

# 基于循环神经网络的文本分析

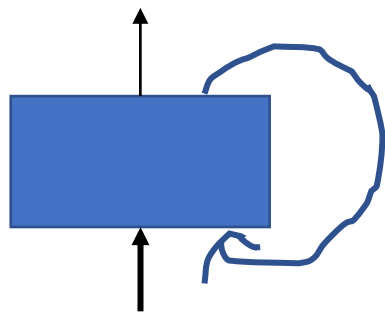
# 主要内容

- 循环神经网络的基本概念与应用
- 门控循环神经网络：LSTM和GRU

# 什么是循环神经网络

## Recurrent Neural Networks (RNNs)

- 循环神经网络：把模型前一时刻的输出反传作为下一时刻的输入



- 用于时序数据分析，基于之前预测未来。比如基于股票之前的价格预测走势，或者预测句子中的下一个词。
- 与前面的学习的模型都不同，循环神经网络可以处理任意长度的句子，而不需要固定长度。
- 在自然语言处理如机器翻译、语音转文本等任务发挥重要作用。

# 为什么要用循环神经网络

- 假设我们要对输入的每个句子进行情感分类。
- 怎么来表示这些句子？

方法1：词袋表示，统计所有句子中包含词的个数作为词典。

局限性：无法捕获词的语义信息和顺序信息。

- 方法2：用词嵌入表示每个词，再取平均得到句子向量。

局限性：依然不能捕获顺序或上下文。

此“神经”是彼“神经”吗



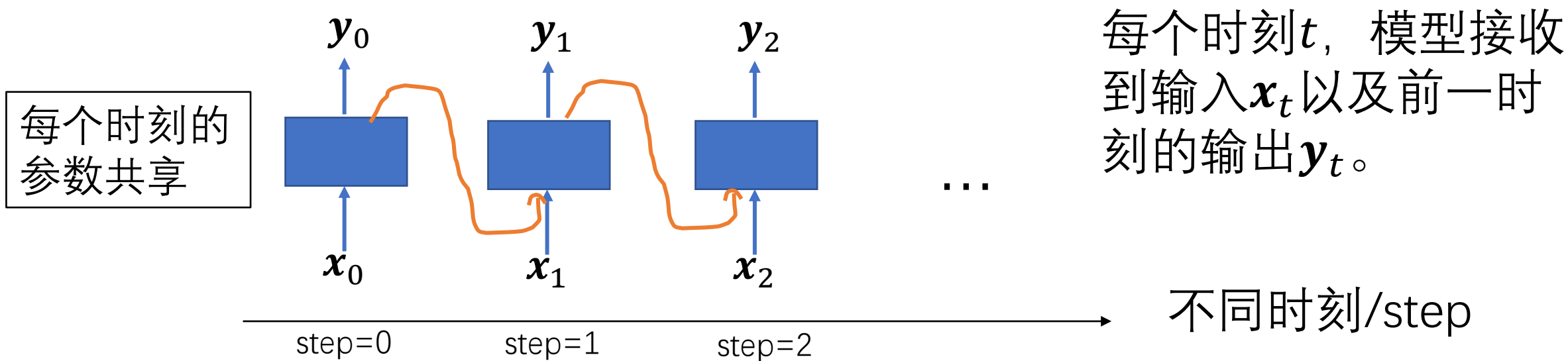
比如：

人的脑**神经**非常发达。当下热门的深度**神经**网络模型就是模仿人脑工作机制提出的。

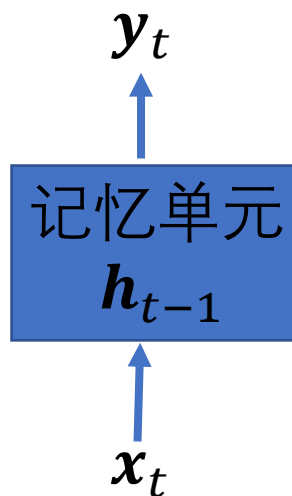
# 为什么要用循环神经网络

- 可以捕获句子中词之间的先后顺序,
- 并可以直接应用于不同长短的句子。

假设一个句子表示为  $x_0, x_1, \dots, x_t$ , 其中  $x_t$  对应该句子中某个词的词向量 (可以是one-hot, 也可以是某种预训练词嵌入向量)。



