Course: Artificial Intelligence and Machine Learning Code:20CS51I

WEEK - 10: Keras

- **What is Keras?**
- **Keras API – Three Programming models.**
  - **Sequential Model**
  - **Functional API**
  - **Model Subclassing**
- **Keras Layers**
- **Custom Keras Layers**

Session No. 4

What is Keras:

- Keras is an open-source high-level Neural Network library, which is written in Python is capable enough to run on Theano, TensorFlow, or CNTK
- It was developed by one of the Google engineers, Francois Chollet. It is made user-friendly, extensible, and modular for facilitating faster experimentation with deep neural networks. It not only supports Convolutional Networks and Recurrent Networks individually but also their combination.
- It cannot handle low-level computations, so it makes use of the Backend library to resolve it.
- The backend library act as a high-level API wrapper for the low-level API, which lets it run on TensorFlow, CNTK, or Theano

What makes Keras special?

- Large adoption in the industry.
- Focus on user experience is the major part of Keras.
- It is a multi-backend and supports multi-platform, which helps all the encoders come together for coding.
- Research community present for Keras works amazingly with the production community.
- Easy to grasp all concepts.
- It supports fast prototyping.

- It seamlessly runs on CPU as well as GPU.
- It provides the freedom to design any architecture, which then later is utilized as an API for the project.
- It is really very simple to get started with.
- Easy production of models actually makes Keras special.

Keras user experience:

1. **Keras is an API designed for humans**

   Best practices are followed by Keras to decrease cognitive load, ensures that the models are consistent, and the corresponding APIs are simple.

2. **Not designed for machines**

   Keras provides clear feedback upon the occurrence of any error that minimizes the number of user actions for the majority of the common use cases.

3. **Easy to learn and use.**

4. **Highly Flexible**

   Keras provide high flexibility to all of its developers by integrating low-level deep learning languages such as TensorFlow or Theano, which ensures that anything written in the base language can be implemented in Keras.

Keras API – Three Programming models:

Sequential model:

The core idea of *Sequential API* is simply arranging the Keras layers in a sequential order and so, it is called *Sequential API*. Most of the ANN also has layers in sequential order and the data flows from one layer to another layer in the given order until the data finally reaches the output layer.

A ANN model can be created by simply calling **Sequential()** API as specified below −

```
from keras.models import Sequential
model = Sequential()
```

Add layers

To add a layer, simply create a layer using Keras layer API and then pass the layer through add()function as specified below −

```
from keras.models import Sequential

model = Sequential()
input_layer = Dense(32, input_shape=(8,)) model.add(input_layer)
hidden_layer = Dense(64, activation='relu'); model.add(hidden_layer)
output_layer = Dense(8)
```

Here, we have created one input layer, one hidden layer and one output layer.

Access the model:

Keras provides few methods to get the model information like layers, input data and output data. Theyare as follows −

- ***model.layers*** − Returns all the layers of the model as list.

```
>>> layers = model.layers
>>> layers
[
  <keras.layers.core.Dense object at 0x000002C8C888B8D0>,
  <keras.layers.core.Dense object at 0x000002C8C888B7B8>
  <keras.layers.core.Dense object at 0x 000002C8C888B898>
]
```

- ***model.inputs*** − Returns all the input tensors of the model as list.

```
>>> inputs = model.inputs
>>> inputs
[<tf.Tensor 'dense_13_input:0' shape=(?, 8) dtype=float32>]
```

- ***model.outputs*** − Returns all the output tensors of the model as list.

```
>>> outputs = model.outputs
>>> outputs
<tf.Tensor 'dense_15/BiasAdd:0' shape=(?, 8) dtype=float32>]
```

- ***model.get_weights*** − Returns all the weights as NumPy arrays.
- ***model.set_weights(weight_numpy_array)*** − Set the weights of the model.

Functional API

Sequential API is used to create models layer-by-layer. Functional API is an alternative approach of creating more complex models. Functional model, you can define multiple input or output that share layers. First, we create an instance for model and connecting to the layers to access input and output to the model. This section explains about functional model in brief.

Create a model

Import an input layer using the below module −

```
>>> from keras.layers import Input
```

Now, create an input layer specifying input dimension shape for the model using the below code −

```
>>> data = Input(shape=(2,3))
```

Define layer for the input using the below module −

```
>>> from keras.layers import Dense
```

Add Dense layer for the input using the below line of code −

```
>>> layer = Dense(2)(data)
>>> print(layer)
```

Define model using the below module −

```
from keras.models import Model
```

Create a model in functional way by specifying both input and output layer −

```
model = Model(inputs = data, outputs = layer)
```

The complete code to create a simple model is shown below −

```
from keras.layers import Input
from keras.models import Model
from keras.layers import Dense


data = Input(shape=(2,3))
layer = Dense(2)(data) model =
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
=
_________________________________________________________________
dense_2 (Dense)              (None, 2, 2)              8
=================================================================
=
Total params: 8
```

Model Sub-Classing:

In **Model Sub-Classing** there are two most important functions ***__init__*** and ***call***. Basically, we will define all the trainable ***tf.keras*** layers or custom implemented layers inside the ***__init__*** method and call those layers based on our network design inside the ***call*** method which isused to perform a forward propagation. (It's quite the same as the ***forward*** method that is used to build the model in PyTorch anyway.)

Keras Layers:

A Keras layer requires ***shape of the input (input_shape)*** to understand the structure of the input data, ***initializer*** to set the weight for each input and finally activators to transform the output to make it non-linear. In between, constraints restricts and specify the range in which the weight of input data to be generated and regularizer will try to optimize the layer (and the model) by dynamically applying the penalties on the weights during optimization process.

