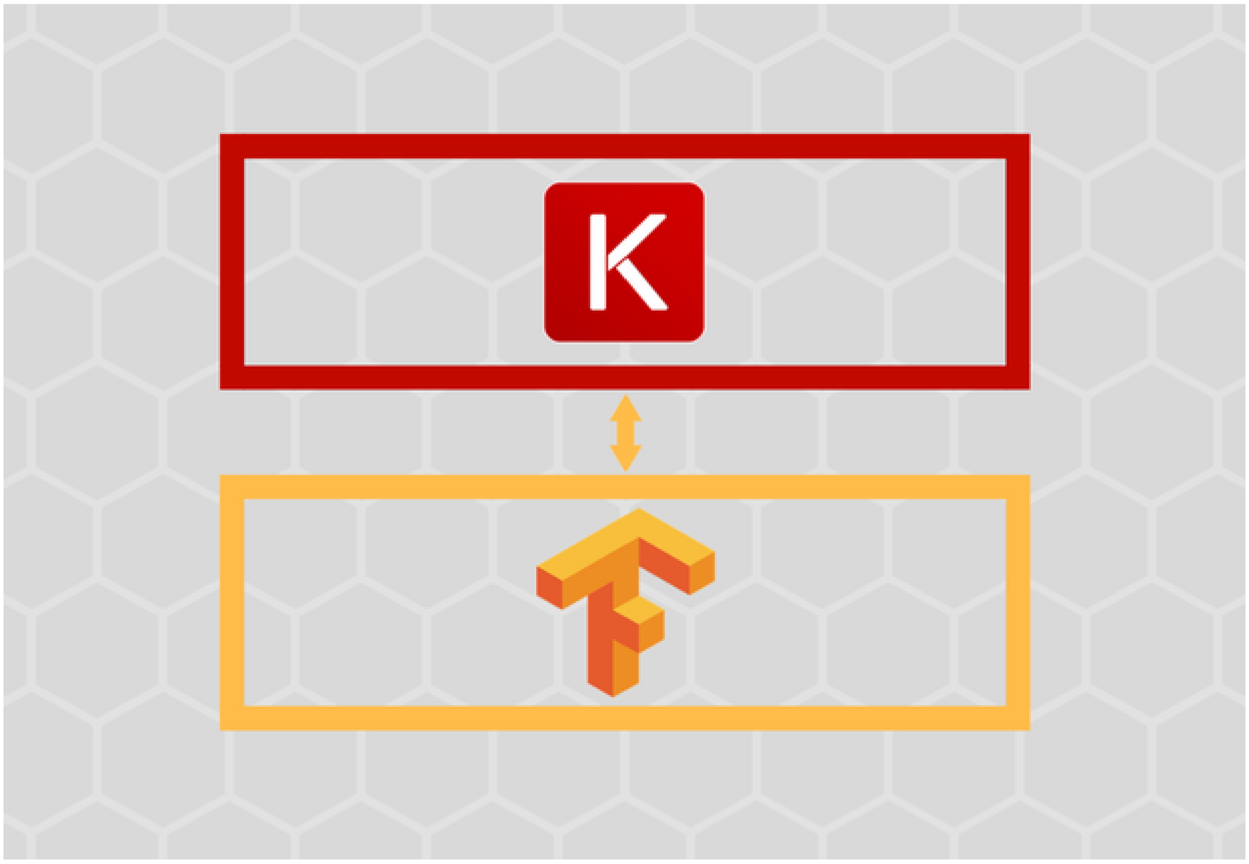
RUBIK'S CODE Freedom. Wisdom. Excellence.

Implementing Simple Neural Network using Keras – With Python Example

FEBRUARY 12, 2018 — 6 COMMENTS

Code that accompanies this article can be downloaded **here**.

Back in 2015. Google released TensorFlow, the library that will change the field of Neural Networks and eventually make it mainstream. Not only that TensorFlow became popular for developing Neural Networks, it also enabled higher-level APIs to run on top of it. One of those APIs is Keras. Keras is written in Python and it is not supporting only TensorFlow. It is capable of running on top of CNTK and Theano. In this article, we are going to use it only in combination with TensorFlow, so if you need help installing TensorFlow or learning a bit about it you can check my **previous article**. There are many benefits of using Keras, and one of the main ones is certainly user-friendliness. API is easily understandable and pretty straight-forward. Another benefit is modularity. A Neural Network (model) can be observed either as a sequence or a graph of standalone, loosely coupled and fully-configurable modules. Finally, Keras is easily extendable.



