Rubik's Code

TensorFlow Tutorial for Beginners with Python Example

Rubik's Code 2 weeks ago

The code that accompanies this article can be received after subscription

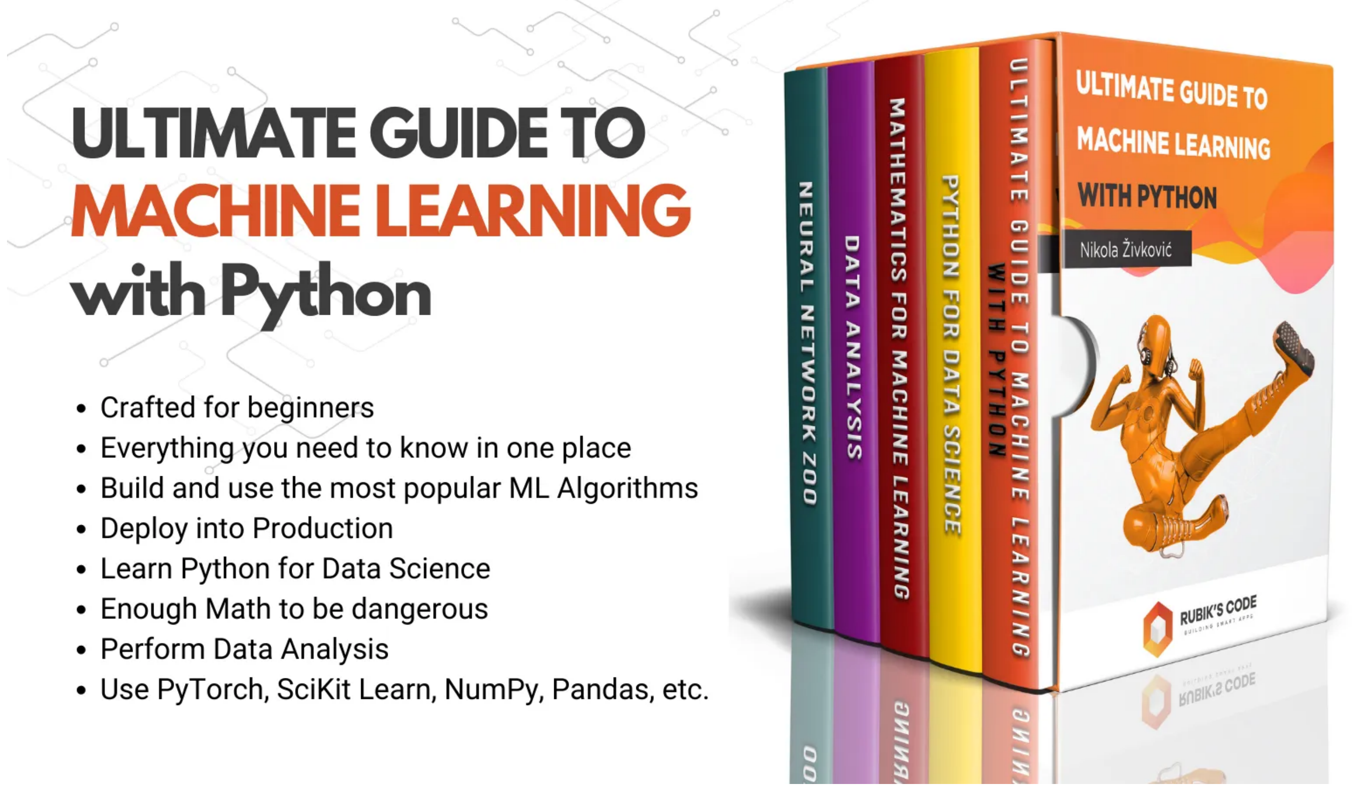
* indicates required

Email Address *

Subscribe

This November *TensorFlow* will celebrate its **fifth** birthday. Over the years it became one of the most loved ML **frameworks** and gathered a massive amount of followers. Google has done a great job and incorporated this framework into Java, C++, JavaScript and most importantly into major data science language **Python**.

