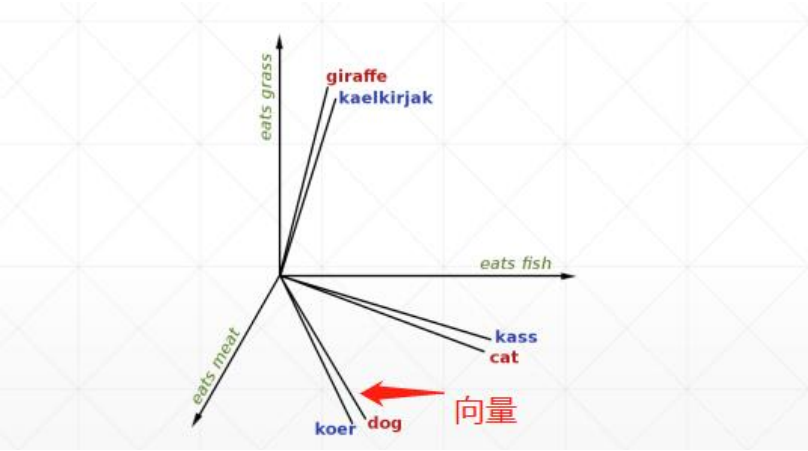
11 循环神经网络 RNN

Embedding Layer

将单词隐射到向量空间里，来比较单词的向量。

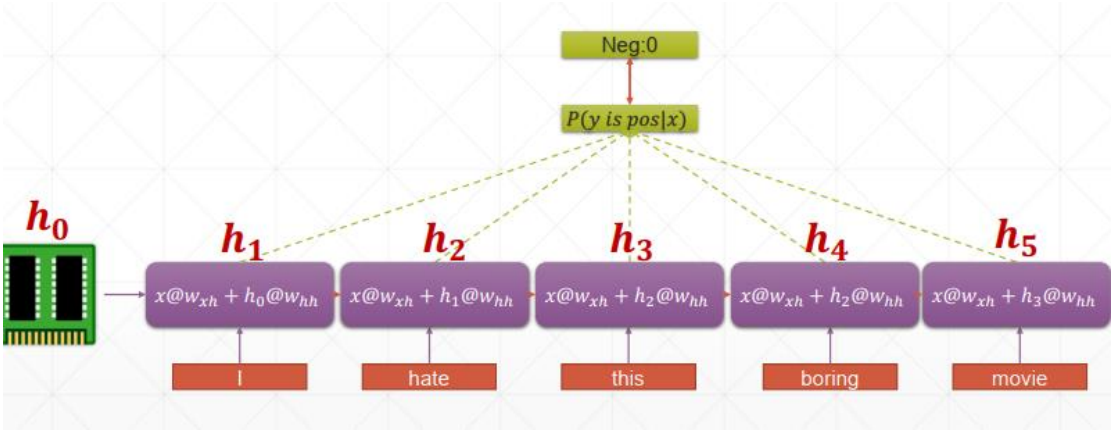


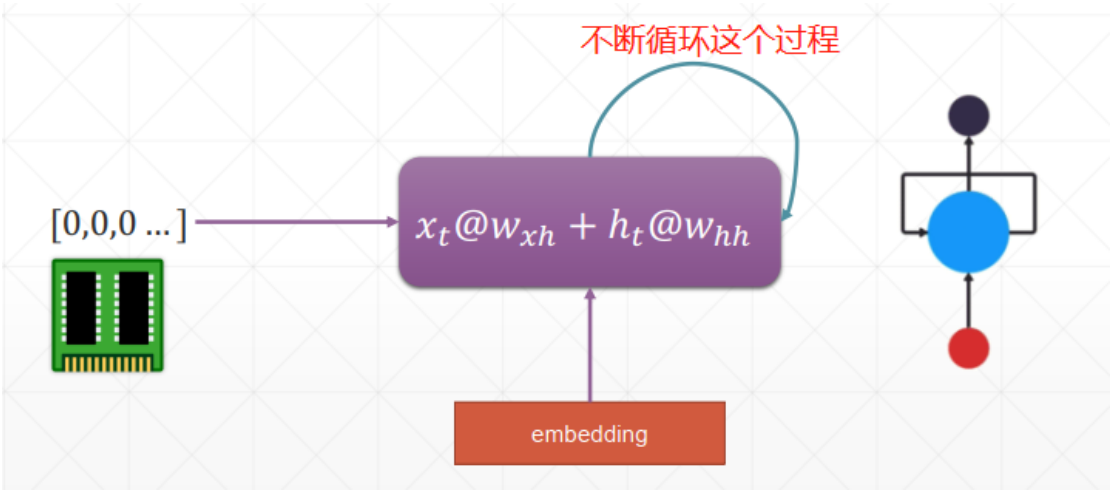
- Random initialized embedding

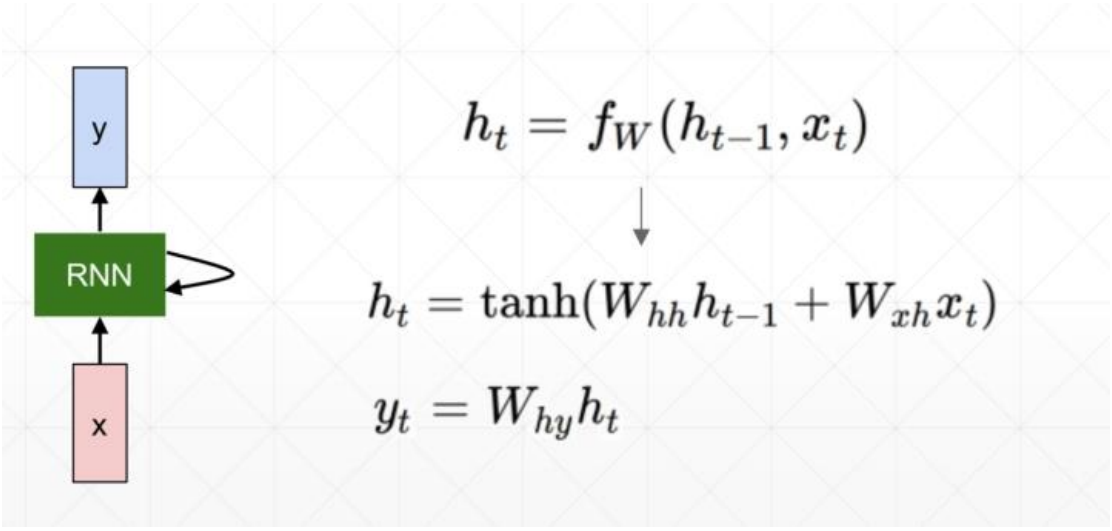
```
In [8]: from tensorflow.keras import layers
In [3]: x=tf.range(5)
In [5]: x=tf.random.shuffle(x)
Out[6]: <tf.Tensor: id=9, shape=(5,), dtype=int32, numpy=array([0, 4, 1, 3, 2])>

In [9]: net=layers.Embedding(10, 4)
In [10]: net(x)
<tf.Tensor: id=25, shape=(5, 4), dtype=float32, numpy=
array([[-0.03535435,  0.01710499,  0.024379  , -0.010867  ],
       [-0.03977622,  0.01753286,  0.00805125,  0.00836002],
       [-0.0119989 ,  0.01030685, -0.01133521,  0.02052242],
       [ 0.02230989, -0.02186236,  0.01720804, -0.03888531],
       [-0.04997355,  0.00911248,  0.01886252,  0.01570504]],
      dtype=float32)>
```

```
In [11]: net.trainable
Out[11]: True

In [12]: net.trainable_variables
[<tf.Variable 'embedding/embeddings:0' shape=(10, 4) dtype=float32, numpy=
 array([[-0.03535435,  0.01710499,  0.024379  , -0.010867  ],
        [-0.0119989 ,  0.01030685, -0.01133521,  0.02052242],
        [-0.04997355,  0.00911248,  0.01886252,  0.01570504],
        [ 0.02230989, -0.02186236,  0.01720804, -0.03888531],
        [-0.03977622,  0.01753286,  0.00805125,  0.00836002],
        [-0.03328228,  0.04350821, -0.03760867, -0.02835478],
        [ 0.03658437, -0.00686272, -0.00069226,  0.02232203],
        [-0.02224611, -0.00517293, -0.00249453,  0.00809779],
        [ 0.00234641, -0.01876179, -0.03804705, -0.02393213],
        [-0.01690916, -0.04798424, -0.04477951, -0.03771662]],
       dtype=float32)>]
```

