Chapter6 报告

2252086 孙靖贻

一、RNN, LSTM, GRU 模型解释

1. RNN（循环神经网络）

循环神经网络（Recurrent Neural Network, RNN）是一种适用于处理序列数据的神经网络。RNN 通过循环连接的隐藏状态，使其能够记住之前的输入信息，从而在时间步之间共享信息。它的主要特点是：

- 时间依赖性：当前状态的计算依赖于前一步的状态，使其能够处理序列数据，如文本、语音和时间序列数据。
- 权重共享：相同的权重参数在不同时间步重复使用，使得 RNN 在处理不同长度的输入时具有一定的泛化能力。
- 梯度消失和梯度爆炸：由于 RNN 采用反向传播算法（BPTT, Backpropagation Through Time）来更新权重，长序列训练时可能会出现梯度消失或梯度爆炸的问题，导致模型难以学习远距离依赖关系。

2. LSTM（长短时记忆网络）

LSTM（Long Short-Term Memory）是 RNN 的改进版本，专门用于解决长序列数据的长期依赖问题。LSTM 通过引入 输入门（Input Gate）、遗忘门（Forget Gate） 和 输出门（Output Gate） 来控制信息的流动，使得网络可以记住长期信息，并选择性地遗忘不重要的信息。LSTM 的核心结构包括：

- 遗忘门（Forget Gate）：决定是否遗忘上一时刻的信息。
- 输入门（Input Gate）：决定当前时刻输入的信息是否存入细胞状态。
- 细胞状态（Cell State）：负责长期记忆的信息流动，可保留长时间的重要信息。
- 输出门（Output Gate）：决定当前时刻的输出。

由于 LSTM 具备更强的记忆能力，它在自然语言处理、语音识别和时间序列预测等任务中广泛应用。

3. GRU（门控循环单元）

GRU（Gated Recurrent Unit）是 LSTM 的简化版本，它只有两个门：

- 更新门（Update Gate）：控制当前输入信息与过去信息的融合程度。
- 重置门（Reset Gate）：控制是否遗忘过去的信息。

GRU 具有与 LSTM 相似的功能，但由于参数更少，计算更为高效，因此在某些任务上比 LSTM 更快，且表现接近甚至优于 LSTM。GRU 适用于对计算资源要求较高的应用场景，如移动端或实时预测任务。

总体而言，RNN 适用于短序列任务，而 LSTM 和 GRU 更适用于长序列任务。LSTM 由于其复杂的门控机制，更适用于需要长期依赖的任务，而 GRU 由于结构更简单，适用于对计算性能有较高要求的应用。

