Keras

About the Tutorial

Keras is an open source deep learning framework for python. It has been developed by an artificial intelligence researcher at Google named **Francois Chollet**. Leading organizations like Google, Square, Netflix, Huawei and Uber are currently using Keras. This tutorial walks through the installation of Keras, basics of deep learning, Keras models, Keras layers, Keras modules and finally conclude with some real-time applications.

Audience

This tutorial is prepared for professionals who are aspiring to make a career in the field of deep learning and neural network framework. This tutorial is intended to make you comfortable in getting started with the Keras framework concepts.

Prerequisites

Before proceeding with the various types of concepts given in this tutorial, we assume that the readers have basic understanding of deep learning framework. In addition to this, it will be very helpful, if the readers have a sound knowledge of Python and Machine Learning.

Copyright & Disclaimer

Table of Contents

1. Keras — Introduction

Deep learning is one of the major subfield of machine learning framework. Machine learning is the study of design of algorithms, inspired from the model of human brain. Deep learning is becoming more popular in data science fields like robotics, artificial intelligence(AI), audio & video recognition and image recognition. Artificial neural network is the core of deep learning methodologies. Deep learning is supported by various libraries such as Theano, TensorFlow, Caffe, Mxnet etc., Keras is one of the most powerful and easy to use python library, which is built on top of popular deep learning libraries like TensorFlow, Theano, etc., for creating deep learning models.

Overview of Keras

Keras runs on top of open source machine libraries like TensorFlow, Theano or Cognitive Toolkit (CNTK). Theano is a python library used for fast numerical computation tasks. TensorFlow is the most famous symbolic math library used for creating neural networks and deep learning models. TensorFlow is very flexible and the primary benefit is distributed computing. CNTK is deep learning framework developed by Microsoft. It uses libraries such as Python, C#, C++ or standalone machine learning toolkits. Theano and TensorFlow are very powerful libraries but difficult to understand for creating neural networks.

Keras is based on minimal structure that provides a clean and easy way to create deep learning models based on TensorFlow or Theano. Keras is designed to quickly define deep learning models. Well, Keras is an optimal choice for deep learning applications.

Features

Keras leverages various optimization techniques to make high level neural network API easier and more performant. It supports the following features:

- Consistent, simple and extensible API.
- Minimal structure - easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.

Benefits

Keras is highly powerful and dynamic framework and comes up with the following advantages:

- Larger community support.
- Easy to test.
- Keras neural networks are written in Python which makes things simpler.
- Keras supports both convolution and recurrent networks.

- Deep learning models are discrete components, so that, you can combine into many ways.

2. Keras — Installation

This chapter explains about how to install Keras on your machine. Before moving to installation, let us go through the basic requirements of Keras.

Prerequisites

You must satisfy the following requirements:

- Any kind of OS (Windows, Linux or Mac)
- Python version 3.5 or higher.

Python

Keras is python based neural network library so python must be installed on your machine. If python is properly installed on your machine, then open your terminal and type **python**, you could see the response similar as specified below,

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

As of now the latest version is **'3.7.2'**. If Python is not installed, then visit the official python link - https://www.python.org/ and download the latest version based on your OS and install it immediately on your system.

Keras Installation Steps

Keras installation is quite easy. Follow below steps to properly install Keras on your system.

Step 1: Create virtual environment

Virtualenv is used to manage Python packages for different projects. This will be helpful to avoid breaking the packages installed in the other environments. So, it is always recommended to use a virtual environment while developing Python applications.

Linux/Mac OS

Linux or mac OS users, go to your project root directory and type the below command to create virtual environment,

```
python3 -m venv kerasenv
```

After executing the above command, "kerasenv" directory is created with **bin,lib and include folders** in your installation location.

Windows

Windows user can use the below command,

```
py -m venv keras
```

Step 2: Activate the environment

This step will configure python and pip executables in your shell path.

Linux/Mac OS

Now we have created a virtual environment named "kerasvenv". Move to the folder and type the below command,

```
$ cd kerasvenv

kerasvenv $ source bin/activate
```

Windows

Windows users move inside the "kerasenv" folder and type the below command,

```
.\env\Scripts\activate
```

Step 3: Python libraries

Keras depends on the following python libraries.

- Numpy
- Pandas
- Scikit-learn
- Matplotlib
- Scipy
- Seaborn

Hopefully, you have installed all the above libraries on your system. If these libraries are not installed, then use the below command to install one by one.

numpy

```
pip install numpy
```

you could see the following response,

```
Collecting numpy
  Downloading
https://files.pythonhosted.org/packages/cf/a4/d5387a74204542a60ad1baa84cd2d3353
c330e59be8cf2d47c0b11d3cde8/
  numpy-3.1.1-cp36-cp36m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.
  macosx_10_10_intel.macosx_10_10_x86_64.whl (14.4MB)
     |████████████████████████████████| 14.4MB 2.8MB/s
```

pandas

```
pip install pandas
```

We could see the following response:

```
Collecting pandas
  Downloading
https://files.pythonhosted.org/packages/cf/a4/d5387a74204542a60ad1baa84cd2d3353
c330e59be8cf2d47c0b11d3cde8/
  pandas-3.1.1-cp36-cp36m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.
  macosx_10_10_intel.macosx_10_10_x86_64.whl (14.4MB)
      |████████████████████████████████| 14.4MB 2.8MB/s
```

matplotlib

```
pip install matplotlib
```

We could see the following response:

```
Collecting matplotlib
  Downloading
https://files.pythonhosted.org/packages/cf/a4/d5387a74204542a60ad1baa84cd2d3353
c330e59be8cf2d47c0b11d3cde8/
  matplotlib-3.1.1-cp36-cp36m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.
  macosx_10_10_intel.macosx_10_10_x86_64.whl (14.4MB)
      |████████████████████████████████| 14.4MB 2.8MB/s
```

scipy

```
pip install scipy
```

We could see the following response:

```
Collecting scipy
  Downloading
https://files.pythonhosted.org/packages/cf/a4/d5387a74204542a60ad1baa84cd2d3353
c330e59be8cf2d47c0b11d3cde8
  /scipy-3.1.1-cp36-cp36m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.
  macosx_10_10_intel.macosx_10_10_x86_64.whl (14.4MB)
      |████████████████████████████████| 14.4MB 2.8MB/s
```

scikit-learn

It is an open source machine learning library. It is used for classification, regression and clustering algorithms. Before moving to the installation, it requires the following:

- Python version 3.5 or higher
- NumPy version 1.11.0 or higher
- SciPy version 0.17.0 or higher

- joblib 0.11 or higher.

Now, we install scikit-learn using the below command:

```
pip install -U scikit-learn
```

Seaborn

Seaborn is an amazing library that allows you to easily visualize your data. Use the below command to install:

```
pip install seaborn
```

You could see the message similar as specified below:

```
Collecting seaborn
  Downloading
https://files.pythonhosted.org/packages/a8/76/220ba4420459d9c4c9c9587c6ce607bf5
6c25b3d3d2de62056efe482dadc
  /seaborn-0.9.0-py3-none-any.whl (208kB)
    100% |████████████████████████████████| 215kB 4.0MB/s
Requirement already satisfied: numpy>=1.9.3 in ./lib/python3.7/site-packages
(from seaborn) (1.17.0)
Collecting pandas>=0.15.2 (from seaborn)
  Downloading
https://files.pythonhosted.org/packages/39/b7/441375a152f3f9929ff8bc2915218ff1a
063a59d7137ae0546db616749f9/
  pandas-0.25.0-cp37-cp37m-macosx_10_9_x86_64.macosx_10_10_x86_64.whl (10.1MB)
    100% |████████████████████████████████| 10.1MB 1.8MB/s
Requirement already satisfied: scipy>=0.14.0 in ./lib/python3.7/site-packages
(from seaborn) (1.3.0)
Collecting matplotlib>=1.4.3 (from seaborn)
  Downloading
https://files.pythonhosted.org/packages/c3/8b/af9e0984f5c0df06d3fab0bf396eb09cb
f05f8452de4e9502b182f59c33b/
  matplotlib-3.1.1-cp37-cp37m-
macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64
  .macosx_10_10_intel.macosx_10_10_x86_64.whl (14.4MB)
    100% |████████████████████████████████| 14.4MB 1.4MB/s
......................................
......................................

Successfully installed cycler-0.10.0 kiwisolver-1.1.0 matplotlib-3.1.1 pandas-
0.25.0
pyparsing-2.4.2 python-dateutil-2.8.0 pytz-2019.2 seaborn-0.9.0
```

Keras Installation Using Python

As of now, we have completed basic requirements for the installtion of Kera. Now, install the Keras using same procedure as specified below:

```
pip install keras
```

Quit virtual environment

After finishing all your changes in your project, then simply run the below command to quit the environment:

```
deactivate
```

Anaconda Cloud

We believe that you have installed anaconda cloud on your machine. If anaconda is not installed, then visit the official link, https://www.anaconda.com/distribution/ and choose download based on your OS.

Create a new conda environment

Launch anaconda prompt, this will open base Anaconda environment. Let us create a new conda environment. This process is similar to virtualenv. Type the below command in your conda terminal:

```
conda create --name PythonCPU
```

If you want, you can create and install modules using GPU also. In this tutorial, we follow CPU instructions.

Activate conda environment

To activate the environment, use the below command:

```
activate PythonCPU
```

Install spyder

Spyder is an IDE for executing python applications. Let us install this IDE in our conda environment using the below command:

```
conda install spyder
```

Install python libraries

We have already known the python libraries numpy, pandas, etc., needed for keras. You can install all the modules by using the below syntax:

Syntax

```
conda install -c anaconda <module-name>
```

For example, you want to install pandas:

```
conda install -c anaconda pandas
```

Like the same method, try it yourself to install the remaining modules.

Install Keras

Now, everything looks good so you can start keras installation using the below command:

```
conda install -c anaconda keras
```

Launch spyder

Finally, launch spyder in your conda terminal using the below command:

```
spyder
```

To ensure everything was installed correctly, import all the modules, it will add everything and if anything went wrong, you will get **module not found** error message.

3. Keras — Backend Configuration

This chapter explains Keras backend implementations TensorFlow and Theano in detail. Let us go through each implementation one by one.

TensorFlow

TensorFlow is an open source machine learning library used for numerical computational tasks developed by Google. Keras is a high level API built on top of TensorFlow or Theano. We know already how to install TensorFlow using pip.

If it is not installed, you can install using the below command:

```
pip install TensorFlow
```

Once we execute keras, we could see the configuration file is located at your home directory inside and go to .keras/keras.json.

keras.json

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

Here,

- **image_data_format** represent the data format.
- **epsilon** represents numeric constant. It is used to avoid **DivideByZero** error.
- **floatx** represent the default data type **float32**. You can also change it to **float16** or **float64** using **set_floatx()** method.
- **backend** denotes the current backend.

Suppose, if the file is not created then move to the location and create using the below steps:

```
> cd home
> mkdir .keras
> vi keras.json
```

Remember, you should specify .keras as its folder name and add the above configuration inside keras.json file. We can perform some pre-defined operations to know backend functions.

Theano

Theano is an open source deep learning library that allows you to evaluate multi-dimensional arrays effectively. We can easily install using the below command:

```
pip install theano
```

By default, keras uses TensorFlow backend. If you want to change backend configuration from TensorFlow to Theano, just change the backend = theano in keras.json file. It is described below:

keras.json

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "theano"
}
```

Now save your file, restart your terminal and start keras, your backend will be changed.