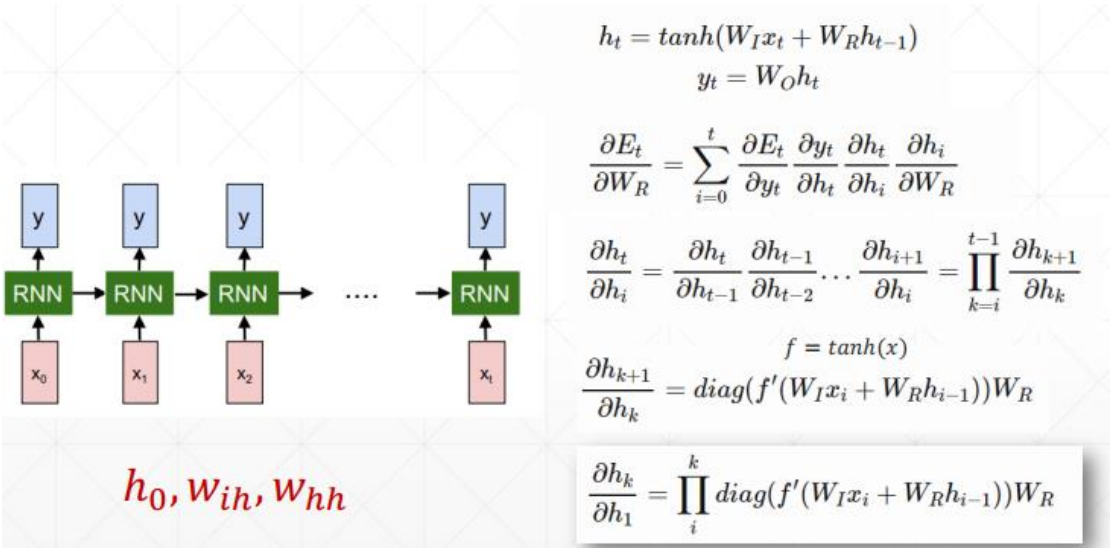
循环神经网络


Neg:0
$P(y\ is\ pos|x)$
h_0
h_1
h_2
h_3
h_4
h_5
$x@w_{xh} + h_0@w_{hh}$
$x@w_{xh} + h_1@w_{hh}$
$x@w_{xh} + h_2@w_{hh}$
$x@w_{xh} + h_2@w_{hh}$
$x@w_{xh} + h_3@w_{hh}$
I
hate
this
boring
movie


不断循环这个过程
[0,0,0 ...]
$x_t@w_{xh} + h_t@w_{hh}$
embedding

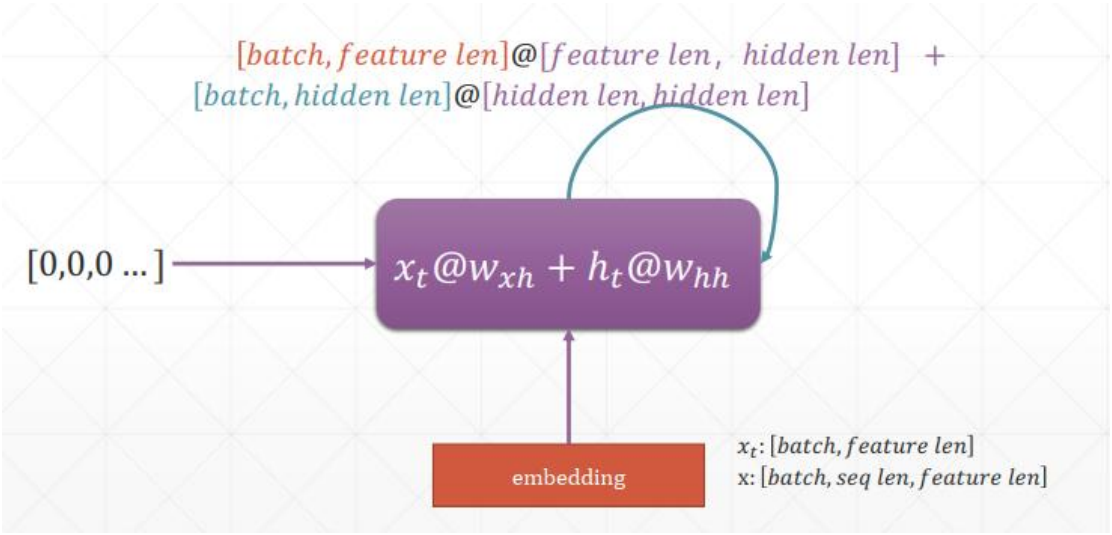

y
RNN
x
$h_t = f_W(h_{t-1}, x_t)$
$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
$y_t = W_{hy}h_t$

梯度的求解



$$h_t = tanh(W_I x_t + W_R h_{t-1})$$
$$y_t = W_O h_t$$
$$\frac{\partial E_t}{\partial W_R} = \sum_{i=0}^{t} \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_i}\frac{\partial h_i}{\partial W_R}$$
$$\frac{\partial h_t}{\partial h_i} = \frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\cdots\frac{\partial h_{i+1}}{\partial h_i} = \prod_{k=i}^{t-1}\frac{\partial h_{k+1}}{\partial h_k}$$
$$f = tanh(x)$$
$$\frac{\partial h_{k+1}}{\partial h_k} = diag(f'(W_I x_i + W_R h_{i-1}))W_R$$
$$\frac{\partial h_k}{\partial h_1} = \prod_{i}^{k} diag(f'(W_I x_i + W_R h_{i-1}))W_R$$

