Chapter 8 Keras Advanced API

> The problem of artificial intelligence is not only a problem of computer science, but also a problem of mathematics, cognitive science and philosophy. − François Chollet

Keras is an open source neural network computing library mainly developed by Python language. It was originally written by François Chollet. It is designed as a highly modular and extensible high-level neural network interface, so that users can quickly complete model building and training without excessive professional knowledge. The Keras library is divided into a front end and a back end. The back end generally calls the existing deep learning framework to implement the underlying operations, such as Theano, CNTK, and TensorFlow. The front end interface is a set of unified interface functions abstracted by Keras. Users can easily switch between different back-end operations through code written by Keras, with greater flexibility. It is precisely because of Keras's high abstraction and ease of use that as of 2019, according to KDnuggets, Keras market share reached 26.6%, an increase of 19.7%, second only to TensorFlow in the same deep learning framework.

There is a staggered relationship between TensorFlow and Keras that is both competitive and cooperative. Even the founder of Keras works at Google. As early as November 2015, TensorFlow was added to Keras backend support. Since 2017, most components of Keras have been integrated into the TensorFlow framework. In 2019, Keras was officially identified as the only high-level interface API for TensorFlow 2, replacing the high-level interfaces such as tf.layers included in the TensorFlow 1. In other words, now you can only use the Keras interface to complete TensorFlow layer model building and training. In TensorFlow, Keras is implemented in the tf.keras submodule.

What is the difference and connection between Keras and tf.keras? In fact, Keras can be understood as a set of high-level API protocols for building and training neural networks. Keras itself has already implemented this protocol. Installing the standard Keras library can easily call TensorFlow, CNTK and other back-ends to complete accelerated calculations. In TensorFlow, a set of Keras protocol is also implemented through tf.keras, which is deeply integrated with TensorFlow, and is only based on TensorFlow back-end operations, and supports TensorFlow more perfectly. For developers using TensorFlow, tf.keras can be understood as an ordinary submodule, which is no different from other submodules such as tf.math and tf.data. Unless otherwise specified, Keras refers to tf.keras instead of the standard Keras library in following chapters.

8.1 Common functional modules

Keras provides a series of high-level neural network related classes and functions, such as classic data set loading function, network layer class, model container, loss function class, optimizer class, and classic model class.

For classic data sets, one line of code can download, manage, and load data sets. These data

sets include Boston house price prediction data set, CIFAR picture data set, MNIST / FashionMNIST handwritten digital picture data set, and IMDB text data set. We have already introduced it in previous chapters.

8.1.1 Common network layer classes

For the common neural network layer, we can use the tensor mode of the underlying interface functions to achieve, which are generally included in the tf.nn module. For common network layers, we generally use the layer method to complete the model construction. A large number of common network layers are provided in the tf.keras.layers namespace (hereinafter using layers to refer to tf.keras.layers), such as fully connected layers, activation function layers, pooling layers, convolutional layers, and recurrent neural network layers. For these network layer classes, you only need to specify the relevant parameters of the network layer at the time of creation and use the __call__ method to complete the forward calculation. When using the __call__ method, Keras will automatically call the forward propagation logic of each layer, which is generally implemented in the call function of the class.

Taking the Softmax layer as an example, it can use the tf.nn.softmax function to complete the Softmax operation in the forward propagation, or it can build the Softmax network layer through the layers.Softmax(axis) class, where the axis parameter specifies the dimension for softmax operation . First, import the relevant sub-modules as follows:

```
import tensorflow as tf
# Do not use "import keras" which will import the standard Keras, not the
one in Tensorflow
from tensorflow import keras
from tensorflow.keras import layers # import common layer class
```

Then create a Softmax layer and use the __call__ method to complete the forward calculation:

```
In [1]:
x = tf.constant([2.,1.,0.1])  # create input tensor
layer = layers.Softmax(axis=-1)  # create Softmax layer
out = layer(x)  # forward propagation
```

After passing through the Softmax network layer, the probability distribution output is:

```
Out[1]:
<tf.Tensor: id=2, shape=(3,), dtype=float32, numpy=array([0.6590012,
0.242433 , 0.0985659], dtype=float32)>
```

Of course, we can also directly complete the calculation through the tf.nn.softmax() function as follows:

```
out = tf.nn.softmax(x)
```

8.1.2 Network container

For common networks, we need to manually call the class instance of each layer to complete

the forward propagation operation. When the network layer becomes deeper, this part of the code appears very bloated. Multiple network layers can be encapsulated into a large network model through the network container Sequential provided by Keras. Only the instance of the network model needs to be called once to complete the sequential propagation operation of the data from the first layer to the last layer.