If you ask the community what are their favorite combination of tools, the most usual answer would be TensorFlow and Python. Thanks to this, we came to the point where this technology is mature enough to ease up its use and "cross the chasm". However, let's start from the beginning and find out what is this technology all about.


ULTIMATE GUIDE TO
MACHINE LEARNING
with Python
• Crafted for beginners
• Everything you need to know in one place
• Build and use the most popular ML Algorithms
• Deploy into Production
• Learn Python for Data Science
• Enough Math to be dangerous
• Perform Data Analysis
• Use PyTorch, SciKit Learn, NumPy, Pandas, etc.
NEURAL NETWORK ZOO
DATA ANALYSIS
MATHEMATICS FOR MACHINE LEARNING
PYTHON FOR DATA SCIENCE
ULTIMATE GUIDE TO MACHINE LEARNING WITH PYTHON
ULTIMATE GUIDE TO MACHINE LEARNING WITH PYTHON
Nikola Živković
RUBIK'S CODE

In this article, we cover:

1. TensorFlow Basics

So, how TensorFlow works? Well, for starters their whole solution is revolving around tensors, primitive unit in TensorFlow. TensorFlow uses a tensor data structure to represent all data.

In math, tensors are geometric objects that describe linear relations between other geometric objects. In TesnsorFlow they are multi-dimensional array or data, ie. matrixes. Ok, it's not as simple as that, but this is whole tensor concept goes deeper in linear algebra that I'd like to go to right now.

Anyhow, we can observe tensors as n-dimensional arrays using which matrix operations are done easily and effectively. For example, in the code below, we defined two constant tensors and add one value to another:

```
import tensorflow as tf

const1 = tf.constant([[1,2,3], [1,2,3]]);
const2 = tf.constant([[3,4,5], [3,4,5]]);

result = tf.add(const1, const2);

with tf.Session() as sess:
  output = sess.run(result)
  print(output)
```

The constants, as you already figured out, are values that don't change. However, TensorFlow has rich API, which is well **documented** and using it we can define other types of data, like variables:

```
import tensorflow as tf

var1 = tf.Variable([[1, 2], [1, 2]], name="variable1")
var2 = tf.Variable([[3, 4], [3, 4]], name="variable2")

result = tf.matmul(var1, var2)

with tf.Session() as sess:
  output = sess.run(result)
  print(output)
```

Apart from tensors, TensorFlow uses data flow graphs. Nodes in the graph represent mathematical operations, while edges represent the tensors communicated between them.

