

华中师范大学《高等程序设计》

期末考试试卷样卷

学号:

线

学生姓名:

年级:

专业:

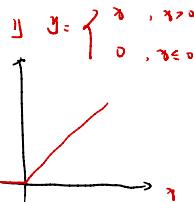
院(系)

题型	选择题	简答题	程序片段	程序设计题	总分
分值	10	24	40	26	100
得分					

得分	评阅人

一、选择题 (共 10 题, 每小题 1 分, 共 10 分)

- $\text{ReLU}(x) = \max(0, x)$, 解决 sigmoid, tanh 的梯度消失
1. 在 PyTorch 中, ReLU 激活函数的主要优点是什么? 防止梯度消失
- A. 引入稀疏性
 - B. 减少计算复杂度(正确)
 - C. 提高模型的非线性所有激活
 - D. 防止梯度消失(正确) 梯度消失为 0
2. 在 PyTorch 中, 如何将一个张量的维度扩展为特定的大小? B
- A. tensor.reshape(size)
 - B. tensor.expand(size)
 - C. tensor.repeat(size)
 - D. tensor.view(size)
3. 在 PyTorch 中, 如何创建一个梯度跟踪的张量? A
- A. torch.tensor(data, requires_grad=True)
 - B. torch.autograd.tensor(data)
 - C. torch.gradient.tensor(data)
 - D. torch.requires_grad(data)
4. 在 PyTorch 中, 卷积操作的主要作用是什么? C
- A. 增加输入数据的维度
 - B. 归一化输入数据(Batch Norm 基层)
 - C. 提取输入数据的局部特征(正确)
 - D. 减少输入数据的维度
5. 在 PyTorch 中, 如何对一个张量进行逐元素对数操作? A
- A. torch.log(tensor)
 - B. tensor.log() (也对)
 - C. torch.logarithm(tensor)
 - D. tensor.logarithm()
6. 什么是随机梯度下降 (SGD) 中的学习率? A
- A. 用于控制每次参数更新的步长(正确)



$x = \text{torch.tensor([1, 2, 3])}$
 $x.requires_grad_()$

↑ 建立梯度追踪

$$\theta = \theta - \eta \cdot \nabla L(\theta)$$

↑ 学习率

- B. 用于评估模型性能 ~~损失函数，准确率~~
 C. 用于增加模型复杂度
D. 用于减少训练数据 ~~数据采样~~

(D) 7. 什么是随机森林算法的主要优点?

- A. 增加模型的复杂度
B. 减少训练数据的需求
 C. 提高模型的训练速度 ~~慢于单个~~
D. 提高模型的稳定性和准确性

~~通过 Boot strap 采样构建多个决策树~~
~~最终通过投票（多数）或平均（回归）输出~~

(C) 8. 在 PyTorch 中, 如何将一个张量从 CPU 转移到 GPU?

- A. tensor.cuda()
B. torch.gpu()
 C. tensor.to('cuda')
D. torch.cuda(tensor)

```
device = torch.device('cuda' if
torch.cuda.is_available()
else 'cpu')
x = x.to(device)
```

(C) 9. 在 PyTorch 中, 如何对一系列张量沿新维度进行拼接操作? ~~at~~ ~~2个(2,3) → dim=0 (4,3)~~ ~~dim=1 (2,6)~~

- A. torch.cat([tensor1, tensor2], dim)
B. torch.concat([tensor1, tensor2], dim)
C. torch.stack([tensor1, tensor2], dim) ~~维数+1~~ ~~2个(3,4) 张量~~
D. torch.join([tensor1, tensor2], dim)

~~dim=0 → (2, 3, 4)~~
~~dim=1 → (3, 2, 4)~~

(A) 10. 在 PyTorch 中, 如何创建一个梯度跟踪的张量?

- A. torch.tensor(data, requires_grad=True)
B. torch.create_tensor(data).requires_grad_()
C. torch.create_tensor(data, requires_grad=True)
D. torch.tensor(data).grad()

得分	评阅人

二、简答题 (共 8 小题, 每小题 3 分, 共 24 分)

1. PyTorch 中的过拟合 (Overfitting) 是什么? 如何防止过拟合?

~~模型在训练中表现良好~~
~~但是在测试集中表现糟糕~~

~~训练时样本过多, 特征值过多导致的~~
~~所以增加训练样本, 减少不必要的特征值~~

1. 过拟合定义: 模型在训练数据上表现极好 (训练误差小), 但在未见过的测试 / 验证数据上表现差 (泛化误差大), 本质是模型学习了训练数据中的噪声而非核心规律。

