南開大學
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《深度学习及应用》课程作业

Lab03：循环神经网络

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：李重仪

2025 年 6 月 18 日

目录

1 实验目的

1. 掌握 RNN 原理；
2. 学会使用 PyTorch 搭建循环神经网络来训练名字识别；
3. 学会使用 PyTorch 搭建 LSTM 网络来训练名字识别。

2 实验要求

1. 老师提供的原始版本 RNN 网络结构（可用 print(net) 打印，复制文字或截图皆可）、在名字识别验证集上的训练 loss 曲线、准确度曲线图以及预测矩阵图；
2. 个人实现的 LSTM 网络结构在上述验证集上的训练 loss 曲线、准确度曲线图以及预测矩阵图；
3. 解释为什么 LSTM 网络的性能优于 RNN 网络（重点部分）；
4. 格式不限。

3 实验原理

3.1 目标任务

本次实验我们主要的任务是完成**名字识别**的任务，也就是根据一个人的名字来确认他的国籍，是一个简单的**多分类任务**。

3.2 实验流程

本次实验，我们的主要步骤如下所示：

- **数据集加载：**加载我们的人名数据集；
- **模型选择：**选择我们所使用的模型，如 RNN，LSTM；
- **模型训练：**利用训练数据集对我们的模型进行训练；
- **模型评估与测试：**对训练后的模型进行测试；
- **绘图展示结果：**通过可视化来展示我们的测试结果。

4 实验环境

本次实验继续使用 A100 的四卡 linux 环境的服务器上进行实验。我们还是建立我们的虚拟环境，通过：

```
python -m venv RNN
source RNN/bin/activate
pip install -r requirements.txt
```

