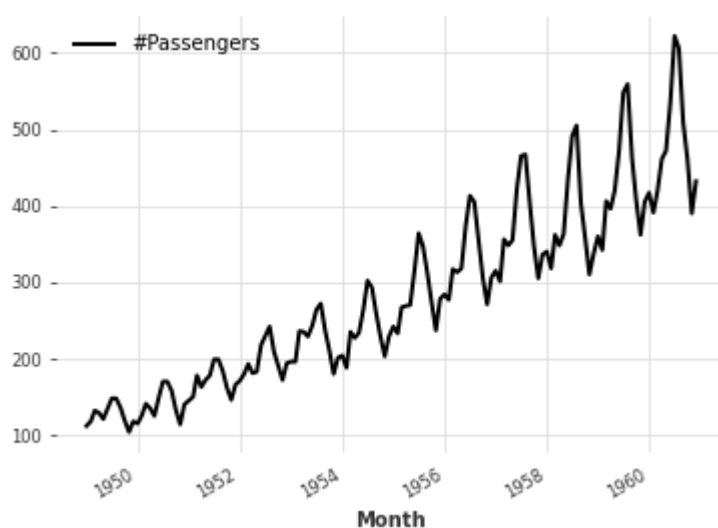


# 循环神经网络-RNN

## 时序序列

时序序列其实就是一组**按时间顺序记录的数值**，它描述的是某个东西在不同时间的变化过程。比如每天的气温、股票的价格、一个店铺每天的客流量等。这些数据按照时间先后排列，显示了它们随着时间的变化趋势。

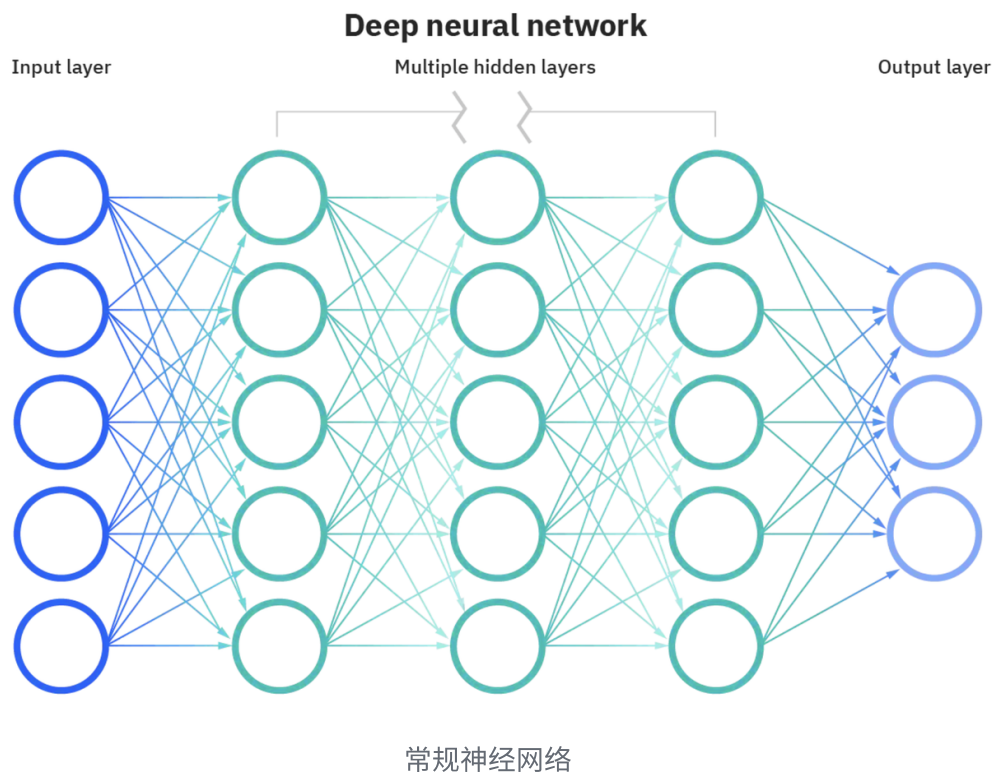
**简单来说**，时序序列就是：**一串按时间排好顺序的数字，告诉我们某件事在时间上的变化规律**。通过分析这些数字，我们能发现它是否有上升、下降的趋势，或者有规律的波动，甚至是一些意料之外的异常变化。



