笔记：循环神经网络

第八章 循环神经网络

8.1 序列模型

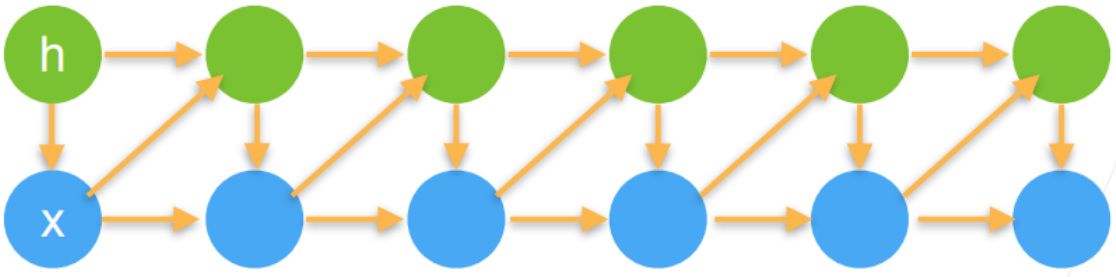
1.很多数据具有时序结构

2.统计工具

- 在时间t观察到x_t,得到T个不独立的随机变量（$x_1, \ldots x_T$)~p(**x**)
- p(**x**) =p（x1)p(x2|x1)p(x3|x1,x2)...p(xT|x1,...xT-1)
- p(**x**) =p（xT)p(xT-1|xT)p(xT-2|xT-1,xT-2)...p(x1|x1,...xT-1)
- 对条件概率建模：p(xt|x1...xt-1)=p(xt|f(x1...xt-1)),对见过的数据建模，也称自回归模型

  3.方案A 马尔可夫假设
- 假设当前数据只和τ个过去数据点相关
- p(xt|x1...xt-1)=p(xt|xt-τ...xt-1)=p(xt|f(xt-τ...xt-1)),例如在过去数据上训练MLP模型
- τ=1时，得到一阶马尔可夫模型， p(**x**) =Πp(xt|xt-1)

  4.方案B 潜变量模型
- 潜变量ht=f(x1...xt-1)->xt=p（xt|ht)
-



8.2 文本预处理

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

将数据集读取到由文本行组成的列表中

```
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')
'''读取一本书'''
def read_time_machine():
    """Load the time machine dataset into a list of text lines."""
    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]
'''把不是字母和空格的都变成空格'''
lines = read_time_machine()
print(f'
print(lines[0])
print(lines[10])
```

text lines: 3221

the time machine by h g wells

twinkled and his usually pale face was flushed and animated the

每个文本序列被拆分成一个标记列表

文本行列表lines-文本序列line-词元列表tokens-词元token

```
def tokenize(lines, token='word'):
    """将文本行拆分为单词或字符标记。"""
    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)
for i in range(11):
    print(tokens[i])
```

['the', 'time', 'machine', 'by', 'h', 'g', 'wells']

[]

[]

[]

[]

['i']

[]

[]

['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to', 'speak', 'of', 'him']

['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes', 'shone', 'and']

['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and', 'animated', 'the']

构建一个字典，通常也叫做*词表*（vocabulary），用来将字符串标记映射到从0开始的数字索引中

```
class Vocab:
    """文本词表"""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
    '''如果某个词出现次数小于min_freq，就不要了；保存那些被保留的词元， 例如：填充词元（“<pad>”）； 序列开始词元（“<bos>”）； 序列结束词元（“<eos>”）'''
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)'''频率排序'''
        self.unk'''unknown记为0''', uniq_tokens = 0, ['<unk>'] +
reserved_tokens
        uniq_tokens += [
            token for token, freq in self.token_freqs
            if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()'''下标和token相互转换'''
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):'''token->index，返回index'''
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):'''index->token，返回token'''
        if not isinstance(indices, (list, tuple)):
```

```
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

def count_corpus(tokens):
    """统计标记的频率。"""
    if len(tokens) == 0 or isinstance(tokens[0], list):
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

构建词汇表

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

[('unk', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to', 6), ('was', 7), ('in', 8), ('that', 9)]

将每一行文本转换成一个数字索引列表

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
    '''当编写_getitem_方法并包含在你的类中时，python解释器会在实例上使用方括号自动调用方法'''
```

words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']

indices: [1, 19, 50, 40, 2183, 2184, 400]

words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and', 'animated', 'the']

indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]

将所有内容打包到`load_corpus_time_machine`函数中

```
def load_corpus_time_machine(max_tokens=-1):
    """返回时光机器数据集的标记索引列表和词汇表。"""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)'''对应字典'''
    corpus = [vocab[token] for line in tokens for token in line]'''每一个单词
```

```
(这里是字母)的数字'''
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

(170580, 28) 28=16+ukn+空格

8.3 语言模型

1.给定文本序列x1-xT，语言模型的目标是估计联合概率p（x1-xT）

2.使用计数来建模

- 假设序列长度为2，$p(x,x')=p(x)p(x'|x)=\frac{n(x)}{n}\frac{n(x,x')}{n(x)}$,n是语料库中的总词数，n（x），n（x，x'）是单个单词和连续单词对的出现次数

  3.N元语法

- 当序列很长时，因为文本量不够大，很可能n（x1-xT）<=1
- 使用马尔可夫假设

  - 一元语法：$p(x_1,x_2,x_3,x_4)=p(x_1)p(x_2)p(x_3)p(x_4)=\frac{n(x_1)}{n}\frac{n(x_2)}{n}\frac{n(x_3)}{n}\frac{n(x_4)}{n}$

  - 一元语法：

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

  4.代码

  1.F1:随机地生成一个小批量数据的特征和标签以供读取。 在随机采样中，每个样本都是在原始的长序列上任意捕获的子序列

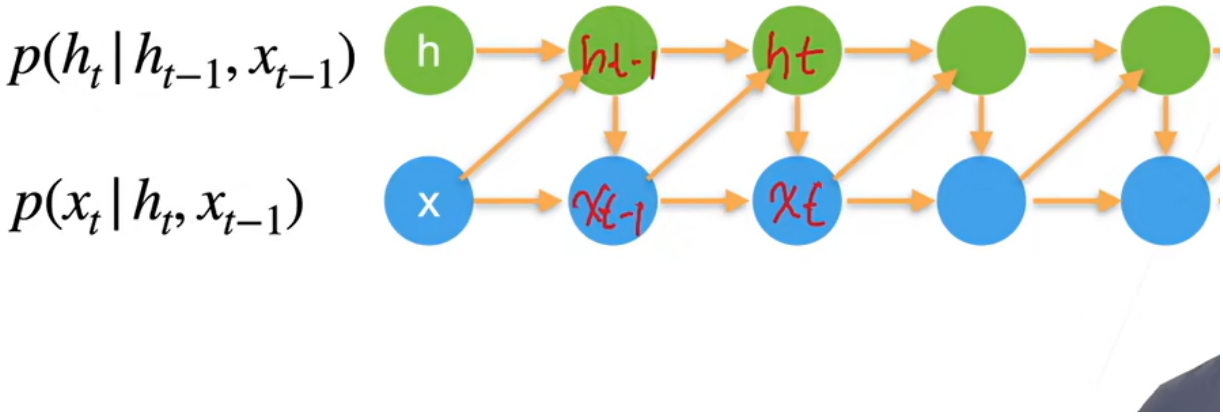
```
def seq_data_iter_random(corpus, batch_size, num_steps'''tau'''):
    """使用随机抽样生成一个小批量子序列。"""
    '''不同于8.1中的遍历每个都需要用很多次，这个是把总长切成n份，然后一个epoch就n份都过一遍。每个epoch开始的起点都是从0~tau中随机取，这样份就不会重复'''
    corpus = corpus[random.randint(0, num_steps - 1):]
    num_subseqs = (len(corpus) - 1) // num_steps
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))'''每个子序列开始的下标'''
    random.shuffle(initial_indices)

