

Lecture 8: 循环神经网络与LSTM

基础概念

序列数据

概念：序列数据

- 引入：
 - 实际中很多数据是有时序结构的；
 - 比如：电影的评价随时间变化而变化，拿奖后评分上升，直到奖项被忘记；
 - 看了很多好电影后，人们的期望变高，比如季节性：贺岁片、暑期档导演、演员的负面报道导致评分变低；
 - 再比如，音乐、文本、视频等内容都是连续的，再比如大地震发生后、很可能会有后续几次较小的余震；

• 概念：统计工具

- 假设在时间 t 观察到了 x_t 。那么得到了 T 个不独立的随机变量：

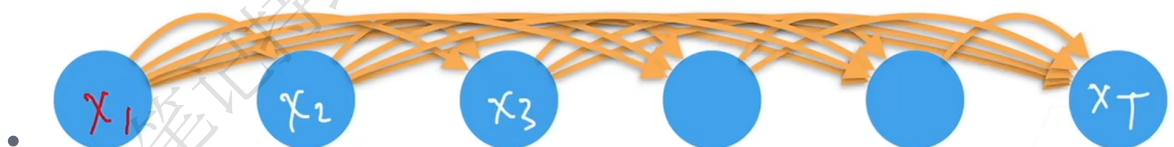
$$(x_1, \dots, x_T) \sim p(x)$$

- 使用条件概率展开：

$$p(a, b) = p(a)p(b|a) = p(b)p(a|b)$$

- 因此可以将概率写出来：

$$p(\mathbf{x}) = p(x_1) \cdot p(x_2 | x_1) \cdot p(x_3 | x_1, x_2) \cdot \dots p(x_T | x_1, \dots, x_{T-1})$$



- 假设给出前面的 $t-1$ 个数据，然后就使用这 $t-1$ 个数据来预测第 t 个数据，这个称之为自回归模型：
 - 对见过的数据建模，也称自回归模型：

$$p(x_t | x_1, \dots, x_{t-1}) = p(x_t | f(x_1, \dots, x_{t-1}))$$

• 总结：

- 时序模型中，当前数据跟之前观察到的数据相关；
- 自回归模型使用自身过去数据来预测未来；
- 马尔科夫模型假设当前只跟最近少数数据相关从而简化模型；
- 潜变量模型使用潜变量来概括历史信息；

文本预处理

代码

```
import collections
import re
from d2l import torch as d2l

d2l.DATA_HUB['time_machine']=(d2l.DATA_URL + "timemachine.txt",
"090b5e7e70c295757f55df93cb0a180b9691891a")

# 读取文字
def read_time():
    with open(d2l.download('time_machine','r')) as f:
        lines = f.readlines()
        return [re.sub('^A-Za-z+', ' ', line).strip().lower() for line in lines]

lines = read_time()

# 获得token
def tokenize(lines, token='word'):
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('错误:未知令牌类型:+' + token)
tokens = tokenize(lines)

# 构建词汇表
class Vocab:
    # 如果一个词汇出现的次数少于min_freq,则将其丢掉
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []

        # 把一堆token后, 计算每一个token的出现的次数
        counter = count_corpus(tokens)

        self.token_freqs = sorted(counter.items(), key=lambda x:x[1] reverse=True)
        # ...
```

语言模型

概念：语言模型

- 概念：
 - 给定文本序列，语言模型的目标是估计其文本的联合概率；
- 概念：最简单的模型 - 基于计数进行建模
 - 假设序列长度为 2，我们预测：

$$p(x, x') = p(x)p(x'|x) = \frac{n(x)}{n} \frac{n(x, x')}{n(x)}$$

- 其中 n 是总词数。 $n(x)$ 是单个单词和连续单词对的出现次数；
- 假设序列长度为 3，我们预测：

$$p(x, x', x'') = p(x)p(x'|x)p(x''|x, x') = \frac{n(x)}{n} \frac{n(x, x')}{n(x)} \frac{n(x, x', x'')}{n(x, x')}$$

- 马尔科夫假设：
 - 如果文本的长度很长时，可以使用马尔可夫假设来缓解这个问题；
 - 一元语法：

$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_1)p(x_2)p(x_3)p(x_4) \\ &= \frac{n(x_1)}{n} \frac{n(x_2)}{n} \frac{n(x_3)}{n} \frac{n(x_4)}{n} \end{aligned}$$

- 二元语法：
 - 每个词只和其上一个词语有关：

$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_1)p(x_2|x_1)p(x_3|x_2)p(x_4|x_3) \\ &= \frac{n(x_1)}{n} \frac{n(x_1, x_2)}{n(x_1)} \frac{n(x_2, x_3)}{n(x_2)} \frac{n(x_3, x_4)}{n(x_3)} \end{aligned}$$

