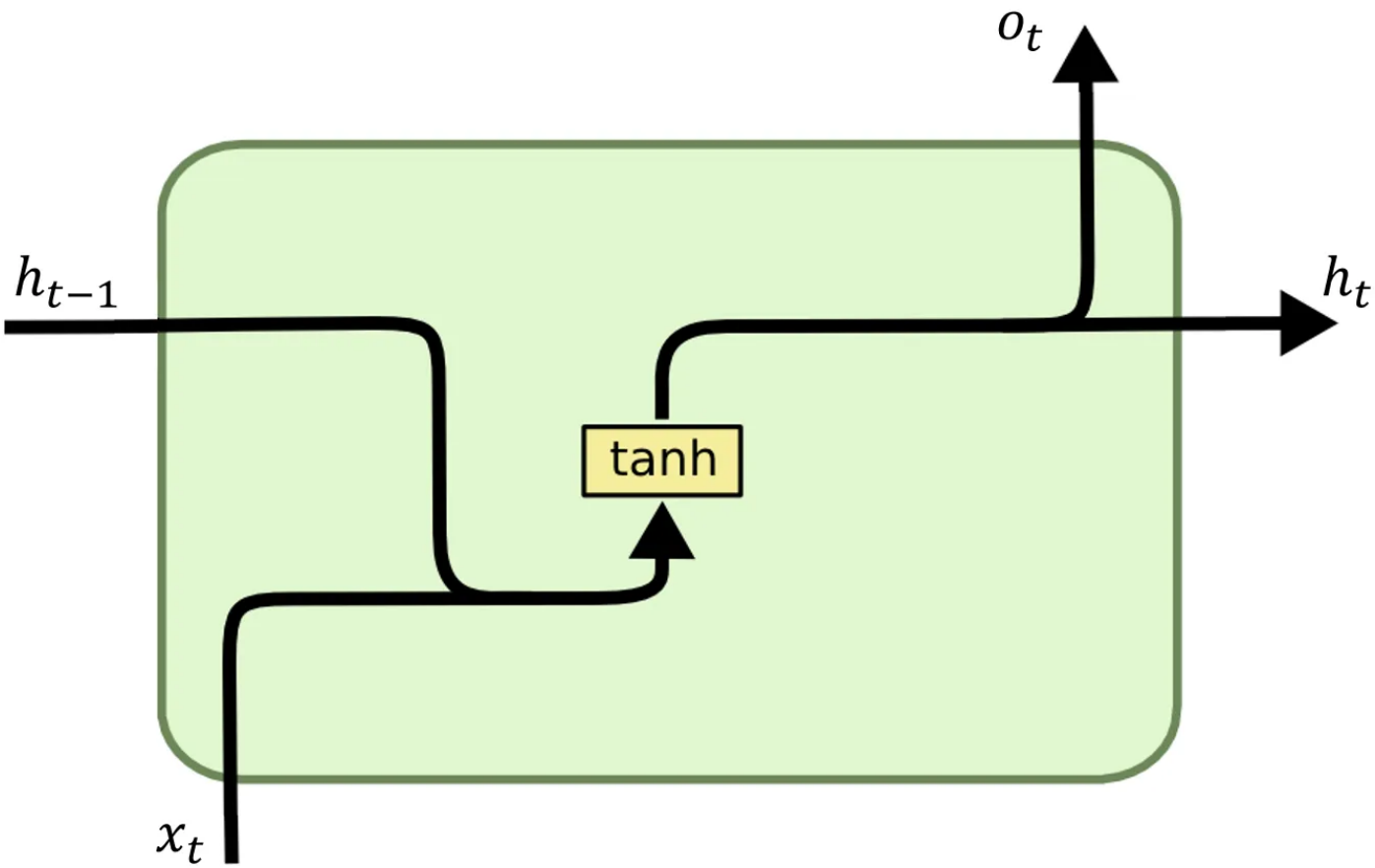
循环神经网络-LSTM

RNN有什么样的问题



梯度消失和梯度爆炸问题

- **梯度消失问题**：在RNN中，由于每个时间步的梯度反向传播都会被相乘，当时间步较多时，反向传播到前面的梯度会逐渐减小，最终变得非常小，导致模型无法有效更新权重。这使得RNN难以捕捉到远距离的信息，只能学习短时间的依赖关系。
- **梯度爆炸问题**：在反向传播中，梯度在多次相乘后也可能变得非常大，导致模型的权重更新过大，引发数值不稳定。RNN在训练长序列时容易出现梯度爆炸问题，需要额外的梯度裁剪（gradient clipping）等技巧来缓解。