二、诗歌生成过程

本次实验采用 PyTorch 进行实现，使用了 RNN（LSTM） 模型来生成诗歌。整个诗歌生成过程如下：

1. 数据处理

首先，通过 process_poems1 或 process_poems2 读取并清理诗歌数据。数据预处理的主要步骤包括：

①过滤掉不符合要求的诗歌（长度过短或过长，包含特殊字符等）。

②添加起始标记 <G> 和结束标记 <E>。

②统计所有字符出现的频率，构建词表。

③将诗歌文本转换为数值索引表示。

2. 训练模型

①采用 LSTM 作为核心网络，输入大小为词嵌入维度(100)，隐藏层维度为 128，使用 2 层 LSTM。

②训练时，模型通过 generate_batch 生成 mini-batch 训练数据。

③采用交叉熵损失函数 NLLLoss，优化器选择 RMSprop 进行优化。

④训练 30 轮，每 20 个 batch 保存一次模型权重。

3. 生成诗歌

①加载训练好的 LSTM 模型。

②选择用户指定的 begin_word 作为诗歌的起始字。

③通过 LSTM 逐步预测下一个字，直到生成结束标记 <E> 或达到最大长度。

④使用 to_word() 将预测的数值索引转换回汉字。

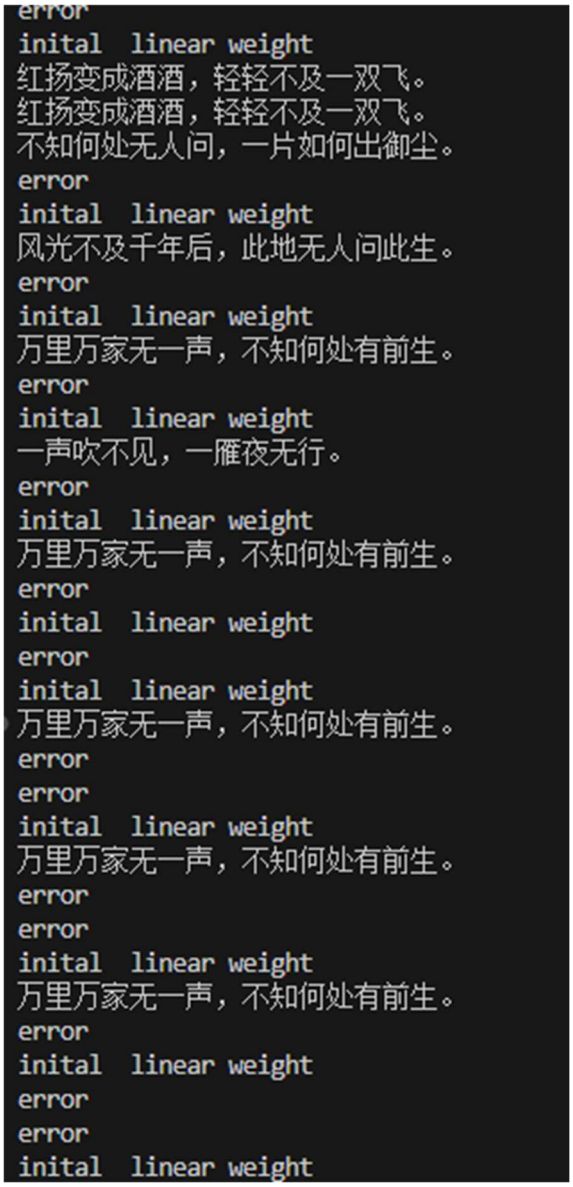
该过程利用 RNN 递归计算的特性，通过给定的起始字不断生成新的字，直至形成完整的诗歌。

三、诗歌训练及生成结果

训练过程截图如下：

```
******************************
epoch 0 batch number 11 loss is: 6.818305969238281
prediction [13, 1, 1, 3, 3, 1, 1, 3, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3]
b_y        [1538, 2174, 175, 426, 2396, 0, 107, 874, 2396, 152, 72, 1, 40, 6, 393, 99, 383, 0, 799, 19, 5, 27, 42, 1, 3, 3]
******************************
epoch 0 batch number 12 loss is: 6.8861613273620605
prediction [1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
b_y        [90, 8, 1556, 83, 3054, 0, 180, 7, 447, 433, 134, 1, 18, 71, 1323, 2329, 309, 0, 950, 108, 1999, 936, 94, 1, 3, 3]
******************************
epoch 0 batch number 13 loss is: 6.5778422355651855
prediction [421, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 3, 3]
b_y        [2067, 116, 338, 44, 5, 0, 609, 1427, 579, 295, 13, 1, 569, 39, 283, 157, 425, 0, 138, 2400, 124, 285, 121, 1, 3, 3]
******************************
epoch 0 batch number 14 loss is: 6.717309951782227
prediction [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 1, 3, 3]
b_y        [433, 331, 22, 159, 29, 0, 2801, 902, 100, 21, 79, 1, 1440, 492, 318, 174, 61, 0, 320, 183, 933, 252, 923, 1, 3, 3]
******************************
epoch 0 batch number 15 loss is: 6.528646469116211
```

生成结果截图如下：



四、实验总结

本次实验基于 RNN（LSTM）实现了古诗自动生成系统。然而，从实验结果来看，模型的生成效果存在较大问题，主要表现为：

1. 生成内容重复：例如“一声吹不见，一雁夜无行。”在多次生成中重复出现。可能的原因是训练不足，或模型过拟合特定的句式。
2. 语义混乱，缺乏连贯性：诗句之间逻辑不连贯，缺乏自然流畅的语义联系。可能是由于训练数据不足，或 RNN 长期依赖问题导致模型无法捕捉较长距离的依赖关系。
3. 最终结果与要求“生成诗歌开头词汇是日 、 红 、 山 、 夜 、 湖、 海 、 月 ”明显不符。
4. inital linear weight 的错误提示可能表明模型的权重初始化不当，影响了训练效果。