```
>>> import keras as k

using theano backend.
```

4. Keras — Overview of Deep learning

Deep learning is an evolving subfield of machine learning. Deep learning involves analyzing the input in layer by layer manner, where each layer progressively extracts higher level information about the input.

Let us take a simple scenario of analyzing an image. Let us assume that your input image is divided up into a rectangular grid of pixels. Now, the first layer abstracts the pixels. The second layer understands the edges in the image. The Next layer constructs nodes from the edges. Then, the next would find branches from the nodes. Finally, the output layer will detect the full object. Here, the feature extraction process goes from the output of one layer into the input of the next subsequent layer.

By using this approach, we can process huge amount of features, which makes deep learning a very powerful tool. Deep learning algorithms are also useful for the analysis of unstructured data. Let us go through the basics of deep learning in this chapter.

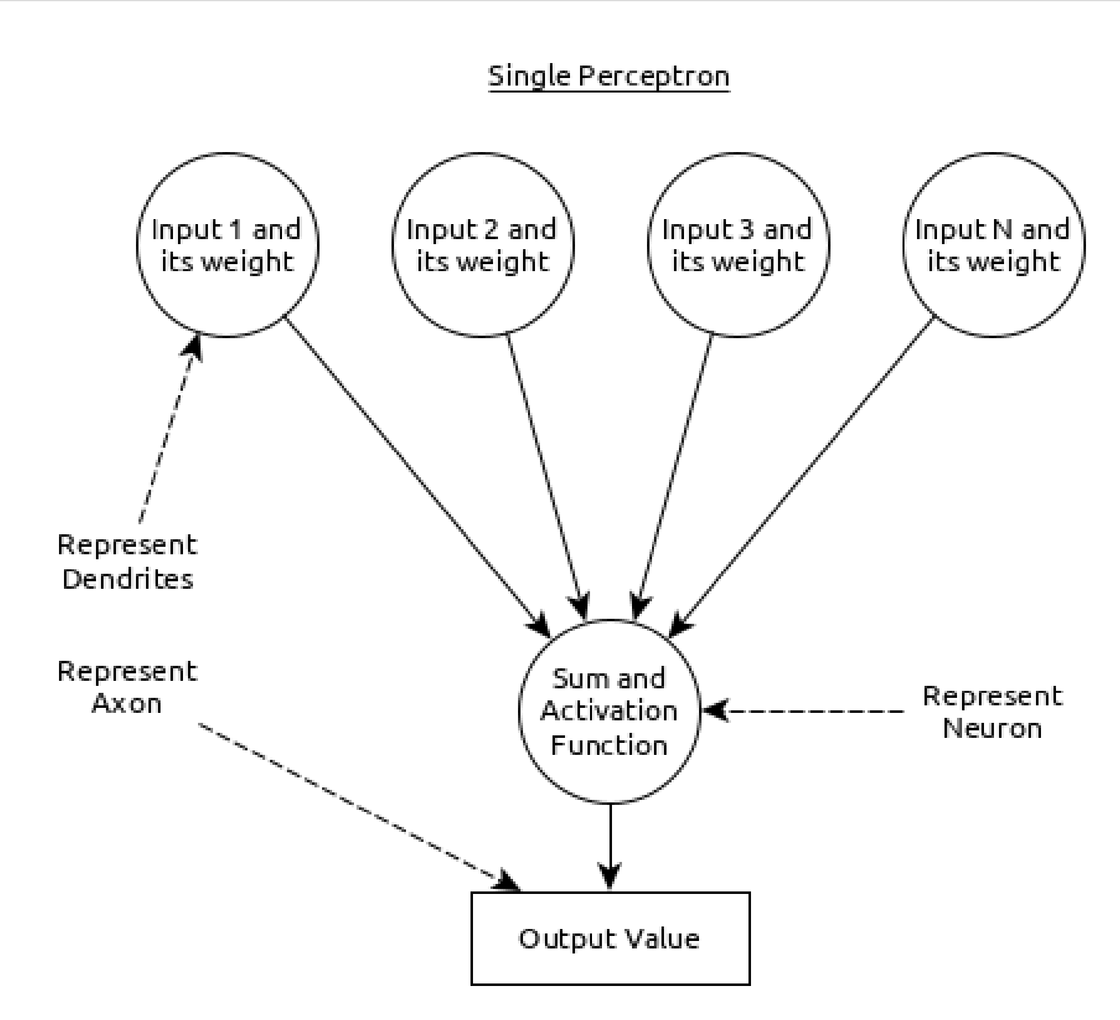
Artificial Neural Networks

The most popular and primary approach of deep learning is using "Artificial neural network" (ANN). They are inspired from the model of human brain, which is the most complex organ of our body. The human brain is made up of more than 90 billion tiny cells called "Neurons". Neurons are inter-connected through nerve fiber called "axons" and "Dendrites". The main role of axon is to transmit information from one neuron to another to which it is connected.

Similarly, the main role of dendrites is to receive the information being transmitted by the axons of another neuron to which it is connected. Each neuron processes a small information and then passes the result to another neuron and this process continues. This is the basic method used by our human brain to process huge about of information like speech, visual, etc., and extract useful information from it.

Based on this model, the first Artificial Neural Network (ANN) was invented by psychologist **Frank Rosenblatt**, in the year of 1958. ANNs are made up of multiple nodes which is similar to neurons. Nodes are tightly interconnected and organized into different hidden layers. The input layer receives the input data and the data goes through one or more hidden layers sequentially and finally the output layer predict something useful about the input data. For example, the input may be an image and the output may be the thing identified in the image, say a "Cat".

A single neuron (called as perceptron in ANN) can be represented as below:



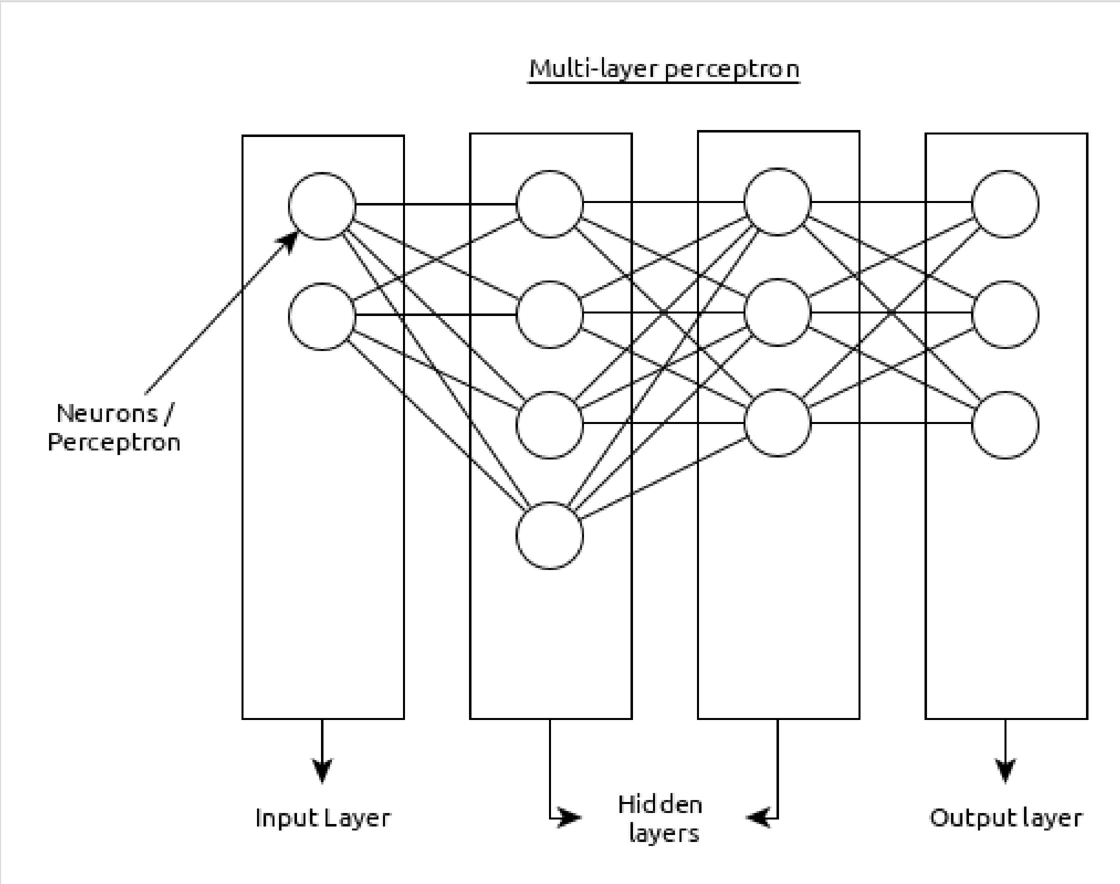
Here,

- Multiple input along with weight represents dendrites.
- Sum of input along with activation function represents neurons. **Sum** actually means computed value of all inputs and activation function represent a function, which modify the **Sum** value into 0, 1 or 0 to 1.
- Actual output represent axon and the output will be received by neuron in next layer.

Let us understand different types of artificial neural networks in this section.

Multi-Layer Perceptron

Multi-Layer perceptron is the simplest form of ANN. It consists of a single input layer, one or more hidden layer and finally an output layer. A layer consists of a collection of perceptron. Input layer is basically one or more features of the input data. Every hidden layer consists of one or more neurons and process certain aspect of the feature and send the processed information into the next hidden layer. The output layer process receives the data from last hidden layer and finally output the result.

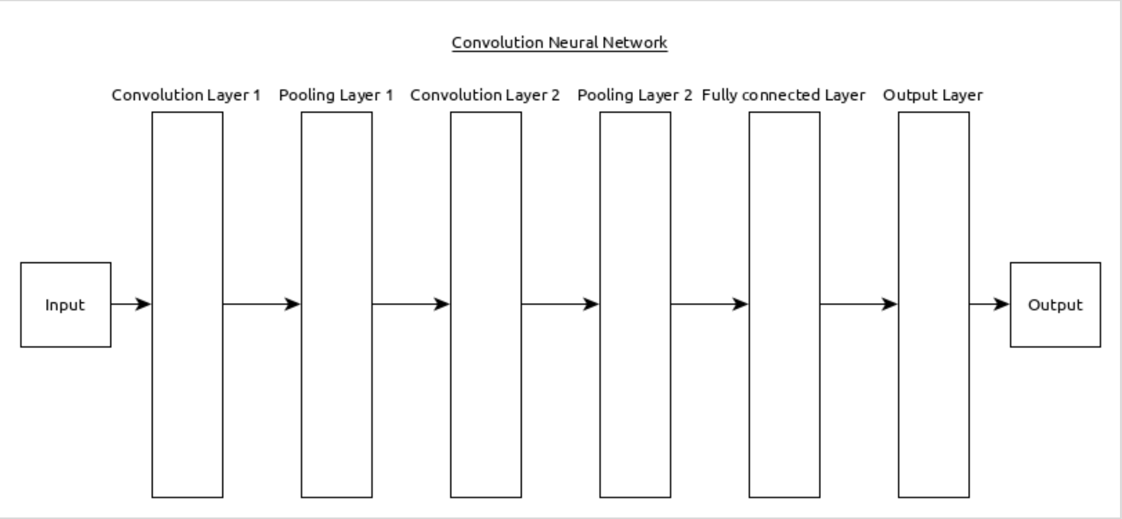


Convolutional Neural Network (CNN)

Convolutional neural network is one of the most popular ANN. It is widely used in the fields of image and video recognition. It is based on the concept of convolution, a mathematical concept. It is almost similar to multi-layer perceptron except it contains series of convolution layer and pooling layer before the fully connected hidden neuron layer. It has three important layers:

- **Convolution layer:** It is the primary building block and perform computational tasks based on convolution function.
- **Pooling layer**: It is arranged next to convolution layer and is used to reduce the size of inputs by removing unnecessary information so computation can be performed faster.
- **Fully connected layer:** It is arranged to next to series of convolution and pooling layer and classify input into various categories.

A simple CNN can be represented as below:



Here,

- 2 series of Convolution and pooling layer is used and it receives and process the input (e.g. image).
- A single fully connected layer is used and it is used to output the data (e.g. classification of image)

Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNN) are useful to address the flaw in other ANN models. Well, Most of the ANN doesn't remember the steps from previous situations and learned to make decisions based on context in training. Meanwhile, RNN stores the past information and all its decisions are taken from what it has learnt from the past.

This approach is mainly useful in image classification. Sometimes, we may need to look into the future to fix the past. In this case bidirectional RNN is helpful to learn from the past and predict the future. For example, we have handwritten samples in multiple inputs. Suppose, we have confusion in one input then we need to check again other inputs to recognize the correct context which takes the decision from the past.

Workflow of ANN

Let us first understand the different phases of deep learning and then, learn how Keras helps in the process of deep learning.

Collect required data

Deep learning requires lot of input data to successfully learn and predict the result. So, first collect as much data as possible.

Analyze data

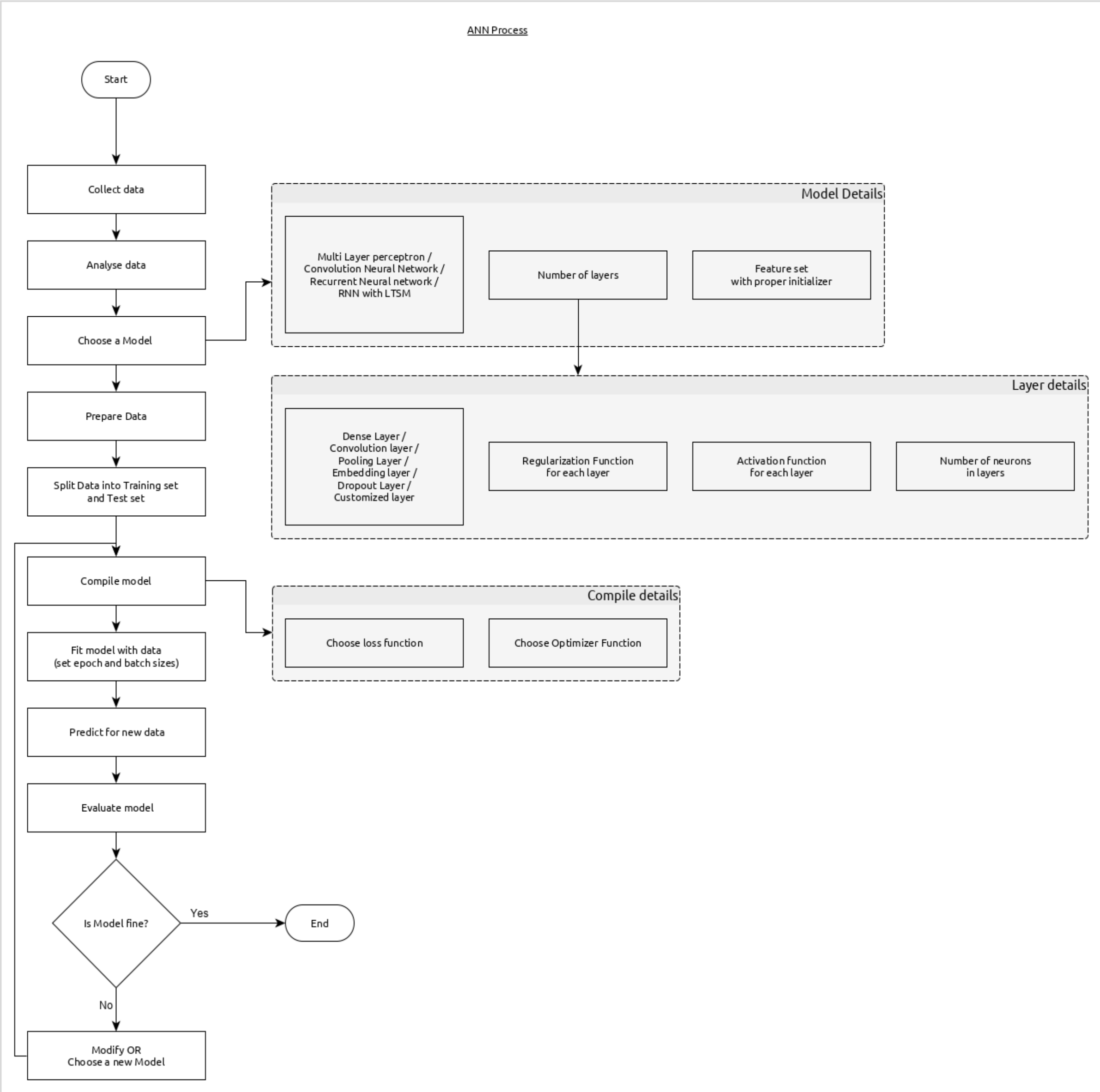
Analyze the data and acquire a good understanding of the data. The better understanding of the data is required to select the correct ANN algorithm.

Choose an algorithm (model)

Choose an algorithm, which will best fit for the type of learning process (e.g image classification, text processing, etc.,) and the available input data. Algorithm is represented by **Model** in Keras. Algorithm includes one or more layers. Each layers in ANN can be represented by **Keras Layer** in Keras.

- **Prepare data**: Process, filter and select only the required information from the data.
- **Split data:** Split the data into training and test data set. Test data will be used to evaluate the prediction of the algorithm / Model (once the machine learn) and to cross check the efficiency of the learning process.
- **Compile the model:** Compile the algorithm / model, so that, it can be used further to learn by training and finally do to prediction. This step requires us to choose **loss function** and **Optimizer. loss function** and **Optimizer** are used in learning phase to find the error (deviation from actual output) and do optimization so that the error will be minimized.
- **Fit the model:** The actual learning process will be done in this phase using the training data set.
- **Predict result for unknown value:** Predict the output for the unknown input data (other than existing training and test data)
- **Evaluate model:** Evaluate the model by predicting the output for test data and cross-comparing the prediction with actual result of the test data.
- **Freeze, Modify or choose new algorithm:** Check whether the evaluation of the model is successful. If yes, save the algorithm for future prediction purpose. If not, then modify or choose new algorithm / model and finally, again train, predict and evaluate the model. Repeat the process until the best algorithm (model) is found.

The above steps can be represented using below flow chart:



ANN Process
Start
Collect data
Analyse data
Choose a Model
Prepare Data
Split Data into Training set and Test set
Compile model
Fit model with data (set epoch and batch sizes)
Predict for new data
Evaluate model
Is Model fine?
Yes
End
No
Modify OR Choose a new Model
Model Details
Multi Layer perceptron / Convolution Neural Network / Recurrent Neural network / RNN with LTSM
Number of layers
Feature set with proper initializer
Layer details
Dense Layer / Convolution layer / Pooling Layer / Embedding layer / Dropout Layer / Customized layer
Regularization Function for each layer
Activation function for each layer
Number of neurons in layers
Compile details
Choose loss function
Choose Optimizer Function

5. Keras — Deep learning with Keras

Keras provides a complete framework to create any type of neural networks. Keras is innovative as well as very easy to learn. It supports simple neural network to very large and complex neural network model. Let us understand the architecture of Keras framework and how Keras helps in deep learning in this chapter.

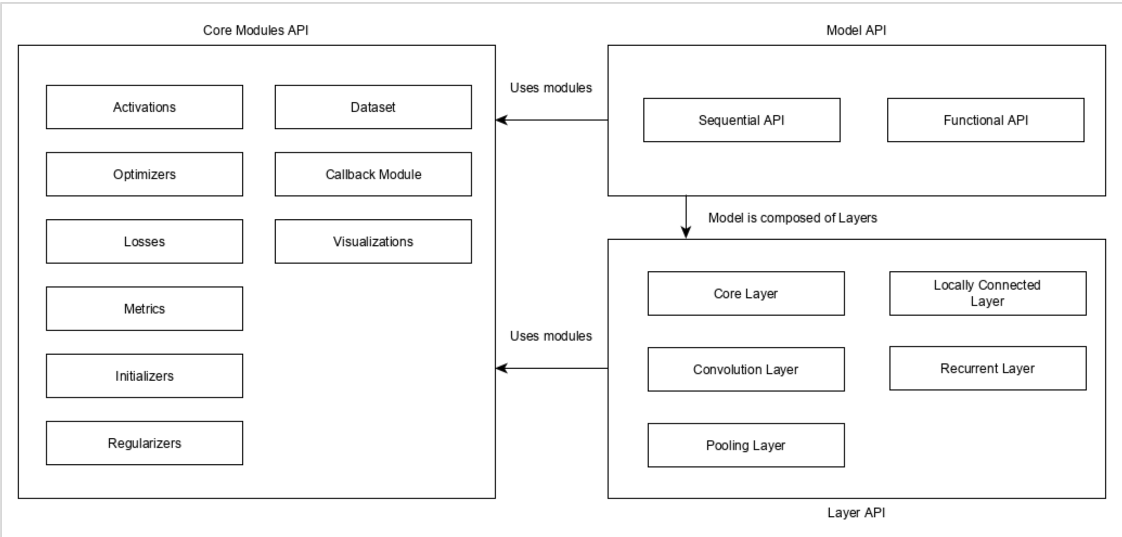
Architecture of Keras

Keras API can be divided into three main categories:

- Model
- Layer
- Core Modules

In Keras, every ANN is represented by **Keras Models**. In turn, every Keras Model is composition of **Keras Layers** and represents ANN layers like input, hidden layer, output layers, convolution layer, pooling layer, etc., Keras model and layer access **Keras modules** for activation function, loss function, regularization function, etc., Using Keras model, Keras Layer, and Keras modules, any ANN algorithm (CNN, RNN, etc.,) can be represented in a simple and efficient manner.

The following diagram depicts the relationship between model, layer and core modules:



Let us see the overview of Keras models, Keras layers and Keras modules.

Model

Keras Models are of two types as mentioned below:

Sequential Model - Sequential model is basically a linear composition of **Keras Layers**. Sequential model is easy, minimal as well as has the ability to represent nearly all available neural networks.

A simple sequential model is as follows:

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

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
```

Where,

- **Line 1** imports **Sequential** model from Keras models
- **Line 2** imports **Dense** layer and **Activation** module
- **Line 4** create a new sequential model using **Sequential** API
- **Line 5** adds a dense layer (Dense API) with **relu** activation (using Activation module) function.

