Chapter 11 Recurrent Neural Network

The powerful rise of artificial intelligence may be the best thing in human history, or it may be the worst thing.
−Steven Hawking

Convolutional neural network uses the local correlation of data and the idea of weight sharing to greatly reduce the amount of network parameters. It is very suitable for pictures with spatial and local correlation. It has been successfully applied to a series of tasks in the field of computer vision. In addition to the spatial dimension, natural signals also have a temporal dimension. Signals with a time dimension are very common, such as the text we are reading, the speech signal emitted when we speak, and the stock market that changes over time. This type of data does not necessarily have local relevance, and the length of the data in the time dimension is also variable. Convolutional neural networks are not good at processing such data.

So analyzing and recognizing this type of signals is a task that must be solved in order to push artificial intelligence to general artificial intelligence. The recurrent neural network that will be introduced in this chapter can better solve such problems. Before introducing the recurrent neural network, let's first introduce the method of representing data in chronological order.

11.1 Sequence representation method

Data with order is generally called a sequence, for example, commodity price data that changes over time is a very typical sequence. Considering the price change trend of a commodity A between January and June, we can record it as a one-dimensional vector: $[x_1, x_2, x_3, x_4, x_5, x_6]$, and its shape is [6]. If you want to represent the price change trend of b goods from January to June, you can record it as a 2-dimensional tensor:

$$\left[\left[x_1^{(1)}, x_2^{(1)}, \cdots, x_6^{(1)}\right], \left[x_1^{(2)}, x_2^{(2)}, \cdots, x_6^{(2)}\right], \cdots, \left[x_1^{(b)}, x_2^{(b)}, \cdots, x_6^{(b)}\right]\right]$$

Where b represents the number of commodities, and the tensor shape is $[b, 6]$.

In this way, the sequence signal is not difficult to represent, only a tensor with shape [b, s] is needed, where b is the number of sequences and s is the length of the sequence. However, many signals cannot be directly represented by a scalar value. For example, to represent feature vectors of length n generated by each timestamp, and a tensor of shape [b, s, n] is required. Consider more complex text data: sentences. The word generated on each timestamp is a character, not a numerical value, and therefore cannot be directly represented by a scalar. We already know that neural networks are essentially a series of math operations such as matrix multiplication and addition. They cannot directly process string data. If you want neural networks to be used for natural language processing tasks, then how to convert words or characters into numerical values becomes particularly critical. Next, we mainly discuss the representation method of text sequence. For other non-numerical signals, please refer to the representation method of text sequence.

For a sentence containing n words, a simple way to represent the words is the One-hot encoding method we introduced earlier. Take English sentences as an example, suppose we only consider the most commonly used 10,000 words, then each word can be expressed as a sparse One-hot vector with one position as 1, and other positions of 0 and a length of 10,000. As shown in Figure 11.1, if only n location names are considered, each location name can be coded as a One-hot vector of length n.

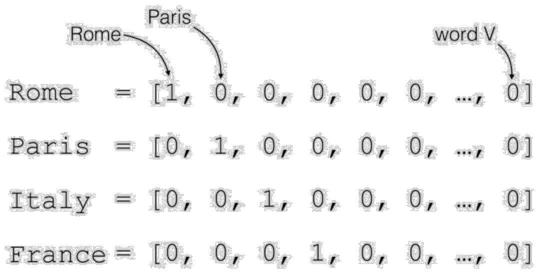

Paris
Rome
word V
Rome = [1, 0, 0, 0, 0, 0, ..., 0]
Paris = [0, 1, 0, 0, 0, 0, ..., 0]
Italy = [0, 0, 1, 0, 0, 0, ..., 0]
France = [0, 0, 0, 1, 0, 0, ..., 0]

Figure 0.1 One-hot encoding of location names

We call the process of encoding text into numbers as Word Embedding. One-hot encoding is simple and intuitive to implement Word Embedding, and the encoding process does not require learning and training. However, the one-hot encoding vector is high-dimensional and extremely sparse, with a large number of positions as 0s. Therefore, it is computationally expensive and also not conducive to the neural network training. From a semantic point of view, One-hot encoding has a serious problem. It ignores the semantic relevance inherent in words. For example, for the words "like", "dislike", "Rome", and "Paris", "like" and "dislike" are strongly related from a semantic point of view. They both indicate the degree of like. "Rome" and " "Paris" is also strongly related. They both indicate two locations in Europe. For a group of such words, if One-hot encoding is used, there is no correlation between the obtained vectors, and the semantic relevance of the original text cannot be well reflected. Therefore, the One-hot encoding has obvious disadvantages.

In the field of natural language processing, there is a special research area about Word Vector so that the semantic level of relevance can be well reflected through the Word Vector. One way to measure the correlation between word vectors is the cosine similarity:

$$\text{similarity}(\boldsymbol{a}, \boldsymbol{b}) \triangleq \cos(\theta) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}| \cdot |\boldsymbol{b}|}$$

Where $\boldsymbol{a}$ and $\boldsymbol{b}$ represent two word vectors. Figure 11.2 shows the similarity between the words "France" and "Italy", and the similarity between the words "ball" and "crocodile", and θ is the angle between the two word vectors. It can be seen that $\cos(\theta)$ better reflects semantic relevance.

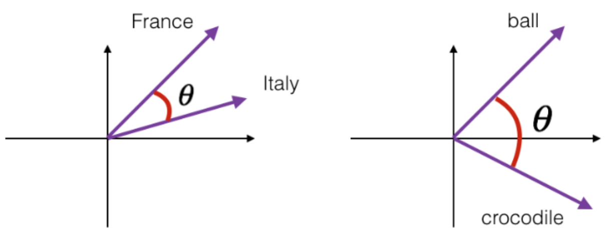


Figure 0.2 Cosine similarity diagram

11.1.1 Embedding layer

In a neural network, the representation vector of a word can be obtained directly through training. We call the representation layer of the word Embedding layer. The Embedding layer is responsible for encoding the word into a word vector $\boldsymbol{v}$. It accepts the word number i using digital encoding, such as 2 for "I", 3 for "me", etc. The total number of words in the system is recorded as N_{vocab}, and the output is vector $\boldsymbol{v}$ with length n:

$$\boldsymbol{v} = f_\theta(i|N_{\text{vocab}}, n)$$

The Embedding layer is very simple to implement. Build a lookup table with shape $[N_{\text{vocab}}, n]$. For any word number i, you only need to query the vector at the corresponding position and return:

$$\boldsymbol{v} = \text{table}[i]$$

The Embedding layer is trainable. It can be placed in front of the neural network to complete the conversion of words to vectors. The resulting representation vector can continue to pass through the neural network to complete subsequent tasks, and calculate the error $\mathcal{L}$. The gradient descent algorithm is used to achieve end-to-end training.

In TensorFlow, a Word Embedding layer can be defined by layers.Embedding(N_{vocab}, n), where the N_{vocab} parameter specifies the number of words, and n specifies the length of the word vector. E.g:

```
x = tf.range(10) # Generate a digital code of 10 words
x = tf.random.shuffle(x) # Shuffle
# Create a layer with a total of 10 words, each word is represented by a
vector of length 4
net = layers.Embedding(10, 4)
out = net(x) # Get word vector
```

The above code creates an Embedding layer of 10 words. Each word is represented by a vector of length 4. You can pass in an input with a number code of 0-9 to get the word vectors of these 4 words. These word vectors are initialized randomly and has not been trained, for example:

```
<tf.Tensor: id=96, shape=(10, 4), dtype=float32, numpy=
array([[-0.00998075, -0.04006485,  0.03493755,  0.03328368],
      [-0.04139598, -0.02630153, -0.01353856,  0.02804044],…
```

We can directly view the query table inside the Embedding layer:

```
In [1]: net.embeddings
Out[1]:
<tf.Variable 'embedding_4/embeddings:0' shape=(10, 4) dtype=float32, numpy=
array([[ 0.04112223,  0.01824595, -0.01841902,  0.00482471],
      [-0.00428962, -0.03172196, -0.04929272,  0.04603403],…
```

The optimizable property of the net.embeddings tensor is True, which means it can be optimized by the gradient descent algorithm.

```
In [2]: net.embeddings.trainable
Out[2]:True
```

11.1.2 Pre-trained word vectors

The lookup table of the Embedding layer is initialized randomly and needs to be trained from scratch. In fact, we can use pre-trained Word Embedding models to get the word representation. The word vector based on pre-trained models is equivalent to transferring the knowledge of the entire semantic space, which can often get better performance.

Currently, the widely used pre-trained models include Word2Vec and GloVe. They have been trained on a massive corpus to obtain a better word vector representation, and can directly export the learned word vector table to facilitate migration to other tasks. For example, the GloVe model GloVe.6B.50d has a vocabulary of 400,000, and each word is represented by a vector of length 50. Users only need to download the corresponding model file in order to use it. The "glove6b50dtxt.zip" model file is about 69MB.

So how to use these pre-trained word vector models to help improve the performance of NLP tasks? Very simple. For the Embedding layer, random initialization is no longer used. Instead, we use the pre-trained model parameters to initialize the query table of the Embedding layer. E.g:

```
# Load the word vector table from the pre-trained model
embed_glove = load_embed('glove.6B.50d.txt')
# Initialize the Embedding layer directly using the pre-trained word vector
table
net.set_weights([embed_glove])
```

The Embedding layer initialized by the pre-trained word vector model can be set to not participate in training: net.trainable = False, then the pre-trained word vector is directly applied to this specific task. If you also want to learn different representations from the pre-trained word vector model, then the Embedding layer can be included in the backpropagation algorithm by setting net.trainable = True, and gradient descent then can be used to fine-tune the word representation.

11.2 Recurrent Neural Network

Now let's consider how to deal with sequence signals. Taking a text sequence as an example, consider a sentence:

"I hate this boring movie"

Through the Embedding layer, it can be converted into a tensor with shape $[b, s, n]$, where b is the number of sentences, s is the sentence length, and n is the length of the word vector. The above sentence can be expressed as a tensor with shape [1,5,10], where 5 represents the length of the sentence word, and 10 represents the length of the word vector.

Next, we will gradually explore a network model that can process sequence signals. We take the sentiment classification task as an example, as shown in Figure 11.3. The sentiment classification task extracts the overall semantic features expressed by the text data, and thereby predict the sentiment type of the input text: positive or negative. From the perspective of classification, sentiment classification is a simple two-classification problem. Unlike image classification, because the input is a text sequence, traditional convolutional neural networks cannot achieve good results. So what type of network is good at processing sequence data?