# 简单RNN



$$h_t = g(h_{t-1}, x_t)$$

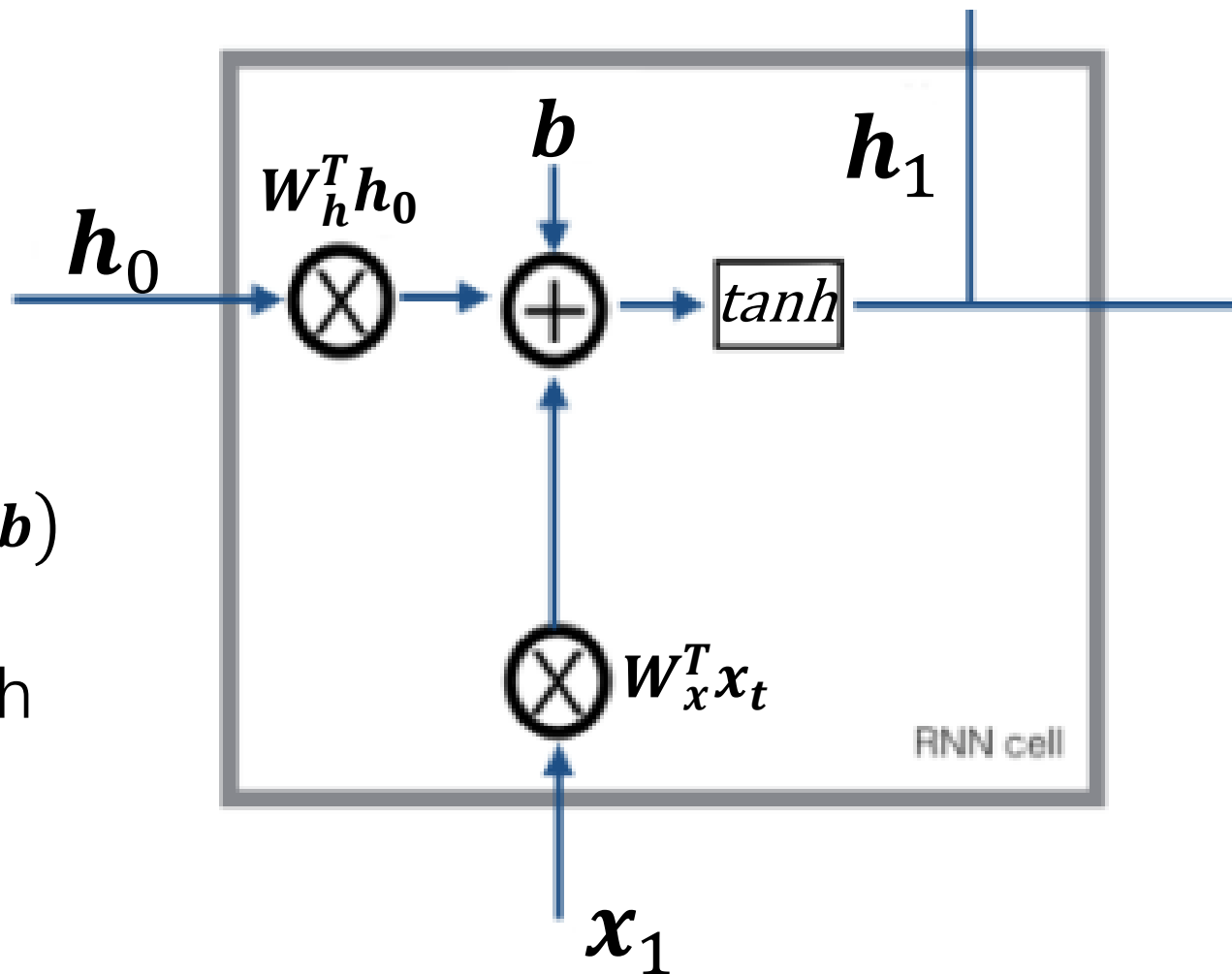
记忆单元保存前一次的隐状态 $h_{t-1}$ , 将其作为当前时刻的输入, 得到当前时刻的输出之后, 记忆单元得到更新。

简单RNN中一般 $y_t = h_t$ , 即把隐状态(hidden state)作为输出。

# 简单RNN

$$h_t = g(W_x^T x_t + W_h^T h_{t-1} + b)$$

一般激活函数 $g$ 选tanh



# 简单RNN

简单记忆单元由单层神经元构成。

$$\begin{aligned} \mathbf{h}_t &= g(\mathbf{W}_x^T \mathbf{x}_t + \mathbf{W}_h^T \mathbf{h}_{t-1} + \mathbf{b}) \\ &= g(\mathbf{W}^T [\mathbf{x}_t \ \mathbf{h}_{t-1}] + \mathbf{b}) \end{aligned} \quad (1)$$

拼接成一个长的向量

其中

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_h \end{bmatrix}$$

一般激活函数 $g$ 选tanh

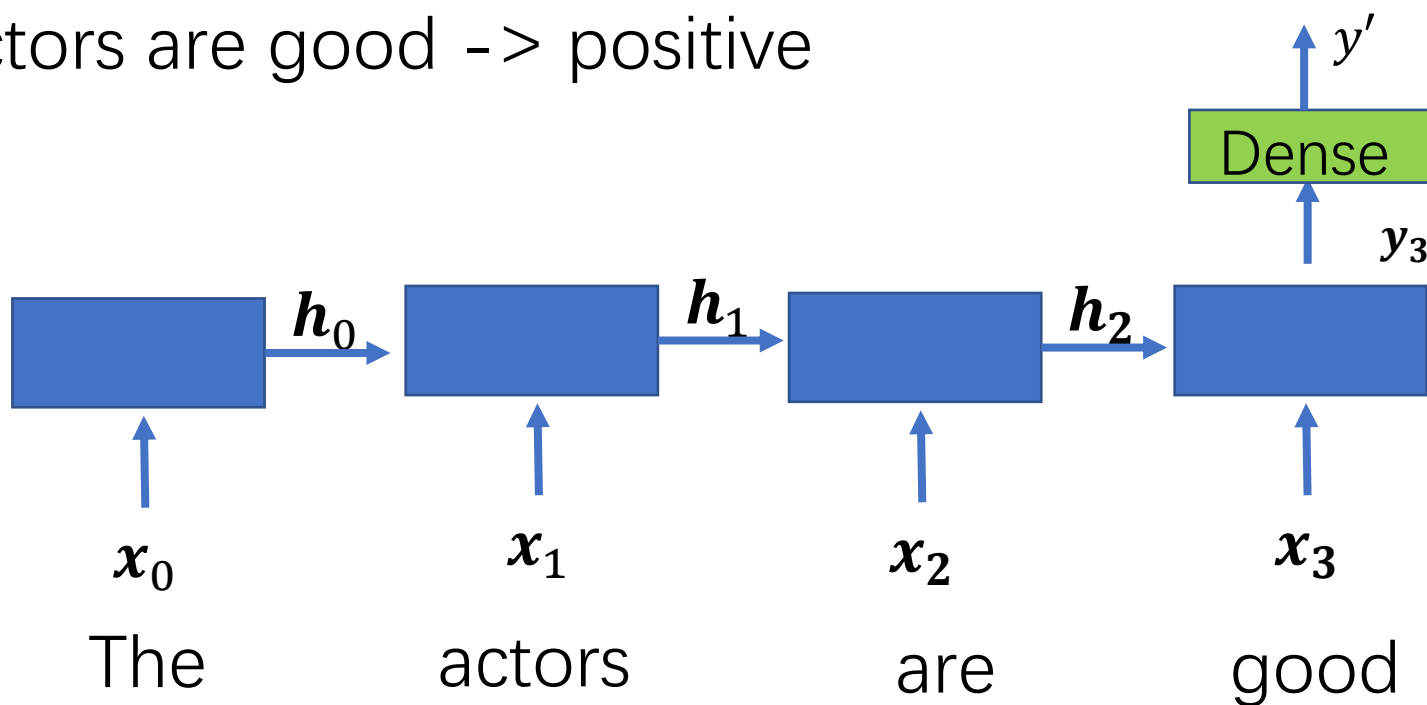
$\mathbf{x}_t$  大小  $m \times 1$ :  $m$ 为每个输入的维度  
 $\mathbf{W}_x$ 大小  $m \times n_{neuros}$ :  $n_{neuros}$ 是隐藏层节点数  
 $\mathbf{W}_y$ 大小  $n_{neuros} \times n_{neuros}$   
 $\mathbf{h}_t$ 和 $\mathbf{b}$ (偏置)大小均为  $n_{neuros} \times 1$



