南開大學
Nankai University

网络空间安全学院

深度学习实验报告

实验三：循环神经网络

姓名：武桐西

学号：2112515

专业：信息安全

指导教师：侯淇彬

2024 年 6 月 25 日

目录

1 实验目的

本次实验的主要内容是循环神经网络（Recurrent Neural Network, RNN）。通过本次实验，需要掌握以下内容：

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 搭建长短期记忆网络（Long Short-Term Memory, LSTM）来训练名字识别。

本次实验的总揽如下：

1. 笔者改进了原教程中给出代码，实现了比较规范通用的数据加载与预处理(通过继承自类`Dataset`)、模型训练与评估方法。
2. 笔者实现了基于`nn.RNN`的 RNN 模型，基于`nn.LSTM`的 LSTM 模型以及不使用`nn.LSTM`的个人实现 LSTM 单元的前向传递逻辑（包括输入门、遗忘门、输出门和细胞状态的更新）的 LSTM 模型。
3. 笔者通过封装数据集以及重载`collate_fn`函数实现了 RNN 和 LSTM 对于批处理（mini-batch）时可变长数据的处理。
4. 笔者对上述三种模型在名字识别数据集上进行训练并测试，分析并解释实验结果。

2 实验环境

本实验在 macOS 系统下测试，搭配 M3 Max 芯片，使用 Python`3.11.5`解释器，PyTorch 版本为`2.2.0`（使用 MPS 进行硬件加速），如表1所示。

表 1: 实验环境

操作系统	macOS	硬件	M3 Max (支持 MPS)
Python	3.11.5	PyTorch	2.2.0 (支持 MPS)

本次实验的代码已上传GitHub。

3 文件目录结构

本次实验的文件目录结构如下：

```
RNN
├── data ..... Data Path
│   ├── eng-fra.txt
│   └── names ..... NameDataset Path
└── models ..... Models
    ├── RNN_Model.py ..... 基于 nn.RNN 的 RNN 模型
    ├── LSTM_Model.py ..... 基于 nn.LSTM 的 LSTM 模型
    └── MyLSTM.py ..... 个人实现 LSTM Cell 的 LSTM 模型（不使用 nn.LSTM）
```

- preprocessing.py ..... 数据预处理（数据集类、collate_fn 函数等）
- trainer.py ..... Trainer
- visualization.py ..... Visualization
- main.py ..... Main Script

4 实验原理与实验过程

在本部分，首先介绍本次实验的目标任务和数据集，然后介绍数据集的封装与预处理、变长序列（批处理）的实现，再说明三种循环神经网络架构（RNN、LSTM、MyLSTM）的实现，最后介绍整个实验的流程。

4.1 目标任务

本次实验需要实现**名字识别**任务，即分辨一个人名的国籍，属于**自然语言处理**（Natural Language Processing, NLP）领域的**分类**任务。

4.2 数据预处理

首先，封装数据集类`NameDataset`，继承自`Dataset`类，需要重写`__init__`、`__len__`和`__getitem__`三个方法。

数据预处理中最重要的是去除原始数据中的非 ASCII 字符（比如一些重音符号等）。

4.3 批处理时可变长序列的处理

为了使用批处理（Mini-Batch）提高模型的训练速度以及训练效果，编写`collate_fn`，用于在数据集加载时对同一批次中不同长度的序列进行 pad 操作，如代码1所示。

代码 1: collate_fn 函数

```python
def collate_fn(data):
    """
    Collate Function
    :param data: data
    :return: collated data
    """
    data.sort(key=lambda x: x[0].size(0), reverse=True)
    names, labels =  zip(*data)
    lengths = torch.tensor([name.size(0) for name in names])  # Convert to tensor
    names = torch.nn.utils.rnn.pad_sequence(names, batch_first=False, padding_value=0)
    labels = torch.stack(labels)

    return names, labels, lengths
```

此后，在 Dataloader 中使用该函数，并在后续的模型中使用 PackedSequence 对象进行可变长序列的操作，提高模型的训练效率和训练效果。

4.4 RNN

重构教程中给出的 RNN 模型，采用`nn.RNN`进行编写。在前向传播时，需要考虑可变长序列数据的处理。本部分的网络结构与代码如附录A中代码2所示。

4.5 LSTM

基于 PyTorch 中提供的`nn.LSTM`，按照与代码2类似的步骤，编写 LSTM 模型的代码。这里不再赘述，详情请参考源代码。

4.6 MyLSTM

手动实现 LSTM 的 Cell 模块，实现 LSTM 单元的前向传递逻辑（包括输入门、遗忘门、输出门和细胞状态的更新），该部分封装在类`CustomLSTMCell`中，如代码3所示。其他部分可 LSTM 的代码基本四类，唯一不同之处在于对于可变长序列数据的处理（PackedSequence）。