难以捕捉长时间依赖

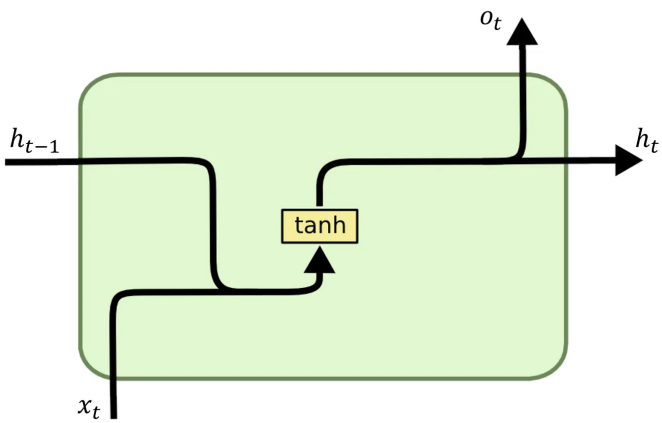
- RNN缺乏有效的机制来记住序列中的长时间依赖信息。它的结构决定了模型只能依赖当前时间步的状态来更新，较难处理需要长时间记忆的任务（例如在语言建模中，早期提到的主题或主语信息在较长序列后续还需要用到）。

无选择性的记忆方式

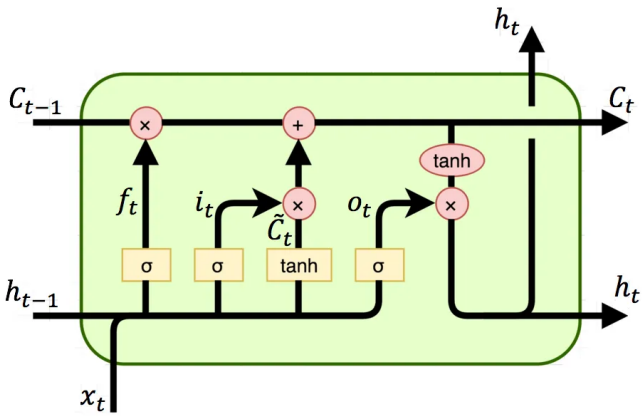
- 在RNN中，所有输入信息都在序列中传播，没有控制信息传递的机制。这种无选择性的记忆方式意味着网络无法主动“记住”或“遗忘”某些信息，这会导致重要信息和无用信息被一视同仁地处理。
- 普通的RNN没有专门的短期和长期记忆结构，所有的历史信息都通过一个单一的隐藏状态来传递，这导致短期信息与长期信息混在一起，容易相互干扰。

