KERAS

INTRODUCTION:

- Keras is an open-source neural network computing library mainly developed in the 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 frontend and a backend. The backend generally calls the existing deep learning framework to implement the underlying operations, such as Theano, CNTK, and TensorFlow.
- The frontend interface is a set of unified interface functions abstracted by Keras. Users can easily switch between different backend operations through Keras.
- 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 2, Keras is implemented in the tf.keras submodule.

COMMON FUNCTIONAL MODULES:

- Keras is a popular deep learning framework that provides a high-level API for building and training neural networks. It offers a variety of common functional modules that you can use to construct neural network architectures.
- Keras provides a series of high-level neural network-related classes and functions, such as classic dataset loading function, network layer class, model container, loss function class, optimizer class, and classic model class.

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, 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.

Examples:

Dense Layer: The Dense layer is a fully connected layer where every neuron in the layer is connected to every neuron in the previous layer. It's commonly used in feed forward neural networks and can be used for tasks like image classification and regression.

```
from keras.layers import Dense
dense_layer = Dense(units=64, activation='relu')
```

Convolutional Layer: The Conv2D and Conv3D layers are used for 2D and 3D convolution operations, respectively. They are essential for tasks like image and video analysis.

```
from keras.layers import Conv2D
conv_layer = Conv2D(filters=32, kernel_size=(3, 3), activation='relu')
```

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 two-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:

```
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()
```

When we encapsulate multiple network layers through Sequential container, the parameter list of each layer will be automatically incorporated into the Sequential container.

MODEL CONFIGURATION, TRAINING AND TESTING IN KERAS

In Keras, you can create, configure, train, and test a neural network model by following a series of steps. Below is an overview of these steps:

1. Model Creation and Configuration:

Import necessary libraries and modules from Keras.

Define the model architecture, including layers, activation functions, and input/output shapes.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=feature_dim))
model.add(Dense(units=num_classes, activation='softmax'))
```

2. **Compile the Model:**

After defining the model, you need to configure its learning process by specifying the optimizer, loss function, and evaluation metrics.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

3. **Data Preparation:**

Prepare your training and testing data by loading, preprocessing, and splitting it into input and target (label) datasets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2,
random_state=42)
```

4. **Model Training:**

Train the model using the fit method. Pass in your training data (features and labels), batch size, number of epochs, and validation data if necessary.

```
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_test, y_test)).
```

5. **Model Evaluation:**

After training, you can evaluate the model's performance on the test dataset using the evaluate method.

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")
```

6. **Making Predictions:**

   Use the trained model to make predictions on new or unseen data.

   ```
   predictions = model.predict(X_new_data)
   ```

7. **Save and Load Models (Optional):**

   You can save your trained model to disk for later use and load it when needed.

   ```
   model.save("my_model.h5") loaded_model =
   keras.models.load_model("my_model.h5")
   ```

8. **Visualization (Optional):**

   You can visualize training history and performance using libraries like Matplotlib.

   ```
   import matplotlib.pyplot as plt plt.plot(history.history['loss'], label='Training Loss')
   plt.plot(history.history['val_loss'], label='Validation Loss') plt.legend() plt.show()
   ```

<u>MODEL SAVING AND LOADING</u>

1. **TENSOR METHOD:**

   In TensorFlow, you can save and load models using the tf.saved_model method. This method provides a standard way to save and load models, making it compatible with TensorFlow Serving and other TensorFlow-based deployment environments. Here's how you can save and load a model using the tf.saved_model method:

Saving a Model:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a simple Sequential model
model = Sequential([
    Dense(units=64, activation='relu', input_dim=feature_dim),
    Dense(units=num_classes, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_test, y_test))

# Save the model using the tf.saved_model method
model.save('my_model')
 # 'my_model' is the directory where the model will be saved
```

Loading a Model:

```
import tensorflow as tf

# Load the model using the tf.saved_model.load method
loaded_model = tf.saved_model.load('my_model')

# Make predictions using the loaded model
predictions = loaded_model(X_new_data)
 # X_new_data is your new data for inference.
```

In this example:

We first create a simple Sequential model and train it.

We save the model using model.save('my_model'), which will create a directory named 'my_model' containing the saved model artifacts.

To load the model, we use tf.saved_model.load('my_model'). This will load the model and return a callable object that you can use for making predictions.

Note that the model will be saved as a TensorFlow SavedModel, which is a directory containing the model's architecture, weights, and other metadata. This format is designed for easy deployment and compatibility with TensorFlow Serving.

Make sure to replace 'my_model' with the path where you want to save or load your model. You can also specify different versions of the model by using different directory names when saving.