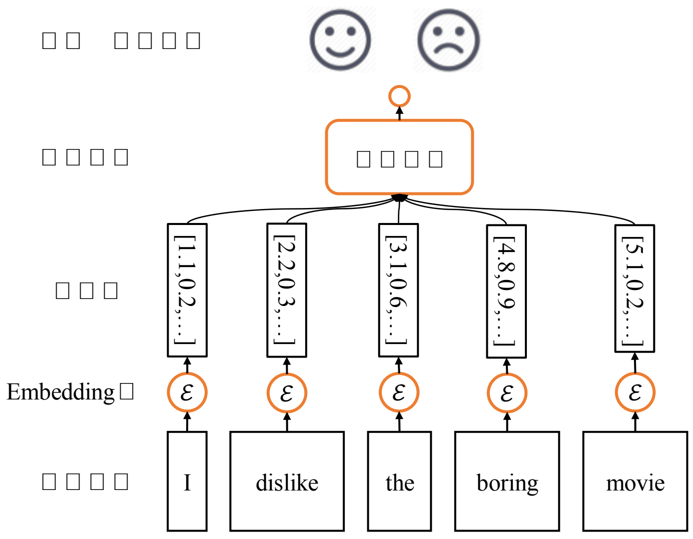


Figure 0.3 Sentiment classification task

11.2.1 Is a fully connected layer feasible?

The first thing we think of is that for each word vector, a fully connected layer network can be used.

$$\boldsymbol{o} = \sigma(\boldsymbol{W}_t \boldsymbol{x}_t + \boldsymbol{b}_t)$$

Extract semantic features, as shown in Figure 11.4. The word vector of each word is extracted through s fully connected layer classification networks 1. The features of all words are finally merged, and the category probability distribution of the sequence is output through the classification network 2. For a sentence of length s, at least s fully-connected network layers are required.

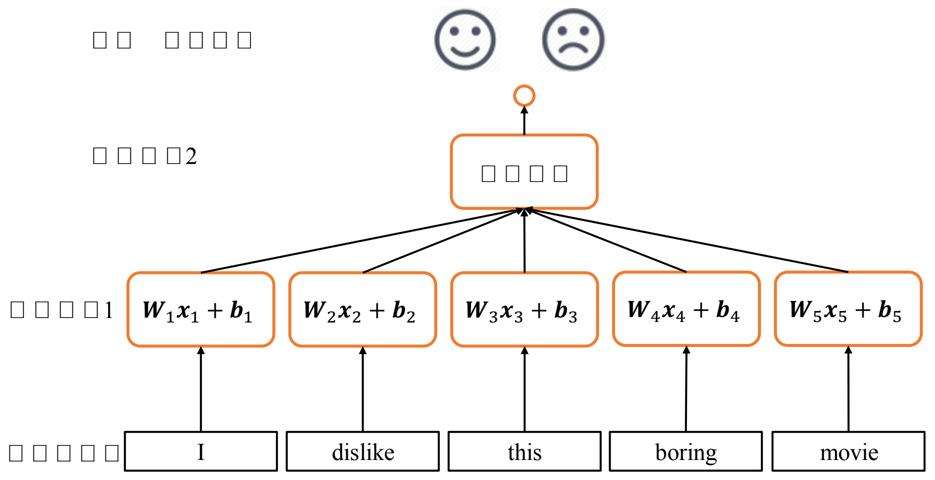


Figure 0.4 Network architecture one

The disadvantages of this scheme are:

- ❑ The amount of network parameters is considerable, and the memory usage and calculation cost are high. At the same time, since the length s of each sequence is not the same, the network structure changes dynamically.
- ❑ Each fully connected layer sub-network $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ can only sense the input of the current word vector, and cannot perceive the context information before and after, resulting in the lack of overall sentence semantics. Each sub-network can only extract high-level features based on its own input.

We will solve these two disadvantages one by one.

11.2.2 Shared weight

When introducing convolutional neural networks, we have learned the reason why convolutional neural networks is better than fully connected networks in processing locally related data is because it makes full use of the idea of weight sharing and greatly reduces the amount of network parameters , which makes the network training more efficient. So, can we learn from the idea of weight sharing when dealing with sequence signals?

In the scheme in Figure 11.4, the network of s fully connected layers does not realize weight sharing. We try to share these s network layer parameters, which is actually equivalent to using a fully connected network to extract the feature information of all words, as shown in Figure 11.5.

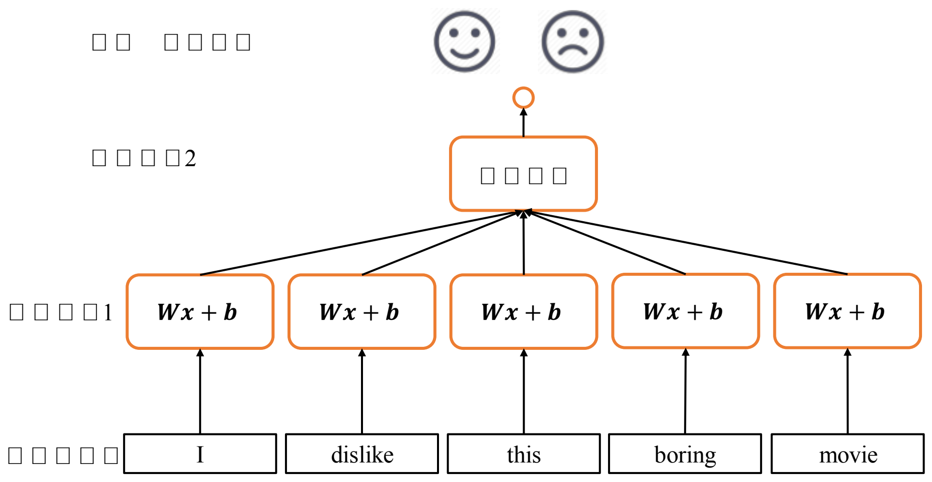


Figure 0.5 Network architecture two

After weight sharing, the amount of parameters is greatly reduced, and network training becomes more stable and efficient. However, this network structure does not consider the order of sequences, and the same output can still be obtained by shuffling the order of the word vectors. Therefore, it cannot obtain effective global semantic information.

11.2.3 Global semantics

How to give the network the ability to extract overall semantic features? In other words, how can the network extract the semantic information of word vectors in order and accumulate it into the global semantic information of the entire sentence? We thought of the memory mechanism. If the network can provide a separate memory variable, each time the feature of the word vector is extracted and the memory variable is refreshed, until the last input is completed, the memory variable at this time stores the semantic features of all sequences, and because of the order of input sequences, the contents of memory variables are closely related to the sequence order.

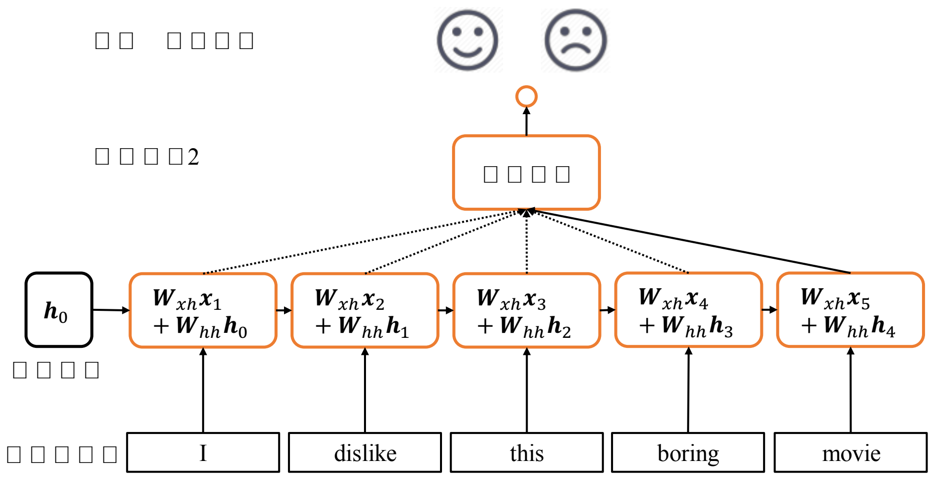


Figure 0.6 Recurrent neural network (no bias added)

We implement the above memory mechanism as a state tensor $\boldsymbol{h}$, as shown in Figure 11.6. In addition to the original $\boldsymbol{W}_{xh}$ parameter sharing, an additional $\boldsymbol{W}_{hh}$ parameter is added here. The state tensor $\boldsymbol{h}$ refresh mechanism for each timestamp t is:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

Where the state tensor $\boldsymbol{h}_0$ is the initial memory state, which can be initialized to all 0s. After the input of s word vectors, the final state tensor $\boldsymbol{h}_s$ of the network is obtained. $\boldsymbol{h}_s$ better represents the global semantic information of the sentence. Passing $\boldsymbol{h}_s$ through a fully connected layer classifier can complete the sentiment classification task.

11.2.4 Recurrent Neural Network

Through step-by-step exploration, we finally proposed a "new" network structure, as shown in Figure 11.7. At each time stamp t, the network layer accepts the input $\boldsymbol{x}_t$ of the current time stamp and the network state vector of the previous time stamp $\boldsymbol{h}_{t-1}$, after

$$\boldsymbol{h}_t = f_\theta(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$$

After transformation, the new state vector $\boldsymbol{h}_t$ of the current time stamp is obtained and written into the memory state, where f_θ represents the operation logic of the network, and θ is the network parameter set. At each time stamp, the network layer has an output to produce $\boldsymbol{o}_t$, $\boldsymbol{o}_t = g_\phi(\boldsymbol{h}_t)$, which is to output the state vector of the network after transformation.

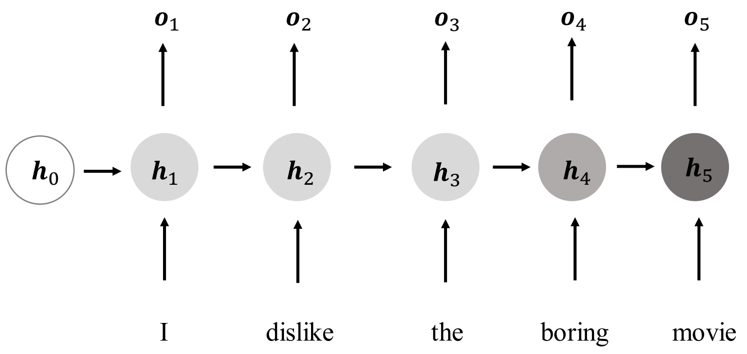


Figure 0.7 Expanded RNN model

The above network structure is folded on the time stamp, as shown in Figure 11.8. The network cyclically accepts each feature vector $\boldsymbol{x}_\mathrm{t}$ of the sequence, refreshes the internal state vector $\boldsymbol{h}_t$, and forms the output $\boldsymbol{o}_\mathrm{t}$ at the same time. For this kind of network structure, we call it the Recurrent Neural Network (RNN).