一个 cell



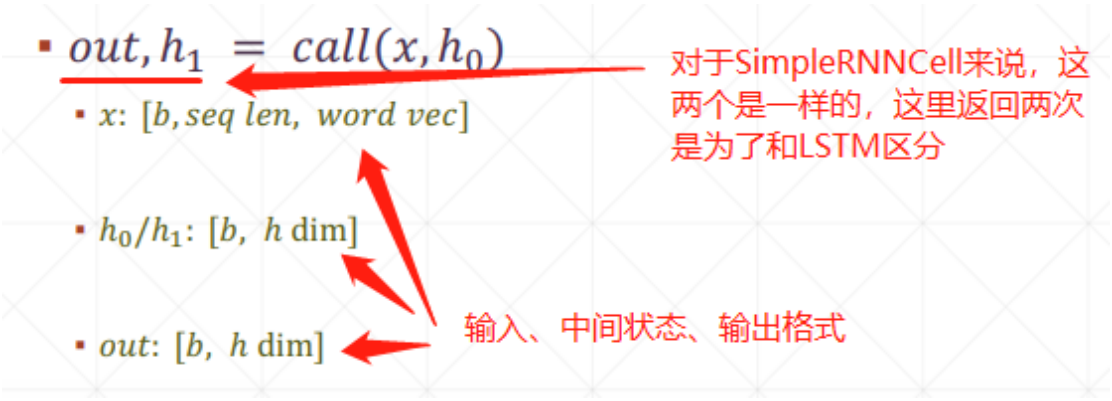
简单的实现

```
In [17]: cell=layers.SimpleRNNCell(3)
In [18]: cell.build(input_shape=(None,4))

In [19]: cell.trainable_variables
[<tf.Variable 'kernel:0' shape=(4, 3) dtype=float32, numpy=
 array([[ 0.23682523,  0.24167228,  0.19834113],
        [ 0.58464265, -0.44347632, -0.23693317],
        [ 0.5130104 , -0.86219984,  0.16108215],
        [-0.37421566, -0.6311711 ,  0.10514995]], dtype=float32)>,
 <tf.Variable 'recurrent_kernel:0' shape=(3, 3) dtype=float32, numpy=
 array([[ 0.7126031 ,  0.39248434,  0.5815092 ],
        [-0.34029973,  0.9182056 , -0.2027184 ],
        [ 0.61350864,  0.05342963, -0.7878784 ]], dtype=float32)>,
 <tf.Variable 'bias:0' shape=(3,) dtype=float32, numpy=array([0., 0., 0.], dtype=float32)>]
```

Wxh

Whh

SimpleRNNCell

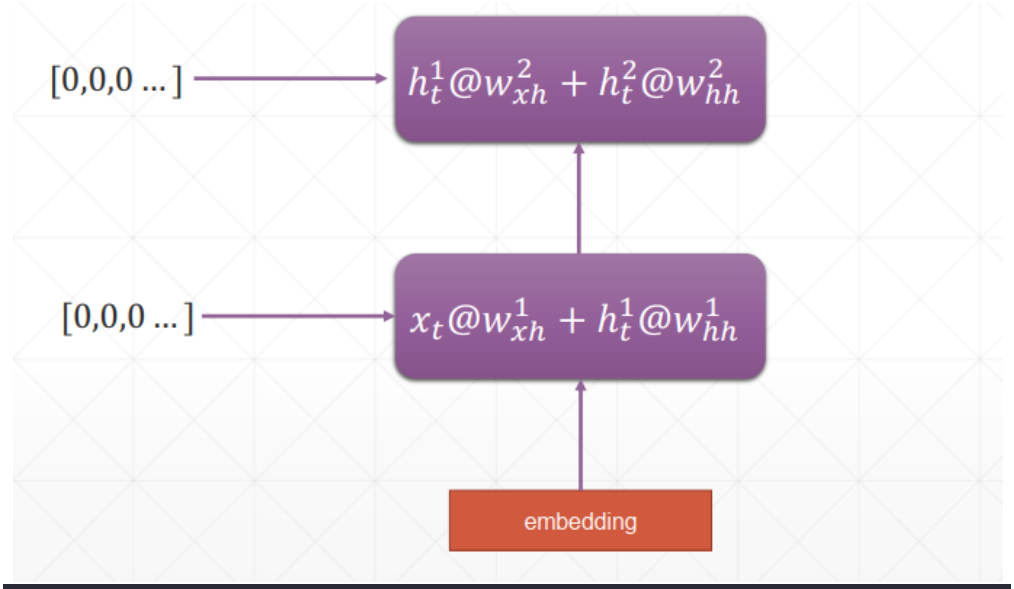



```
4个句子，每个80个单词，每个单词用100维向量表示
In [4]: x=tf.random.normal([4,80,100])
In [5]: xt0=x[:,0,:]

In [6]: cell=tf.keras.layers.SimpleRNNCell(64)

In [9]: out, xt1=cell(xt0, [tf.zeros([4,64])])

In [12]: out.shape, xt1[0].shape
Out[12]: (TensorShape([4, 64]), TensorShape([4, 64]))

In [13]: id(out), id(xt1[0])
Out[13]: (1724126403608, 1724126403608)
```