2. 防止方法 (任答 3 种及以上):

- 数据增强: 对训练数据进行随机变换 (如图片翻转、裁剪、旋转), 增加数据多样性。
- 正则化: L1 正则化 (权重绝对值惩罚)、L2 正则化 (权重平方惩罚, PyTorch 中通过 weight_decay 参数实现)。
- Dropout: 训练时随机丢弃部分神经元, 防止过度依赖特定神经元 (`torch.nn.Dropout`)。
- 早停 (Early Stopping): 监控验证集性能, 当性能不再提升时停止训练, 避免过度训练。
- 模型简化: 减少网络层数、神经元个数, 降低模型复杂度。
- 迁移学习: 使用预训练模型, 冻结部分层参数, 仅微调顶层。

2. 在 PyTorch 中, 如何使用预训练模型进行迁移学习? 请用代码展示如何加载一个预训练的 ResNet 模型并在新数据上进行微调。

1. 迁移学习定义: 将预训练模型(在大规模数据集上训练好的模型)的知识迁移到新的小规模数据集中, 核心是“复用特征提取器, 微调分类头”。

2. 代码实现:

```
python ^

import torch
import torchvision.models as models
import torch.nn as nn

# 加载预训练ResNet-18模型, 冻结特征提取层参数
resnet = models.resnet18(pretrained=True) # 加载预训练权重
for param in resnet.parameters():
    param.requires_grad = False # 冻结所有层(特征提取部分)

# 替换顶层分类头(适配新任务的类别数, 假设新任务为5类)
num_classes = 5
resnet.fc = nn.Linear(resnet.fc.in_features, num_classes) # 新分类头, 仅该层可训练

# 定义优化器(仅优化分类头参数)
optimizer = torch.optim.SGD(resnet.fc.parameters(), lr=0.01, momentum=0.9)
```

3. 在 PyTorch 中, 什么是参数优化 (optimizer)? 请用代码展示如何定义一个 SGD 优化器并使用它更新模型参数。

1. 优化器定义: 优化器是实现梯度下降算法的工具, 用于根据反向传播计算的梯度更新模型参数, 最小化损失函数, 核心是“根据梯度调整参数步长”。

2. 代码实现:

```
python ^

import torch
import torch.nn as nn

# 定义简单模型(示例)
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(10, 2)
    def forward(self, x):
        return self.linear(x)

model = SimpleModel() # 初始化模型
criterion = nn.CrossEntropyLoss() # 定义损失函数
# 定义SGD优化器(学习率0.01, 动量0.9)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# 训练循环中的参数更新步骤
inputs = torch.randn(32, 10) # 模拟输入数据(batch_size=32, 特征数=10)
labels = torch.randint(0, 2, (32,)) # 模拟标签

optimizer.zero_grad() # 梯度清零(避免累积)
outputs = model(inputs) # 前向传播
loss = criterion(outputs, labels) # 计算损失
loss.backward() # 反向传播(计算梯度)
optimizer.step() # 更新参数
```

4. 在 PyTorch 中, 什么是梯度裁剪 (gradient clipping)? 它有什么作用? 请解释并用代码演示如何进行梯度裁剪。

1. 梯度裁剪定义: 对反向传播计算出的梯度进行数值限制, 将梯度的 L2 范数(或最大值)限制在预设阈值内, 避免梯度爆炸。

2. 作用: 解决深层网络(尤其是 RNN、LSTM)训练中因梯度累积导致的梯度爆炸问题, 使训练过程稳定。

```
import torch
import torch.nn as nn

# 定义简单RNN模型(易出现梯度爆炸)
model = nn.RNN(input_size=10, hidden_size=20, num_layers=2)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# 训练循环中的梯度裁剪步骤
inputs = torch.randn(5, 32, 10) # 序列长度5, batch_size=32, 输入维度10
labels = torch.randn(5, 32, 20) # 模拟标签

optimizer.zero_grad()
outputs, _ = model(inputs)
loss = criterion(outputs, labels)
loss.backward() # 反向传播计算梯度

# 梯度裁剪, 限制所有参数的梯度L2范数不超过1.0
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

optimizer.step() # 更新参数
```

