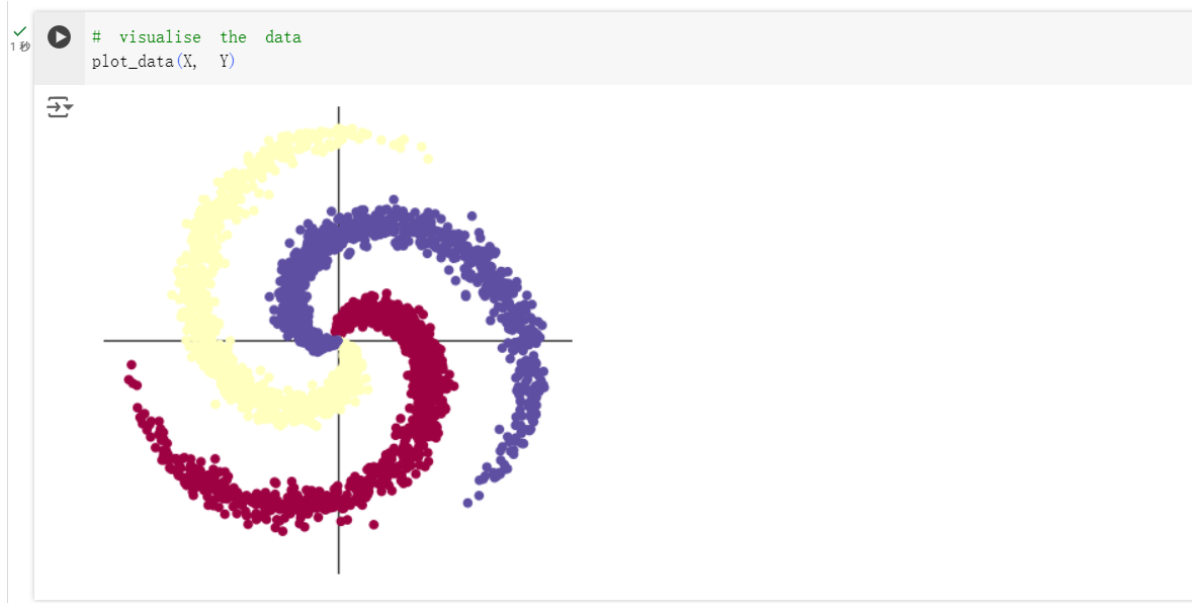
3.生成螺旋形样本集

```
X = torch.zeros(N * C, D).to(device)
Y = torch.zeros(N * C, dtype=torch.long).to(device)
for c in range(C):
    index = 0
    t = torch.linspace(0, 1, N) # 在[0, 1]间均匀的取10000个数, 赋给t
    # 下面的代码不用理解太多, 总之是根据公式计算出三类样本 (可以构成螺旋形)
    # torch.randn(N) 是得到 N 个均值为0, 方差为 1 的一组随机数, 注意要和 rand 区分开
    inner_var = torch.linspace((2*math.pi/C)*c, (2*math.pi/C)*(2+c), N) + torch.randn(N) * 0.2

    # 每个样本的(x,y)坐标都保存在 X 里
    # Y 里存储的是样本的类别, 分别为 [0, 1, 2]
    for ix in range(N * c, N * (c + 1)):
        X[ix] = t[index] * torch.FloatTensor((math.sin(inner_var[index]), math.cos(inner_var[index])))
        Y[ix] = c
        index += 1

print("Shapes:")
print("X:", X.size())
print("Y:", Y.size())

Shapes:
X: torch.Size([3000, 2])
Y: torch.Size([3000])
```



个人想法和解读：生成螺旋形样本集，为多类别分类任务构建训练数据。并画出数据集图像。

1. 初始化数据:

```
X = torch.zeros(N * C, D).to(device) # 创建 (N*C, D) 大小的零矩阵存储样本特征，  
N*C 为总样本数，D 是特征维度。  
Y = torch.zeros(N * C, dtype=torch.long).to(device) # 创建 (N*C) 大小的零矩阵存  
储样本标签，类别为 long 型整数。
```

2. 生成螺旋形数据:

- `torch.linspace(0, 1, N)` 生成 0 到 1 间的 `N` 个等间隔点，用于控制样本分布。
- 通过 `inner_var` 生成随机扰动的角度值，以形成螺旋形轨迹。

3. 样本分类:

- `for c in range(C):` 循环生成 `C` 类数据。
- 每类样本的 `x`、`y` 坐标按照螺旋公式计算并存入 `x`，对应的类别存入 `Y`。

最终，`x` 包含所有样本的特征数据，`Y` 保存了相应的类别标签。螺旋数据结构适合用于测试神经网络的分类能力。

4. 训练模型

```
[6] learning_rate = 1e-3
    lambda_l2 = 1e-5

    # nn 包用来创建线性模型
    # 每一个线性模型都包含 weight 和 bias
    model = nn.Sequential(
        nn.Linear(D, H),
        nn.Linear(H, C)
    )
    model.to(device) # 把模型放到GPU上

    # nn 包含多种不同的损失函数, 这里使用的是交叉熵 (cross entropy loss) 损失函数
    criterion = torch.nn.CrossEntropyLoss()

    # 这里使用 optim 包进行随机梯度下降 (stochastic gradient descent) 优化
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=lambda_l2)

    # 开始训练
    for t in range(1000):
        # 把数据输入模型, 得到预测结果
        y_pred = model(X)
        # 计算损失和准确率
        loss = criterion(y_pred, Y)
        score, predicted = torch.max(y_pred, 1)
        acc = (Y == predicted).sum().float() / len(Y)
        print(' [EPOCH]: %i, [LOSS]: %.6f, [ACCURACY]: %.3f' % (t, loss.item(), acc))
        display.clear_output(wait=True)

        # 反向传播前把梯度置 0
        optimizer.zero_grad()
        # 反向传播优化
        loss.backward()
        # 更新全部参数
        optimizer.step()
```

[EPOCH]: 999, [LOSS]: 0.861541, [ACCURACY]: 0.504

个人想法和解读:

超参数设置:

```
learning_rate = 1e-3
lambda_l2 = 1e-5
```

- 设置学习率 `learning_rate` 和 L2 正则化参数 `lambda_l2`, 用于梯度更新和防止过拟合。

构建模型:

```
model = nn.Sequential(
    nn.Linear(D, H),
    nn.Linear(H, C)
)
```

- 使用

```
nn.Sequential
```

创建了一个简单的两层线性神经网络:

- 第一层将输入特征 `D` (2 维) 映射到隐藏层 `H` (100 维)。
- 第二层将隐藏层映射到输出类别 `C` (3 类)。

损失函数和优化器:

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
weight_decay=lambda_l2)
```

- 使用交叉熵损失函数 (`CrossEntropyLoss`) 来计算预测和真实标签之间的误差。
- 使用随机梯度下降 (SGD) 作为优化器, 并添加 L2 正则化 (`weight_decay`) 来防止过拟合。

训练过程:

```
for t in range(1000):
    y_pred = model(x)
    loss = criterion(y_pred, Y)
    score, predicted = torch.max(y_pred, 1)
    acc = (Y == predicted).sum().float() / len(Y)
    print('[EPOCH]: %i, [LOSS]: %.6f, [ACCURACY]: %.3f' % (t, loss.item(), acc))
    display.clear_output(wait=True)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

- 通过循环进行 1000 个 epoch 的训练：
 - 将输入数据 `x` 输入模型，得到预测结果 `y_pred`。
 - 使用交叉熵损失函数计算损失 `loss`。
 - 使用 `torch.max` 得到预测的类别，计算当前训练集上的准确率 `acc`。
 - 打印每个 epoch 的损失和准确率。
 - 通过反向传播（`loss.backward()`）计算梯度，并使用优化器（`optimizer.step()`）更新模型参数。

在每个 epoch 后，清除输出（`display.clear_output(wait=True)`）以实时显示训练过程中的损失和准确率。

5.画图展示结果:

```

✓ [7] print(y_pred.shape)
0秒 print(y_pred[10, :])
print(score[10])
print(predicted[10])

⇨ torch.Size([3000, 3])
tensor([-0.2245, -0.2594, -0.2080], grad_fn=<SliceBackward0>)
tensor(-0.2080, grad_fn=<SelectBackward0>)
tensor(2)