LSTM-长短期记忆网络（Long Short-Term Memory）

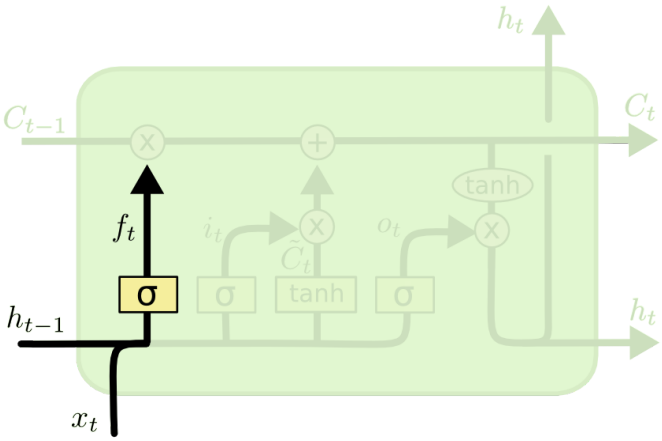
模型架构



RNN

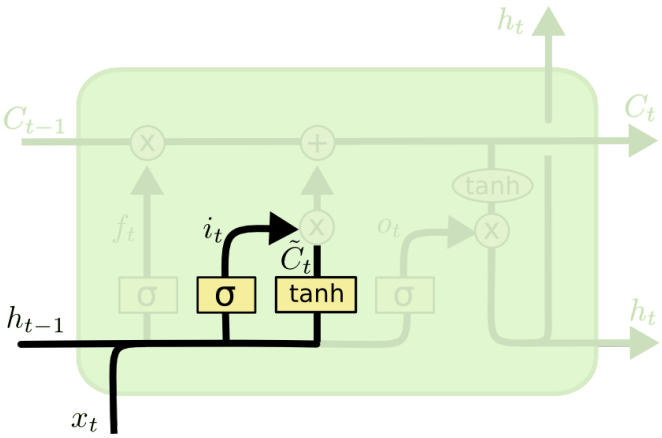


LSTM



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

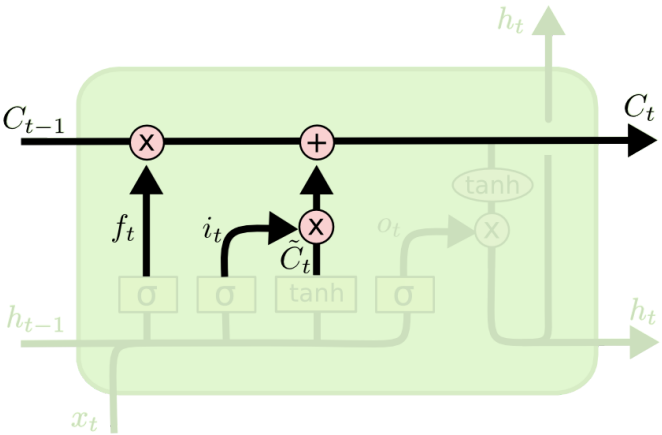
遗忘门



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

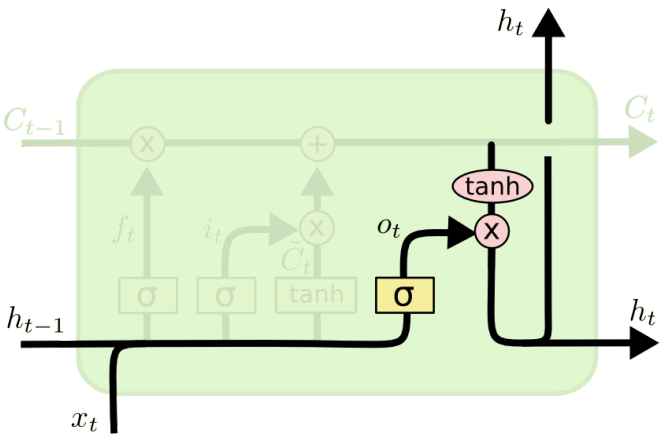
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

