南　　开　　大　　学

计　算　机　学　院

深度学习及应用实验报告

实验三　循环神经网络的实现

姓名：苏航

学号：2111039

专业：信息安全

2024 年 5 月 30 日

摘要

本实验在理解基本循环神经网络的基础上, 实现了原始 RNN 对名字数据集的识别. 同时我们也尝试了对基本神经网络的改进, 实现了 LSTM 网络结构, 实现了对名字数据集较高的识别率. 我们在实验中也感受到 LSTM 相对于基本 RNN 的优势.

关键字：循环神经网络 (RNN);LSTM

目录

一、 实验要求 **1**

二、 实验内容 **1**

三、 实验过程 **1**

（一） 循环神经网络的基本原理 1

（二） 原始 RNN 的实现 1

（三） 原始 RNN 的训练与评估 2

（四） LSTM 的实现 5

（五） 实验结果分析 8

四、 总结 **8**

一、 实验要求

本次实验要求掌握 RNN 原理, 学会在 pytorch 框架下搭建循环神经网络和 LSTM 网络实现名字的识别

.

二、 实验内容

打印原始版本 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; 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; 解释为什么 LSTM 网络的性能优于 RNN 网络

三、 实验过程

(一) 循环神经网络的基本原理

循环神经网络是一种特殊的神经网络，设计用于处理序列数据，如时间序列、文本等。相较于传统的前馈神经网络，RNN 在处理序列数据时具有记忆性，能够利用序列中的时间信息。

RNN 的核心思想是在网络的循环结构中引入一个隐藏状态，它能够捕捉到序列数据中的历史信息，并将这些信息传递到下一个时间步。这种结构使得 RNN 可以处理可变长度的输入序列，并在学习中考虑到序列中的顺序关系。在每个时间步，RNN 接受输入数据和前一个时间步的隐藏状态，并产生一个新的隐藏状态和输出。这种隐藏状态的更新方式允许网络对当前输入与过去输入的相关性进行建模，从而在序列数据中捕捉到长期的依赖关系。具体的模型结构如下所示:

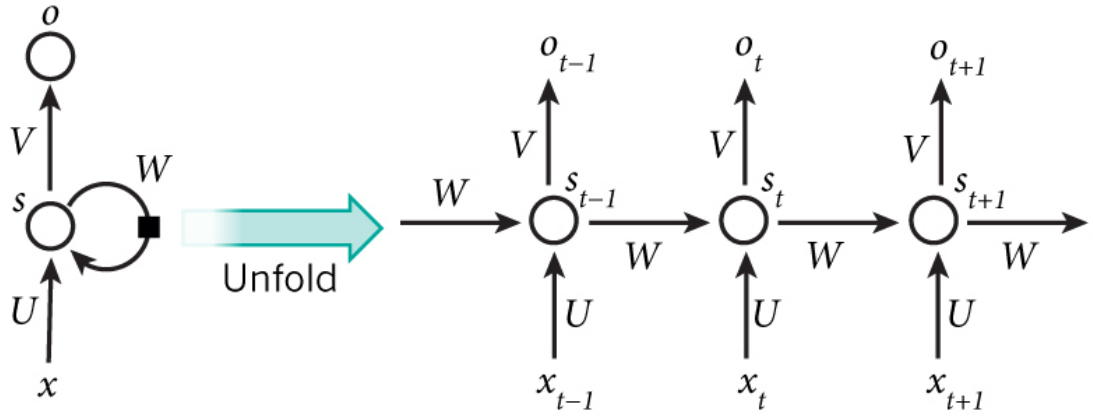

图 1: 循环神经网络模型图

(二) 原始 RNN 的实现

实验要求我们复现简单的 CNN 网络结构, 如下所示. 对于这个 RNN 首先有两个全连接层 i2h 和 i2o, 他们的输入维度都是 inputsize+hiddensize, 输出维度一个是 hiddensize 和 outputsize; 该模型选取的激活函数是 LogSoftmax. 前向传播部分是该模型的重点, 首先获取当前时间步的输入 input 以及上个时间步的隐藏状态 hidden, 然后把它们拼接在一起, 然后计算出新

的隐藏状态和输出, 输出再通过激活函数算出概率分布; 隐藏层的初始化也是很简单的初始隐藏状态是一个全零张量，大小为 (1, hiddensize)