改进方向

1. 增加训练数据

- 确保训练数据足够大，提高模型的泛化能力。
- 过滤掉质量较差的诗歌样本，提升训练数据质量。

2. 调整模型结构
- 可以尝试使用 GRU 或 Transformer 代替 LSTM，提高生成文本的连贯性。
- 使用 Attention Mechanism 让模型能够关注重要的历史信息。

3. 优化训练过程
- 适当调整 learning rate，使用 scheduler 进行学习率衰减，防止模型陷入局部最优。
- 采用 Batch Normalization 或 Layer Normalization 以加速训练收敛。

4. 改进解码策略
- 目前可能使用的是 Greedy Decoding（贪心解码），可以改为 Beam Search 以提升生成质量。
- 结合 Top-k Sampling 和 Temperature Scaling，增加生成的多样性。

本次实验虽然实现了诗歌生成的基本功能，但仍有较大的优化空间，未来可以通过改进模型结构和训练策略来提升生成质量。

实验代码：

rnn.py:

```
import torch.nn as nn
import torch
from torch.autograd import Variable
import torch.nn.functional as F

import numpy as np

def weights_init(m):
    classname = m.__class__.__name__  #   obtain the class name
    if classname.find('Linear') != -1:
        weight_shape = list(m.weight.data.size())
        fan_in = weight_shape[1]
        fan_out = weight_shape[0]
        w_bound = np.sqrt(6. / (fan_in + fan_out))
        m.weight.data.uniform_(-w_bound, w_bound)
        m.bias.data.fill_(0)
        print("inital  linear weight ")


class word_embedding(nn.Module):
    def __init__(self,vocab_length , embedding_dim):
        super(word_embedding, self).__init__()
        w_embeding_random_intial = np.random.uniform(-
1,1,size=(vocab_length ,embedding_dim))
        self.word_embedding = nn.Embedding(vocab_length,embedding_dim)
        self.word_embedding.weight.data.copy_(torch.from_numpy(w_embedin
g_random_intial))
```

```
    def forward(self,input_sentence):
        """
        :param input_sentence:  a tensor ,contain several word index.
        :return: a tensor ,contain word embedding tensor
        """
        sen_embed = self.word_embedding(input_sentence)
        return sen_embed


class RNN_model(nn.Module):
    def __init__(self,
batch_sz ,vocab_len ,word_embedding,embedding_dim, lstm_hidden_dim):
        super(RNN_model,self).__init__()

        self.word_embedding_lookup = word_embedding
        self.batch_size = batch_sz
        self.vocab_length = vocab_len
        self.word_embedding_dim = embedding_dim
        self.lstm_dim = lstm_hidden_dim
        ##########################################
        # here you need to define the "self.rnn_lstm"  the input size is
"embedding_dim" and the output size is "lstm_hidden_dim"
        # the lstm should have two layers, and the  input and output
tensors are provided as (batch, seq, feature)
        # ???
        self.rnn_lstm = nn.LSTM(input_size=embedding_dim,
                                hidden_size=lstm_hidden_dim,
                                num_layers=2,
                                batch_first=True)
        ##########################################
        self.fc = nn.Linear(lstm_hidden_dim, vocab_len )
        self.apply(weights_init) # call the weights initial function.

        self.softmax = nn.LogSoftmax() # the activation function.
        # self.tanh = nn.Tanh()
    def forward(self,sentence,is_test = False):
        batch_input = self.word_embedding_lookup(sentence).view(1,-
1,self.word_embedding_dim)
        # print(batch_input.size()) # print the size of the input
        ################################################
        # here you need to put the "batch_input"  input the self.lstm
which is defined before.
        # the hidden output should be named as output, the initial
hidden state and cell state set to zero.
        # ???
```

```
        h0 = torch.zeros(2, batch_input.size(0), self.lstm_dim,
device=batch_input.device)
        c0 = torch.zeros(2, batch_input.size(0), self.lstm_dim,
device=batch_input.device)

        output, _ = self.rnn_lstm(batch_input, (h0, c0))
        ################################################
        out = output.contiguous().view(-1,self.lstm_dim)

        out =  F.relu(self.fc(out))

        out = self.softmax(out)

        if is_test:
            prediction = out[ -1, : ].view(1,-1)
            output = prediction
        else:
           output = out
        # print(out)
        return output
```