For example, the 2-layer fully connected network with a separate activation function layer can be encapsulated as a network through the Sequential container.

```
from tensorflow.keras import layers, Sequential
network = Sequential([
    layers.Dense(3, activation=None), # Fully-connected layer without
activation function
    layers.ReLU(),# activation function layer
    layers.Dense(2, activation=None), # Fully-connected layer without
activation function
    layers.ReLU() # activation function layer
])
x = tf.random.normal([4,3])
out = network(x)
```

The Sequential container can also continue to add a new network layer through the add() method to dynamically create a network:

```
In [2]:
layers_num = 2
network = Sequential([]) # Create an empty container
for _ in range(layers_num):
    network.add(layers.Dense(3)) # add fully-connected layer
    network.add(layers.ReLU())# add activation layer
network.build(input_shape=(4, 4))
network.summary()
```

The above code can create a network structure with the number of layers specified by the layers_num parameter. When the network creation is completed, the network layer class does not create member variables such as internal weight tensors. Using the build method, you can specify the input size which will automatically create internal tensors for all layers. Through the summary() function, you can easily print out the network structure and parameters. The results are as follows:

```
Out[2]:
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              multiple                  15
_________________________________________________________________
```

```
re_lu_2 (ReLU)               multiple                  0
_________________________________________________________________
dense_3 (Dense)              multiple                  12
_________________________________________________________________
re_lu_3 (ReLU)               multiple                  0
=================================================================
Total params: 27
Trainable params: 27
Non-trainable params: 0
_________________________________________________________________
```

It can be seen that the Layer columns is the name of each layer which is maintained internally by TensorFlow and is not the same as the object name of Python. Param # column is the number of parameters of the layer. Total params counts the total number of parameters. Trainable params is the total number of parameters to be optimized. Non-trainable params is the total number of parameters that do not need to be optimized.

When we encapsulate multiple network layers through Sequential container, the parameter list of each layer will be automatically incorporated into the Sequential container. The trainable_variables and variables of the Sequential object contain the list of tensors to be optimized and tensors of all layers, for example:

```
In [3]: # print name and shape of trainable variables
for p in network.trainable_variables:
    print(p.name, p.shape)
Out[3]:
dense_2/kernel:0 (4, 3)
dense_2/bias:0 (3,)
dense_3/kernel:0 (3, 3)
dense_3/bias:0 (3,)
```

The Sequential container is one of the most commonly used classes. It is very useful for quickly building multi-layer neural networks. It should be used as much as possible to simplify the implementation of the network model.

8.2 Model configuration, training and testing

When training the network, the general process is to obtain the output value of the network through forward calculation, then calculate the network error through the loss function, and then calculate and update the gradient through automatic differentiation tool, and test the network performance occasionally. For this commonly used training logic, it can be directly implemented through high-level interfaces such as model configuration and training provided by Keras, which is concise and clear.

8.2.1 Model configuration

In Keras, there are two special classes: keras.Model and keras.layers.Layer. The Layer class is the parent class of the network layer, and it defines some common functions of the network layer, such as adding weights and managing weight lists. The Model class is the parent class of the network. In addition to the functions of the Layer class, convenient functions such as saving model, loading model, training and testing model are added. Sequential is also a subclass of Model, so it has all the functions of the Model class.

Let's introduce the model configuration and training functions of the Model class and its subclasses. Taking the network encapsulated by the Sequential container as an example, we first create a 5-layer fully connected network for MNIST handwritten digital picture recognition. The code is as follows:

```
# Create a 5-layer fully connected network
network = Sequential([layers.Dense(256, activation='relu'),
                      layers.Dense(128, activation='relu'),
                      layers.Dense(64, activation='relu'),
                      layers.Dense(32, activation='relu'),
                      layers.Dense(10)])
network.build(input_shape=(4, 28*28))
network.summary()
```

After the network is created, the normal process is to iterate over multiple Epochs in the data set, generate training data in batches, do forward propagation calculation, then calculate the error value through the loss function, and automatically calculate the gradient and update the network parameters by back propagation. Since this part of the logic is very general, the compile() and fit() functions are provided in Keras to facilitate the logic. First, specify the optimizer object, loss function type, evaluation metrics and other settings used by the network through the compile function. This step is called configuration.

```
# Import optimizer, loss function module
from tensorflow.keras import optimizers,losses
# Use Adam optimizer with learning rate of 0.01
# Use cross-entropy loss function with Softmax
network.compile(optimizer=optimizers.Adam(lr=0.01),
        loss=losses.CategoricalCrossentropy(from_logits=True),
        metrics=['accuracy'] # Set accuracy as evaluation metric
    )
```

The optimizer, loss function and other parameters specified in the compile() function are also the parameters that we need to set during our own training. Keras implements this part of the common logic internally to improve development efficiency.