```

```

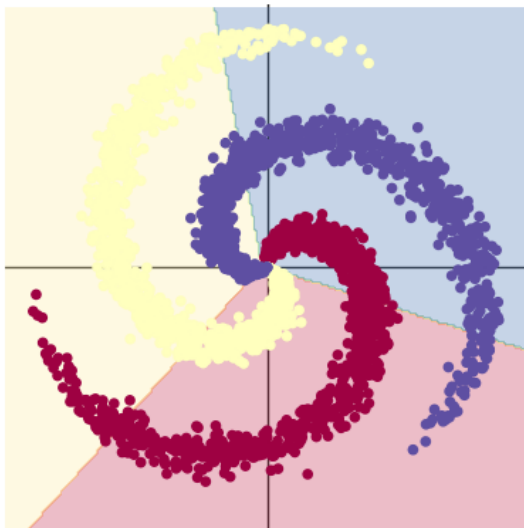
✓ # Plot trained model
1秒 print(model)
plot_model(X, Y, model)

```

```

⇨ Sequential(
  (0): Linear(in_features=2, out_features=100, bias=True)
  (1): Linear(in_features=100, out_features=3, bias=True)
)

```



6.更新版本的神经网络模型训练过程，区别在于使用了 ReLU 激活函数和 Adam 优化器

```

✓ learning_rate = 1e-3
13秒 lambda_l2 = 1e-5

# 这里可以看到，和上面模型不同的是，在两层之间加入了一个 ReLU 激活函数
model = nn.Sequential(
    nn.Linear(D, H),
    nn.ReLU(),
    nn.Linear(H, C)
)
model.to(device)

# 下面的代码和之前是完全一样的，这里不过多叙述
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=lambda_l2) # built-in L2

# 训练模型，和之前的代码是完全一样的
for t in range(1000):
    y_pred = model(X)
    loss = criterion(y_pred, Y)
    score, predicted = torch.max(y_pred, 1)
    acc = ((Y == predicted).sum().float() / len(Y))
    print("[EPOCH]: %i, [LOSS]: %.6f, [ACCURACY]: %.3f" % (t, loss.item(), acc))
    display.clear_output(wait=True)

    # zero the gradients before running the backward pass.
    optimizer.zero_grad()
    # Backward pass to compute the gradient
    loss.backward()
    # Update params
    optimizer.step()

```

```

⇨ [EPOCH]: 999, [LOSS]: 0.178407, [ACCURACY]: 0.949

```

个人想法和解读：

1. 模型结构：

```

python复制代码model = nn.Sequential(
    nn.Linear(D, H),
    nn.ReLU(),
    nn.Linear(H, C)
)

```

- 在两层线性层之间加入了 **ReLU** 激活函数 (`nn.ReLU()`)。ReLU 是一种常用的激活函数，可以引入非线性，帮助网络更好地学习复杂的模式。

2. 优化器:

python

复制代码

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=lambda_12)
```

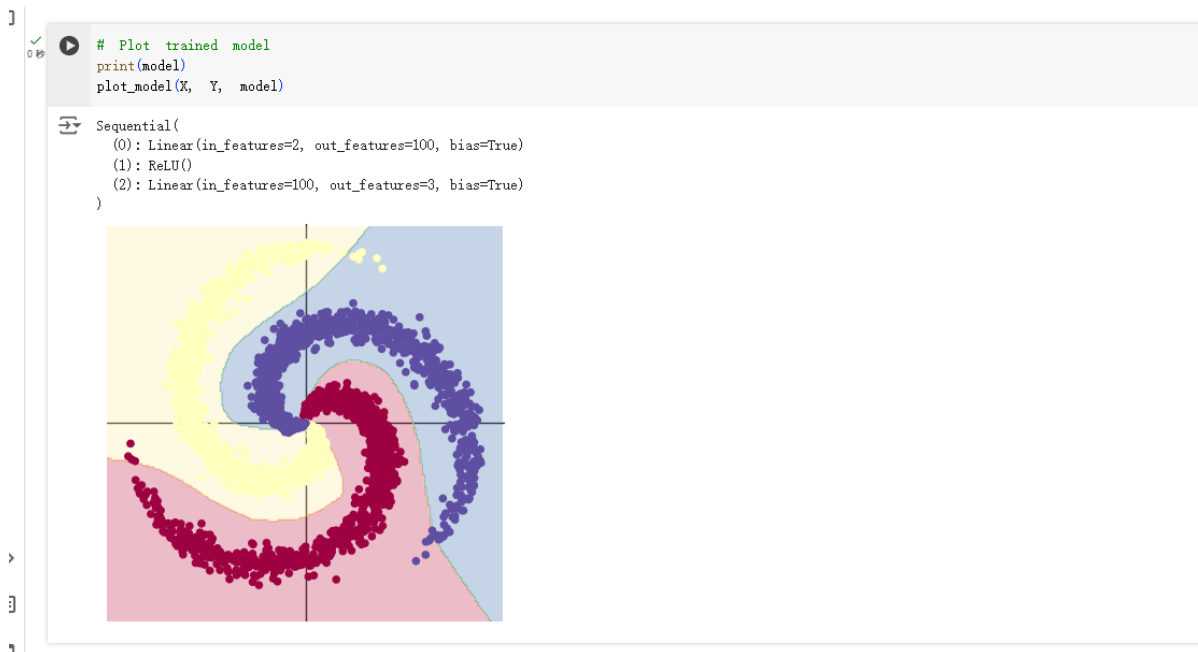
- 采用 **Adam 优化器** (`torch.optim.Adam`)，它通常比 SGD 收敛得更快，因为它自适应调整学习率。这使得训练过程更加高效，并且与之前一样添加了 L2 正则化。

3. 训练过程:

- 训练流程与之前的代码基本相同：
 - 前向传播：计算预测 `y_pred`。
 - 计算损失：使用交叉熵损失函数。
 - 计算准确率：根据预测结果与真实标签比较。
 - 反向传播：通过 `loss.backward()` 计算梯度。
 - 更新参数：通过 `optimizer.step()` 更新模型权重。
 - 每个 epoch 的损失和准确率都会打印出来。

通过加入 ReLU 激活函数和 Adam 优化器，这个模型可能在复杂数据上收敛得更快，并能学习到更复杂的特征。

7. 训练结果



可见，数据分类更加精准。

【第二部分：问题总结】

1、AlexNet有哪些特点？为什么可以比LeNet取得更好的性能？

特点：

- (1) 具有5层卷积层，3层全连接层。能够学习到更加复杂和抽象的特征。
- (2) 使用了 ReLU (Rectified Linear Unit) 作为激活函数，代替了传统的 sigmoid 和 tanh 激活函数。ReLU 函数具有计算简单、收敛速度快的特点，同时缓解了梯度消失问题。
- (3) 引入了 Dropout 技术来随机丢弃一部分神经元，以防止过拟合。
- (4) 通过数据增强（如图像裁剪、翻转、旋转等）来增加训练数据的多样性，进一步提升模型的泛化性能。
- (5) AlexNet 是基于 ImageNet 数据集进行训练的，这是一个包含 100 万张图像的大规模数据集。AlexNet 通过在如此大规模的数据上进行训练，显著提高了其性能。

更好的性能：

- (1) **深度网络**：LeNet 只有 2 个卷积层和 2 个全连接层，模型相对较浅，适用于简单的数据集如 MNIST。AlexNet 的 8 层深度网络使其能够提取更加复杂的特征，适应更大的数据集如 ImageNet。
- (2) **大规模数据集训练**：LeNet 主要用于小型数据集如 MNIST，而 AlexNet 是为 ImageNet 这样的百万级数据集设计的，通过大规模数据的训练，AlexNet 能够学习到更多通用且复杂的特征。
- (3) **ReLU 激活函数**：LeNet 使用的是 sigmoid 或 tanh 激活函数，这些函数在深度网络中容易出现梯度消失问题。而 AlexNet 使用 ReLU 激活函数，解决了梯度消失问题，并且使网络收敛速度更快。
- (4) **更强的正则化**：AlexNet 引入了 Dropout 和数据增强等正则化技术，极大地提高了模型的泛化能力。LeNet 没有这些现代正则化方法，在复杂数据上容易过拟合。
- (5) **并行计算能力**：AlexNet 使用了 GPU 并行计算，使得它能够处理比 LeNet 更大的模型和更多的参数，同时也显著缩短了训练时间。

2、激活函数有哪些作用？

- (1) 引入非线性，使神经网络能够处理复杂的非线性问题；
- (2) 帮助网络学习复杂的特征和模式，提升表达能力；
- (3) 控制输出范围，防止网络输出过大或过小，进而避免梯度爆炸或消失的问题；