2. TensorFlow Ecosystem

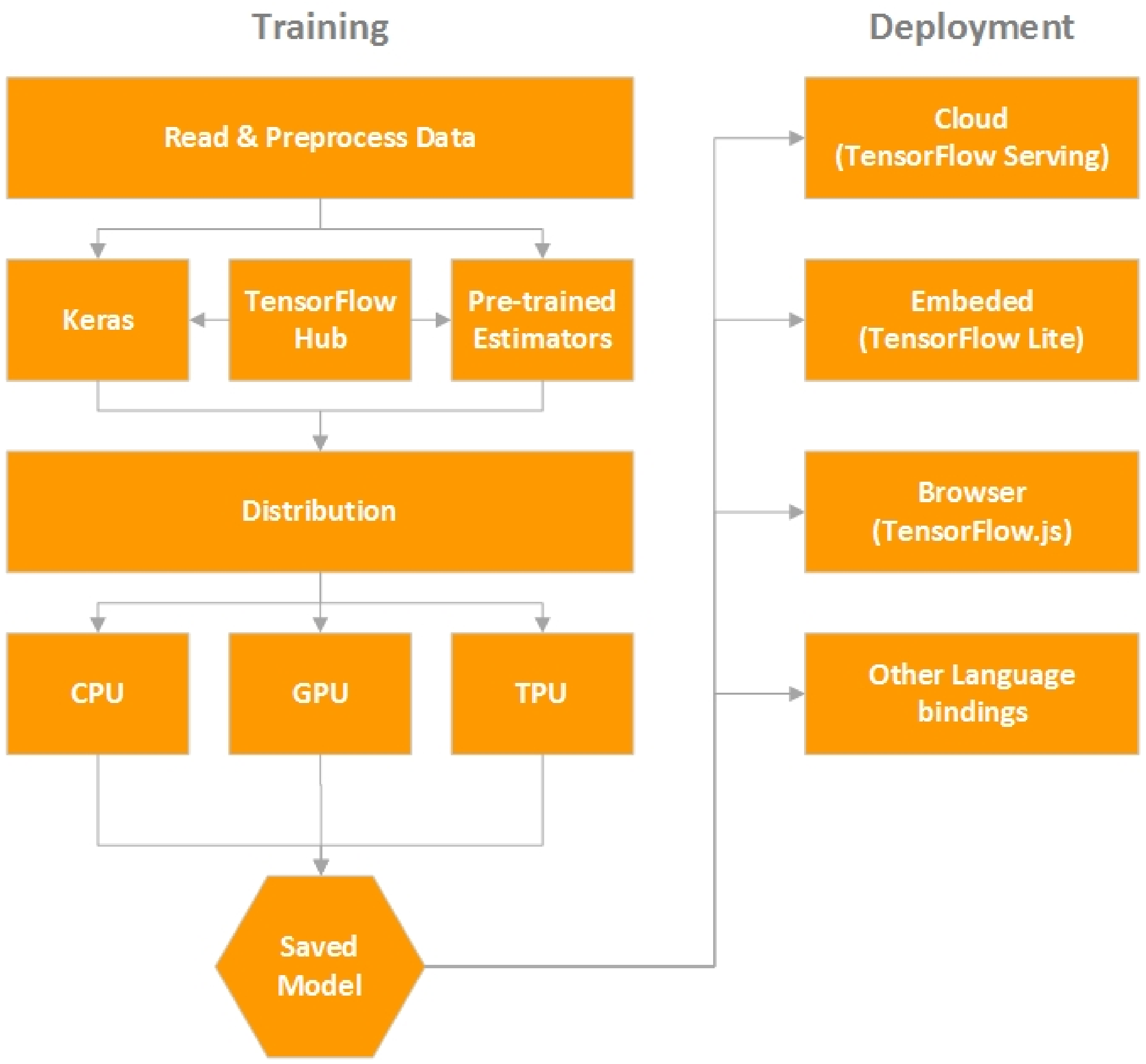
Of course, we don't want just to do simple arithmetic operations we want to use this library for building predictors, classifiers, generative models, neural networks and so on. In general, when you are building such solutions, we have to go through **several steps**:

- **Analysis** and preprocessing of the data
- **Building** and **training** a model (machine learning model, neural network, ...)
- **Evaluating** model
- Making new predictions

Since training of these models can be an expensive and long process we might want to use different machines to do this. Training these models on CPU can take quite a long time, so using GPU is always better options.

The fastest option for training these models is **tensor processing unit** or **TPU**s. These were introduced by Google back in 2016. and they are is an AI accelerator application-specific integrated circuit (ASIC). However, they are still quite expensive.

Apart from this, we want to deploy our model to different platforms, like cloud, embedded systems (IoT) or integrate it in other languages. That is why *TensorFlow* ecosystem looks something like this:



We can see that all these major points of developing these solutions are covered within this **ecosystem**. When it comes to Python, we usually analyze and handle data using libraries like *numpy* and *pandas*. Then we use this data to push it into the model that we have built.

