# RNN 循环神经网络 2（1vN、Nv1、Seq2Seq 及简单Attention）


> **Author**：Zhouqi Hua, Tongji University
>
> **Email**：henryhua0721@foxmail.com

>
> **Date**：2024/5/13
>
> **Code**：`RNN_Seq2Seq.py`

## 🌟 一句话模型总结

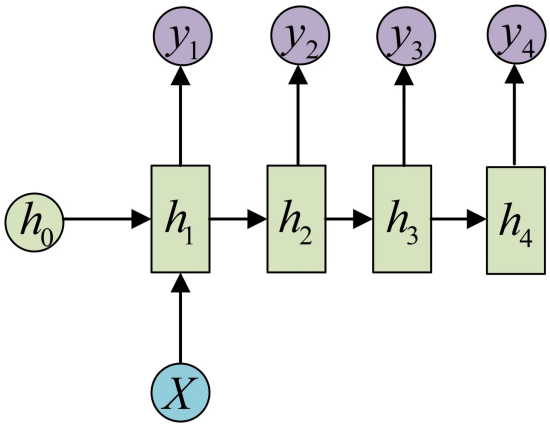
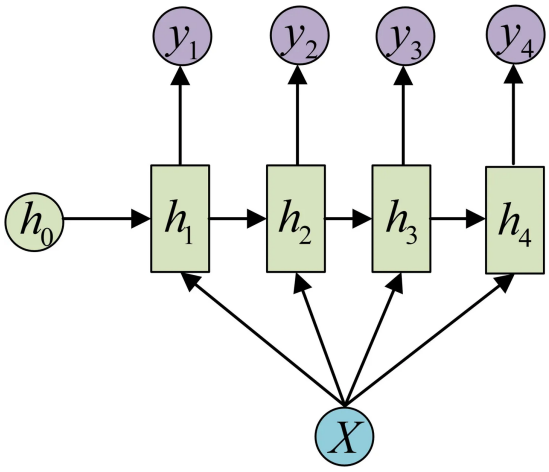
传统的 `RNN` 为 **N v N** 模式（输入序列长度为 `N`，输出长度对应为 `N`），对于不同问题可能出现 **1 v N** / **N v 1** / **N v M**（即 `Seq2Seq` 情况，引出 `Encoder-Decoder` 结构）以及简单的 `Attention` 机制。

## 🌟 模型架构解释

### 1 v N

对应**单个元素的输入，输出为序列**的情况。

对于单个输入的位置，可以做以下分类：

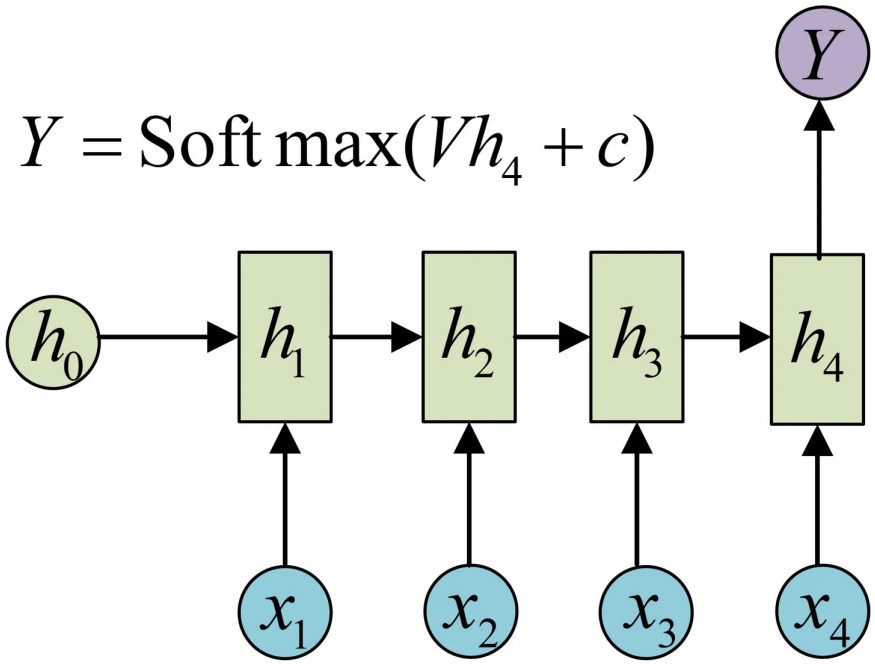
| 输入在序列开始 | 输入在序列每个阶段 |
| --- | --- |
|  |  |