    def data(pos):
        return corpus[pos:pos + num_steps]
```

```
    num_batches = num_subseqs // batch_size'''在n段里面取batch'''
    for i in range(0, batch_size * num_batches, batch_size):
        initial_indices_per_batch = initial_indices[i:i + batch_size]'''取了
batch size个开始的下标'''
        X = [data(j) for j in initial_indices_per_batch]'''生成batch size个
段'''
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield torch.tensor(X), torch.tensor(Y)
```

2.F2:保证两个相邻的小批量中的子序列在原始序列上也是相邻的

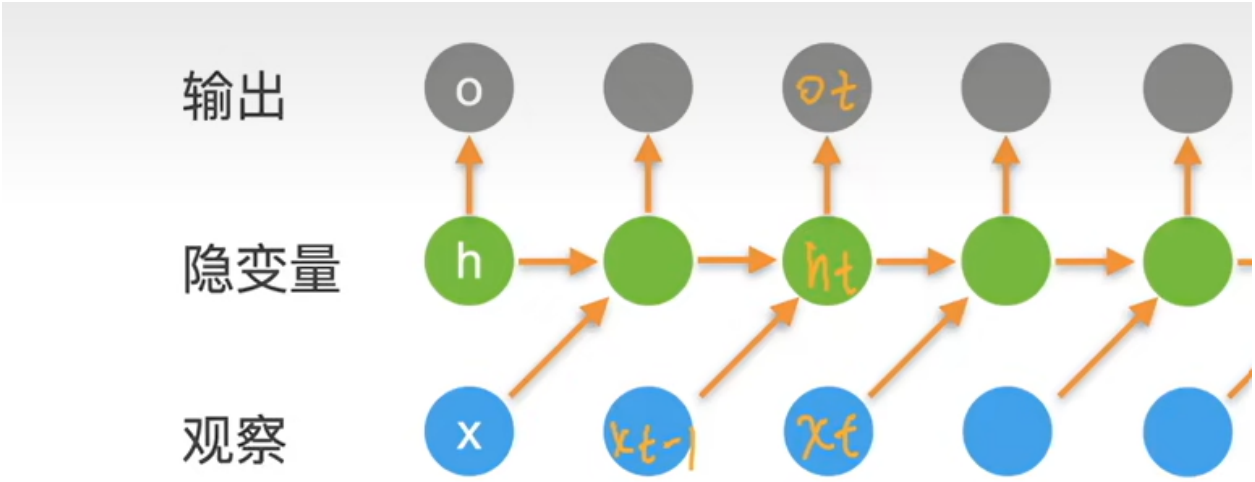
8.4 循环神经网络

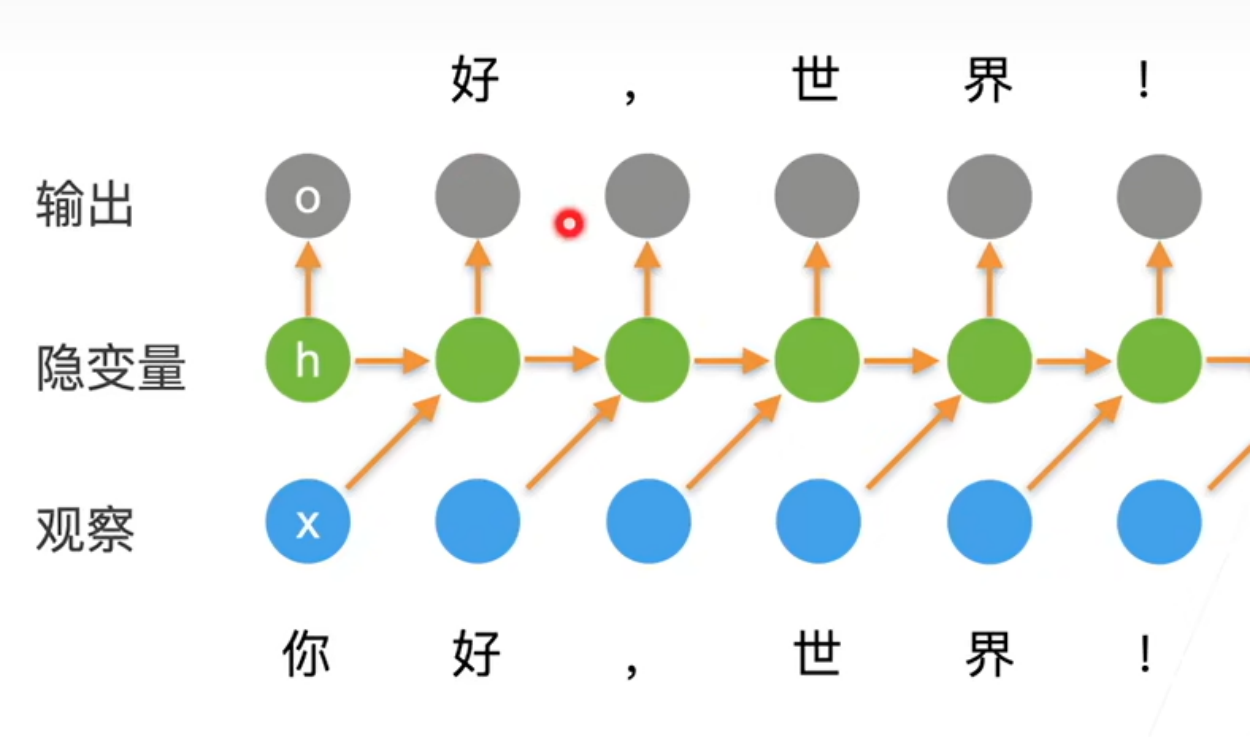
1.



更新隐藏状态：$h_t = \phi(W_{hh}h_{t-1} + W_{hx}x_{t-1} + b_h)$ (去掉$W_{hh}h_{t-1}$就是MLP)

$o_t = W_{ho}h_t + b_o$





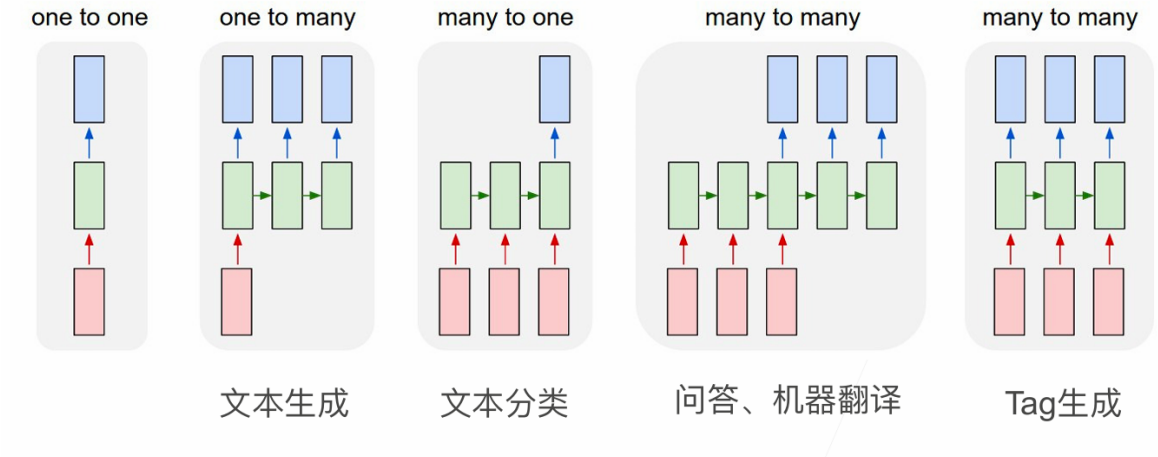
2.困惑度 perplexity

- 衡量一个语言模型的好坏可以用平均交叉熵$\pi = \frac{1}{n}\sum_{i=1}^{n} -logp(x_t|x_{t-1},...)$ 其中p是语言模型预测的概率，xt是真实词
- NLP使用困惑度exp（pi）来衡量，1表示完美，无穷大是最差情况

  3.梯度裁剪
- 迭代中计算T个时间步上的梯度，在反向传播中产生长度为O（T）的矩阵乘法链，导致数值不稳定
- 梯度裁剪能有效预防梯度爆炸

  - 如果梯度长度超过θ,那么将拖回θ：$g \leftarrow min(1, \frac{\theta}{\|g\|})g$

  4.其他应用



5.由于在当前时间步中， 隐状态使用的定义与前一个时间步中使用的定义相同， 因此 $h_t = \phi(W_{hh}h_{t-1} + W_{hx}x_{t-1} + b_h)$的计算是循环的

8.5 循环神经网络的从零开始实现

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

独热编码

给一个下标，用向量来表示

将每个索引映射为相互不同的单位向量： 假设词表中不同词元的数目为N（即`len(vocab)`）， 词元索引的范围为0到N−1。 如果词元的索引是整数i， 那么我们将创建一个长度为N的全0向量， 并将第i处的元素设置为1。 此向量是原始词元的一个独热向量

