Keras

Keras is a high-level library that allows the use of TensorFlow as a backend deep learning library. TensorFlow team has included Keras in TensorFlow Core as module **tf.keras**. Apart from TensorFlow, Keras also supports Theano and CNTK.

We will focus on how to build different kinds of deep learning and machine learning models with both the **low-level TensorFlow API** and the **high-level Keras API**.

Installing Keras

Keras can be installed in Python 3 with the following command:

```
pip3 install keras
```

Neural Network Models in Keras

Neural network models in Keras are defined as the graph of layers. The models in Keras can be created using the sequential or the functional APIs. Both the **functional** and **sequential** APIs can be used to build any kind of models. The functional API makes it easier to build the complex models that have multiple inputs, multiple outputs and shared layers.

We use the sequential API for simple models built from simple layers and the functional API for complex models involving branches and sharing of layers. Also Building simple models with the functional API makes it easier to grow the models into complex models with branching and sharing. Hence for our work, we always use the functional API.

Workflow for building models in Keras:

- Create the model
- Create and add layers to the model
- Compile the model
- Train the model
- Use the model for prediction or evaluation

We will look at each of these steps.

1. Creating the Keras model
    a. The Keras model can be created using the sequential API or functional API.
    b. Sequential API for creating the Keras model
        i. In the sequential API, create the empty model with the following code:

```
model = Sequential()
```

        ii. We can now add the layers to this model.
        iii. Alternatively, we can also pass all the layers as a list to the constructor. As an example, we add four layers by passing them to the constructor using the following code:

```
model = Sequential([ Dense(10, input_shape=(256,)),
                     Activation('tanh'),
                     Dense(10),
                     Activation('softmax')
                    ])
```

    c. Functional API for creating the Keras model
        i. In the functional API, we create the model as an instance of the **Model** class that takes an input and output parameter. The input and output parameters represent one or more input and output tensors, respectively.
        ii. As an example, use the following code to instantiate a model from the functional API:

```
model = Model(inputs=tensor1, outputs=tensor2)
```

        iii. In the above code, **tensor1** and **tensor2** are either tensors or objects that can be treated like tensors, for example, Keras **layer** objects.
        iv. If there are more than one input and output tensors, they can be passed as a list, as shown in the following example

```
model = Model(inputs=[i1,i2,i3], outputs=[o1,o2,o3])
```

2. Keras Layers
    a. Keras provides several built-in layer classes for the easy construction of the network architecture.
    b. Keras core layers
        i. The Keras core layers implement fundamental operations that are used in almost every kind of network architecture.
        ii. The following tables give a summary and description of the layers provided by Keras 2:

Layer name	Description
`Dense`	This is a simple fully connected neural network layer. This layer produces the output of the following function: **activation((inputs x weights)+bias)** where **activation** refers to the activation function passed to the layer, which is `None` by default.
`Activation`	This layer applies the specified activation function to the output. This layer produces the output of the following function: **activation(inputs)** where **activation** refers to the activation function passed to the layer. The following activation functions are available to instantiate this layer: `softmax` , `elu` , `selu` , `softplus` , `softsign` , `relu` , `tanh` , `sigmoid` , `hard_sigmoid` , and `linear`
`Dropout`	This layer applies the dropout regularization to the inputs at a specified dropout rate.
`Flatten`	This layer flattens the input, that is, for a three-dimensional input, it flattens and produces a one-dimensional output.
`Reshape`	This layer converts the input to the specified shape.
`Permute`	This layer reorders the input dimensions as per the specified pattern.
`RepeatVector`	This layer repeats the input by the given number of times. Thus, if the input is a 2D tensor of shape (#samples, #features) and the layer is given **n** times to repeat, then the output will be a 3D tensor of shape (#samples, n, #features).
`Lambda`	This layer wraps the provided function as a layer. Thus, the inputs are passed through the custom function provided to produce the outputs. This layer provides ultimate extensibility to Keras users to add their own custom functions as layers.
`ActivityRegularization`	This layer applies L1 or L2, or a combination of both kinds of regularization to its inputs. This layer is applied to the output of an activation layer or to the output of a layer that has an activation function.
`Masking`	This layer masks or skips those time steps in the input tensor where all the values in the input tensor are equal to the mask value provided as an argument to the layer.

c. Keras convolutional layers

i. These layers implement the different type of convolution, sampling, and cropping operations for convolutional neural networks:

Layer Name	Description
`Conv1D`	This layer applies convolutions over a single spatial or temporal dimension to the inputs.
`Conv2D`	This layer applies two-dimensional convolutions to the inputs.
`SeparableConv2D`	This layer applies a depth-wise spatial convolution on each input channel, followed by a pointwise convolution that mixes together the resulting output channels.
`Conv2DTranspose`	This layer reverts the shape of convolutions to the shape of the inputs that produced those convolutions.
`Conv3D`	This layer applies three-dimensional convolutions to the inputs.
`Cropping1D`	This layer crops the input data along the temporal dimension.
`Cropping2D`	This layer crops the input data along the spatial dimensions, such as width and height in the case of an image.
`Cropping3D`	This layer crops the input data along the spatio-temporal, that is all three dimensions.
`UpSampling1D`	This layer repeats the input data by specified times along the time axis.
`UpSampling2D`	This layer repeats the row and column dimensions of the input data by specified times along the two dimensions.
`UpSampling3D`	This layer repeats the three dimensions of the input data by specified times along the three dimensions.
`ZeroPadding1D`	This layer adds zeros to the beginning and end of the time dimension.
`ZeroPadding2D`	This layer adds rows and columns of zeros to the top, bottom, left, or right of a 2D tensor.
`ZeroPadding3D`	This layer adds zeros to the three dimensions of a 3D tensor.

d. Keras pooling layers

i. These layers implement the different pooling operations for convolutional neural networks:

Layer Name	Description
`MaxPooling1D`	This layer implements the max pooling operation for one-dimensional input data.
`MaxPooling2D`	This layer implements the max pooling operation for two-dimensional input data.
`MaxPooling3D`	This layer implements the max pooling operation for three-dimensional input data.
`AveragePooling1D`	This layer implements the average pooling operation for two-dimensional input data.
`AveragePooling2D`	This layer implements the average pooling operation for two-dimensional input data.
`AveragePooling3D`	This layer implements the average pooling operation for three-dimensional input data.
`GlobalMaxPooling1D`	This layer implements the global max pooling operation for one-dimensional input data.
`GlobalAveragePooling1D`	This layer implements the global average pooling operation for one-dimensional input data.
`GlobalMaxPooling2D`	This layer implements the global max pooling operation for two-dimensional input data.
`GlobalAveragePooling2D`	This layer implements the global average pooling operation for two-dimensional input data.

e. Keras locally-connected layers

i. These layers are useful in convolutional neural networks:

Layer Name	Description
`LocallyConnected1D`	This layer applies convolutions over a single spatial or temporal dimension to the inputs, by applying a different set of filters at each different patch of the input, thus not sharing the weights.
`LocallyConnected2D`	This layer applies convolutions over two dimensions to the inputs, by applying a different set of filters at each different patch of the input, thus not sharing the weights.

f. Keras recurrent layers

i. These layers implement different variants of recurrent neural networks:

Layer Name	Description
`SimpleRNN`	This layer implements a fully connected recurrent neural network.
`GRU`	This layer implements a gated recurrent unit network.
`LSTM`	This layer implements a long short-term memory network.

g. Keras embedding layers

i. Presently, there is only one embedding layer option available:

Layer Name	Description
`Embedding`	This layer takes a 2D tensor of shape `(batch_size, sequence_length)` consisting of indexes, and produces a tensor consisting of dense vectors of shape `(batch_size, sequence_length, output_dim)` .

h. Keras merge layers

i. These layers merge two or more input tensors and produce a single output tensor by applying a specific operation that each layer represents:

Layer Name	Description
`Add`	This layer computes the element-wise addition of input tensors.
`Multiply`	This layer computes the element-wise multiplication of input tensors
`Average`	This layer computes the element-wise average of input tensors.
`Maximum`	This layer computes the element-wise maximum of input tensors.
`Concatenate`	This layer concatenates the input tensors along a specified axis.
`Dot`	This layer computes the dot product between samples in two input tensors.
`add` , `multiply` , `average` , `maximum` , `concatenate` , and `dot`	These functions represent the functional interface to the respective merge layers described in this table.

i. Keras advanced activation layers

i. These layers implement advanced activation functions that cannot be implemented as a simple underlying backend function.

Layer Name	Description
`LeakyReLU`	This layer computes the leaky version of the `ReLU` activation function.
`PReLU`	This layer computes the parametric `ReLU` activation function.
`ELU`	This layer computes the exponential linear unit activation function.
`ThresholdedReLU`	This layer computes the thresholded version of the `ReLU` activation function.

j. Keras normalization layers

i. Presently, there is only one normalization layer available:

Layer name	Description
`BatchNormalization`	This layer normalizes the outputs of the previous layer at each batch, such that the output of this layer is approximated to have a mean close to zero and a standard deviation close to 1.

k. Keras noise layers

i. These layers can be added to the model to prevent overfitting by adding noise; they are also known as regularization layers. These layers operate the same way as the Dropout() and ActivityRegularizer() layers in the core layers section

Layer name	Description
`GaussianNoise`	This layer applies additive zero-centered Gaussian noise to the inputs.
`GaussianDropout`	This layer applies multiplicative one-centered Gaussian noise to the inputs.
`AlphaDropout`	This layer drops a certain percentage of inputs, such that the mean and variance of the outputs after the dropout match closely with the mean and variance of the inputs.