常见的应用：

- 看图写话（**image caption**）
  - $X$：单一的图像特征
  - $Y$：输出的文本序列
- 从类别生成语音或音乐
  - $X$：单一类别特征
  - $Y$：输出的语音序列或音乐

### N v 1

对应输入为一个序列，输出为单独的值的情况。




常见应用集中于序列分类、评估等问题：

- 输入一段文字判别它所属的类别
- 输入一个句子判断其情感倾向
- 输入一段视频并判断它的类别

### N v M (Seq2Seq)

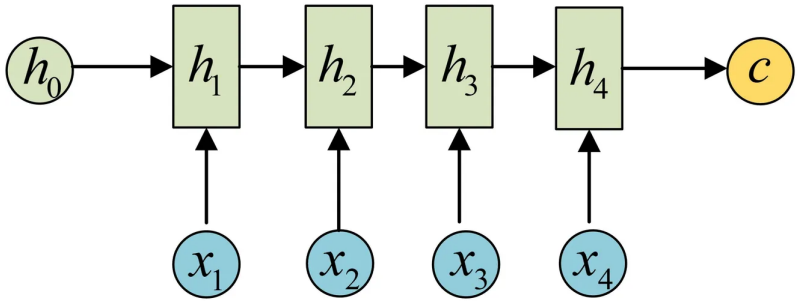
RNN 应用中最为常见的情况。大多数情况下无法保证输入序列和输出序列的长度一致（如机器翻译任务等），因此 `Seq2Seq` 模式通常可以分解为 **N v M = N v 1 + 1 v M**，使用隐变量 $c$ 代表中间的 `1` 状态。