```
F.one_hot(torch.tensor([0, 2]), len(vocab))
```

tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0]])

小批量形状是(批量大小, 时间步数)

转置的目的：使我们能够更方便地通过最外层的维度， 一步一步地更新小批量数据的隐状态

```
X = torch.arange(10).reshape((2, 5))
F.one_hot(X.T, 28).shape
```

torch.Size([5, 2, 28])

初始化循环神经网络模型的模型参数

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

一个`init_rnn_state`函数在初始化时返回隐藏状态

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),)
```

下面的`rnn`函数定义了如何在一个时间步计算隐藏状态和输出

```
def rnn(inputs, state, params):
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

创建一个类来包装这些函数

```
class RNNModelScratch:
    """从零开始实现的循环神经网络模型"""
    def __init__(self, vocab_size, num_hiddens, device, get_params,
```

```
                 init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

检查输出是否具有正确的形状

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
Y, new_state = net(X.to(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

(torch.Size([10, 28]), 1, torch.Size([2, 512]))

首先定义预测函数来生成用户提供的`prefix`之后的新字符

```
def predict_ch8(prefix, num_preds, net, vocab, device):
    """在`prefix`后面生成新字符。"""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape(
        (1, 1))
    for y in prefix[1:]:
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])

predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

梯度裁剪

```python
def grad_clipping(net, theta):
    """裁剪梯度。"""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad**2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

定义一个函数来训练只有一个迭代周期的模型

```python
def train_epoch_ch8(net, train_iter, loss, updater, device,
use_random_iter):
    """训练模型一个迭代周期（定义见第8章）。"""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)
    for X, Y in train_iter:
        if state is None or use_random_iter:
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                state.detach_()
            else:
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
        else:
            l.backward()
            grad_clipping(net, 1)
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

训练函数支持从零开始或使用高级API实现的循环神经网络模型

```
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """训练模型（定义见第8章）。"""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(net, train_iter, loss, updater, device,
                                     use_random_iter)
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    print(f'困惑度 {ppl:.1f}, {speed:.1f} 标记/秒 {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

8.6 循环神经网络的简洁实现

第九章 现代循环神经网络

9.1 门控循环单元GRU

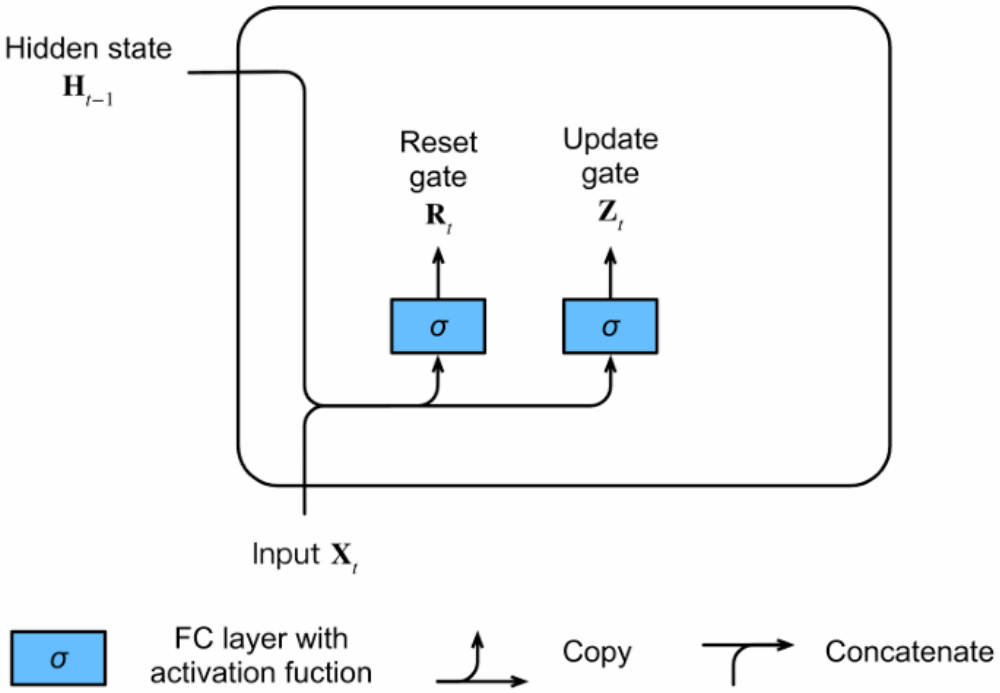
1.不是所有观察值同等重要->只记住相关的观察需要

- 能关注的机制：更新门
- 能遗忘的机制：重置门

  2.门

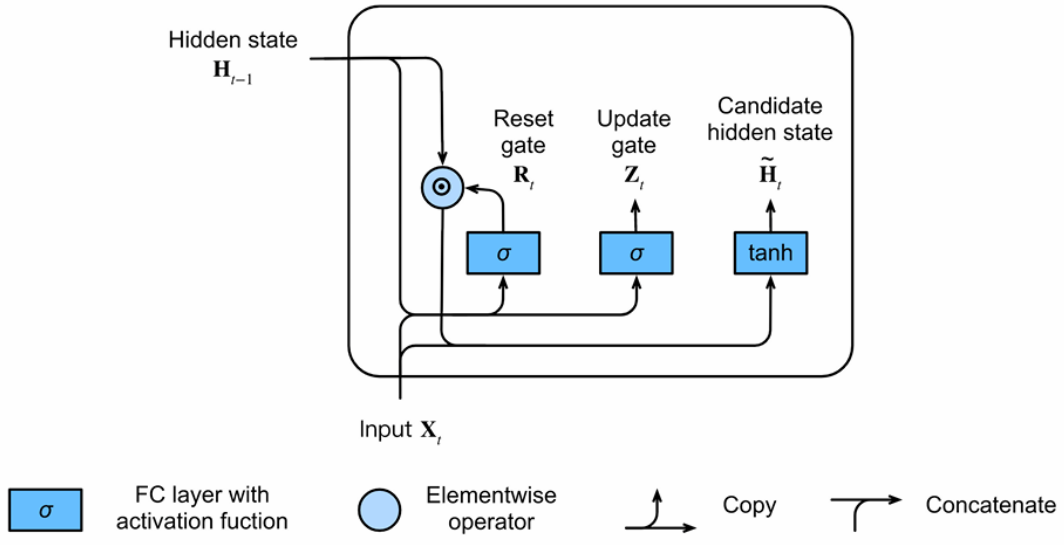
  σ是sigmoid

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r),$$
$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z)$$