main.py:

```
import numpy as np
import collections
import torch
from torch.autograd import Variable
import torch.optim as optim

import rnn as rnn_lstm

start_token = 'G'
end_token = 'E'
batch_size = 64


def process_poems1(file_name):
    """

    :param file_name:
    :return: poems_vector  have tow dimmention ,first is the poem, the
second is the word_index
    e.g. [[1,2,3,4,5,6,7,8,9,10],[9,6,3,8,5,2,7,4,1]]

    """
```

```
    poems = []
    with open(file_name, "r", encoding='utf-8', ) as f:
        for line in f.readlines():
            try:
                title, content = line.strip().split(':')
                # content = content.replace(' ', '').replace(',
','').replace('。','')
                content = content.replace(' ', '')
                if '_' in content or '(' in content or '（' in content
or '《' in content or '[' in content or \
                                start_token in content or end_token in
content:
                    continue
                if len(content) < 5 or len(content) > 80:
                    continue
                content = start_token + content + end_token
                poems.append(content)
            except ValueError as e:
                print("error")
                pass
    # 按诗的字数排序
    poems = sorted(poems, key=lambda line: len(line))
    # print(poems)
    # 统计每个字出现次数
    all_words = []
    for poem in poems:
        all_words += [word for word in poem]
    counter = collections.Counter(all_words)  # 统计词和词频。
    count_pairs = sorted(counter.items(), key=lambda x: -x[1])  # 排序
    words, _ = zip(*count_pairs)
    words = words[:len(words)] + (' ',)
    word_int_map = dict(zip(words, range(len(words))))
    poems_vector = [list(map(word_int_map.get, poem)) for poem in poems]
    return poems_vector, word_int_map, words

def process_poems2(file_name):
    """
    :param file_name:
    :return: poems_vector  have tow dimmention ,first is the poem, the
second is the word_index
    e.g. [[1,2,3,4,5,6,7,8,9,10],[9,6,3,8,5,2,7,4,1]]

    """
    poems = []
```

```
    with open(file_name, "r", encoding='utf-8', ) as f:
        # content = ''
        for line in f.readlines():
            try:
                line = line.strip()
                if line:
                    content = line.replace(' '' ', '').replace(',
','').replace('。','')
                    if '_' in content or '(' in content or '（' in
content or '《' in content or '[' in content or \
                                    start_token in content or end_token
in content:
                        continue
                    if len(content) < 5 or len(content) > 80:
                        continue
                    # print(content)
                    content = start_token + content + end_token
                    poems.append(content)
                    # content = ''
            except ValueError as e:
                # print("error")
                pass
    # 按诗的字数排序
    poems = sorted(poems, key=lambda line: len(line))
    # print(poems)
    # 统计每个字出现次数
    all_words = []
    for poem in poems:
        all_words += [word for word in poem]
    counter = collections.Counter(all_words)  # 统计词和词频。
    count_pairs = sorted(counter.items(), key=lambda x: -x[1])  # 排序
    words, _ = zip(*count_pairs)
    words = words[:len(words)] + (' ',)
    word_int_map = dict(zip(words, range(len(words))))
    poems_vector = [list(map(word_int_map.get, poem)) for poem in poems]
    return poems_vector, word_int_map, words

def generate_batch(batch_size, poems_vec, word_to_int):
    n_chunk = len(poems_vec) // batch_size
    x_batches = []
    y_batches = []
    for i in range(n_chunk):
        start_index = i * batch_size
        end_index = start_index + batch_size
```

```
        x_data = poems_vec[start_index:end_index]
        y_data = []
        for row in x_data:
            y  = row[1:]
            y.append(row[-1])
            y_data.append(y)
        """
        x_data             y_data
        [6,2,4,6,9]       [2,4,6,9,9]
        [1,4,2,8,5]       [4,2,8,5,5]
        """
        # print(x_data[0])