每月航班乘客数

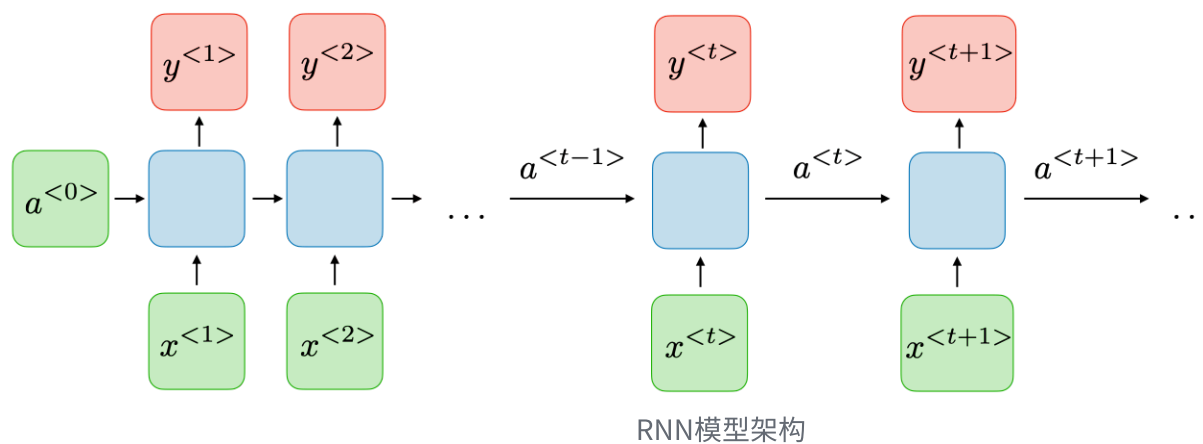
其实语言也是一种时序序列，人们渴望对语言模型进行建模完成翻译、分类等任务，于是RNN应运而生

## RNN-最初的循环神经网络



## 模型架构

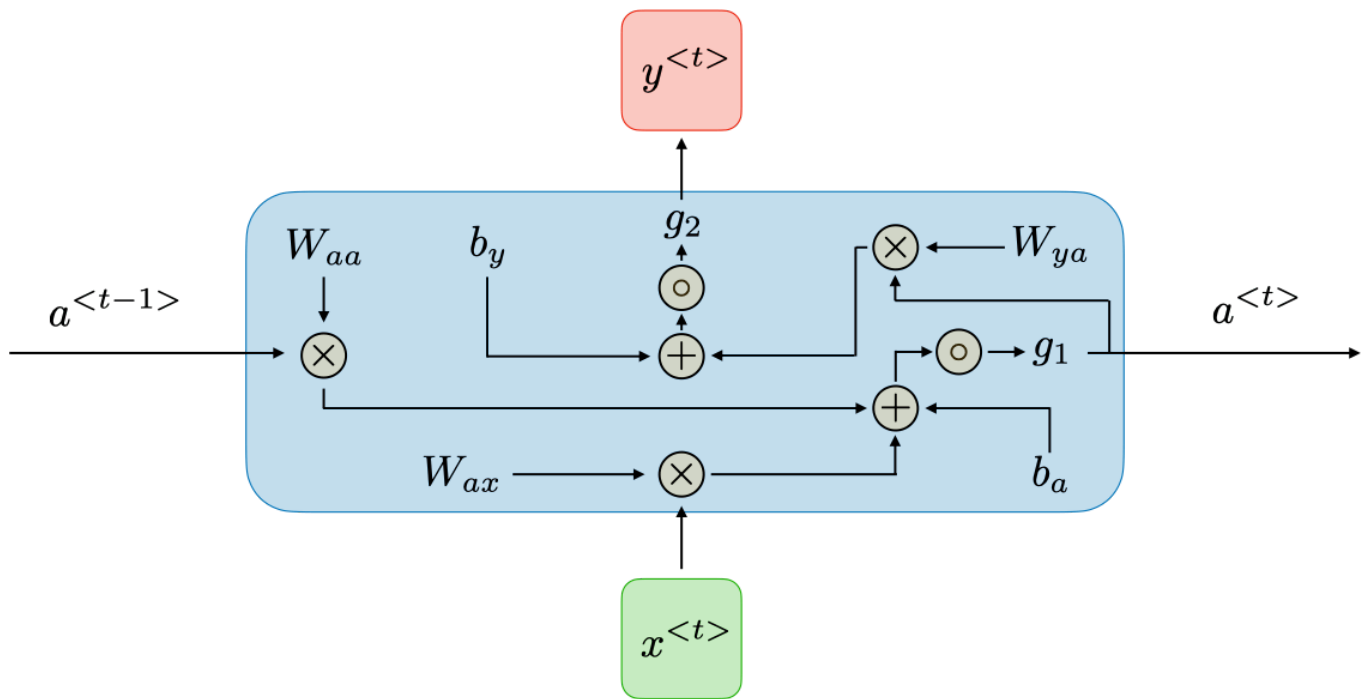
传统的循环神经网络（Recurrent Neural Networks，简称 RNN）是一类允许将前一个输出作为输入并保留隐藏状态的神经网络。它们的典型结构如下：