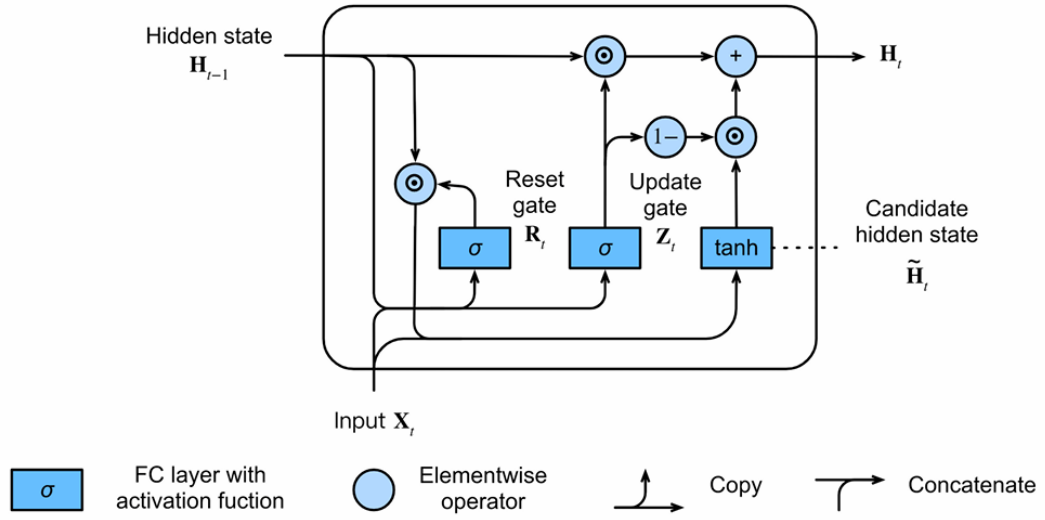
候选隐藏状态

$$\tilde{H}_t = \tanh(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h)$$



Rt可以学习，sigmoid介于0-1之间，0就忘了过去的Ht，只和Xt有关，1就过去的Ht全留着，和正常的隐藏状态一样

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t$$



Zt可以学习，sigmoid介于0-1之间，0就拿来候选Ht（调整过的，用了Xt），1就直接用上一个Ht，和新输入Xt没有关系

3.门控循环单元具有以下两个显著特征：

- 重置门有助于捕获序列中的短期依赖关系
- 更新门有助于捕获序列中的长期依赖关系

9.2 长短期记忆网络LSTM

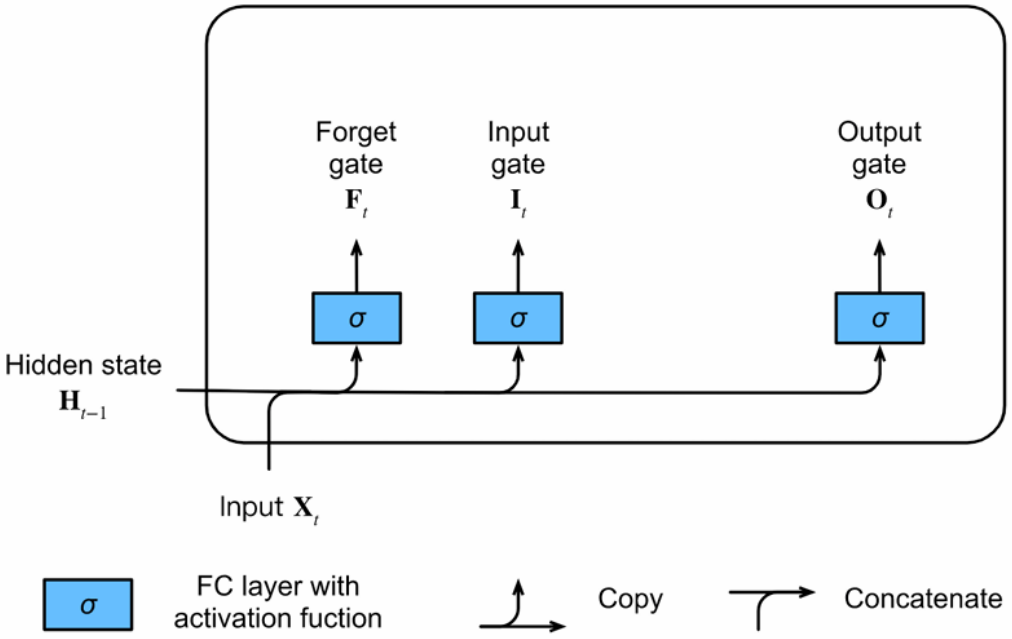
1.

- 忘记门：将值朝0减少
- 输入门：决定是否忽略掉输入数据
- 输出门：决定是不是使用隐状态

2.

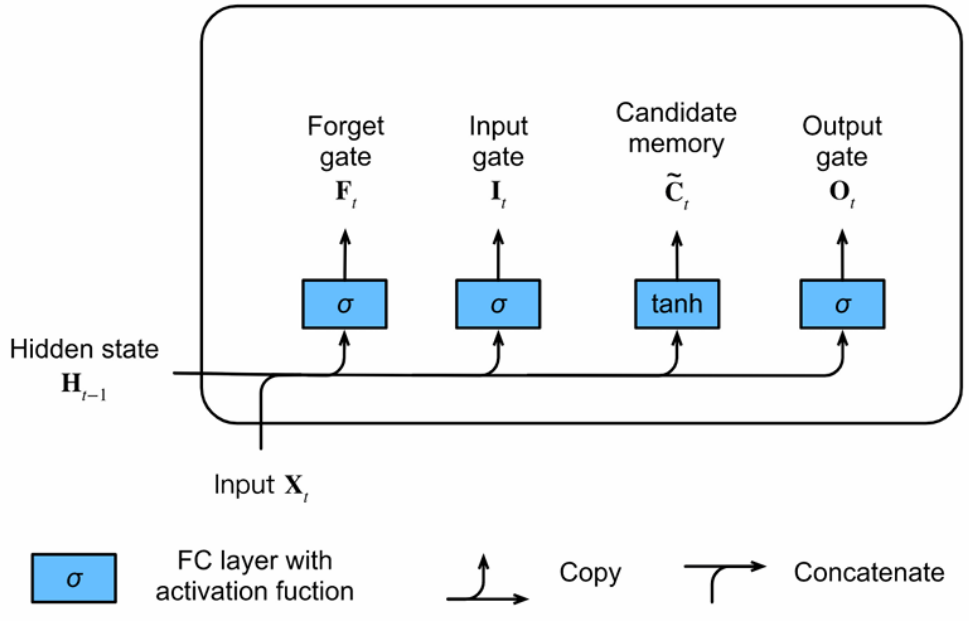
门

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$
$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$
$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$