前半阶段（**N v 1**）最常用的方法是把**最后一个隐状态赋值给** $c$（或做一定变换）：

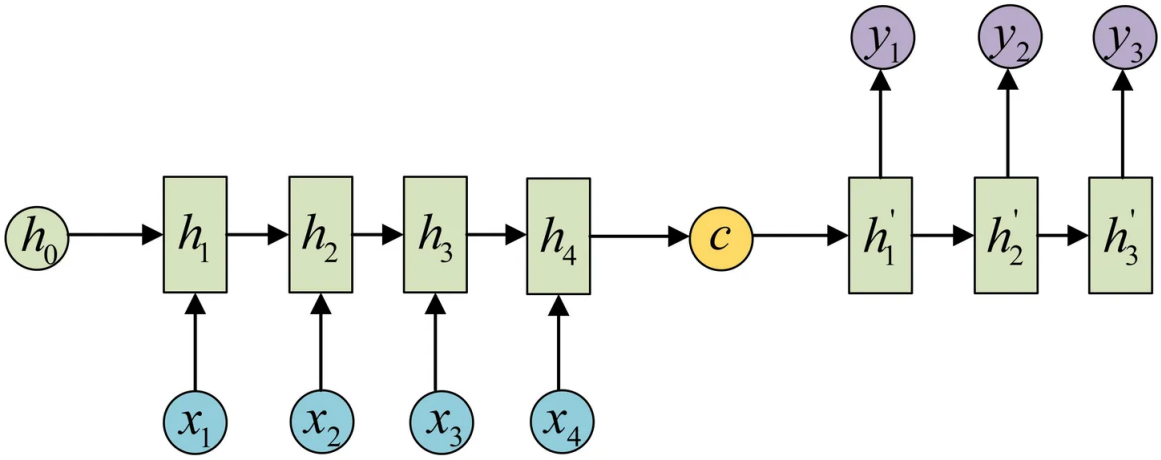
$$（1）\ c = h_4$$

$$（2）\ c = q(h_4)$$

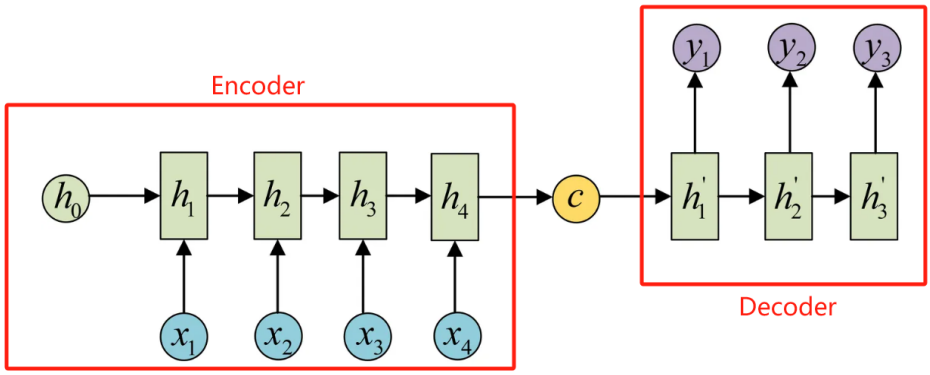
$$（3）\ c = q(h_1, h_2, h_3, h_4)$$



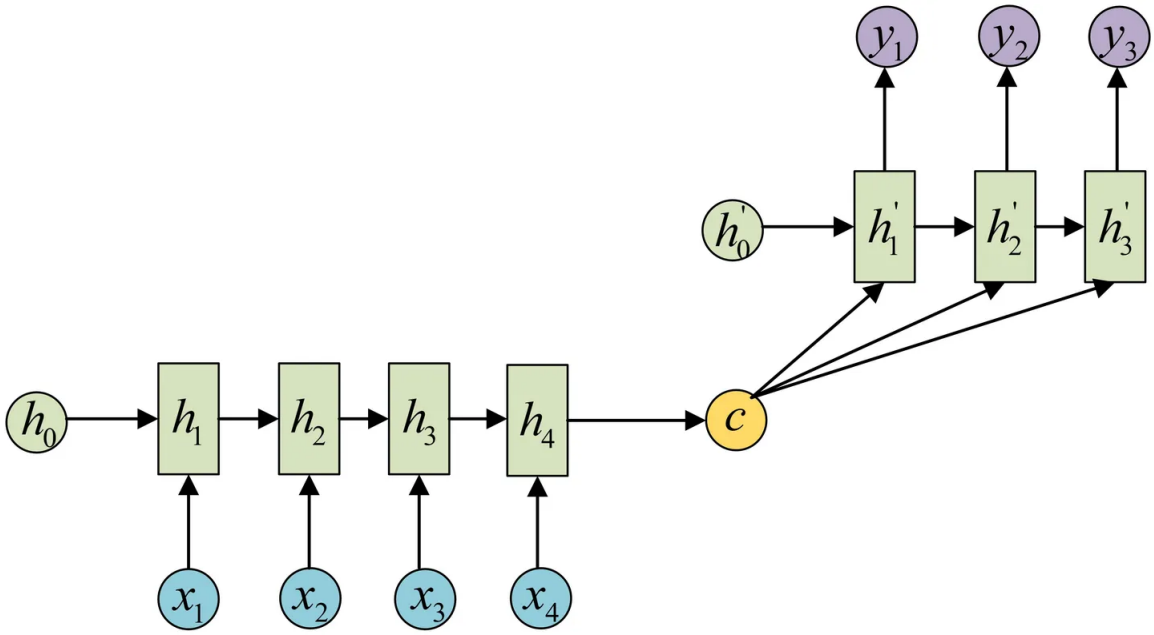

后半阶段（**1 v M**）使用另一个 `RNN` 进行处理，输出一个序列：




上述 `Seq2Seq` 模型也被称为 `Encoder-Decoder` 模式：




当然也可以将 $c$ 作为 `Decoder` 每一阶段的输入：

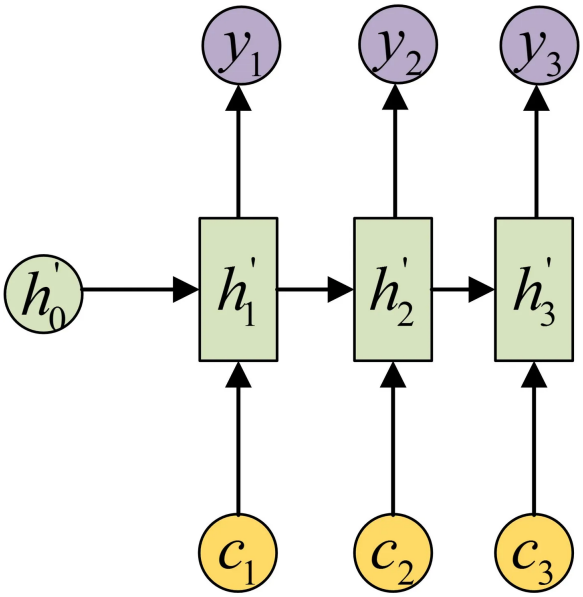



常用的应用有：

- 机器翻译
- 文本摘要
- 阅读理解
- 语音识别
- 文本生成

### 简单 Attention 机制

上述 `Seq2Seq` 中使用单变量 $c$ 可能会导致**性能瓶颈**（集成能力不足，导致输入序列较长时输出精度下降），因此在 `Decoder` 中对于不同的时间步使用不用的 $c$。




