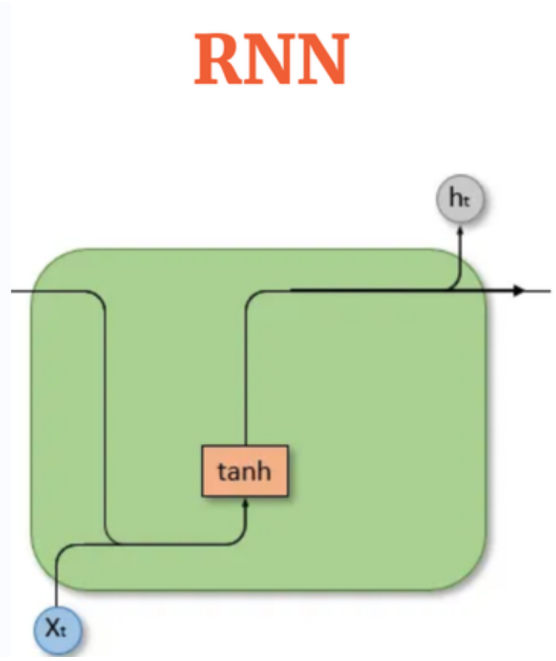
NNDL Homework 4

> 作业主要内容：
>
> 1. 补全程序，主要是前面的3个空和 生成诗歌的一段代码。(tensorflow) [pytorCH 需要补全 对应的 rnn.py 文件中的两处代码]（pytorch 和 tensorflow 两个版本中任意选择一个即可。）
> 2. 解释一下 RNN ， LSTM，GRU模型，
> 3. 叙述一下 这个诗歌生成的过程。
> 4. 生成诗歌 开头词汇是“日 、 红 、 山 、 夜 、 湖、 海 、 月”，等词汇作为begin word,把生成的截图放到报告里面。[pytorch 版本 需要放一张 训练时候的截图。]

RNN, LSTM, GRU模型介绍

RNN（循环神经网络）

RNN是处理序列数据的最基础的神经网络结构。它通过在每个时间步重复使用相同的权重，来处理任意长度的序列。RNN的核心思想是使用隐藏状态（h_t），这个隐藏状态包含了之前时间步的信息。



- **数学公式**：

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

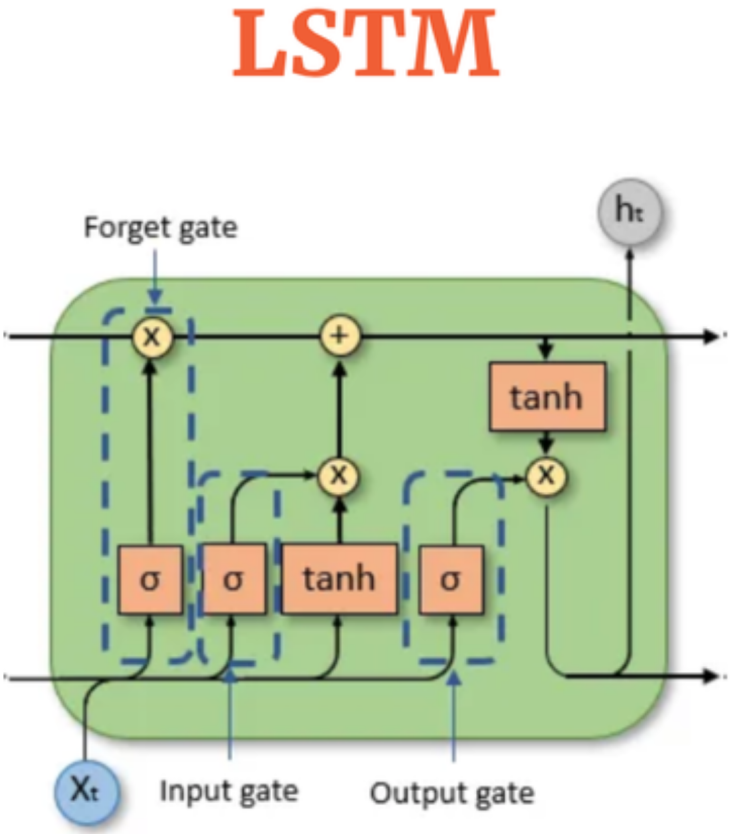
$$y_t = W_{hy}h_t + b_y$$

其中，h_t是当前时间步的隐藏状态，x_t是当前输入，y_t是当前输出，W和b分别是权重矩阵和偏置项。

- **优点**：简单，易于理解和实现。
- **缺点**：梯度消失或梯度爆炸问题，难以学习长期依赖关系。

LSTM（长短期记忆网络）

LSTM通过引入三个门（输入门i_t、遗忘门f_t和输出门o_t）和单元状态c_t，克服了RNN的梯度消失问题，能够学习长期依赖关系。



- **数学公式**：

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$h_t = o_t \odot \tanh(c_t)$$