4.7 实验流程

本次实验的实验流程如下：

1. **数据加载与预处理**：加载名字识别数据集，过滤非 ASCII 字符，划分训练集与验证集，并对同一批次的不同长度序列进行 padding 操作。
2. **模型选择**：根据命令行参数的解析选择要使用的模型。
3. **模型训练**：使用 Trainer 类进行模型训练。
4. **模型评估**：在验证集上评估模型，并保存模型。
5. **结果可视化**：绘制训练过程中的损失以及准确率曲线（训练集和验证集），并绘制混淆矩阵。

5 实验结果与分析

5.1 实验设置

本实验中，所有的超参数设置均相同，如表2所示。

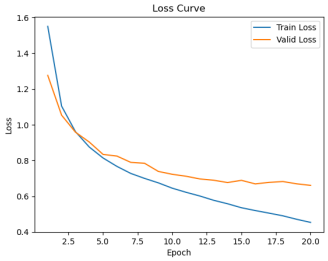
表 2: 实验设置

batch size	64	# of epoch	20
loss function	CrossEntropyLoss	learning rate	0.001
模型层数	1	隐藏层维度	128
优化器	Adam	硬件加速	MPS

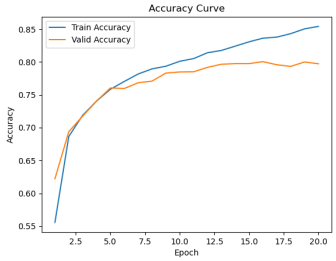
需要注意的是，本实验中所给的数据集并没有划分为训练集与测试集，因此这里笔者首先自行随机划分训练集与验证集，其中训练集的比例为 0.8。

5.2 结果与分析

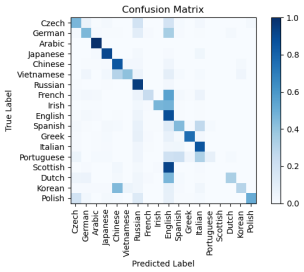
基于`nn.RNN`的 RNN 模型（RNN）、基于`nn.LSTM`的 LSTM 模型（LSTM）和笔者自行实现的不使用`nn.LSTM`的 LSTM 模型（MyLSTM）的实验结果分别如图5.1、图5.2和图5.3所示。



(a) 损失曲线

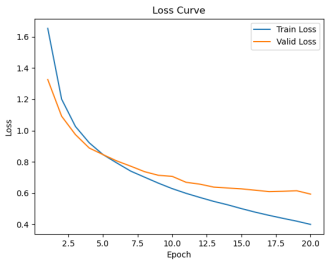


(b) 准确率曲线

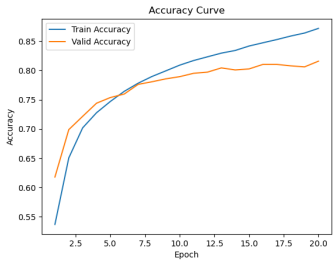


(c) 混淆矩阵

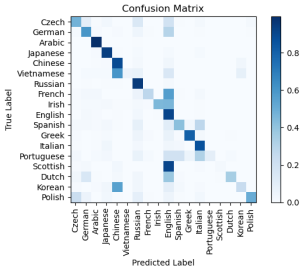
图 5.1: RNN Model 结果



(a) 损失曲线

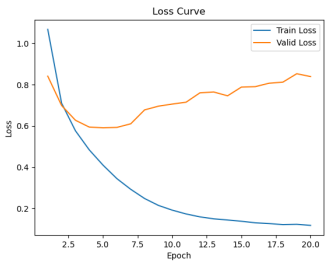


(b) 准确率曲线

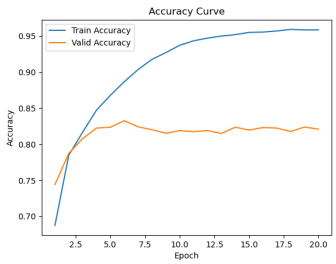


(c) 混淆矩阵

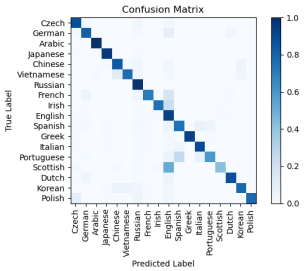
图 5.2: LSTM Model 结果



(a) 损失曲线



(b) 准确率曲线



(c) 混淆矩阵

图 5.3: MyLSTM Model 结果

从图中可知，三种模型均达到了 80%+ 的验证集准确率，特别是 MyLSTM，实现了 83.26% 的验证集准确率。从损失曲线和准确率曲线上可以看出，训练集与验证集上的损失（或准确率）具有同步变化的趋势，先快速下降（上升），随后趋于平稳，逐渐收敛，三种模型均**不存在过拟合**现象（MyLSTM 存在轻微的过拟合现象）。从混淆矩阵中可以看出，三种模型在各个类别上表现存在差异，原因可能是这些类别本身（对于人类而言）难以区分（例如，种族等历史渊源比较近的国家），另一种可能的原因是由于数据预处理中去除了非 ASCII 字符，因此可能导致一些可以原字符能够区分的变得不可区分。