8.2.2 Model training

After the model is configured, the data sets for training and validation can be sent through the fit() function. This step is called model training.

```
# Training dataset is train_db, and validation dataset is val_db
# Train 5 epochs and validate every 2 epoch
# Training record and history is saved in history variable
history = network.fit(train_db, epochs=5, validation_data=val_db,
validation_freq=2)
```

Train_db can a tf.data.Dataset object or a Numpy array. The epochs parameter specifies the number of Epochs for training iterations. The validation_data parameter specifies the data set used for validation and the validation frequency is controlled by valdiation_freq.

The above code can achieve the network training and validation functions. The fit function will return the history of the training process data records, where history.history is a dictionary object, including the loss of the training process, evaluation metrics and other records, such as

```
In [4]: history.history # print training record
Out[4]:
{'loss': [0.31980024444262184,  # training loss
  0.1123824894875288,
  0.07620834542314212,
  0.05487803366283576,
  0.041726120284820596],  # training accuracy
'accuracy': [0.904, 0.96638334, 0.97678334, 0.9830833, 0.9870667],
 'val_loss': [0.09901347314302303, 0.09504951824009701],  # validation loss
 'val_accuracy': [0.9688, 0.9703]}  # validation accuracy
```

The operation of the fit() function represents the training process of the network, so it will consume considerable training time and return after the training is completed. The historical data generated during the training can be obtained through the return value object. It can be seen that the code implemented through the compile & fit method is very concise and efficient, which greatly reduces the development time. However, because the interface is very high-level, the flexibility is also reduced, and it is up to the user to decide whether to use it.

8.2.3 Model testing

The Model class can not only easily complete the network configuration, training and validation, but also is very convenient for prediction and testing. We will elaborate on the difference between validation and testing in the chapter of overfitting. Here, validaiton and testing can be understood as a way of model evaluation.

The Model.predict(x) method can complete the model prediction, for example:

```
# Load one batch of test dataset
x,y = next(iter(db_test))
```

```
print('predict x:', x.shape) # print the batch shape
out = network.predict(x) # prediction
print(out)
```

Where the out is the output of the network. Through the above code, the trained model can be used to predict the label information of new samples.

If you simply need to test the performance of the model, you can use Model.evaluate(db) to test all the samples on the db dataset and print out the performance indicators, for example:

```
network.evaluate(db_test)
```

8.3 Model saving and loading

After the model training is completed, the model needs to be saved to the file system to facilitate subsequent model testing and deployment. In fact, it is also a good habit to save the model state at intervals during training, which is especially important for training large-scale networks. Generally, a large-scale network requires several days or even weeks of training. Once the training process is interrupted or an accident occurs, the previous training progress will be lost. If the model state can be saved to the file system intermittently, even if an accident such as a downtime occurs, it can be recovered from the latest network state file, thereby avoiding wasting a lot of training time and computing resources. Therefore, the preservation and loading of the model is very important.

In Keras, there are three commonly used methods for saving and loading models.

8.3.1 Tensor method

The state of the network is mainly reflected in the structure of the network and tensor data within the network layer. Therefore, under the condition of having the source file of the network structure, it is the most lightweight way to directly save the network tensor parameters to the file system. Taking the MNIST handwritten digital picture recognition model as an example, the current network parameters can be saved by calling the Model.save_weights(path) method. The code is as follows:

```
network.save_weights('weights.ckpt') # Save tensor data of the model
```

The above code saves the network model to the weights.ckpt file. When needed, first we create a network object, and then call the load_weights(path) method of the network object to load the tensor value saved in the specified model file to the current network parameters, for example:

```
# Save tensor data of the model
network.save_weights('weights.ckpt')
print('saved weights.')
del network # delete network object
# Create similar network
network = Sequential([layers.Dense(256, activation='relu'),
                      layers.Dense(128, activation='relu'),
                      layers.Dense(64, activation='relu'),
```

```
                layers.Dense(32, activation='relu'),
                layers.Dense(10)])
network.compile(optimizer=optimizers.Adam(lr=0.01),
        loss=tf.losses.CategoricalCrossentropy(from_logits=True),
        metrics=['accuracy']
    )
# Load weights from file
network.load_weights('weights.ckpt')
print('loaded weights!')
```

This method of saving and loading the network is the most lightweight. The file only saves the values of the tensor parameters, and there are no other additional structural parameters. But it needs to use the same network structure to be able to restore the network state correctly, so it is generally used in the case of having network source files.

8.3.2 Network method

Let's introduce a method that does not require network source files, and only needs model parameter files to recover the network model. The model structure and model parameters can be saved to the path file through the Model.save(path) function, and the network structure and network parameters can be restored through keras.models.load_model(path) without the need for network source files .