Sequential model exposes **Model** class to create customized models as well. We can use sub-classing concept to create our own complex model.

Functional API: Functional API is basically used to create complex models.

Layer

Each Keras layer in the Keras model represent the corresponding layer (input layer, hidden layer and output layer) in the actual proposed neural network model. Keras provides a lot of pre-build layers so that any complex neural network can be easily created. Some of the important Keras layers are specified below,

- Core Layers
- Convolution Layers
- Pooling Layers
- Recurrent Layers

A simple python code to represent a neural network model using **sequential** model is as follows:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
```

Where,

- **Line 1** imports **Sequential** model from Keras models
- **Line 2** imports **Dense** layer, **Dropout** layer and **Activation** module
- **Line 4** create a new sequential model using **Sequential** API
- **Line 5** adds a dense layer (Dense API) with **relu** activation (using Activation module) function.
- **Line 6** adds a dropout layer (Dropout API) to handle over-fitting.
- **Line 7** adds another dense layer (Dense API) with **relu** activation (using Activation module) function.
- **Line 8** adds another dropout layer (Dropout API) to handle over-fitting.
- **Line 9** adds final dense layer (Dense API) with **softmax** activation (using Activation module) function.

Keras also provides options to create our own customized layers. Customized layer can be created by sub-classing the **Keras.Layer** class and it is similar to sub-classing Keras models.

Core Modules

Keras also provides a lot of built-in neural network related functions to properly create the Keras model and Keras layers. Some of the function are as follows:

- **Activations module** - Activation function is an important concept in ANN and activation modules provides many activation function like softmax, relu, etc.,
- **Loss module** - Loss module provides loss functions like mean_squared_error, mean_absolute_error, poisson, etc.,
- **Optimizer module** - Optimizer module provides optimizer function like adam, sgd, etc.,
- **Regularizers** - Regularizer module provides functions like **L1** regularizer, **L2** regularizer, etc.,

Let us learn Keras modules in detail in the upcoming chapter.

6. Keras — Modules

As we learned earlier, Keras modules contains pre-defined classes, functions and variables which are useful for deep learning algorithm. Let us learn the modules provided by Keras in this chapter.

Available modules

Let us first see the list of modules available in the Keras.

- **Initializers:** Provides a list of initializers function. We can learn it in details in *Keras layer* chapter. during model creation phase of machine learning.

- **Regularizers:** Provides a list of regularizers function. We can learn it in details in *Keras Layers* chapter.

- **Constraints:** Provides a list of constraints function. We can learn it in details in *Keras Layers* chapter.

- **Activations:** Provides a list of activator function. We can learn it in details in *Keras Layers* chapter.

- **Losses:** Provides a list of loss function. We can learn it in details in *Model Training* chapter.

- **Metrics:** Provides a list of metrics function. We can learn it in details in *Model Training* chapter.

- **Optimizers:** Provides a list of optimizer function. We can learn it in details in *Model Training* chapter.

- **Callback:** Provides a list of callback function. We can use it during the training process to print the intermediate data as well as to stop the training itself (**EarlyStopping** method) based on some condition.

- **Text processing:** Provides functions to convert text into NumPy array suitable for machine learning. We can use it in data preparation phase of machine learning.

- **Image processing:** Provides functions to convert images into NumPy array suitable for machine learning. We can use it in data preparation phase of machine learning.

- **Sequence processing:** Provides functions to generate time based data from the given input data. We can use it in data preparation phase of machine learning.

- **Backend:** Provides function of the *backend* library like *TensorFlow* and *Theano*.

- **Utilities:** Provides lot of utility function useful in deep learning.

Let us see **backend** module and **utils** model in this chapter.

backend module

***backend* module** is used for keras backend operations. By default, keras runs on top of *TensorFlow* backend. If you want, you can switch to other backends like Theano or CNTK. Defualt backend configuration is defined inside your root directory under *.keras/keras.json* file.

Keras *backend* module can be imported using below code:

```
>>> from keras import backend as k
```

If we are using default backend *TensorFlow*, then the below function returns *TensorFlow* based information as specified below:

```
>>> k.backend()
'tensorflow'

>>> k.epsilon()
1e-07

>>> k.image_data_format()
'channels_last'

>>> k.floatx()
'float32'
```

Let us understand some of the significant backend functions used for data analysis in brief:

get_uid()

It is the identifier for the default graph. It is defined below:

```
>>> k.get_uid(prefix='')
1

>>> k.get_uid(prefix='')
2
```

reset_uids

It is used resets the uid value.

```
>>> k.reset_uids()
```

Now, again execute the *get_uid()*. This will be reset and change again to 1.

```
>>> k.get_uid(prefix='')
1
```

placeholder

It is used instantiates a placeholder tensor. Simple placeholder to hold 3-D shape is shown below:

```
>>> data = k.placeholder(shape=(1,3,3))
>>> data
<tf.Tensor 'Placeholder_9:0' shape=(1, 3, 3) dtype=float32>

If you use int_shape(), it will show the shape.

>>> k.int_shape(data)
(1, 3, 3)
```

dot

It is used to multiply two tensors. Consider a and b are two tensors and c will be the outcome of multiply of ab. Assume a shape is (4,2) and b shape is (2,3). It is defined below,

```
>>> a = k.placeholder(shape=(4,2))
>>> b = k.placeholder(shape=(2,3))
>>> c = k.dot(a,b)
>>> c
<tf.Tensor 'MatMul_3:0' shape=(4, 3) dtype=float32>
>>>
```

ones

It is used to initialize all as **one** value.

```
>>> res = k.ones(shape=(2,2))

#print the value

>>> k.eval(res)
array([[1., 1.],
       [1., 1.]], dtype=float32)
```

batch_dot

It is used to perform the product of two data in batches. Input dimension must be 2 or higher. It is shown below:

```
>>> a_batch = k.ones(shape=(2,3))
>>> b_batch = k.ones(shape=(3,2))
>>> c_batch = k.batch_dot(a_batch,b_batch)
>>> c_batch
<tf.Tensor 'ExpandDims:0' shape=(2, 1) dtype=float32>
```

variable

It is used to initializes a variable. Let us perform simple transpose operation in this variable.

```
>>> data = k.variable([[10,20,30,40],[50,60,70,80]])  #variable initialized
here
>>> result = k.transpose(data)
>>> print(result)
Tensor("transpose_6:0", shape=(4, 2), dtype=float32)
>>> print(k.eval(result))
[[10. 50.]
 [20. 60.]
 [30. 70.]
 [40. 80.]]
```

If you want to access from numpy:

```
>>> data = np.array([[10,20,30,40],[50,60,70,80]])

>>> print(np.transpose(data))
[[10 50]
 [20 60]
 [30 70]
 [40 80]]

>>> res = k.variable(value=data)
>>> print(res)
<tf.Variable 'Variable_7:0' shape=(2, 4) dtype=float32_ref>
```

is_sparse(tensor)

It is used to check whether the tensor is sparse or not.

```
>>> a = k.placeholder((2, 2), sparse=True)

>>> print(a)
SparseTensor(indices=Tensor("Placeholder_8:0", shape=(?, 2), dtype=int64),
 values=Tensor("Placeholder_7:0", shape=(?,), dtype=float32),
 dense_shape=Tensor("Const:0", shape=(2,), dtype=int64))

>>> print(k.is_sparse(a))
True
```

to_dense()

It is used to converts sparse into dense.

```
>>> b = k.to_dense(a)
>>> print(b)
Tensor("SparseToDense:0", shape=(2, 2), dtype=float32)
>>> print(k.is_sparse(b))
False
```

random_uniform_variable

It is used to initialize using ***uniform distribution*** concept.

```
k.random_uniform_variable(shape, mean, scale)
```

Here,

- **shape** - denotes the rows and columns in the format of tuples.
- **mean** - mean of uniform distribution.
- **scale** - standard deviation of uniform distribution.

Let us have a look at the below example usage:

```
>>> a = k.random_uniform_variable(shape=(2, 3), low=0, high=1)
>>> b = k. random_uniform_variable(shape=(3,2), low=0, high=1)
>>> c = k.dot(a, b)
>>> k.int_shape(c)
(2, 2)
```

utils module

utils provides useful utilities function for deep learning. Some of the methods provided by the ***utils*** module is as follows:

HDF5Matrix

It is used to represent the input data in HDF5 format.

```
from keras.utils import HDF5Matrix

data = HDF5Matrix('data.hdf5', 'data')
```

to_categorical

It is used to convert class vector into binary class matrix.

```
>>> from keras.utils import to_categorical

>>> labels = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> to_categorical(labels)
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]], dtype=float32)
```

normalize

It is used to normalize the NumPy array.

```
>>> from keras.utils import normalize

>>> normalize([1, 2, 3, 4, 5])
array([[0.13483997, 0.26967994, 0.40451992, 0.53935989, 0.67419986]])
```

print_summary

It is used to print the summary of the model.

```
from keras.utils import print_summary

print_summary(model)
```

plot_model

It is used to create the model representation in dot format and save it to file.

```
from keras.utils import plot_model

plot_model(model,to_file='image.png')
```

This ***plot_model*** will generate an image to understand the performance of model.

7. Keras — Layers

As learned earlier, Keras layers are the primary building block of Keras models. Each layer receives input information, do some computation and finally output the transformed information. The output of one layer will flow into the next layer as its input. Let us learn complete details about layers in this chapter.

Introduction

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 understand the basic concept in the next chapter. Before understanding the basic concept, 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',
kernel_regularizer=None, kernel_constraint='MaxNorm', activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(8))
```

where,

- **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 **Input Shape**, **(16,)** as well. Otherwise, the output of the previous layer will be used as input 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 set as 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.

Basic Concept of Layers

Let us understand the basic concept of layer as well as how Keras supports each concept.

Input shape

In machine learning, all type of input data like text, images or videos will be first converted into array of numbers and then feed into the algorithm. Input numbers may be single dimensional array, two dimensional array (matrix) or multi-dimensional array. We can specify the dimensional information using **shape**, a tuple of integers. For example, **(4,2)** represent matrix with four rows and two columns.

```
>>> import numpy as np
>>> shape = (4, 2)
>>> input = np.zeros(shape)
>>> print(input)
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
>>>
```

Similarly, **(3,4,2)** three dimensional matrix having three collections of 4x2 matrix (two rows and four columns).

```
>>> import numpy as np
>>> shape = (3, 4, 2)
>>> input = np.zeros(shape)
>>> print(input)
[[[0. 0.]
  [0. 0.]
```

```
  [0. 0.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]
  [0. 0.]]]
>>>
```

To create the first layer of the model (or input layer of the model), shape of the input data should be specified.

Initializers

In Machine Learning, weight will be assigned to all input data. ***Initializers*** module provides different functions to set these initial weight. Some of the ***Keras Initializer*** function are as follows:

Zeros

Generates **0** for all input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Zeros()
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

Where, ***kernel_initializer*** represent the initializer for kernel of the model.

Ones

Generates **1** for all input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Ones()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

Constant

Generates a constant value (say, **5**) specified by the user for all input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Constant(value=0)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

where, **value** represent the constant value

RandomNormal

Generates value using normal distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

where,

- ***mean*** represent the mean of the random values to generate
- ***stddev*** represent the standard deviation of the random values to generate
- ***seed*** represent the values to generate random number

RandomUniform

Generates value using uniform distribution of input data.

```
from keras import initializers

my_init = initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

where,

- ***minval*** represent the lower bound of the random values to generate
- ***maxval*** represent the upper bound of the random values to generate

TruncatedNormal

Generates value using truncated normal distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)
```

```
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

VarianceScaling

Generates value based on the input shape and output shape of the layer along with the specified scale.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.VarianceScaling(scale=1.0, mode='fan_in',
distribution='normal', seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

where,

- **scale** represent the scaling factor
- **mode** represent any one of **fan_in, fan_out** and **fan_avg** values
- **distribution** represent either of **normal** or **uniform**

VarianceScaling

It finds the ***stddev*** value for normal distribution using below formula and then find the weights using normal distribution,

```
stddev = sqrt(scale / n)
```

where **n** represent,

- number of input units for mode = fan_in
- number of out units for mode = fan_out
- average number of input and output units for mode = fan_avg

Similarly, it finds the *limit* for uniform distribution using below formula and then find the weights using uniform distribution,

```
limit = sqrt(3 * scale / n)
```

lecun_normal

Generates value using lecun normal distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)
```

```
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the ***stddev*** using the below formula and then apply normal distribution

```
stddev = sqrt(1 / fan_in)
```

where, ***fan_in*** represent the number of input units.

lecun_uniform

Generates value using lecun uniform distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.lecun_uniform(seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the ***limit*** using the below formula and then apply uniform distribution

```
limit = sqrt(3 / fan_in)
```

where,

- **fan_in** represents the number of input units
- **fan_out** represents the number of output units

glorot_normal

Generates value using glorot normal distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.glorot_normal(seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the ***stddev*** using the below formula and then apply normal distribution

```
stddev = sqrt(2 / (fan_in + fan_out))
```

where,

- ***fan_in*** represents the number of input units
- ***fan_out*** represents the number of output units

glorot_uniform

Generates value using glorot uniform distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.glorot_uniform(seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the ***limit*** using the below formula and then apply uniform distribution

```
limit = sqrt(6 / (fan_in + fan_out))
```

where,

- ***fan_in*** represent the number of input units.
- ***fan_out*** represent the number of output units.

he_normal

Generates value using he normal distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the *stddev* using the below formula and then apply normal distribution.

```
stddev = sqrt(2 / fan_in)
```

where, ***fan_in*** represent the number of input units.

he_uniform

Generates value using he uniform distribution of input data.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.he_normal(seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

It finds the ***limit*** using the below formula and then apply uniform distribution.