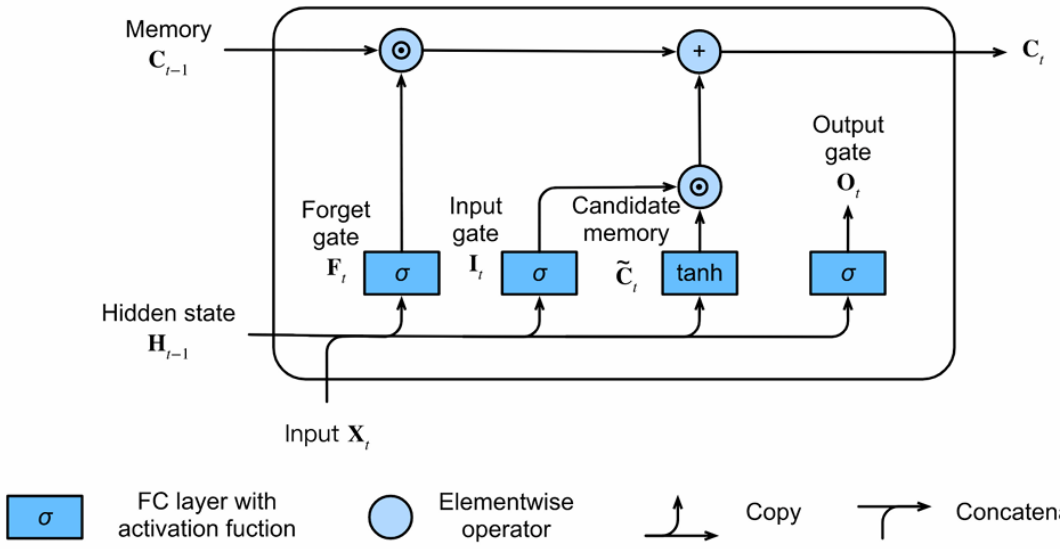
候选记忆单元

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



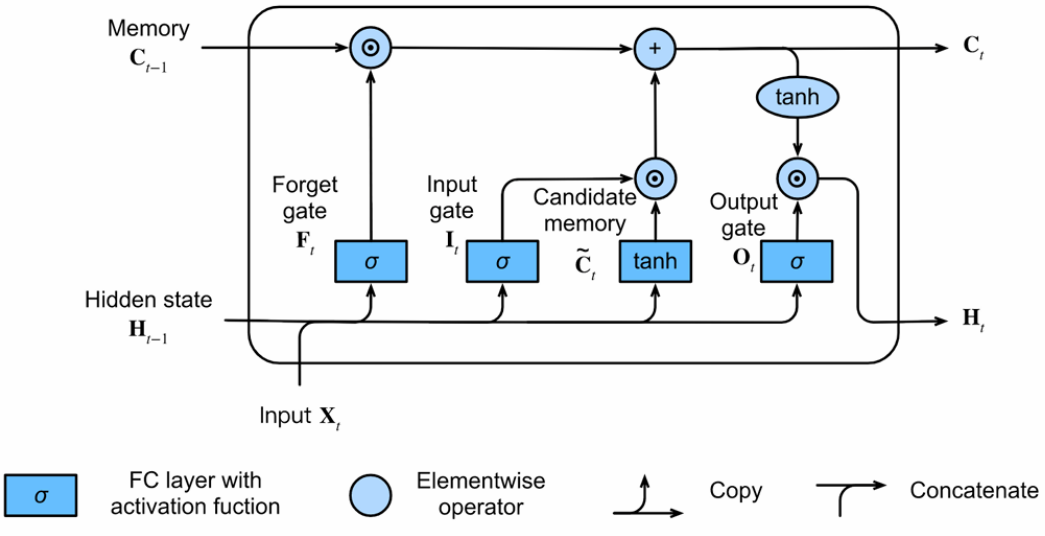
记忆单元

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



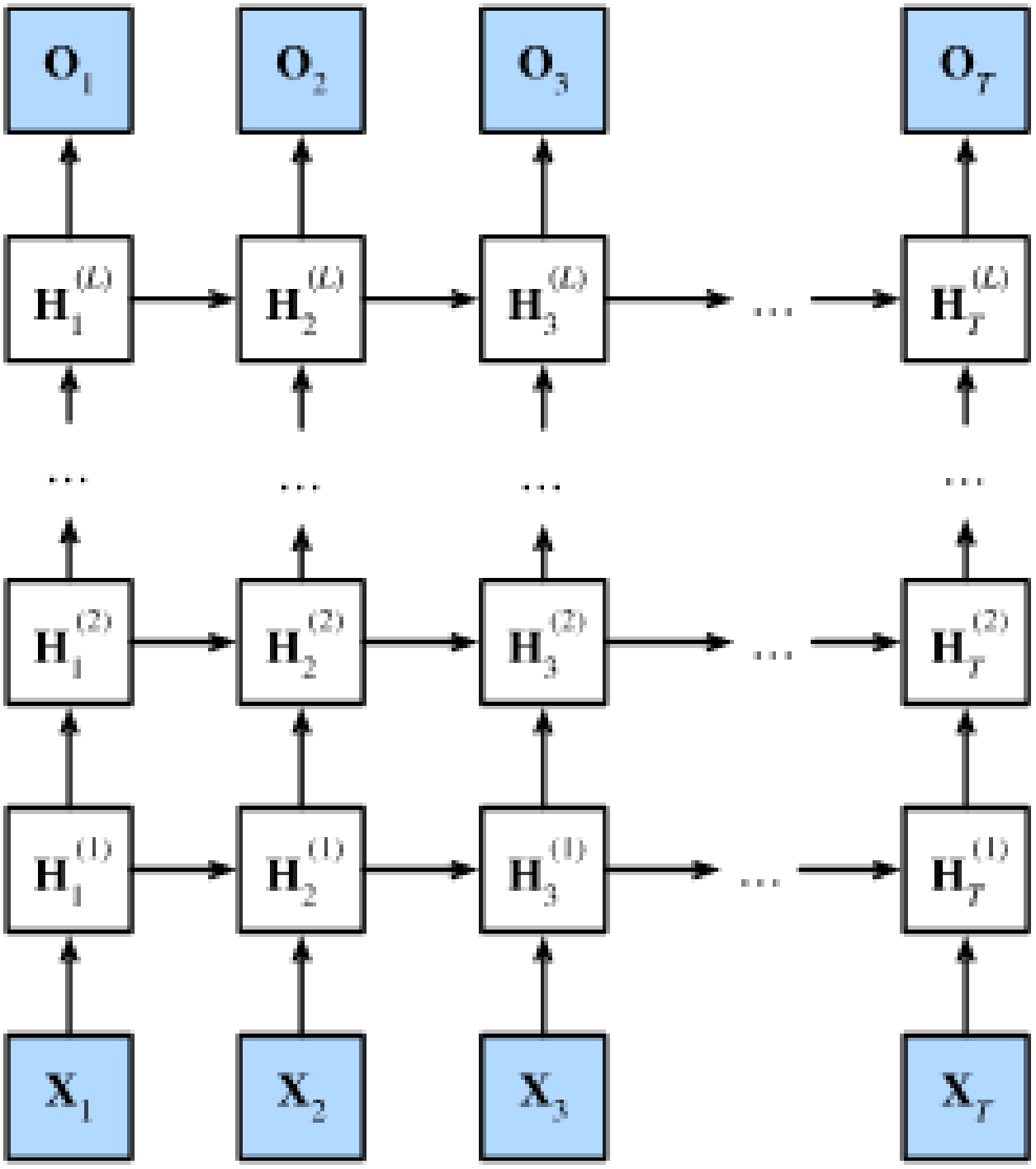
隐状态

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



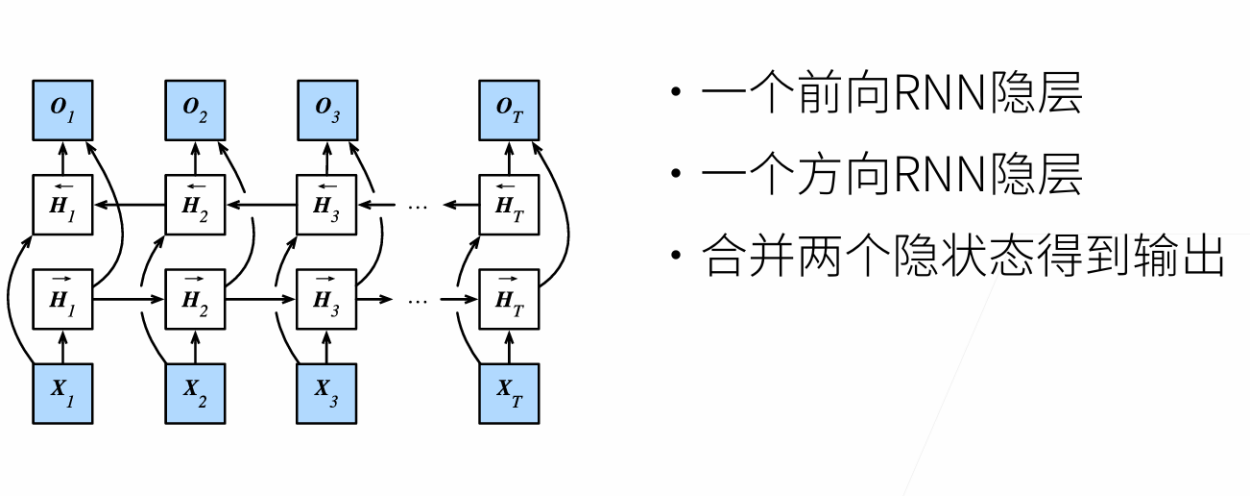
只要输出门接近1，我们就能够有效地将所有记忆信息传递给预测部分， 而对于输出门接近0，我们只保留记忆元内的所有信息，而不需要更新隐状态。（只有隐状态才会传递到输出层， 而记忆元Ct不直接参与输出计算）

9.3 深度循环神经网络


$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\mathbf{H}_1^{(L)}$