This is a bit out of the scope of this article, and data analysis is a topic for itself. However, *TensorFlow* is giving us some **modules** using which we can do some preprocessing and feature engineering. Apart from that, it provides **datasets** (*tensorflow.datasets*) that we can use for training some of our custom solutions and for research in general.

The most important parts of *TensorFlow* is *TensorFlow Hub*. There we can find numerous modules and **low-level APIs** that we can use. On top of these let's say core modules we can find high-level API – **Keras**. We might say that road for 2.0 version was paved in *TensorFlow* 1.10.0 when *Keras* was incorporated as default High-Level API.

Before this Keras was a separate library and *tensorflow.contrib* module was used for this purpose. With TensorFlow 1.10.0 we got the news that *tensorflow.contrib* module will be soon **removed** and that *Keras* is taking over. And that was one of the main focuses of TensorFlow 2.0, to **ease up the use and to clean up the API**. In fact, many APIs from 1.0 are either moved or completely removed. For example, *tf.app* and *tf.flags* no longer exist and some less used functions from *tf.** are moved to other modules.

Apart from this High-Level API which we will use later in this article, there are several **pre-trained** models. These models are trained on some set of data and can be customized for your solution.

This approach in the development of a machine learning solution is also called **transferred learning**. Transferred learning is gaining popularity among artificial intelligence engineers because it is speeding up the process. Of course, you may choose to use these pre-trained models as out of the box solutions.

There are several **distribution** options for *TensorFlow* as well, so we can choose which platform we want to train our models. This is decided during the installation of the framework, so we will investigate it more in the later chapters. We can choose *TensorFlow* distribution that runs on CPU, GPU or TPU.

Finally, once we built our model, we can save it. This model can be incorporated into other applications on different **platforms**. Here we are entering the world of **deployment**. It is important to note that building a model is a completely different process from the rest of the application development. In general, data scientist build these models and save them. Later these models are called from business logic components of the application.

3. TensorFlow Installation and Setup

TensorFlow provides APIs for a wide range of languages, like Python, C++, Java, Go, Haskell and R (in a form of a third-party library). Also, it supports different types of operating systems. In this article, we are going to use Python on Windows 10 so only installation process on this platform will be covered.

TensorFlow supports only **Python 3.5 above**, so make sure that you one of those versions installed on your system. For other operating systems and languages you can check **official installation guide**. Another thing we need to know is hardware configuration of our system. There are two options for installing TensorFlow:

- TensorFlow with CPU support only.
- TensorFlow with GPU support.

If your system has an NVIDIA® GPU then you can install TensorFlow with GPU support. Of course, GPU version is faster, but CPU is easier to install and to configure.

If you are using **Anaconda** installing TensorFlow can be done following these steps:

1. Create a conda environment “tensorflow” by running the command:

```
conda create -n tensorflow pip python=3.5
```

2. Activate created environment by issuing the command:

```
activate tensorflow
```

3. Invoke the command to install TensorFlow inside your environment. For the CPU version run this command:

```
pip install --ignore-installed --upgrade tensorflow
```

For GPU version run the command:

```
pip install --ignore-installed --upgrade tensorflow-gpu
```

Of course, you can install TensorFlow using “native pip”, too. For the CPU version run:

```
pip3 install --upgrade tensorflow
```

For GPU TensorFlow version run the command:

```
pip3 install --upgrade tensorflow-gpu
```

Cool, now we have our TensorFlow installed. Let’s run through the problem we are going to solve.

4. Using Predefined TensorFlow Modules

4.1 Iris Data Set Classification Problem

Iris Data Set, along with the **MNIST dataset**, is probably one of the best-known datasets to be found in the pattern recognition literature. It is sort of “Hello World” example for machine learning classification problems. It was first introduced by Ronald Fisher back in 1936. He was British statistician and botanist and he used this example in this paper *The use of multiple measurements in taxonomic problems,* which is often referenced to this day. The dataset contains 3 classes of 50 instances each. Each class refers to one type of iris plant: *Iris setosa*, *Iris virginica,* and *Iris versicolor*. First class is linearly separable from the other two, but the latter two are not linearly separable from each other. Each record 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*)

The goal of the neural network, we are going to create is to predict the class of the Iris flower based on other attributes. Meaning it needs to create a **model**, which is going to describe a relationship between attribute values and the class.

