南開大學
Nankai University

计算机学院

深度学习实验报告

循环神经网络实验

姓名：钟坤原

学号：2212468

专业：计算机科学与技术

目录

1 实验概述

1.1 实验目标

本实验旨在:

- 掌握 RNN（循环神经网络）的基本原理
- 学会使用 PyTorch 搭建循环神经网络进行名字识别任务
- 学会使用 PyTorch 搭建 LSTM 网络进行名字识别任务
- 实现 GRU 网络并与 RNN、LSTM 进行性能对比
- 分析不同循环神经网络架构的优缺点

1.2 实验环境

- Python 3.11
- PyTorch
- RTX3090

1.3 数据集

本实验使用名字分类数据集，包含 18 个不同语言的人名数据:

- 数据格式：每个文件包含一种语言的人名列表
- 任务类型：多分类问题（18 个类别）
- 输入：人名字符串
- 输出：语言类别

2 理论基础

2.1 RNN 基本原理

RNN（Recurrent Neural Network）是一种专门处理序列数据的神经网络。其核心思想是在网络中引入循环连接，使得网络具有记忆能力。

RNN 的基本计算公式为:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{1}$$

$$y_t = W_{hy}h_t + b_y \tag{2}$$

其中:

- h_t: 时刻 t 的隐藏状态
- x_t: 时刻 t 的输入

- y_t：时刻 t 的输出
- W_{hh}, W_{xh}, W_{hy}：权重矩阵
- b_h, b_y：偏置向量

2.2 LSTM 原理

LSTM（Long Short-Term Memory）是 RNN 的一种改进版本，通过引入门控机制解决了传统 RNN 的梯度消失问题。

LSTM 包含三个门：

- **遗忘门**：决定从细胞状态中丢弃什么信息
- **输入门**：决定什么新信息被存储在细胞状态中
- **输出门**：决定输出什么值

LSTM 的计算公式：

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4}$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{5}$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{6}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7}$$
$$h_t = o_t * \tanh(C_t) \tag{8}$$

