chap6_RNN 实验报告

2254235 许昊珵

解释一下 RNN ，LSTM，GRU 模型

1. RNN（Recurrent Neural Network）—— 基础循环神经网络

特点：

- 适用于处理序列数据，如文本、语音、时间序列等。
- 通过隐藏状态（hidden state）存储和传递信息，使得当前时间步的计算依赖于前面时间步的信息。

缺点：

- **梯度消失/爆炸问题**：当序列较长时，反向传播过程中梯度可能会消失或爆炸，导致模型难以学习长期依赖关系。
- **短期记忆**：由于梯度消失，模型难以保留远距离的信息。
- **数学表示：** 假设 x_t 是当前输入，h_t 是当前隐藏状态，h_{t-1} 是上一时刻的隐藏状态：
- $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$
- 其中 W_h 和 W_x 是可训练权重，b 是偏置。

2. LSTM（Long Short-Term Memory）—— 长短时记忆网络

为了解决 RNN 的长期依赖问题，LSTM 引入了 **门控机制**，主要由 **遗忘门（Forget Gate）**、**输入门（Input Gate）** 和 **输出门（Output Gate）** 组成。

LSTM 结构：

- **遗忘门**：决定是否保留过去的记忆。
- **输入门**：决定是否接受当前输入的信息。
- **输出门**：决定当前时间步的隐藏状态输出。

优点：

- 通过门控机制，能有效捕捉长期依赖关系。
- 避免梯度消失问题，使得模型能记住较长时间的信息。

缺点：

- 计算复杂度较高，相比 RNN 需要更多参数，训练较慢。

3. GRU（Gated Recurrent Unit）—— 门控循环单元

GRU 是 LSTM 的改进版本，它减少了 LSTM 复杂度，同时保持了类似的性能。GRU 主要由两个门组成：

- **更新门（Update Gate）**：控制过去信息与新信息的融合。
- **重置门（Reset Gate）**：控制遗忘过去的信息。

优点：

- 计算开销比 LSTM 低，训练速度更快。
- 由于参数较少，能在小数据集上表现良好。

- 仍然能捕捉长期依赖。

缺点：

- 由于结构更简单，在某些复杂任务中可能不如 LSTM 强大。

叙述一下诗歌生成的过程

此 pytorch 框架的程序使用循环神经网络（RNN）来生成唐诗，并采用字符级别的生成方式，即根据当前输入的字符预测下一个字符，逐步生成完整的诗句。整个生成过程可以分为数据预处理、模型构建、训练、诗歌生成四个主要阶段。

1. 数据预处理

在 process_poems1 这个函数中，程序加载包含大量唐诗的 poems.txt 文件，每行是一首诗，然后过滤掉过长、过短或者包含特殊符号的诗句。每首诗的开头加上 start_token（起始符号），结尾加上 end_token（结束符号）。接着构建字典，统计所有出现的汉字，建立 word_int_map，将汉字映射到整数索引，vocabularies 则是所有可能的汉字集合，方便后续解码。最后将诗歌内容转换为索引列表，方便后续输入神经网络。

2. 模型构建

输入层使用 word_embedding 作为词嵌入，将整数索引转换成固定维度的向量表示，捕捉字与字之间的语义关系。循环神经网络通过 LSTM（长短时记忆网络）进行训练，捕捉诗歌的上下文信息。输出层采用 Softmax 激活函数预测下一个字符的概率分布，选择概率最高的字符作为下一个输出。

3. 模型训练

在 run_training 训练过程中，使用 poems_vector 作为训练数据：每一首诗的输入是一个字的索引序列，训练目标是预测下一个字的索引。通过 rnn_model 计算前向传播输出，预测下一个字符。使用 CrossEntropyLoss 计算交叉熵损失，进行反向传播，优化 RNN_model 的参数。整体循环多次训练 30 轮，提高生成诗歌的多样性。

4. 诗歌生成

在 gen_poem(begin_word) 这个函数中，程序按照以下步骤进行诗歌生成：

(1) 加载训练好的模型

```
rnn_model.load_state_dict(torch.load('./poem_generator_rnn', map_location=device,
weights_only=True))
```

这行代码将预训练的模型参数加载到 rnn_model，确保模型能够正常进行推理。

(2) 设定起始输入

```
poem = start_token + begin_word  # 用 start_token 开头，确保与训练一致
word = begin_word
```