To summarise, Keras layer requires below minimum details to create a complete layer.

- Shape of the input data
- Number of neurons / units in the layer
- Initializers
- Regularizers
- Constraints
- Activations

let us create a simple Keras layer using Sequential model API to get the idea of how Keras model and layer works.

```
from keras.models import Sequential

from keras.layers import Activation, Dense
from keras import initializers

from keras import regularizers
from keras import constraints


model = Sequential()


model.add(Dense(32,    input_shape=(16,),    kernel_initializer    =    'he_uniform',
```

- **Line 1-5** imports the necessary modules.
- **Line 7** creates a new model using Sequential API.
- **Line 9** creates a new ***Dense*** layer and add it into the model. ***Dense*** is an entry level layer provided by Keras, which accepts the number of neurons or units (32) as its required parameter. If the layer is first layer, then we need to provide **InputShape, (16,)** as well. Otherwise, the output of the previous layer will be used asinput of the next layer. All other parameters are optional.
    - First parameter represents the number of units (neurons).
    - ***input_shape*** represent the shape of input data.
    - ***kernel_initializer*** represent initializer to be used. ***he_uniform*** function is set as value.
    - ***kernel_regularizer*** represent **regularizer** to be used. None is setas value.
    - ***kernel_constraint represent constraint to be used. MaxNorm function is set as value.***
    - ***activation represent activation to be used. relu function is set as value.***
- **Line 10 creates second Dense layer with 16 units and set relu as the activation function.**
- **Line 11** creates final Dense layer with 8 units.

Custom Keras Layers:

Keras allows to create our own customized layer. Once a new layer is created, it can be used in any model without any restriction. Let us learn how to create new layer in this chapter.

Keras provides a base layer class, Layer which can sub-classed to create our own customized layer. Let us create a simple layer which will find weight based on normal distribution and then do the basic computation of finding the summation of the product of input and its weight during training.

Step 1: Import the necessary module

First, let us import the necessary modules −

```
from keras import backend as K
from keras.layers import Layer
```

Here,

- **backend** is used to access the **dot** function.
- **Layer** is the base class and we will be sub-classing it to create our layer

Step 2: Define a layer class

Let us create a new class, **MyCustomLayer** by sub-classing **Layer class** −

```
class MyCustomLayer(Layer):
  ...
```

Step 3: Initialize the layer class

Let us initialize our new class as specified below −

```
def __init__(self, output_dim, **kwargs):
  self.output_dim = output_dim
  super(MyCustomLayer, self).__init__(**kwargs)
```

Here,

- **Line 2** sets the output dimension.
- **Line 3** calls the base or super layer's **init** function.

Step 4: Implement build method

build is the main method and its only purpose is to build the layer properly. It can do anything related to the inner working of the layer. Once the custom functionality is done, we can call the base class **build** function. Our custom **build** function is as follows −

```
def build(self, input_shape):
  self.kernel = self.add_weight(name = 'kernel',
    shape = (input_shape[1], self.output_dim),
    initializer = 'normal', trainable = True)
  super(MyCustomLayer, self).build(input_shape)
```

Here,

- **Line 1** defines the **build** method with one argument, **input_shape**. Shape of the input data is referred by input_shape.
- **Line 2** creates the weight corresponding to input shape and set it in the kernel. It is our custom functionality of the layer. It creates the weight using 'normal' initializer.
- **Line 6** calls the base class, **build** method.

Step 5: Implement call method

call method does the exact working of the layer during training process.

Our custom **call** method is as follows

```
def call(self, input_data):
  return K.dot(input_data, self.kernel)
```

Here,

- **Line 1** defines the **call** method with one argument, **input_data**. input_data is the input data for our layer.
- **Line 2** return the dot product of the input data, **input_data** and our layer's kernel, **self.kernel**

Step 6: Implement compute_output_shape method

```
def compute_output_shape(self, input_shape): return (input_shape[0], self.output_dim)
```

Here,

- **Line 1** defines **compute_output_shape** method with one argument **input_shape**
- **Line 2** computes the output shape using shape of input data and output dimension set while initializing the layer.

Implementing the **build, call** and **compute_output_shape** completes the creating a customized layer. The final and complete code is as follows

```
from keras import backend as K from keras.layers import Layer
class MyCustomLayer(Layer):
  def __init__(self, output_dim, **kwargs):
    self.output_dim = output_dim
    super(MyCustomLayer, self).__init__(**kwargs)
  def build(self, input_shape): self.kernel =
```

```
    self.add_weight(name = 'kernel',
    shape = (input_shape[1], self.output_dim),
    initializer = 'normal', trainable = True)
    super(MyCustomLayer, self).build(input_shape) #
    Be sure to call this at the end
  def call(self, input_data): return K.dot(input_data, self.kernel)
  def compute_output_shape(self, input_shape): return (input_shape[0], self.output_dim)
```

Using our customized layer

Let us create a simple model using our customized layer as specified below −

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

model = Sequential()
model.add(MyCustomLayer(32, input_shape = (16,)))
model.add(Dense(8, activation = 'softmax')) model.summary()
```

Here,

- Our **MyCustomLayer** is added to the model using 32 units and **(16,)** as input shape

Running the application will print the model summary as below −

```
Model: "sequential_1"
_________________________________________________________________
Layer (type) Output Shape Param
#=================================================================
my_custom_layer_1 (MyCustomL (None, 32) 512
_________________________________________________________________
dense_1 (Dense) (None, 8) 264
=================================================================
Total params: 776
Trainable params: 776
Non-trainable params: 0
_________________________________________________________________
```