# 应用于分类

- 输入：序列sequence，比如一个句子（词序列）
- 输出：类别

The actors are good -> positive



由具体分类  
任务确定

每个词表示为  
one-hot 向量

对该序列的分类一般用最后一个状态的输出

# 实现： 单层RNN

```
model = keras.models.Sequential ([  
    keras.layers.SimpleRNN(20, input_shape = [None, 1])  
])
```

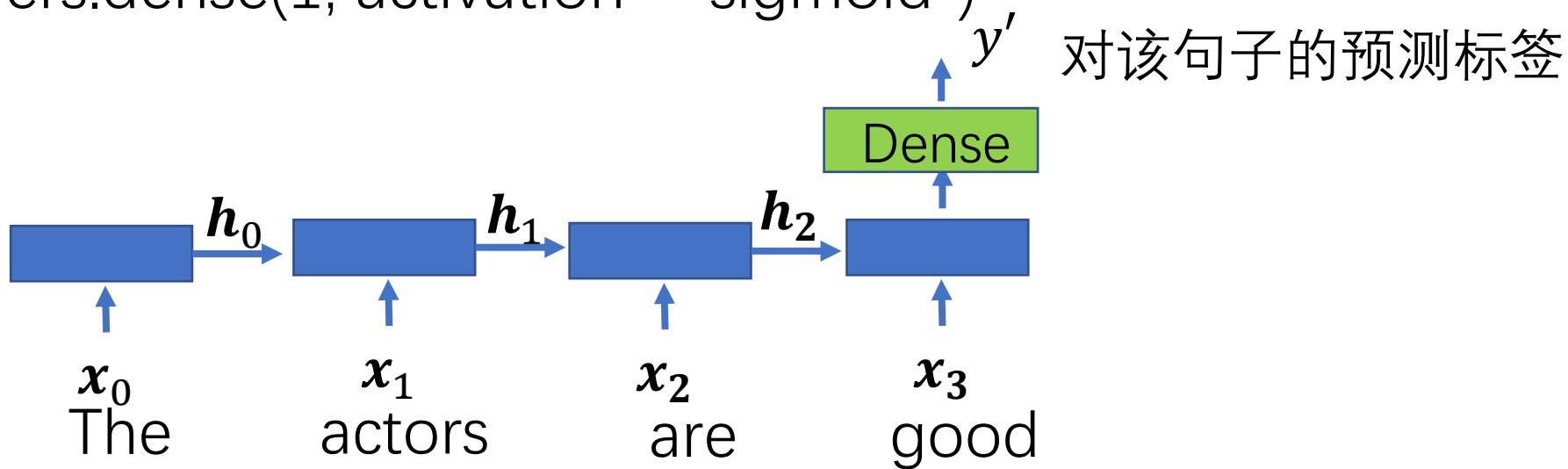
一般 $\mathbf{h}_{init} = \mathbf{0}$

第 $t=0$ 个时刻， $\mathbf{h}_{init}$ 和 $\mathbf{x}_0$ 输入到20个节点(决定hidden state 的维度)的简单RNN，根据公式（1）得到 $\mathbf{h}_0$ ，进入第 $t=1$ 个时刻，把 $\mathbf{h}_0$ 和 $\mathbf{x}_1$ 输入计算 $\mathbf{h}_1$ 。

重复以上过程，直到最后一个输出。

# 实现：单层RNN，二分类

```
model = keras.models.Sequential ([  
    keras.layers.SimpleRNN(20, input_shape = [None, 1])  
    keras.layers.dense(1, activation = "sigmoid")  
])
```



把RNN最后一个状态输出到一个全连接（1个节点）得到预测值，计算损失后反传梯度对模型参数进行更新。


# 简单循环神经网络的局限性

- 存在梯度消失和梯度爆炸问题。
- 隐状态存储了历史信息，使得RNN具有记忆能力，但是状态在每个时刻都会被更新，因此该记忆是短期的(short term memory)。
- 为了提高循环神经网络的记忆周期，即增加对长期记忆能力，提出了很多改进，基于门控机制的方法，LSTM和GRU是其中最流行性的两种。

# 线性+非线性

简单RNN中即公式（1）隐状态 $\mathbf{h}_t$ 与 $\mathbf{h}_{t-1}$ 为非线性关系，导致梯度消失问题。但如果直接改成线性关系，如 $\mathbf{h}_t = \mathbf{h}_{t-1} + g(\mathbf{x}_t; \theta)$ ，又丢失了非线性激活的性质，降低了模型的表达能力。 $\theta$ 表示所有模型参数，包括权重和偏置。