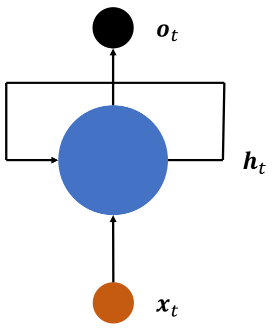


Figure 0.8 Folded RNN model

More specifically, if we use the tensors $\boldsymbol{W}_{xh}$, $\boldsymbol{W}_{hh}$ and bias $\boldsymbol{b}$ to parameterize the f_θ network, and use the following ways to update the memory state, we call this kind of network a basic recurrent neural network, unless otherwise specified, generally speaking, the recurrent neural network refers to this realization.

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

In the recurrent neural network, the activation function uses the tanh function more, and we can choose not to use the bias $\boldsymbol{b}$ to further reduce the amount of parameters. The state vector $\boldsymbol{h}_t$ can be directly used as output, that is, $\boldsymbol{o}_t = \boldsymbol{h}_t$, or a simple linear transformation of $\boldsymbol{h}_t$ can be done to $\boldsymbol{o}_t = W_{ho}\boldsymbol{h}_t$ to get the network output $\boldsymbol{o}_t$ on each time stamp.

11.3 Gradient propagation

Through the update expression of the recurrent neural network, it can be seen that the output is derivable to the tensors $\boldsymbol{W}_{xh}$, $\boldsymbol{W}_{hh}$ and bias $\boldsymbol{b}$, and the automatic gradient descent algorithm can be used to solve the gradient of the network. Here we simply derive the gradient propagation formula of RNN and explore its characteristics.

Consider the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$, where $\mathcal{L}$ is the error of the network, and only consider the difference between the last output $\boldsymbol{o}_t$ at t and the true value. Since $\boldsymbol{W}_{hh}$ is shared by the weight of each timestamp i, when calculating $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$, it is necessary to sum the gradients on each intermediate timestamp i, using the chain rule to expand as

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}} = \sum_{i=1}^{t} \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t}\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}\frac{\partial^+ \boldsymbol{h}_i}{\partial \boldsymbol{W}_{hh}}$$

where $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t}$ can be obtained directly based on the loss function, in the case of $\boldsymbol{o}_t = \boldsymbol{h}_t$:

$$\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t} = I$$

而$\frac{\partial^+ \boldsymbol{h}_i}{\partial \boldsymbol{W}_{hh}}$的梯度将$\boldsymbol{h}_i$展开后也可以求得：

And the gradient of $\frac{\partial^+ \boldsymbol{h}_i}{\partial W_{hh}}$ can also be obtained after expanding $\boldsymbol{h}_i$:

$$\frac{\partial^+ \boldsymbol{h}_i}{\partial \boldsymbol{W}_{hh}} = \frac{\partial \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b})}{\partial \boldsymbol{W}_{hh}}$$

Among them $\frac{\partial^+ \boldsymbol{h}_i}{\partial \boldsymbol{W}_{hh}}$ only considers the gradient propagation of one time stamp, that is, the "direct" partial derivative, which is different from $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$ that considers the gradient propagation of all timestamps $i = 1, \cdots, t$.

Therefore, we only need to derive the expression of $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ to complete the gradient derivation of the recurrent neural network. Using the chain rule, we divide $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ into the gradient expression of successive timestamps:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{i+1}}{\partial \boldsymbol{h}_i} = \prod_{k=i}^{t-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}$$

Consider

$$\boldsymbol{h}_{k+1} = \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_{k+1} + \boldsymbol{W}_{hh}\boldsymbol{h}_k + \boldsymbol{b})$$

then

$$\begin{aligned}\frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} &= \boldsymbol{W}_{hh}^T diag\big(\sigma'(\boldsymbol{W}_{xh}\boldsymbol{x}_{k+1} + \boldsymbol{W}_{hh}\boldsymbol{h}_k + \boldsymbol{b})\big) \\ &= \boldsymbol{W}_{hh}^T diag(\sigma'(\boldsymbol{h}_{k+1}))\end{aligned}$$

where $diag(\boldsymbol{x})$ takes each element of the vector $\boldsymbol{x}$ as the diagonal element of the matrix, and obtains a diagonal matrix with all other elements being 0, for example:

$$diag([3,2,1]) = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Therefore,

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i} = \prod_{j=i}^{t-1} diag\left(\sigma'(\boldsymbol{W}_{xh}\boldsymbol{x}_{j+1} + \boldsymbol{W}_{hh}\boldsymbol{h}_j + \boldsymbol{b})\right)\boldsymbol{W}_{hh}$$

So far, the gradient derivation of $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$ is completed.

Since deep learning frameworks can help us automatically derive gradients, we only need to understand the gradient propagation mechanism of the recurrent neural network. In the process of deriving $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$, we found that the gradient of $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ includes the continuous multiplication operation of $\boldsymbol{W}_{hh}$, which is the root cause of the difficulty in training the recurrent neural network the reason. We will discuss it later,

11.4 How to use RNN layers

After introducing the principle of the recurrent neural network, let's learn how to implement the RNN layer in TensorFlow. In TensorFlow, the $\sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b})$ calculation can be completed by layers.SimpleRNNCell() function. It should be noted that in TensorFlow, RNN stands for recurrent neural network in a general sense. For the basic recurrent neural network we are currently introducing, it is generally called SimpleRNN. The difference between SimpleRNN and SimpleRNNCell is that the layer with Cell only completes the forward operation of one timestamp, while the layer without Cell is generally implemented based on the Cell layer, which has already completed multiple timestamp cycles internally. Therefore, it is more convenient and faster to use.

We first introduce the use of SimpleRNNCell, and then introduce the use of SimpleRNN layer.

11.4.1 SimpleRNNCell

Take a certain input feature length n=4 and Cell state vector feature length h=3 as an example. First, we create a SimpleRNNCell without specifying the sequence length s. The code is as follows:

```
In [3]:
cell = layers.SimpleRNNCell(3) # Create RNN Cell, memory vector length is 3
cell.build(input_shape=(None,4)) # Output feature length n=4
cell.trainable_variables # Print wxh, whh, b tensor
Out[3]:
[<tf.Variable 'kernel:0' shape=(4, 3) dtype=float32, numpy=…>,
 <tf.Variable 'recurrent_kernel:0' shape=(3, 3) dtype=float32, numpy=…>,
 <tf.Variable 'bias:0' shape=(3,) dtype=float32, numpy=array([0., 0., 0.],
dtype=float32)>]
```

It can be seen that SimpleRNNCell maintains 3 tensors internally, the kernel variable is the tensor $\boldsymbol{W}_{xh}$, the recurrent_kernel variable is the tensor $\boldsymbol{W}_{hh}$, and the bias variable is the bias vector $\boldsymbol{b}$. However, the memory vector $\boldsymbol{h}$ of RNN is not maintained by SimpleRNNCell, and the user needs to initialize the vector $\boldsymbol{h}_0$ and record the $\boldsymbol{h}_t$ on each time stamp.

The forward operation can be completed by calling the Cell instance:

$$\boldsymbol{o}_t, [\boldsymbol{h}_t] = \text{Cell}(\boldsymbol{x}_t, [\boldsymbol{h}_{t-1}])$$

For SimpleRNNCell, $\boldsymbol{o}_t = \boldsymbol{h}_t$, is the same object. There's no additional linear layer conversion. $[\boldsymbol{h}_t]$ is wrapped in a List. This setting is for uniformity with RNN variants such as LSTM and GRU. In the initialization phase of the recurrent neural network, the state vector $\boldsymbol{h}_0$ is generally initialized to an all-zero vector, for example:

```
In [4]:
```

```
# Initialize state vector. Wrap with list, unified format
h0 = [tf.zeros([4, 64])]
x = tf.random.normal([4, 80, 100]) # Generate input tensor, 4 sentences of
80 words
xt = x[:,0,:] # The first word of all sentences
# Construct a Cell with input feature n=100, sequence length s=80, state
length=64
cell = layers.SimpleRNNCell(64)
out, h1 = cell(xt, h0) # Forward calculation
print(out.shape, h1[0].shape)
Out[4]: (4, 64) (4, 64)
```

It can be seen that after one timestamp calculation, the shape of the output and the state tensor are both [b, h], and the ids of the two are printed as follows:

```
In [5]:print(id(out), id(h1[0]))
Out[5]:2154936585256 2154936585256
```

The two ids are the same, that is, the state vector is directly used as the output vector. For the training of length s, it is necessary to loop through the Cell class s times to complete one forward operation of the network layer. E.g:

```
h = h0 # Save a list of state vectors on each time stamp
# Unpack the input in the dimension of the sequence length to get xt:[b,n]
for xt in tf.unstack(x, axis=1):
    out, h = cell(xt, h) # Forward calculation, both out and h are covered
# The final output can aggregate the output on each time stamp, or just take
the output of the last time stamp
out = out
```

The output variable out of the last time stamp will be the final output of the network. In fact, you can also save the output on each timestamp, and then sum or average it as the final output of the network.

11.4.2 Multi-layer SimpleRNNCell network

Like the convolutional neural network, although the recurrent neural network has been expanded many times on the time axis, it can only be counted as one network layer. By stacking multiple Cell classes in the depth direction, the network can achieve the same effect as a deep convolutional neural network, which greatly improves the expressive ability of the network. However, compared with the number of deep layers of tens or hundreds of convolutional neural networks, recurrent neural networks are prone to gradient diffusion and gradient explosion. Deep recurrent neural networks are very difficult to train. The current common recurrent neural network models generally have number of layers less than 10.

Here we take a two-layer recurrent neural network as an example to introduce the use of Cell class to build a multilayer RNN network. First create two SimpleRNNCell cells as follows:

```
x = tf.random.normal([4,80,100])
xt = x[:,0,:] # Take first timestamp of the input x0
# Construct 2 Cells, first cell0, then cell1, the memory state vector length
is 64
cell0 = layers.SimpleRNNCell(64)
cell1 = layers.SimpleRNNCell(64)
h0 = [tf.zeros([4,64])] # initial state vector of cell0
h1 = [tf.zeros([4,64])] # initial state vector of cell1
```

Calculate multiple times on the time axis to realize the forward operation of the entire network. The input xt on each time stamp first passes through the first layer to get the output out0, and then passes through the second layer to get the output out1. The code is as follows:

```
for xt in tf.unstack(x, axis=1):
    # xt is input and output is out0
    out0, h0 = cell0(xt, h0)
    # The output out0 of the previous cell is used as the input of this cell
    out1, h1 = cell1(out0, h1)
```

The above method first completes the propagation of the input on one time stamp on all layers, and then calculates the input on all time stamps in a loop.