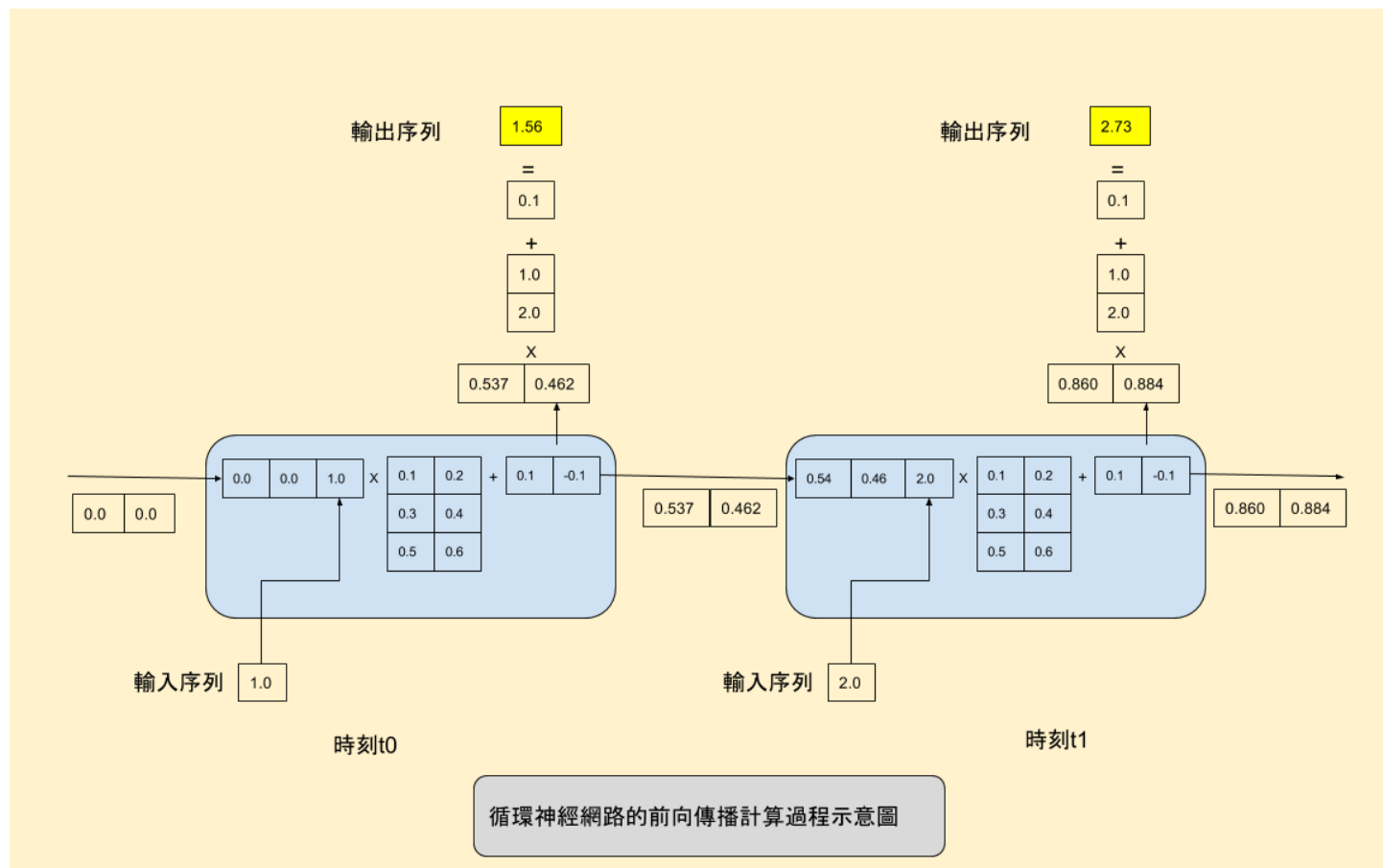
对于每个**时间步t**，激活值  $a^{<t>}$ （hidden）和输出  $y^{<t>}$ （output）的表达式如下：