$\mathbf{H}_2^{(L)}$
$\mathbf{H}_3^{(L)}$
$\mathbf{H}_T^{(L)}$
$\mathbf{H}_1^{(2)}$
$\mathbf{H}_2^{(2)}$
$\mathbf{H}_3^{(2)}$
$\mathbf{H}_T^{(2)}$
$\mathbf{H}_1^{(1)}$
$\mathbf{H}_2^{(1)}$
$\mathbf{H}_3^{(1)}$
$\mathbf{H}_T^{(1)}$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

9.4 双向循环神经网络

1.RNN只看过去，但我们也可以看未来（完形填空）



2.总结：

- 双向循环神经网络通过反向更新的隐藏层来利用反向时间信息
- 通常用来对序列抽取特征、填空，而不是预测未来（推理）

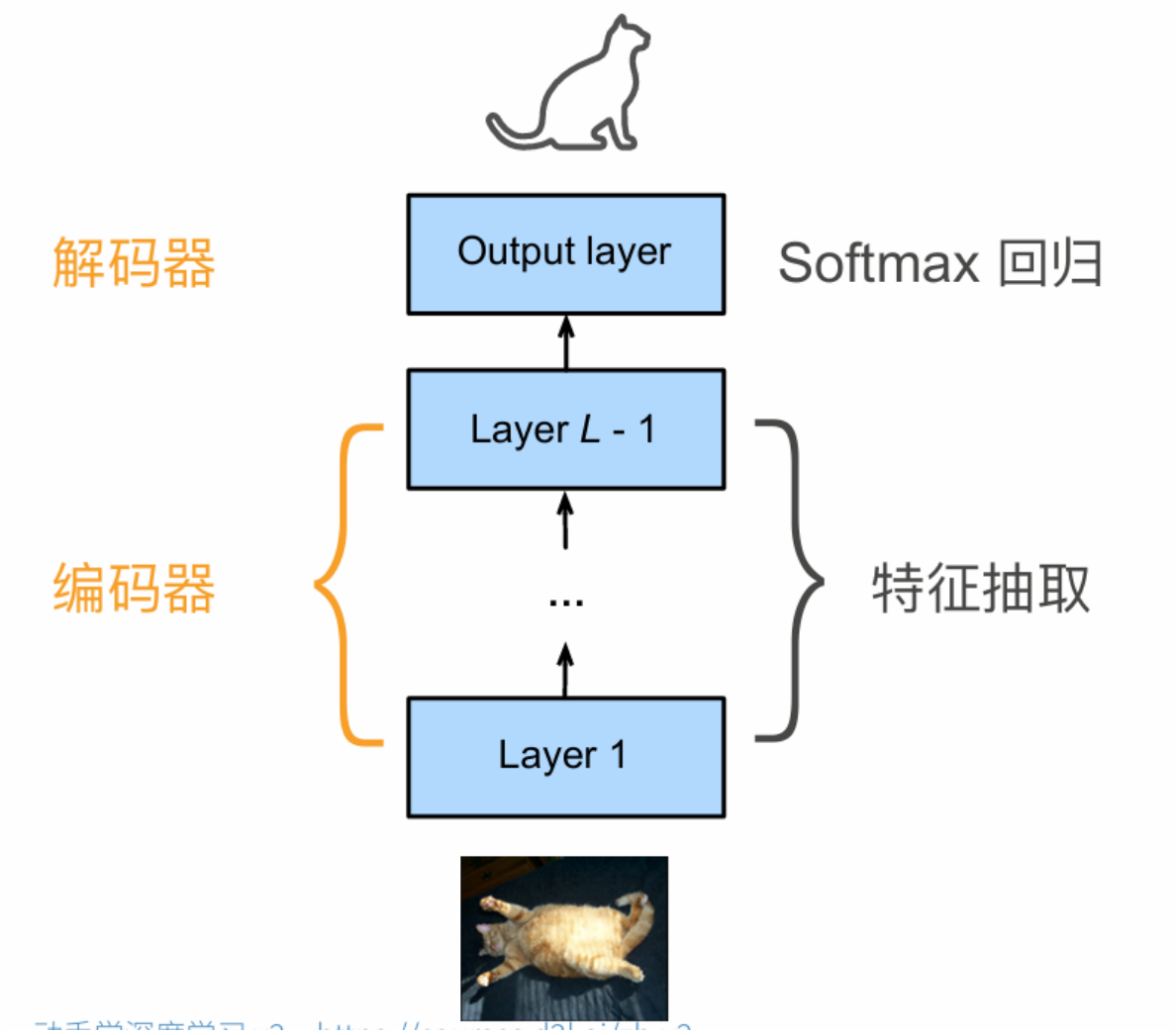
9.5 机器翻译与数据集

- 下载和预处理数据集
- 几个预处理步骤
- 词元化
- 词汇表
- 序列样本都有一个固定的长度 截断或填充文本序列
- 转换成小批量数据集用于训练
- 训练模型