        # print(y_data[0])
        # exit(0)
        x_batches.append(x_data)
        y_batches.append(y_data)
    return x_batches, y_batches


def run_training():
    # 处理数据集
    poems_vector, word_to_int, vocabularies =
process_poems1('./poems.txt')

    print("finish loading data")
    BATCH_SIZE = 100

    # 设置 GPU 设备
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

    torch.manual_seed(5)
    word_embedding =
rnn_lstm.word_embedding(vocab_length=len(word_to_int) + 1,
embedding_dim=100)

    rnn_model = rnn_lstm.RNN_model(batch_sz=BATCH_SIZE,
vocab_len=len(word_to_int) + 1,
                                   word_embedding=word_embedding,
embedding_dim=100, lstm_hidden_dim=128)
    rnn_model.to(device)  # 将模型移动到 GPU

    optimizer = optim.RMSprop(rnn_model.parameters(), lr=0.01)
    loss_fun = torch.nn.NLLLoss().to(device)  # 将损失函数移动到 GPU
```

```
    for epoch in range(30):
        batches_inputs, batches_outputs = generate_batch(BATCH_SIZE,
poems_vector, word_to_int)
        n_chunk = len(batches_inputs)

        for batch in range(n_chunk):
            batch_x = batches_inputs[batch]
            batch_y = batches_outputs[batch]  # (batch, time_step)

            loss = 0
            for index in range(BATCH_SIZE):
                x = np.array(batch_x[index], dtype=np.int64)
                y = np.array(batch_y[index], dtype=np.int64)

                x = Variable(torch.from_numpy(np.expand_dims(x,
axis=1))).to(device)  # 数据转 GPU
                y = Variable(torch.from_numpy(y)).to(device)  # 数据转
GPU

                pre = rnn_model(x)
                loss += loss_fun(pre, y)

                if index == 0:
                    _, pre = torch.max(pre, dim=1)
                    print('prediction', pre.cpu().data.tolist())  # 转换
到 CPU 以便打印
                    print('b_y       ', y.cpu().data.tolist())
                    print('*' * 30)

            loss = loss / BATCH_SIZE
            print("epoch", epoch, 'batch number', batch, "loss is:",
loss.cpu().data.tolist())

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(rnn_model.parameters(), 1)
            optimizer.step()

            if batch % 20 == 0:
                torch.save(rnn_model.state_dict(),
'./poem_generator_rnn')
                print("finish save model")
```

```
def to_word(predict, vocabs):  # 预测的结果转化成汉字
    sample = np.argmax(predict)

    if sample >= len(vocabs):
        sample = len(vocabs) - 1

    return vocabs[sample]


def pretty_print_poem(poem):  # 令打印的结果更工整
    shige=[]
    for w in poem:
        if w == start_token or w == end_token:
            break
        shige.append(w)
    poem_sentences = poem.split('。')
    for s in poem_sentences:
        if s != '' and len(s) > 10:
            print(s + '。')


def gen_poem(begin_word):
    poems_vector, word_int_map, vocabularies =
process_poems1('./poems.txt')

    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

    word_embedding =
rnn_lstm.word_embedding(vocab_length=len(word_int_map) + 1,
embedding_dim=100)
    rnn_model = rnn_lstm.RNN_model(batch_sz=64,
vocab_len=len(word_int_map) + 1,
                                   word_embedding=word_embedding,
embedding_dim=100, lstm_hidden_dim=128)

    rnn_model.load_state_dict(torch.load('./poem_generator_rnn'))
    rnn_model.to(device)  # 将模型移动到 GPU
    rnn_model.eval()

    poem = begin_word
```

```
    word = begin_word

    while word != end_token:
        input_data = np.array([word_int_map[w] for w in poem],
dtype=np.int64)
        input_data =
Variable(torch.from_numpy(input_data)).to(device)  # 数据转 GPU

        output = rnn_model(input_data, is_test=True)
        word = to_word(output.cpu().data.tolist()[-1], vocabularies)  #
转换到 CPU 以便使用
        poem += word

        if len(poem) > 30:
            break

    return poem




run_training()  # 如果不是训练阶段 ，请注销这一行 。 网络训练时间很长。


pretty_print_poem(gen_poem("日"))
pretty_print_poem(gen_poem("红"))
pretty_print_poem(gen_poem("山"))
pretty_print_poem(gen_poem("夜"))
pretty_print_poem(gen_poem("湖"))
pretty_print_poem(gen_poem("湖"))
pretty_print_poem(gen_poem("湖"))
pretty_print_poem(gen_poem("君"))
```