$$a^{<t>} = g_1(W_{aa}a^{t-1} + W_{ax}x^{<t>} + b_a) \text{ and } y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

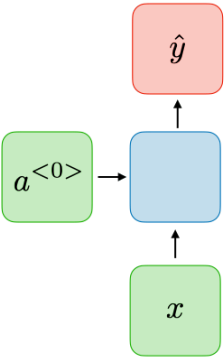
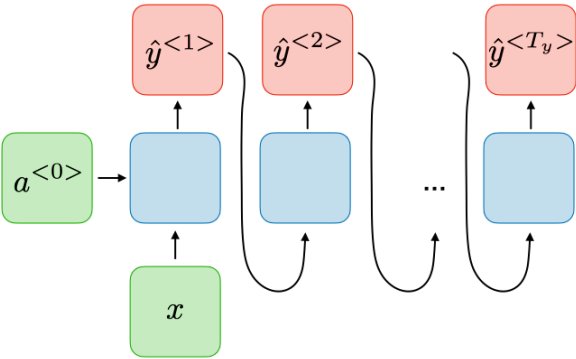
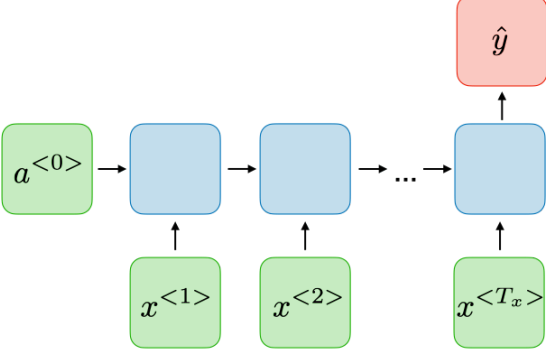
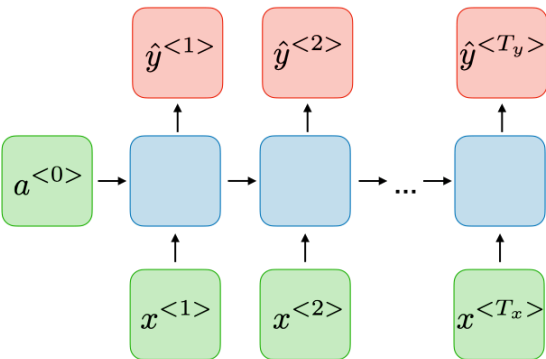
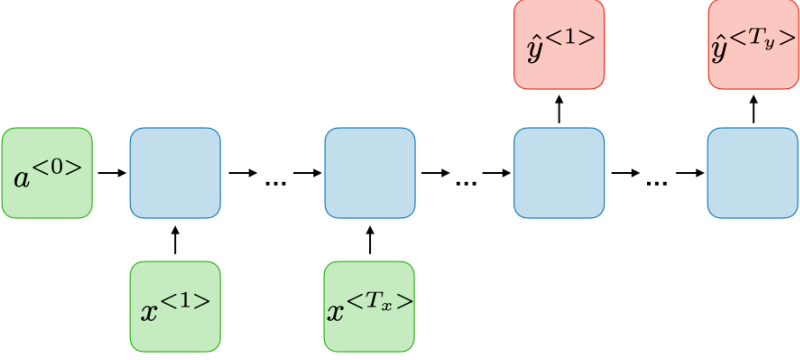
$W_{ax}, W_{aa}, W_{ya}, b_a, b_y$  是共享的参数矩阵，而  $g_1, g_2$  是激活函数