9.6 编码器-解码器架构

1.CNN

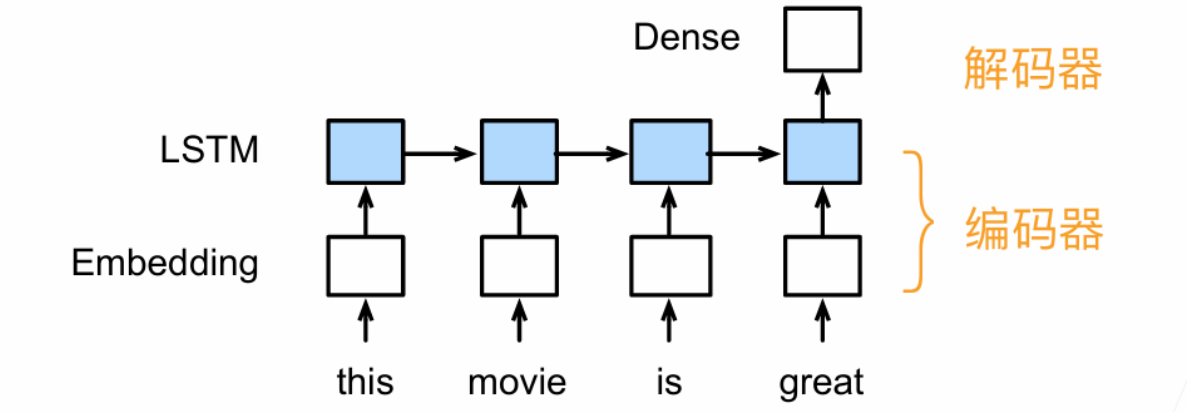
- 编码器：将输入编程成中间表达形式（特征）
- 解码器：将中间表示解码输出



-

2.RNN

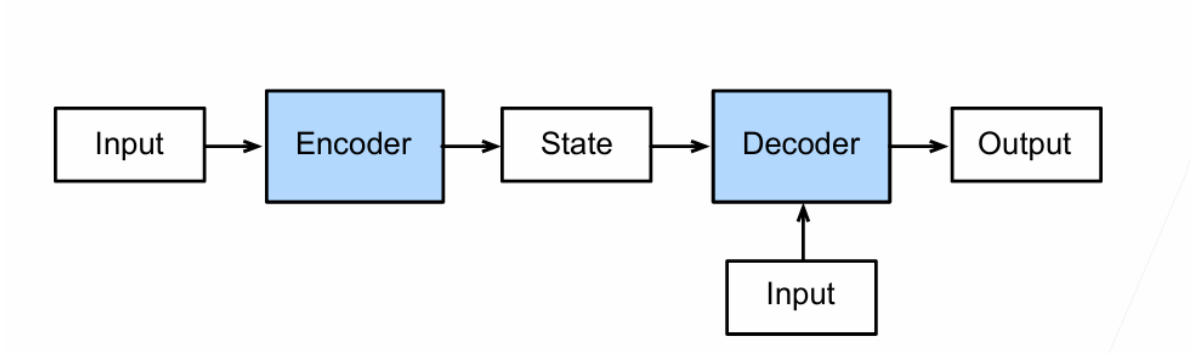
- 编码器：将文本表示成向量
- 解码器：向量表示成输出



-

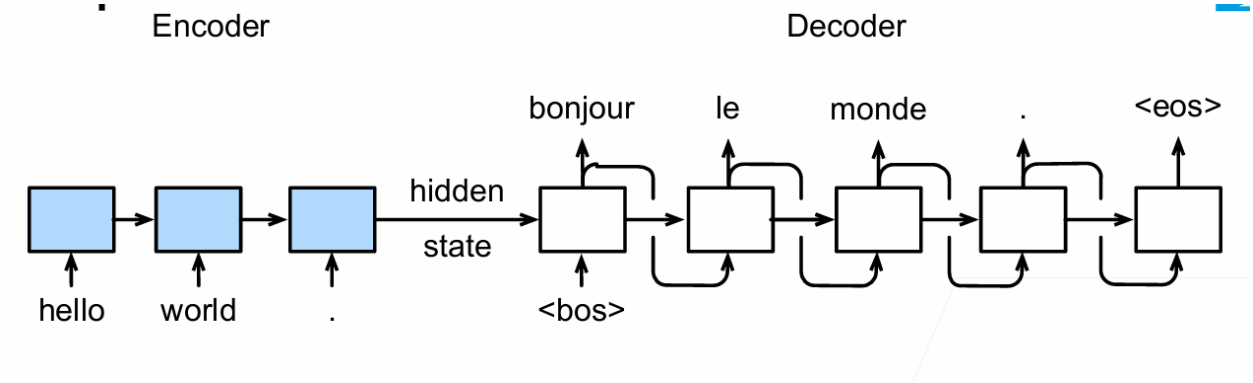
3.架构

- 编码器处理输入
- 解码器处理输出



9.7 序列到序列学习seq2seq

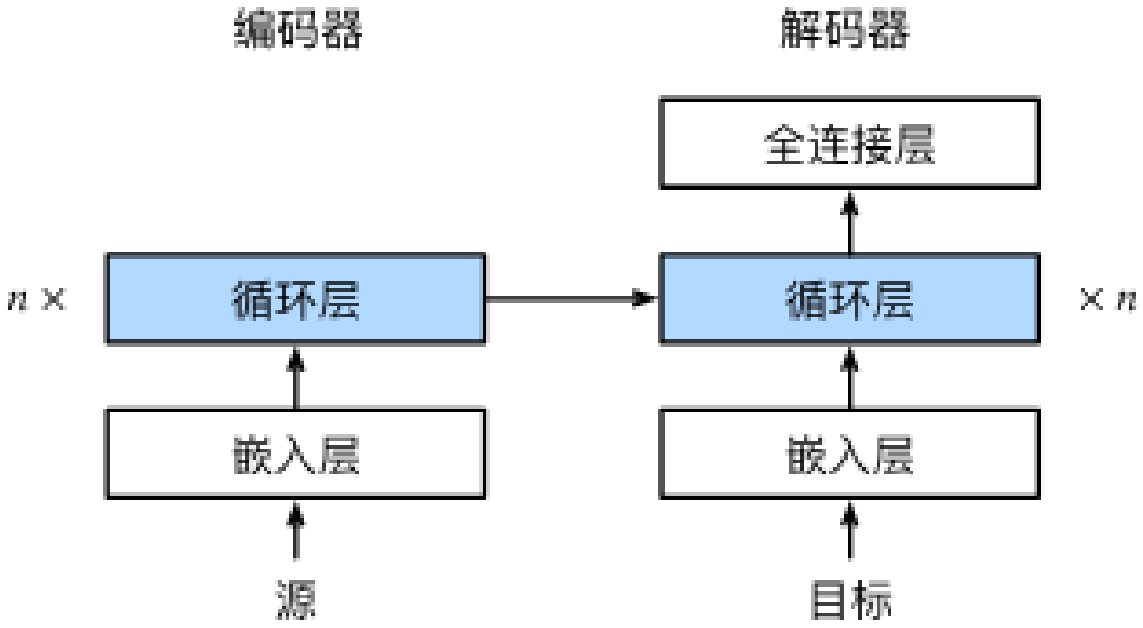
1.seq2seq



- 编码器是一个RNN，读取输入句子（可以是双向）
- 解码器使用另外一个RNN来输出

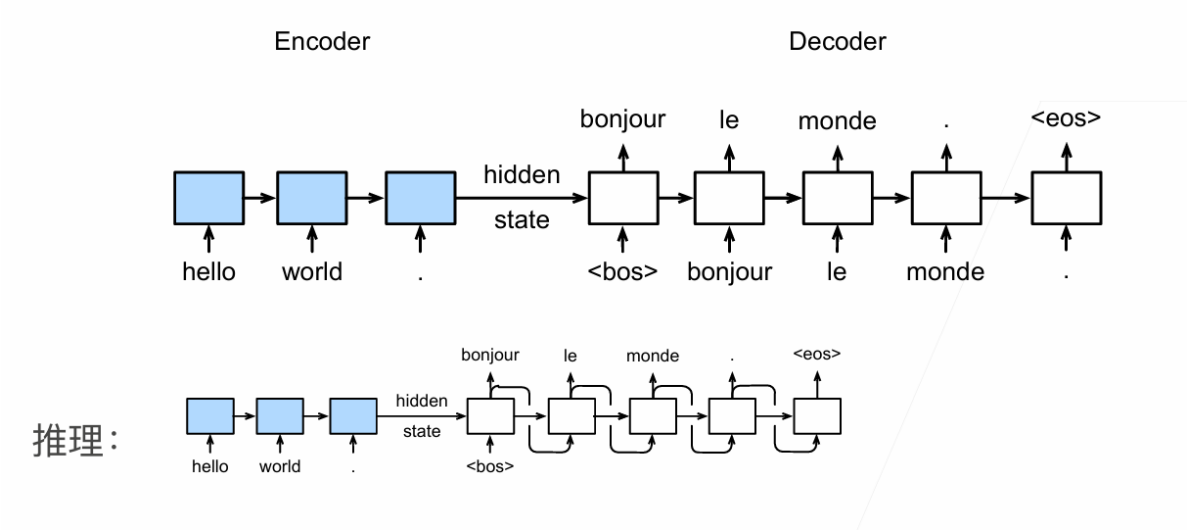
2.细节

- 编码器是没有输出的RNN
- 编码器最后时间步的隐藏状态用作解码器的初始隐藏状态



3.训练

训练时解码器使用目标句子作为输入



4.衡量生成的好坏：BLEU

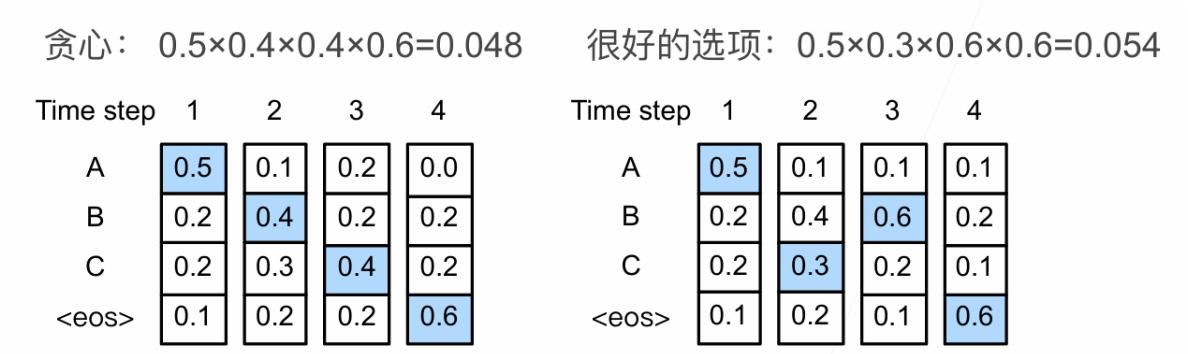
- pn是预测中所有n-gram的精度
  - e.g.:标签序列A B C D E F 和预测序列A B B C D
  - p1=4/5，预测序列中五个只有第二个B没有出现
  - p2=3/4，预测序列中四个2元，BB没出现
  - p3=1/3，p4=0
- BLEU：