W 和 b

```
In [6]:  cell=tf.keras.layers.SimpleRNNCell(64)

In [17]: cell.trainable_variables
[<tf.Variable 'simple_rnn_cell/kernel:0' shape=(100, 64) dtype=float32, numpy=
 array([[-0.14681616, -0.18293515,  0.01196897, ..., -0.15498586,
         -0.09611963,  0.1502908 ],
        [ 0.05362315, -0.05283108,  0.1457149 , ..., -0.11584248,
        [-0.10094493, -0.10836099,  0.18547966, ...,  0.15367018,
         -0.05795772, -0.11807813]], dtype=float32)>,

 <tf.Variable 'simple_rnn_cell/recurrent_kernel:0' shape=(64, 64) dtype=float32, numpy=
 array([[ 1.01237297e-01,  8.35651010e-02, -1.47625804e-04, ...,
         -8.24909434e-02, -1.28735945e-01,  4.13244553e-02],
        [-1.57783236e-02,  6.91108108e-02,  5.74547313e-02, ...,
         -1.16356298e-01, -2.58456618e-02, -1.97714210e-01]], dtype=float32)>,

 <tf.Variable 'simple_rnn_cell/bias:0' shape=(64,) dtype=float32, numpy=
 array([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., 0., 0., 0., 0., 0.], dtype=float32)>]
```

多层 RNN




```
In [4]: x=tf.random.normal([4,80,100])
In [5]: xt0=x[:,0,:]

In [6]:  cell=tf.keras.layers.SimpleRNNCell(64)
In [14]: cell2=tf.keras.layers.SimpleRNNCell(64)
In [6]: state0 = [tf.zeros([4,64])]
In [6]: state1 = [tf.zeros([4,64])]

In [9]: out0, state0=cell(xt0, state0)
In [15]: out2, state2=cell2(out, state0)

In [16]: out2.shape, state2[0].shape
Out[16]: (TensorShape([4, 64]), TensorShape([4, 64]))
```

```
        state0 = [tf.zeros([batchsz, units])]
        state1 = [tf.zeros([batchsz, units])]
        for word in tf.unstack(x, axis=1): # word: [b, 100]
            # h1 = x*wxh+h0*whh
            # out0: [b, 64]
            out0, state0 = self.rnn_cell0(word, state0, training)
            # out1: [b, 64]
            out1, state1 = self.rnn_cell1(out0, state1, training)
```

等价

```
self.rnn = keras.Sequential([
    layers.SimpleRNN(units, dropout=0.5, return_sequences=True, unroll=True),
    layers.SimpleRNN(units, dropout=0.5, unroll=True)
])

  # x: [b, 80, 100] => [b, 64]
x = self.rnn(x)
```

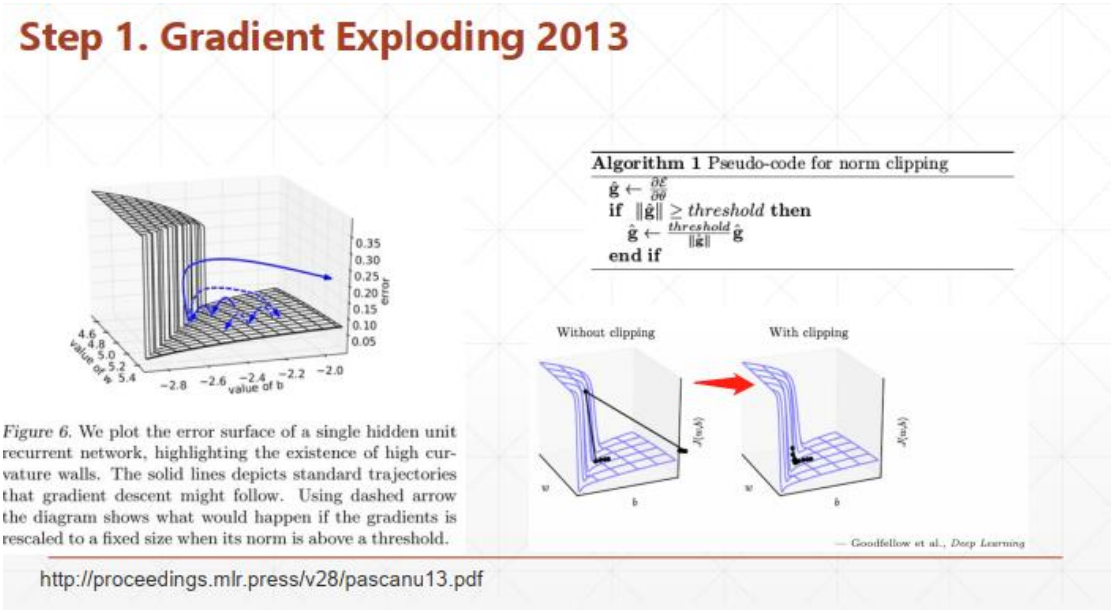
RNN 实战

- ✓ SimpleRNNCell
  - ✓ single layer
  - ✓ multi-layers
- ✓ RNNCell

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers

tf.random.set_seed(22)
np.random.seed(22)
assert tf.__version__.startswith('2.')