这样就可以完成我们的本次实验的环境配置了！

5 实验过程

5.1 原始 RNN 网络结构展示

我们首先根据老师提供的示例代码，进行初始 RNN 框架 [1] 的构建。对应的代码在报告的附录中进行展示。

构建完毕后，我们使用 print(net) 进行网络结构的展示，如下所示。

```
RNN(
  (i2h): Linear(in_features=185, out_features=128, bias=True)
  (i2o): Linear(in_features=185, out_features=18, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

可以看到，这一个网络主要主要包含两部分，一部分是根据当前的输入和历史 hidden 信息更新 hidden 信息，另一部分是根据当前的输入和历史的 hidden 生成输出，这两部分可以使用两个简单的线性层来实现。

在进行训练的和推理的时候，我们依次将文本中的每一个单词作为输入，输入到 RNN 中，然后更新 hidden，并且产生输出，除了最后一个输入的输出之外，其他的输出都可以忽略。将最后一个单词的输出拿来进行分类预测。

此部分我们仅展示网络的设计过程，对应的结果我们将在**实验结果与分析**中进行展示。后续，为了适配我们的数据集进行训练，我对该网络结构进行了简单的改写，使其能够去处理对应的训练任务。

5.2 LSTM 网络设计

在本次实验中，本人实现了 LSTM 网络 [2] 结构的设计。我们跟 RNN 一样，也设计了一个单层的网络结构。

LSTM 网络结构主要分为三个部分，分别是输入门、输出门、遗忘门。我们的模型包含以下核心组件：

- **LSTM 层：**处理输入序列并提取特征；
- **全连接层：**将 LSTM 的输出映射到分类空间；
- **可选的序列打包机制：**优化处理可变长度序列。

下面简要介绍一下各结构的作用。

1. **遗忘门 (Forget Gate)：**遗忘门决定上一时刻的细胞状态 C_{t-1} 有多少信息需要被遗忘。具体的公式为 $f_t = \sigma\left(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}\right)$。在我们的代码中，遗忘门的计算被封装在底层。此处就不再介绍。
2. **输入门 (Input Gate)：**控制当前输入 x_t 有多少信息被添加到细胞状态。对应的公式为：

$$i_t = sigmoid(W_i * [h_{t-1}, x_t] + b_i)$$

$$candidate_t = tanh(W_c * [h_{t-1}, x_t] + b_c)$$

$$C_t = f_t * C_{t-1} + i_t * candidate_t$$

3. **输出门 (Output Gate)：**决定当前细胞状态 C_t 有多少信息被输出到隐藏状态。对应的公式如下所示：

$$o_t = sigmoid(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

我们设计的 LSTM 模型的优化之处在于：

1. LSTM 通过门控机制和细胞状态，使得梯度可以更稳定地传播；