In fact, it is also possible to first complete the calculation of all time stamps input on the first layer, and save the output list of the first layer on all time stamps, and then calculate the propagation of the second layer, the third layer, etc. as below:

```
# Save the output above all timestamps of the previous layer
middle_sequences = []
# Calculate the output on all timestamps of the first layer and save
for xt in tf.unstack(x, axis=1):
    out0, h0 = cell0(xt, h0)
    middle_sequences.append(out0)
# Calculate the output on all timestamps of the second layer
# If it is not the last layer, you need to save the output above all
timestamps
for xt in middle_sequences:
    out1, h1 = cell1(xt, h1)
```

In this way, we need an additional List to save the information of all timestamps in the previous layer: middle_sequences.append(out0). These two methods have the same effect, and you can choose the coding style you like.

It should be noted that each layer of the recurrent neural network at each time stamp has a state output. For subsequent tasks, which state output should we collect and is the most effective? Generally speaking, the state of the last-level Cell may preserve the global semantic features of the high-level, so the output of the last-level is generally used as the input of the subsequent task network. More specifically, the state output on the last timestamp of each layer contains the global information of the entire sequence. If you only want to use one state variable to complete subsequent tasks, such as sentiment classification problems, generally the output of the last layer at the last timestamp is most suitable.

11.4.3 SimpleRNN layer

Through the use of the SimpleRNNCell layer, we can understand every detail of the forward operation of the recurrent neural network. In actual use, for simplicity, we do not want to manually implement the internal calculation process of the recurrent neural network, such as the initialization of the state vector at each layer and the operation of each layer on the time axis. Using the SimpleRNN high-level interface can help us achieve this goal very conveniently.

For example, if we want to complete the forward operation of a single-layer recurrent neural network, it can be easily implemented as follows:

```
In [6]:
layer = layers.SimpleRNN(64) # Create a SimpleRNN layer with a state vector
length of 64
x = tf.random.normal([4, 80, 100])
out = layer(x) # Like regular convolutional networks, one line of code can
get the output
out.shape
Out[6]: TensorShape([4, 64])
```

As you can see, SimpleRNN can complete the entire forward operation process with only one line of code, and it returns the output on the last time stamp by default. If you want to return the output list on all timestamps, you can set return_sequences=True as follows:

```
In [7]:
# When creating the RNN layer, set the output to return all timestamps
layer = layers.SimpleRNN(64,return_sequences=True)
out = layer(x) # Forward calculation
out # Output, automatic concat operation
Out[7]:
<tf.Tensor: id=12654, shape=(4, 80, 64), dtype=float32, numpy=
array([[[ 0.31804922,  0.7904409 ,  0.13204293, ...,  0.02601025,
        -0.7833339 ,  0.65577114],…>
```

As you can see, the returned output tensor shape is [4,80,64], and the middle dimension 80 is the timestamp dimension. Similarly, we can achieve multi-layer recurrent neural networks by stacking multiple SimpleRNNs, such as a two-layer network, and its usage is similar to that of a normal network. E.g:

```
net = keras.Sequential([ # Build a 2-layer RNN network
# Except for the last layer, the output of all timestamps needs to be
returned to be used as the input of the next layer
layers.SimpleRNN(64, return_sequences=True),
layers.SimpleRNN(64),
])
```

```
out = net(x) # Forward calculation
```

Each layer needs the state output of the previous layer at each time stamp, so except for the last layer, all RNN layers need to return the state output at each time stamp, which is achieved by setting return_sequences=True. As you can see, using the SimpleRNN layer is similar to the usage of convolutional neural networks, which is very concise and efficient.

11.5 Hands-on RNN sentiment classification

Now let's use the basic RNN network to solve the sentiment classification problem. The network structure is shown in Figure 11.9. The RNN network has two layers. The semantic features of the sequence signal are extracted cyclically. The state vector $\boldsymbol{h}_s^{(2)}$ of the last time stamp of the second RNN layer is used as the global semantic feature representation of the sentence. It is sent to the classification network 3 formed by a fully connected layer, and the probability that the sample x is a positive emotion P (x is positive emotion | x) $\in [0,1]$ is obtained.

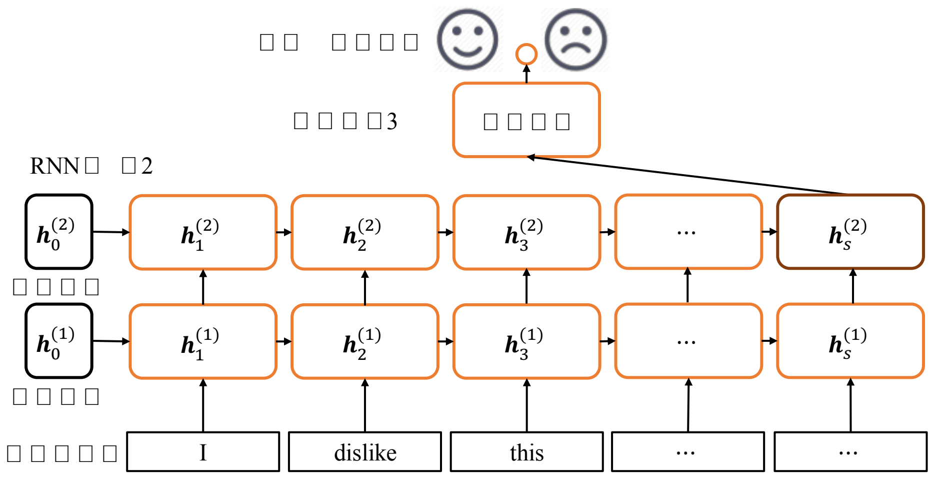


Figure 0.9 Network structure of sentiment classification task

11.5.1 Data Set

The classic IMDB movie review data set is used here to complete the sentiment classification task. The IMDB movie review data set contains 50,000 user reviews. The evaluation tags are divided into negative and positive. User reviews with IMDB rating <5 are marked as 0, which means negative; user reviews with IMDB rating >=7 are marked as 1, which means positive. 25,000 film reviews were used for the training set and 25,000 were used for the test set.

The IMDB data set can be loaded by datasets tool provided by Keras as follows:

```
In [8]:
batchsz = 128 # Batch size
total_words = 10000 # Vocabulary size N_vocab
```

```
max_review_len = 80 # The maximum length of the sentence s, the sentence
part greater than will be truncated, and the sentence less than will be
filled
embedding_len = 100 # Word vector feature length n
# Load the IMDB data set, the data here is coded with numbers, and a number
represents a word
(x_train, y_train), (x_test, y_test) =
keras.datasets.imdb.load_data(num_words=total_words)
# Print the input shape, the shape of the label
print(x_train.shape, len(x_train[0]), y_train.shape)
print(x_test.shape, len(x_test[0]), y_test.shape)
Out[8]:
(25000,) 218 (25000,)
(25000,) 68 (25000,)
```

As you can see, x_train and x_test are one-dimensional arrays with a length of 25,000. Each element of the array is a list of indefinite length, which stores each sentence encoded by numbers. For example, the first sentence of the training set has a total of 218 words, and the first sentence of the test set has 68 words, and each sentence contains the sentence start marker ID.

So how is each word encoded as a number? We can get the coding scheme by looking at its coding table, for example:

```
In [9]:
# Digital code table
word_index = keras.datasets.imdb.get_word_index()
# Print out the words and corresponding numbers in the coding table
for k,v in word_index.items():
   print(k,v)
Out[10]:
   …diamiter 88301
   moveis 88302
   mardi 14352
   wells' 11583
   850pm 88303…
```

Since the key of the coding table is a word and the value is an ID, the coding table is flipped and the coding ID of the flag bit is added. The code is as follows:

```
# The first 4 IDs are special bits
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["<PAD>"] = 0  # Fill flag
word_index["<START>"] = 1 # Start flag
word_index["<UNK>"] = 2  # Unknown word sign
word_index["<UNUSED>"] = 3
```

```
# Flip code table
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])
```

For a digitally encoded sentence, it is converted into string data by the following function:

```
def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
```

For example, to convert a sentence, the code is as follows:

```
In [11]:decode_review(x_train[0])
Out[11]:
"<START> this film was just brilliant casting location scenery story
direction everyone's…<UNK> father came from…
```

For sentences with uneven lengths, a threshold is artificially set. For sentences larger than this length, select some words to be truncated, you can choose to cut off the beginning of the sentence or the end of the sentence. For sentences less than this length, you can choose to fill at the beginning or end of a sentence. The sentence truncation function can be conveniently realized by the keras.preprocessing.sequence.pad_sequences() function, for example:

```
# Truncate and fill sentences so that they are of equal length, here long
sentences retain the part behind the sentence, and short sentences are
filled in front
x_train = keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=max_review_len)
x_test = keras.preprocessing.sequence.pad_sequences(x_test,
maxlen=max_review_len)
```

After truncating or filling to the same length, wrap it into a data set object through the Dataset class, and add the commonly used data set processing flow, the code is as follows:

```
In [12]:
# Build a data set, break up, batch, and discard the last batch that is not
enough batchsz
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)
# Statistical data set attributes
print('x_train shape:', x_train.shape, tf.reduce_max(y_train),
tf.reduce_min(y_train))
print('x_test shape:', x_test.shape)
Out[12]:
x_train shape: (25000, 80) tf.Tensor(1, shape=(), dtype=int64) tf.Tensor(0,
shape=(), dtype=int64)
x_test shape: (25000, 80)
```

It can be seen that the sentence length after truncation and filling is unified to 80, which is the set sentence length threshold. The drop_remainder=True parameter discards the last batch, because its real batch size may be smaller than the preset batch size.

11.5.2 Network Model

We create a custom model class MyRNN, inherited from the Model base class, we need to create a new Embedding layer, two RNN layers, and one classification layer as follows:

```
class MyRNN(keras.Model):
    # Use Cell method to build a multi-layer network
    def __init__(self, units):
        super(MyRNN, self).__init__()
        # [b, 64], construct Cell initialization state vector, reuse
        self.state0 = [tf.zeros([batchsz, units])]
        self.state1 = [tf.zeros([batchsz, units])]
        # Word vector encoding [b, 80] => [b, 80, 100]
        self.embedding = layers.Embedding(total_words, embedding_len,
                                          input_length=max_review_len)
        # Construct 2 Cells and use dropout technology to prevent overfitting
        self.rnn_cell0 = layers.SimpleRNNCell(units, dropout=0.5)
        self.rnn_cell1 = layers.SimpleRNNCell(units, dropout=0.5)
        # Construct a classification network to classify the output features
of CELL, 2 classification
        # [b, 80, 100] => [b, 64] => [b, 1]
        self.outlayer = layers.Dense(1)
```

The word vector is encoded as length n=100, and the state vector length of RNN is h=units. The classification network completes a binary classification task, so the output node is set to 1.