First save the MNIST handwritten digital picture recognition model to a file, and delete the network object:

```
# Save model and parameters to a file
network.save('model.h5')
print('saved total model.')
del network # Delete the network
```

The structure and state of the network can be recovered through the model.h5 file, and there is no need to create network objects in advance. The code is as follows:

```
# Recover the model and parameters from a file
network = keras.models.load_model('model.h5')
```

As you can see, in addition to storing model parameters, the model.h5 file should also save network structure information. You can directly recover the network object from the file without creating a model in advance.

8.3.3 SavedModel method

TensorFlow is favored by the industry, not only because the excellent neural network layer API support, but also because it has powerful ecosystem, including mobile and web support. When the model needs to be deployed to other platforms, the SavedModel method proposed by TensorFlow is platform-independent.

By tf.saved_model.save(network, path), the model can be saved to the path directory as

follows:

```
#  Save model and parameters to a file
tf.saved_model.save(network, 'model-savedmodel')
print('saving savedmodel.')
del network # Delete network object
```

The following network files appear in the file system model-savedmodel directory, as shown in Figure 8.1:

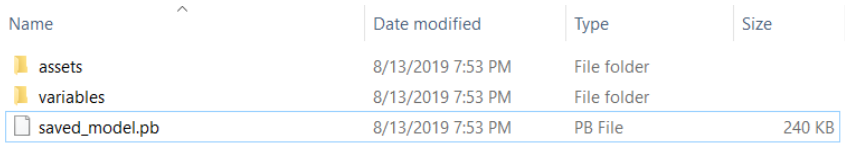

图 8.1 SavedModel method directory

Users don't need to care about the file saving format, they only need to restore the model object through the tf.saved_model.load function. After recovering the model instance, we complete the calculation of the test accuracy rate and achieve the following:

```
print('load savedmodel from file.')
# Recover network and parameter from files
network =  tf.saved_model.load('model-savedmodel')
# Accuracy metrics
acc_meter = metrics.CategoricalAccuracy()
for x,y in ds_val:   # Loop through test dataset
    pred = network(x) # Forward calculation
    acc_meter.update_state(y_true=y, y_pred=pred) # Update stats
# Print accuracy
print("Test Accuracy:%f" % acc_meter.result())
```

8.4 Custom network

Although Keras provides many common network layer classes, the network used for deep learning are far more than that. Researchers generally implement relatively new network layers on their own. Therefore, it is very important to master the custom network layer and the realization of the network.

For the network layer that needs to create customized logic, it can be implemented through a custom class. When creating a customized network layer class, you need to inherit from the layers.Layer base class. When creating a custom network class, you need to inherit from the keras.Model base class, so the custom class created in this way can easily use the Layer/Model base class. The parameter management and other functions provided by the class can also be used interactively with other standard network layer classes.

8.4.1 Custom network layer

For a custom network layer, we at least need to implement the initialization(__init__) method

and the forward propagation logic. Let's take a specific custom network layer as an example, assuming that a fully connected layer without bias vectors is needed, that is, bias is 0, and the fixed activation function is ReLU. Although this can be created through the standard Dense layer, we still explain how to implement a custom network layer by implementing this "special" network layer class.

First create a class and inherit from the base Layer class. Create an initialization method and call the initialization function of the parent class. Because it is a fully connected layer, two parameters need to be set: the length of the input feature inp_dim and the length of the output feature outp_dim, and the shape size is created by self.add_variable(name, shape). The name tensor $\boldsymbol{W}$ is set to be optimized.

```
class MyDense(layers.Layer):
    # Custom layer
    def __init__(self, inp_dim, outp_dim):
        super(MyDense, self).__init__()
        # Create weight tensor and set to be trainable
        self.kernel = self.add_variable('w', [inp_dim, outp_dim],
trainable=True)
```

需要注意的是，self.add_variable 会返回张量$\boldsymbol{W}$的 Python 引用，而变量名 name 由 TensorFlow 内部维护，使用的比较少。我们实例化 MyDense 类，并查看其参数列表，例如：

It should be noted that self.add_variable will return a Python reference to the tensor $\boldsymbol{W}$, and the variable name is maintained internally by TensorFlow and is used less often. We instantiate the MyDense class and view its parameter list, for example:

```
In [5]: net = MyDense(4,3) # Input dimension is 4 and output dimension is 3.
net.variables,net.trainable_variables  # Check the trainable parameters
Out[5]:
# All parameters
([<tf.Variable 'w:0' shape=(4, 3) dtype=float32, numpy=…
# Trainable parameters
 [<tf.Variable 'w:0' shape=(4, 3) dtype=float32, numpy=…
```

You can see that the tensor $\boldsymbol{W}$ is automatically included in the parameter list.