输入门和候选记忆元



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

新的记忆元

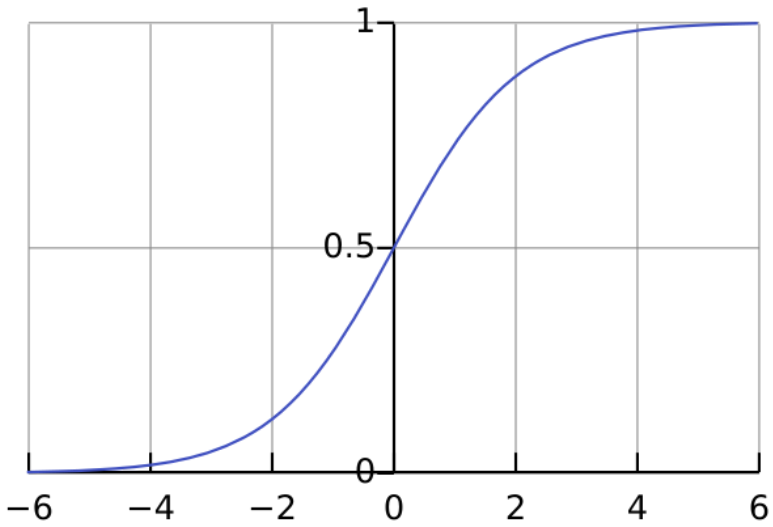


$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$

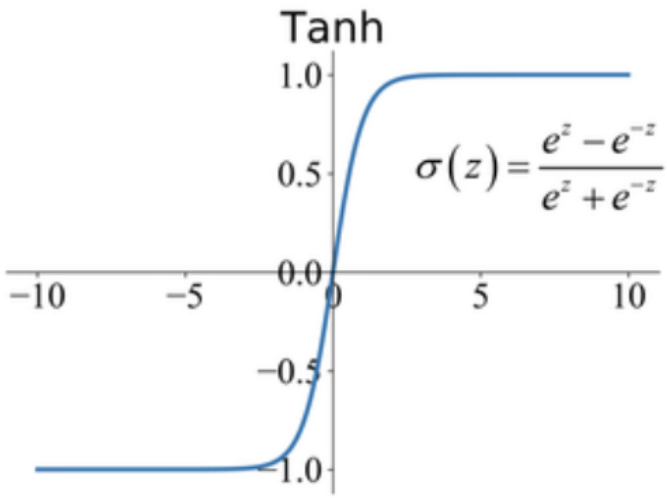
$$h_t = o_t * \tanh\left(C_t\right)$$

输出门和新的隐藏hidden

激活函数



sigmoid



tanh

Ct和Ht的作用

记忆单元 （Cell State）

记忆单元 C_t 是LSTM中的**长期记忆**部分，负责在序列的长时间跨度内存储重要信息。

- **作用**：记忆单元 C_t 用于存储贯穿整个序列的关键信息，允许LSTM保留长期依赖（long-term dependencies）。通过遗忘门和输入门的调控， C_t 中的信息能够长期保留，也可以在适当时候被更新或遗忘。这样可以有效缓解梯度消失问题，使得LSTM可以更好地学习到长时间跨度的依赖关系。
- **特点**：记忆单元 C_t 是一个累积的状态，不直接暴露给输出层。它通过在每个时间步的更新，保留了来自序列初期甚至更早信息的长期记忆。
- **公式**： $Ct = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$
- 其中 f_t 是遗忘门， i_t 是输入门， $\tilde{C}_t$ 是候选记忆单元。

隐藏状态 （Hidden State）

隐藏状态 h_t 是LSTM的**短期记忆**和当前时间步的输出，它包含了时间步 t 时刻的即时信息。

- **作用**：隐藏状态 h_t 作为 LSTM 的输出，被传递给下一层或下一时间步。在实际应用中，h_t 包含了LSTM当前时刻的状态信息，适合用于短期依赖任务。例如，语言生成任务中，每个时间步的输出hth_tht 能够捕获当前词或片段的信息，以便生成下一步的输出。
- **特点**：隐藏状态是经过输出门和 tanh激活函数的调控结果，具有即时性和短期性。因为每个时间步都会生成新的 h_t ，它不会像 CCC 那样在时序中累积长久的信息，而是更反映当前时刻的上下文。
- **公式**： $h_t = o_t \cdot \tanh(C_t)$
- 其中，oto_tot 是输出门，CtC_tCt 是当前时间步的记忆单元。

总结区别

特性	记忆单元 C	隐藏状态 h
作用	存储长期信息，维护序列的长时间依赖	当前时刻的输出，用于短期记忆和传递
更新频率	累积式更新，包含长期信息	每个时间步生成新的 h_t
传递方式	通过门控机制逐步调整	直接作为LSTM层的输出传递给下一层
激活函数	无直接激活	$\tanh$ 激活后的值，与输出门相乘

在实际应用中，**记忆单元** 扮演了长期信息存储的角色，而 **隐藏状态** 负责在每个时间步中输出即时信息。这种分工让LSTM可以同时捕捉长、短期的依赖关系，并有效解决传统RNN在处理长序列时遇到的梯度消失问题。

LSTM实践

从0实现

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

class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # 输入到隐藏状态的权重
        self.W_x = nn.Linear(input_size, 4 * hidden_size)
        # 上一时刻隐藏状态到当前隐藏状态的权重
        self.W_h = nn.Linear(hidden_size, 4 * hidden_size)

```