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

搭建成功模型后我们可以把它输出出来:

```
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)
)
```

图 2: 原始 RNN 网络结构图

由此可见该网络的结构还是非常简单的

(三) 原始 RNN 的训练与评估

搭建完基本的 RNN 后我们就可以开始对其展开训练与评估, 实验要求我们采用名字数据集, 在进行训练之前我们需要对其进行预处理, 代码如下:

```
# 创建自定义数据集类
class NamesDataset(Dataset):
    def __init__(self, category_lines, all_categories, all_letters):
        self.category_lines = category_lines
        self.all_categories = all_categories
        self.all_letters = all_letters
        self.n_categories = len(all_categories)
        self.data = []
        for category in all_categories:
            for line in category_lines[category]:
                category_idx = all_categories.index(category)
```

```
                line_tensor = self.lineToTensor(line)
                self.data.append((category_idx, line_tensor))
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        category_idx, line_tensor = self.data[idx]
        return category_idx, line_tensor
    def letterToIndex(self, letter):
        return self.all_letters.find(letter)
    def lineToTensor(self, line):
        tensor = torch.zeros(len(line), len(self.all_letters))
        for li, letter in enumerate(line):
            tensor[li][self.letterToIndex(letter)] = 1
        return tensor

# 初始化数据集
dataset = NamesDataset(category_lines, all_categories, all_letters)
# 将数据集拆分为训练集和测试集
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, test_size])
def collate_fn(batch):
    # 获取类别张量和行张量
    category_tensors, line_tensors = zip(*batch)
    # 将类别转换为张量
    category_tensors = torch.tensor(category_tensors, dtype=torch.long)
    # 获取每个序列的长度
    lengths = [tensor.size(0) for tensor in line_tensors]
    max_length = max(lengths)
    # 创建填充后的张量
    padded_tensors = torch.zeros(len(line_tensors), max_length, n_letters)
    for i, tensor in enumerate(line_tensors):
        end = lengths[i]
        padded_tensors[i, :end, :] = tensor
    return category_tensors, padded_tensors
# 创建DataLoader
train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True,
    collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False,
    collate_fn=collate_fn)
```

在使用该数据集进行训练的时候一定要注意维度的匹配, 本实验多次遭遇到了处理之后的数据在训练函数中不匹配的问题, 需要我们对训练函数进行调整

```
def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()
    rnn.zero_grad()
    # 注意 line_tensor 的维度应该是 [batch_size, seq_len, input_size]
```

```
    # 我们需要将其转换为 [seq_len, batch_size, input_size] 以符合 RNN 的输入
    line_tensor = line_tensor.permute(1, 0, 2)
    for i in range(line_tensor.size(0)):
        output, hidden = rnn(line_tensor[i], hidden)
    loss = criterion(output, category_tensor)
    loss.backward()
    optimizer.step()
    return output, loss.item()
def evaluate(line_tensor):
    hidden = rnn.initHidden()
    line_tensor = line_tensor.permute(1, 0, 2)
    for i in range(line_tensor.size(0)):
        output, hidden = rnn(line_tensor[i], hidden)
    return output
```

实现该网络的训练和评估后我们得到下面的结果:

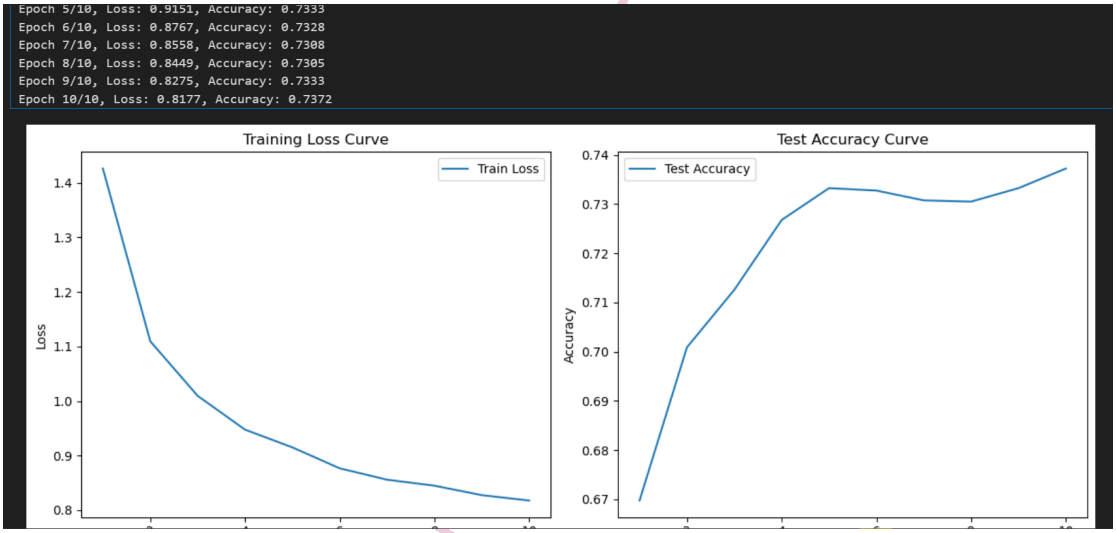

图 3: RNN 的损失和准确率曲线

我们可以看到原始网络结构的准确率相对较低, 只有 0.73. 而且较早地出现了过拟合的情况, 下面我们讲继续展示该模型的混淆矩阵图

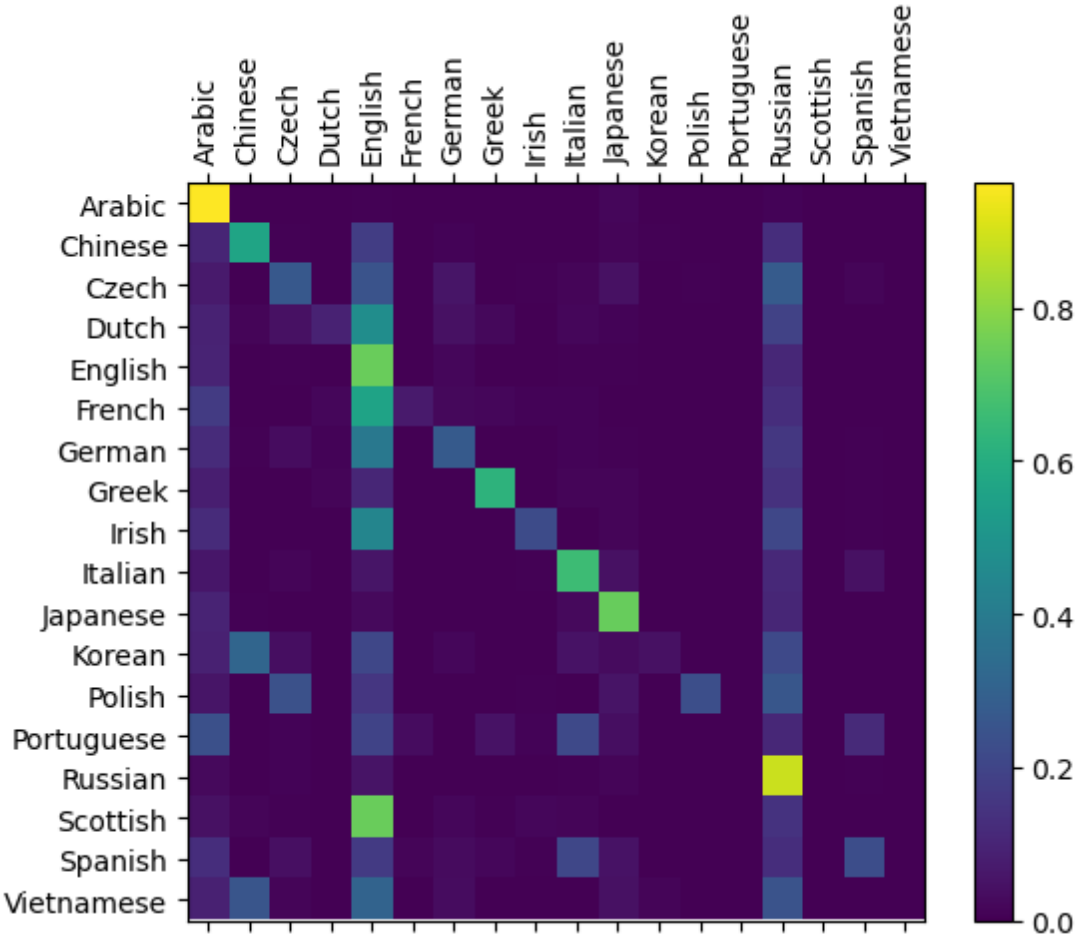

图 4: RNN 的混淆矩阵图

原始的 RNN 存在着不足, 这可能是因为该模型过于简单, 还有就是 RNN 模型本身的性质决定的, 它无法捕获长期依赖关系, 还有可能存在着梯度消失和梯度爆炸的问题, 要解决这些问题需要我们引入一个新的网络结构, 这个网络结构就是 LSTM