Installation and Setup

As mentioned before, Keras is running on top of TensorFlow. So, in order for this library to work, you first need to **install TensorFlow**. Another thing I need to mention is that for the purposes of this article, I am using Windows 10 and **Python 3.6**. Also, I am using Spyder IDE for the development so examples in this article may variate for other operating systems and platforms. Since Keras is a Python library installation of it is pretty standard. You can use “native pip” and install it using this command:

```
pip install keras
```

Or if you are using Anaconda you can install Keras by issuing the command:

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

Alternatively, the installation process can be done by using Github source. Firstly, you would have to clone the code from the repository:

```
git clone https://github.com/keras-team/keras.git
```

After that, you need to position the terminal in that folder and run the install command:

```
python setup.py install
```

Sequential Model and Keras Layers

One of the major points for using Keras is that it is one user-friendly API. It has two types of models:

- Sequential model
- Model class used with functional API

Sequential model is probably the most used feature of Keras. Essentially it represents the array of Keras Layers. It is convenient for the fast building of different types of Neural Networks, just by adding layers to it. There are many types of Keras Layers, too. The most basic one and the one we are going to use in this article is called *Dense.* It has many options for setting the inputs, activation functions and so on. Apart from *Dense,* Keras API provides different types of layers for Convolutional Neural Networks, Recurrent Neural Networks, etc. This is out of the scope of this post, but we will cover it in fruther posts. So, let's see how one can build a Neural Network using *Sequential* and *Dense.*

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

model = Sequential()
```

```
model.add(Dense(3, input_dim=2, activation='relu'))
model.add(Dense(1, activation='softmax'))
```

sequential_dense_example.py hosted with ❤ by **GitHub** **view raw**

In this sample, we first imported the *Sequential* and *Dense* from Keras. Than we instantiated one object of the *Sequential* class. After that, we added one layer to the Neural Network using function *add* and *Dense* class. The first parameter in the *Dense* constructor is used to define a number of neurons in that layer. What is specific about this layer is that we used *input_dim* parameter. By doing so, we added additional input layer to our network with the number of neurons defined in *input_dim* parameter. Basically, by this one call, we added two layers. First one is input layer with two neurons, and the second one is the hidden layer with three neurons.

Another important parameter, as you may notice, is *activation* parameter. Using this parameter we define **activation function** for all neurons in a specific layer. Here we used *'relu'* value, which indicates that neurons in this layer will use **Rectifier activation function**. Finally, we call *add* method of the *Sequential* object once again and add another layer. Because we are not using *input_dim* parameter one layer will be added, and since it is the last layer we are adding to our Neural Network it will also be the output layer of the network.

Iris Data Set Classification Problem

Like in the **previous article,** we will use Iris Data Set Classification Problem for this demonstration. Iris Data Set is famous dataset in the world of pattern recognition and it is considered to be "Hello World" example for machine learning classification problems. It was first introduced by Ronald Fisher, British statistician and botanist, back in 1936. In his paper *The use of multiple measurements in taxonomic problems,* he used data collected for three different classes of Iris plant: *Iris setosa*, *Iris virginica,* and *Iris versicolor*.

This dataset contains 50 instances for each class. What is interesting about it is that first class is linearly separable from the other two, but the latter two are not linearly separable from each other. Each instance has five attributes:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm
- Class (*Iris setosa*, *Iris virginica, Iris versicolor*)

In next chapter we will build Neural Network using Keras, that will be able to predict the class of the Iris flower based on the provided attributes.

Code

Keras programs have similar to the workflow of TensorFlow programs. We are going to follow this procedure:

- Import the dataset
- Prepare data for processing

- Create the model
- Training
- Evaluate accuracy of the model
- Predict results using the model

Training and evaluating processes are crucial for any Artificial Neural Network. These processes are usually done using two datasets, one for training and other for testing the accuracy of the trained network. In the real world, we will often get just one dataset and then we will split them into two separate datasets. For the training set, we usually use 80% of the data and another 20% we use to evaluate our model. This time this is already done for us. You can download training set and test set with code that accompanies this article from **here**.

However before we go any further, we need to import some libraries. Here is the list of the libraries that we need to import.