- (4) 激活函数还能加速训练过程，因为它们计算简单且能有效缓解梯度消失。

3、梯度消失现象是什么？

梯度消失现象是指在深层神经网络中，反向传播时，随着层数的增加，梯度逐渐变得非常小，导致靠近输入层的权重更新几乎停滞。这是由于激活函数的导数在输入较大或较小时趋近于 0，导致梯度在反向传播中逐层相乘时逐渐缩小。当梯度消失时，模型无法有效训练，尤其在深层网络中，导致模型性能下降或无法收敛。

4、神经网络是更宽好还是更深好？

更宽的网络：适合处理相对简单的问题，能够增加每层的表达能力，快速捕捉全局信息。宽度的增加有时能提升模型性能，但很快会遇到容量限制，导致网络学习不到更复杂的特征。

更深的网络：适合学习复杂的层级特征，能够逐步抽象数据的高级模式。深层网络有更强的表示能力，尤其对于视觉任务和序列任务更为有效。然而，深度网络容易出现梯度消失和过拟合问题，需要通过技术如残差连接 (ResNet) 等解决。

所以，更深的网络更容易在实际应用中。

5、为什么要使用Softmax?

Softmax 将输出层的每个值通过指数函数处理，并归一化为 0 到 1 之间的概率，且所有类别的概率和为 1。这样，网络的输出可以被解释为每个类别的预测概率。

6、SGD 和 Adam 哪个更有效?

SGD (随机梯度下降) :

优点: 实现简单，适合大规模数据集，泛化能力好（在一些任务中容易找到更好的全局最优解）。

缺点: 更新过程中，学习率是固定的，调参较难。收敛速度较慢，尤其是在复杂或稀疏梯度的情况下。

Adam (自适应矩估计) :

优点: 融合了动量和自适应学习率策略，对每个参数单独调整学习率，收敛速度较快，适合稀疏梯度或高噪声数据。常常在很多任务中表现更好。

缺点: 在某些任务中泛化能力不如 SGD，有时会过拟合，并且需要更多内存。

所以，在大多数实际应用中，尤其是当数据较复杂、训练时间较短时，Adam 表现更优；在需要精细调参、对模型的泛化能力要求更高的情况下，SGD 通常表现更好。Adam 在大多数情况下收敛更快，而 SGD 在优化到全局最优时可能更稳健。

其他问题: 在训练神经网络时，使用批量归一化和不使用批量归一化，哪种方法更有效?

批量归一化通过标准化每层的输入来加速训练过程，使网络能够更快收敛，并允许使用更大的学习率，从而缓解了深层网络中的梯度消失问题。此外，BN 在保持激活值的均值和方差稳定的同时，有助于提高模型的泛化能力，减少过拟合的风险。然而，批量归一化也增加了计算成本，因为每个小批量都需要计算均值和方差，并且对小批量大小较为敏感。尽管不使用批量归一化可以简化模型结构，并在某些简单任务中表现良好，但通常会导致训练速度变慢，特别是在深层网络和复杂任务中。因此，绝大多数情况下，建议使用批量归一化以提高训练效率和模型性能。

二、问题总结与体会

在代码练习部分，我通过 Google Colab 平台完成了 PyTorch 的基础操作和螺旋数据分类的实验。这些实践让我体验到了深度学习框架的强大以及云端运行的便利。在实现神经网络进行简单数据分类的过程中，我直观地感受到了模型训练与预测的过程，以及如何利用激活函数和损失函数优化模型性能。

在对一系列问题的思考中，我更深入地理解了神经网络的设计和 optimization 原则。例如，AlexNet 相较于 LeNet 在性能上的提升，得益于其更深的网络结构和更有效的特征提取能力；激活函数在引入非线性、控制输出范围等方面的重要性。这些问题的讨论使我对深度学习的理论背景和应用场景有了更全面的认识。