采用线性和非线性结合的方式：

$$\mathbf{h}_t = \mathbf{h}_{t-1} + g(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$$


# LSTM

门控机制，通过门向量与状态元素相乘来控制状态中每个元素的输出。不同于逻辑单元中的开关，这里的门是一种“软门”，即取值在(0,1)之间。

在LSTM中，引入内部状态 $\mathbf{c}_t$ 专门进行线性的循环信息传递，同时（非线性地）输出信息给外部状态 $\mathbf{h}_t$ ：

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

符号 $\odot$ 表示向量的元素相乘。

以上式子中  $\tilde{\mathbf{c}}_t$  表示当前时刻的候选内部状态，其他三个为不同的门。

# LSTM

$$\begin{aligned} \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \end{aligned}$$

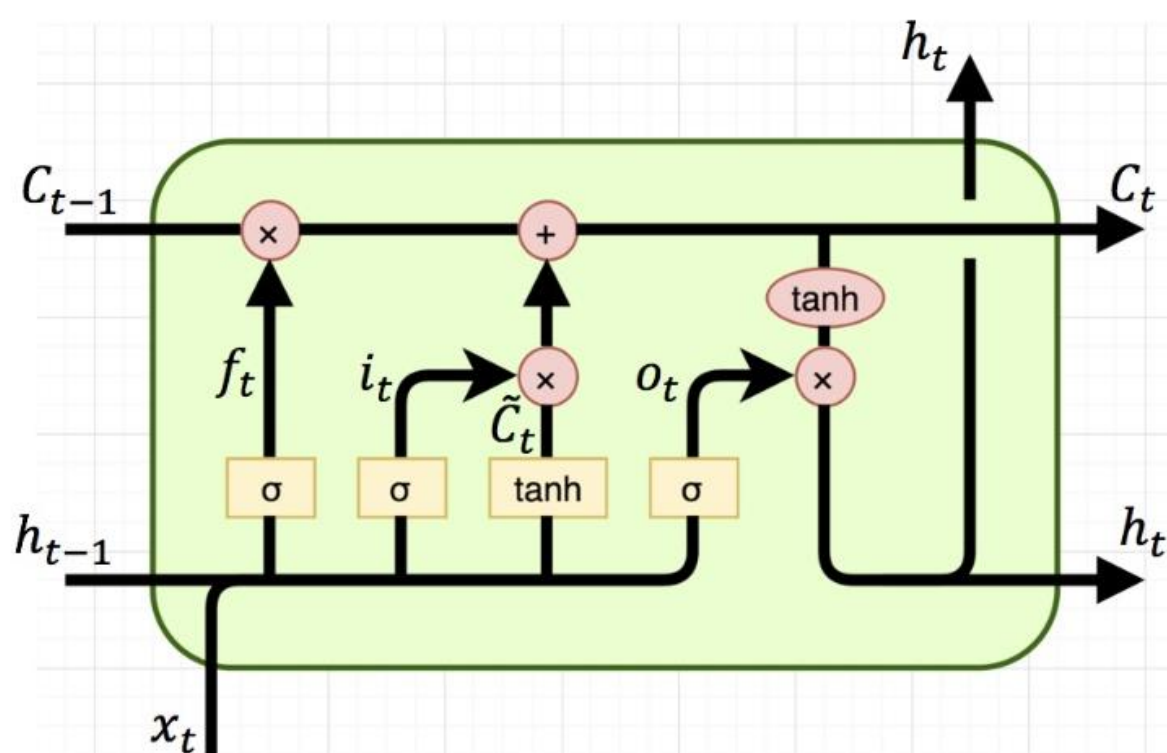
候选内部状态

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c^T \mathbf{x}_t + \mathbf{U}_c^T \mathbf{h}_{t-1} + \mathbf{b}_c)$$

$\mathbf{f}_t$ 为遗忘门：控制上一个时刻的内部状态  
 $\mathbf{c}_{t-1}$ 需要遗忘多少信息。

$\mathbf{i}_t$ 为输入门：控制当前时刻的候选状态  
 $\tilde{\mathbf{c}}_t$ 有多少信息要保存。

$\mathbf{o}_t$ 为输出门：控制当前时刻的内部状态经过非线性激活后有多少信息需要输出给外部状态。



分别用下面三个式子计算：

$$\mathbf{f}_t = \sigma(\mathbf{W}_f^T \mathbf{x}_t + \mathbf{U}_f^T \mathbf{h}_{t-1} + \mathbf{b}_f)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i^T \mathbf{x}_t + \mathbf{U}_i^T \mathbf{h}_{t-1} + \mathbf{b}_i)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o^T \mathbf{x}_t + \mathbf{U}_o^T \mathbf{h}_{t-1} + \mathbf{b}_o)$$