By modifying to self.kernel = self.add_variable('w', [inp_dim, outp_dim], trainable = False), we can set the tensor $\boldsymbol{W}$ not to be trainable and then observe the management state of the tensor:

```
([<tf.Variable 'w:0' shape=(4, 3) dtype=float32, numpy=…], # All parameters
[])# Trainable parameters
```

As you can see, the tensor is not managed by trainable_variables at this time. In addition, class member variables created as tf.Variable in class initialization are also automatically included in tensor management, for example:

```
        self.kernel = tf.Variable(tf.random.normal([inp_dim, outp_dim]),
                                  trainable=False)
```

The list of managed tensors is printed out as follows:

```
# All parameters
([<tf.Variable 'Variable:0' shape=(4, 3) dtype=float32, numpy=…],
[])# Trainable paramters
```

After the initialization of the custom class, we will design the forward calculation logic. For this example, only the matrix operation $\boldsymbol{O} = \boldsymbol{X}@\boldsymbol{W}$ needs to be completed and the fixed ReLU activation function can be used. The code is as follows:

```
    def call(self, inputs, training=None):
        # Forward calculation
        # X@W
        out = inputs @ self.kernel
        # Run activation function
        out = tf.nn.relu(out)
        return out
```

As shown above, the forward calculation logic is implemented in the call(inputs, training = None) function, where inputs parameter represents input and is passed in by the user. The training parameter is used to specify the state of the model: True means training mode and False indicates testing mode, and default value is None, which is the test mode. Since the training and test modes of the fully connected layer are logically consistent, no additional processing is required here. For the network layer whose test and training modes are inconsistent, the logic to be executed needs to be designed according to the training parameters.

8.4.2 Custom network

After completing the custom fully connected layer class implementation, we created the MNIST handwritten digital picture model based on the "unbiased fully connected layer" described above.

The custom network class can be easily encapsulated into a network model through the Sequential container like other standard classes:

```
network = Sequential([MyDense(784, 256), # Use custom layer
                      MyDense(256, 128),
                      MyDense(128, 64),
                      MyDense(64, 32),
                      MyDense(32, 10)])
network.build(input_shape=(None, 28*28))
network.summary()
```

It can be seen that by stacking our custom network layer classes, a 5-layer fully connected layer network can also be realized. Each layer of the fully connected layer has no bias tensor, and the activation function uses the ReLU function.

The Sequential container is suitable for a network model in which data propagates in order from the first layer to the second layer, and then from the second layer to the third layer, and

propagates in this manner. For complex network structures, for example, the input of the third layer is not only the output of the second layer, but also the output of the first layer. At this time, it is more flexible to use a custom network. Let's create a custom network class. First create a class, and inherit from the Model base class, and then respectively create the corresponding network layer object as follows:

```
class MyModel(keras.Model):
    # Custom network class
    def __init__(self):
        super(MyModel, self).__init__()
        # Create the network
        self.fc1 = MyDense(28*28, 256)
        self.fc2 = MyDense(256, 128)
        self.fc3 = MyDense(128, 64)
        self.fc4 = MyDense(64, 32)
        self.fc5 = MyDense(32, 10)
```

Then implement the forward operation logic of the custom network as follows:

```
    def call(self, inputs, training=None):
        # Forward calculation
        x = self.fc1(inputs)
        x = self.fc2(x)
        x = self.fc3(x)
        x = self.fc4(x)
        x = self.fc5(x)
        return x
```

This example can be implemented directly using the Sequential container method. But the forward calculation logic of the custom network can be freely defined and more general. We will see the superiority of the custom network in the chapter of convolutional neural networks.

8.5 Model zoo

For commonly used network models, such as ResNet and VGG, you do not need to manually create them. They can be implemented directly with the keras.applications submodule with a line of code. At the same time, you can also load pre-trained models by setting the weights parameters.

8.5.1 Load model

Taking the ResNet50 network model as an example, the network after removing the last layer of ResNet50 is generally used as the feature extraction subnetwork for the new task, that is, using the pre-trained network parameters on the ImageNet data set to initialize, and appending one fully connected layer corresponding to the number of data categories according to the category of the custom task, so that new tasks can be learned quickly and efficiently on the basis of the pre-trained network.