  - $exp(min(0, 1 - \frac{len_{label}}{len_{pred}})) \prod_{n=1}^{k} p_n^{1/2^n}$
  - if len label> len pred,后一项为负，那么exp负数就变成很小的数了，BLEU越大越好，最大就是exp0=1，所以min用来惩罚过短的预测

- 后面的连乘 pn<1所以n越大这个乘积越大，即长匹配有高权重

9.8 束搜索

1.贪心搜索

- 在seq2seq中使用了贪心搜索来预测序列，即将当前时刻预测概率最大的词输出
- 但贪心并不一定最优
- e.g.



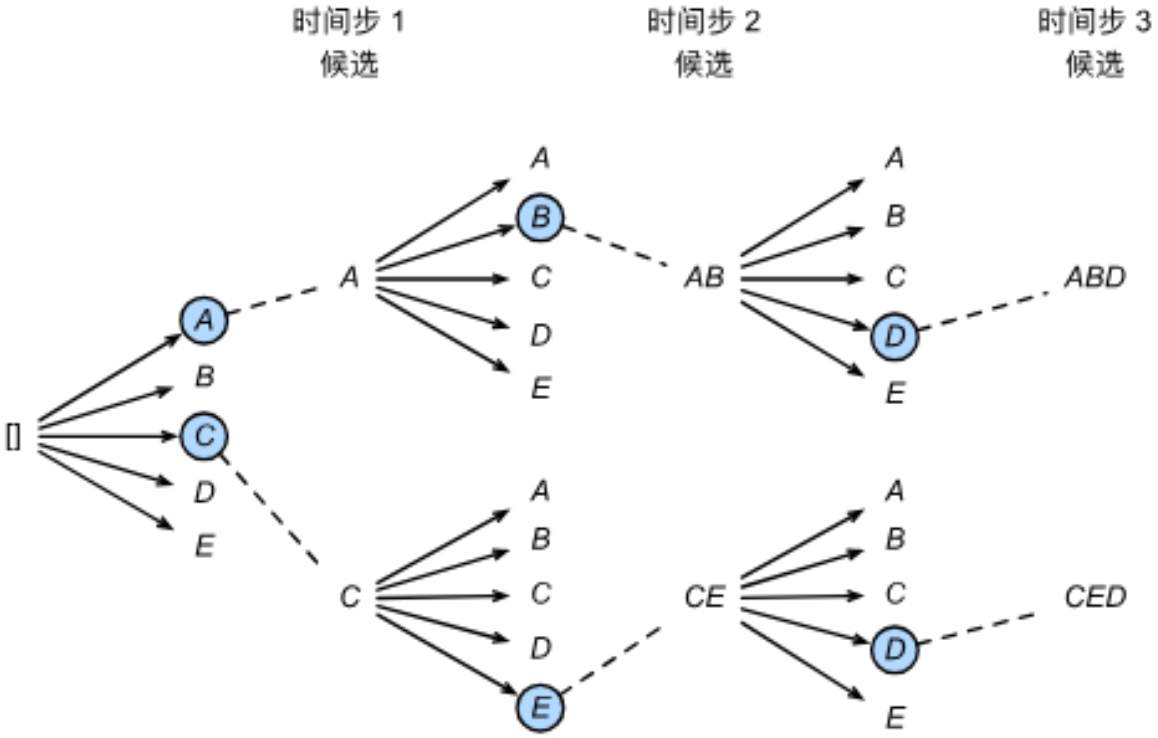
2.穷举搜索

- 最优算法：对所有可能的序列，计算概率，选取最好的
- 但计算上不可行

3.束搜索

- 保存最好的k个候选
- 在每个时刻，对每个候选新加一项（n种可能），在kn中选出最好的k个

- e.g.



- 每个候选的最终分数是：

$$\frac{1}{L^{\alpha}} \log p(y_1, \ldots, y_L) = \frac{1}{L^{\alpha}} \sum_{t'=1}^{L} \log p(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$$

  - 通常 $\alpha = 0.75$

- 长句子的概率越低，为了避免每次都只找短的，所以前面乘了一个东西，L是长度，越长，分母越大，这个玩意儿越小，而log出来是负数，所以整体值越大，相当于对长句子做了补偿