5. 在 PyTorch 中, 如何进行模型评估? 请解释并用代码展示如何评估模型在验证集上的性能。

- 模型评估定义：在训练完成后，使用验证集 / 测试集测试模型的泛化能力，核心是关闭梯度计算（节省内存、避免参数更新），计算评估指标（如准确率、损失）。
- 核心步骤：设置模型为评估模式 (`model.eval()`) → 关闭梯度计算 (`torch.no_grad()`) → 遍历验证集→ 计算指标。

```

import torch
import torch.nn as nn
from torch.utils.data import DataLoader

# 假设已定义模型、验证数据集val_dataset
model = SimpleModel() # 复用第3题的SimpleModel
model.load_state_dict(torch.load('model.pth')) # 加载训练好的权重
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
criterion = nn.CrossEntropyLoss()

model.eval() # 设置为评估模式（关闭Dropout、BatchNorm的训练模式）
total_loss = 0.0
correct = 0
total = 0

with torch.no_grad(): # 关闭梯度计算
    for inputs, labels in val_loader:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        total_loss += loss.item() * inputs.size(0) # 累积损失

        # 计算准确率（分类任务）
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# 计算平均损失和准确率
avg_loss = total_loss / total
accuracy = 100 * correct / total
print(f'验证集平均损失: {avg_loss:.4f}, 准确率: {accuracy:.2f}%')

```

6. 在 PyTorch 中，什么是循环神经网络 (RNN)？它有什么作用？请解释并用代码展示如何定义和使用一个简单的 RNN。

- RNN 定义：一种处理序列数据的神经网络，通过隐藏状态 (hidden state) 保存历史信息，核心是“当前输出依赖于当前输入和历史输入”，结构为 $h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$ 。
- 作用：处理时序数据（如文本、语音、时间序列），捕捉序列中的依赖关系（如文本中的上下文语义、时间序列的趋势）。

```

import torch
import torch.nn as nn

# 定义简单RNN模型(输入维度10, 隐层维度20, 1层)
class SimpleRNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(input_size=10, hidden_size=20, num_layers=1, batch_first=True)
        self.fc = nn.Linear(20, 2) # 分类头(2类)

    def forward(self, x):
        # x: (batch_size, seq_len, input_size) = (32, 5, 10)
        out, hidden = self.rnn(x) # out: (32, 5, 20), hidden: (1, 32, 20)
        # 取最后一个时间步的输出用于分类
        out = self.fc(out[:, -1, :]) # (32, 2)
        return out

# 实例化模型并测试
model = SimpleRNN()
inputs = torch.randn(32, 5, 10) # 32个样本, 每个样本序列长度5, 输入维度10
outputs = model(inputs)
print(f'输出形状: {outputs.shape}') # 输出: torch.Size([32, 2])

```

7. 什么是自注意力机制 (Self-Attention)？它在 Transformer 中的作用是什么？

- 自注意力机制定义：一种允许序列中每个位置关注自身及其他位置信息的机制，通过计算每个位置与所有位置的相似度（注意力权重），加权求和得到该位置的上下文表示，公式为：

- $Q = XW_Q$ (查询矩阵)、 $K = XW_K$ (键矩阵)、 $V = XW_V$ (值矩阵)

- 注意力权重： $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$

- 在 Transformer 中的作用：

- 捕捉长距离依赖：不受序列长度限制，可直接计算任意两个位置的依赖关系（优于 RNN 的顺序处理）。
- 并行计算：无需像 RNN 那样逐时间步处理，提高训练效率。
- 自适应关注：自动学习序列中重要的位置信息，赋予更高权重。

8. 什么是深度学习中的反向传播算法？

1. 反向传播算法 (Backpropagation) 定义：一种高效计算神经网络损失函数对各参数梯度的算法，基于链式法则，从输出层反向遍历网络，逐层计算梯度。
 2. 核心步骤：
 - 前向传播：输入数据通过网络计算输出值，记录各层的中间结果。
 - 计算损失：根据输出值与真实标签计算损失函数（如 MSE、CrossEntropyLoss）。
 - 反向传播：从输出层开始，计算损失函数对各层参数的梯度（链式法则： $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h} \cdot \frac{\partial h}{\partial w}$ ）。
 - 参数更新：使用梯度下降优化器更新参数 ($\theta = \theta - \eta \cdot \nabla L$)。
 3. 意义：是深度学习模型训练的核心，解决了深层网络梯度计算的效率问题，使大规模神经网络的训练成为可能。
-

得分	评阅人

三、程序片段 (共 20 题，每小题 2 分，共 40 分)