batchsz = 128

# 将最常用的 10000 个单词编码，剩下的单词统一用一个编码
total_words = 10000
# 句子最长的长度
max_review_len = 80
embedding_len = 100
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_da
ta(num_words=total_words)
# x_train ==> [b,80] , x_test ==> [b,80]
x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxl
en=max_review_len)
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen
=max_review_len)

db_train = tf.data.Dataset.from_tensor_slices((x_train, y_train))
db_train = db_train.shuffle(1000).batch(batchsz, drop_remainder=True)
db_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
db_test = db_test.batch(batchsz, drop_remainder=True)
print('x_train shape:', x_train.shape, tf.reduce_max(y_train), tf.red
uce_min(y_train))
print('x_test shape:', x_test.shape)

class MyRNN(tf.keras.Model):

    def __init__(self, units):
```

```
        super(MyRNN, self).__init__()

        self.state0 = [tf.zeros([batchsz, units])]
        self.state1 = [tf.zeros([batchsz, units])]

        # transform text to embedding representation, [b,80] ==> [b,8
0,100]
        self.embedding = layers.Embedding(total_words, embedding_len,
                                          input_length=max_review_len
)

        self.rnn_cell0 = layers.SimpleRNNCell(units, dropout=0.2)
        self.rnn_cell1 = layers.SimpleRNNCell(units, dropout=0.2)

        self.outlayer = layers.Dense(1)

    def call(self, inputs, training=None):
        # [b, 80]
        x = inputs
        # embedding: [b,80] ==> [b,80,100]
        x = self.embedding(x)
        # rnn cell compute, [b,80,100] ==> [b,64]
        state0 = self.state0
        state1 = self.state1
        for word in tf.unstack(x, axis=1):
            out0, state0 = self.rnn_cell0(word, state0, training)
            out1, state1 = self.rnn_cell1(out0, state1)

        # out: [b,64] ==> [b,1]
        x = self.outlayer(out1)
        prob = tf.sigmoid(x)
        return prob
def main():
    units = 64
    epochs = 4
    model = MyRNN(units=units)
    model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                  loss=tf.losses.BinaryCrossentropy(),
                  metrics=['accuracy'])
    model.fit(db_train, epochs=epochs, validation_data=db_test)

    model.evaluate(db_test)
main()
```

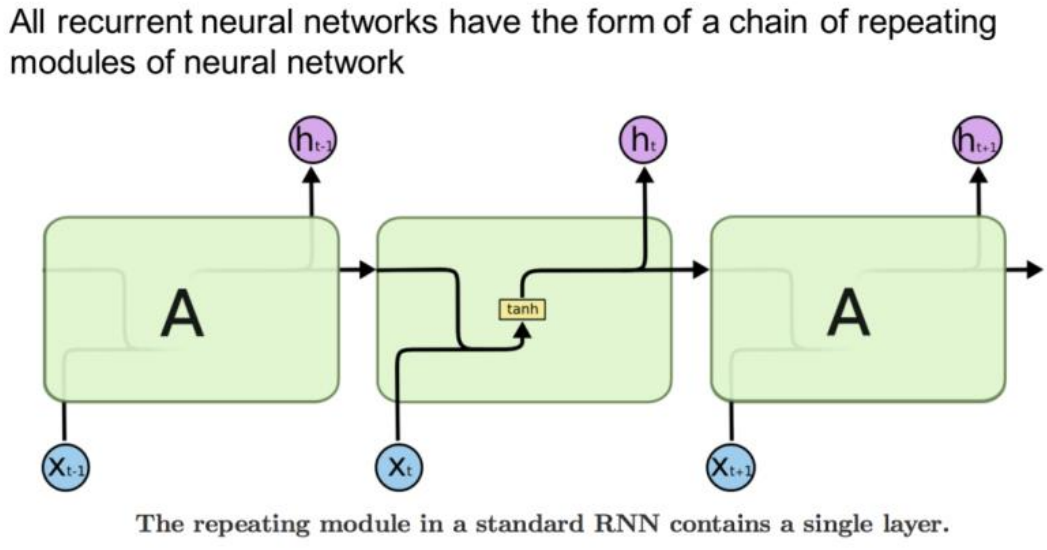
梯度弥散和梯度爆炸



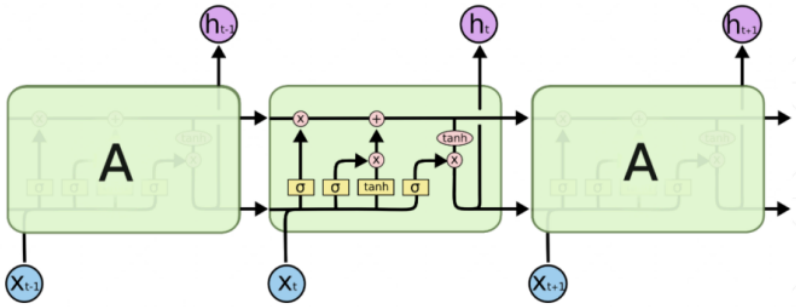
进行梯度裁剪