First, use the Keras model zoo to load the pre-trained ResNet50 network by ImageNet. The code is as follows:

```
# Load ImageNet pre-trained network. Exclude the last layer.
resnet = keras.applications.ResNet50(weights='imagenet',include_top=False)
resnet.summary()
# test the output
x = tf.random.normal([4,224,224,3])
out = resnet(x) # get output
out.shape
```

The above code automatically downloads the model structure and pre-trained network parameters on the ImageNet dataset from the server. By setting the include_top parameter to False, we choose to remove the last layer of ResNet50. The size of the output feature map of the network is $[b, 7, 7, 2048]$. For a specific task, we need to set a custom number of output nodes. Taking 100 classification tasks as an example, we rebuild a new network based on ResNet50. Create a new pooling layer (the pooling layer here can be understood as a function of downsampling in high and wide dimensions), and reduce the features dimension from $[b, 7, 7, 2048]$ to $[b, 2048]$ as bellow.

```
In [6]:
# New pooling layer
global_average_layer = layers.GlobalAveragePooling2D()
# Use last layer's output as this layer's input
x = tf.random.normal([4,7,7,2048])
# Use pooling layer to reduce dimension from [4,7,7,2048] to
[4,1,1,2048],and squeeze to [4,2048]
out = global_average_layer(x)
print(out.shape)
Out[6]: (4, 2048)
```

Finally, create a new fully connected layer and set the number of output nodes to 100. The code is as follows:

```
In [7]:
# New fully connected layer
fc = layers.Dense(100)
# Use last layer's output as this layer's input
x = tf.random.normal([4,2048])
out = fc(x)
print(out.shape)
Out[7]: (4, 100)
```

After creating a pre-trained ResNet50 feature sub-network, a new pooling layer and a fully connected layer, we re-use the Sequential container to encapsulate a new network:

```
# Build a new network using previous layers
mynet = Sequential([resnet, global_average_layer, fc])
mynet.summary()
```

You can see the structure information of the new network model is:

```
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Model)             (None, None, None, 2048)  23587712
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 100)               204900
=================================================================
Total params: 23,792,612
Trainable params: 23,739,492
Non-trainable params: 53,120
```

By setting resnet.trainable = False, you can choose to freeze the network parameters of the ResNet part and only train the newly created network layer, so that the network model training can be completed quickly and efficiently. Of course, you can also update all the parameters of the network on a custom task.

8.6 Metrics

In the training process of the network, metrics such as accuracy and recall rate are often required. Keras provides some commonly used metrics in the keras.metrics module.

There are four main steps in the use of Keras metrics: creating a new metrics container, writing data, reading statistical data and clearing the measuring container.

8.6.1 Create a metrics container

In the keras.metrics module, it provides many commonly used metrics classes, such as Mean, Accuracy, and CosineSimilarity. Below we take the statistical error as an example. In the forward operation, we will get the average error of each batch, but we want to count the average error of each step, so we choose to use the Mean metrics. Create a new average measuring metric as follows:

```
loss_meter = metrics.Mean()
```

8.6.2 Write data

New data can be written through the update_state function, and the metric will record and process the sampled data according to its own logic. For example, the loss value is collected once at the end of each Step:

```
    # Record the sampled data, and convert the tensor to an ordinary value through the float() function
    loss_meter.update_state(float(loss))
```

After the above sampling code is placed at the end of each Batch operation, the meter will automatically calculate the average value based on the sampled data.

8.6.3 Read statistical data

After sampling multiple times of data, you can choose to call the measurer's result() function to obtain statistical values. For example, the interval statistical loss average is as follows:

```
# Print the average loss during the statistical period
print(step, 'loss:', loss_meter.result())
```

8.6.4 Clear the container

Since the metric container will record all historical data, it is necessary to clear the historical status when starting a new round of statistics. It can be realized by reset_states() function. For example, after reading the average error every time, clear the statistical information to start the next round of statistics as follows:

```
if step % 100 == 0:
    # Print the average loss
    print(step, 'loss:', loss_meter.result())
    loss_meter.reset_states() # reset the state
```

8.6.5 Hands-on accuracy metric

According to the method of using the metric tool, we use the Accuracy metric to count the accuracy rate during the training process. First create a new accuracy measuring container as follows:

```
acc_meter = metrics.Accuracy()
```

After each forward calculation is completed, record the training accuracy rate. It should be noted that the parameters of the update_state function of the Accuracy class are the predicted value and the true value, not the accuracy rate of the current batch. We write the label and prediction result of the current Batch sample into the metric as follows:

```
# [b, 784] => [b, 10, network output
out = network(x)
# [b, 10] => [b], feed into argmax()
pred = tf.argmax(out, axis=1)
pred = tf.cast(pred, dtype=tf.int32)
# record the accuracy
acc_meter.update_state(y, pred)
```

After counting the predicted values of all batches in the test set, print the average accuracy of the statistics and clear the metric container. The code is as follows:

```
print(step, 'Evaluate Acc:', acc_meter.result().numpy())
acc_meter.reset_states() # reset metric
```

8.7 Visualization

In the process of network training, it is very important to improve the development efficiency and monitor the training progress of the network through the Web terminal and visualize the training results. TensorFlow provides a special visualization tool called TensorBoard, which writes monitoring data to the file system through TensorFlow and uses the web backend to monitor the corresponding file directory, thus allowing users to view network monitoring data.