(3) 逐字生成

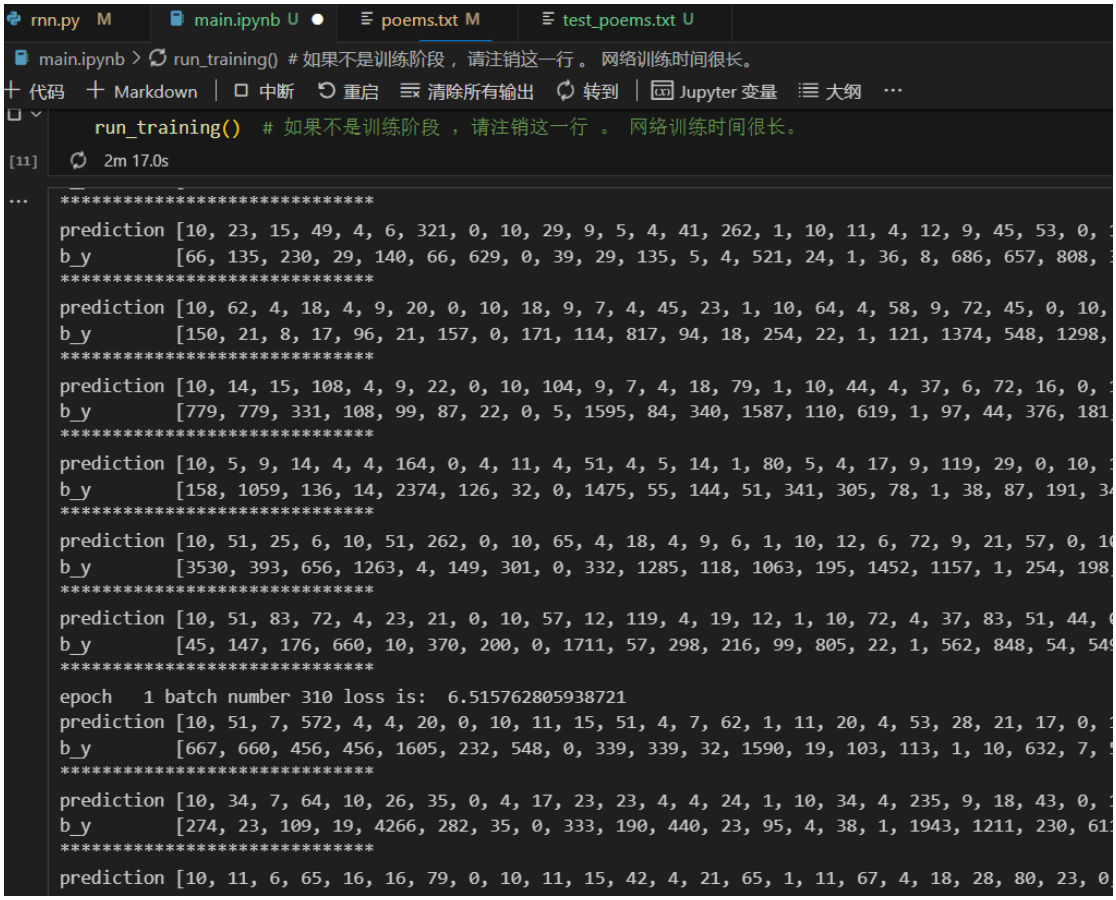
循环执行：

1. 把当前已有的诗歌 poem 转换成索引序列。
2. 传入 rnn_model，预测下一个字。
3. 选出概率最高的字 word，并拼接到 poem 后。
4. 如果达到 100 个字或遇到 end_token，则停止生成。

(4) 格式化输出

去除 start_token 和 end_token，按照句号分句，并进行排版，使其更符合诗歌格式。

训练过程截图



实验总结

基于 RNN 的古诗生成

1. 实验目的

本实验旨在使用 Recurrent Neural Network（RNN） 生成符合汉语韵律的古诗，并通过优化数据预处理、模型训练和推理策略，提高生成诗歌的质量与连贯性。

2. 实验流程

1. 数据预处理
    - 处理唐诗语料库，去除特殊字符，并构建 word_int_map（词-索引映射）和 vocabularies（词表）。
    - 使用 start_token 和 end_token 标记每首诗的起始和结束，确保训练时模型能够学习到完整的诗句结构。
2. 模型构建
    - 采用 RNN（LSTM）进行训练，其中：
        - word_embedding：词向量嵌入层，提升语义表达能力。

        - lstm_hidden_dim=128：LSTM 隐藏层维度，学习诗歌的上下文信息。
    - 通过 torch.nn.LSTM 设计模型，并在 GPU 上进行加速计算。
3. 模型训练
    - 设置 batch_size=64 进行小批量梯度下降训练。
    - 使用 CrossEntropyLoss 计算损失，并采用 Adam 优化器进行参数更新。
    - 训练 epoch=30 轮次，观察损失下降情况。
4. 诗歌生成
    - 给定 begin_word 作为诗歌的首字，并逐步预测下一个字。
    - 通过 to_word() 选择最高概率的字，拼接成完整的诗句。
    - 采用 pretty_print_poem() 格式化输出，使诗歌更加易读。

3. 关键问题与解决方案

1. 生成诗歌首字不匹配 begin_word
    - 问题原因：训练时使用 start_token 作为输入，而测试时直接用 begin_word，导致分布不匹配。
    - 解决方案：测试时修改输入，使模型能够从学习过的 begin_word 开始生成诗歌。
2. 模型训练速度极慢
    - 问题原因：在 CPU 上进行模型训练。
    - 解决方案：安装 CUDA 版本的 pytorch 库，在 GPU 上进行训练，速度提高了约 20 倍。

4. 实验结果分析

- 改进前：诗歌的首字不符合 begin_word，训练效率极低。
- 改进后：
    - 首字符合用户指定
    - 诗句结构合理，符合唐诗格式
    - 能迅速完成训练，便于调试分析结果

5. 未来优化方向

1. 更强的模型
    - 尝试 Transformer-based 模型（如 GPT、BERT），捕捉更远距离的上下文关系。
    - 采用双向 LSTM，提升诗歌生成的流畅度。
2. 更多诗歌风格
    - 目前的模型倾向于五言诗，可以扩展到七言诗或律诗。
    - 引入风格控制机制，如“田园诗”、“边塞诗”等，增强可控性。
3. 优化训练数据
    - 通过数据增强（如同义词替换、句子重构）扩展训练集，提高模型的泛化能力。
    - 采用更大规模的诗歌语料，提升模型表现。

6. 结论

本实验成功实现了基于 RNN 的古诗自动生成，优化了开头字匹配、内容丰富度和诗句流畅度，并提出了未来改进方向。实验表明，通过调整模型结构、训练策略和推理方法，可以进一步提升诗歌的质量和多样性。