根据要求，编写代码片段 (1-3 行)。例如：

题目：请导入 torch 包

答案：import torch

1. 创建一个形状为(3, 3)的随机张量，并计算其均值

```
import torch
x = torch.randn(3, 3)
print(x.mean)      / 按行先 print(x.mean(dim=1))  
                   约
```

2. 创建一个形状为(2, 3)的全1张量，并计算其L2范数

```
import torch
x = torch.ones(2, 3)
print(torch.norm(x, p=2))  
                   L2  
                   ↓
```

3. 定义一个 Adam 优化器，学习率为 0.001

```
import torch
import torch.nn as nn
model = nn.Linear(10, 5)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

4. 加载一个预训练的 ResNet-18 模型

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

`nn.MaxPool2d(2, stride=2)`

`padding=1)`

5. 创建一个形状为(3, 3)的随机张量，并进行 2x2 的最大池化操作

```
import torch
import torch.nn as nn
x = torch.randn(3, 3).unsqueeze(0).unsqueeze(0)
pool = nn.MaxPool2d(2)      print(pool(x).shape)
```

6. 使用 DataLoader 加载一个数据集 dataset，批量大小为 4

```
from torch.utils.data import DataLoader
dataLoader = DataLoader(dataset=dataset, batch_size=4, shuffle=True,
                        num_workers=4)
```

7. 定义一个包含 64 个单元的 LSTM 层

```
python ^ ① 运行 ② ③ ④ ⑤ ⑥ ⑦
import torch.nn as nn
lstm = nn.LSTM(input_size=10, hidden_size=64, num_layers=1, batch_first=True)
```

- 解析：LSTM 层的核心参数 `input_size`（输入维度）、`hidden_size`（隐藏层单元数）、`batch_first`（是否 batch 维度在前）；核心考点：LSTM 层定义 API。

8. 创建一个形状为(3, 224, 224)的随机张量，并通过 ResNet-18 模型进行前向传播

```
python ^ ① 运行 ② ③ ④ ⑤ ⑥ ⑦
import torch
import torchvision.models as models
model = models.resnet18(pretrained=True)
x = torch.randn(3, 224, 224).unsqueeze(0) # 扩展为(batch, channel, H, W)
output = model(x)
```

- 解析：ResNet-18 要求输入形状为 (batch, 3, 224, 224)，需添加 batch 维度；核心考点：模型输入形状适配，前向传播。
- 举一反三：若要求“冻结模型特征层，仅训练分类头”，则添加 `for param in model.parameters(): param.requires_grad = False.`

9. 创建一个形状为(4, 4)的全 0 张量，并设置 `requires_grad` 为 True

```
import torch
x = torch.zeros(4, 4, requires_grad=True)
```

↑
启用梯度跟踪

10. 创建一个随机张量，并将其移动到 GPU 设备（如果可用）

```
import torch
x = torch.randn(3, 3)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
x = x.to(device)
```

model = model.to(device)

11. 使用 `torch.optim` 创建一个学习率为 0.01 的 SGD 优化器

```
import torch
import torch.nn as nn
model = nn.Linear(5, 2)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

- 解析：SGD 优化器需传入模型参数和学习率；核心考点：优化器定义 API。
- 举一反三：若要求“添加动量 0.9 和权重衰减 0.001”，则修改为 `optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=0.001)`。

12. 定义一个全连接层，并初始化其权重为均值为 0、方差为 0.01 的正态分布

```
import torch
import torch.nn as nn
linear = nn.Linear(10, 5)
nn.init.normal_(linear.weight, mean=0.0, std=0.01)
```

- 解析：`nn.init.normal_()` 用于权重初始化，`linear.weight` 是全连接层的权重参数；核心考点：权重初始化 API。
- 举一反三：若要求“初始化偏置为 0”，则添加 `nn.init.constant_(linear.bias, 0.0)`。

13. 创建一个形状为(3, 3)的全零张量

```
import torch
```

```
torch.zeros(3, 3)
```

14. 定义一个 ReLU 激活函数

```
import torch.nn as nn
```

```
relu = nn.ReLU()
```

} Sigmoid

} LeakyReLU

15. 创建一个 Adam 优化器

```
python ^
```

```
import torch
import torch.nn as nn
model = nn.Linear(8, 3)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

- 解析：Adam 优化器的默认学习率为 0.001，需传入模型参数；核心考点：优化器定义 API。

- 举一反三：若要求“自定义 betas 参数”，则修改为 `optimizer = torch.optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999))`。

16. 加载一个预训练的 VGG16 模型