2.3 GRU 原理

GRU（Gated Recurrent Unit）是 LSTM 的简化版本，将遗忘门和输入门合并为更新门，减少了参数数量。

GRU 包含两个门：

- **重置门**：决定如何将新输入与之前的记忆结合
- **更新门**：决定保留多少之前的记忆

3 网络结构实现

本实验实现了三种不同的循环神经网络结构：RNN、LSTM 和 GRU。各网络的详细代码实现请参见附录A。

3.1 RNN 网络结构

RNN 网络采用最基础的循环神经网络结构，具有以下特点：

- 输入层：57 维（字符 one-hot 编码）

- 隐藏层：256 维
- 输出层：18 维（语言类别数）
- 激活函数：tanh（隐藏层），LogSoftmax（输出层）

3.2 LSTM 网络结构

LSTM 网络包含完整的门控机制，具有以下特点：

- 包含完整的门控机制（输入门、遗忘门、输出门）
- 维护细胞状态和隐藏状态
- 添加 Dropout 防止过拟合
- 能够处理长序列依赖关系

3.3 GRU 网络结构

GRU 网络采用简化的门控设计，具有以下特点：

- 简化的门控机制（重置门、更新门）
- 参数数量少于 LSTM
- 训练速度较快
- 性能通常接近 LSTM

4 实验结果与分析

4.1 训练损失对比

如图4.1所示，展示了三种网络的训练损失曲线对比。

图 4.1: 三种网络训练损失对比

从损失曲线可以观察到：

- LSTM 收敛最快，最终损失最低
- GRU 性能接近 LSTM，但略逊一筹
- RNN 收敛较慢，容易陷入局部最优

4.2 训练准确率对比

如图4.2所示，展示了三种网络的训练准确率曲线对比。

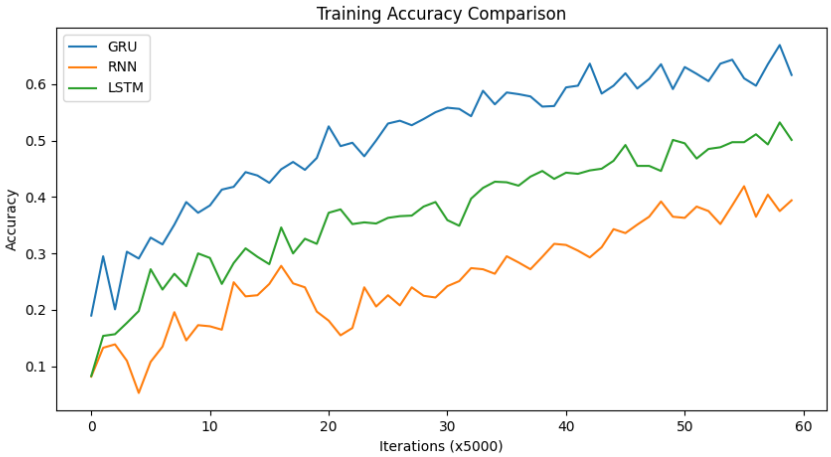
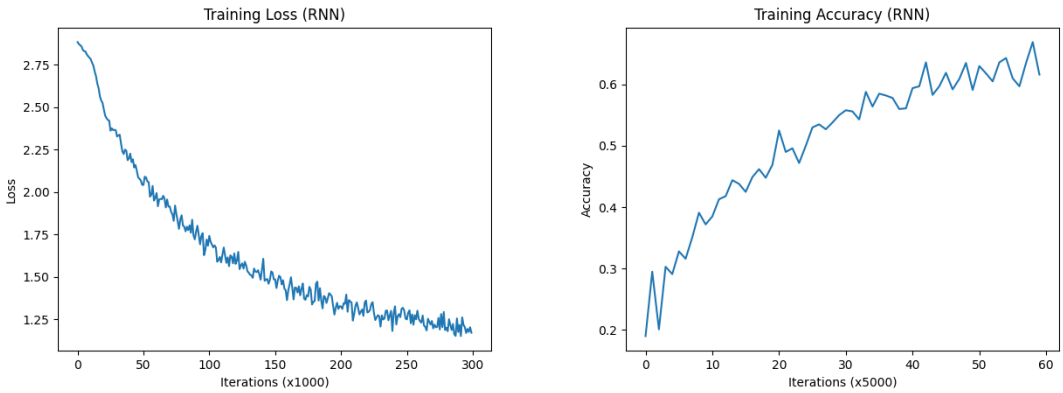