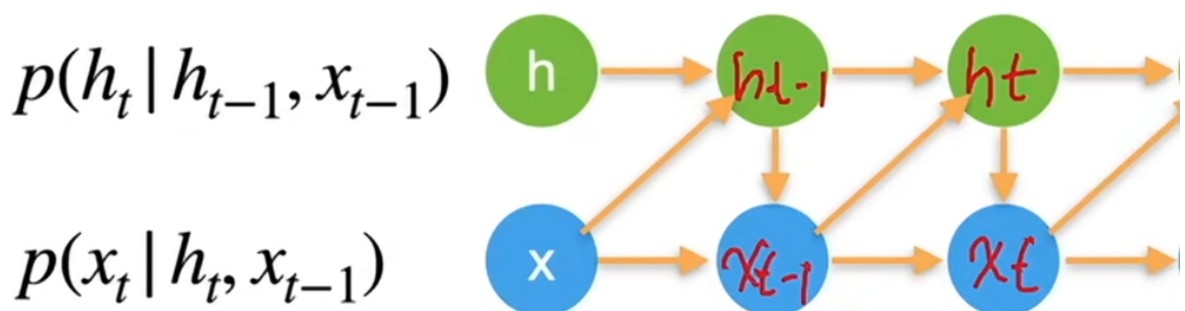
- 总结：
 - (1) 语言模型估计文本序列的联合概率；
 - (2) 使用统计方法时常采用 n 元语法；

循环神经网络

基本概念

概念： RNN - 循环神经网络

- 引入：潜变量回归模型：

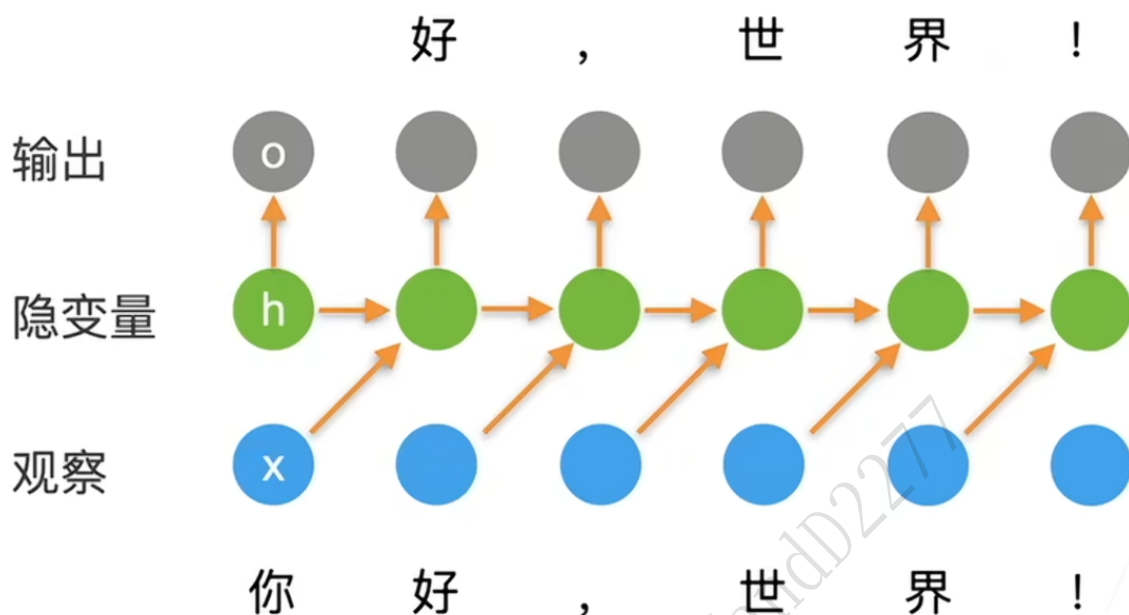


- 其中， h_t 和 h_{t-1} 与 x_{t-1} 有关， x_t 和 h_t 与 x_{t-1} 有关；

- 概念：循环神经网络

- 图示：

- 举例：输入的内容与输出的预测内容；



- 当前时刻的输出是为了预测当前时刻的输入，但得不到当前时刻的输入；

- 概念：

- 更新状态时，除了和前一时刻的 x_{t-1} 有关，还和前一时刻的 h_{t-1} 有关，如果去掉 h_{t-1} 就退化为 MLP，因此和 MLP 的区别就是使用了一个额外的 h_{t-1} 来存储时序信息；

更新隐藏状态： $\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_{t-1} + \mathbf{b}_h)$

