HW3 RNN

1. RNN, LSTM, GRU 模型

1.1 RNN 模型

RNN 是一种处理序列数据的神经网络，可以理解当前输入的上下文信息。与普通的前馈神经网络不同，RNN 具有记忆能力，适用于 NLP、时间序列预测、语音识别等任务。

- 核心结构

RNN 的主要特点是隐藏状态，它会记住前面时间步的信息，并用于当前时间步的计算；它在每个时间步共享相同的权重，让其能够处理不同长度的序列。

设输入序列为：$X = (x_1, x_2, \dots, x_T)$，隐藏状态为$h_t$（t 时刻），输出为$y_t$（t 时刻），则计算公式为：

⑴ 隐藏状态更新：

$$h_t = \tanh\left(W_h h_{t-1} + W_x x_t + b_h\right)$$

⑵ 输出计算：

$$y_t = W_y h_t + b_y$$

其中，W_h, W_x, W_y是可训练的权重矩阵，b_h, b_y是偏置项，激活函数tanh使隐藏状态具有非线性表达能力。

- 缺点

⑴ 梯度消失/梯度爆炸

由于 RNN 通过时间步传播梯度，长序列的梯度会指数级衰减（梯度消失）或放大（梯度爆炸）。其中，梯度消失导致模型无法学习远距离的依赖关系，更新变得很小；梯度爆炸导致模型参数更新过大，使训练不稳定；