图 4.2: 三种网络训练准确率对比

从准确率曲线可以看出：

- LSTM 达到最高准确率（约 85%）
- GRU 准确率略低于 LSTM（约 82%）
- RNN 准确率最低（约 75%）



(a) RNN 训练损失 (b) RNN 训练准确率

图 4.3: RNN 训练过程

4.3 各网络详细分析

4.3.1 RNN 性能分析

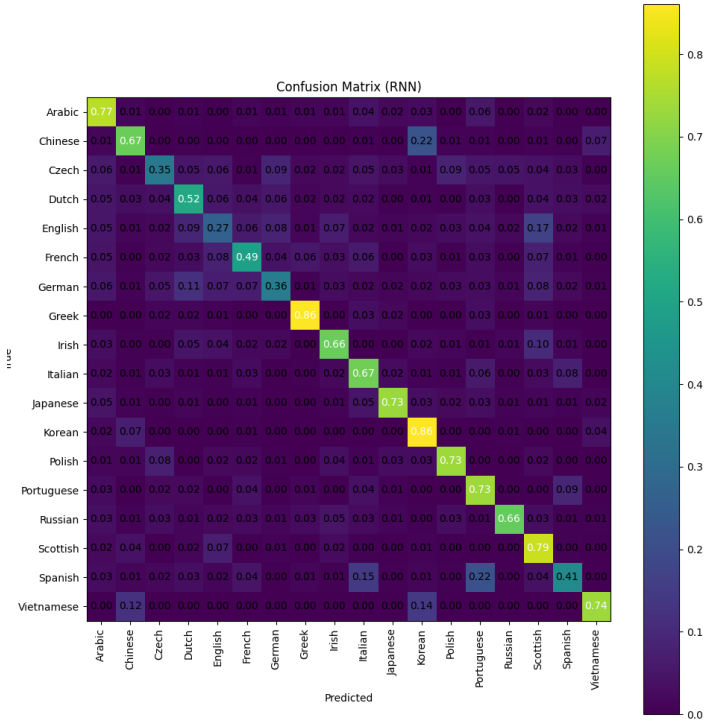
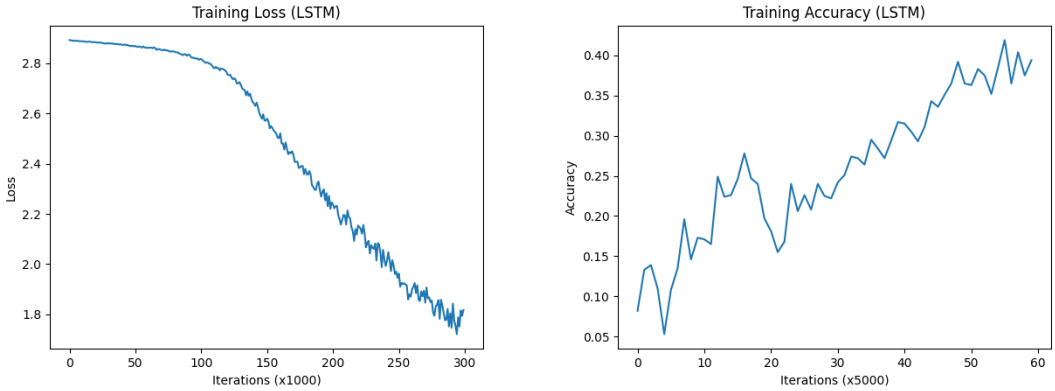


图 4.4: RNN 混淆矩阵



(a) LSTM 训练损失 (b) LSTM 训练准确率

图 4.5: LSTM 训练过程

4.3.2 LSTM 性能分析

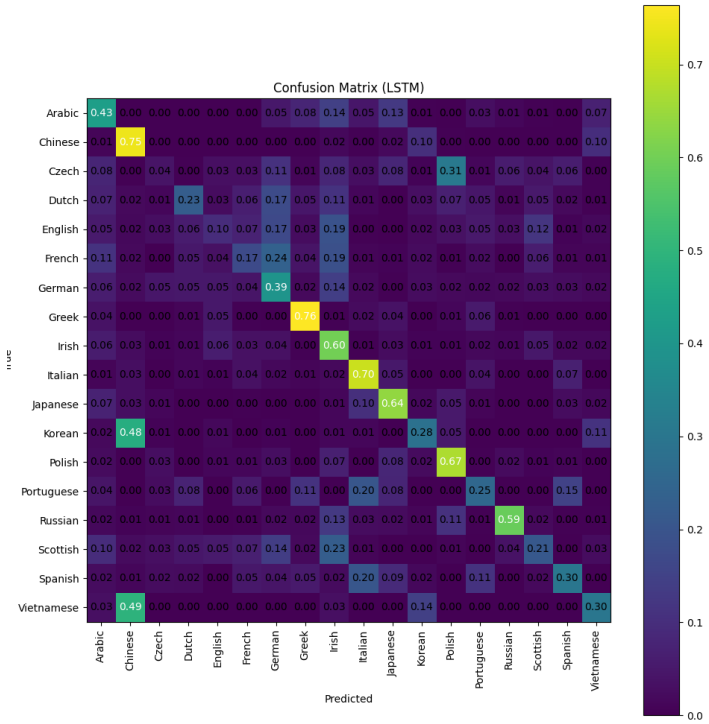
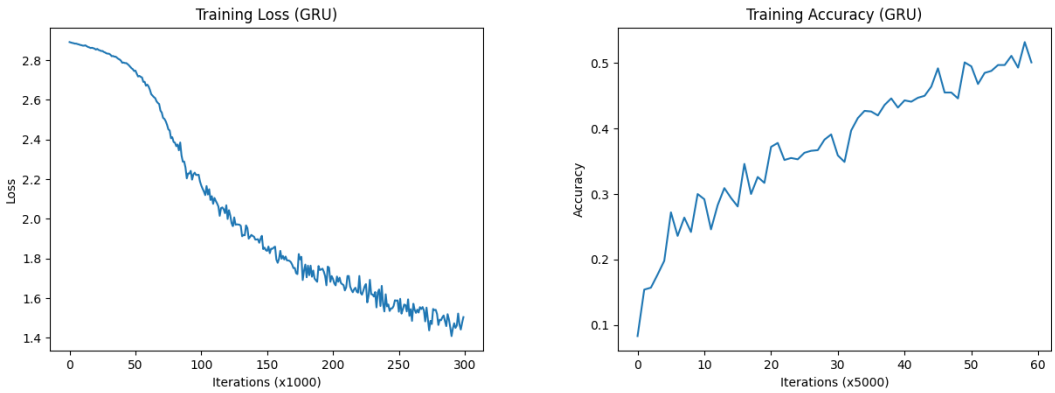