输出： $\mathbf{o}_t = \phi(\mathbf{W}_{ho}\mathbf{h}_t + \mathbf{b}_o)$

- 输出就是使用当前预测得到的 h_t 来乘以权重，然后加上偏置值；

- 总结：

- 循环神经网络的输出取决于当下输入和前一段时间的隐变量，应用到语言模型中时，循环神经网络根据当前词预测下一次时刻词，通常使用困惑度来衡量语言模型的好坏；

困惑度与梯度裁剪

概念： 困惑度

- 概念：

- 衡量一个语言模型的好坏可以用平均交叉熵；

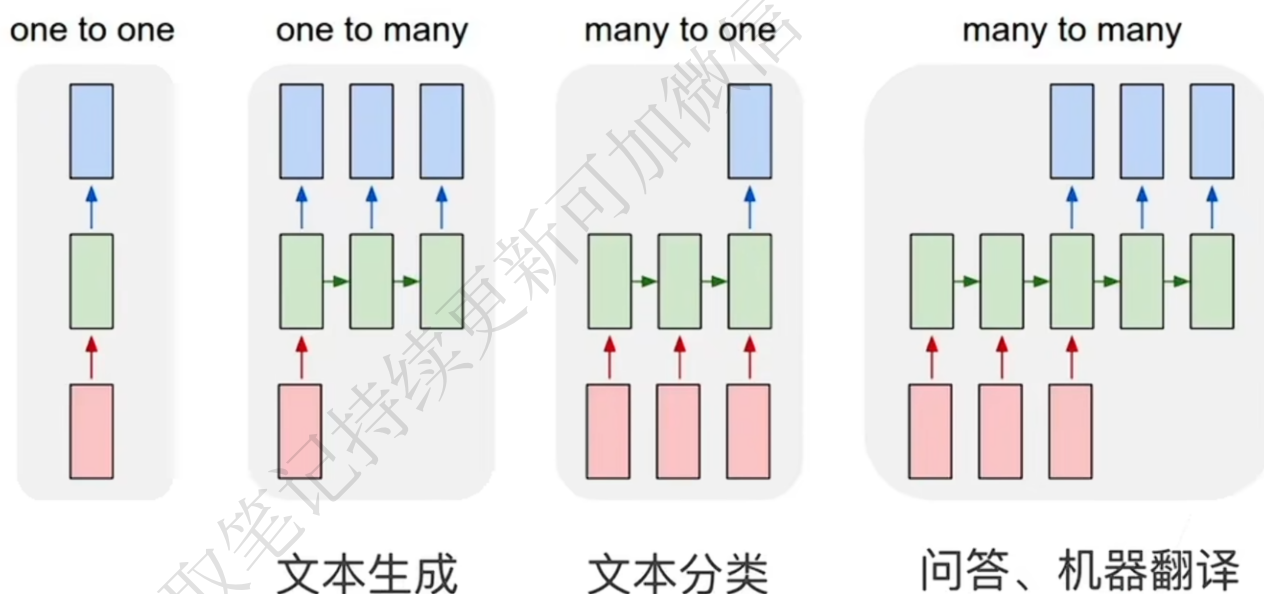
$$\pi = \frac{1}{n} \sum_{i=1}^n -\log p(x_t | x_{t-1}, \dots)$$

- 历史原因 **NLP** 使用困惑度 $\exp(z)$ 来衡量是平均每次可能选项，1 表示完美，无穷大是最差情况；
- 概念：梯度剪裁
 - 迭代中计算这 T 个时间步上的梯度，在反向传播过程中产生长度为 $O(T)$ 的矩阵乘法链，导致数值不稳定；
 - 因此使用梯度剪裁能有效预防梯度爆炸：
 - 如果梯度长度超过 θ ，那么拖影回长度 θ ；

$$\mathbf{g} \leftarrow \min \left(1, \frac{\theta}{\|\mathbf{g}\|} \right) \mathbf{g}$$

RNN 的应用

概念：RNN 的应用

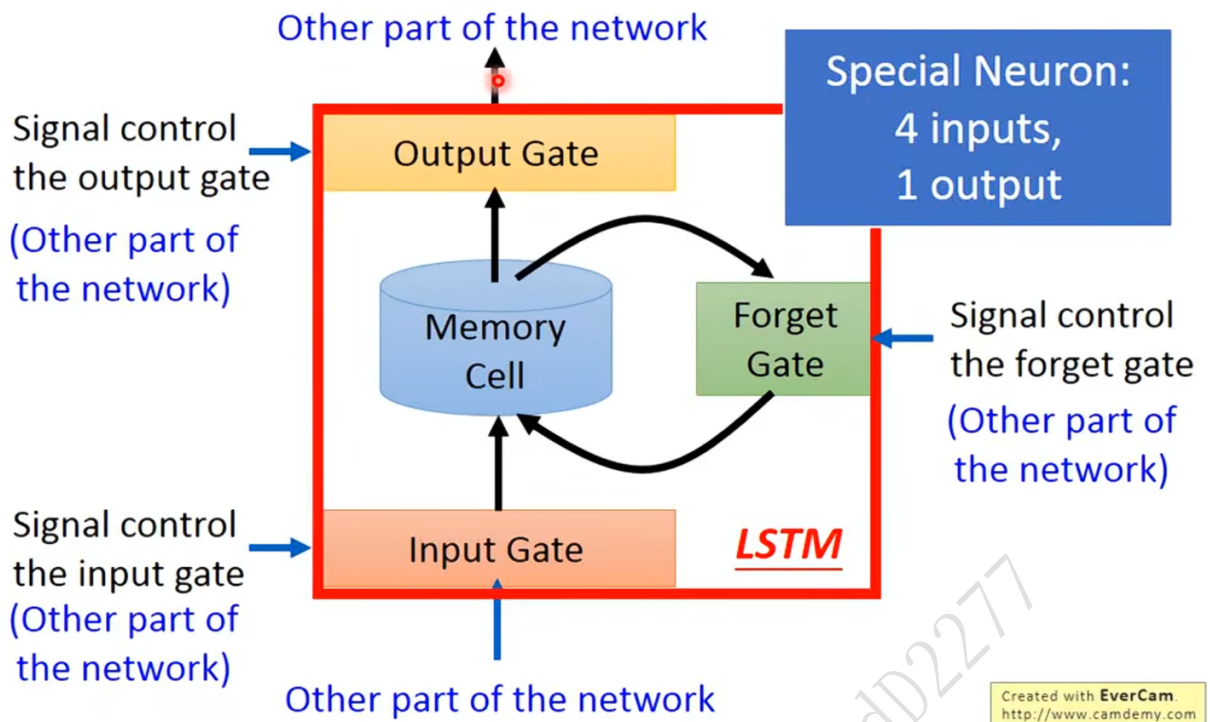


LSTM 长短期记忆网络

基础概念

概念：LSTM - 长短期记忆网络

- 引入：
 - **LSTM**（长短期记忆网络）是一种特殊的循环神经网络（**RNN**）结构，专门用于解决传统 **RNN** 在处理长序列数据时的梯度消失或梯度爆炸问题；
 - **LSTM** 的核心是其记忆单元（**cell state**）和隐藏状态（**hidden state**），并通过输入门（**input gate**）、遗忘门（**forget gate**）和输出门（**output gate**）来控制信息的流动；
- 图示：LSTM 结构