因此每一个 $c$ 会**自动选取**与当前所要输出的 $y$ 最合适的上下文信息。

具体来说，使用 $a_{ij}$ 衡量 `Encoder` 中第 $j$ 阶段的 $h_j$ 和解码时第 $i$ 阶段的相关性，最终 `Decoder` 中第 $i$ 阶段的输入的上下文信息 $c_i$ 就来自于所有 $h_j$ 对 $a_{ij}$ 的加权和。这里的 $a_{ij}$ 就是**注意力机制**的雏形（表示**关注的多少**）。

## 🌟 模型优缺点

针对 `Seq2Seq` 模型分析：

- 优点
  - **变长序列处理**：能够处理变长序列，对于不定长序列有很高的适应性
  - **序列生成问题**：可用于各种序列生成问题，如机器翻译、对话系统等
- 缺点
  - **训练复杂度**：训练时间长，因为必须处理整个序列。
  - **显存需求高**：由于解码器需要根据编码器的状态来生成输出，因此编码器的状态必须被保存在内存中，导致易爆显存。

## 🌟 代码解析

针对 `Seq2Seq` 的代码分析：

### 导入模块

```python
import torch
import torch.nn as nn
import torch.optim as optim
```

## 定义Encoder模型

```
# 定义Encoder模型
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)     # view(1, 1, -1) 为了匹配 RNN 的输入
        output = embedded                                   # output 为了和 Decoder 的输入匹配
        output, hidden = self.rnn(output, hidden)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, 1, self.hidden_size)
```

## 定义Decoder模型

```
# 定义Decoder模型
class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.rnn = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)       # view(1, 1, -1) 为了匹配 RNN 的输入
        output = torch.relu(output)                         # relu 作为激活函数
        output, hidden = self.rnn(output, hidden)           # output 为了和 Encoder 的输出匹配
        output = self.softmax(self.out(output[0]))          # output 为了和目标匹配
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, 1, self.hidden_size)
```

## 定义Seq2Seq整体模型

```
# 定义Seq2Seq模型
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
```

```
        self.decoder = decoder

    def forward(self, input, target, teacher_forcing_ratio=0.5):
        target_length = target.size(0)                          # 目标长度
        encoder_hidden = self.encoder.init_hidden()             # 初始化 Encoder
隐藏层

        input_length = input.size(0)                            # 输入长度
        for ei in range(input_length):                          # Encoder 阶段
            _, encoder_hidden = self.encoder(input[ei], encoder_hidden) #
Encoder 隐藏层

        decoder_input = torch.tensor([[START_TOKEN]])           # 开始标记
        decoder_hidden = encoder_hidden                         # Decoder 隐藏层初
始化

        use_teacher_forcing = True if random.random() <
teacher_forcing_ratio else False

        if use_teacher_forcing:
            for di in range(target_length):
                decoder_output, decoder_hidden = self.decoder(
                    decoder_input, decoder_hidden)
                decoder_input = target[di]                      # 使用教师强制作为
下一个输入
        else:
            for di in range(target_length):
                decoder_output, decoder_hidden = self.decoder(
                    decoder_input, decoder_hidden)
                _, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze().detach()         # 使用预测作为下一
个输入

        return decoder_output
```

注意这里通过参数 `teacher_forcing_ratio` 设定了一个**"教师强制"模式**，区别于预测模式。

- `教师强制模式`：**解码器的下一个输入是目标序列的当前元素。**这种方式可以加速模型的收敛，但可能会导致训练时和预测时的表现不一致，因为训练时每一步都是正确的输入，而预测时每一步的输入是上一步的输出，可能会有错误。
- `预测模式`：**解码器的下一个输入是解码器自己在当前步骤的输出。**这种方式使得模型在训练和预测时的行为更一致，但可能会导致训练速度较慢。

---

## 🎓 附录

参考：

- 完全图解RNN、RNN变体、Seq2Seq、Attention机制 - 知乎 (zhihu.com)
- 通熟易懂RNN｜RNN与RNN的变种结构｜上 - 知乎 (zhihu.com)
- 详细介绍seq2seq模型，包括它的原理、优点、缺点、公式推导以及与LSTM之间的关系...-CSDN博客

- 2. RNN神经网络模型的不同结构 - hyc339408769 - 博客园 (cnblogs.com)