## 微观运算流程



RNN种类	架构图	例子
One-to-one		传统神经网络

		
One-to-many		生成模型
Many-to-one		分类模型
Many-to-many		NER命名实体识别
Many-to-many		机器翻译

# RNN的优缺点如下

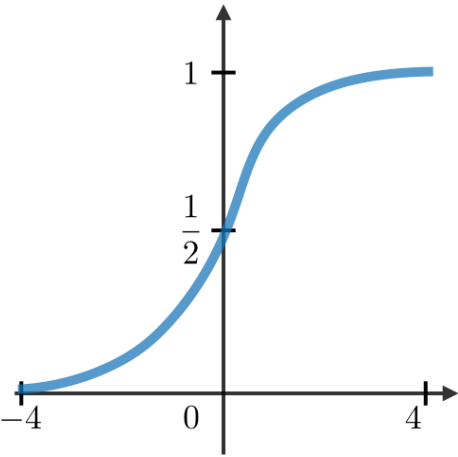
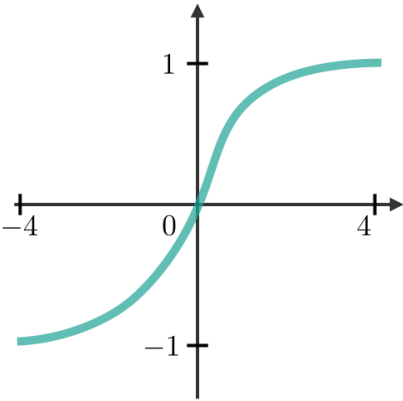
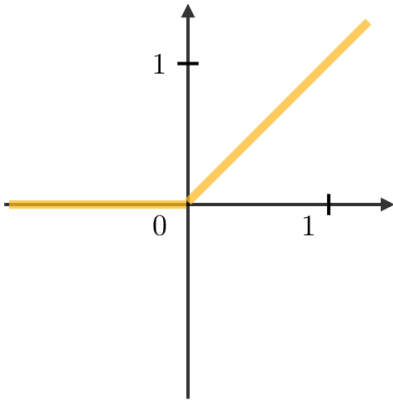
优点	缺点
<ul style="list-style-type: none"><li>可以处理任意长度的输入</li><li>模型大小不随输入大小而增加</li><li>计算过程会考虑历史信息</li><li>权重在时间步长上共享</li></ul>	<ul style="list-style-type: none"><li>计算速度较慢</li><li>难以获取较长时间前的信息</li><li>当前状态无法考虑任何未来的输入</li></ul>



## RNN最大问题：难以解决长距离依赖

为何会导致这一问题？

RNN中常使用的激活函数：

sigmoid	Tanh	ReLU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		

在 RNN 中经常会遇到梯度消失和梯度爆炸现象。产生这些现象的原因是，由于梯度的乘法效应，随着层数的增加，梯度可能会呈指数级衰减或增长，因此很难捕捉到长期依赖关系。

# RNN实践

## 利用pytorch从零实现RNN模型

## 导入必要的库

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
```

## 步骤 2：定义 RNN 类

创建一个 RNN 类，并定义它的初始化方法、前向传播方法等。

```
1 class SimpleRNN(nn.Module):
2     def __init__(self, input_size, hidden_size, output_size):
3         super(SimpleRNN, self).__init__()
4         self.hidden_size = hidden_size
5
6         # 定义 RNN 层参数
7         self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
8         self.i2o = nn.Linear(hidden_size, output_size)
9         self.softmax = nn.LogSoftmax(dim=1)
10
11     def forward(self, input, hidden):
12         combined = torch.cat((input, hidden), 1)
13         hidden = self.i2h(combined)
14         output = self.i2o(hidden)
15         output = self.softmax(output)
16         return output, hidden
17
18     def init_hidden(self):
19         # 初始化隐藏层
20         return torch.zeros(1, self.hidden_size)
21
```

## 步骤 3：创建数据和超参数

设定 RNN 的输入、隐藏层大小、输出大小等超参数。这里用简单的随机数据进行演示。

```
1 input_size = 5    # 输入的特征维度
2 hidden_size = 10  # 隐藏层的特征维度
3 output_size = 2   # 输出的特征维度（例如二分类问题）
4
5 rnn = SimpleRNN(input_size, hidden_size, output_size)
6
```