概念：LSTM 架构

- 概念：记忆单元与隐藏状态
 - 记忆单元：
 - 记忆单元是 LSTM 的核心部分，用于保存网络的长期记忆；
 - 它是一个连续的向量，贯穿整个网络的序列处理过程，能够存储和传递重要的历史信息；
 - 记忆单元的状态在每个时间步上都会根据输入门、遗忘门和输出门的调控进行更新；
 - 隐藏状态：
 - 隐藏状态是 LSTM 在每个时间步的输出，用于传递到下一个时间步或作为网络的最终输出；
 - 它反映了网络在当前时间步的短期记忆，是基于记忆单元的状态经过输出门处理后得到的；
- 概念：三种门
 - 忘记门：决定保留多少前一时刻的记忆；
 - I_t
 - 遗忘门的作用是决定记忆单元中哪些信息需要被遗忘；
 - 它通过激活函数（如 sigmoid）将输入和上一时刻的隐藏状态进行加权求和，并加上偏置值后，得到一个介于 0 和 1 之间的值，其表示需要遗忘的信息比例；
 - 公式：

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$

- 输入门：决定当前时刻输入的 x 多大程度上影响记忆单元；
 - F_t
 - 输入门的作用是控制当前输入信息有多少被写入到记忆单元中；

- 通过激活函数（如 `sigmoid`）将输入 (x_t) 和上一时刻的隐藏状态 (H_{t-1}) 进行加权求和，得到一个介于 0 和 1 之间的值，表示**当前输入信息的重要性**；
- 公式：

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$

- 输出门：决定是不是使用隐状态；
 - O_t
 - 输出门的作用是决定**记忆单元中的哪些信息需要被输出到隐藏状态**中；
 - 它通过激活函数（如 `sigmoid`）将输入和上一时刻的隐藏状态进行加权求和，得到一个介于 0 和 1 之间的值，表示需要输出的信息比例；
 - 公式：

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

记忆单元

概念：记忆单元

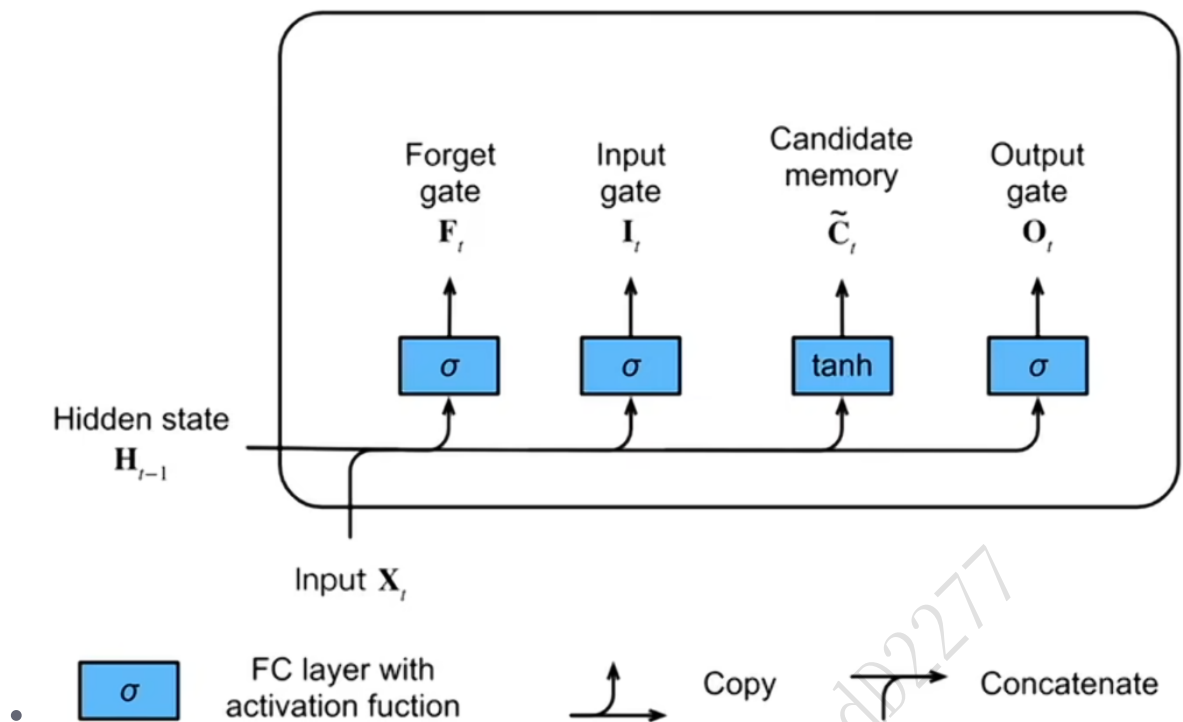
- 概念：候选记忆单元
 - 概念：
 - 候选记忆单元是 `LSTM` 在每个时间步生成的新信息，用于更新记忆单元的内容；
 - 它由当前输入和上一时刻的隐藏状态经过一个激活函数（通常是 `tanh`）计算得到，表示当前时间步可能需要存储的新信息；
 - 候选记忆单元是**当前时间步生成的新信息**，它通过**输入门**的调控决定有多少新信息被加入到记忆单元中；
 - 其有可能用、有可能不用；
 - 公式：
 - 图示：