2. 在序列建模任务 (如语音识别、机器翻译、时间序列预测等) 中表现显著优于传统模型。

5.3 LSTM 优化设计

该部分我们实现了对原始 LSTM 网络的优化。我们的核心优化如下所示。

1. **透明的门控机制实现：**相比原始实现的黑盒 nn.LSTM，这里完全透明地展示了 LSTM 的四个关键门控结构，便于理解、调试和修改内部机制。
2. **先进的正交权重初始化：**使用正交初始化代替默认随机初始化，能更好地保持梯度范数，缓解 RNN 训练中的梯度消失/爆炸问题，提升模型收敛速度和稳定性。
3. **模块化分层架构：**明确分层结构，每层有独立参数；第一层接收原始输入，后续层接收前一层的隐藏状态；比原始实现更清晰地展示多层 LSTM 的数据流。
4. **增强的隐藏状态管理：**提供专用方法初始化隐藏状态，比原始实现更清晰易用，同时支持分层状态管理。
5. **端到端分类输出处理：**直接输出 log 概率，便于与 NLLLoss 配合使用，比原始实现更完整地封装了分类任务所需组件。

6 实验结果与分析

6.1 实验设置

为了统一对我们的三个模型进行测试，我们选取了相同的参数来进行实验。我们统一训练 50 个轮次，用 Adam 优化器进行优化训练，学习率为 0.001，采用 CrossEntropyLoss 作为我们的损失计算。我们使用 A100 40GB 的 GPU 进行训练与测试，下面展示我们的测试结果。

6.2 结果展示

6.2.1 RNN 结果展示

展示一下我们的训练结果可视化的准确率、损失以及矩阵图，如图6.1所示。

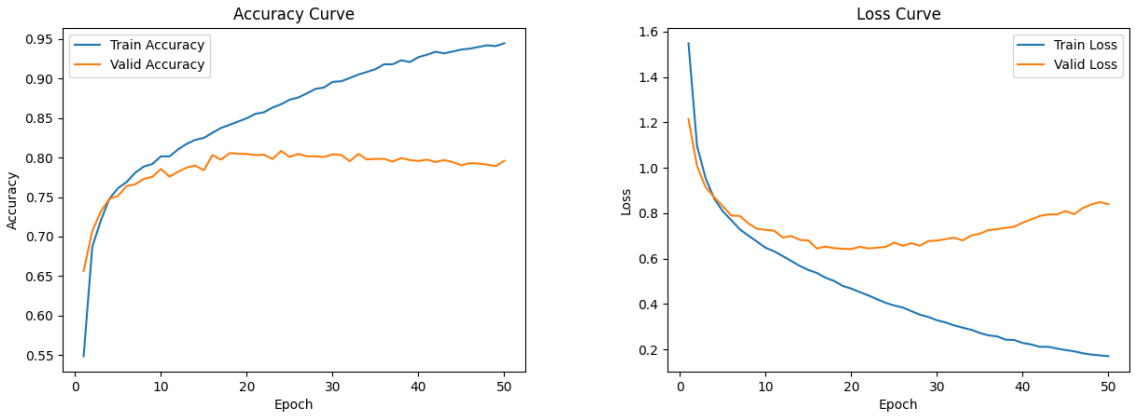

图 6.1: RNN 网络在数据集上的测试情况

可以发现，我们在验证的过程当中，还是出现了一些过拟合的情况，可能是由于网络结构过于复杂导致的过度拟合。我们的验证集准确率大概是在 79.58% 左右。

下面展示一下我们的混淆矩阵图，如图6.2所示。

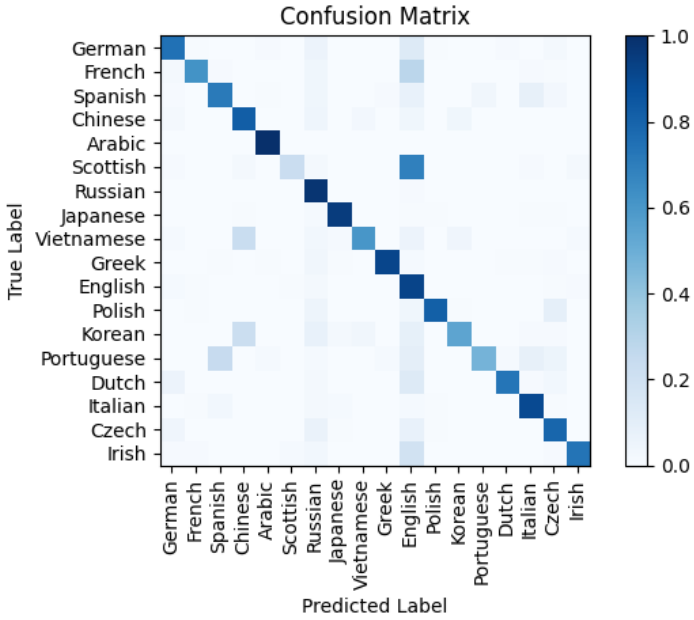

图 6.2: RNN 网络的混淆矩阵图

我们发现，实际上在大部分分类任务上，我们的误差都已经非常小了，只有在部分任务上，比如将 Scottish 识别为 English，这可能是因为这两种语言有一些相似导致的。总体来说，我们的 RNN 网络很好地完成了分类任务。

6.2.2 LSTM 结果展示

展示一下我们的训练结果可视化的准确率、损失以及矩阵图，如图6.3所示。

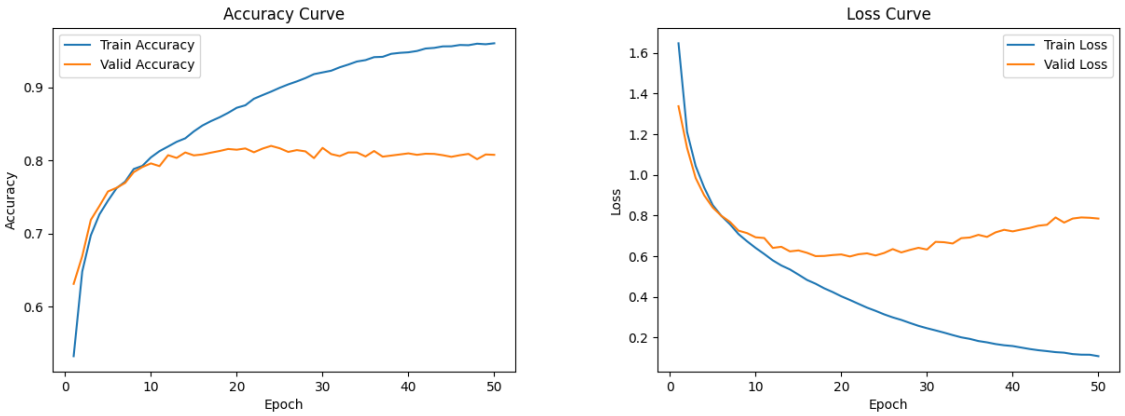

图 6.3: LSTM 网络在数据集上的测试情况

此处我们的过拟合情况相比于前面的 RNN 就好一些了，我们最后的验证准确率大概到了 80.77% 左右。

下面展示一下我们的混淆矩阵图，如图6.4所示。

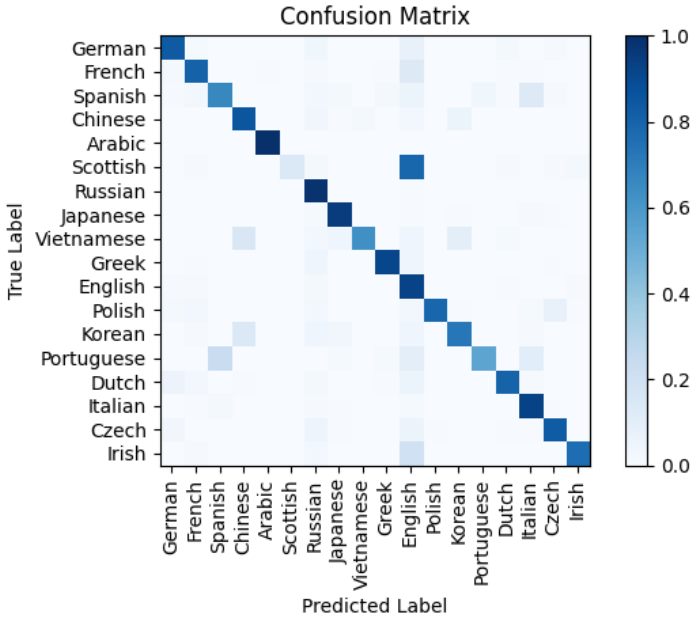

图 6.4: LSTM 网络的混淆矩阵图

我们发现，实际上混淆矩阵图和之前的 RNN 的差不多，也是基本上可以完成大部分的判定，但是对于 Scottish 和 English 可能不能做出很好的判断。

6.2.3 LSTM OurSelves 结果展示