5.3 对比总结

表 3: 实验结果

Model	Accuracy
RNN	80.05%
LSTM	81.68%
MyLSTM	83.26%

由表3可知，三种模型的表现相差不大，MyLSTM 的表现最好。理论上 LSTM 应该优于 RNN，但是这里的差别不大，LSTM 仅以微弱优势胜过 RNN，推测原因可能是该数据集相对简单，二者表现差距不大。

LSTM 相较于 RNN 的优越性 从理论上来讲, LSTM 要优于 RNN, 因为其引入了**门控机制**, LSTM 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（cell state）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。

6 总结与体会

在本次实验中，笔者深入了解循环神经网络的原理，并编程实现基础 RNN 和 RNN 的一些变种（如引入门控机制的 LSTM、GRU 等), 初步了解了自然语言处理 (Natural Language Processing, NLP) 的基本流程和方法，掌握了处理序列数据的方法。

本次实验中，笔者的一些实现包括：

- 笔者改进了原教程中给出代码,实现了比较规范通用的数据加载与预处理(通过继承自类`Dataset`)、模型训练与评估方法。
- 笔者实现了基于`nn.RNN`的 RNN 模型，基于`nn.LSTM`的 LSTM 模型以及不使用`nn.LSTM`的个人实现 LSTM 单元的前向传递逻辑（包括输入门、遗忘门、输出门和细胞状态的更新）的 LSTM 模型。
- 笔者通过封装数据集以及重载`collate_fn`函数实现了 RNN 和 LSTM 对于批处理（mini-batch）时可变长数据的处理。
- 笔者对上述三种模型在名字识别数据集上进行训练并测试，分析并解释实验结果。
- 实验结果比较成功，笔者收获满满。

存在的问题 在实验过程中，笔者发现 RNN 的每一个 Epoch 的训练时间波动较大，每一个 Epoch 的训练时间会越来越长，目前尚不清楚这一现象产生的原因。笔者对此有两个猜测：一是操作系统的内存换入换出等调度机制导致（与操作系统有关）；二是在训练过程中存在内存泄漏问题，导致训练时间变长。笔者希望后续能够继续探究该现象产生的原因。

参考文献

[1] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai.

附录 A 部分代码

代码 2: RNN Model

```python
class RNN(nn.Module):
    """
    RNN Model
    """
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        """
        Constructor
        """
         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):
        """
        Forward

        :param x: input (L x N x D) where L is the sequence length, N is the batch size, D is the feature dimension
        :param h0: initial hidden state (num_layers x N x hidden_size) (default: None -> zero initialization)
        :param lengths: lengths of sequences (N) (default: None for sequences of the same length)
        :return: output (N x num_classes), hidden state (num_layers x N x hidden_size)
        """
        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:
            # NOTE: enforce_sorted can be set to True for better performance(Default: True)
            #       because in this implementation, Dataloader already sorts the sequences
            #       If the input sequences are not sorted, set enforce_sorted to False
            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
```

代码 3: Custom LSTM Cell

```
class CustomLSTMCell(nn.Module):
    """
    Custom LSTM Cell
    """
    def __init__(self, input_size, hidden_size):
        """
        Constructor

        :param input_size: feature dimension of input
        :param hidden_size: number of features in the hidden state
        """
         super(CustomLSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.W_i = nn.Linear(input_size, hidden_size)
        self.U_i = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_f = nn.Linear(input_size, hidden_size)
        self.U_f = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_o = nn.Linear(input_size, hidden_size)
        self.U_o = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_c = nn.Linear(input_size, hidden_size)
        self.U_c = nn.Linear(hidden_size, hidden_size, bias=False)

```

```
25      def forward(self, x, h, c):
26          """
27          Forward pass for a single time step
28
29          :param x: input tensor at current time step (N x D)
30          :param h: hidden state tensor from previous time step (N x hidden_size)
31          :param c: cell state tensor from previous time step (N x hidden_size)
32          :return: (new hidden state, new cell state)
33          """
34          i_t = torch.sigmoid(self.W_i(x) + self.U_i(h))
35          f_t = torch.sigmoid(self.W_f(x) + self.U_f(h))
36          o_t = torch.sigmoid(self.W_o(x) + self.U_o(h))
37          c_tilda = torch.tanh(self.W_c(x) + self.U_c(h))
38          c_new = f_t * c + i_t * c_tilda
39          h_new = o_t * torch.tanh(c_new)
40          return h_new, c_new
```