```
limit = sqrt(6 / fan_in)
```

where, ***fan_in*** represent the number of input units

Orthogonal

Generates a random orthogonal matrix.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Orthogonal(gain=1.0, seed=None)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

where, ***gain*** represent the multiplication factor of the matrix.

Identity

Generates identity matrix.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Identity(gain=1.0)
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_initializer=my_init))
```

Constraints

In machine learning, a constraint will be set on the parameter (weight) during optimization phase. *Constraints* module provides different functions to set the constraint on the layer. Some of the constraint functions are as follows:

NonNeg

Constrains weights to be non-negative.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import constraints

my_constrain = constraints.NonNeg()
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_constraint=my_constrain))
```

where, ***kernel_constraint*** represent the constraint to be used in the layer.

UnitNorm

Constrains weights to be unit norm.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import constraints

my_constrain = constraints.UnitNorm(axis=0)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_constraint=my_constrain))
```

MaxNorm

Constrains weight to norm less than or equals to the given value.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import constraints

my_constrain = constraints.MaxNorm(max_value=2, axis=0)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_constraint=my_constrain))
```

where,

- **max_*value*** represent the upper bound
- *axis* represent the dimension in which the constraint to be applied. e.g. in Shape (2,3,4) axis 0 denotes first dimension, 1 denotes second dimension and 2 denotes third dimension

MinMaxNorm

Constrains weights to be norm between specified minimum and maximum values.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import constraints

my_constrain = constraints.MinMaxNorm(min_value=0.0, max_value=1.0, rate=1.0,
axis=0)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,),
kernel_constraint=my_constrain))
```

where, ***rate*** represent the rate at which the weight constrain is applied.

Regularizers

In machine learning, regularizers are used in the optimization phase. It applies some penalties on the layer parameter during optimization. Keras regularization module provides below functions to set penalties on the layer. Regularization applies per-layer basis only.

L1 Regularizer

It provides L1 based regularization.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import regularizers

my_regularizer = regularizers.l1(0.)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,), 
kernel_regularizer=my_regularizer))
```

where, ***kernel_regularizer*** represent the regularizer to be used in the layer.

L2 Regularizer

It provides L2 based regularization.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import regularizers

my_regularizer = regularizers.l2(0.)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,), 
kernel_regularizer=my_regularizer))
```

L1 and L2 Regularizer

It provides both L1 and L2 based regularization.

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import regularizers

my_regularizer = regularizers.l1_l2(0.)
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,), 
kernel_regularizer=my_regularizer))
```

Activations

In machine learning, activation function is a special function used to find whether a specific neuron is activated or not. Basically, the activation function does a nonlinear transformation of the input data and thus enable the neurons to learn better. Output of a neuron depends on the activation function.

As you recall the concept of single perception, the output of a perceptron (neuron) is simply the result of the activation function, which accepts the summation of all input multiplied with its corresponding weight plus overall bias, if any available.

```
result = Activation(SUMOF(input * weight) + bias)
```

So, activation function plays an important role in the successful learning of the model. Keras provides a lot of activation function in the *activations* module. Let us learn all the activations available in the module.

linear

Applies Linear function. Does nothing.

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

model = Sequential()
model.add(Dense(512, activation='linear', input_shape=(784,)))
```

where, ***activation*** refers the activation function of the layer. It can be specified simply by the name of the function and the layer will use corresponding activators.

elu

Applies Exponential linear unit.

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

model = Sequential()
model.add(Dense(512, activation='elu', input_shape=(784,)))
```

selu

Applies Scaled exponential linear unit.

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

model = Sequential()
model.add(Dense(512, activation='selu', input_shape=(784,)))
```

relu

Applies Rectified Linear Unit.

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

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
```

softmax

Applies Softmax function.

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

model = Sequential()
model.add(Dense(512, activation='softmax', input_shape=(784,)))
```

softplus

Applies Softplus function.

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

model = Sequential()
model.add(Dense(512, activation='softplus', input_shape=(784,)))
```

softsign

Applies Softsign function.

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

model = Sequential()
model.add(Dense(512, activation='softsign', input_shape=(784,)))
```

tanh

Applies Hyperbolic tangent function.

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

model = Sequential()
model.add(Dense(512, activation='tanh', input_shape=(784,)))
```

sigmoid

Applies Sigmoid function.

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

model = Sequential()
model.add(Dense(512, activation='sigmoid', input_shape=(784,)))
```

hard_sigmoid

Applies Hard Sigmoid function.

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

model = Sequential()
model.add(Dense(512, activation='hard_sigmoid', input_shape=(784,)))
```

exponential

Applies exponential function.

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

model = Sequential()
model.add(Dense(512, activation='exponential', input_shape=(784,)))
```

Dense Layer

Dense layer is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.

```
output = activation(dot(input, kernel) + bias)
```

where,

- **input** represent the input data
- **kernel** represent the weight data
- **dot** represent numpy dot product of all input and its corresponding weights
- **bias** represent a biased value used in machine learning to optimize the model
- **activation** represent the activation function.

Let us consider sample input and weights as below and try to find the result:

- input as 2 x 2 matrix **[[1, 2], [3, 4]]**
- kernel as 2 x 2 matrix **[[0.5, 0.75], [0.25, 0.5]]**
- bias value as **0**
- activation as **linear**. As we learned earlier, linear activation does nothing.

```
>>> import numpy as np

>>> input = [ [1, 2], [3, 4] ]
>>> kernel = [ [0.5, 0.75], [0.25, 0.5] ]
>>> result = np.dot(input, kernel)
>>> result
array([[1.  , 1.75],
       [2.5 , 4.25]])
>>>
```

result is the output and it will be passed into the next layer.

The output shape of the *Dense* layer will be affected by the number of neuron / units specified in the *Dense* layer. For example, if the input shape is **(8,)** and number of unit is 16, then the output shape is **(16,)**. All layer will have batch size as the first dimension and so, input shape will be represented by **(None, 8)** and the output shape as **(None, 16).** Currently, batch size is *None* as it is not set. *Batch size* is usually set during training phase.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.input_shape
(None, 8)
>>> layer_1.output_shape
(None, 16)
>>>
```

where,

- **layer_1.input_shape** returns the input shape of the layer.
- **layer_1.output_shape** returns the output shape of the layer.

The argument supported by **Dense layer** is as follows:

- ***units*** represent the number of units and it affects the output layer.
- ***activation*** represents the activation function.
- ***use_bias*** represents whether the layer uses a bias vector.
- ***kernel_initializer*** represents the initializer to be used for kernel.
- ***bias_initializer*** represents the initializer to be used for the bias vector.
- ***kernel_regularizer*** represents the regularizer function to be applied to the kernel weights matrix.
- ***bias_regularizer*** represents the regularizer function to be applied to the bias vector.
- ***activity_regularizer*** represents the regularizer function tp be applied to the output of the layer.
- ***kernel_constraint*** represent constraint function to be applied to the kernel weights matrix.
- ***bias_constraint*** represent constraint function to be applied to the bias vector.

As you have seen, there is no argument available to specify the ***input_shape*** of the input data. ***input_shape*** is a special argument, which the layer will accept only if it is designed as first layer in the model.

Also, all Keras layer has few common methods and they are as follows:

get_weights

Fetch the full list of the weights used in the layer.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.get_weights()
>>> [array([[-0.19929028,  0.4162618 ,  0.20081699, -0.25589502,  0.3612864 ,
     0.25088787, -0.47544873,  0.0321095 , -0.26070702, -0.24102116,
     0.32778358,  0.4667952 , -0.43322265, -0.14500427,  0.04341269,
    -0.34929228],
       [ 0.41898954,  0.42256463,  0.2399621 , -0.272717  , -0.37069297,
    -0.37802136,  0.11428618,  0.12749982,  0.10182762,  0.14897704,
     0.06569374,  0.15424263,  0.42638576,  0.34037888, -0.15504825,
    -0.0740819 ],
       [-0.3132702 ,  0.34885168, -0.3259498 , -0.47076607,  0.33696914,
    -0.49143505, -0.04318619, -0.11252558,  0.29669464, -0.28431225,
    -0.43165374, -0.49687648,  0.13632   , -0.21099591, -0.10608876,
    -0.13568914],
       [-0.27421212, -0.180812  ,  0.37240648,  0.25100648, -0.07199466,
    -0.23680925, -0.21271884, -0.08706653,  0.4393121 ,  0.23259485,
     0.2616762 ,  0.23966897, -0.4502542 ,  0.0058881 ,  0.14847124,
     0.08835125],
       [-0.36905527,  0.08948278, -0.19254792,  0.26783705,  0.25979865,
    -0.46963632,  0.32761025, -0.25718856,  0.48987913,  0.3588251 ,
    -0.06586111,  0.2591269 ,  0.48289275,  0.3368858 , -0.17145419,
    -0.35674667],
       [-0.32851398,  0.42289603, -0.47025883,  0.29027188, -0.0498147 ,
     0.46215963, -0.10123312,  0.23069787,  0.00844061, -0.11867595,
    -0.2602347 , -0.27917898,  0.22910392,  0.18214619, -0.40857887,
     0.2606709 ],
       [-0.19066167, -0.11464512, -0.06768692, -0.21878994, -0.2573272 ,
     0.13698077,  0.45221198,  0.10634196,  0.06784797,  0.07192957,
     0.2946936 ,  0.04968262, -0.15899467,  0.15757453, -0.1343019 ,
     0.24561536],
       [-0.04272163,  0.48315823, -0.13382411,  0.01752126, -0.1630218 ,
     0.4629662 , -0.21412933, -0.1445911 , -0.03567278, -0.20948446,
     0.15742278,  0.11139905,  0.11066687,  0.17430818,  0.36413217,
     0.19864106]], dtype=float32), array([0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0.],
      dtype=float32)]
>>>
```

- ***set_weights***: Set the weights for the layer
- ***get_config***: Get the complete configuration of the layer as an object which can be reloaded at any time.

```
config = layer_1.get_config()
```

from_config

Load the layer from the configuration object of the layer.

```
config = layer_1.get_config()
reload_layer = Dense.from_config(config)
```

input_shape

Get the input shape, if only the layer has single node.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.get_weights()
>>> layer_1.input_shape
(None, 8)
```

input

Get the input data, if only the layer has single node.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.get_weights()
>>> layer_1.input
<tf.Tensor 'dense_1_input:0' shape=(?, 8) dtype=float32>
```

- ***get_input_at***: Get the input data at the specified index, if the layer has multiple node
- ***get_input_shape_at***: Get the input shape at the specified index, if the layer has multiple node
- ***output_shape***: Get the output shape, if only the layer has single node.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.get_weights()
```

```
>>> layer_1.output_shape
(None, 16)
```

output

Get the output data, if only the layer has single node.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_1.get_weights()
>>> layer_1.output
<tf.Tensor 'dense_1/BiasAdd:0' shape=(?, 16) dtype=float32>
```

- ***get_output_at***: Get the output data at the specified index, if the layer has multiple node
- ***get_output_shape_ at***: Get the output shape at the specified index, if the layer has multiple node

Dropout Layers

Dropout is one of the important concept in the machine learning. It is used to fix the over-fitting issue. Input data may have some of the unwanted data, usually called as ***Noise***. ***Dropout*** will try to remove the noise data and thus prevent the model from over-fitting.

Dropout has three arguments and they are as follows:

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

- **rate** represent the fraction of the input unit to be dropped. It will be from 0 to 1.
- **noise_shape** represent the dimension of the shape in which the dropout to be applied. For example, the input shape is **(batch_size, timesteps, features).** Then, to apply dropout in the timesteps, **(batch_size, 1, features)** need to be specified as ***noise_shape***
- **seed** - random seed.

Flatten Layers

Flatten is used to flatten the input. For example, if flatten is applied to layer having input shape as **(batch_size, 2,2),** then the output shape of the layer will be **(batch_size, 4)**

Flatten has one argument as follows:

```
keras.layers.Flatten(data_format=None)
```

data_format is an optional argument and it is used to preserve weight ordering when switching from one data format to another data format. It accepts either **channels_last** or **channels_first** as value. **channels_last** is the default one and it identifies the input shape as **(batch_size, ..., channels)** whereas **channels_first** identifies the input shape as **(batch_size, channels, ...)**

A simple example to use ***Flatten*** layers is as follows:

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense, Flatten
>>>
>>>
>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,8))
>>> model.add(layer_1)
>>> layer_2 = Flatten()
>>> model.add(layer_2)
>>> layer_2.input_shape
(None, 8, 16)
>>> layer_2.output_shape
(None, 128)
>>>
```

where, the second layer input shape is **(None, 8, 16)** and it gets flattened into **(None, 128).**

Reshape Layers

Reshape is used to change the shape of the input. For example, if reshape with argument **(2,3)** is applied to layer having input shape as **(batch_size, 3, 2)**, then the output shape of the layer will be **(batch_size, 2, 3)**

Reshape has one argument as follows:

```
keras.layers.v(target_shape)
```

A simple example to use ***Reshape*** **layers** is as follows:

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense, Reshape
>>>
>>>
>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,8))
>>> model.add(layer_1)
>>> layer_2 = Reshape((16, 8))
>>> model.add(layer_2)
>>> layer_2.input_shape
(None, 8, 16)
>>> layer_2.output_shape
(None, 16, 8)
>>>
```

where, **(16, 8)** is set as target shape.

Permute Layers

Permute is also used to change the shape of the input using pattern. For example, if **Permute** with argument **(2, 1)** is applied to layer having input shape as **(batch_size, 3, 2)**, then the output shape of the layer will be **(batch_size, 2, 3)**

Permute has one argument as follows:

```
keras.layers.Permute(dims)
```

A simple example to use ***Permute*** layers is as follows:

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense, Permute
>>>
>>>
>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8, 8))
>>> model.add(layer_1)
>>> layer_2 = Permute((2, 1))
>>> model.add(layer_2)
>>> layer_2.input_shape
(None, 8, 16)
>>> layer_2.output_shape
(None, 16, 8)
>>>
```

where, **(2, 1)** is set as pattern.

RepeatVector Layers

RepeatVector is used to repeat the input for set number, **n** of times. For example, if ***RepeatVector*** with argument **16** is applied to layer having input shape as **(batch_size, 32)**, then the output shape of the layer will be **(batch_size, 16, 32)**

RepeatVector has one arguments and it is as follows:

```
keras.layers.RepeatVector(n)
```

A simple example to use ***RepeatVector*** layers is as follows:

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense, RepeatVector
>>>
>>>
>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape=(8,))
>>> model.add(layer_1)
>>> layer_2 = RepeatVector(16)
>>> model.add(layer_2)
>>> layer_2.input_shape
(None, 16)
>>> layer_2.output_shape
```

```
(None, 16, 16)
>>>
```

where, **16** is set as repeat times.

Lambda Layers

Lambda is used to transform the input data using an expression or function. For example, if ***Lambda*** with expression **lambda x: x ** 2** is applied to a layer, then its input data will be squared before processing.

RepeatVector has four arguments and it is as follows:

```
keras.layers.Lambda(function, output_shape=None, mask=None, arguments=None)
```

- ***function*** represent the lambda function.
- ***output_shape*** represent the shape of the transformed input.
- ***mask*** represent the mask to be applied, if any.
- ***arguments*** represent the optional argument for the lamda function as dictionary.

Convolution Layers

Keras contains a lot of layers for creating *Convolution* based ANN, popularly called as *Convolution Neural Network (CNN)*. All convolution layer will have certain properties (as listed below), which differentiate it from other layers (say *Dense* layer).

Filters: It refers the number of filters to be applied in the convolution. It affects the dimension of the output shape.

kernel size: It refers the length of the convolution window.

Strides: It refers the stride length of the convolution.

Padding: It refers the how padding needs to be done on the output of the convolution. It has three values which are as follows: -

- ***valid*** means no padding
- ***causal*** means causal convolution.
- ***same*** means the output should have same length as input and so, padding should be applied accordingly

Dilation Rate: dilation rate to be applied for dilated convolution.

Another important aspect of the convolution layer is the data format. The data format may be to two type,

channel_last: channel_last specifies that the channel data is placed as last entry. Here, *channel* refers the actual data and it will be placed in the last dimension of the input space.

For example, let us consider an input shape, **(30, 10, 128)**. Here, the value in first dimension, **30** refers the batch size, the value in second dimension, **10** refers the ***timesteps*** in temporal convolution and the value in third dimension **128** refers the actual values of the input. This is the default setting in Keras.

channel_first: channel_first is just opposite to ***channet_last***. Here, the input values are placed in the second dimension, next to batch size.