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$$

- 解释：

- $\tilde{C}_t$ 是候选记忆单元。
- W_C 和 b_C 是权重矩阵和偏置项。
- h_{t-1} 是上一时刻的隐藏状态。
- x_t 是当前输入 ”。

- 图示：



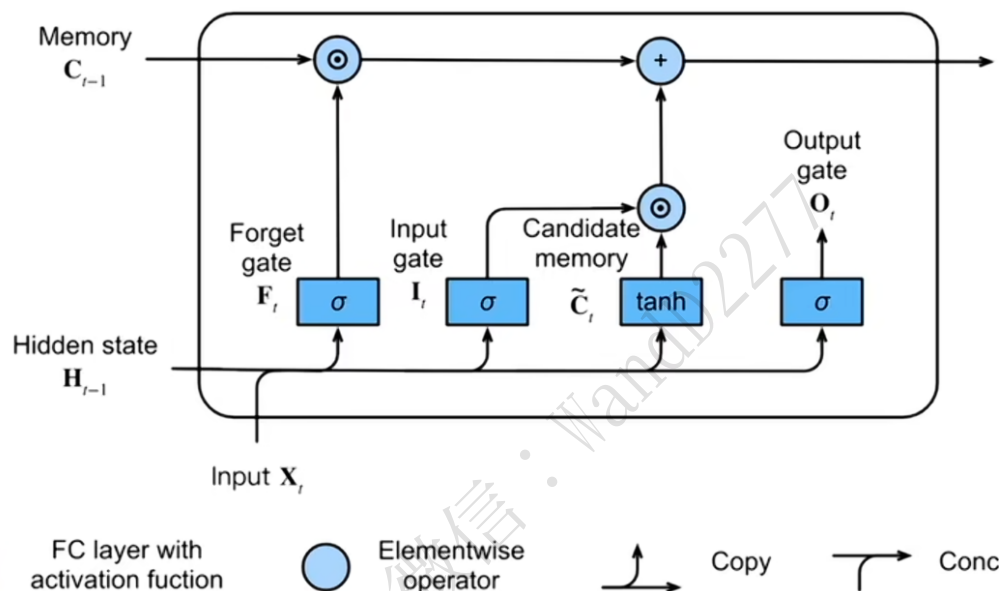
- 概念：记忆单元
 - 概念：
 - 记忆单元是 LSTM 的核心部分，用于存储长期依赖信息；
 - 它通过遗忘门和输入门的调控，结合旧的记忆和新的候选记忆单元来更新自身状态；
 - 实现：记忆单元的更新
 - 记忆单元 C_t 的更新，取决于输入门和忘记门；
 - 比如：
 - 如果输入门为 0，即 $F_t = 0$ ，则尽量不要去记住这个 C_t
 - 如果忘记门 $I_t = 0$ 则表示把现在的丢掉， $I_t = 1$ 则表示尽量的去用它；
 - 由下面的公式可知，当前的 C_t 十分的灵活，它即可选择只使用当前输入的 x 的状态（候选记忆单元），也可以选择直接使用上一个时刻的状态 C_{t-1} ，也可以选择两个都不使用（隐藏门和输入门都为 0）；
 - 总结：
 - 输入门控制是否使用上一个时刻的状态，隐藏门控制是否使用当前的输入 x 的状态，也就是使用候选记忆单元；
 - 公式：
 - 图示：

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

- 解释：

- C_t 是当前时刻的记忆单元状态。
- C_{t-1} 是上一时刻的记忆单元状态。
- f_t 是遗忘门的输出，控制旧记忆的保留程度。
- i_t 是输入门的输出，控制新候选记忆的加入程度”。

• 图示：



• 作用：

- 记忆单元的设计使得 LSTM 能够灵活地处理序列数据中的长短期依赖关系，避免了传统 RNN 中的梯度消失问题；

分析：梯度流的稳定性

- **加性更新**：细胞状态的更新 $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ 是累加操作而非传统 RNN 的乘积操作，避免梯度指数级衰减。
- **门控的导数**：sigmoid 和 tanh 的导数在合理范围内（如 tanh 导数为 $1 - \tanh^2(x)$ ），进一步稳定梯度。