The use of TensorBoard requires cooperation between the model code and the browser. Before using TensorBoard, you need to install the TensorBoard library. The installation command is as follows:

```
# Install TensorBoard
pip install tensorboard
```

Next, we introduce how to use the TensorBoard tool to monitor the progress of network training in the model side and the browser side.

8.7.1 Model side

On the model side, you need to create a Summary class that writes monitoring data when needed. First create an instance of the monitoring object class through tf.summary.create_file_writer, and specify the directory where the monitoring data is written. The code is as follows:

```
# Create a monitoring class, the monitoring data will be written to the
log_dir directory
summary_writer = tf.summary.create_file_writer(log_dir)
```

We take monitoring error and visual image data as examples to introduce how to write monitoring data. After the forward calculation is completed, for the scalar data such as error, we record the monitoring data through the tf.summary.scalar function and specify the time stamp step parameter. The step parameter here is similar to the time scale information corresponding to each data, and can also be understood as the coordinates of the data curve, so it should not be repeated. Each type of data is distinguished by the name of the string, and similar data needs to be written to the database with the same name. E.g:

```
with summary_writer.as_default():
    # write the current loss to train-loss database
    tf.summary.scalar('train-loss', float(loss), step=step)
```

TensorBoard distinguishes different types of monitoring data by string ID, so for error data, we named it "train-loss", other types of data can not be written to prevent data pollution.

For picture-type data, you can write monitoring picture data through the tf.summary.image function. For example, during training, the sample image can be visualized by the tf.summary.image function. Since the tensor in TensorFlow generally contains multiple samples, the tf.summary.image function accepts tensor data of multiple pictures and sets the max_outputs parameter to select the maximum number of displayed pictures. The code is as follows:

```
with summary_writer.as_default():
```

```
            # log accuracy
            tf.summary.scalar('test-acc', float(total_correct/total),
step=step)
            # log images
            tf.summary.image("val-onebyone-images:", val_images,
max_outputs=9, step=step)
```

Run the model program, and the corresponding data will be written to the specified file directory in real time.

8.7.2 Browser side

When running the program, the monitoring data is written to the specified file directory. If you want to remotely view and visualize these data in real time, you also need to use a browser and a web backend. The first step is to open the web backend. Run command "tensorboard --logdir path" in terminal and specify the file directory path monitored by the web backend, then you can open the web backend monitoring process, as shown in Figure 8.2:

```
F:\深度学习与TensorFlow入门实战\素材\lesson28-可视化\logs>tensorboard --logdir .
c:\conda\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the second argument
t` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float
  from ._conv import register_converters as _register_converters
TensorBoard 1.14.0a20190603 at http://DESKTOP-C6H6KQF:6006/ (Press CTRL+C to quit)
```

Figure 8.2 Open Web Server

Open a browser and enter the URL http://localhost: 6006 (you can also remotely access through the IP address, the specific port number may change depending on the command prompt) to monitor the progress of the network training. TensorBoard can display multiple monitoring records at the same time. On the left side of the monitoring page, you can select monitoring records, as shown in Figure 8.3:

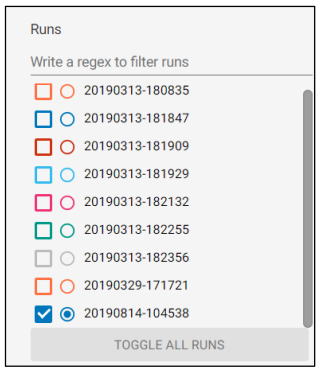

Figure 8.3 Snapshot of Tensorboard

On the upper end of the monitoring page, you can choose different types of data monitoring pages, such as scalar monitoring page SCALARS, picture visualization page IMAGES, etc. For this example, we need to monitor the training error and test accuracy rate for scalar data, and its curve can be viewed on the SCALARS page, as shown in Figure 8.4 and Figure 8.5.