Let us see check the all the layer used for CNN provided by Keras layers in this chapter.

Conv1D

***Conv1D* layer** is used in temporal based CNN. The input shape of the *ConvID* will be in below format:

```
(batch_size, timesteps, features)
```

where,

- ***batch_size*** refers the size of the batch.
- ***timesteps*** refers the number of time steps provided in the input.
- ***features*** refer the number of features available in the input.

The output shape of the *Conv1D* is as follows:

```
(batch_size, new_steps, filters)
```

where, ***filters*** refer the number of filters specified as one of the arguments.

The signature of the *ConvID* function and its arguments with default value is as follows:

```
keras.layers.Conv1D(
    filters,
    kernel_size,
    strides=1,
    padding='valid',
    data_format='channels_last',
    dilation_rate=1,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None)
```

Conv2D

It is a convolution 2D layer. It creates a convolutional kernel with the layer input creates a tensor of outputs. input_shape refers the tuple of integers with RGB value in data_format="channels_last".

The signature of the ***Conv2D*** function and its arguments with default value is as follows:

```
keras.layers.Conv2D
  (filters,
  kernel_size,
  strides=(1, 1),
  padding='valid',
  data_format=None,
  dilation_rate=(1, 1),
  activation=None,
  use_bias=True,
  kernel_initializer='glorot_uniform',
  bias_initializer='zeros',
  kernel_regularizer=None,
  bias_regularizer=None,
  activity_regularizer=None,
  kernel_constraint=None,
  bias_constraint=None)
```

Here,

- ***strides*** refer an integer specifying the strides of the convolution along the height and width.

Pooling Layer

It is used to perform max pooling operations on temporal data. The signature of the ***MaxPooling1D*** function and its arguments with default value is as follows:

```
keras.layers.MaxPooling1D
  (pool_size=2,
  strides=None,
  padding='valid',
  data_format='channels_last')
```

Here,

- **pool_size** refers the max pooling windows.
- **strides** refer the factors for downscale.

Similarly, MaxPooling2D and MaxPooling3D are used for Max pooling operations for spatial data.

Locally connected layer

Locally connected layers are similar to Conv1D layer but the difference is Conv1D layer weights are shared but here weights are unshared. We can use different set of filters to apply different patch of input.

Locally connected layer has one arguments and it is as follows:

```
keras.layers.LocallyConnected1D(n)
```

A simple example to use *Locally connected 1D* layer is as follows:

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense,LocallyConnected1D
>>> model = Sequential()

# apply a unshared weight convolution 1-dimension of length 3 to a sequence
with
# 10 timesteps, with 16 output filters

>>> model.add(LocallyConnected1D(16, 3, input_shape=(10, 8)))

# add a new conv1d on top
>>> model.add(LocallyConnected1D(8, 3))
```

The signature of the *Locally connected 1D layer* function and its arguments with default value is as follows:

```
keras.layers.LocallyConnected1D
  (filters,
  kernel_size,
  strides=1,
  padding='valid',
  data_format=None,
  activation=None,
  use_bias=True,
  kernel_initializer='glorot_uniform',
  bias_initializer='zeros',
  kernel_regularizer=None,
  bias_regularizer=None,
  activity_regularizer=None,
  kernel_constraint=None,
  bias_constraint=None)
```

Here,

- ***kernel_initializer*** refers initializer for the kernel weights matrix
- ***kernel_regularizer*** is used to apply regularize function to the kernel weights matrix.
- ***bias_regularizer*** is used to apply regularizer function to the bias vector.
- ***activity_regularizer*** is used to apply regularizer function to the output of the layer.

Similarly, we can use 2D and 3D layers as well.

Recurrent Layer

It is used in Recurrent neural networks(RNN). It is defined as shown below:

```
keras.engine.base_layer.wrapped_fn()
```

It supports the following parameters:

- ***cell*** refers an instance.
- ***return_sequences*** return the last output in the output sequence, or the full sequence.
- ***return_state*** returns the last state in addition to the output.
- ***go_backwards*** returns a boolean result. If the value is true, then process the input sequence backwards otherwise return the reversed sequence.
- ***stateful*** refers the state for each index.
- ***unroll*** specifies whether the network to be unrolled or not.
- ***input_dim*** refers the input dimension.
- ***input_length*** refers the length of input sequence.

Merge Layer

It is used to merge a list of inputs. It supports *add()*, *subtract()*, *multiply()*, *average()*, *maximum()*, *minimum()*, *concatenate()* and *dot()* functionalities.

Adding a layer

It is used to add two layers. Syntax is defined below:

```
keras.layers.add(inputs)
```

Simple example is shown below:

```
>>> a = input1 = keras.layers.Input(shape=(16,))
>>> x1 = keras.layers.Dense(8, activation='relu')(a)
>>> a =keras.layers.Input(shape=(16,))
>>> x1 = keras.layers.Dense(8, activation='relu')(a)
>>> b =  keras.layers.Input(shape=(32,))
>>> x2 = keras.layers.Dense(8, activation='relu')(b)
>>> summ = = keras.layers.add([x1, x2])
>>> summ = keras.layers.add([x1, x2])
>>> model = keras.models.Model(inputs=[a,b],outputs=summ)
```

subtract layer

It is used to subtract two layers. The syntax is defined below:

```
keras.layers.subtract(inputs)
```

In the above example, we have created two input sequence. If you want to apply *subtract()*, then use the below coding:

```
subtract_result = keras.layers.subtract([x1, x2])
result = keras.layers.Dense(4)(subtract_result)
model = keras.models.Model(inputs=[a,b], outputs=result)
```

multiply layer

It is used to multiply two layers. Syntax is defined below:

```
keras.layers.multiply(inputs)
```

If you want to apply multiply two inputs, then you can use the below coding:

```
mul_result = keras.layers.multiply([x1, x2])
result = keras.layers.Dense(4)(mul_result)
model = keras.models.Model(inputs=[a,b], outputs=result)
```

maximum()

It is used to find the maximum value from the two inputs. syntax is defined below:

```
keras.layers.maximum(inputs)
```

minimum()

It is used to find the minimum value from the two inputs. syntax is defined below:

```
keras.layers.minimum(inputs)
```

concatenate

It is used to concatenate two inputs. It is defined below:

```
keras.layers.concatenate(inputs, axis=-1)
```

Functional interface to the Concatenate layer.

Here, ***axis*** refers to Concatenation axis.

dot

It returns the dot product from two inputs. It is defined below:

```
keras.layers.dot(inputs, axes, normalize=False)
```

Here,

- ***axes*** refer axes to perform the dot product.
- ***normalize*** determines whether dot product is needed or not.

Embedding Layer

It performs embedding operations in input layer. It is used to convert positive into dense vectors of fixed size. Its main application is in text analysis. The signature of the *Embedding layer* function and its arguments with default value is as follows,

```
keras.layers.Embedding
  (input_dim,
  output_dim,
  embeddings_initializer='uniform',
  embeddings_regularizer=None,
  activity_regularizer=None,
  embeddings_constraint=None,
  mask_zero=False,
  input_length=None)
```

Here,

- ***input_dim*** refers the input dimension.
- ***output_dim*** refers the dimension of the dense embedding.
- ***embeddings_initializer*** refers the initializer for the embeddings matrix
- ***embeddings_regularizer*** refers the regularizer function applied to the embeddings matrix.
- ***activity_regularizer*** refers the regularizer function applied to the output of the layer.
- ***embeddings_constraint*** refers the constraint function applied to the embeddings matrix
- ***mask_zero*** refers the input value should be masked or not.
- ***input_length*** refers the length of input sequence.

8. Keras — Customized Layer

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
_________________________________________________________________
```

9. Keras — Models

As learned earlier, Keras model represents the actual neural network model. Keras provides a two mode to create the model, simple and easy to use *Sequential API* as well as more flexible and advanced *Functional API*. Let us learn now to create model using both *Sequential* and *Functional* API in this chapter.

Sequential

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)
model.add(output_layer)
```

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. They are 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.

Serialize the model

Keras provides methods to serialize the model into object as well as json and load it again later. They are as follows:

- ***get_config():*** Returns the model as an object.

```
config = model.get_config()
```

- ***from_config():*** It accept the model configuration object as argument and create the model accordingly.

```
new_model = Sequential.from_config(config)
```

- ***to_json():*** Returns the model as an json object.

```
>>> json_string = model.to_json()
>>> json_string
'{"class_name": "Sequential", "config": {"name": "sequential_10", "layers":
[{"class_name": "Dense", "config": {"name": "dense_13", "trainable": true,
 "batch_input_shape": [null, 8], "dtype": "float32", "units": 32, "activation":
"linear", "use_bias": true, "kernel_initializer": {"class_name": "Vari
anceScaling", "config": {"scale": 1.0, "mode": "fan_avg", "distribution":
"uniform", "seed": null}}, "bias_initializer": {"class_name": "Zeros", "conf
ig": {}}, "kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null}}, {"
class_name": "Dense", "config": {"name": "dense_14", "trainable": true,
"dtype": "float32", "units": 64, "activation": "relu", "use_bias": true, "kern
el_initializer": {"class_name": "VarianceScaling", "config": {"scale": 1.0,
"mode": "fan_avg", "distribution": "uniform", "seed": null}}, "bias_initia
lizer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null,
"bias_regularizer": null, "activity_regularizer": null, "kernel_constraint"
: null, "bias_constraint": null}}, {"class_name": "Dense", "config": {"name":
"dense_15", "trainable": true, "dtype": "float32", "units": 8, "activati
```

```
on": "linear", "use_bias": true, "kernel_initializer": {"class_name":
"VarianceScaling", "config": {"scale": 1.0, "mode": "fan_avg", "distribution":
"
uniform", "seed": null}}, "bias_initializer": {"class_name": "Zeros", "config":
{}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_r
egularizer": null, "kernel_constraint": null, "bias_constraint": null}}]},
"keras_version": "2.2.5", "backend": "tensorflow"}'
>>>
```

- ***model_from_json():*** Accepts json representation of the model and create a new model.

```
from keras.models import model_from_json

new_model = model_from_json(json_string)
```

- ***to_yaml():*** Returns the model as a yaml string.

```
>>> yaml_string = model.to_yaml()
>>> yaml_string
'backend: tensorflow\nclass_name: Sequential\nconfig:\n  layers:\n  -
class_name: Dense\n    config:\n      activation: linear\n
activity_regular
izer: null\n      batch_input_shape: !!python/tuple\n      - null\n      - 8\n
bias_constraint: null\n      bias_initializer:\n        class_name
: Zeros\n        config: {}\n      bias_regularizer: null\n      dtype:
float32\n      kernel_constraint: null\n      kernel_initializer:\n        cla
ss_name: VarianceScaling\n        config:\n          distribution: uniform\n
mode: fan_avg\n          scale: 1.0\n          seed: null\n
 kernel_regularizer: null\n      name: dense_13\n      trainable: true\n
units: 32\n      use_bias: true\n  - class_name: Dense\n    config:\n
   activation: relu\n      activity_regularizer: null\n      bias_constraint:
null\n      bias_initializer:\n        class_name: Zeros\n        config
: {}\n      bias_regularizer: null\n      dtype: float32\n
kernel_constraint: null\n      kernel_initializer:\n        class_name:
VarianceScalin
g\n        config:\n          distribution: uniform\n          mode: fan_avg\n
scale: 1.0\n          seed: null\n      kernel_regularizer: nu
ll\n      name: dense_14\n      trainable: true\n      units: 64\n
use_bias: true\n  - class_name: Dense\n    config:\n      activation: linear\n
      activity_regularizer: null\n      bias_constraint: null\n
bias_initializer:\n        class_name: Zeros\n        config: {}\n
bias_regu
larizer: null\n      dtype: float32\n      kernel_constraint: null\n
kernel_initializer:\n        class_name: VarianceScaling\n        config:\n
     distribution: uniform\n          mode: fan_avg\n          scale: 1.0\n
seed: null\n      kernel_regularizer: null\n      name: dense
_15\n      trainable: true\n      units: 8\n      use_bias: true\n  name:
sequential_10\nkeras_version: 2.2.5\n'
>>>
```

- ***model_from_yaml():*** Accepts yaml representation of the model and create a new model.

```
from keras.models import model_from_yaml

new_model = model_from_yaml(yaml_string)
```

Summarise the model

Understanding the model is very important phase to properly use it for training and prediction purposes. Keras provides a simple method, ***summary*** to get the full information about the model and its layers.

A summary of the model created in the previous section is as follows:

```
>>> model.summary()
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 32)                288
_________________________________________________________________
dense_14 (Dense)             (None, 64)                2112
_________________________________________________________________
dense_15 (Dense)             (None, 8)                 520
=================================================================
Total params: 2,920
Trainable params: 2,920
Non-trainable params: 0
_________________________________________________________________
>>>
```

Train and Predict the model

Model provides function for training, evaluation and prediction process. They are as follows:

- ***compile:*** Configure the learning process of the model
- ***fit:*** Train the model using the training data
- ***evaluate:*** Evaluate the model using the test data
- ***predict:*** Predict the results for new input

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)
Tensor("dense_1/add:0", shape=(?, 2, 2), dtype=float32)
```

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 = Model(inputs=data,outputs=layer)

model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 2, 3)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2, 2)              8
=================================================================
Total params: 8
Trainable params: 8
Non-trainable params: 0
_________________________________________________________________
```

10. Keras — Model Compilation

Previously, we studied the basics of how to create model using Sequential and Functional API. This chapter explains about how to compile the model. The compilation is the final step in creating a model. Once the compilation is done, we can move on to training phase.

Let us learn few concepts required to better understand the compilation process.

Loss

In machine learning, **Loss** function is used to find error or deviation in the learning process. Keras requires loss function during model compilation process.

Keras provides quite a few loss function in the **losses** module and they are as follows:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- huber_loss
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity
- is_categorical_crossentropy

All above loss function accepts two arguments:

- **y_true** - true labels as tensors
- **y_pred** - prediction with same shape as **y_true**

Import the *losses* module before using loss function as specified below:

```
from keras import losses
```

Optimizer

In machine learning, **Optimization** is an important process which optimize the input weights by comparing the prediction and the loss function. Keras provides quite a few optimizer as a module, *optimizers* and they are as follows:

SGD: Stochastic gradient descent optimizer.

```
keras.optimizers.SGD(learning_rate=0.01, momentum=0.0, nesterov=False)
```

RMSprop: RMSProp optimizer.

```
keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

Adagrad: Adagrad optimizer.

```
keras.optimizers.Adagrad(learning_rate=0.01)
```

Adadelta: Adadelta optimizer.

```
keras.optimizers.Adadelta(learning_rate=1.0, rho=0.95)
```

Adam: Adam optimizer.

```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
amsgrad=False)
```

Adamax: Adamax optimizer from Adam.

```
keras.optimizers.Adamax(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

Nadam: Nesterov Adam optimizer.