2. USING 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')
```

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.

3. SAVED MODEL METHOD:

TensorFlow is favored by the industry, not only because of 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
```

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())
```

CUSTOM NETWORK LAYERS:

In Keras, you can create custom neural network architectures by subclassing the keras.Model class. This approach allows you to define your own forward pass logic and create highly customized network architectures.

Here's how to create a custom neural network in Keras:

#Import the necessary libraries:

```
import tensorflow as tf
from tensorflow.keras.layers import Layer
```

Define a custom layer or set of layers. You can do this by creating a class that inherits from keras.layers.Layer. Implement the **__init__ method** to define layer parameters and the call method to specify the layer's forward pass.

For example, here's how you can define a custom layer:

```
class CustomLayer(Layer):
    def __init__(self, num_units, activation='relu'):
        super(CustomLayer, self).__init__()
        self.num_units = num_units
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        # Define layer variables and weights here
        self.kernel = self.add_weight("kernel", shape=[input_shape[-1], self.num_units])
        self.bias = self.add_weight("bias", shape=[self.num_units])

    def call(self, inputs):
        # Define the layer's forward pass logic
        return self.activation(tf.matmul(inputs, self.kernel) + self.bias)
```

Create a custom model by subclassing keras.Model and define the architecture by composing custom layers.

```
class CustomModel(tf.keras.Model):
    def __init__(self):
        super(CustomModel, self).__init__()
        self.layer1 = CustomLayer(num_units=64, activation='relu')
        self.layer2 = CustomLayer(num_units=10, activation='softmax')

    def call(self, inputs):
        x = self.layer1(inputs)
        return self.layer2(x)
```

Compile the custom model with an optimizer, loss function, and metrics.

```
model = CustomModel()
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Train the model using your custom data and fit it to your training data:

model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_val, y_val))

Evaluate and use the custom model just like any other Keras model:

loss, accuracy = model.evaluate(X_test, y_test)
predictions = model.predict(X_new_data)

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.

For an example, 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',inclu
de_top=False)
resnet.summary()
# test the output
x = tf.random.normal([4,224,224,3])
out = resnet(x) # get output
out.shape
```

For a specific task, we need to set a custom number 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 in the following.

```
# 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 Number
=================================================================
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.

METRICS:

In Keras, metrics are used to evaluate the performance of machine learning or deep learning models. Metrics provide quantitative measures that help assess how well a model is performing on a given task, such as classification or regression. Keras provides a wide range of built-in metrics that you can use during model training and evaluation. Here are some common metrics available in Keras:

- Accuracy ('accuracy'): Measures the proportion of correctly classified samples in classification tasks. It's commonly used for classification problems with balanced classes.

- Precision ('precision'): Calculates the ratio of true positives to the sum of true positives and false positives. It is used to assess how many of the positively predicted instances were actually positive. Useful in situations where false positives are costly.

- Recall ('recall') or Sensitivity ('sensitivity') or True Positive Rate ('tp_rate'): Measures the ratio of true positives to the sum of true positives and false negatives. It quantifies how many of the actual positive instances were correctly predicted as positive. Important when missing positive cases is costly.

- F1-Score ('f1'): Harmonic mean of precision and recall. Useful when there is an imbalance between classes and you want to balance precision and recall.

- AUC-ROC ('roc_auc'): Area under the Receiver Operating Characteristic (ROC) curve. It's used to evaluate binary classification models and measures the model's ability to distinguish between positive and negative samples.

- Mean Absolute Error ('mae'): Measures the average absolute difference between predicted and actual values in regression tasks. It is robust to outliers.

- Mean Squared Error ('mse'): Measures the average squared difference between predicted and actual values in regression tasks. It penalizes larger errors more than MAE.

- Root Mean Squared Error ('rmse'): The square root of the MSE, which provides an error metric in the same units as the target variable.

- Mean Absolute Percentage Error ('mape'): Measures the average percentage difference between predicted and actual values in regression tasks. Useful when you want to assess the percentage error.

- Cosine Proximity ('cosine_proximity'): Measures the cosine similarity between predicted and actual values. Often used in recommendation systems.

- Cohen's Kappa ('kappa'): Measures the agreement between predicted and actual values while accounting for chance agreement. Useful for classification tasks when the class distribution is imbalanced.

- Top-K Categorical Accuracy ('top_k_categorical_accuracy'): Measures the accuracy of the top-k predictions in multi-class classification. It's often used when you care about whether the correct class is within the top-k predictions.

You can choose one or more of these metrics when compiling your Keras model using the model.compile() method.