# GRU

引入更新门 $\mathbf{z}_t$ 来控制当前状态需要从历史状态中保留多少信息，以及从候选状态中接收多少新信息。

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t$$

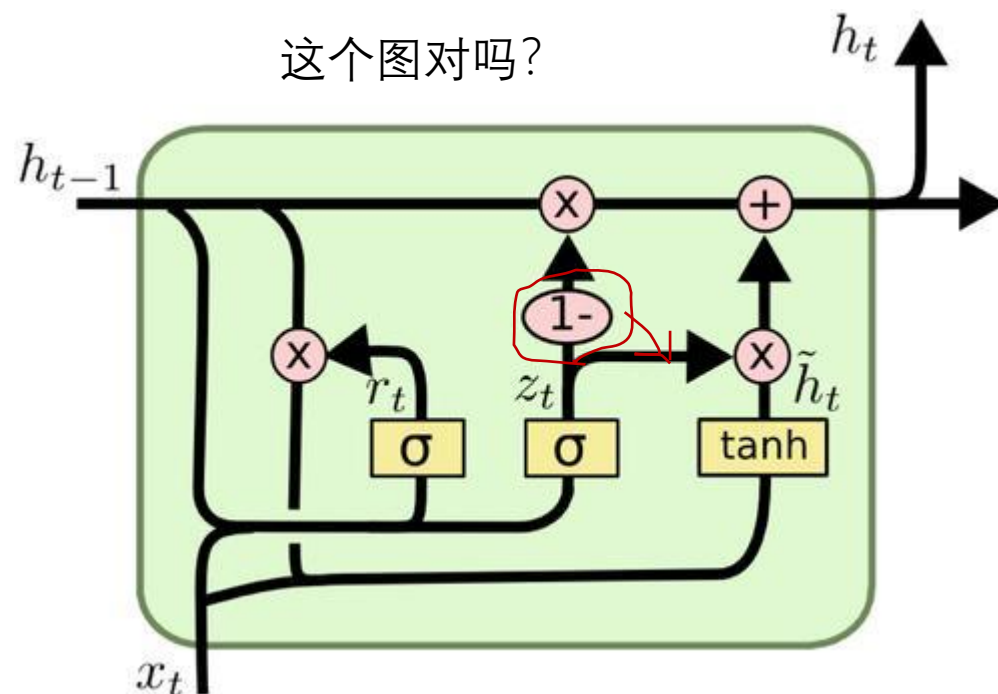
$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h^T \mathbf{x}_t + \mathbf{U}_h^T (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h)$$

更新门

$$\mathbf{z}_t = \sigma(\mathbf{W}_z^T \mathbf{x}_t + \mathbf{U}_z^T \mathbf{h}_{t-1} + \mathbf{b}_z)$$

重置门

$$\mathbf{r}_t = \sigma(\mathbf{W}_r^T \mathbf{x}_t + \mathbf{U}_r^T \mathbf{h}_{t-1} + \mathbf{b}_r)$$



对比LSTM

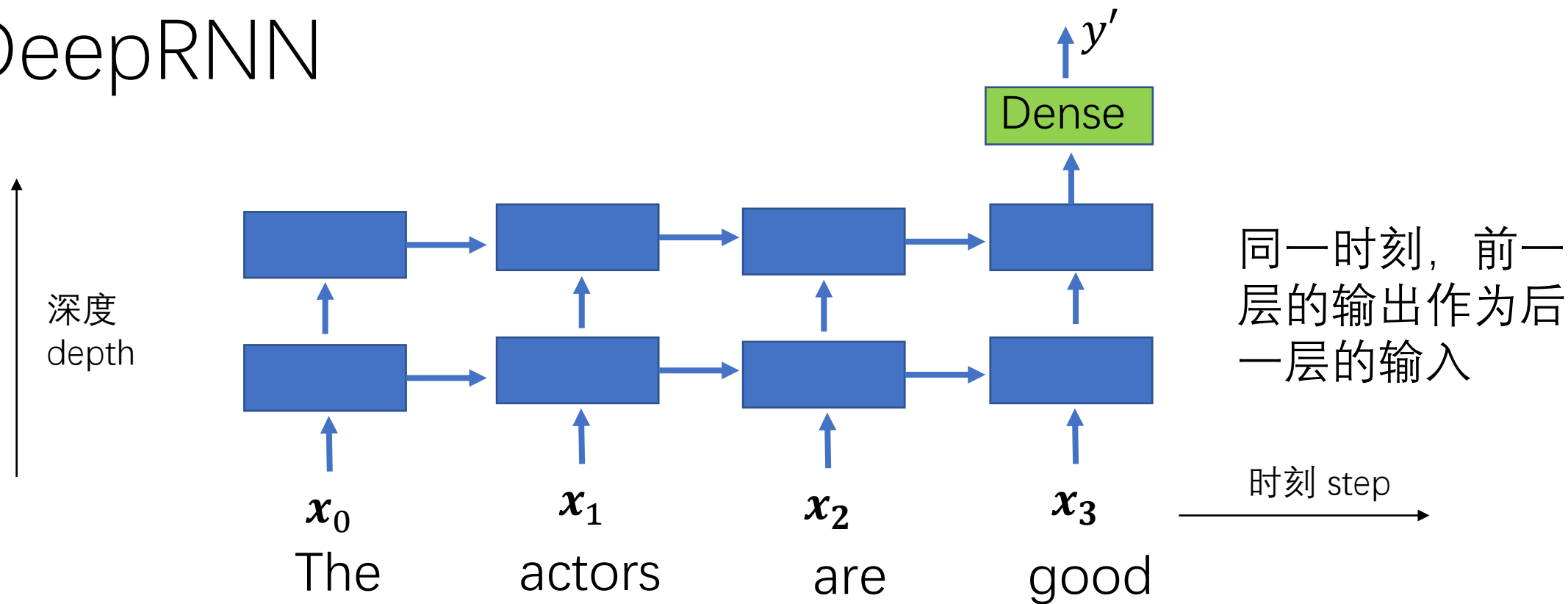
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c^T \mathbf{x}_t + \mathbf{U}_c^T \mathbf{h}_{t-1} + \mathbf{b}_c)$$