```python
# Importing libraries
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
import numpy
import pandas as pd
```

importing.py hosted with ❤ by **GitHub** **view raw**

As you can see we are importing Keras dependencies, *NumPy* and P*andas. NumPy* is the fundamental package for scientific computing and *Pandas* provides easy to use data structures and data analysis tools.

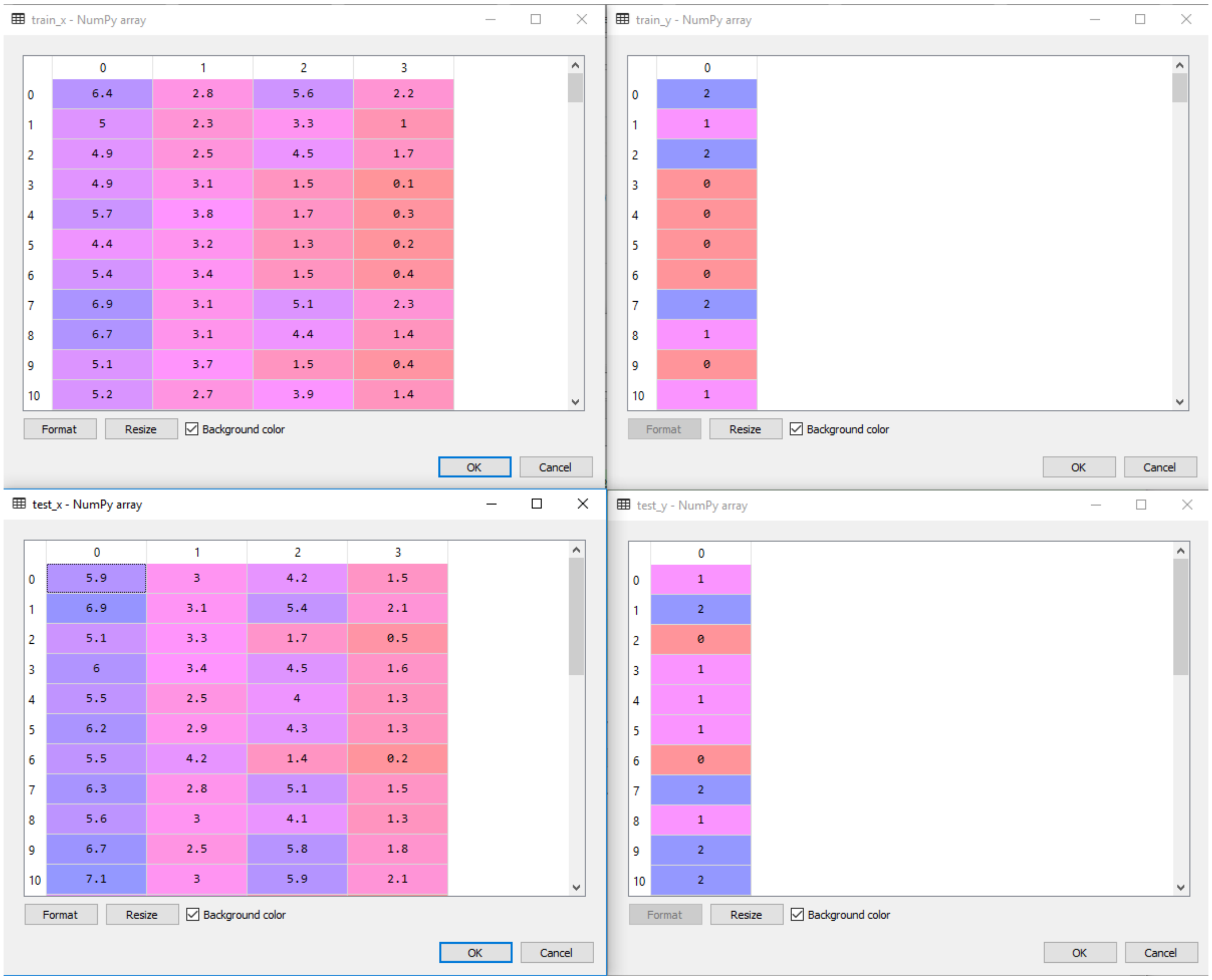
After we imported libraries, we can proceed with importing the data and preparing it for the processing. We are going to use *Pandas* for importing data:

```python
# Import training dataset
training_dataset = pd.read_csv('iris_training.csv', names=COLUMN_NAMES, header=0)
train_x = training_dataset.iloc[:, 0:4].values
train_y = training_dataset.iloc[:, 4].values

# Import testing dataset
test_dataset = pd.read_csv('iris_test.csv', names=COLUMN_NAMES, header=0)
test_x = test_dataset.iloc[:, 0:4].values
test_y = test_dataset.iloc[:, 4].values
```

import_data.py hosted with ❤ by **GitHub** **view raw**

Firstly, we used *read_csv* function to import the dataset into local variables, and then we separated inputs *(train_x, test_x)* and expected outputs *(train_y, test_y)* creating four separate matrixes. Here is how they look like:



train_x

	0	1	2	3
0	6.4	2.8	5.6	2.2
1	5	2.3	3.3	1
2	4.9	2.5	4.5	1.7
3	4.9	3.1	1.5	0.1
4	5.7	3.8	1.7	0.3
5	4.4	3.2	1.3	0.2
6	5.4	3.4	1.5	0.4
7	6.9	3.1	5.1	2.3
8	6.7	3.1	4.4	1.4
9	5.1	3.7	1.5	0.4
10	5.2	2.7	3.9	1.4

train_y

	0
0	2
1	1
2	2
3	0
4	0
5	0
6	0
7	2
8	1
9	0
10	1

test_x

	0	1	2	3
0	5.9	3	4.2	1.5
1	6.9	3.1	5.4	2.1
2	5.1	3.3	1.7	0.5
3	6	3.4	4.5	1.6
4	5.5	2.5	4	1.3
5	6.2	2.9	4.3	1.3
6	5.5	4.2	1.4	0.2
7	6.3	2.8	5.1	1.5
8	5.6	3	4.1	1.3
9	6.7	2.5	5.8	1.8
10	7.1	3	5.9	2.1

test_y

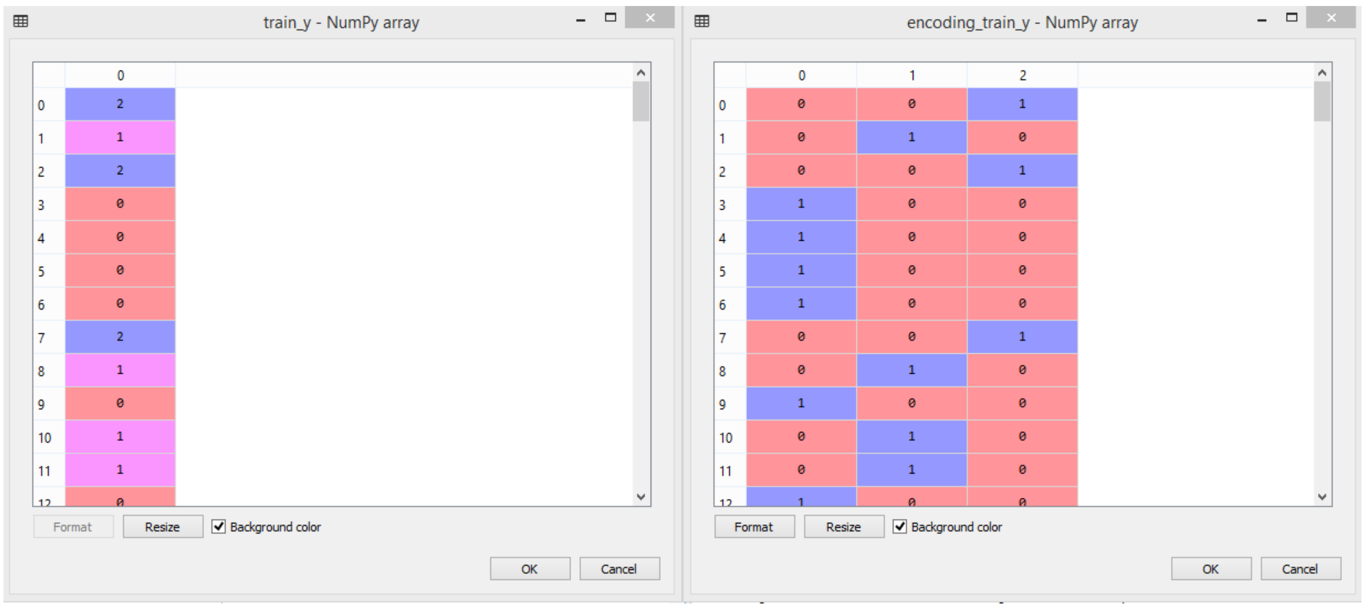
	0
0	1
1	2
2	0
3	1
4	1
5	1
6	0
7	2
8	1
9	2
10	2

However, our data is not prepared for processing yet. If we take a look at our expected output values, we can notice that we have three values: 0, 1 and 2. Value 0 is used to represent Iris setosa, value 1 to represent Iris versicolor and value 2 to represent virginica. The good news about these values is that we didn't get string values in the dataset. If you end up in that situation, you would need to use some kind of encoder so you can format data to something similar as we have in our current dataset. For this purpose, one can use ***LabelEncoder*** of sklearn library. Bad news about these values in the dataset is that they are not applicable to *Sequential* model. What we want to do is reshape the expected output from a vector that contains values for each class value to a matrix with a boolean for each class value. This is called **one-hot encoding**. In order to achieve this, we will use *np_utils* from the Keras library:

```python
# Encoding training dataset
encoding_train_y = np_utils.to_categorical(train_y)

# Encoding training dataset
encoding_test_y = np_utils.to_categorical(test_y)
```

If you still have doubt what one-hot encoding is doing, observe image below. There are displayed *train_y* variable and *encoding_train_y* variable. Notice that first value in *train_y* is 2 and see the corresponding value for that row in *encoding_train_y.*



	0
0	2
1	1
2	2
3	0
4	0
5	0
6	0
7	2
8	1
9	0
10	1
11	1
12	0

	0	1	2
0	0	0	1
1	0	1	0
2	0	0	1
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0
7	0	0	1
8	0	1	0
9	1	0	0
10	0	1	0
11	0	1	0
12	1	0	0

Once we imported and prepared the data we can create our model. We already know we need to do this by using *Sequence* and *Dense* class. So, let's do it:

```python
# Creating a model
model = Sequential()
model.add(Dense(10, input_dim=4, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(3, activation='softmax'))

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

This time we are creating:

- one input layer with four nodes, because we are having four attributes in our input values

- two hidden layers with ten neurons each
- one output layer with three neurons, because we are having three output classes

In hidden layers, neurons use **Rectifier activation function**, while in output layer neurons use Softmax activation function (ensuring that output values are in the range of 0 and 1). After that, we compile our model, where we define our **cost function** and optimizer. In this instance, we will use Adam **gradient descent optimization algorithm** with a logarithmic cost function (called *categorical_crossentropy* in Keras).

Finally, we can train our network:

```
# Training a model
model.fit(train_x, encoding_train_y, epochs=300, batch_size=10)
```