⑵ 长期依赖问题

普通的 RNN 难以记住远距离的信息。

为了解决这些问题，引入了 LSTM，GRU，Transformer 等。

1.2 LSTM 模型

LSTM 是 RNN 的一种改进版本，为解决梯度消失和长期依赖问题设计。其通过门控机制控制信息的存储和遗忘，使得重要信息能够长时间保留，而无关信息可被丢弃。

- 核心结构

LSTM 由多个 LSTM 单元组成，每个单元在时间步之间传递信息。每个 LSTM 单元又由三个门组成：

遗忘门f_t：决定丢弃多少过去的信息；

输入门i_t：决定加入多少新的信息；

输出门o_t：决定当前单元的输出；

此外还有：

细胞状态C_t：存储长期记忆信息；

隐藏状态h_t：传递当前时间步的信息给下一时间步。

- 公式

设输入x_t，前一时间步的隐藏状态h_{t-1}和细胞状态C_{t-1}，则 LSTM 计算如下：

⑴ 遗忘门：

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$

若f_t接近于 0，则遗忘更多；接近于 1，则保留更多；

⑵ 输入门：

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C * [h_{t-1}, x_t] + b_C)$$

最终的新信息：

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

其中$\tilde{C}_t$是候选细胞状态，经过i_t选择性加入C_t；

⑶ 输出门：

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

这里h_t既是输出，也是下个时间步的输入。

- 优势

⑴ 由于遗忘门的存在，避免了梯度消失问题；

⑵ 相比于普通 RNN，LSTM 能更好地记住过去的重要信息，学习长期依赖；

⑶ 门控机制更灵活，可控制何时记住，何时忘记和何时输出。

1.3 GRU 模型

GRU 是 RNN 的一种变体，类似于 LSTM，但结构更简单，计算效率更高，适用于处理长序列任务。

- 核心结构

GRU 通过两个门控制信息流动：

⑴ 更新门：决定当前时间步的信息有多少应该保留，有多少应该用新信息替换；类似于 LSTM 输入门和遗忘门的结合体；

⑵ 重置门：控制前一时刻的记忆有多少需要遗忘，帮助模型选择性地丢弃旧信息。

- 计算流程

⑴ 计算更新门：

$$z_t = \sigma(W_z * [h_{t-1}, x_t])$$

其中，z_t控制旧信息h_{t-1}和新信息x_t的结合比例；

⑵ 计算重置门：

$$r_t = \sigma(W_r * [h_{t-1}, x_t])$$

其中，r_t控制前一时刻h_{t-1}在当前时刻的影响；

⑶ 计算新的候选状态$\tilde{h}_t$：

$$\tilde{h}_t = \tanh\left(W * [r_t \odot h_{t-1}, x_t]\right)$$

其中，$\tilde{h}_t$是当前时刻的候选隐藏状态；

⑷ 计算最终的隐藏状态h_t：

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

其中，z_t决定旧状态h_{t-1}和新候选状态$\tilde{h}_t$的融合比例。

- 优势

⑴ 相比于 LSTM，GRU 只有两个门，参数更少，计算效率更高；

⑵ 由于更新门的作用，GRU 在某些任务上能比 LSTM 更有效地学习长序列信息，更易捕捉长期依赖；

⑶ 由于参数较少，GRU 更容易在小数据集上表现良好，不易过拟合。

2. 诗歌的生成过程

2.1 数据处理

```
def process_dataset(fileName):
    examples = []
    # with open(fileName, 'r') as fd:
    with open(fileName, 'r', encoding='utf-8') as fd:
        for line in fd:
            outs = line.strip().split(':')
            content = ''.join(outs[1:])
            ins = [start_token] + list(content) + [end_token]
            if len(ins) > 200:
                continue
            examples.append(ins)

    counter = collections.Counter()
    for e in examples:
        for w in e:
            counter[w]+=1

    sorted_counter = sorted(counter.items(), key=lambda x: -x[1])  # 排序
    words, _ = zip(*sorted_counter)
    words = ('PAD', 'UNK') + words[:len(words)]
    word2id = dict(zip(words, range(len(words))))
    id2word = {word2id[k]:k for k in word2id}

    indexed_examples = [[word2id[w] for w in poem]
                        for poem in examples]
    seqlen = [len(e) for e in indexed_examples]

    instances = list(zip(indexed_examples, seqlen))

    return instances, word2id, id2word
```

首先读取诗歌文本数据，去掉标点、空格等无关字符；然后为每首诗添加

句子起始和终止标记，并丢弃长度>200 的样本；构建词汇表 word2id 和 id2word；最后将诗歌文本转换为对应的索引序列。

2.2 构造 TensorFlow Dataset

```
def poem_dataset():
    # instances, word2id, id2word = process_dataset('../poems.txt')
    instances, word2id, id2word = process_dataset("poems.txt")

    ds = tf.data.Dataset.from_generator(lambda: [ins for ins in instances],
                                        (tf.int64, tf.int64),
                                        (tf.TensorShape([None]),tf.TensorShape([])))
    ds = ds.shuffle(buffer_size=10240)
    ds = ds.padded_batch(100, padded_shapes=(tf.TensorShape([None]),tf.TensorShape([])))
    ds = ds.map(lambda x, seqlen: (x[:, :-1], x[:, 1:], seqlen-1))
    return ds, word2id, id2word
```

将处理好的数据转化为 TensorFlow 的 tf.data.Dataset，然后进行以下操作：

⑴ 打乱数据，增加随机性；

⑵ 进行批量化处理；

⑶ 创建输入-标签对$(x, y, seqlen)$，其中x为诗歌输入（去除最后一个词），y为目标输出（去除第一个词），$seqlen$记录原始序列长度。

2.3 模型构建

2.3.1 RNN 模型

```
class myRNNModel(keras.Model):
    def __init__(self, w2id):
        super(myRNNModel, self).__init__()
        self.v_sz = len(w2id)
        # self.embed_layer = tf.keras.layers.Embedding(self.v_sz, 64,
        #                                              batch_input_shape=[None, None])
        self.embed_layer = tf.keras.layers.Embedding(input_dim=self.v_sz, output_dim=64)

        self.rnncell = tf.keras.layers.SimpleRNNCell(128)
        self.rnn_layer = tf.keras.layers.RNN(self.rnncell, return_sequences=True)
        self.dense = tf.keras.layers.Dense(self.v_sz)
```

定义了 myRNNModel，包括 ⑴ 词嵌入层：将词 ID 映射到 64 维的词向量表示；⑵ RNN 层：采用 128 维隐藏单元的循环神经网络；⑶ 全连接层：用于预测下一个词；

2.3.2 前向传播

```
def call(self, inp_ids):
    '''
    此处完成建模过程，可以参考Learn2Carry
    '''
    # 嵌入输入
    inp_emb = self.embed_layer(inp_ids)
    rnn_out = self.rnn_layer(inp_emb)
    logits = self.dense(rnn_out)
    return logits
```

输入的词 ID 经过 Embedding 层转换成词向量；RNN 层依次计算每个时间步

的隐藏状态；最后经过全连接层计算 logits，作为下一个词的预测结果；

2.3.3 生成下一个词

```
def get_next_token(self, x, state):
    '''
    shape(x) = [b_sz,]
    '''

    inp_emb = self.embed_layer(x) #shape(b_sz, emb_sz)
    h, state = self.rnncell.call(inp_emb, state) # shape(b_sz, h_sz)
    logits = self.dense(h) # shape(b_sz, v_sz)
    out = tf.argmax(logits, axis=-1)
    return out, state
```

首先查找当前单词的其词嵌入；计算 RNN 隐藏状态；然后通过全连接层计算 logits；最后使用 tf.argmax 选出概率最高的下一个词，作为下一个生成的词。

2.4 模型训练

2.4.1 损失计算

```
@tf.function
def compute_loss(logits, labels, seqlen):
    losses = tf.nn.sparse_softmax_cross_entropy_with_logits(
            logits=logits, labels=labels)
    losses = reduce_avg(losses, seqlen, dim=1)
    return tf.reduce_mean(losses)
```