```
7 # 创建随机数据：例如一个序列数据，包含 3 个时间步
8 sequence = [torch.randn(1, input_size) for _ in range(3)] # 3 时间步的序列
9 target = torch.tensor([1]) # 假设目标输出是分类标签 1
```

## 步骤 4：定义损失函数和优化器

使用交叉熵损失函数以及 SGD 优化器。

```
1 criterion = nn.NLLLoss()
2 optimizer = optim.SGD(rnn.parameters(), lr=0.01)
```

## 步骤 5：训练循环

定义一个简单的训练循环。

```
1 # 训练循环
2 num_epochs = 100
3 for epoch in range(num_epochs):
4     rnn.zero_grad() # 清除前一轮的梯度
5     hidden = rnn.init_hidden() # 初始化隐藏层
6
7     # 输入序列的前向传播
8     for input in sequence:
9         output, hidden = rnn(input, hidden)
10
11     # 计算损失
12     loss = criterion(output, target)
13
14     # 反向传播
15     loss.backward()
16     optimizer.step()
17
18     # 打印损失值
19     if epoch % 10 == 0:
20         print(f'Epoch {epoch}, Loss: {loss.item()}')
21
```

## 解释说明

- 1. 数据输入：**在训练过程中，模型接收一个序列，每一个时间步的数据经过前向传播，将输出与更新后的隐藏状态返回。

2. **损失计算**：只在序列的最后一个时间步计算损失。对于“许多对一”模型，通常选择最后一个时间步的输出与真实标签进行对比。
3. **梯度更新**：反向传播的梯度计算和参数更新，通过 `loss.backward()` 和 `optimizer.step()` 完成。

## 直接使用RNN模型

### 导入必要的库

```
1 import torch
2 import torch.nn as nn
```

### 定义RNN结构

RNN模型由输入层、隐藏层和输出层组成。以下是构建RNN的代码示例：

```
1 class SimpleRNN(nn.Module):
2     def __init__(self, input_size, hidden_size, output_size, num_layers=1):
3         super(SimpleRNN, self).__init__()
4
5         # 定义RNN层
6         self.rnn = nn.RNN(input_size, hidden_size, num_layers,
7                             batch_first=True)
8
9         # 定义全连接层，用于将RNN的输出映射到目标输出
10        self.fc = nn.Linear(hidden_size, output_size)
11
12    def forward(self, x):
13        # 初始化隐藏状态
14        h0 = torch.zeros(num_layers, x.size(0), hidden_size)
15
16        # 将输入x传入RNN
17        out, hn = self.rnn(x, h0)
18
19        # 取最后一个时间步的输出，传入全连接层
20        out = self.fc(out[:, -1, :])
21
22        return out
```

这里，`input_size` 表示输入特征的数量，`hidden_size` 表示隐藏层神经元的数量，`output_size` 表示输出层神经元的数量。



## 初始化模型参数

```
1 input_size = 10      # 输入特征的维度
2 hidden_size = 20     # 隐藏层的维度
3 output_size = 1      # 输出的维度（例如二分类任务中为1）
4 num_layers = 2       # RNN的层数
```

## 实例化模型

```
1 model = SimpleRNN(input_size, hidden_size, output_size, num_layers)
```

## 使用模型进行前向传播

假设输入 `x` 是一个张量，形状为 `(batch_size, sequence_length, input_size)`，例如：

```
1 batch_size = 5
2 sequence_length = 7
3 x = torch.randn(batch_size, sequence_length, input_size)
4
5 # 前向传播
6 output = model(x)
7
8 print("Output shape:", output.shape)
```

## 解释各部分含义

- `input_size`：每个时间步输入的特征维度。
- `hidden_size`：隐藏层的维度，决定了隐藏状态的大小。
- `num_layers`：RNN的层数。
- `output_size`：模型输出的维度，常用于映射为实际需要的输出格式。

在这个例子中，模型先将输入传入RNN层得到时间序列的输出，然后选取最后一个时间步的输出，经过全连接层后输出最终结果。

## 常见操作和技巧

- **初始化隐藏状态**：通常在每次前向传播中都需要初始化隐藏状态 `h0`，尤其是在处理独立的数据批次时。

- **选择时间步的输出：**在很多任务中，我们只关心最后一个时间步的输出（例如句子分类任务），但也可以取所有时间步的输出用于不同任务（如机器翻译）。
- **多层RNN：**增加 `num_layers` 可以创建深层RNN，捕捉更复杂的模式。