3. Adding Layers to the Keras Model
   a. All the layers earlier need to be added to the model we created earlier.
   b. Sequential API to add layers to the Keras model
      i. In the sequential API, we can create layers by instantiating an object of one of the layer types given earlier. The created layers are then added to the model using the **model.add()** function. As an example, we will create a model and then add two layers to it:

```
model = Sequential()
model.add(Dense(10, input_shape=(256,))
model.add(Activation('tanh'))
model.add(Dense(10))
```

```
model.add(Activation('softmax'))
```

c. Functional API to add layers to the Keras Model
    i. In the functional API, the layers are created first in a functional manner, and then while creating the model, the input and output layers are provided as tensor arguments
    ii. Here is an example:
        1. First, create the input layer:

```
input = Input(shape=(64,))
```

        2. Next, create the dense layer from the input layer in a functional way:

```
hidden = Dense(10)(inputs)
```

        3. In the same way, create further hidden layers building from the previous layers in a functional way:

```
hidden = Activation('tanh')(hidden)
hidden = Dense(10)(hidden)
output = Activation('tanh')(hidden)
```

        4. Finally, instantiate the model object with the input and output layers:

```
model = Model(inputs=input, outputs=output)
```

4. Compiling the Keras model
    a. The model built needs to be compiled with the **model.compile()** method before it can be used for training and prediction. The full signature of the **compile()** method is as follows:

```
compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)
```

    b. The compile method takes three arguments:
        i. **optimizer** - We can specify your own function or one of the functions provided by Keras. This function is used to update the parameters in the optimization iterations. Keras offers the following built-in optimizer functions:
            1. **SGD**
            2. **RMSprop**
            3. **Adagrad**
            4. **Adadelta**
            5. **Adam**
            6. **Adamax**
            7. **Nadam**
        ii. **loss** - We can specify your own loss function or use one of the provided loss functions. The optimizer function optimizes the parameters so that

the output of this loss function is minimized. Keras provides the following loss functions:

1. **mean_squared_error**
2. **mean_absolute_error**
3. **mean_absolute_pecentage_error**
4. **mean_squared_logarithmic_error**
5. **squared_hinge**
6. **hinge**
7. **categorical_hinge**
8. **sparse_categorical_crossentropy**
9. **binary_crossentropy**
10. **poisson**
11. **cosine proximity**
12. **binary_accuracy**
13. **categorical_accuracy**
14. **sparse_categorical_accuracy**
15. **top_k_categorical_accuracy**
16. **sparse_top_k_categorical_accuracy**

iii. **metrics**: The third argument is a list of metrics that need to be collected while training the model. If verbose output is on, then the metrics are printed for each iteration. The metrics are like loss functions; some are provided by Keras with the ability to write our own metrics functions. All the loss functions also work as the metric function.

5. Training the Keras model
    a. Training a Keras model is as simple as calling the **model.fit()** method. The full signature of this method is as follows:

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None,
    validation_split=0.0, validation_data=None, shuffle=True,
    class_weight=None, sample_weight=None, initial_epoch=0)
```

    b. For the example model that we created earlier, train the model with the following code:

```
model.fit(x_data, y_labels)
```

6. Predicting with the Keras model
    a. The trained model can be used either to predict the value with the **model.predict()** method or to evaluate the model with the **model.evaluate()** method.

```
predict(self, x, batch_size=32, verbose=0)
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

Additional modules in Keras

Keras provides several additional modules that supplement the basic workflow with additional functionalities. Some of the modules are as follows:

- The **preprocessing** module provides several functions for the preprocessing of sequence, image, and text data.
- The **datasets** module provides several functions for quick access to several popular datasets, such as CIFAR10 images, CIFAR100 images, IMDB movie reviews, Reuters newswire topics, MNIST handwritten digits, and Boston housing prices.
- The **initializers** module provides several functions to set initial random weight parameters of layers, such as **Zeros**, **Ones**, **Constant**, **RandomNormal**, **RandomUniform**, **TruncatedNormal**, **VarianceScaling**, **Orthogonal, Identity**, **lecun_normal**, **lecun_uniform**, **glorot_normal**, **glorot_uniform**, **he_normal**, and **he_uniform**.
- The models module provides several functions to restore the model architectures and weights, such as **model_from_json**, **model_from_yaml**, and **load_model**. The model architectures can be saved using the **model.to_yaml()** and **model.to_json()** methods. The model weights can be saved by calling the **model.save()** method. The weights get saved in an **HDF5** file.
- The **applications** module provides several pre-built and pre-trained models such as Xception, VGG16, VGG19, ResNet50, Inception V3, InceptionResNet V2, and MobileNet. We will use the pre-built models to predict with our datasets. We retrain the pre-trained models in the applications module with our datasets from a slightly different domain.

Keras sequential model example for MNIST dataset

Click here for Implementation in jupyter notebook