展示一下我们的训练结果可视化的准确率、损失以及矩阵图，如图6.5所示。

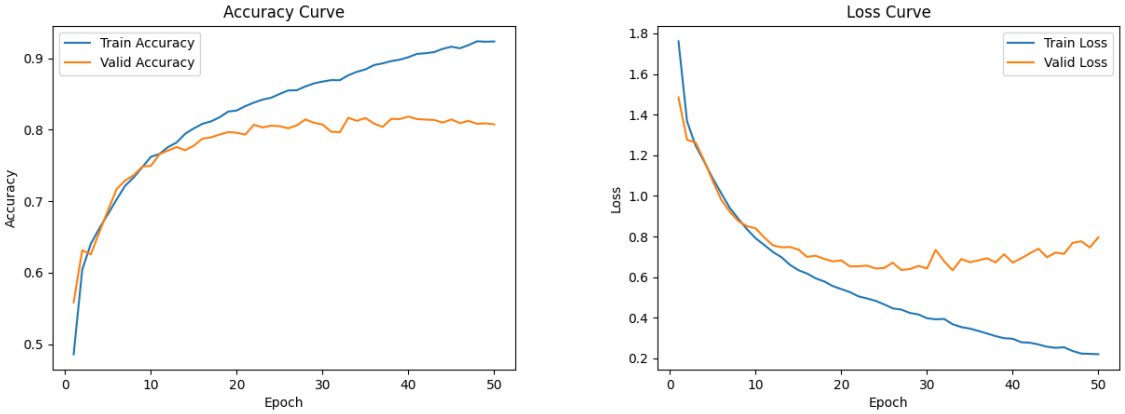

图 6.5: LSTM OurSelves 网络在数据集上的测试情况

可以看到，基本上没有出现过拟合的情况，而且通过我们的训练日志也可以看出，我们的准确率是逐步上升的！使用我们自行构建的 LSTM 网络，将我们的验证集准确率提升到了 81.84% 左右。

下面展示一下我们的混淆矩阵图，如图6.6所示。

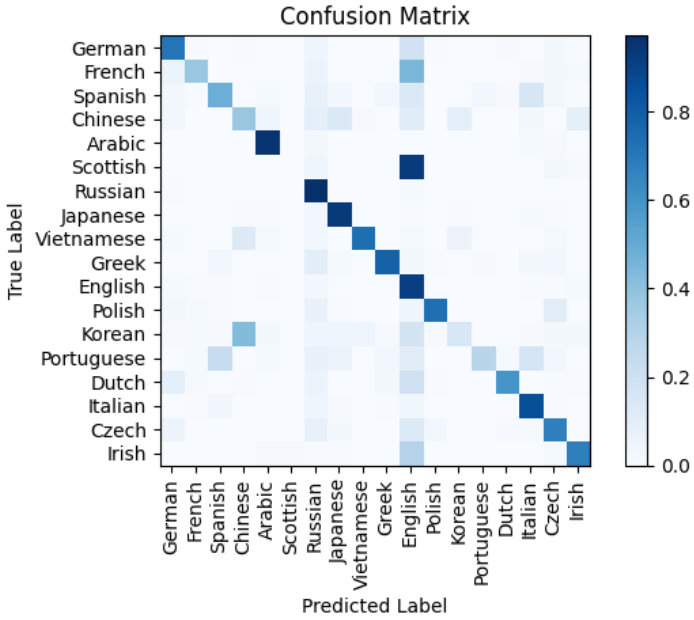

图 6.6: LSTM OurSelves 网络的混淆矩阵图

我们通过图像还是可以看出，在 Scottish 上的识别容易识别为 English，而其他的图像都可以基本上做到很好的识别。从中，我们可以看出我们自身模型的优越性。

6.3 对比总结——为什么 LSTM 优于 RNN

根据我们对三个模型的训练与验证，我们得到以下的结果：

模型名称	训练准确率	验证准确率	训练损失	验证损失
RNN	94.43%	79.58%	0.1698	0.8393
LSTM	96.01%	80.77%	0.1083	0.7852
LSTM OurSelves	90.15%	**81.84%**	0.2970	0.6719

表 1: 三个模型在数据集上的测试结果

我们发现，通过我们的实验结果可以看出，虽然我们的原始模型 LSTM 和 RNN 有着很好的训练准确率，但是他们的验证准确率却不如我们自己设计的 LSTM 网络。虽然我们的 LSTM 只有 90.15% 的训练准确率，但是我们的验证准确率是最高的。

然后，我们开始分析为什么 LSTM 会比我们的 RNN 要具有更好的性能。

LSTM 通过引入门机制来解决 RNN 在处理长序列数据时的长期依赖关系问题。LSTM 的门机制使得它可以更有效地捕捉到序列中的长距离依赖关系，从而提高了其在处理长序列数据时的表现。

LSTM 引入了门控机制，通过引入**遗忘门（forget gate）、输入门（input gate）和输出门（output gate）**，有效地解决了传统 RNN 中梯度消失和梯度爆炸的问题，使得模型能够更好地捕捉长期依赖关系，解决了长期依赖的问题。

LSTM 具有**细胞状态（cellstate）和隐状态（hidden state）**，细胞状态通过遗忘门和输入门进行更新，使得模型可以选择性地记住或遗忘信息，从而保持相关的长时间依赖信息。

所以我们的 LSTM 网络可以去很好地学习一些长期依赖的关系问题，使得我们的验证准确率得到了进一步的提升。

7 总结与体会

在本次实验中，我深入了解了 RNN 和 LSTM 的网络结构，并对其分别进行了编程实现。我的主要工作如下所示：

- 完成了 RNN、LSTM 以及变体的实现；
- 基于数据集的格式进行初步处理，将三个模型统一进行训练和测试；
- 通过输出的准确率图像和混淆矩阵图像，分析训练结果和部分样例失败的原因。

存在的问题就是，找不到一个很好的方法去解决 Scottish 的识别错误的问题，可能这个问题我们暂时无法使用简单的 LSTM 网络来解决，可能整个准确率不算特别高，就是因为这些数据集的问题。

8 文件目录结构

本次实验的代码文件目录结构如下所示：