4.2 TensorFlow Workflow

Most of the TensorFlow codes follow this workflow:

- Import the dataset
- Extend dataset with additional columns to describe the data
- Select the type of model
- Training
- Evaluate accuracy of the model
- Predict results using the model

If you followed my previous **blog posts**, one could notice that training and evaluating processes are important parts of developing any Artificial Neural Network. These processes are usually done on two datasets, one for training and other for testing the accuracy of the trained network. Often, we get just one set of data, that we need to split into two separate datasets and that use one for training and other for testing. The ratio is usually 80% to 20%. This time this is already done for us. You can download training set and test set with code that accompanies this article.

4.3 Python Code

Let's dive in! The first thing we need to do is to import the dataset and to parse it. For this, we are going to use another Python library – *Pandas*. This is another open source library that provides easy to use data structures and data analysis tools for the Python.

```
# Import `tensorflow` and `pandas`
import tensorflow as tf
import pandas as pd

COLUMN_NAMES = [
        'SepalLength',
        'SepalWidth',
        'PetalLength',
        'PetalWidth',
        'Species'
        ]

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

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

As you can see, first 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:

Great! We prepared data that is going to be used for training and for testing. Now, we need to define feature columns, that are going to help our Neural Network.

```
# Setup feature columns
columns_feat = [
    tf.feature_column.numeric_column(key='SepalLength'),
    tf.feature_column.numeric_column(key='SepalWidth'),
    tf.feature_column.numeric_column(key='PetalLength'),
    tf.feature_column.numeric_column(key='PetalWidth')
]
```

We now need to choose model we are going to use. In our problem, we are trying to predict a class of Iris Flower based on the attributes data. That is why we are going to choose one of the estimators from the TensorFlow API.

An object of the Estimator class encapsulates the logic that builds a TensorFlow graph and runs a TensorFlow session. For this purpose, we are going to use *DNNClassifier*. We are going to add two hidden layers with ten neurons in each.

```
# Build Neural Network - Classifier
classifier = tf.estimator.DNNClassifier(
    feature_columns=columns_feat,
    # Two hidden layers of 10 nodes each.
    hidden_units=[10, 10],
    # The model is classifying 3 classes
    n_classes=3)
```

After that, we will train our neural network with the data we picked from the training dataset. Firstly, we will define training function. This function needs to supply neural network with data from the training set by extending it and creating multiple batches.

Training works best if the training examples are in random order. That is why the *shuffle* function has been called. To sum it up, *train_function* creates batches of data using passed training dataset, by randomly picking data from it and supplying it back to *train* method of *DNNClassifier*.

```
# Define train function
def train_function(inputs, outputs, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices((dict(inputs), outputs))
    dataset = dataset.shuffle(1000).repeat().batch(batch_size)
    return dataset.make_one_shot_iterator().get_next()


# Train the Model.
classifier.train(
    input_fn=lambda:train_function(train_x, train_y, 100),
    steps=1000)
```

Finally, we call *evaluate* function that will evaluate our neural network and give us back accuracy of the network.