```
keras.optimizers.Nadam(learning_rate=0.002, beta_1=0.9, beta_2=0.999)
```

Import the *optimizers* module before using optimizers as specified below:

```
from keras import optimizers
```

Metrics

In machine learning, **Metrics** is used to evaluate the performance of your model. It is similar to *loss* function, but not used in training process. Keras provides quite a few metrics as a module, **metrics** and they are as follows:

- accuracy
- binary_accuracy
- categorical_accuracy
- sparse_categorical_accuracy
- top_k_categorical_accuracy

- sparse_top_k_categorical_accuracy
- cosine_proximity
- clone_metric

Similar to loss function, metrics also accepts below two arguments:

- **y_true** - true labels as tensors
- **y_pred** - prediction with same shape as **y_true**

Import the *metrics* module before using metrics as specified below:

```
from keras import metrics
```

Compile the model

Keras model provides a method, **compile()** to compile the model. The argument and default value of the **compile()** method is as follows:

```
compile(optimizer, loss=None,
        metrics=None,
        loss_weights=None,
        sample_weight_mode=None,
        weighted_metrics=None,
        target_tensors=None)
```

The important arguments are as follows:

- loss function
- Optimizer
- metrics

A sample code to compile the mode is as follows:

```
from keras import losses
from keras import optimizers
from keras import metrics

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=[metrics.categorical_accuracy])
```

where,

- loss function is set as **mean_squared_error**
- optimizer is set as **sgd**
- metrics is set as **metrics.categorical_accuracy**

Model Training

Models are trained by NumPy arrays using ***fit()***. The main purpose of this fit function is used to evaluate your model on training. This can be also used for graphing model performance. It has the following syntax:

```
model.fit(X, y, epochs=, batch_size=)
```

Here,

- **X, y** - It is a tuple to evaluate your data.
- **epochs** - no of times the model is needed to be evaluated during training.
- **batch_size** - training instances.

Let us take a simple example of numpy random data to use this concept.

Create data

Let us create a random data using numpy for x and y with the help of below mentioned command:

```
import numpy as np

x_train = np.random.random((100,4,8))
y_train = np.random.random((100,10))
```

Now, create random validation data,

```
x_val = np.random.random((100,4,8))
y_val = np.random.random((100,10))
```

Create model

Let us create simple sequential model:

```
from keras.models import Sequential

model = Sequential()
```

Add layers

Create layers to add model:

```
from keras.layers import LSTM, Dense

# add a sequence of vectors of dimension 16
model.add(LSTM(16, return_sequences=True))
model.add(Dense(10, activation='softmax'))
```

compile model

Now model is defined. You can compile using the below command:

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

Apply *fit()*

Now we apply *fit()* function to train our data:

```
model.fit(x_train, y_train, batch_size=32, epochs=5, validation_data=(x_val,
y_val))
```

Create a Multi-Layer Perceptron ANN

We have learned to create, compile and train the Keras models.

Let us apply our learning and create a simple MPL based ANN.

Dataset module

Before creating a model, we need to choose a problem, need to collect the required data and convert the data to NumPy array. Once data is collected, we can prepare the model and train it by using the collected data. Data collection is one of the most difficult phase of machine learning. Keras provides a special module, *datasets* to download the online machine learning data for training purposes. It fetches the data from online server, process the data and return the data as training and test set. Let us check the data provided by Keras dataset module. The data available in the module are as follows,

- CIFAR10 small image classification
- CIFAR100 small image classification
- IMDB Movie reviews sentiment classification
- Reuters newswire topics classification
- MNIST database of handwritten digits
- Fashion-MNIST database of fashion articles
- Boston housing price regression dataset

Let us use the ***MNIST database of handwritten digits*** (or *minst*) as our input. *minst* is a collection of 60,000, 28x28 grayscale images. It contains 10 digits. It also contains 10,000 test images.

Below code can be used to load the dataset:

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

where

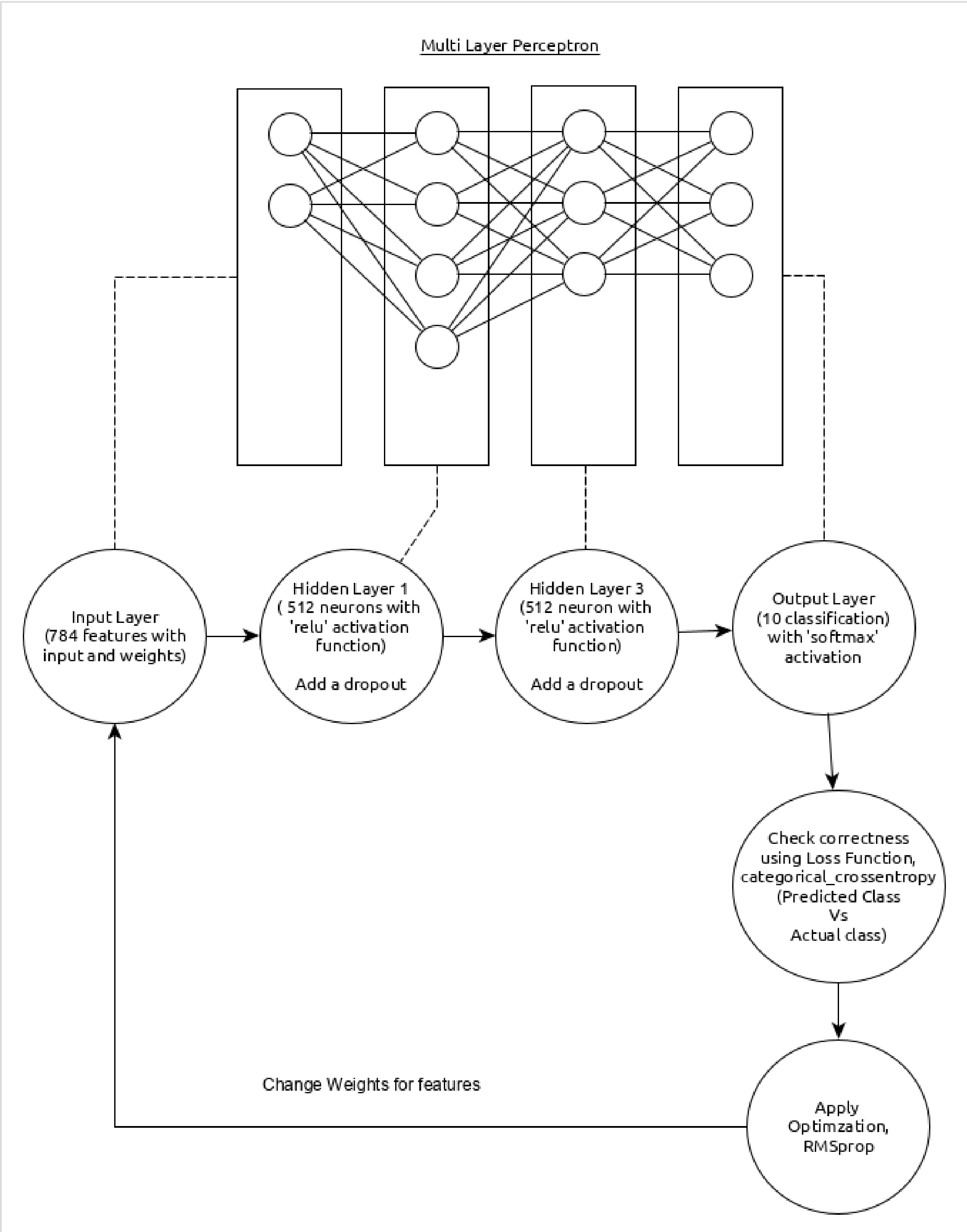
- **Line 1** imports **minst** from the keras dataset module.
- **Line 3** calls the **load_data** function, which will fetch the data from online server and return the data as 2 tuples, First tuple, **(x_train, y_train)** represent the

training data with shape, **(number_sample, 28, 28)** and its digit label with shape, **(number_samples,).** Second tuple, **(x_test, y_test)** represent test data with same shape.

Other dataset can also be fetched using similar API and every API returns similar data as well except the shape of the data. The shape of the data depends on the type of data.

Create a model

Let us choose a simple multi-layer perceptron (MLP) as represented below and try to create the model using Keras.



The core features of the model are as follows:

- Input layer consists of 784 values (28 x 28 = 784).
- First hidden layer, **Dense** consists of 512 neurons and 'relu' activation function.
- Second hidden layer, **Dropout** has 0.2 as its value.
- Third hidden layer, again Dense consists of 512 neurons and 'relu' activation function.
- Fourth hidden layer, **Dropout** has 0.2 as its value.
- Fifth and final layer consists of 10 neurons and 'softmax' activation function.

- Use **categorical_crossentropy** as loss function.
- Use **RMSprop()** as Optimizer.
- Use **accuracy** as metrics.
- Use 128 as batch size.
- Use 20 as epochs.

Step 1: Import the modules

Let us import the necessary modules.

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop
import numpy as np
```

Step 2: Load data

Let us import the mnist dataset.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Step 3: Process the data

Let us change the dataset according to our model, so that it can be feed into our model.

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

Where

- **reshape** is used to reshape the input from (28, 28) tuple to (784,)
- **to_categorical** is used to convert vector to binary matrix

Step 4: Create the model

Let us create tha actual model.

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
```

```
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

Step 5: Compile the model

Let us compile the model using selected loss function, optimizer and metrics.

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

Step 6: Train the model

Let us train the model using ***fit()*** method.

```
history = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=20,
                    verbose=1,
                    validation_data=(x_test, y_test))
```

Final thoughts

We have created the model, loaded the data and also trained the data to the model. We still need to evaluate the model and predict output for unknown input, which we learn in upcoming chapter.

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop
import numpy as np

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

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

history = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=20,
                    verbose=1,
                    validation_data=(x_test, y_test))
```

Executing the application will give the below content as output:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 7s 118us/step - loss: 0.2453 -
acc: 0.9236 - val_loss: 0.1004 - val_acc: 0.9675
Epoch 2/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.1023 -
acc: 0.9693 - val_loss: 0.0797 - val_acc: 0.9761
Epoch 3/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0744 -
acc: 0.9770 - val_loss: 0.0727 - val_acc: 0.9791
Epoch 4/20
60000/60000 [==============================] - 7s 110us/step - loss: 0.0599 -
acc: 0.9823 - val_loss: 0.0704 - val_acc: 0.9801
Epoch 5/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0504 -
acc: 0.9853 - val_loss: 0.0714 - val_acc: 0.9817
Epoch 6/20
60000/60000 [==============================] - 7s 111us/step - loss: 0.0438 -
acc: 0.9868 - val_loss: 0.0845 - val_acc: 0.9809
Epoch 7/20
60000/60000 [==============================] - 7s 114us/step - loss: 0.0391 -
acc: 0.9887 - val_loss: 0.0823 - val_acc: 0.9802
Epoch 8/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0364 -
acc: 0.9892 - val_loss: 0.0818 - val_acc: 0.9830
Epoch 9/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0308 -
acc: 0.9905 - val_loss: 0.0833 - val_acc: 0.9829
Epoch 10/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0289 -
acc: 0.9917 - val_loss: 0.0947 - val_acc: 0.9815
Epoch 11/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0279 -
acc: 0.9921 - val_loss: 0.0818 - val_acc: 0.9831
Epoch 12/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0260 -
acc: 0.9927 - val_loss: 0.0945 - val_acc: 0.9819
Epoch 13/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0257 -
acc: 0.9931 - val_loss: 0.0952 - val_acc: 0.9836
Epoch 14/20
```

```
60000/60000 [==============================] - 7s 112us/step - loss: 0.0229 -
acc: 0.9937 - val_loss: 0.0924 - val_acc: 0.9832
Epoch 15/20
60000/60000 [==============================] - 7s 115us/step - loss: 0.0235 -
acc: 0.9937 - val_loss: 0.1004 - val_acc: 0.9823
Epoch 16/20
60000/60000 [==============================] - 7s 113us/step - loss: 0.0214 -
acc: 0.9941 - val_loss: 0.0991 - val_acc: 0.9847
Epoch 17/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0219 -
acc: 0.9943 - val_loss: 0.1044 - val_acc: 0.9837
Epoch 18/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0190 -
acc: 0.9952 - val_loss: 0.1129 - val_acc: 0.9836
Epoch 19/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0197 -
acc: 0.9953 - val_loss: 0.0981 - val_acc: 0.9841
Epoch 20/20
60000/60000 [==============================] - 7s 112us/step - loss: 0.0198 -
acc: 0.9950 - val_loss: 0.1215 - val_acc: 0.9828
```

11. Keras — Model Evaluation and Model Prediction

This chapter deals with the model evaluation and model prediction in Keras.

Let us begin by understanding the model evaluation.

Model Evaluation

Evaluation is a process during development of the model to check whether the model is best fit for the given problem and corresponding data. Keras model provides a function, evaluate which does the evaluation of the model. It has three main arguments,

- Test data
- Test data label
- verbose - true or false

Let us evaluate the model, which we created in the previous chapter using test data.

```
score = model.evaluate(x_test, y_test, verbose=0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Executing the above code will output the below information:

```
0
```

The test accuracy is 98.28%. We have created a best model to identify the handwriting digits. On the positive side, we can still scope to improve our model.

Model Prediction

Prediction is the final step and our expected outcome of the model generation. Keras provides a method, *predict* to get the prediction of the trained model. The signature of the *predict* method is as follows,

```
predict(x,
    batch_size=None,
    verbose=0,
    steps=None,
    callbacks=None,
    max_queue_size=10, workers=1, use_multiprocessing=False)
```

Here, all arguments are optional except the first argument, which refers the unknown input data. The shape should be maintained to get the proper prediction.

Let us do prediction for our MPL model created in previous chapter using below code:

```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=1)[:5]
label = np.argmax(y_test,axis=1)[:5]

print(pred)
print(label)
```

Here,

- **Line 1** call the predict function using test data.
- **Line 2** gets the first five prediction
- **Line 3** gets the first five labels of the test data.
- **Line 5 - 6** prints the prediction and actual label.

The output of the above application is as follows:

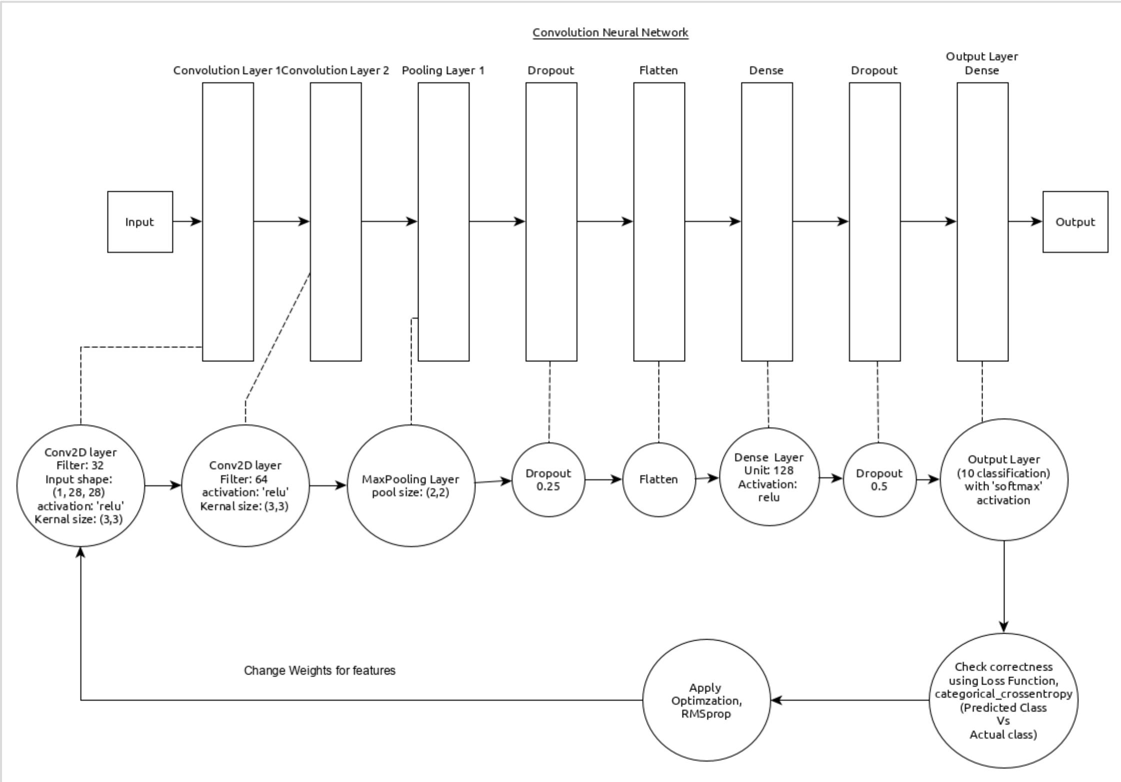
```
[7 2 1 0 4]
[7 2 1 0 4]
```

The output of both array is identical and it indicate that our model predicts correctly the first five images.

12. Keras — Convolution Neural Network

Let us modify the model from MPL to **Convolution Neural Network (CNN)** for our earlier digit identification problem.

CNN can be represented as below:



The core features of the model are as follows:

- Input layer consists of (1, 8, 28) values.
- First layer, ***Conv2D*** consists of 32 filters and 'relu' activation function with kernel size, (3,3).
- Second layer, ***Conv2D*** consists of 64 filters and 'relu' activation function with kernel size, (3,3).
- Thrid layer, ***MaxPooling*** has pool size of (2, 2).
- Fourth layer, ***Dropout*** has 0.25 as its value.
- Fifth layer, ***Flatten*** is used to flatten all its input into single dimension.
- Sixth layer, ***Dense*** consists of 128 neurons and 'relu' activation function.
- Seventh layer, ***Dropout*** has 0.5 as its value.
- Eighth and final layer consists of 10 neurons and 'softmax' activation function.
- Use **categorical_crossentropy** as loss function.
- Use **Adadelta()** as Optimizer.

- Use **accuracy** as metrics.
- Use 128 as batch size.
- Use 20 as epochs.

Step 1: Import the modules

Let us import the necessary modules.

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import numpy as np
```

Step 2: Load data

Let us import the mnist dataset.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Step 3: Process the data

Let us change the dataset according to our model, so that it can be feed into our model.

```
img_rows, img_cols = 28, 28

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

The data processing is similar to MPL model except the shape of the input data and image format configuration.

Step 4: Create the model

Let us create tha actual model.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Step 5: Compile the model

Let us compile the model using selected loss function, optimizer and metrics.

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Step 6: Train the model

Let us train the model using **fit()** method.