(四) LSTM 的实现

长短时记忆网络 (LSTM) 是 RNN 的一种变体, 相比传统的 RNN 结构，LSTM 引入了门控机制，可以更好地捕捉序列数据中的长期依赖关系. LSTM 通过引入门控单元来实现对信息的记忆和遗忘。一个典型的 LSTM 单元包含三个关键部分：输入门（Input Gate）：决定是否将当前输入加入到 LSTM 状态中; 遗忘门（Forget Gate）：决定是否从 LSTM 状态中遗忘一些信息; 输出门（Output Gate）：决定是否将 LSTM 状态输出给下一个时间步; 除了上述三个门, LSTM 还有一个称为“细胞状态”（Cell State）的组件，负责存储和传递信息。 具体的模型结构如下所示:

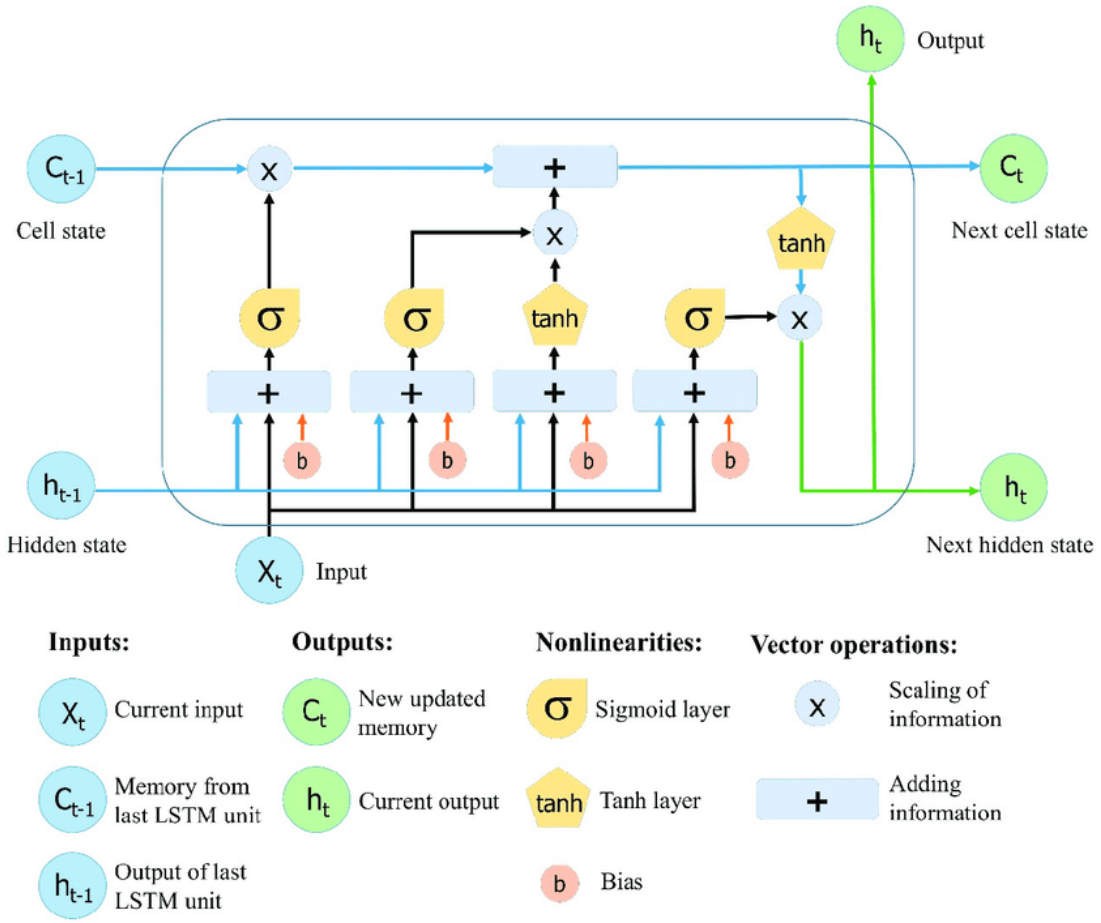

图 5: LSTM 模型示意图

下面我们根据 pytorch 提供的框架构建 LSTM:

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size)
        self.i2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
        # 初始化线性层的权重参数
        init.xavier_uniform_(self.i2o.weight)
    def forward(self, input, hidden):
        output, hidden = self.lstm(input.view(1, 1, -1), hidden)
        output = self.i2o(output.view(1, -1))
        output = self.softmax(output)
        return output, hidden
    def initHidden(self):
        # 初始化隐藏状态和细胞状态
        return torch.zeros(1, 1, self.hidden_size),torch.zeros(1, 1, self.
            hidden_size)
        return out
```

在 LSTM 中最引人瞩目的是引入了一个 LSTM 层, 为了模型的泛化能力我们使用 xavieruniform 该函数初始化权重参数. 前向传播函数也发生变化, 同时隐藏状态由一个元组 (h,c) 表示, 所以初始化隐藏状态的函数也要发生改变. 接下来我们观察其评估结果:

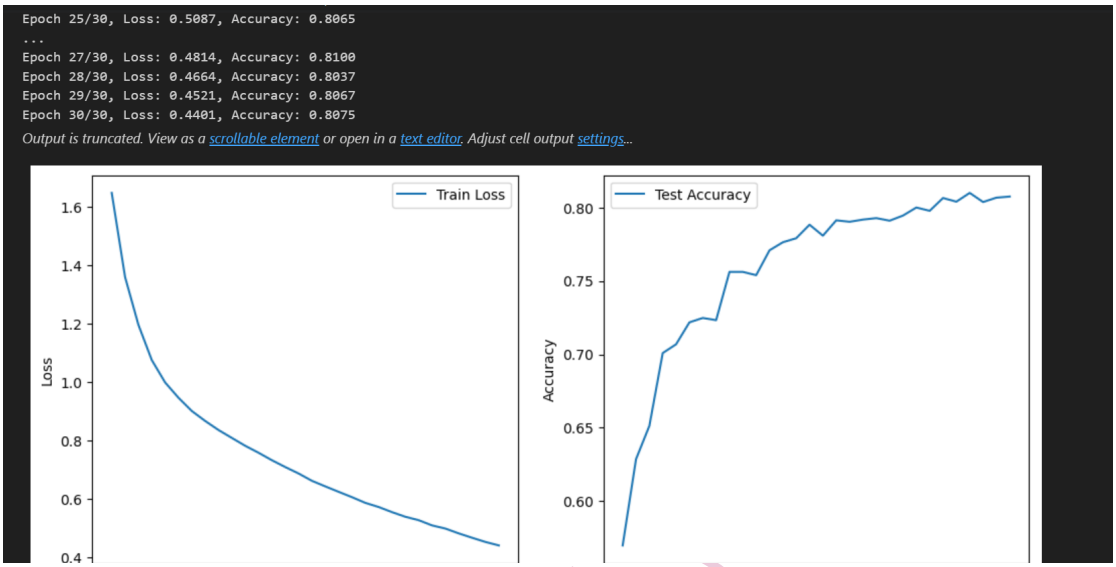

图 6: LSTM 的损失和准确率曲线

我们观察到 LSTM 模型的效果显著比 RNN 的准确率较高, 但是在训练所需的时间也更长, 收敛所需要的迭代次数也更多, 接下来我们来继续看其混淆矩阵

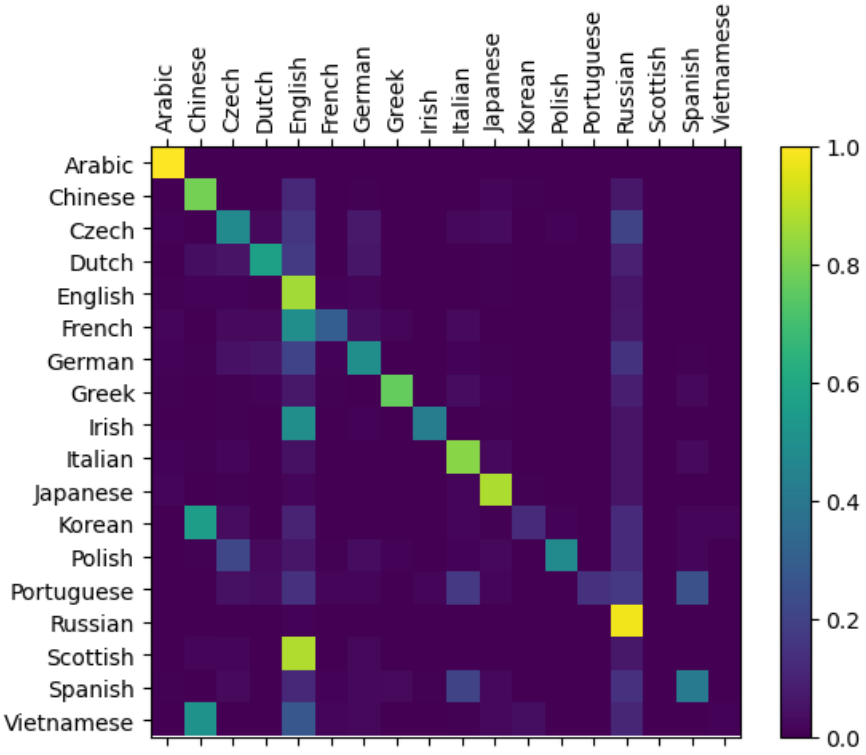

图 7: LSTM 的混淆矩阵

我们观察到其混淆矩阵也比原始的 RNN 更加清晰

（五） 实验结果分析