隐状态

概念：隐状态

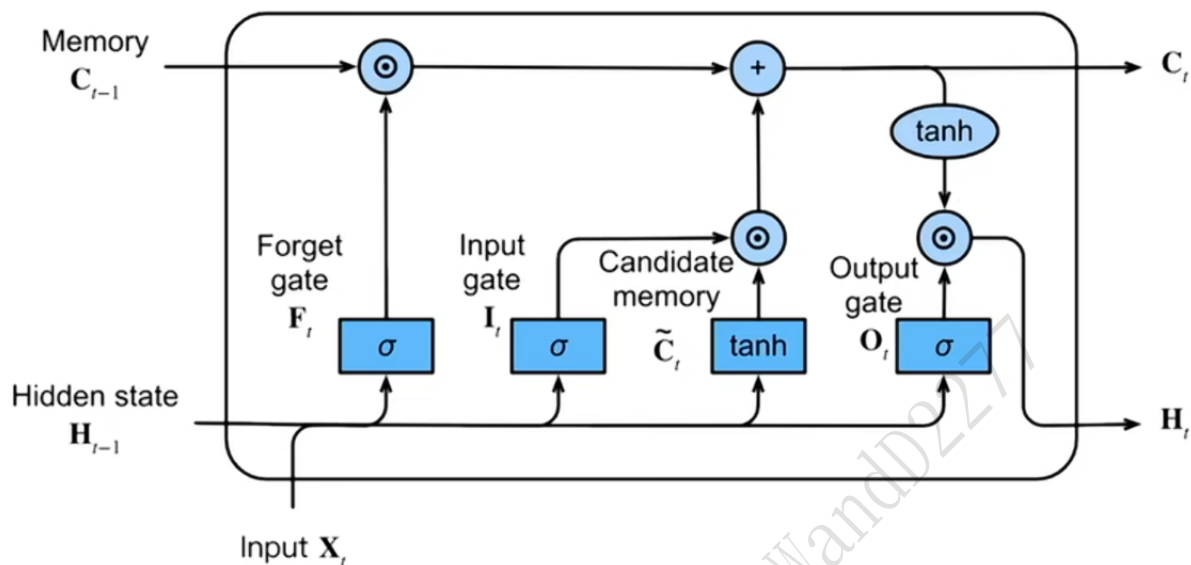
• 概念：

- 假设隐藏状态需要在 -1 到 $+1$ 之间的话，需要给它加上激活函数 tanh；
- 并且再和输出门点乘，用于控制要不要输出。如果 $O_t = 1$ 表示直接输出， $O_t = 0$ 表示完全不输出；

• 公式：

$$H_t = O_t \odot \tanh(C_t)$$

• 图示:



总结

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$$

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

$$H_t = O_t \odot \tanh(C_t)$$

代码实现

代码

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

```

import torch.nn.functional as F

class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        """
        LSTM单元实现
        参数:
            input_size: 输入特征维度
            hidden_size: 隐藏层维度
        """
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        # 合并输入和隐藏状态的权重参数 (4个门组合计算)
        # 权重矩阵维度: [input_size + hidden_size, 4 * hidden_size]
        # 偏置维度: [4 * hidden_size]
        self.weight_ih = nn.Parameter(torch.Tensor(input_size, 4 * hidden_size))
        self.weight_hh = nn.Parameter(torch.Tensor(hidden_size, 4 * hidden_size))
        self.bias = nn.Parameter(torch.Tensor(4 * hidden_size))

        # 参数初始化
        self.reset_parameters()

    def reset_parameters(self):
        """ Xavier均匀分布初始化参数 """
        nn.init.xavier_uniform_(self.weight_ih)
        nn.init.xavier_uniform_(self.weight_hh)
        nn.init.zeros_(self.bias) # 偏置初始化为0

    def forward(self, x, state):
        """
        前向传播
        参数:
            x: 输入张量, 形状 [batch_size, input_size]
            state: 元组 (h_prev, c_prev), 隐藏状态和记忆单元
                每个形状都是 [batch_size, hidden_size]
        返回:
            (h_next, c_next): 新的隐藏状态和记忆单元
        """
        h_prev, c_prev = state
        batch_size = x.size(0)

        # 合并输入和前一隐藏状态
        combined = torch.cat((x, h_prev), dim=1) # [batch_size, input_size + hidden_size]

        # 组合门计算 (同时计算四个门以提高效率)
        # 公式: gates = W_ih * x + W_hh * h_prev + b
        gates = (torch.mm(combined, torch.cat((self.weight_ih, self.weight_hh),
dim=0)))

```

```

        + self.bias) # [batch_size, 4 * hidden_size]

# 将组合结果拆分为四个门（顺序：输入门、遗忘门、候选记忆、输出门）
# 每个门的维度都是 [batch_size, hidden_size]
i_gate = gates[:, :self.hidden_size] # 输入门
f_gate = gates[:, self.hidden_size:2*self.hidden_size] # 遗忘门
g_gate = gates[:, 2*self.hidden_size:3*self.hidden_size] # 候选记忆单元
o_gate = gates[:, 3*self.hidden_size:] # 输出门

# 应用激活函数
i_gate = torch.sigmoid(i_gate) # 输入门 (0~1)
f_gate = torch.sigmoid(f_gate) # 遗忘门 (0~1)
g_gate = torch.tanh(g_gate) # 候选记忆 (-1~1)
o_gate = torch.sigmoid(o_gate) # 输出门 (0~1)

#####
# 记忆单元更新（核心公式）
# c_next = f_gate * c_prev + i_gate * g_gate
# 意义：遗忘旧记忆 + 添加新候选记忆
#####
c_next = f_gate * c_prev + i_gate * g_gate

#####
# 隐藏状态更新
# h_next = o_gate * tanh(c_next)
# 意义：基于新记忆生成输出
#####
h_next = o_gate * torch.tanh(c_next)

return h_next, c_next

```

```

class LSTM(nn.Module):
    """ 完整的LSTM模型（处理序列输入） """
    def __init__(self, input_size, hidden_size):
        super(LSTM, self).__init__()
        self.lstm_cell = LSTMCell(input_size, hidden_size)
        self.hidden_size = hidden_size

    def forward(self, x, initial_state=None):
        """
        参数:
            x: 输入序列, 形状 [seq_len, batch_size, input_size]
            initial_state: 初始状态 (h0, c0)
        返回:
            outputs: 所有时间步的隐藏状态 [seq_len, batch_size, hidden_size]
            (h_last, c_last): 最终状态
        """
        seq_len, batch_size, _ = x.size()
        outputs = []

        # 初始化隐藏状态