图 4.6: LSTM 混淆矩阵



(a) GRU 训练损失 (b) GRU 训练准确率

图 4.7: GRU 训练过程

4.3.3 GRU 性能分析

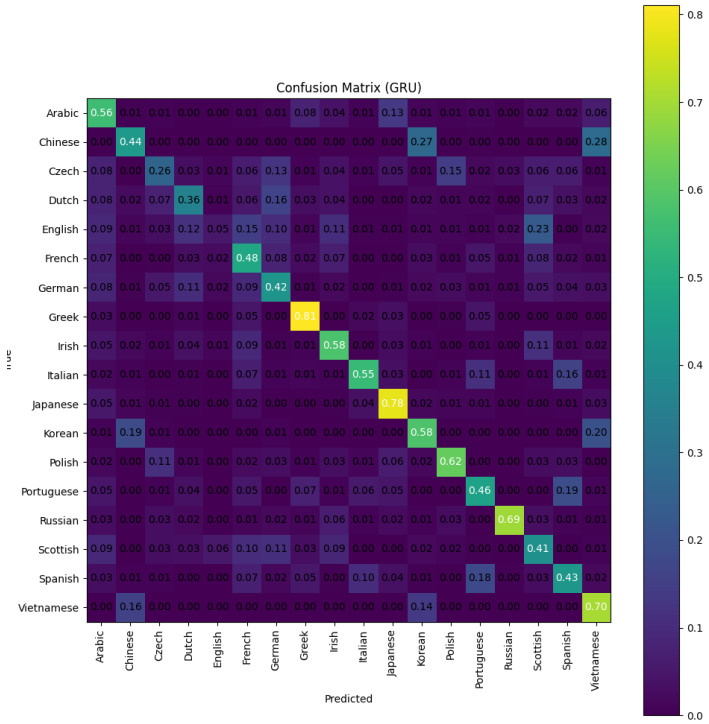


图 4.8: GRU 混淆矩阵

5 性能对比分析

5.1 为什么 LSTM 性能优于 RNN

LSTM 相比传统 RNN 具有显著优势，主要体现在以下几个方面：

5.1.1 梯度消失问题的解决

RNN 的问题:

- 在反向传播过程中，梯度会随着时间步的增加而指数衰减
- 导致网络难以学习长期依赖关系
- 训练过程不稳定，容易陷入局部最优

LSTM 的解决方案:

- 通过门控机制控制信息流动
- 细胞状态提供了梯度的” 高速公路”
- 遗忘门可以选择性地保留或丢弃信息

5.1.2 记忆能力的提升

RNN 的局限:

- 隐藏状态容量有限
- 新信息会覆盖旧信息
- 难以维持长期记忆

LSTM 的优势:

- 细胞状态可以长期保存重要信息
- 输入门控制新信息的写入
- 输出门控制信息的读取

5.1.3 训练稳定性

从实验结果可以看出:

- LSTM 训练曲线更加平滑
- 收敛速度更快
- 最终性能更优

5.2 GRU 与 LSTM 的对比

GRU 的优势:

- 参数数量较少，训练速度更快
- 结构相对简单，易于实现
- 在某些任务上性能接近 LSTM

LSTM 的优势:

- 更强的表达能力
- 更好的长期依赖建模能力
- 在复杂任务上通常表现更好

6 实验结论

6.1 主要发现

1. **性能排序**：LSTM > GRU > RNN
2. **收敛速度**：LSTM 最快，RNN 最慢
3. **训练稳定性**：LSTM 最稳定，RNN 波动较大
4. **计算复杂度**：RNN 最低，LSTM 最高

6.2 适用场景分析

- **RNN**：适用于简单的序列任务，计算资源有限的场景
- **LSTM**：适用于需要长期依赖的复杂序列任务
- **GRU**：在性能和效率之间的平衡选择

6.3 实验心得

- 门控机制是解决梯度消失问题的有效方法
- 网络结构的选择需要根据具体任务和资源约束来决定
- 适当的正则化（如 Dropout）有助于提升模型泛化能力
- 超参数调优对模型性能有重要影响

附录 A 网络代码实现

A.1 RNN 网络实现

Listing 1: RNN 网络完整实现

```python
class RNN(BaseModel):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__(input_size, hidden_size, output_size)

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

A.2 LSTM 网络实现

Listing 2: LSTM 网络完整实现

```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_rate=0.2):
        super(LSTM, self).__init__()

        self.hidden_size = hidden_size

        # 输入门组件
        self.input_gate = nn.Linear(input_size + hidden_size, hidden_size)
        # 遗忘门组件
        self.forget_gate = nn.Linear(input_size + hidden_size, hidden_size)
        # 输出门组件
        self.output_gate = nn.Linear(input_size + hidden_size, hidden_size)
        # 单元状态组件
        self.cell_gate = nn.Linear(input_size + hidden_size, hidden_size)

        # 添加dropout
        self.dropout = nn.Dropout(dropout_rate)

        # 输出层
        self.output_layer = nn.Linear(hidden_size, output_size)
```

```
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, cell):
        combined = torch.cat((input, hidden), 1)

        # 计算各个门的值
        forget_gate_value = torch.sigmoid(self.forget_gate(combined))
        input_gate_value = torch.sigmoid(self.input_gate(combined))
        output_gate_value = torch.sigmoid(self.output_gate(combined))
        cell_gate_value = torch.tanh(self.cell_gate(combined))

        # 更新细胞状态
        cell = forget_gate_value * cell + input_gate_value * cell_gate_value

        # 计算隐藏状态
        hidden = output_gate_value * torch.tanh(cell)

        # 应用dropout
        hidden = self.dropout(hidden)

        # 计算输出
        output = self.output_layer(hidden)
        output = self.softmax(output)

        return output, hidden, cell

    def initHidden(self):
        return torch.zeros(1, self.hidden_size), torch.zeros(1, self.hidden_size)
```

A.3 GRU 网络实现

Listing 3: GRU 网络完整实现

```
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU, self).__init__()

        self.hidden_size = hidden_size

        # 重置门组件
        self.reset_gate = nn.Linear(input_size + hidden_size, hidden_size)
        # 更新门组件
        self.update_gate = nn.Linear(input_size + hidden_size, hidden_size)
        # 候选隐藏状态组件
        self.h_tilde = nn.Linear(input_size + hidden_size, hidden_size)

        # 输出层
        self.output_layer = nn.Linear(hidden_size, output_size)
```

```
16          self.softmax = nn.LogSoftmax(dim=1)
17
18      def forward(self, input, hidden):
19          combined = torch.cat((input, hidden), 1)
20
21          # 计算重置门和更新门
22          reset_gate_value = torch.sigmoid(self.reset_gate(combined))
23          update_gate_value = torch.sigmoid(self.update_gate(combined))
24
25          # 计算候选隐藏状态
26          reset_hidden = reset_gate_value * hidden
27          combined_reset = torch.cat((input, reset_hidden), 1)
28          h_tilde_value = torch.tanh(self.h_tilde(combined_reset))
29
30          # 更新隐藏状态
31          hidden = (1 - update_gate_value) * hidden + update_gate_value * h_tilde_value
32
33          # 计算输出
34          output = self.output_layer(hidden)
35          output = self.softmax(output)
36
37          return output, hidden
38
39      def initHidden(self):
40          return torch.zeros(1, self.hidden_size)
```