```
python ^
```

```
import torchvision.models as models
vgg16 = models.vgg16(pretrained=True)
```

- 解析：`torchvision.models.vgg16()` 提供 VGG16 模型，`pretrained=True` 加载预训练权重；核心考点：预训练模型加载 API。

- 举一反三：若要求“不加载预训练权重”，则修改为 `vgg16 = models.vgg16(pretrained=False)`。

17. 创建一个形状为(4, 4)的全 1 张量

18. 定义一个线性层，输入大小为 10，输出大小为 5

```
python ^
```

```
import torch.nn as nn
linear = nn.Linear(in_features=10, out_features=5)
```

- 解析：`nn.Linear` 的核心参数 `in_features`（输入维度）、`out_features`（输出维度）；核心考点：全连接层定义 API。

- 举一反三：若要求“定义一个输入 20、输出 8 的线性层”，则修改为 `linear = nn.Linear(20, 8)`。

19. 请对张量 x 进行反向传播

```
loss = criterion(outputs, labels) # 假设已计算损失
loss.backward()
```

- 解析：反向传播需基于损失张量调用 `backward()` 方法，自动计算梯度；核心考点：反向传播 API。

- 举一反三：若要求“计算特定张量的梯度”，则修改为 `loss.backward(retain_graph=True)`（保留计算图，用于多次反向传播）。

20. 请将张量 x 移动到 GPU 设备

```
import torch
x = torch.randn(2, 2)

x = x.cuda()
```

	得分	评阅人	<p style="margin: 0;">四、程序设计题（共3题，第1、2小题各8分，第3小题10分，共26分）</p> <p>1. 请使用 PyTorch 定义一个全连接神经网络模型类，该网络包含一个输入层、两个隐藏层和一个输出层。输入层有 12 个节点，第一个隐藏层有 24 个节点，第二个隐藏层有 12 个节点，输出层有 6 个节点。请使用 Tanh 作为激活函数。（8 分）</p> <pre style="font-family: monospace; margin-top: 20px;"> 1 import torch 2 import torch.nn as nn 3 4 class FullyConnectedNet(nn.Module): 5 def __init__(self): 6 super(FullyConnectedNet, self).__init__() 7 # 定义网络层: 输入层(12)→隐藏层1(24)→隐藏层2(12)→输出层(6) 8 self.fc1 = nn.Linear(12, 24) # 输入层→第一个隐藏层 9 self.fc2 = nn.Linear(24, 12) # 第一个隐藏层→第二个隐藏层 10 self.fc3 = nn.Linear(12, 6) # 第二个隐藏层→输出层 11 self.tanh = nn.Tanh() # Tanh激活函数 12 13 def forward(self, x): 14 # 前向传播: 输入→激活→隐藏层1→激活→隐藏层2→输出层 15 x = self.tanh(self.fc1(x)) # 第一个隐藏层+Tanh激活 16 x = self.tanh(self.fc2(x)) # 第二个隐藏层+Tanh激活 17 x = self.fc3(x) # 输出层(无激活, 分类任务后续可加Softmax) 18 return x 19 20 # 测试模型 21 model = FullyConnectedNet() 22 input_tensor = torch.randn(32, 12) # batch_size=32, 输入维度12 23 output = model(input_tensor) 24 print(f'模型输出形状: {output.shape}') # 输出: torch.Size([32, 6]) 25 </pre>
--	----	-----	---

线
书
密

2. 请使用 PyTorch 定义一个卷积神经网络模型类，该网络包含三个卷积层和两个全连接层。输入为单通道的灰度图片，第一个卷积层有 8 个 3×3 的卷积核，第二个卷积层有 16 个 3×3 的卷积核，第三个卷积层有 32 个 3×3 的卷积核。第一个全连接层有 128 个节点，第二个全连接层有 10 个节点。请使用 ReLU 作为激活函数。（8 分）