```

```

if initial_state is None:
    h = torch.zeros(batch_size, self.hidden_size).to(x.device)
    c = torch.zeros(batch_size, self.hidden_size).to(x.device)
else:
    h, c = initial_state

# 遍历时间步
for t in range(seq_len):
    h, c = self.lstm_cell(x[t], (h, c))
    outputs.append(h.unsqueeze(0)) # 保持维度一致性

outputs = torch.cat(outputs, dim=0) # [seq_len, batch_size, hidden_size]
return outputs, (h, c)

```

LSTM - Attention

概念*: LSTM + Attention

- 概念:
 - 使用 Attention 可以更多的来把握输入特征的信息;
 - LSTM-Attention 模型是一种将长短期记忆网络 (Long Short-Term Memory, LSTM) 与注意力机制 (Attention) 结合的深度学习模型, 广泛应用于时间序列预测、自然语言处理等领域;
 - LSTM 用于捕捉时间序列中的长短期依赖, 注意力机制则能够动态地为不同的时间步分配权重, 提升型在处理复杂序列时的能力;
- 作用:
 - 注意力机制的引入解决了传统 LSTM 在处理长序列时可能忽略一些关键时间步的问题;
 - 就是 LSTM 可能会忽视过去很长时间之前的很重要的时间步;
 - Attention 机制通过为每个时间步的隐藏状态分配一个权重, 能够更灵活地选择重要的信息;
 - 其主要思想是: 根据输入序列中的隐藏状态 h , 计算其与当前时刻 t 的相关性, 并基于相关性动态调整各时间步的贡献;

- 概念: 相似度计算

$$e_t = v^T \tanh(W_h \cdot h_t)$$

- 概念: 权重分配
 - 使用 softmax 函数将注意力得分转换为权重

$$\alpha_t = \frac{\exp(e_t)}{\sum_{k=1}^T \exp(e_k)}$$

- 概念: 加权和
 - 对所有时间步的隐藏状态进行加权求和, 得到上下文向量 $h_{\text{attention}}$

.

$$h_{\text{attention}} = \sum_{t=1}^T \alpha_t h_t$$

.

网络结构及其实现

概念：网络结构

- 1. 输入层：接收输入的时间序列数据；
- 2. **LSTM** 层：捕捉序列中的长短期依赖，生成每个时间步的隐藏状态；
- 3. 注意力机制层：根据不同时间步的隐藏状态计算重要性权重，生成加权的上下文向量；
- 4. 输出层：根据上下文向量生成最终的预测值。

实例分析

分析： **LSTM-Attention** 实例

- 数据：

时间步	数据值
1	100
2	102
3	98
4	105
5	103

- 计算：
 - 1. 首先，我们将前五个时间步的数据作为输入，送入 **LSTM** 网络。**LSTM** 根据之前的时间步信息，逐步更新其记忆单元和隐藏状态。假设我们通过 **LSTM** 计算得到以下隐藏状态：
 - $h_1 = 0.5$
 - $h_2 = 0.7$
 - $h_3 = 0.4$
 - $h_4 = 0.8$
 - $h_5 = 0.6$
 - 2. 注意力机制部分

- 注意力机制会根据当前的任务，计算每个隐藏状态的重要性权重。假设我们计算得到以下注意力得分 e

- $e_1 = 0.1$

- $e_2 = 0.3$

- $e_3 = 0.2$

- $e_4 = 0.4$

- $e_5 = 0.2$

- 3. 通过 `Softmax` 函数得到注意力权重

- $\alpha_1 = 0.1, \alpha_2 = 0.2, \alpha_3 = 0.2, \alpha_4 = 0.3, \alpha_5 = 0.2$

- 这些权重代表了每个时间步在预测中所占的比例；

- 4. 加权求和

- 最后，我们根据注意力权重，对每个时间步的隐藏状态进行加权求和，得到最终的上下文向量；

$$h_{\text{attention}} = 0.1 \cdot 0.5 + 0.2 \cdot 0.7 + 0.2 \cdot 0.4 + 0.3 \cdot 0.8 + 0.2 \cdot 0.6$$

$$h_{\text{attention}} = 0.05 + 0.14 + 0.08 + 0.24 + 0.12 = 0.63$$

- 5. 预测结果

- 通过 `LSTM` 和 Attention 机制提取出的上下文向量 $h_{\text{attention}}$ ，我们可以预测下一个时间步的数据值；
- 假设最终预测的第 6 个时间步的值为 `104`，这表示模型认为在接下来的时间步中，数据值为 `104`；