```
with tf.GradientTape() as tape:
    logits = model(x)
    loss = criteon(y, logits)

grads = tape.gradient(loss, model.trainable_variables)
# MUST clip gradient here or it will disconverge!
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

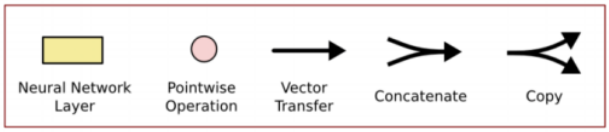
12 LSTM (Long short-tem memory)



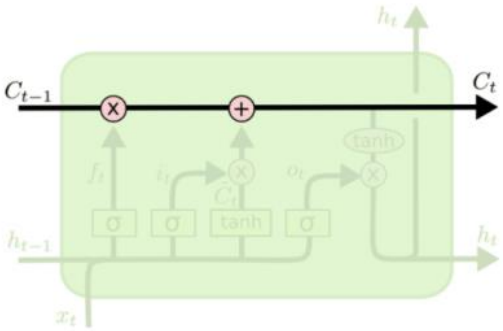
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer there are four, interacting in a very special way.



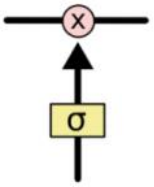
The repeating module in an LSTM contains four interacting layers.



The Core Idea Behind LSTMs : Cell State

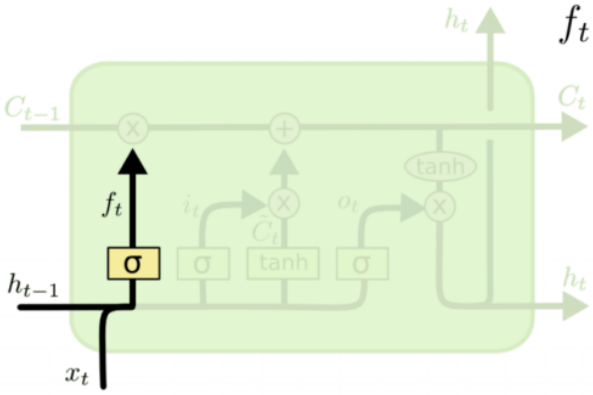


Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



An LSTM has three of these gates, to protect and control the cell state.

LSTM : Forget gate



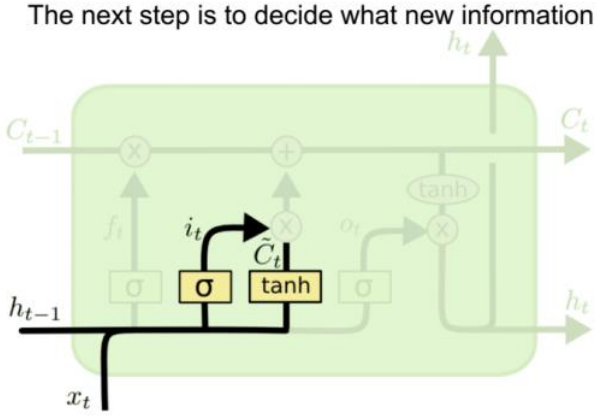
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

It looks at h_{t-1} and x_t and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.

A 1 represents “completely keep this” while a 0 represents “completely get rid of this”.

LSTM : Input gate and Cell State

The next step is to decide what new information we're going to store in the cell state.



a sigmoid layer called the "**input gate layer**" decides which values we'll update.