```
1 import torch
2 import torch.nn as nn
3
4 class ConvNet(nn.Module):
5     def __init__(self):
6         super(ConvNet, self).__init__()
7         # 卷积层部分: 输入(1, H, W)→Conv1→Conv2→Conv3
8         self.conv_layers = nn.Sequential(
9             # 卷积层1: 1→8个 $3 \times 3$ 卷积核, padding=1(保持尺寸), ReLU激活
10            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, padding=1),
11            nn.ReLU(),
12            # 卷积层2: 8→16个 $3 \times 3$ 卷积核, padding=1, ReLU激活
13            nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1),
14            nn.ReLU(),
15            # 卷积层3: 16→32个 $3 \times 3$ 卷积核, padding=1, ReLU激活
16            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1),
17            nn.ReLU(),
18            nn.AdaptiveAvgPool2d((7, 7)) # 自适应池化, 确保输出尺寸为(32, 7, 7), 适配全连接层
19        )
20
21         # 全连接层部分: 32*7*7→128→10
22         self.fc_layers = nn.Sequential(
23             nn.Linear(32 * 7 * 7, 128), # 卷积输出展平后输入全连接层1
24             nn.ReLU(),
25             nn.Linear(128, 10) # 全连接层2(输出10类)
26         )
27
28     def forward(self, x):
29         # 前向传播: 卷积层→展平→全连接层
30         x = self.conv_layers(x) # 卷积层特征提取: (batch, 1, H, W)→(batch, 32, 7, 7)
31         x = x.view(x.size(0), -1) # 展平: (batch, 32, 7, 7)→(batch, 32*7*7)
32         x = self.fc_layers(x) # 全连接层分类: (batch, 1568)→(batch, 10)
33         return x
34
35 # 测试模型(假设输入图片尺寸为(28, 28), 单通道)
36 model = ConvNet()
37 input_tensor = torch.randn(32, 1, 28, 28) # batch_size=32, 1通道, 28x28图片
38 output = model(input_tensor)
39 print(f'模型输出形状: {output.shape}') # 输出: torch.Size([32, 10])
40
```

3. 请使用 PyTorch 实现一个循环神经网络模型类，该网络包含一个 LSTM 层和两个全连接层。LSTM 层有 128 个节点，第一个全连接层有 64 个节点，第二个全连接层有 10 个节点。请使用 ReLU 作为激活函数。（10 分）

```
1 import torch
2 import torch.nn as nn
3
4 class LSTMNet(nn.Module):
5     def __init__(self, input_size=10, seq_len=5):
6         super(LSTMNet, self).__init__()
7         self.input_size = input_size # 每个时间步的输入维度
8         self.seq_len = seq_len # 序列长度
9         self.hidden_size = 128 # LSTM隐藏层节点数
10
11     # LSTM层: input_size→hidden_size, batch_first=True (batch维度在前)
12     self.lstm = nn.LSTM(
13         input_size=input_size,
14         hidden_size=self.hidden_size,
15         num_layers=1,
16         batch_first=True,
17         bidirectional=False # 单向LSTM
18     )
19
20     # 全连接层部分: hidden_size→64→10
21     self.fc_layers = nn.Sequential(
22         nn.Linear(self.hidden_size, 64), # LSTM输出→全连接层1
23         nn.ReLU(),
24         nn.Linear(64, 10) # 全连接层2 (输出10类)
25     )
26
27     def forward(self, x):
28         # 前向传播: LSTM层→全连接层
29         # x: (batch_size, seq_len, input_size) → 示例: (32, 5, 10)
30         lstm_out, (hidden, cell) = self.lstm(x) # lstm_out: (32,5,128); hidden: (1,32,128)
31
32         # 取LSTM最后一个时间步的隐藏状态作为全连接层输入
33         # hidden[-1]: 取最后一层的隐藏状态 → (32, 128)
34         out = self.fc_layers(hidden[-1]) # (32,128)→(32,64)→(32,10)
35
36         return out
37
38 # 测试模型 (输入序列长度5, 每个时间步输入维度10)
39 model = LSTMNet(input_size=10, seq_len=5)
40 input_tensor = torch.randn(32, 5, 10) # batch_size=32, seq_len=5, input_size=10
41 output = model(input_tensor)
42 print(f'模型输出形状: {output.shape}') # 输出: torch.Size([32, 10])
```

- 解析：核心考点是 LSTM 层的定义与使用、序列数据的处理（取最后时间步输出）、LSTM 与全连接层的维度匹配；需注意 `batch_first=True` 时输入形状为 `(batch, seq_len, input_size)`，LSTM 的隐藏状态形状为 `(num_layers, batch, hidden_size)`，需取最后一层的隐藏状态 (`hidden[-1]`) 作为全连接层的输入。
- 举一反三：若要求“使用双向 LSTM”，则修改 LSTM 层参数 `bidirectional=True`，并调整全连接层输入维度为 `hidden_size * 2`（双向 LSTM 的隐藏状态是两个方向的拼接），即 `nn.Linear(self.hidden_size * 2, 64)`。