```
codes ..........................................项目总目录
├── .gitignore ..................................git 忽略文件
├── main.py .....................................主程序入口
├── plot.py .....................................绘图脚本
├── pre.py ......................................数据预处理脚本
├── requirements.txt ............................依赖库清单
├── train.py ....................................模型训练脚本
├── __init__.py .................................包初始化文件
├── results .....................................训练结果目录
│   ├── rnn .....................................RNN 模型结果
│   │   ├── accuracy_curve.png ..................准确率曲线
│   │   ├── confusion_matrix.png ................混淆矩阵
│   │   ├── log.txt .............................训练日志
│   │   ├── loss_curve.png ......................损失曲线
│   │   └── model.pth ...........................模型权重文件
│   ├── lstm_ourselves ..........................自定义 LSTM 模型结果
│   │   ├── accuracy_curve.png ..................准确率曲线
│   │   ├── confusion_matrix.png ................混淆矩阵
│   │   ├── log.txt .............................训练日志
│   │   ├── loss_curve.png ......................损失曲线
│   │   └── model.pth ...........................模型权重文件
│   └── lstm ....................................标准 LSTM 模型结果
│       ├── accuracy_curve.png ..................准确率曲线
│       ├── confusion_matrix.png ................混淆矩阵
│       ├── log.txt .............................训练日志
│       ├── loss_curve.png ......................损失曲线
│       └── model.pth ...........................模型权重文件
├── model .......................................模型定义模块
│   ├── LSTM.py .................................标准 LSTM 模型
│   ├── LSTM_OurSelves.py .......................自定义 LSTM 模型
│   └── RNN.py ..................................RNN 模型
└── data ........................................数据集目录
```

本次实验的有关代码和文件，都已经上传至我的个人 github 中。您可以通过访问**此链接**来查阅我的代码文件。

附录 A RNN 代码设计