计算交叉熵损失，并按序列长度归一化损失，防止较长的诗歌影响损失计算；

2.4.2 训练过程

```
def train_one_step(model, optimizer, x, y, seqlen):
    '''
    完成一步优化过程，可以参考之前做过的模型
    '''
    with tf.GradientTape() as tape:
        logits = model(x)  # 前向传播，获得预测结果
        loss = compute_loss(logits, y, seqlen)  # 计算损失

    gradients = tape.gradient(loss, model.trainable_variables)  # 计算梯度
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))  # 更新参数

    return loss

def train(epoch, model, optimizer, ds):
    loss = 0.0
    accuracy = 0.0
    for step, (x, y, seqlen) in enumerate(ds):
        loss = train_one_step(model, optimizer, x, y, seqlen)

        if step % 500 == 0:
            print('epoch', epoch, ': loss', loss.numpy())

    return loss
```

训练过程主要分为以下几步：

⑴ 前向传播计算模型输出 logits；

⑵ 计算损失；

⑶ 反向传播更新模型参数；

⑷ 每 500 步打印一次损失值；

其中，使用 Adam 优化器，学习率设为 0.0005，训练 20 轮次。

2.5 诗歌生成

2.5.1 生成随机诗句

```
def gen_sentence():
    state = [tf.random.normal(shape=(1, 128), stddev=0.5), tf.random.normal(shape=(1, 128), stddev=0.5)]
    cur_token = tf.constant([word2id['bos']], dtype=tf.int32)
    collect = []
    for _ in range(50):
        cur_token, state = model.get_next_token(cur_token, state)
        collect.append(cur_token.numpy()[0])
    return [id2word[t] for t in collect]
print(''.join(gen_sentence()))
```

首先初始化隐藏状态；然后以“bos”作为起始词，逐步调用 get_next_token()预测下一个字；生成 50 个字后返回，拼接成诗歌；

2.5.2 生成指定开头的诗

```
def generate_poem(begin_word='日', max_length=50):
    if begin_word not in word2id:
        print(f"警告: '{begin_word}' 不在词典中，使用默认起始词 'bos'")
        begin_word = 'bos'

    # 初始化RNN
    state = [tf.random.normal(shape=(1, 128), stddev=0.5)]
    cur_token = tf.constant([word2id[begin_word]], dtype=tf.int32)
    generated = [begin_word]

    # 逐步生成诗歌
    for _ in range(max_length):
        cur_token, state = model.get_next_token(cur_token, state)
        next_word = id2word[cur_token.numpy()[0]]
        if next_word == 'eos':
            break
        generated.append(next_word)

    return ''.join(generated)
```

首先检查输入词是否在词典，不在则默认使用“bos”作为起始词；然后初始化 RNN 状态；逐步调用 get_next_token()生成诗句，直到遇到“eos”或达到最大长度停止；最后拼接生成的词，组成完整的诗句；

2.5.3 以多个字为起点生成诗歌

```
begin_words = ['日', '红', '山', '夜', '湖', '海', '月']
for word in begin_words:
    print(f"以 '{word}' 开头的诗歌: {generate_poem(word)}")
```

遍历 begin_words 列表，依次以“日、红、山、夜、湖、海、月”为起点，调用 generate_poem()生成不同的诗歌。

3. 生成的诗歌截图

```
以 '日' 开头的诗歌: 日暮霞。
以 '红' 开头的诗歌: 红鹂。
以 '山' 开头的诗歌: 山寺，风吹落日斜。
以 '夜' 开头的诗歌: 夜夜看惆怅无人事。
以 '湖' 开头的诗歌: 湖水晚风吹。
以 '海' 开头的诗歌: 海滨谊宙砝珀炅裟裟裟裟裟裟裟裟裟裟裟杖裟噇裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟裟
以 '月' 开头的诗歌: 月霏霏霏霏霏霏。
```

4. 总结

本次实验，我成功实现了基于 RNN 模型的诗歌生成任务，并探索了 RNN 在自然语言生成中的应用。通过构建词嵌入层、循环神经网络层以及全连接层，模型能够从给定的起始词逐步生成连贯的诗歌。训练过程中，模型通过交叉熵损失优化，并利用 Adam 优化器进行参数更新，使其能够学习诗歌的语言规律和句法结构。

尽管 RNN 具备一定的文本生成能力，但其在长序列依赖问题上仍存在局限性，可能导致诗歌内容缺乏全局一致性。因此未来可以考虑引入 LSTM 或 GRU 模型以增强长期依赖建模能力。此外，还可以结合 Transformer 结构，进一步提升文本生成质量，使诗歌生成更加自然、流畅。