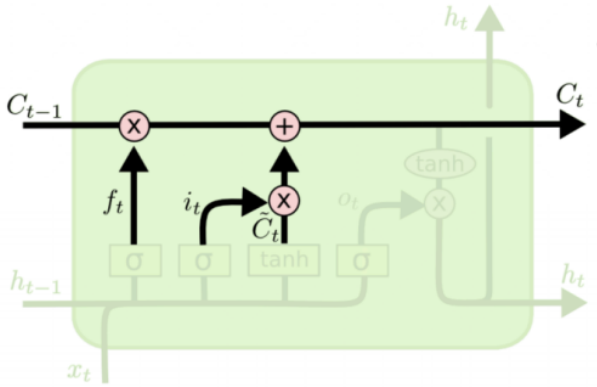
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

a tanh layer creates a vector of new candidate values, that could be added to the state.

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

LSTM : Input gate and Cell State

It's now time to update the old cell state into the new cell state



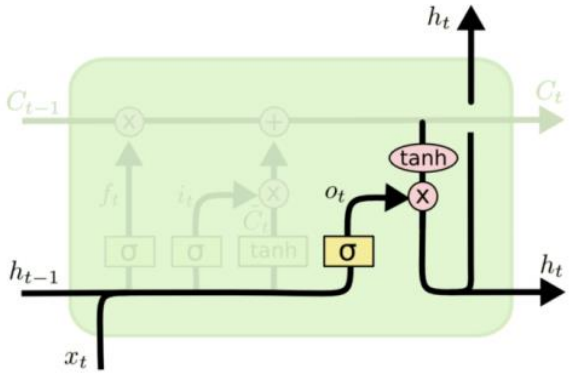
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

We multiply the old state by ft forgetting the things we decided to forget earlier.

Then, we add the new candidate values, scaled by how much we decided to update each state value.

LSTM : Output

Finally, we need to decide what we're going to output.

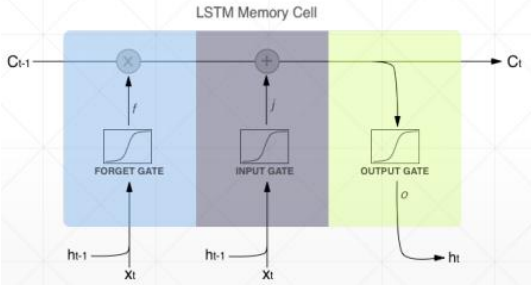


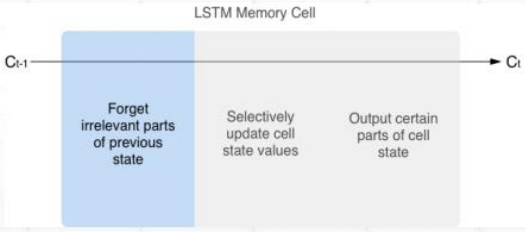
First, we run a sigmoid layer which decides what parts of the cell state we're going to output.

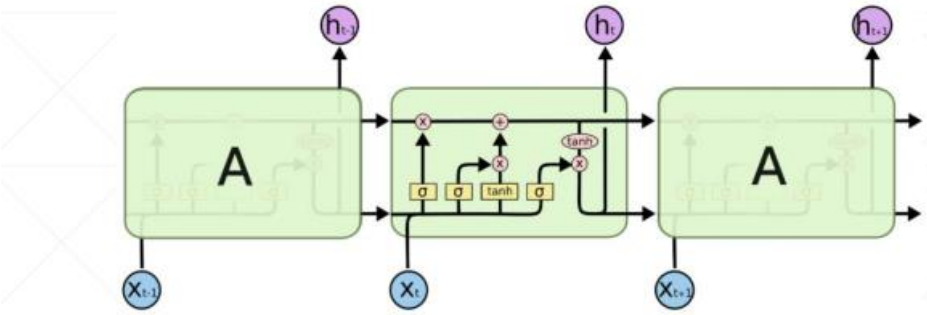
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

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



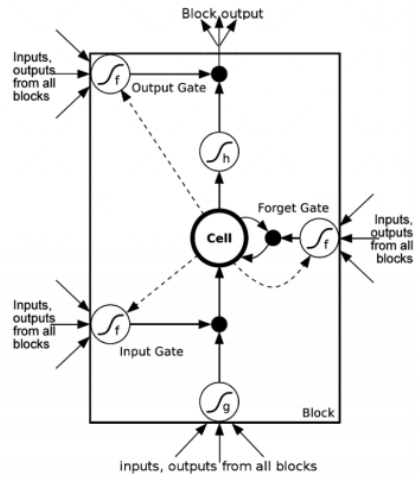




$$\begin{pmatrix} \mathbf{i}^{(t)} \\ \mathbf{f}^{(t)} \\ \mathbf{o}^{(t)} \\ \tilde{\mathcal{C}}^{(t)} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{pmatrix} \tag{6}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \tilde{\mathcal{C}} \tag{7}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh(\mathbf{c}^{(t)}). \tag{8}$$



input gate	forget gate	behavior
0	1	remember the previous value
1	1	add to the previous value
0	0	erase the value
1	0	overwrite the value