```python
class RNN(nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, num_classes):

        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=False)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, h0=None, lengths=None):

        batch_size = x.size(1)
        if h0 is None:
            h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size,
            ↪  dtype=x.dtype, device=x.device)
        if lengths is not None:
            x = nn.utils.rnn.pack_padded_sequence(x, lengths, batch_first=False,
            ↪  enforce_sorted=True)

        out, h = self.rnn(x, h0)

        idx = [-1] * batch_size
        if lengths is not None:
            out, idx = nn.utils.rnn.pad_packed_sequence(out, batch_first=False)
            idx = [i - 1 for i in idx]

        last_sequence_list = []
        for i in range(batch_size):
            last_sequence_list.append(out[idx[i], i, :])
        out = torch.stack(last_sequence_list)

        out = self.fc(out)
        return out, h
```

附录 B LSTM 代码设计

```python
class LSTM(nn.Module):

    def __init__(self, input_size, hidden_size, num_layers, num_classes):
```

```
4
5           super(LSTM, self).__init__()
6           self.hidden_size = hidden_size
7           self.num_layers = num_layers
8           self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=False)
9           self.fc = nn.Linear(hidden_size, num_classes)
10
11      def forward(self, x, h0=None, c0=None, lengths=None):
12
13          batch_size = x.size(1)
14          if h0 is None:
15              h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size,
                ↪  dtype=x.dtype, device=x.device)
16          if c0 is None:
17              c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size,
                ↪  dtype=x.dtype, device=x.device)
18          if lengths is not None:
19              x = nn.utils.rnn.pack_padded_sequence(x, lengths, batch_first=False,
                ↪  enforce_sorted=True)
20
21          out, (h, c) = self.lstm(x, (h0, c0))
22
23          idx = [-1] * batch_size
24          if lengths is not None:
25              out, idx = nn.utils.rnn.pad_packed_sequence(out, batch_first=False)
26              idx = [i - 1 for i in idx]
27
28          last_sequence_list = []
29          for i in range(batch_size):
30              last_sequence_list.append(out[idx[i], i, :])
31          out = torch.stack(last_sequence_list)
32
33          out = self.fc(out)
34          return out, (h, c)
```

附录 C LSTM OurSelves 代码设计