CREATE A METRIC CONTAINER, WRITE DATA, READ STATISTICAL DATA, CLEAR THE CONTAINER, ACCURACY METRIC

In Keras, you can create a custom metric container, write data to it, read statistical data from it, clear the container, and implement an example metric like accuracy. Here's how you can do that:

Create a Metric Container:
You can create a custom metric container by subclassing keras.metrics.Metric and implementing the necessary methods.

```
import tensorflow as tf
from tensorflow.keras.metrics import Metric

class CustomMetric(Metric):
    def __init__(self, name='custom_metric', **kwargs):
        super(CustomMetric, self).__init__(name=name, **kwargs)
        self.values = self.add_weight(name='values', initializer='zeros')
        self.count = self.add_weight(name='count', initializer='zeros')
    def update_state(self, y_true, y_pred, sample_weight=None):
        # Custom metric logic: Update the values and count
        values = ...  # Calculate your custom metric value here
        self.values.assign_add(tf.reduce_sum(values))
        self.count.assign_add(tf.cast(tf.shape(y_true)[0], dtype=tf.float32))

    def result(self):
        # Calculate and return the final metric value
        return self.values / self.count
```

Write Data to the Metric Container:
To use your custom metric, you can instantiate it and then update its state by calling update_state() during model training or evaluation.

```
custom_metric = CustomMetric()

# During training or evaluation loop:
for batch in dataset:
    predictions = model(batch['input'])
    custom_metric.update_state(batch['target'], predictions)
```

Read Statistical Data:
You can read the statistical data from your custom metric by calling the result() method.

```
metric_value = custom_metric.result().numpy()
print(f'Custom Metric Value: {metric_value:.4f}')
```

Clear the Metric Container:
You can clear the metric container's state by calling the reset_states() method.

```
custom_metric.reset_states()
```

Accuracy Metric Example:
Here's how you can implement the accuracy metric as an example:

```
class AccuracyMetric(Metric):
    def __init__(self, name='accuracy', **kwargs):
        super(AccuracyMetric, self).__init__(name=name, **kwargs)
        self.correct_predictions = self.add_weight(name='correct_predictions',
initializer='zeros')
        self.total_samples = self.add_weight(name='total_samples', initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Calculate the number of correct predictions
        correct = tf.cast(tf.equal(tf.argmax(y_true, axis=-1), tf.argmax(y_pred, axis=-1)),
dtype=tf.float32)
        self.correct_predictions.assign_add(tf.reduce_sum(correct))
        self.total_samples.assign_add(tf.cast(tf.shape(y_true)[0], dtype=tf.float32))

    def result(self):
        # Calculate and return accuracy
    return self.correct_predictions / self.total_samples
```

VISUALISATION IN KERAS : MODEL SIDE AND BROWSER SIDE:

Visualizing the performance and architecture of a Keras model can be done on both the model side (inside your Python script) and the browser side (using tools like TensorBoard). Here's how you can perform visualization in both contexts:

Model Side Visualization:

Plot Training History:

You can plot the training history of your Keras model directly within your Python script using libraries like Matplotlib. This allows you to visualize metrics like loss and accuracy during training.

```
import matplotlib.pyplot as plt
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_val, y_val))

# Plot training history
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.show()
```

Model Summary:

You can print a summary of your model's architecture to the console to understand its structure and the number of trainable parameters.

```
model.summary()
```

Browser Side Visualization (TensorBoard):

TensorBoard is a powerful tool provided by TensorFlow that allows you to visualize various aspects of your model, such as training metrics, model architecture, and even custom metrics. Here's how to use TensorBoard for visualization:

Install TensorFlow (if not already installed):

Make sure you have TensorFlow installed. You can install it using pip:

```
pip install tensorflow
```

Logging Metrics to TensorBoard: During model training, you can log metrics to TensorBoard using a Keras callback.
Here's an example:

```
from tensorflow.keras.callbacks import TensorBoard
tensorboard_callback = TensorBoard(log_dir='./logs', histogram_freq=1)
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_val, y_val), callbacks=[tensorboard_callback])
```

Start TensorBoard:

In your terminal, navigate to the directory where you're running your Python script and use the following command to start TensorBoard:

```
tensorboard --logdir=./logs
```

This command will start TensorBoard and provide a URL you can open in your browser to access the visualization.

Access TensorBoard in Your Browser:

Open a web browser and go to the URL provided by TensorBoard (typically, it's http://localhost:6006). Here, you can view various visualizations, including training metrics, model architecture, and more.

By combining model-side visualization in your Python script with browser-side visualization using TensorBoard, you can gain a comprehensive understanding of your Keras model's performance and architecture during training and evaluation.