```
model.fit(x_train, y_train,
          batch_size=128,
          epochs=12,
          verbose=1,
          validation_data=(x_test, y_test))
```

Executing the application will output the below information:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 84s 1ms/step - loss: 0.2687 -
acc: 0.9173 - val_loss: 0.0549 - val_acc: 0.9827
Epoch 2/12
60000/60000 [==============================] - 86s 1ms/step - loss: 0.0899 -
acc: 0.9737 - val_loss: 0.0452 - val_acc: 0.9845
Epoch 3/12
60000/60000 [==============================] - 83s 1ms/step - loss: 0.0666 -
acc: 0.9804 - val_loss: 0.0362 - val_acc: 0.9879
Epoch 4/12
60000/60000 [==============================] - 81s 1ms/step - loss: 0.0564 -
acc: 0.9830 - val_loss: 0.0336 - val_acc: 0.9890
Epoch 5/12
60000/60000 [==============================] - 86s 1ms/step - loss: 0.0472 -
acc: 0.9861 - val_loss: 0.0312 - val_acc: 0.9901
Epoch 6/12
60000/60000 [==============================] - 83s 1ms/step - loss: 0.0414 -
acc: 0.9877 - val_loss: 0.0306 - val_acc: 0.9902
Epoch 7/12
60000/60000 [==============================] - 89s 1ms/step - loss: 0.0375 -
```

```
acc: 0.9883 - val_loss: 0.0281 - val_acc: 0.9906
Epoch 8/12
60000/60000 [==============================] - 91s 2ms/step - loss: 0.0339 -
acc: 0.9893 - val_loss: 0.0280 - val_acc: 0.9912
Epoch 9/12
60000/60000 [==============================] - 89s 1ms/step - loss: 0.0325 -
acc: 0.9901 - val_loss: 0.0260 - val_acc: 0.9909
Epoch 10/12
60000/60000 [==============================] - 89s 1ms/step - loss: 0.0284 -
acc: 0.9910 - val_loss: 0.0250 - val_acc: 0.9919
Epoch 11/12
60000/60000 [==============================] - 86s 1ms/step - loss: 0.0287 -
acc: 0.9907 - val_loss: 0.0264 - val_acc: 0.9916
Epoch 12/12
60000/60000 [==============================] - 86s 1ms/step - loss: 0.0265 -
acc: 0.9920 - val_loss: 0.0249 - val_acc: 0.9922
```

Step 7: Evaluate the model

Let us evaluate the model using test data.

```
score = model.evaluate(x_test, y_test, verbose=0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Executing the above code will output the below information:

```
Test loss: 0.024936060590433316
Test accuracy: 0.9922
```

The test accuracy is 99.22%. We have created a best model to identify the handwriting digits.

Step 8: Predict

Finally, predict the digit from images as below:

```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=1)[:5]
label = np.argmax(y_test,axis=1)[:5]

print(pred)
print(label)
```

The output of the above application is as follows:

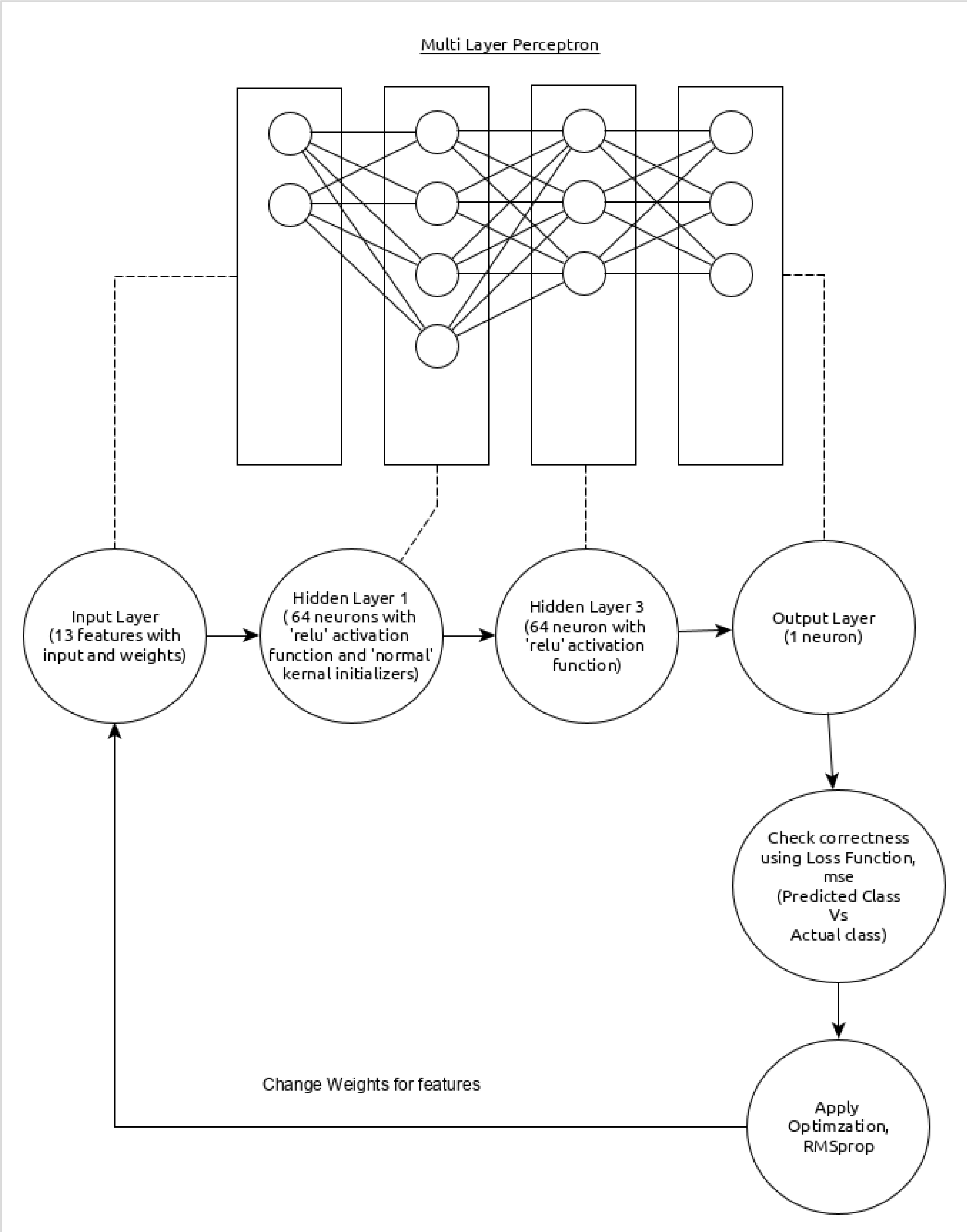
```
[7 2 1 0 4]
[7 2 1 0 4]
```

The output of both array is identical and it indicate our model correctly predicts the first five images.

13. Keras — Regression Prediction using MPL

In this chapter, let us write a simple MPL based ANN to do regression prediction. Till now, we have only done the classification based prediction. Now, we will try to predict the next possible value by analyzing the previous (continuous) values and its influencing factors.

The Regression MPL can be represented as below:



The core features of the model are as follows:

- Input layer consists of (13,) values.
- First layer, *Dense* consists of 64 units and 'relu' activation function with 'normal' kernel initializer.
- Second layer, *Dense* consists of 64 units and 'relu' activation function.
- Output layer, *Dense* consists of 1 unit.
- Use **mse** as loss function.
- Use **RMSprop** as Optimizer.

- Use **accuracy** as metrics.
- Use 128 as batch size.
- Use 500 as epochs.

Step 1: Import the modules

Let us import the necessary modules.

```
import keras
from keras.datasets import boston_housing

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop

from keras.callbacks import EarlyStopping

from sklearn import preprocessing
from sklearn.preprocessing import scale
```

Step 2: Load data

Let us import the Boston housing dataset.

```
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

Here,

boston_housing is a dataset provided by Keras. It represents a collection of housing information in Boston area, each having 13 features.

Step 3: Process the data

Let us change the dataset according to our model, so that, we can feed into our model. The data can be changed using below code:

```
x_train_scaled = preprocessing.scale(x_train)
scaler = preprocessing.StandardScaler().fit(x_train)
x_test_scaled = scaler.transform(x_test)
```

Here, we have normalized the training data using **sklearn.preprocessing.scale** function. **preprocessing.StandardScaler().fit** function returns a scalar with the normalized mean and standard deviation of the training data, which we can apply to the test data using **scalar.transform** function. This will normalize the test data as well with the same setting as that of training data.

Step 4: Create the model

Let us create the actual model.

```
model = Sequential()
model.add(Dense(64, kernel_initializer='normal', activation='relu',
```

```
input_shape=(13,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(1))
```

Step 5: Compile the model

Let us compile the model using selected loss function, optimizer and metrics.

```
model.compile(loss='mse',
        optimizer=RMSprop(),
        metrics=['mean_absolute_error'])
```

Step 6: Train the model

Let us train the model using **fit()** method.

```
history = model.fit(x_train_scaled, y_train,
                    batch_size=128,
                    epochs=500,
                    verbose=1,
                    validation_split = 0.2,
                    callbacks = [EarlyStopping(monitor = 'val_loss', patience =
20)])
```

Here, we have used callback function, **EarlyStopping.** The purpose of this callback is to monitor the loss value during each epoch and compare it with previous epoch loss value to find the improvement in the training. If there is no improvement for the **patience** times, then the whole process will be stopped.

Executing the application will give the below information as output:

```
Train on 323 samples, validate on 81 samples
Epoch 1/500
2019-09-24 01:07:03.889046: I
tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports
instructions that this TensorFlow binary was not co
mpiled to use: AVX2
323/323 [==============================] - 0s 515us/step - loss: 562.3129 -
mean_absolute_error: 21.8575 - val_loss: 621.6523 - val_mean_absolute_erro
r: 23.1730
Epoch 2/500
323/323 [==============================] - 0s 11us/step - loss: 545.1666 -
mean_absolute_error: 21.4887 - val_loss: 605.1341 - val_mean_absolute_error
: 22.8293
Epoch 3/500
323/323 [==============================] - 0s 12us/step - loss: 528.9944 -
mean_absolute_error: 21.1328 - val_loss: 588.6594 - val_mean_absolute_error
: 22.4799
Epoch 4/500
323/323 [==============================] - 0s 12us/step - loss: 512.2739 -
mean_absolute_error: 20.7658 - val_loss: 570.3772 - val_mean_absolute_error
: 22.0853
Epoch 5/500
```

```
323/323 [==============================] - 0s 9us/step - loss: 493.9775 -
mean_absolute_error: 20.3506 - val_loss: 550.9548 - val_mean_absolute_error:
 21.6547
..........
..........
..........
Epoch 143/500
323/323 [==============================] - 0s 15us/step - loss: 8.1004 -
mean_absolute_error: 2.0002 - val_loss: 14.6286 - val_mean_absolute_error: 2.
5904
Epoch 144/500
323/323 [==============================] - 0s 19us/step - loss: 8.0300 -
mean_absolute_error: 1.9683 - val_loss: 14.5949 - val_mean_absolute_error: 2.
5843
Epoch 145/500
323/323 [==============================] - 0s 12us/step - loss: 7.8704 -
mean_absolute_error: 1.9313 - val_loss: 14.3770 - val_mean_absolute_error: 2.
4996
```

Step 7: Evaluate the model

Let us evaluate the model using test data.

```
score = model.evaluate(x_test_scaled, y_test, verbose=0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Executing the above code will output the below information:

```
Test loss: 21.928471583946077
Test accuracy: 2.9599233234629914
```

Step 8: Predict

Finally, predict using test data as below:

```
prediction = model.predict(x_test_scaled)
print(prediction.flatten())
print(y_test)
```

The output of the above application is as follows:

```
[ 7.5612316 17.583357  21.09344   31.859276  25.055613  18.673872
 26.600405  22.403967  19.060272  22.264952  17.4191    17.00466
 15.58924   41.624374  20.220217  18.985565  26.419338  19.837091
 19.946192  36.43445   12.278508  16.330965  20.701359  14.345301
 21.741161  25.050423  31.046402  27.738455   9.959419  20.93039
 20.069063  14.518344  33.20235   24.735163  18.7274     9.148898
 15.781284  18.556862  18.692865  26.045074  27.954073  28.106823
 15.272034  40.879818  29.33896   23.714525  26.427515  16.483374
 22.518442  22.425386  33.94826   18.831465  13.2501955 15.537227
 34.639984  27.468002  13.474407  48.134598  34.39617   22.85031
```