```
1   class LSTMCell(nn.Module):
2       def __init__(self, input_size, hidden_size):
3           super(LSTMCell, self).__init__()
4           self.input_size = input_size
```

```
        self.hidden_size = hidden_size

        # Input gate
        self.W_ii = nn.Parameter(torch.randn(hidden_size, input_size))
        self.W_hi = nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.b_i = nn.Parameter(torch.zeros(hidden_size))

        # Forget gate
        self.W_if = nn.Parameter(torch.randn(hidden_size, input_size))
        self.W_hf = nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.b_f = nn.Parameter(torch.zeros(hidden_size))

        # Output gate
        self.W_io = nn.Parameter(torch.randn(hidden_size, input_size))
        self.W_ho = nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.b_o = nn.Parameter(torch.zeros(hidden_size))

        # Cell state
        self.W_ig = nn.Parameter(torch.randn(hidden_size, input_size))
        self.W_hg = nn.Parameter(torch.randn(hidden_size, hidden_size))
        self.b_g = nn.Parameter(torch.zeros(hidden_size))

        # Initialize weights
        self.init_weights()

    def init_weights(self):
        for param in self.parameters():
            if len(param.shape) >= 2:
                nn.init.orthogonal_(param)
            else:
                nn.init.zeros_(param)

    def forward(self, x, h_prev, c_prev):
        """
        x: input tensor of shape (batch_size, input_size)
        h_prev: previous hidden state (batch_size, hidden_size)
        c_prev: previous cell state (batch_size, hidden_size)
        """
        # Input gate
        i_t = torch.sigmoid(x @ self.W_ii.t() + h_prev @ self.W_hi.t() + self.b_i)

        # Forget gate
```

```
        f_t = torch.sigmoid(x @ self.W_if.t() + h_prev @ self.W_hf.t() + self.b_f)

        # Output gate
        o_t = torch.sigmoid(x @ self.W_io.t() + h_prev @ self.W_ho.t() + self.b_o)

        # Cell candidate
        g_t = torch.tanh(x @ self.W_ig.t() + h_prev @ self.W_hg.t() + self.b_g)

        # Cell state update
        c_t = f_t * c_prev + i_t * g_t

        # Hidden state update
        h_t = o_t * torch.tanh(c_t)

        return h_t, c_t

class CustomLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(CustomLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # Create LSTM layers
        self.lstm_cells = nn.ModuleList([
            LSTMCell(input_size if i == 0 else hidden_size, hidden_size)
            for i in range(num_layers)
        ])

        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h0=None, lengths=None):

        batch_size = x.size(1)
        seq_len = x.size(0)

        # Initialize hidden and cell states
        if h0 is None:
            h_states = [torch.zeros(batch_size, self.hidden_size, device=x.device)
                        for _ in range(self.num_layers)]
            c_states = [torch.zeros(batch_size, self.hidden_size, device=x.device)
                        for _ in range(self.num_layers)]
```

```
        else:
            h_states, c_states = h0

        # Process each time step
        outputs = []
        for t in range(seq_len):
            x_t = x[t]

            # Process each layer
            for layer in range(self.num_layers):
                if layer == 0:
                    h_states[layer], c_states[layer] = self.lstm_cells[layer](
                        x_t, h_states[layer], c_states[layer]
                    )
                else:
                    h_states[layer], c_states[layer] = self.lstm_cells[layer](
                        h_states[layer-1], h_states[layer], c_states[layer]
                    )

            outputs.append(h_states[-1])

        # Stack outputs
        outputs = torch.stack(outputs)

        # Use the last output for classification
        final_output = self.fc(outputs[-1])
        final_output = self.softmax(final_output)

        return final_output, (h_states, c_states)

    def init_hidden(self, batch_size, device):
        h_states = [torch.zeros(batch_size, self.hidden_size, device=device)
                    for _ in range(self.num_layers)]
        c_states = [torch.zeros(batch_size, self.hidden_size, device=device)
                    for _ in range(self.num_layers)]
        return h_states, c_states
```

参考文献

[1] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.