```
# Define evaluation function
def evaluation_function(attributes, classes, batch_size):
    attributes=dict(attributes)
    if classes is None:
        inputs = attributes
    else:
        inputs = (attributes, classes)
    dataset = tf.data.Dataset.from_tensor_slices(inputs)
    assert batch_size is not None, "batch_size must not be None"
    dataset = dataset.batch(batch_size)
    return dataset.make_one_shot_iterator().get_next()

# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda:evaluation_function(test_x, test_y, 100))
```

When we run this code I’ve got these results:

So, I got the accuracy of 0.93 for my neural network, which is pretty good. After this, we can call our classifier using single data and get predictions for it.

5. Keras – TensorFlow’s High-Level API

Today, **Keras** is default High-Level API of the *TensorFlow*. In this article, we will use this API to build a simple neural network later, so let’s explore a little bit how it functions. Depending on the type of a problem we can use a variety of layers for the neural network that we want to build.

Essentially, Keras is providing different types of layers (*tensorflow.keras.layers*) which we need to connect into a meaningful graph that will solve our problem. There are several ways in which we can do this API when building deep learning models:

- Using Sequential class
- Using Functional API
- Model subclassing

The first approach is the simplest one. We are using *Sequential* class, which is actually a **placeholder** for layers and we add layers in the order we want to. We may want to choose this approach when we want to build neural networks in the **fastest** way possible.

There are many types of Keras layers we can choose from, 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. So, let's see how one can build a **Neural Network** using *Sequential* and *Dense*.

6. Building Neural Network with TensorFlow, Keras and Python

In order to solve this problem, we are going to take steps we defined in one of the previous chapters:

- Analysis and preprocessing of the data
- Building and training a model
- Evaluating model
- Making new predictions

6.1 Data Analysis and Preprocessing

Data analysis is a topic for itself. In here, we will not go so deep into **feature engineering** and analysis, but we are going to observe some basic steps:

- **Univariate Analysis** – Analysing types and nature of every feature.
- **Missing Data Treatment** – Detecting missing data and making a strategy about it.
- **Correlation Analysis** – Comparing features among each other.
- **Splitting Data** – Because we have one set of information we need to make a separate set of data for training the neural network and set of data to evaluate the neural network.

Using the information that we gather during this analysis we can take appropriate actions during the creation of the model itself. First, we **import** the data:

```
COLUMN_NAMES = [

        'SepalLength',

        'SepalWidth',

        'PetalLength',

        'PetalWidth',

        'Species'

        ]


data = pd.read_csv('iris_data.csv', names=COLUMN_NAMES, header=0)

data.head()
```

As you can see we use *Pandas* library for this, and we also print out first five rows of data. Here is how that looks like:

Once this is done, we want to see what is the **nature** of every feature. For that we can use *Pandas* as well:

```
data.dtypes
```

Output looks like this:

As we can see the *Species* or the output has type *int64*. However, we understand that this is not what we want it to be. We want this feature to be a **categorical variable**. This means we need to modify this data a little bit, again using *Pandas*:

```
data['Species'] = data['Species'].astype("category")

data.dtypes
```

Once this is done, we check is there **missing data** in our data set. This is done using this function:

```
print(data.isnull().sum())
```

Output of this call is:

Missing data can be a problem for our neural network. If there is missing data in our dataset, we need to define a **strategy** on how to handle it. Some of the approaches are that missing values are replaced with the **average** value of the feature or its **max** value.

However, there is no silver bullet and sometimes different strategies give better results than the others. Ok, off to the **correlation analysis**. During this step, we are checking how features relate to each other. Using *Pandas* and *Seaborn* modules we were able to get an image which shows matrix with levels of dependency between some of the features – **correlation matrix**:

```
corrMatt = data[["SepalLength","SepalWidth","PetalLength","PetalWidth","Species"]].corr()

mask = np.array(corrMatt)

mask[np.tril_indices_from(mask)] = False

fig,ax= plt.subplots()

fig.set_size_inches(20,10)

sn.heatmap(corrMatt, mask=mask,vmax=.8, square=True,annot=True)
```

Here is how that matrix looks like:

We wanted to find the relationship between *Spices* and some of the features using this correlation matrix. The values, as you can see, are between -1 and 1. We are aiming for the ones that have a value close to 1 or -1, which means that these features have t**oo much in common,** ie. too much **influence** on each other.

If we have that situation it is suggested to provide just one of those features to a model. This way we would avoid the situation in which our model gives overly optimistic (or plain wrong) **predictions**. However, in this dataset, we are having little information one way or another, so if we would remove all dependencies, we would have no

data

Finally, lets split data into **training** and **testing** set. Because a client will usually give us one large chunk of data we need to leave some data for the testing. Usually, this ratio is 80:20. In this article, we will use 70:30, just to play around. For this purpose we use a function from *SciKit Learn* library:

```
output_data = data["Species"]

input_data = data.drop("Species",axis=1)

X_train, X_test, y_train, y_test = train_test_split(input_data, output_data, test_size=0.3, random_state=42
```

In the end, we have four variables that contain **input** data for training and testing, and **output** data for training and testing. We can build our model now.

6.2 Building and Training a Neural Network

We need a quite simple neural network for this **classification**. In here, we use model sub-classing approach, but you may try out other approaches as well. Here is how *IrisClassifier* class looks like:

```
class IrisClassifier(Model):
  def __init__(self):
    super(IrisClassifier, self).__init__()
    self.layer1 = Dense(10, activation='relu')
    self.layer2 = Dense(10, activation='relu')
    self.outputLayer = Dense(3, activation='softmax')

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

model = IrisClassifier()

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

It is the small neural network, with two layers of 10 neurons. The final layer is having 3 neurons because there are 3 classes of Iris flower. Also, in the final layer as the activation function is using *softmax*.

This means that we will get an output in the form of **probability**. Let's **train** this neural network. For this, we are using the *fit* method and pass prepared training data:

```
model.fit(X_train, y_train, epochs=300, batch_size=10)
```

The number of **epochs** is defining how much time the whole training set will be passed through the network. This can last for a couple of minutes and output looks like this:

And we are done. We created a model and trained it. Now, we have to evaluate it and see if we have good results.

6.3 Evaluating and New Predictions

Evaluation is done with the call of the *evaluate* method. We provide testing data to it and it runs predictions for every sample and compare it with the real result:

```
scores = model.evaluate(X_test, y_test)
print("\nAccuracy: %.2f%%" % (scores[1]*100))
```

In this particular case we got the accuracy of 95.56%:

```
45/45 [==============================] - 0s 756us/step
Accuracy: 95.56%
```

Finally, lets get some predictions:

```
prediction = model.predict(X_test)
prediction1 = pd.DataFrame({'IRIS1':prediction[:,0],'IRIS2':prediction[:,1], 'IRIS3':prediction[:,2]})
prediction1.round(decimals=4).head()
```

Here are the results that we got in comparison with real results:

These good results would be fishy if we are working on some other dataset with real data. We could suspect that 'overfitting' happened. However, on this simple dataset, we will accept these results as good results.

7. TensorFlow vs PyTorch

TensorFlow/Keras and PyTorch are the most popular deep learning frameworks. In the previous article, we wrote about **PyTorch**. In general, the difference is in speed (models are faster trained with PyTorch) and PyTorch feels, well…more pythonic, so to say. PyTorch is also pure Object oriented, while with TensorFlow you have options. Also, TensorFlow is dominating the industry, while PyTorch is popular in research.

Conclusion

Neural networks have been around for a long time and almost all important concepts were introduced back to 1970s or 1980s. The problem that was stopping the whole field to take off was that back then we had no powerful computers and GPUs to run these kinds of processes. Now, not only we can do that, but Google made Neural Networks popular by making this great tool – TensorFlow publically available.

Thanks for reading!

Share:

Categories: AI, Python

Tags: artificaial inteligance, artificial neural networks, datascience, deep learning, machine learning, neural networks, software development, TensorFlow, tensorflow examples, tensorflow github, tensorflow tutorial, tensorflow vs pytorch

Leave a Comment

Rubik's Code

Back to top