The forward propagation logic is as follows: the input sequence completes the word vector encoding through the Embedding layer, loops through the two RNN layers to extracts semantic features, takes the state vector output of the last time stamp of the last layer and sends it to the classification network. The output probability is obtained after the Sigmoid activation function as below:

```
    def call(self, inputs, training=None):
        x = inputs # [b, 80]
        # Word vector embedding: [b, 80] => [b, 80, 100]
        x = self.embedding(x)
        # Pass 2 RNN CELLs,[b, 80, 100] => [b, 64]
        state0 = self.state0
        state1 = self.state1
        for word in tf.unstack(x, axis=1): # word: [b, 100]
            out0, state0 = self.rnn_cell0(word, state0, training)
            out1, state1 = self.rnn_cell1(out0, state1, training)
        # Last layer's last time stamp as the network output: [b, 64] => [b,
1]
        x = self.outlayer(out1, training)
        # Pass through activation function, p(y is pos|x)
        prob = tf.sigmoid(x)

        return prob
```

11.5.3 Training and testing

For simplicity, here we use Keras' Compile&Fit method to train the network. Set the optimizer to Adam optimizer, the learning rate is 0.001, the error function uses the 2-class cross-entropy loss function BinaryCrossentropy, and the test metric uses the accuracy rate. The code is as below:

```
def main():
    units = 64 # RNN state vector length n
    epochs = 20 # Training epochs

    model = MyRNN(units) # Create the model
    # Compile
    model.compile(optimizer = optimizers.Adam(0.001),
                  loss = losses.BinaryCrossentropy(),
                  metrics=['accuracy'])
    # Fit and validate
    model.fit(db_train, epochs=epochs, validation_data=db_test)
    # Test
    model.evaluate(db_test)
```

After 20 Epoch training, the network achieves 80.1% accuracy rate at testing dataset.

11.6 Gradient vanishing and gradient exploding

The training of recurrent neural networks is not stable, and the depth of the network cannot be arbitrarily deepened. Why do recurrent neural networks have difficulty in training? Let's briefly review of the key expressions in the gradient derivation:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i} = \prod_{j=i}^{t-1} diag\left(\sigma'\left(\boldsymbol{W}_{xh}\boldsymbol{x}_{j+1} + \boldsymbol{W}_{hh}\boldsymbol{h}_j + \boldsymbol{b}\right)\right)\boldsymbol{W}_{hh}$$

In other words, the gradient $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ from time stamp i to time stamp t includes the continuous multiplication operation of $\boldsymbol{W}_{hh}$. When the largest eigenvalue of $\boldsymbol{W}_{hh}$ is less than 1, multiple consecutive multiplication operations will make the element value of $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ close to zero; when the value of $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ is greater than 1, multiple consecutive multiplication operations will make the value of $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$ explosively increase.

We can intuitively feel the generation of gradient vanishing and gradient exploding from the following two examples:

In [13]:

```
W = tf.ones([2,2]) # Create a matrix
eigenvalues = tf.linalg.eigh(W)[0] # Calculate eigenvalue
eigenvalues
```

Out[13]:

```
<tf.Tensor: id=923, shape=(2,), dtype=float32, numpy=array([0., 2.],
dtype=float32)>
```

It can be seen that the maximum eigenvalue of the all-one matrix is 2. Calculate the $\boldsymbol{W}^1 \sim \boldsymbol{W}^{10}$ of the $\boldsymbol{W}$ matrix and draw it as a graph of the power and the L2-norm of the matrix, as shown in Figure 11.10. It can be seen that when the maximum eigenvalue of the $\boldsymbol{W}$ matrix is greater than 1, the matrix multiplication will make the result larger and larger.

```
val = [W]
for i in range(10): # Matrix multiplication n times
    val.append([val[-1]@W])
# Calculate L2 norm
norm = list(map(lambda x:tf.norm(x).numpy(),val))
```

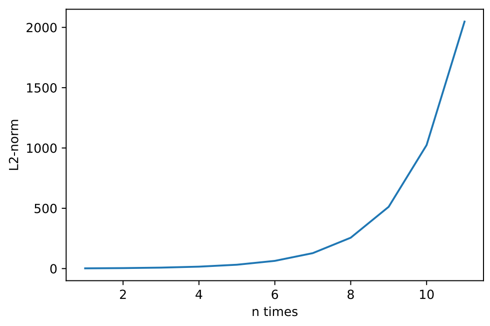


Figure 0.10 Matrix multiplication when the largest eigenvalue is greater than 1

Consider the case when the maximum eigenvalue is less than 1.

In [14]:

```
W = tf.ones([2,2])*0.4 # Create a matrix
eigenvalues = tf.linalg.eigh(W)[0] # Calculate eigenvalues
print(eigenvalues)
```

Out[14]:

```
tf.Tensor([0.  0.8], shape=(2,), dtype=float32)
```

It can be seen that the maximum eigenvalue of the $\boldsymbol{W}$ matrix at this time is 0.8. In the same way, considering the results of multiple multiplications of the $\boldsymbol{W}$ matrix as follows:

```
val = [W]
for i in range(10):
    val.append([val[-1]@W])
# Calculate the L2 norm
norm = list(map(lambda x:tf.norm(x).numpy(),val))
plt.plot(range(1,12),norm)
```

Its L2-norm curve is shown in Figure 11.11. It can be seen that when the maximum eigenvalue of the $\boldsymbol{W}$ matrix is less than 1, the matrix multiplication will make the result smaller and smaller, close to 0.

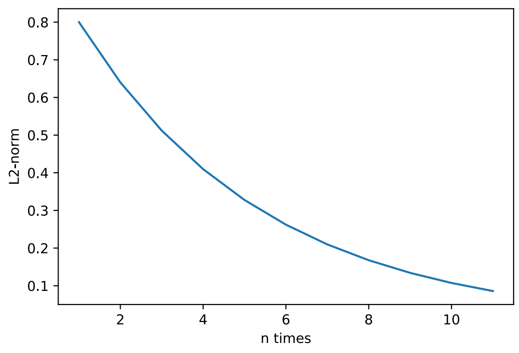


Figure 0.11 Matrix multiplication when the largest eigenvalue is less than 1

We call the phenomenon that the gradient value is close to 0 Gradient Vanishing, and the phenomenon that the gradient value is far greater than 1 is called Gradient Exploding. Gradient Vanishing and Gradient Exploding are two situations that are easy to appear in the process of neural network optimization, and they are also not conducive to network training.

Consider the gradient descent algorithm:

$$\theta' = \theta - \eta \nabla_\theta \mathcal{L}$$

When gradient vanishing occurs, $\nabla_\theta \mathcal{L} \approx 0$, at this time $\theta' \approx \theta$, which means that the parameters remain unchanged after each gradient update, and the parameters of the neural network cannot be updated for a long time. The specific performance is that $\mathcal{L}$ has almost no change, other evaluation indicators, such as accuracy, also remain the same. When the gradient exploding occurs, $\nabla_\theta \mathcal{L} \gg 1$, the update step size of the gradient $\eta \nabla_\theta \mathcal{L}$ is very large, so that the updated θ' and θ are very different, and the network $\mathcal{L}$ has a sudden change, and even oscillates back and forth with non-convergence.

By deriving the gradient propagation formula of the recurrent neural network, we found that the recurrent neural network is prone to gradient vanishing and gradient exploding. So how to solve these two problems?

11.6.1 Gradient Clipping

Gradient exploding can be solved to a certain extent by Gradient Clipping. Gradient Clipping is very similar to tensor limiting. It also limits the value or norm of the gradient tensor to a small interval, thereby reducing the gradient value far greater than 1 and avoiding gradient exploding.

In deep learning, there are three commonly used Gradient Clipping methods.

- ❑ Limit the value of the tensor directly so that all the elements of the tensor $\boldsymbol{W}$ are $w_{ij} \in [\min, \max]$. In TensorFlow, it can be achieved through the tf.clip_by_value() function. E.g:

```
In [15]:
a=tf.random.uniform([2,2])
tf.clip_by_value(a,0.4,0.6) # Gradient value clipping
Out[15]:
<tf.Tensor: id=1262, shape=(2, 2), dtype=float32, numpy=
array([[0.5410726, 0.6      ],
       [0.4      , 0.6      ]], dtype=float32)>
```

- ❑ Limit the norm of the gradient tensor $\boldsymbol{W}$. For example, the L2 norm of W - $\|\boldsymbol{W}\|_2$ is constrained between [0,max]. If $\|\boldsymbol{W}\|_2$ is greater than the max value, use

$$\boldsymbol{W}' = \frac{\boldsymbol{W}}{\|\boldsymbol{W}\|_2} \cdot \max$$

to restrict $\|\boldsymbol{W}\|_2$ to max. This can be done through the tf.clip_by_norm function. E.g:

```
In [16]:
a=tf.random.uniform([2,2]) * 5
# Clip by norm
b = tf.clip_by_norm(a, 5)
# Norm before and after clipping
tf.norm(a),tf.norm(b)
Out[16]:
(<tf.Tensor: id=1338, shape=(), dtype=float32, numpy=5.380655>,
 <tf.Tensor: id=1343, shape=(), dtype=float32, numpy=5.0>)
```

It can be seen that for tensors with L2 norm greater than max, the norm value is reduced to 5 after clipping.

- ❑ The update direction of the neural network is represented by the gradient tensor $\boldsymbol{W}$ of all parameters. The first two methods only consider a single gradient tensor, and so the update direction of the network may change. If the norm of the gradient $\boldsymbol{W}$ of all parameters can be considered, and equal scaling can be achieved, then the gradient value of the network can be well restricted without changing the update direction of the network. This is the third method of gradient clipping: global norm clipping. In TensorFlow, the norm of the overall network gradient $\boldsymbol{W}$ can be quickly scaled through the tf.clip_by_global_norm function.

Let $\boldsymbol{W}^{(i)}$ denote the *i*-th gradient tensor of the network parameters. Use the following formula to calculate the global norm of the network.

$$\text{global\_norm} = \sqrt{\sum_i \|W^{(i)}\|_2{}^2}$$

For the i-th parameter $\boldsymbol{W}^{(i)}$, use the following formula to clip.

$$\boldsymbol{W}^{(i)} = \frac{\boldsymbol{W}^{(i)} \cdot \text{max\_norm}}{\max(\text{global\_norm}, \text{max\_norm})}$$

where max_norm is the global maximum norm value specified by the user. E.g:

```
In [17]:
w1=tf.random.normal([3,3]) # Create gradient tensor 1
w2=tf.random.normal([3,3]) # Create gradient tensor 2
# Calculate global norm
global_norm=tf.math.sqrt(tf.norm(w1)**2+tf.norm(w2)**2)
# Clip by global norm and max norm=2
(ww1,ww2),global_norm=tf.clip_by_global_norm([w1,w2],2)
# Calcualte global norm after clipping
global_norm2 = tf.math.sqrt(tf.norm(ww1)**2+tf.norm(ww2)**2)
# Print the global norm before cropping and the global norm after cropping
print(global_norm, global_norm2)
Out[17]:
tf.Tensor(4.1547523, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
```

It can be seen that after clipping, the global norm of the gradient group of the network parameters is reduced to max_norm=2. It should be noted that tf.clip_by_global_norm returns two objects of the clipped tensor - List and global_norm, where global_norm represents the global norm sum of the gradient before clipping.