```
    def forward(self, x, hidden_state):
        # 解包 hidden_state，包含 (h_t-1, C_t-1)
        h_prev, c_prev = hidden_state

        # 计算门控机制：包含遗忘门、输入门、候选记忆单元和输出门
        gates = self.W_x(x) + self.W_h(h_prev)

        # 分割成四个部分：分别为遗忘门、输入门、候选记忆单元和输出门
        f_gate, i_gate, candidate, o_gate = torch.chunk(gates, 4, dim=1)

        # 遗忘门：用 sigmoid 激活，控制遗忘多少前一时刻的信息
        f_gate = torch.sigmoid(f_gate)

        # 输入门：用 sigmoid 激活，控制新信息的写入
        i_gate = torch.sigmoid(i_gate)

        # 候选记忆单元：用 tanh 激活，生成新候选信息
        candidate = torch.tanh(candidate)

        # 输出门：用 sigmoid 激活，控制哪些信息传递到下一个隐藏状态
        o_gate = torch.sigmoid(o_gate)

        # 更新记忆单元：前一时刻的记忆单元通过遗忘门保留部分，当前时刻的候选记忆单元通过输入门写入
        c_t = f_gate * c_prev + i_gate * candidate

        # 更新隐藏状态：当前记忆单元通过 tanh 激活后，通过输出门决定传递哪些信息
        h_t = o_gate * torch.tanh(c_t)

        # 返回新的隐藏状态 (h_t, c_t)
        return h_t, (h_t, c_t)


# 测试自定义的LSTMCell
input_size = 3   # 输入维度
hidden_size = 5  # 隐藏状态维度

# 实例化LSTMCell
lstm_cell = LSTMCell(input_size, hidden_size)

# 创建一个时间步的输入数据和初始状态
x = torch.randn(1, input_size)         # 输入数据，维度为 (batch_size, input_size)
h_prev = torch.zeros(1, hidden_size)  # 上一个时间步的隐藏状态
c_prev = torch.zeros(1, hidden_size)  # 上一个时间步的记忆单元状态

# 执行前向传播
h_t, (h_next, c_next) = lstm_cell(x, (h_prev, c_prev))
```

```
61
62 print("输出的隐藏状态 h_t:", h_t)
63 print("下一个时间步的隐藏状态 h_next:", h_next)
64 print("下一个时间步的记忆单元状态 c_next:", c_next)
65
```

简洁使用

```
1  import torch
2  import torch.nn as nn
3
4  # 定义LSTM模型
5  class LSTMModel(nn.Module):
6      def __init__(self, input_size, hidden_size, num_layers, output_size):
7          super(LSTMModel, self).__init__()
8          self.hidden_size = hidden_size
9          self.num_layers = num_layers
10
11         # LSTM层
12         self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
   batch_first=True)
13
14         # 全连接层，用于输出预测结果
15         self.fc = nn.Linear(hidden_size, output_size)
16
17     def forward(self, x):
18         # 初始化隐藏状态和记忆单元状态
19         h0 = torch.zeros(self.num_layers, x.size(0),
   self.hidden_size).to(x.device)  # 隐藏状态
20         c0 = torch.zeros(self.num_layers, x.size(0),
   self.hidden_size).to(x.device)  # 记忆单元状态
21
22         # LSTM 前向传播
23         out, _ = self.lstm(x, (h0, c0))
24
25         # 取最后一个时间步的输出
26         out = self.fc(out[:, -1, :])  # 取最后一个时间步的隐藏状态进行预测
27         return out
28
29 # 测试LSTM模型
30 input_size = 10    # 输入维度
31 hidden_size = 20   # 隐藏层维度
32 num_layers = 2     # LSTM层数
33 output_size = 1    # 输出维度
34
```

```
# 实例化模型
model = LSTMModel(input_size, hidden_size, num_layers, output_size)

# 创建随机输入数据 (batch_size, seq_length, input_size)
x = torch.randn(5, 7, input_size)  # 例如 batch_size=5, 序列长度=7, 输入维度=10
output = model(x)

```