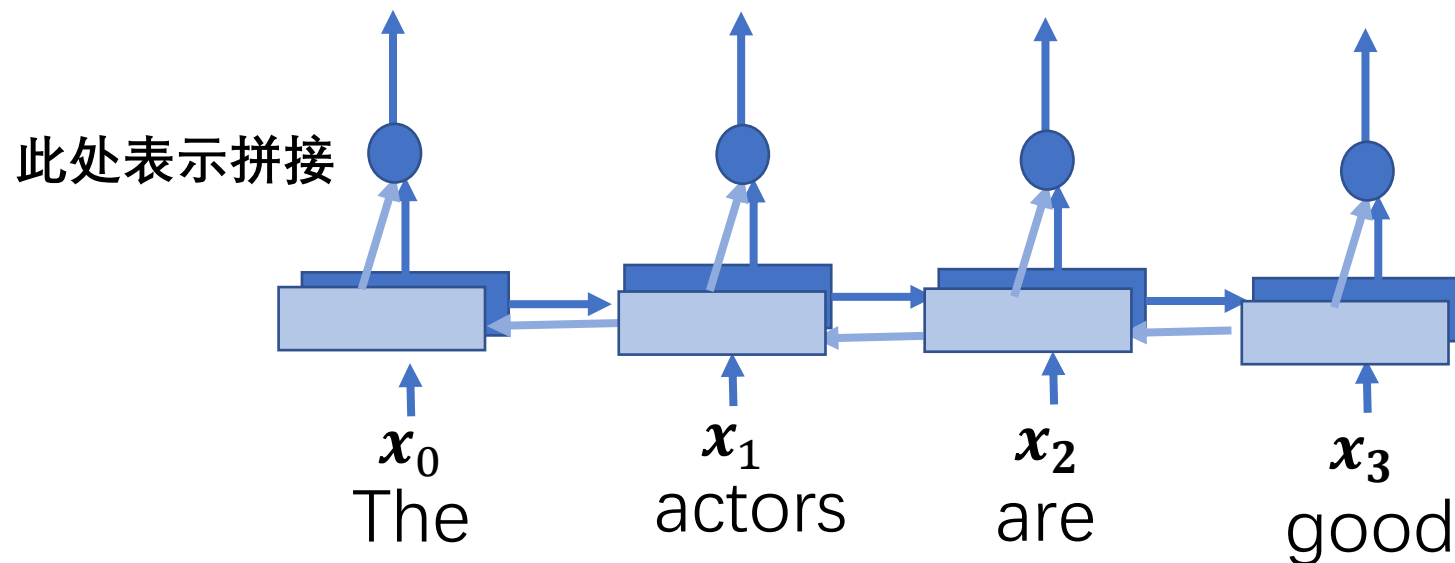
# DeepRNN



```
model = keras.models.Sequential ([  
    keras.layers.SimpleRNN(20, return_sequence = True, input_shape = [None, 1]),  
    keras.layers.SimpleRNN(20),  
    keras.layers.dense(1, activation = "sigmoid")  
])
```

需要返回所有step时的输出时

# 双向 RNN



1. 两个**独立**的RNN负责不同方向的顺序。一个前往后，另一个从后往前。

即两个的输入为同一个，但输入顺序不同。

2. 把两个方向的RNN的输出拼接起来

```
keras.layers.Bidirectional(keras.layers.GRU(20, input_shape = [None, 1]))
```

# 实例：IMDB情感分类

## IMDB数据集

包含50000条英文电影评论，来自[Internet Movie Database](https://www.imdb.com/)  
每条评论有一个0或1的标签，分别代表该评论为负面或正面。

从keras.datasets载入该数据（已经分成25000条训练，25000测试）。

```
▶ (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
```

# 直接导入预处理好的数据

打印训练集中第一个记录的前面10个元素（每个元素对应一个词的id）

```
▶ X_train[0][:10]
```

```
0]: [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

把id转成对应的单词

```
▶ word_index = keras.datasets.imdb.get_word_index()
id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
    id_to_word[id_] = token
" ".join([id_to_word[id_] for id_ in X_train[0][:10]])
```

所有标点被移除，字母转成小写，空格作为分割，以出现次数从高到低进行索引，即出现次数越多的词对应的id越小。

id=0,1,2的为特殊记号:  
id=0: 填充 (pad),  
id=1: 序列的开始(sos) ,  
id=2: 未知单词(unknown)。

```
'<sos> this film was just brilliant casting location scenery story'
```

# 从原始数据进行预处理

## 从tensorflow导入原始数据

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
```

```
datasets.keys()
```

```
dict_keys(['test', 'train', 'unsupervised'])
```

```
train_size = info.splits["train"].num_examples
test_size = info.splits["test"].num_examples
```

```
train_size, test_size
```

```
(25000, 25000)
```

# 从原始数据进行预处理

## 两条原始评论

```
for X_batch, y_batch in datasets["train"].batch(2).take(1):  
    for review, label in zip(X_batch.numpy(), y_batch.numpy()):  
        print("Review:", review.decode("utf-8")[:200], "...")  
        print("Label:", label, "= Positive" if label else "= Negative")  
        print()
```

Review: This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Iron  
oth are great actors, but this must simply be their worst role in history. Even their great acting ..  
Label: 0 = Negative

Review: I have been known to fall asleep during films, but this is usually due to a combination of thi  
luding, really tired, being warm and comfortable on the sette and having just eaten a lot. However ..  
Label: 0 = Negative

# 从原始数据进行预处理

```
def preprocess(X_batch, y_batch):  
    X_batch = tf.strings.substr(X_batch, 0, 300)  
    X_batch = tf.strings.regex_replace(X_batch, rb"<br\s*/?>", b" ")  
    X_batch = tf.strings.regex_replace(X_batch, b"['^a-zA-Z"]", b" ")  
    X_batch = tf.strings.split(X_batch)  
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

- 1.把每个输入截断，只包含300个字符，使得长度一样
  2. 把<br \>用空格代替
  3. 非字母和单引号用空格代替
  - 4.用空格作为分割
- 返回对应张量（对所有评论进行padding，最终长度一样）

# 从原始数据进行预处理

- 建立词汇表

```
from collections import Counter

vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

```
vocabulary.most_common()[:3]
```

```
[(b'<pad>', 214309), (b'the', 61137), (b'a', 38564)]
```

```
len(vocabulary)
```

```
53893
```

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

只保留出现次数最多的10000个词



# 从原始数据进行预处理

- 建立词汇表

```
word_to_id = {word: index for index, word in enumerate(truncated_vocabulary)}  
for word in b"This movie was faaaaaantastic".split():  
    print(word_to_id.get(word) or vocab_size)
```