```
 24.042334  17.747198  14.7837715 18.187277  23.655672  22.364983
 13.858193  22.710032  14.371148   7.1272087 35.960033  28.247292
 25.3014    14.477208  25.306196  17.891165  20.193708  23.585173
 34.690193  12.200583  20.102983  38.45882   14.741723  14.408362
 17.67158   18.418497  21.151712  21.157492  22.693687  29.809034
 19.366991  20.072294  25.880817  40.814568  34.64087   19.43741
 36.2591    50.73806   26.968863  43.91787   32.54908   20.248306 ]
[ 7.2 18.8 19.  27.  22.2 24.5 31.2 22.9 20.5 23.2 18.6 14.5 17.8 50.
 20.8 24.3 24.2 19.8 19.1 22.7 12.  10.2 20.  18.5 20.9 23.  27.5 30.1
  9.5 22.  21.2 14.1 33.1 23.4 20.1  7.4 15.4 23.8 20.1 24.5 33.  28.4
 14.1 46.7 32.5 29.6 28.4 19.8 20.2 25.  35.4 20.3  9.7 14.5 34.9 26.6
  7.2 50.  32.4 21.6 29.8 13.1 27.5 21.2 23.1 21.9 13.  23.2  8.1  5.6
 21.7 29.6 19.6  7.  26.4 18.9 20.9 28.1 35.4 10.2 24.3 43.1 17.6 15.4
 16.2 27.1 21.4 21.5 22.4 25.  16.6 18.6 22.  42.8 35.1 21.5 36.  21.9
 24.1 50.  26.7 25. ]
```

The output of both array have around 10-30% difference and it indicate our model predicts with reasonable range.

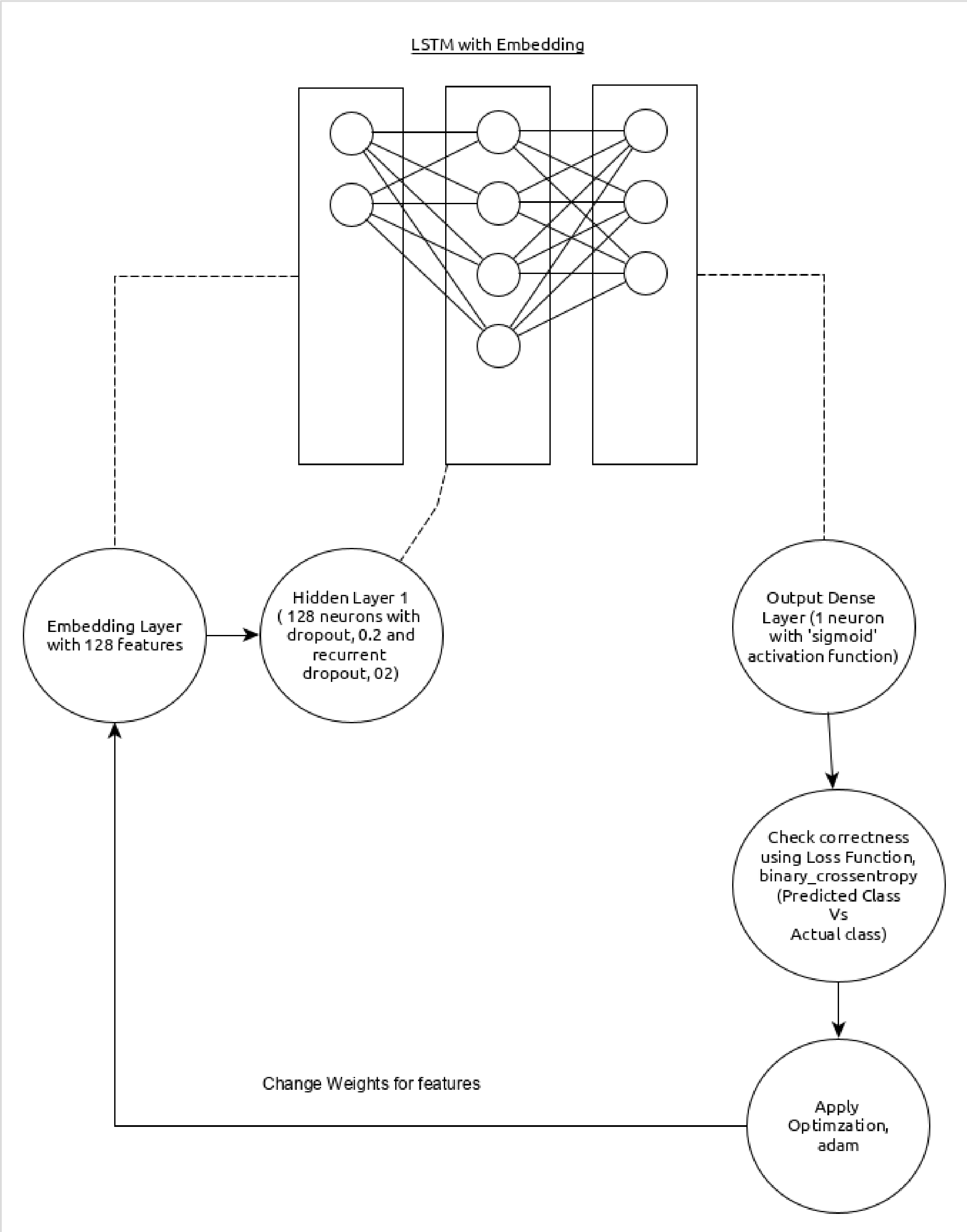
14. Keras — Time Series Prediction using LSTM RNN

In this chapter, let us write a simple Long Short Term Memory (LSTM) based RNN to do sequence analysis. A sequence is a set of values where each value corresponds to a particular instance of time. Let us consider a simple example of reading a sentence. Reading and understanding a sentence involves reading the word in the given order and trying to understand each word and its meaning in the given context and finally understanding the sentence in a positive or negative sentiment.

Here, the words are considered as values, and first value corresponds to first word, second value corresponds to second word, etc., and the order will be strictly maintained. **Sequence Analysis** is used frequently in natural language processing to find the sentiment analysis of the given text.

Let us create a LSTM model to analyze the IMDB movie reviews and find its positive/negative sentiment.

The model for the sequence analysis can be represented as below:



The core features of the model are as follows:

- Input layer using Embedding layer with 128 features.
- First layer, *Dense* consists of 128 units with normal dropout and recurrent dropout set to 0.2.
- Output layer, *Dense* consists of 1 unit and 'sigmoid' activation function.
- Use **binary_crossentropy** as loss function.
- Use **adam** as Optimizer.
- Use **accuracy** as metrics.

- Use 32 as batch size.
- Use 15 as epochs.
- Use 80 as the maximum length of the word.
- Use 2000 as the maximum number of word in a given sentence.

Step 1: Import the modules

Let us import the necessary modules.

```
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb
```

Step 2: Load data

Let us import the imdb dataset.

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=2000)
```

Here,

- **imdb** is a dataset provided by Keras. It represents a collection of movies and its reviews.
- **num_words** represent the maximum number of words in the review.

Step 3: Process the data

Let us change the dataset according to our model, so that it can be fed into our model. The data can be changed using the below code:

```
x_train = sequence.pad_sequences(x_train, maxlen=80)
x_test = sequence.pad_sequences(x_test, maxlen=80)
```

Here,

sequence.pad_sequences convert the list of input data with shape, **(data)** into 2D NumPy array of shape **(data, timesteps).** Basically, it adds timesteps concept into the given data. It generates the timesteps of length, **maxlen**.

Step 4: Create the model

Let us create the actual model.

```
model = Sequential()
model.add(Embedding(2000, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
```

Here,

We have used **Embedding layer** as input layer and then added the LSTM layer. Finally, a **Dense layer** is used as output layer.

Step 5: Compile the model

Let us compile the model using selected loss function, optimizer and metrics.

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

Step 6: Train the model

Let us train the model using **fit()** method.

```
model.fit(x_train, y_train,
          batch_size=32,
          epochs=15,
          validation_data=(x_test, y_test))
```

Executing the application will output the below information:

```
Epoch 1/15
2019-09-24 01:19:01.151247: I
tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports
instructions that this TensorFlow binary was not co
mpiled to use: AVX2
25000/25000 [==============================] - 101s 4ms/step - loss: 0.4707 -
acc: 0.7716 - val_loss: 0.3769 - val_acc: 0.8349
Epoch 2/15
25000/25000 [==============================] - 95s 4ms/step - loss: 0.3058 -
acc: 0.8756 - val_loss: 0.3763 - val_acc: 0.8350
Epoch 3/15
25000/25000 [==============================] - 91s 4ms/step - loss: 0.2100 -
acc: 0.9178 - val_loss: 0.5065 - val_acc: 0.8110
Epoch 4/15
25000/25000 [==============================] - 90s 4ms/step - loss: 0.1394 -
acc: 0.9495 - val_loss: 0.6046 - val_acc: 0.8146
Epoch 5/15
25000/25000 [==============================] - 90s 4ms/step - loss: 0.0973 -
acc: 0.9652 - val_loss: 0.5969 - val_acc: 0.8147
Epoch 6/15
25000/25000 [==============================] - 98s 4ms/step - loss: 0.0759 -
acc: 0.9730 - val_loss: 0.6368 - val_acc: 0.8208
Epoch 7/15
25000/25000 [==============================] - 95s 4ms/step - loss: 0.0578 -
acc: 0.9811 - val_loss: 0.6657 - val_acc: 0.8184
Epoch 8/15
25000/25000 [==============================] - 97s 4ms/step - loss: 0.0448 -
acc: 0.9850 - val_loss: 0.7452 - val_acc: 0.8136
Epoch 9/15
25000/25000 [==============================] - 95s 4ms/step - loss: 0.0324 -
acc: 0.9894 - val_loss: 0.7616 - val_acc: 0.8162
```

```
Epoch 10/15
25000/25000 [==============================] - 100s 4ms/step - loss: 0.0247 -
acc: 0.9922 - val_loss: 0.9654 - val_acc: 0.8148
Epoch 11/15
25000/25000 [==============================] - 99s 4ms/step - loss: 0.0169 -
acc: 0.9946 - val_loss: 1.0013 - val_acc: 0.8104
Epoch 12/15
25000/25000 [==============================] - 90s 4ms/step - loss: 0.0154 -
acc: 0.9948 - val_loss: 1.0316 - val_acc: 0.8100
Epoch 13/15
25000/25000 [==============================] - 89s 4ms/step - loss: 0.0113 -
acc: 0.9963 - val_loss: 1.1138 - val_acc: 0.8108
Epoch 14/15
25000/25000 [==============================] - 89s 4ms/step - loss: 0.0106 -
acc: 0.9971 - val_loss: 1.0538 - val_acc: 0.8102
Epoch 15/15
25000/25000 [==============================] - 89s 4ms/step - loss: 0.0090 -
acc: 0.9972 - val_loss: 1.1453 - val_acc: 0.8129
25000/25000 [==============================] - 10s 390us/step
```

Step 7: Evaluate the model

Let us evaluate the model using test data.

```
score, acc = model.evaluate(x_test, y_test,
                            batch_size=32)
print('Test score:', score)
print('Test accuracy:', acc)
```

Executing the above code will output the below information:

```
Test score: 1.145306069601178
Test accuracy: 0.81292
```

15. Keras — Applications

Keras applications module is used to provide pre-trained model for deep neural networks. Keras models are used for prediction, feature extraction and fine tuning. This chapter explains about Keras applications in detail.

Pre-trained models

Trained model consists of two parts model Architecture and model Weights. Model weights are large file so we have to download and extract the feature from ImageNet database. Some of the popular pre-trained models are listed below,

- ResNet
- VGG16
- MobileNet
- InceptionResNetV2
- InceptionV3

Loading a model

Keras pre-trained models can be easily loaded as specified below:

```
import keras

import numpy as np

from keras.applications import vgg16, inception_v3, resnet50, mobilenet

#Load the VGG model
vgg_model = vgg16.VGG16(weights='imagenet')

#Load the Inception_V3 model
inception_model = inception_v3.InceptionV3(weights='imagenet')

#Load the ResNet50 model
resnet_model = resnet50.ResNet50(weights='imagenet')

#Load the MobileNet model
mobilenet_model = mobilenet.MobileNet(weights='imagenet')
```

Once the model is loaded, we can immediately use it for prediction purpose. Let us check each pre-trained model in the upcoming chapters.

16. Keras — Real Time Prediction using *ResNet* Model

ResNet is a pre-trained model. It is trained using *ImageNet*. ResNet model weights pre-trained on *ImageNet*. It has the following syntax:

```
keras.applications.resnet.ResNet50
  (include_top=True,
  weights='imagenet',
  input_tensor=None,
  input_shape=None,
  pooling=None,
  classes=1000)
```

Here,

- ***include_top*** refers the fully-connected layer at the top of the network.
- ***weights*** refer pre-training on ImageNet.
- ***input_tensor*** refers optional Keras tensor to use as image input for the model.
- ***input_shape*** refers optional shape tuple. The default input size for this model is 224x224.
- ***classes*** refer optional number of classes to classify images.

Let us understand the model by writing a simple example:

Step1: import the modules

Let us load the necessary modules as specified below:

```
>>> import PIL
>>> from keras.preprocessing.image import load_img
>>> from keras.preprocessing.image import img_to_array
>>> from keras.applications.imagenet_utils import decode_predictions
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from keras.applications.resnet50 import ResNet50
>>> from keras.applications import resnet50
```

Step2: Select an input

Let us choose an input image, ***Lotus*** as specified below:

```
>>> filename = 'banana.jpg'

>>> ## load an image in PIL format
>>> original = load_img(filename, target_size=(224, 224))

>>> print('PIL image size',original.size)
```

```
PIL image size (224, 224)
>>> plt.imshow(original)
<matplotlib.image.AxesImage object at 0x1304756d8>
>>> plt.show()
```

Here, we have loaded an image **(banana.jpg)** and displayed it.

Step 3: Convert images into NumPy array

Let us convert our input, **Banana** into NumPy array, so that it can be passed into the model for the purpose of prediction.

```
>>> #convert the PIL image to a numpy array
>>> numpy_image = img_to_array(original)

>>> plt.imshow(np.uint8(numpy_image))
<matplotlib.image.AxesImage object at 0x130475ac8>

>>> print('numpy array size',numpy_image.shape)
numpy array size (224, 224, 3)

>>> # Convert the image / images into batch format
>>> image_batch = np.expand_dims(numpy_image, axis=0)

>>> print('image batch size', image_batch.shape)
image batch size (1, 224, 224, 3)
>>>
```

Step4: Model prediction

Let us feed our input into the model to get the predictions.

```
# prepare the image for the resnet50 model
>>>
>>> processed_image = resnet50.preprocess_input(image_batch.copy())

>>> # create resnet model
>>> resnet_model = resnet50.ResNet50(weights='imagenet')
>>> Downloading data from https://github.com/fchollet/deep-learning-
models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102858752/102853048 [==============================] - 33s 0us/step

>>> # get the predicted probabilities for each class
>>> predictions = resnet_model.predict(processed_image)

>>> # convert the probabilities to class labels
>>> label = decode_predictions(predictions)
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_inde
x.json
40960/35363 [==================================] - 0s 0us/step

>>> print(label)
```

Output

```
[[('n07753592', 'banana', 0.99229723), ('n03532672', 'hook', 0.0014551596),
('n03970156', 'plunger', 0.0010738898), ('n07753113', 'fig', 0.0009359837)
, ('n03109150', 'corkscrew', 0.00028538404)]]
```

Here, the model predicted the images as banana correctly.

17. Keras — Pre-Trained Models

In this chapter, we will learn about the pre-trained models in Keras. Let us begin with VGG16.

VGG16

VGG16 is another pre-trained model. It is also trained using *ImageNet*. The syntax to load the model is as follows:

```
keras.applications.vgg16.VGG16(include_top=True,
                 weights='imagenet',
                 input_tensor=None,
                 input_shape=None,
                 pooling=None,
                 classes=1000)
```

The default input size for this model is 224x224.

MobileNetV2

MobileNetV2 is another pre-trained model. It is also trained uing **ImageNet**.

The syntax to load the model is as follows:

```
keras.applications.mobilenet_v2.MobileNetV2
  (input_shape=None,
  alpha=1.0,
  include_top=True,
  weights='imagenet',
  input_tensor=None,
  pooling=None,
  classes=1000)
```

Here,

alpha controls the width of the network. If the value is below 1, decreases the number of filters in each layer. If the value is above 1, increases the number of filters in each layer. If alpha = 1, default number of filters from the paper are used at each layer.

The default input size for this model is **224x224**.

InceptionResNetV2

InceptionResNetV2 is another pre-trained model. It is also trained using **ImageNet**. The syntax to load the model is as follows:

```
keras.applications.inception_resnet_v2.InceptionResNetV2
    (include_top=True, weights='imagenet',
```

```
        input_tensor=None, input_shape=None,
        pooling=None, classes=1000)
```

This model and can be built both with 'channels_first' data format (channels, height, width) or 'channels_last' data format (height, width, channels).

The default input size for this model is **299x299**.

InceptionV3

InceptionV3 is another pre-trained model. It is also trained uing **ImageNet.** The syntax to load the model is as follows:

```
keras.applications.inception_v3.InceptionV3
  (include_top=True,
  weights='imagenet',
  input_tensor=None,
  input_shape=None,
  pooling=None, classes=1000)
```

Here,

The default input size for this model is **299x299**.

Conclusion

Keras is very simple, extensible and easy to implement neural network API, which can be used to build deep learning applications with high level abstraction. Keras is an optimal choice for deep leaning models.