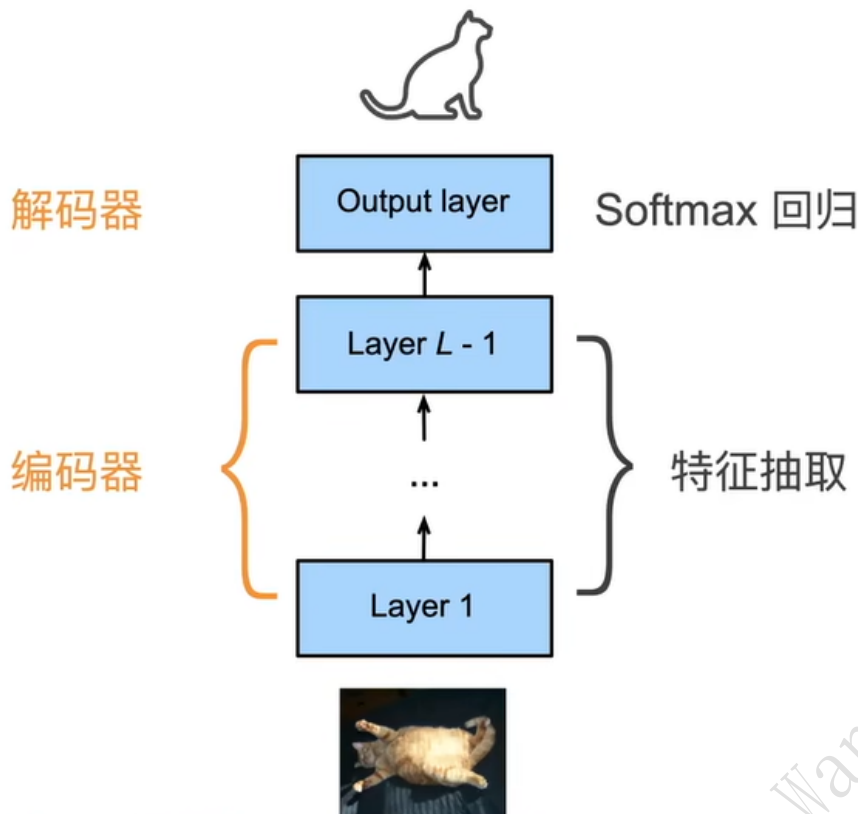
编码器和解码器

引入

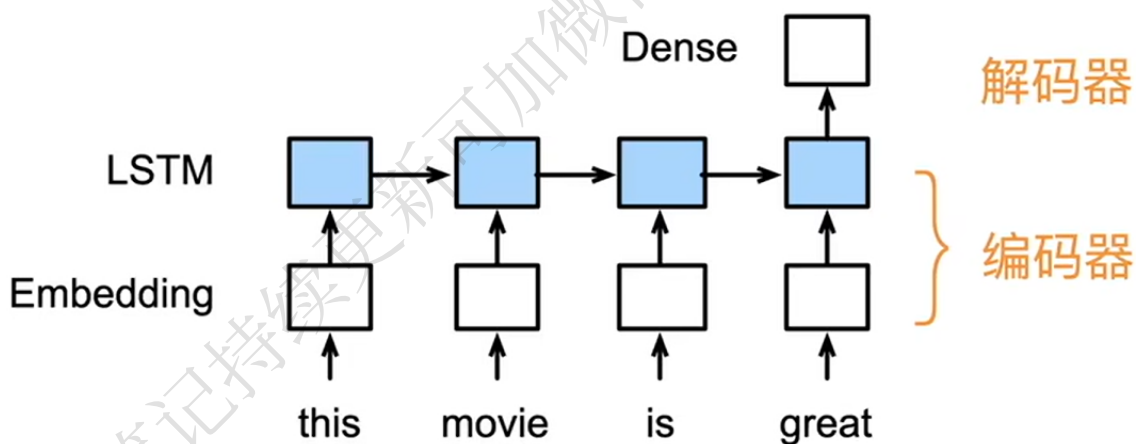
概念：编码器和解码器

- 引入：

- 在 `CNN` 网络中，编码器就是将输入编程成中间的表达形式 (特征)，而解码器就将中间表示解码成输出；



- 概念：RNN 与解码器、编码器
- 图示：

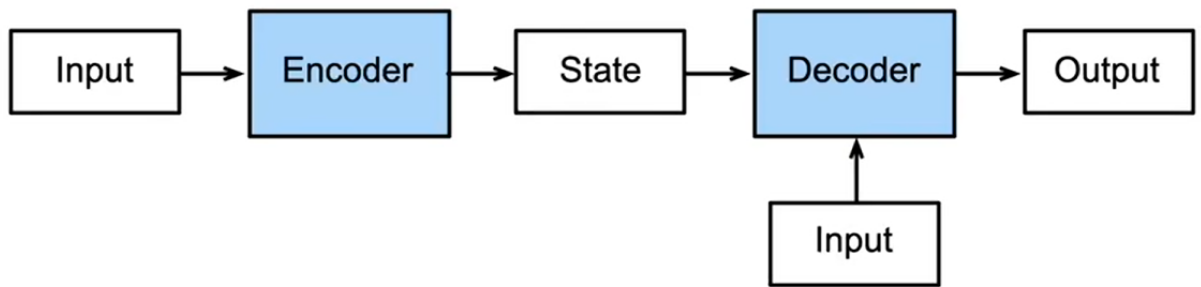


- 概念：
 - 编码器就是将文本表示为向量；
 - 解码器就是将向量表示为输出；

编码器-解码器架构

概念： 编码器-解码器架构

- 概念：
 - 一个模型被分成两个部分，编码器处理输入，解码器生成输出；
- 图示：



• 代码:

```
class EncoderDecoder(nn.Module):  
    """编码器-解码器结构的基类。"""  
    def __init__(self, encoder, decoder, **kwargs):  
        super(EncoderDecoder, self).__init__(**kwargs)  
        self.encoder = encoder  
        self.decoder = decoder  
  
    def forward(self, enc_X, dec_X, *args):  
        enc_outputs = self.encoder(enc_X, *args)  
        dec_state = self.decoder.init_state(enc_outputs, *args)  
        return self.decoder(dec_X, dec_state)
```