22

12            词汇表中的词的id为0-9999，不在词汇表里面的词的id为10000

11

10000

```
words = tf.constant(truncated_vocabulary)  
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)  
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)  
num_oov_buckets = 1000  
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

建立1000个OOV桶，  
把OOV词映射到其中一个  
桶，id从10000到10999。

```
table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
```

```
<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[ 22,  12,  11, 10053]])>
```

# 实例：IMDB情感分类

```
def encode_words(X_batch, y_batch):  
    return table.lookup(X_batch), y_batch
```

```
train_set = datasets["train"].repeat().batch(32).map(preprocess)  
train_set = train_set.map(encode_words).prefetch(1)
```

```
for X_batch, y_batch in train_set.take(1):  
    print(X_batch)  
    print(y_batch)
```

```
tf.Tensor(  
[[ 22  11  28 ...  0  0  0]  
 [  6  21  70 ...  0  0  0]  
 [4099 6881  1 ...  0  0  0]  
 ...  
 [ 22  12 118 ... 331 1047  0]  
 [1757 4101 451 ...  0  0  0]  
 [3365 4392  6 ...  0  0  0]], shape=(32, 60), dtype=int64)  
tf.Tensor([0 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0], shape=(32,), dtype=int64)
```

# 实例：IMDB情感分类

初始Embedding也可以直接是某种预训练好的词向量

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           mask_zero=True, # 忽略padding
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

先用[embedding层](#)对索引表示的句子进行线性变换得到初始embedding，把每个词表示为128维向量，然后再依次输入2层GRU，把第2层GRU的最后一个输出再经过一个单个节点的全连接得到分类结果。

# 长度不同的句子

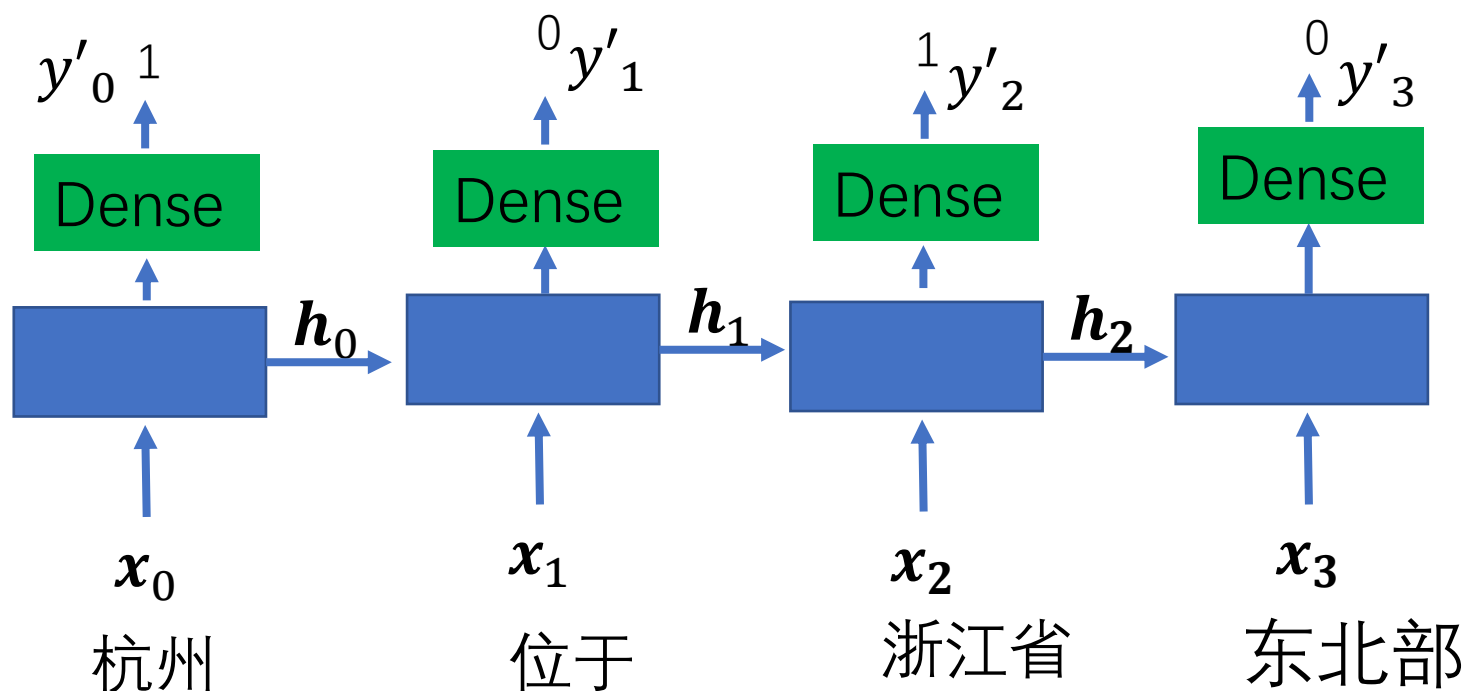
- 循环网络可以处理长度不同的句子（单个输入时）
- 但是训练时需要基于mini-batch的输入，同一个batch内的句子必须长度一样，否则不同用张量表示
- 有些问题上不能用截断，比如机器翻译
- 把长度相近的句子打包在一个batch，然后进行padding使得长度一样。

解决了技术性问题，但怎么处理padding这些的结果？

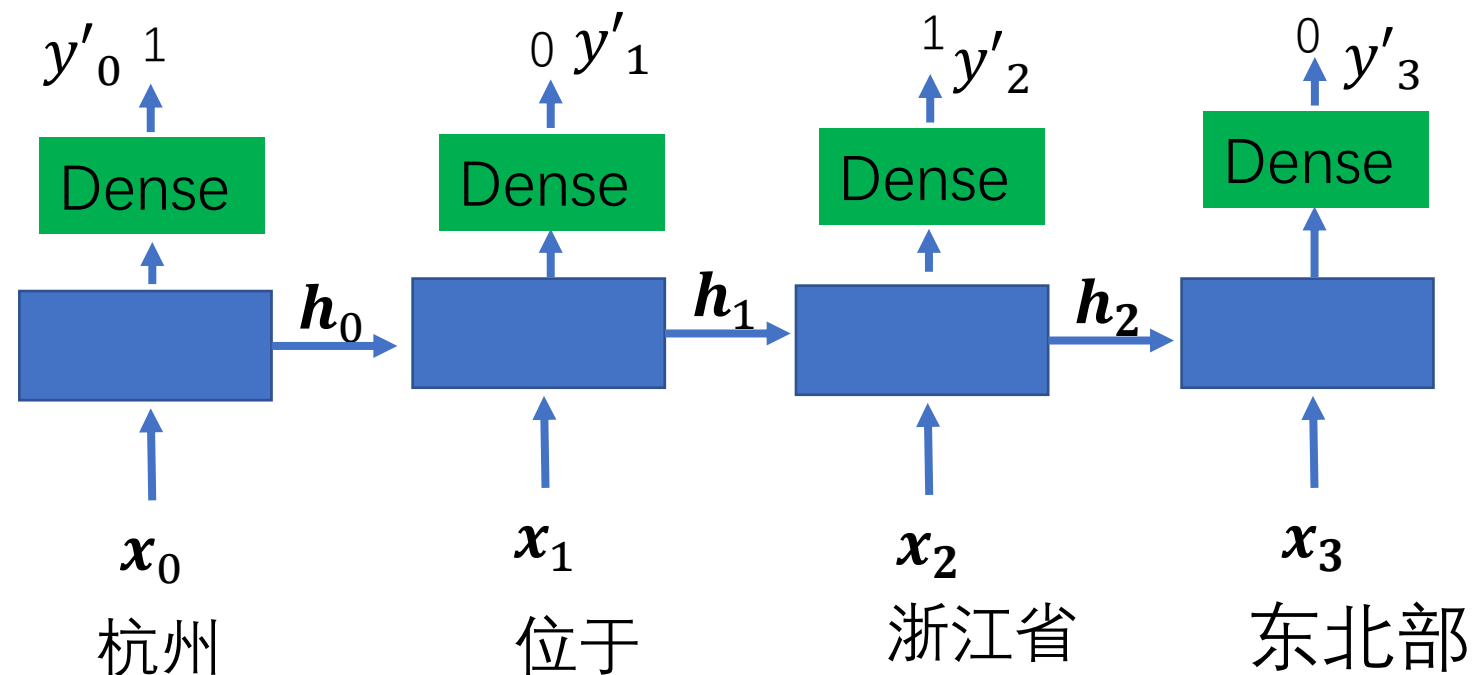
设置mask\_zero=True，后面的层都忽略padding（id=0）的token。

# Sequence to sequence 问题：命名实体识别

- 输入：向量序列，比如一个句子（词序列，每个词为一个向量）
- 输出：等长标量序列。每个词一个二分类标签（1表示这个词是某个命名实体的一部分，否则0）。



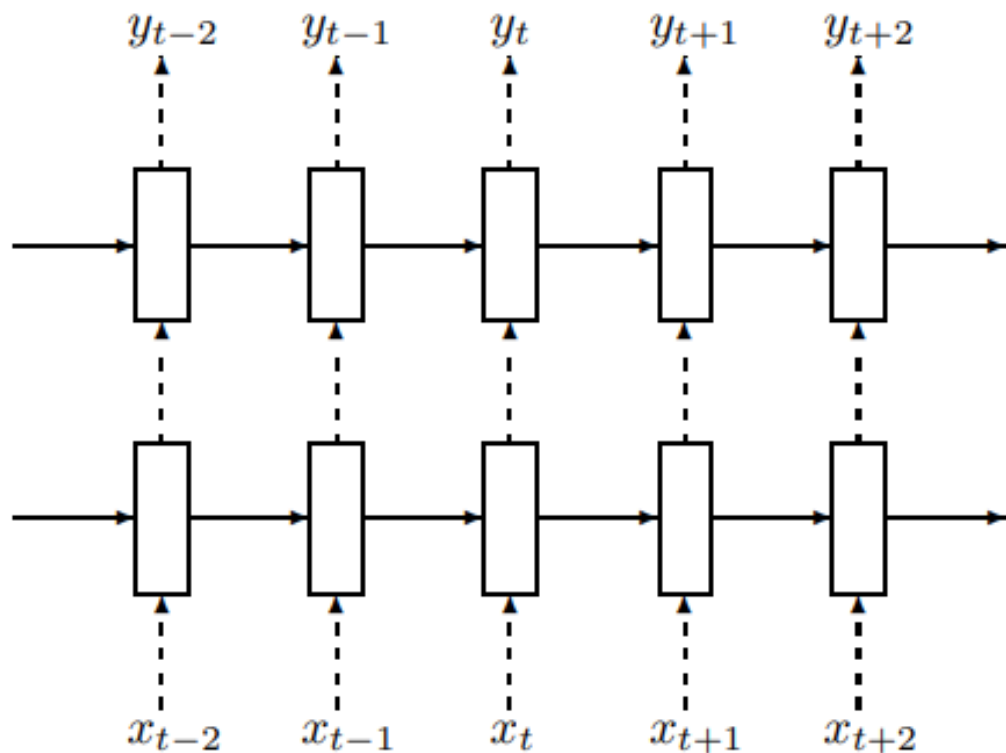
# 命名实体识别



整个序列的损失:

$$loss = \sum_{t=0}^{step} l(y_t, y'_t)$$

# RNN中的dropout



为了不影响在时间维度上的记忆能力，只在非时间维度方向（深度）对**连接**进行dropout。

并且在不同时刻共享同一种掩码。