Through gradient clipping, the gradient exploding phenomenon can be suppressed. As shown in Figure 11.12 below, the error value J of the $J(w, b)$ function represented by the surface in the figure under different network parameters w and b. There is a region where the gradient of the $J(w, b)$ function changes greatly. When parameters enter this area, gradient exploding are prone to occur, which makes the network state deteriorate rapidly. Figure 11.12 on the right shows the optimized trajectory after adding gradient clipping. Since the gradient is effectively restricted, the step size of each update is effectively controlled, thereby preventing the network from suddenly deteriorating.

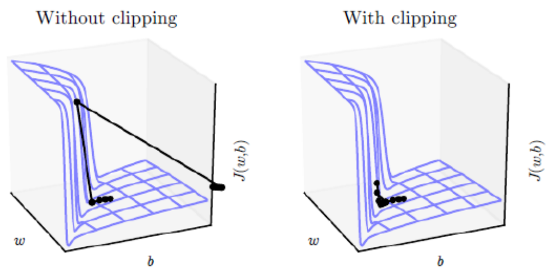


Figure 0.12 Diagram of the optimized trajectory of gradient clipping [1]

During network training, gradient clipping is generally performed after the gradient is calculated and before the gradient is updated. E.g:

```
with tf.GradientTape() as tape:
  logits = model(x) # Forward calculation
  loss = criteon(y, logits) # Calculate error
# Calcualte gradients
grads = tape.gradient(loss, model.trainable_variables)
grads, _ = tf.clip_by_global_norm(grads, 25) # Global norm clipping
# Update parameters using clipped gradient
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

11.6.2 Gradient vanishing

The gradient vanishing phenomenon can be suppressed by a series of measures such as increasing the learning rate, reducing the network depth, and adding Skip Connection.

Increasing the learning rate η can prevent gradient vanishing to a certain extent. When gradient vanishing occurs, the gradient of the network $\nabla_\theta\mathcal{L}$ is close to 0. At this time, if the learning rate η is also small, such as η=$1\text{e}-5$, the gradient update step is even smaller. By increasing the learning rate, such as letting $\eta = 1\text{e}-2$, it is possible to quickly update the state of the network and escape the gradient vanishing area.

For deep neural networks, the gradient gradually propagates from the last layer to the first layer, and gradient vanishing is generally more likely to appear in the first few layers of the network. Before the emergence of deep residual networks, it was very difficult to train deep networks with dozens or hundreds of layers. The gradients of the previous layers of the network were very prone to gradient vanishing, which made the network parameters not updated for a long time. The deep residual network better overcomes the gradient vanishing phenomenon, so that the number of neural network layers can reach hundreds or thousands. Generally speaking, reducing the network depth can reduce the gradient vanishing phenomenon, but after the number of network layers is reduced, the network expression ability will be weaker.

11.7 RNN short-term memory

In addition to the training difficulty of recurrent neural networks, there is a more serious problem, that is, short-term memory. Consider a long sentence:

Today's weather is so beautiful, even though an unpleasant thing happened on the road..., I immediately adjusted my state and happily prepared for a beautiful day.

According to our understanding, the reason why we " happily prepared for a beautiful day" is that "Today's weather is so beautiful" which is mentioned at the beginning of the sentence. It can be seen that humans can understand long sentences well, but recurrent neural networks are not necessarily. Researchers have found that when recurrent neural networks process long sentences, they can only understand information within a limited length, while useful information in a longer range cannot be used well. We call this phenomenon short-term memory.

So, can this short-term memory be prolonged so that the recurrent neural network can effectively use the training data in a longer range, thereby improving model performance? In 1997, Swiss artificial intelligence scientist Jürgen Schmidhuber proposed the Long Short-Term

Memory (LSTM) model. Compared with the basic RNN network, LSTM has longer memory and is better at processing longer sequence data. After LSTM was proposed, it has been widely used in tasks such as sequence prediction and natural language processing, almost replacing the basic RNN model .

Next, we will introduce the more popular and powerful LSTM network.

11.8 LSTM principle

The basic RNN network structure is shown in Figure 11.13. After the state vector $\boldsymbol{h}_{t-1}$ of the previous time stamp and the input $\boldsymbol{x}_t$ of the current time stamp are linearly transformed, the new state vector $\boldsymbol{h}_t$ is obtained through the activation function tanh. Compared with the basic RNN network which has only one state vector $\boldsymbol{h}_t$, LSTM adds a new state vector $\boldsymbol{C}_t$, and at the same time introduces a gate control mechanism, which controls the forgetting and updating of information through the gate control unit, as shown in Figure 11.14.

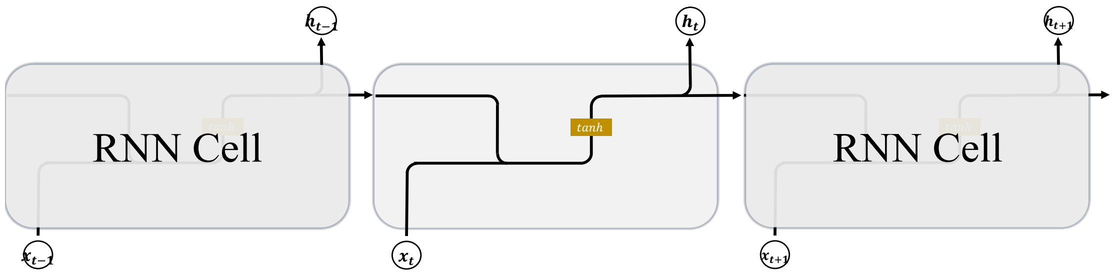


Figure 0.13 Basic RNN structure

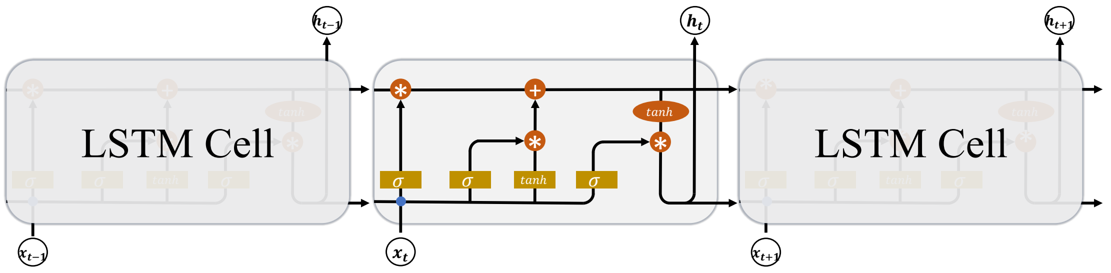


Figure 0.14 LSTM structure

In LSTM, there are two state vectors $\boldsymbol{c}$ and $\boldsymbol{h}$, where $\boldsymbol{c}$ is the internal state vector of LSTM, which can be understood as the memory state vector of LSTM, and $\boldsymbol{h}$ represents the output vector of LSTM. Compared with the basic RNN, LSTM separates the internal memory and output into two variables, and uses three gates: Input Gate, Forget Gate and Output Gate to control the internal information flow.

The gate mechanism can be understood as a way of controlling the data flow, analogous to a water valve: when the water valve is fully opened, water flows unimpeded; when the water valve is fully closed, the water flow is completely blocked. In LSTM, the valve opening degree are represented by the gate control value vector $\boldsymbol{g}$, as shown in Figure 11.15, the gate control is compressed to the interval between [0,1] through the $\sigma(\boldsymbol{g})$ activation function. When $\sigma(\boldsymbol{g}) = 0$, all gates are closed, and output is $\boldsymbol{o} = 0$. When $\sigma(\boldsymbol{g}) = 1$, all gates are open, and output is $\boldsymbol{o} = \boldsymbol{x}$. Through the gate mechanism, the data flow can be better controlled.

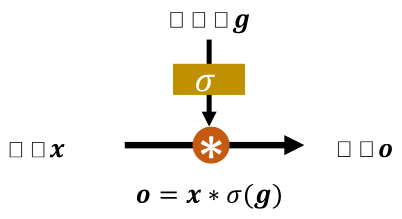


Figure 0.15 Gate mechanism

Below we respectively introduce the principles and functions of the three gates.

11.8.1 Forget gate

The forget gate acts on the LSTM state vector $\boldsymbol{c}$ to control the impact of the memory $\boldsymbol{c}_{t-1}$ of the previous time stamp on the current time stamp. As shown in Figure 11.16, the control variable $\boldsymbol{g}_f$ of the forget gate is determined by

$$\boldsymbol{g}_f = \sigma\left(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f\right)$$

where $\boldsymbol{W}_f$and $\boldsymbol{b}_f$ are the parameter tensors of the forget gate, which can be automatically optimized by the backpropagation algorithm. σ is the activation function, and the Sigmoid function is generally used. When $\boldsymbol{g}_f = 1$, the forget gates are all open, and LSTM accepts all the information of the previous state $\boldsymbol{c}_{t-1}$. When the gating $\boldsymbol{g}_f = 0$, the forget gate is closed, and LSTM directly ignores $\boldsymbol{c}_{t-1}$, and the output is a vector of 0. This is why it's called the Forget Gate.

After passing through the forget gate, the state vector of LSTM becomes $\boldsymbol{g}_f \boldsymbol{c}_{t-1}$.

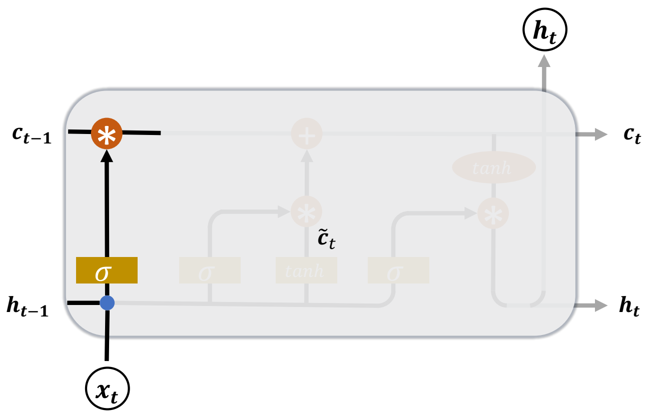


Figure 0.16 Forget Gate

11.8.2 Input gate

The input gate is used to control the degree to which the LSTM receives input. First, the new input vector $\tilde{\boldsymbol{c}}_t$ is obtained by nonlinear transformation of the input $\boldsymbol{x}_t$ of the current time stamp and the output $\boldsymbol{h}_{t-1}$ of the previous time stamp:

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c)$$

where $\boldsymbol{W}_c$ and $\boldsymbol{b}_c$ are the parameters of the input gate, which need to be automatically optimized by the back propagation algorithm, and tanh is the activation function, which is used to normalize the input to [-1,1]. $\tilde{\boldsymbol{c}}_t$ does not completely refresh the memory that enters the LSTM, but controls the amount of input received through the input gate. The control variables of the input gate also come from the input $\boldsymbol{x}_t$ and the output $\boldsymbol{h}_{t-1}$:

$$\boldsymbol{g}_i = \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i)$$

where $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ are the parameters of the input gate, which need to be automatically optimized by the back propagation algorithm, and σ is the activation function, and the Sigmoid function is generally used. The input gate control variable $\boldsymbol{g}_i$ determines how LSTM accepts the new input $\tilde{\boldsymbol{c}}_t$ of the current time stamp: when $\boldsymbol{g}_i = 0$, LSTM does not accept any new input $\tilde{\boldsymbol{c}}_t$; when $\boldsymbol{g}_i = 1$, LSTM accepts all new input $\tilde{\boldsymbol{c}}_t$, As shown in Figure 11.17.

After passing through the input gate, the vector to be written into Memory is $\boldsymbol{g}_i\tilde{\boldsymbol{c}}_t$.

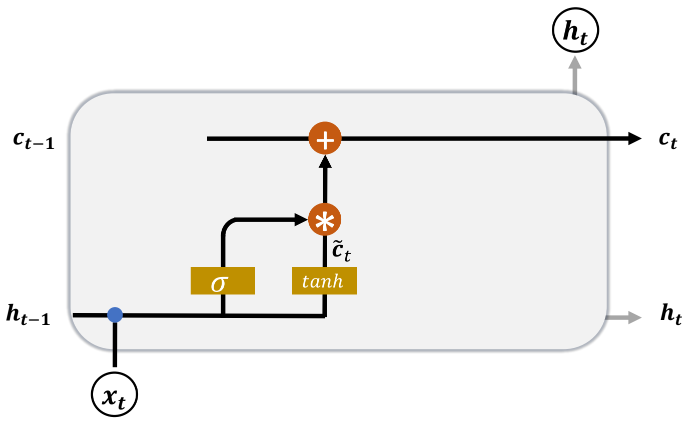


Figure 0.17 Input gate

11.8.3 Update Memory

Under the control of the forget gate and the input gate, LSTM selectively reads the memory $\boldsymbol{c}_{t-1}$ of the previous time stamp and the new input $\tilde{\boldsymbol{c}}_t$ of the current time stamp. The refresh mode of the state vector $\boldsymbol{c}_t$ is:

$$\boldsymbol{c}_t = \boldsymbol{g}_i\tilde{\boldsymbol{c}}_t + \boldsymbol{g}_f\boldsymbol{c}_{t-1}$$

The new state vector $\boldsymbol{c}_t$ obtained is the state vector of the current time stamp, as shown in Figure 11.17.

11.8.4 Output gate

The internal state vector $\boldsymbol{c}_t$ of LSTM is not directly used for output, which is different from the basic RNN. The state vector $\boldsymbol{h}$ of the basic RNN network is used for both memory and output, so the basic RNN can be understood as the state vector $\boldsymbol{c}$ and the output vector $\boldsymbol{h}$ are the same object. In LSTM, the state vector is not totally outputted, but selectively under the action of the output gate. The gate variable $\boldsymbol{g}_o$ of the output gate is:

$$\boldsymbol{g}_o = \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o)$$

where $\boldsymbol{W}_o$ and $\boldsymbol{b}_o$ are the parameters of the output gate, which also need to be automatically optimized by the back propagation algorithm. σ is the activation function, and the Sigmoid function is generally used. When the output gate $\boldsymbol{g}_o = 0$, the output is closed, and the internal memory of LSTM is completely blocked and cannot be used as an output. At this time, the output is a vector of 0; when the output gate $\boldsymbol{g}_o = 1$, the output is fully open, and the LSTM state vector $\boldsymbol{c}_t$ is all used for output. The output of LSTM is composed of:

$$\boldsymbol{h}_t = \boldsymbol{g}_o \cdot \tanh(\boldsymbol{c}_t)$$

That is, the memory vector $\boldsymbol{c}_t$ interacts with the input gate after passing the tanh activation function to obtain the output of the LSTM. Since $\boldsymbol{g}_o \in [0,1]$ and $\tanh(\boldsymbol{c}_t) \in [-1,1]$, the output of LSTM is $\boldsymbol{h}_t \in [-1,1]$.

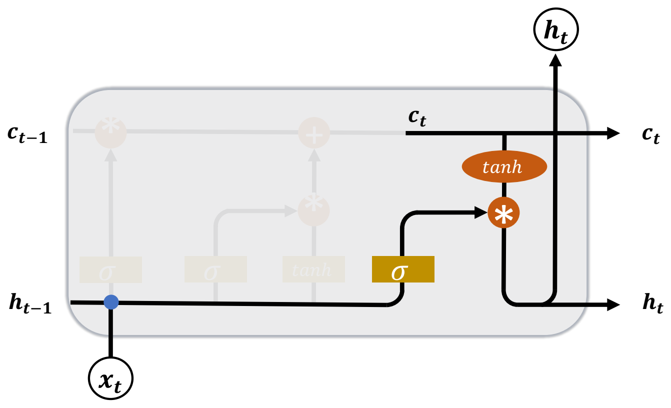


Figure 0.18 Output gate

11.8.5 Summary

Although LSTM has a large number of state vectors and gates, the calculation process is relatively complicated. But since each gate control function is clear, the role of each state is also easier to understand. Here, the typical gating behavior is listed and the LSTM behavior of the code is explained, as shown in Table 11.1.

Table 0.1 Typical behavior of input gate and forget gate

Input gating	Forget Gating	LSTM behavior
0	1	Only use memory
1	1	Integrated input and memory
0	0	Clear memory
1	0	Input overwrites memory

11.9 How to use the LSTM layer

In TensorFlow, there are also two ways to implement LSTM networks. Either LSTMCell can be used to manually complete the cyclic operation on the time stamp, or the forward operation can be completed in one step through the LSTM layer.

11.9.1 LSTMCell

The usage of LSTMCell is basically the same as SimpleRNNCell. The difference is that there are two state variables - List for LSTM, namely $[\boldsymbol{h}_t, \boldsymbol{c}_t]$, which need to be initialized separately. The first element of List is $\boldsymbol{h}_t$ and the second element is $\boldsymbol{c}_t$. When the cell is called to complete the forward operation, two elements are returned. The first element is the output of the cell, which is $\boldsymbol{h}_t$, and the second element is the updated state List of the cell: $[\boldsymbol{h}_t, \boldsymbol{c}_t]$. First create a new LSTM Cell with a state vector length of $h = 64$, where the length of the state vector $\boldsymbol{c}_t$ and the output vector $\boldsymbol{h}_t$ are both h. The code is as follows:

```
In [18]:

x = tf.random.normal([2,80,100])

xt = x[:,0,:] # Get a timestamp input

cell = layers.LSTMCell(64) # Create LSTM Cell

# Initialization state and output List,[h,c]

state = [tf.zeros([2,64]),tf.zeros([2,64])]

out, state = cell(xt, state) # Forward calculation

# View the id of the returned element

id(out),id(state[0]),id(state[1])

Out[18]: (1537587122408, 1537587122408, 1537587122728)
```

It can be seen that the returned output out is the same as the id of the first element $\boldsymbol{h}_t$ of the List, which is consistent with the original intention of the basic RNN and is for the unification of the format.

By unrolling the loop operation on the timestamp, the forward propagation of a layer can be completed, and the writing method is the same as the basic RNN. E.g:

```
# Untie it in the sequence length dimension, and send it to the LSTM Cell
unit in a loop

for xt in tf.unstack(x, axis=1):

    # Forward calculation

    out, state = cell(xt, state)
```

The output can use only the output on the last time stamp, or it can aggregate the output vectors on all time stamps.

11.9.2 LSTM layer

Through the layers.LSTM layer, the operation of the entire sequence can be conveniently completed at one time. First create a new LSTM network layer, for example:

```
# Create an LSTM layer with a memory vector length of 64

layer = layers.LSTM(64)
```

```
# The sequence passes through the LSTM layer and returns the output h of the
last time stamp by default
out = layer(x)
```

After forward propagation through the LSTM layer, only the output of the last timestamp will be returned by default. If you need to return the output above each timestamp, you need to set the return_sequences=True. E.g:

```
# When creating the LSTM layer, set to return the output on each timestamp
layer = layers.LSTM(64, return_sequences=True)
# Forward calculation, the output on each timestamp is automatically
concated to form a tensor
out = layer(x)
```

The out returned at this time contains the status output above all timestamps, and its shape is [2,80,64], where 80 represents 80 timestamps.

For multi-layer neural networks, you can wrap multiple LSTM layers with Sequential containers, and set all non-final layer networks return_sequences=True, because the non-final LSTM layer needs the output of all timestamps of the previous layer as input. E.g:

```
# Like the CNN network, LSTM can also be simply stacked layer by layer
net = keras.Sequential([
    layers.LSTM(64, return_sequences=True), # The non-final layer needs to
return all timestamp output
    layers.LSTM(64)
])
# Once through the network model, you can get the output of the last layer
and the last time stamp
out = net(x)
```

11.10 GRU introduction

LSTM has a longer memory capacity and has achieved better performance than the basic RNN model on most sequence tasks. More importantly, LSTM is not prone to gradient vanishing. However, the LSTM structure is relatively complex, the calculation cost is high, and the model parameters are large. Therefore, scientists try to simplify the calculation process inside LSTM, especially to reduce the number of gates. Studies found that the forget gate is the most important gate control in LSTM [2], and even found that the simplified version of the network with only the forget gate is better than the standard LSTM network on multiple benchmark data sets. Among many simplified versions of LSTM, Gated Recurrent Unit (GRU) is one of the most widely used RNN variants. GRU merges the internal state vector and output vector into a state vector $\boldsymbol{h}$, and the number of gates is also reduced to two: Reset Gate and Update Gate, as shown in Figure 11.19.

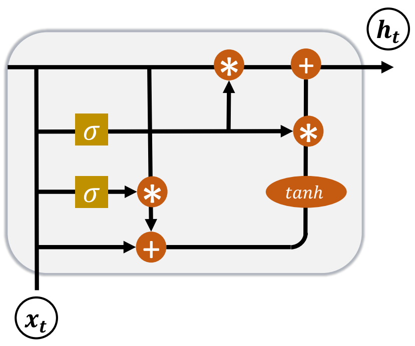


Figure 0.19 GRU network structure

Let's introduce the principle and function of reset gate and update gate respectively.

11.10.1 Reset door

The reset gate is used to control the amount of the state $\boldsymbol{h}_{t-1}$ of the last time stamp into the GRU. The gating vector $\boldsymbol{g}_r$ is obtained by transforming the current time stamp input $\boldsymbol{x}_t$ and the last time stamp state $\boldsymbol{h}_{t-1}$, the relationship is as follows:

$$\boldsymbol{g}_r = \sigma(\boldsymbol{W}_r[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_r)$$

where $\boldsymbol{W}_r$ and $\boldsymbol{b}_r$ are the parameters of the reset gate, which are automatically optimized by the back propagation algorithm, σ is the activation function, and the Sigmoid function is generally used. The gating vector $\boldsymbol{g}_r$ only controls the state $\boldsymbol{h}_{t-1}$, but not the input $\boldsymbol{x}_t$:

$$\widetilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h[\boldsymbol{g}_r \boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_h)$$

When $\boldsymbol{g}_r = 0$, the new input $\widetilde{\boldsymbol{h}}_t$ all comes from the input $\boldsymbol{x}_t$, and $\boldsymbol{h}_{t-1}$ is not accepted, which is equivalent to resetting $\boldsymbol{h}_{t-1}$. When $\boldsymbol{g}_r = 1$, $\boldsymbol{h}_{t-1}$ and input $\boldsymbol{x}_t$ jointly generate a new input $\widetilde{\boldsymbol{h}}_t$, as shown in Figure 11.20.

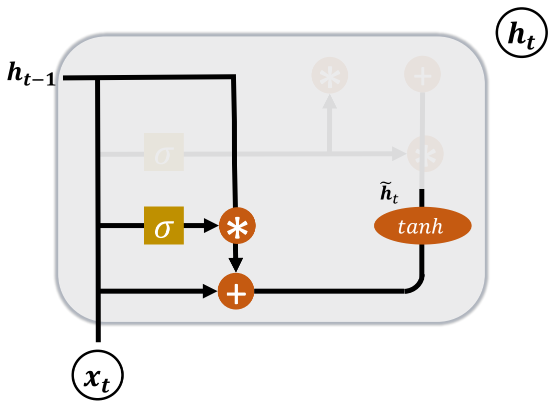


Figure 0.20 Reset gate

11.10.2 Update gate

The update gate controls the degree of influence of the last time stamp state $\boldsymbol{h}_{t-1}$ and the new input $\widetilde{\boldsymbol{h}}_t$ on the new state vector $\boldsymbol{h}_t$. Update the gating vector $\boldsymbol{g}_z$ by

$$\boldsymbol{g}_z = \sigma(\boldsymbol{W}_z[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_z)$$

where $\boldsymbol{W}_z$ and $\boldsymbol{b}_z$ are the parameters of the update gate, which are automatically optimized by the back propagation algorithm, σ is the activation function, and the Sigmoid function is generally used. $\boldsymbol{g}_z$ is used to control the new input $\widetilde{\boldsymbol{h}}_t$ signal, and $\mathbf{1}-\boldsymbol{g}_z$ is used to control the state $\boldsymbol{h}_{t-1}$ signal:

$$\boldsymbol{h}_t = (\mathbf{1}-\boldsymbol{g}_z)\boldsymbol{h}_{t-1} + \boldsymbol{g}_z\widetilde{\boldsymbol{h}}_t$$

It can be seen that the updates of $\widetilde{\boldsymbol{h}}_t$ and $\boldsymbol{h}_{t-1}$ to $\boldsymbol{h}_t$ are in a state of competing with each other. When the update gate $\boldsymbol{g}_z = 0$, all $\boldsymbol{h}_t$ comes from the last time stamp state $\boldsymbol{h}_{t-1}$; when the update gate $\boldsymbol{g}_z = 1$, all $\boldsymbol{h}_t$ comes from the new input $\widetilde{\boldsymbol{h}}_t$.

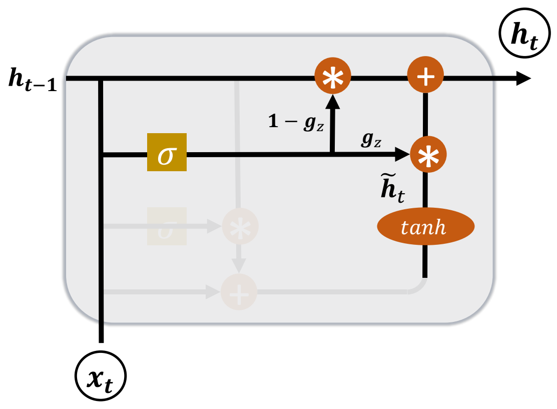


Figure 0.21 Update gate

11.10.3 How to use GRU

Similarly, in TensorFlow, there are also Cell and layer methods to implement GRU networks. The usage of GRUCell and GRU layer is very similar to the previous SimpleRNNCell, LSTMCell, SimpleRNN and LSTM. First, use GRUCell to create a GRU Cell object, and cyclically unroll operations on the time axis. E.g:

```
In [19]:
# Initialize the state vector, there is only one GRU
h = [tf.zeros([2,64])]
cell = layers.GRUCell(64) # New GRU Cell, vector length is 64
# Untie in the timestamp dimension, loop through the cell
for xt in tf.unstack(x, axis=1):
    out, h = cell(xt, h)
# Out shape
out.shape
Out[19]:TensorShape([2, 64])
```

You can easily create a GRU network layer through the layers.GRU class, and stack a network of multiple GRU layers through the Sequential container. E.g:

```
net = keras.Sequential([
    layers.GRU(64, return_sequences=True),
    layers.GRU(64)
])
```

```
out = net(x)
```

11.11 Hands-on LSTM/GRU sentiment classification

Earlier we introduced the sentiment classification problem, and used the SimpleRNN model to solve the problem. After introducing the more powerful LSTM and GRU networks, we upgraded the network model. Thanks to the unified format of TensorFlow's recurrent neural network related interfaces, only a few modifications on the original code can be perfectly upgraded to the LSTM or GRU model.

11.11.1 LSTM model

First, let's use the Cell method. There are two state lists of the LSTM network, and the $\boldsymbol{h}$ and $\boldsymbol{c}$ vectors of each layer need to be initialized respectively. E.g:

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

Modify the model to LSTMCell model as below:

```
self.rnn_cell0 = layers.LSTMCell(units, dropout=0.5)
self.rnn_cell1 = layers.LSTMCell(units, dropout=0.5)
```

对于层方式，只需要修改网络模型一处即可，修改如下：

Other codes can run without modification. For the layer method, only one part of the network model needs to be modified, as follows:

```
# Build RNN, replace with LSTM class
self.rnn = keras.Sequential([
    layers.LSTM(units, dropout=0.5, return_sequences=True),
    layers.LSTM(units, dropout=0.5)
])
```

11.11.2 GRU model

For the Cell method, there is only one GRU state List. Like the basic RNN, you only need to modify the type of cell created. The code is as follows:

```
# Create 2 Cells
self.rnn_cell0 = layers.GRUCell(units, dropout=0.5)
self.rnn_cell1 = layers.GRUCell(units, dropout=0.5)
```

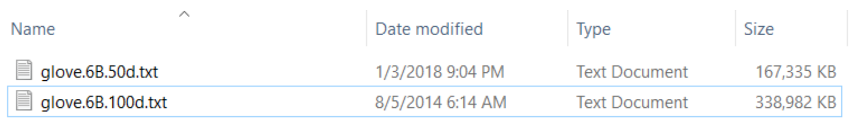
For the layer method, just modify the network layer type as follows:

```
# Create RNN
self.rnn = keras.Sequential([
    layers.GRU(units, dropout=0.5, return_sequences=True),
    layers.GRU(units, dropout=0.5)
])
```

11.12 Pre-trained word vectors

In the sentiment classification task, the Embedding layer is trained from scratch. In fact, for text processing tasks, most of the domain knowledge is shared, so we can use the word vectors trained on other tasks to initialize the Embedding layer to complete the domain knowledge transfer. Start training based on the pre-trained Embedding layer, and good results can be achieved with a small number of samples.

We take the pre-trained GloVe word vector as an example to demonstrate how to use the pre-trained word vector model to improve task performance. First, download the pre-trained GloVe word vector table from the official website. We choose the file glove.6B.100d.txt with a feature length of 100, and each word is represented by a vector of length 100, which can be decompressed after downloading.



Name	Date modified	Type	Size
glove.6B.50d.txt	1/3/2018 9:04 PM	Text Document	167,335 KB
glove.6B.100d.txt	8/5/2014 6:14 AM	Text Document	338,982 KB

Figure 0.22 GloVe word vector model file

Use the Python file IO code to read the word encoding vector table and store it in the Numpy array. code show as below:

```
print('Indexing word vectors.')
embeddings_index = {} # Extract words and their vectors and save them in a
dictionary
# Word vector model file storage path
GLOVE_DIR = r'C:\Users\z390\Downloads\glove6b50dtxt'
with open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'),encoding='utf-8') as
f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
print('Found %s word vectors.' % len(embeddings_index))
```

The GloVe.6B version stores a vector table of 400,000 words in total. We only considered up to 10,000 common words. We obtained the word vectors from the GloVe model according to the number code table of the words and wrote them into the corresponding positions as below:

```
num_words = min(total_words, len(word_index))
embedding_matrix = np.zeros((num_words, embedding_len)) # Word vector table
for word, i in word_index.items():
    if i >= MAX_NUM_WORDS:
        continue # Filter out other words
    embedding_vector = embeddings_index.get(word) # Query word vector from
GloVe
```

```
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector # Write the corresponding 
location
print(applied_vec_count, embedding_matrix.shape)
```

After obtaining the vocabulary data, use the vocabulary to initialize the Embedding layer, and set the Embedding layer not to participate in gradient optimization as below:

```
        # Create Embedding layer
        self.embedding = layers.Embedding(total_words, embedding_len,
                                          input_length=max_review_len,
                                          trainable=False)# Does not participate in 
gradient updates
        self.embedding.build(input_shape=(None, max_review_len))
        # Initialize the Embedding layer using the GloVe model
        self.embedding.set_weights([embedding_matrix])# initialization
```

The other parts are consistent. We can simply compare the training results of the Embedding layer initialized by the pre-trained GloVe model with the training results of the randomly initialized Embedding layer. After training 50 Epochs, the accuracy of the pre-training model reached 84.7%, an increase of approximately 2%.

11.13 Reference

[1] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[2] J. Westhuizen and J. Lasenby, “The unreasonable effectiveness of the forget gate,” *CoRR,* abs/1804.04849, 2018.