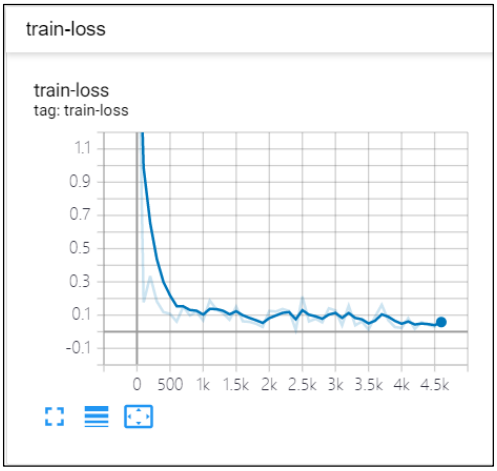

Figure 8.4 Training loss curve

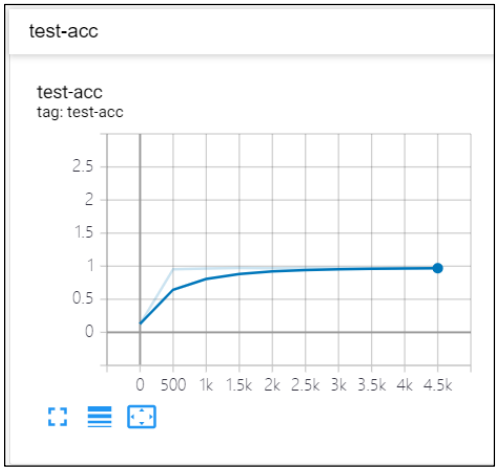

Figure 8.5 Training accuracy curve

On the IMAGES page, you can view images at each Step as shown in Figure 8.6.

Figure 8.6 Pictures from each step

In addition to monitoring scalar data and image data, TensorBoard also supports functions such as viewing histogram distribution of tensor data through tf.summary.histogram, and printing text information through tf.summary.text. E.g:

```
with summary_writer.as_default():
    tf.summary.scalar('train-loss', float(loss), step=step)
    tf.summary.histogram('y-hist',y, step=step)
    tf.summary.text('loss-text',str(float(loss)))
```

You can view the histogram of the tensor on the HISTOGRAMS page, as shown in Figure 8.7, and you can view the text information on the TEXT page, as shown in Figure 8.8.

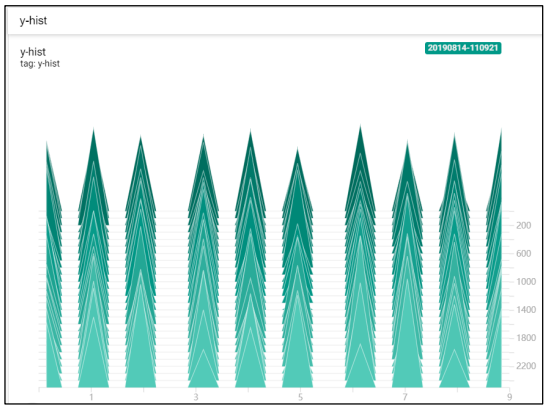

Figure 8.7 Tensorboard histogram

Figure 8.8 Tensorboard text visualization

In fact, in addition to TensorBoard, the Visdom tool developed by Facebook can also facilitate the visualization of data, and supports a variety of visualization methods in real-time, and is more convenient to use. Figure 8.9 shows the visualization of Visdom data. Visdom can directly accept PyTorch's tensor-type data, but cannot directly accept TensorFlow's tensor-type data. It needs to be converted to a Numpy array. For readers pursuing rich visualization methods and real-time monitoring, Visdom may be a better choice.

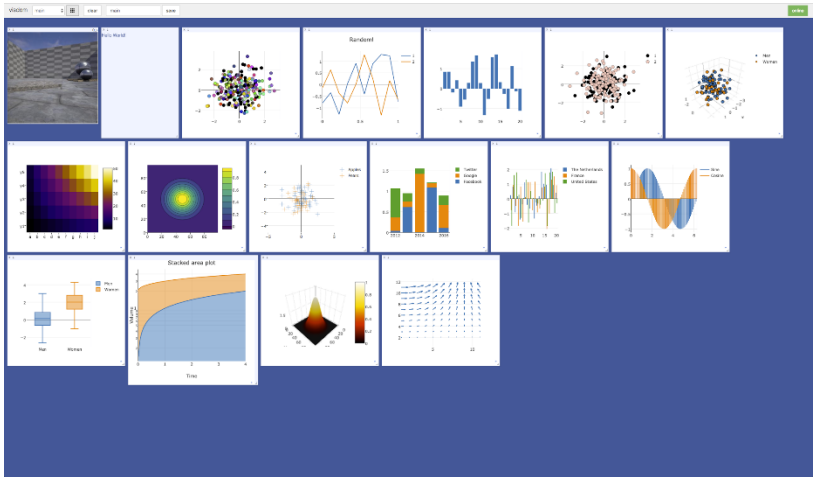

Figure 8.9 Visdom snapshot[①]

[①] Image source: https://github.com/facebookresearch/visdom