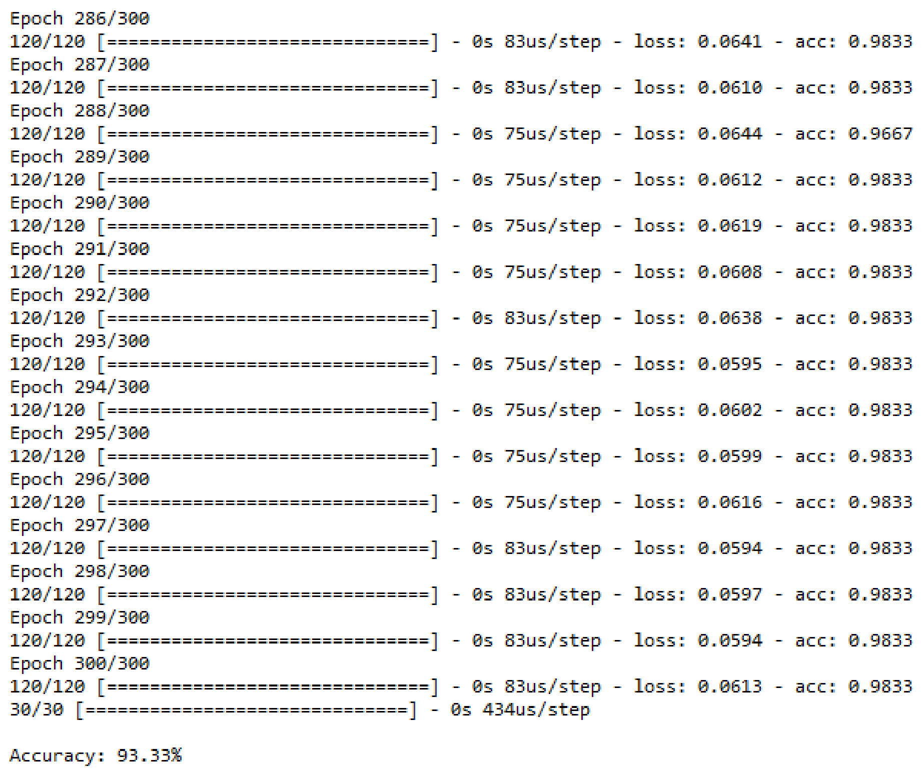
train.py hosted with ❤ by **GitHub** **view raw**

And evaluate it:

```
# Evaluate the model
scores = model.evaluate(test_x, encoding_test_y)
print("\nAccuracy: %.2f%%" % (scores[1]*100))
```

evaluate.py hosted with ❤ by **GitHub** **view raw**

If we run this code, we will get these results:



```
Epoch 286/300
120/120 [==============================] - 0s 83us/step - loss: 0.0641 - acc: 0.9833
Epoch 287/300
120/120 [==============================] - 0s 83us/step - loss: 0.0610 - acc: 0.9833
Epoch 288/300
120/120 [==============================] - 0s 75us/step - loss: 0.0644 - acc: 0.9667
Epoch 289/300
120/120 [==============================] - 0s 75us/step - loss: 0.0612 - acc: 0.9833
Epoch 290/300
120/120 [==============================] - 0s 75us/step - loss: 0.0619 - acc: 0.9833
Epoch 291/300
120/120 [==============================] - 0s 75us/step - loss: 0.0608 - acc: 0.9833
Epoch 292/300
120/120 [==============================] - 0s 83us/step - loss: 0.0638 - acc: 0.9833
Epoch 293/300
120/120 [==============================] - 0s 75us/step - loss: 0.0595 - acc: 0.9833
Epoch 294/300
120/120 [==============================] - 0s 75us/step - loss: 0.0602 - acc: 0.9833
Epoch 295/300
120/120 [==============================] - 0s 75us/step - loss: 0.0599 - acc: 0.9833
Epoch 296/300
120/120 [==============================] - 0s 75us/step - loss: 0.0616 - acc: 0.9833
Epoch 297/300
120/120 [==============================] - 0s 83us/step - loss: 0.0594 - acc: 0.9833
Epoch 298/300
120/120 [==============================] - 0s 83us/step - loss: 0.0597 - acc: 0.9833
Epoch 299/300
120/120 [==============================] - 0s 83us/step - loss: 0.0594 - acc: 0.9833
Epoch 300/300
120/120 [==============================] - 0s 83us/step - loss: 0.0613 - acc: 0.9833
30/30 [==============================] - 0s 434us/step

Accuracy: 93.33%
```

Since we have built the same network on the same dataset as we did with TensorFlow in the **previous article** we got the same accuracy – 0.93. That is pretty good. After this, we can call our classifier using single data and get predictions for it.

Conclusion

Keras is one awesome API which makes building Artificial Neural Networks easier. It is quite easy getting used to it. In this article, we just scratched the surface of this API and in next posts, we will explore how we can implement different types of Neural Networks using this API.

Thanks for reading!

This article is a part of Artificial Neural Networks Series, which you can check out **here**.

Read more posts from the author at **Rubik's Code**.