从上面的实验结果我们可以得到 LSTM 网络的性能要显著优于原始的 RNN, 具体原因如下:

1. LSTM 能处理长距离依赖关系:RNN 在处理长序列时会遇到梯度消失和梯度爆炸问题。这意味着当序列长度较长时，网络会难以学习到序列前部的信息，因为梯度会在反向传播过程中逐渐消失.LSTM 通过引入三个门（输入门、遗忘门和输出门）和一个记忆单元来有效地解决这个问题. 记忆单元允许信息在较长时间内保持和传递，从而在训练过程中保留长期依赖关系.

2.LSTM 能控制信息流动:LSTM 中的门结构使网络能够选择性地保留和丢弃信息. 具体来说：输入门决定哪些新信息需要加入记忆单元; 遗忘门决定哪些信息需要从记忆单元中移除; 输出门决定哪些信息从记忆单元输出. 这种控制机制使得 LSTM 可以更灵活地处理和保持重要的信息，而不会因为序列过长而丢失有价值的上下文。

3.LSTM 能改善梯度传播:LSTM 的记忆单元设计使得梯度在反向传播过程中可以在时间步之间顺畅地流动，从而减少了梯度消失问题. 即使在长序列上，LSTM 也能有效地训练，并保持较好的性能.

从模型结构的本质上概括:RNN 在每个时间步上更新隐藏状态，直接将先前时间步的隐藏状态与当前输入结合. 然而，标准 RNN 在处理长序列时效果不佳，因为梯度消失或爆炸会导致训练困难. 但是 LSTM 引入了复杂的单元结构和门控机制，能够有效地学习长时间依赖，解决了 RNN 在处理长序列时遇到的问题.

需要着重说明的是, 由于 LSTM 引入了复杂的门控机制和长期记忆机制,LSTM 的计算需要更多的参数和计算量, 其复杂度较高; 同时其复杂性导致了运行不太直观, 缺乏解释性; 更为致命的是 LSTM 需要大量的数据来避免过拟合, 如果数据不足 LSTM 面临着泛化能力不足的问题. 这就导致实际应用中 LSTM 收敛过慢, 例如: 在本次实验中传统 RNN 只需要 3min 就可以收敛, 而 LSTM 需要近 1h

对 LSTM 的优化也有很多, 其中大量使用的方法包括但不限于: 计算优化, 模型简化 (例如使用 GRU 模型, 本文并未实现), 数据量不足也可以通过数据增强和迁移学习来实习.

四、 总结

本次实验成功实现循环神经网络模型, 并成功实现了 LSTM. 通过这次实验我们对循环神经网络的结构有了更加深刻的理解.