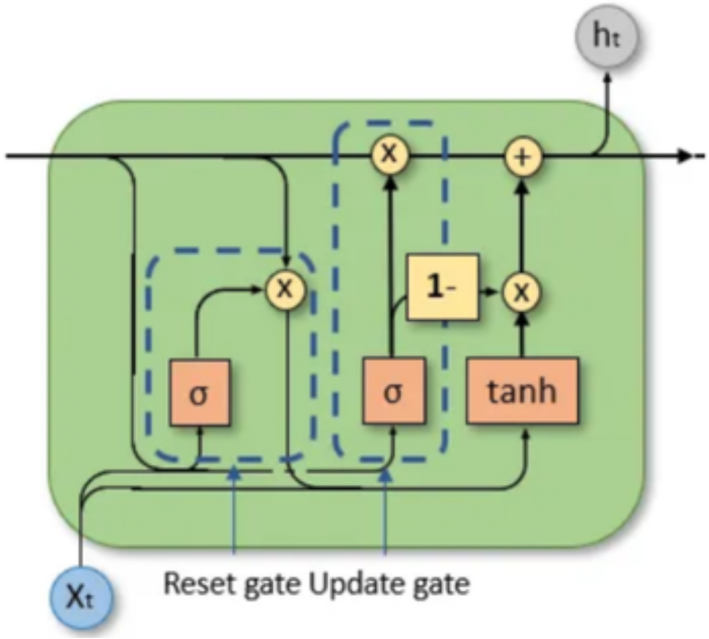
  其中，σ是sigmoid激活函数，$\odot$表示Hadamard乘积（元素相乘）。

- **优点**：能有效地处理长期依赖问题，被广泛应用于各种序列任务中。
- **缺点**：相对RNN，LSTM有更多的参数，计算上更复杂，训练时间更长。

GRU（门控循环单元）

GRU是LSTM的一种变体，它通过合并遗忘门和输入门到一个更新门z_t，以及将单元状态和隐藏状态合并，简化了模型结构。

GRU



- **数学公式**：

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

$$\widetilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$

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

  这里，z_t是更新门，控制前一隐藏状态h_{t-1}对当前隐藏状态h_t的影响；r_t是重置门，控制是否允许前一隐藏状态影响候选隐藏状态$\widetilde{h}_t$。

- **优点**：与LSTM相比，计算效率更高，参数更少，但在很多任务中能够提供与LSTM相当的性能。
- **缺点**：虽然简化了模型结构，但在某些复杂问题上，可能不如LSTM表现出色。

比较

- **性能**：LSTM和GRU通常能够比标准RNN更好地处理长序列和捕捉长期依赖关系，但在特定任务上哪种模型更优取决于具体问题和数据。
- **计算复杂度**：RNN < GRU < LSTM，LSTM和GRU由于其复杂的门控机制，在计算和参数量上都高于标渐RNN。
- **使用场景**：对于需要捕捉长期依赖的任务，LSTM和GRU更为合适。对于较短序列或者对模型复杂度有限制的场景，简单的RNN或许更加高效。

诗歌生成过程

数据处理

1. **读取数据**：首先从文件中读取诗歌数据，每行一个诗句。数据预处理包括添加开始和结束标记（`bos`和`eos`），这有助于模型学习诗句的开始和结束。
2. **构建词汇表**：通过统计所有诗句中出现的字符，构建一个词汇表。词汇表中每个字符都将被分配一个唯一的ID，ID即字符在所有字符中出现的顺序。
3. **转换诗句**：将每个诗句转换为字符ID的序列。这一步骤使得模型能够处理文本数据。
4. **生成数据集**：构建TensorFlow数据集，该数据集输出三个元素：输入序列ds、目标序列（输入序列向右移动一位）和序列长度。数据集还会被打乱和批量化。

模型定义

1. **定义模型**：模型使用嵌入层将字符ID转换为密集向量，这些向量随后被送入RNN层处理。RNN层能够捕捉序列中的时序依赖关系。最后，一个全连接层用于预测下一个字符的概率分布。
2. **前向传播**：在模型的`call`方法中，实现了前向传播的逻辑，即如何根据输入序列生成对下一个字符的预测。

训练过程

1. **计算损失**：使用序列损失函数计算模型输出和目标序列之间的差异。损失函数会考虑实际序列长度，以忽略填充的影响。
2. **梯度下降**：通过反向传播算法计算模型参数的梯度，并使用优化器（如Adam）更新权重，以最小化损失函数。

生成过程

1. **生成诗句**：一旦模型被训练，可以使用它来生成新的诗句。生成过程从开始标记`bos`开始，模型预测下一个字符，直到预测到结束标记`eos`或达到最大长度。
2. **采样**：在每一步生成字符时，根据模型预测的概率分布采样下一个字符。这里使用的是贪婪采样，即直接选择最高概率的字符。
3. **状态传递**：在生成过程中，RNN的隐藏状态被传递到下一个时间步，以保持序列的上下文信息。

诗歌生成

```
def generate_poem_with_begin_word(begin_word):
    if begin_word not in word2id:
        print(f"给定的开始词 '{begin_word}' 不在词汇表中。")
        return []

    state = [tf.random.normal(shape=(1, 128), stddev=0.5), tf.random.normal(shape=(1, 128), stddev=0.5)]
    cur_token = tf.expand_dims(tf.constant([word2id[begin_word]], dtype=tf.int32), 0)
    collect = [begin_word]

    for _ in range(50):  # 可调整生成的长度
        cur_token, state = model.get_next_token(cur_token, state)
        cur_word_id = cur_token.numpy()[0][0]  # 获取生成的词汇ID
        cur_word = id2word[cur_word_id]
        if cur_word == end_token:  # 如果生成结束标记，则停止生成
            break
        collect.append(cur_word)
        cur_token = tf.expand_dims(tf.constant([cur_word_id], dtype=tf.int32), 0)

    return ''.join(collect)

begin_words = ["日", "红", "山", "夜", "湖", "海", "月"]
for word in begin_words:
    poem = generate_poem_with_begin_word(word)
    print(poem)
```

[28] ✓ 0.0s

```
日暮云雨。
红霞里。
山上月，云声满水，云里无人不可知。
夜，何处不知何处。
湖上江边，日日无人。
海风雨满山中。
月上春，一夜风声过。
```