Develop Your First Neural Network in Python With Keras Step-By-Step

by **Jason Brownlee** on May 24, 2016 in **Deep Learning**

Keras is a powerful easy-to-use Python library for developing and evaluating deep learningmodels.

It wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in a few short lines of code.

In this post, you will discover how to create your first neural network model in Python using Keras.

Let's get started.

- **Update Feb/2017**: Updated prediction example so rounding works in Python 2 and Python 3.
- **Update Mar/2017**: Updated example for Keras 2.0.2, TensorFlow 1.0.1 and Theano 0.9.0.
- **Update Mar/2018**: Added alternate link to download the dataset as the original appears to have been taken down.

Develop Your First Neural Network in Python With Keras Step-By-Step
Photo by Phil Whitehouse, some rights reserved.

Tutorial Overview

There is not a lot of code required, but we are going to step over it slowly so that you will know how to create your own models in the future.

The steps you are going to cover in this tutorial are as follows:

1. Load Data.
2. Define Model.
3. Compile Model.
4. Fit Model.
5. Evaluate Model.
6. Tie It All Together.

This tutorial has a few requirements:

1. You have Python 2 or 3 installed and configured.
2. You have SciPy (including NumPy) installed and configured.
3. You have Keras and a backend (Theano or TensorFlow) installed and configured.

If you need help with your environment, see the tutorial:

- How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda

Create a new file called **keras_first_network.py** and type or copy-and-paste the code into the file as you go.

1. Load Data

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to set the random number seed.

This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code.

You can initialize the random number generator with any seed you like, for example:

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

```
3 import numpy
4 # fix random seed for reproducibility
5 numpy.random.seed(7)
```

Now we can load our data.

In this tutorial, we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

As such, it is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values, and ideal for our first neural network in Keras.

- Dataset File
- Dataset Details

Download the dataset and place it in your local working directory, the same as your python file. Save it with the file name:

```
1 pima-indians-diabetes.csv
```

You can now load the file directly using the NumPy function **loadtxt()**. There are eight input variables and one output variable (the last column). Once loaded we can split the dataset into input variables (X) and the output class variable (Y).

```
1 # load pima indians dataset
2 dataset = numpy.loadtxt("pima-indians-di
3 # split into input (X) and output (Y) va
4 X = dataset[:,0:8]
5 Y = dataset[:,8]
```

We have initialized our random number generator to ensure our results are reproducible and loaded our data. We are now ready to define our neural network model.

Note, the dataset has 9 columns and the range 0:8 will select columns from 0 to 7, stopping before index 8. If this is new to you, then you can learn more about array slicing and ranges in this post:

- How to Index, Slice and Reshape NumPy Arrays for Machine Learning in Python

2. Define Model

Models in Keras are defined as a sequence of layers.

We create a Sequential model and add layers one at a time until we are happy with our network topology.

The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the **input_dim** argument and setting it to 8 for the 8 input variables.

How do we know the number of layers and their types?

This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough to capture the structure of the problem if that helps at all.

In this example, we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the Dense class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as **init** and specify the activation function using the **activation** argument.

In this case, we initialize the network weights to a small random number generated from a uniform distribution ('**uniform**'), in this case between 0 and 0.05 because that is the default uniform weight initialization in Keras. Another traditional alternative would be '**normal**' for small random numbers generated from a Gaussian distribution.

We will use the rectifier ('**relu**') activation function on the first two layers and the sigmoid function in the output layer. It used to be the case that sigmoid and tanh activation functions were preferred for all layers. These days, better performance is achieved using the rectifier activation function. We use a sigmoid on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5.

We can piece it all together by adding each layer. The first layer has 12 neurons and expects 8 input variables. The second hidden layer has 8 neurons and finally, the output layer has 1 neuron to predict the class (onset of diabetes or not).

```
1 # create model
2 model = Sequential()
3 model.add(Dense(12, input_dim=8, activat
4 model.add(Dense(8, activation='relu'))
5 model.add(Dense(1, activation='sigmoid')
```

3. Compile Model

Now that the model is defined, we can compile it.

Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU or GPU or even distributed.

When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to make predictions for this problem.

We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training.

In this case, we will use logarithmic loss, which for a binary classification problem is defined in Keras as "**binary_crossentropy**". We will also use the efficient gradient descent algorithm "**adam**" for no other reason that it is an efficient default. Learn more about the Adam optimization algorithm in the paper "Adam: A Method for Stochastic Optimization".

Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.

```
1 # Compile model
```

```
2 model.compile(loss='binary_crossentropy'
```

4. Fit Model

We have defined our model and compiled it ready for efficient computation.

Now it is time to execute the model on some data.

We can train or fit our model on our loaded data by calling the **fit()** function on the model.

The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the **nepochs**argument. We can also set the number of instances that are evaluated before a weight update in the network is performed, called the batch size and set using the **batch_size**argument.

For this problem, we will run for a small number of iterations (150) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.

```
1 # Fit the model
2 model.fit(X, Y, epochs=150, batch_size=1
```

This is where the work happens on your CPU or GPU.

No GPU is required for this example, but if you're intersted in how to run large models on GPU hardware cheaply in the cloud, see this post:

- How To Develop and Evaluate Large Deep Learning Models with Keras on Amazon Web Services

5. Evaluate Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset.

This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the **evaluate()** function on your model and pass it the same input and output used to train the model.

This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

```
1 # evaluate the model
2 scores = model.evaluate(X, Y)
3 print("\n%s: %.2f%%" % (model.metrics_na
```

6. Tie It All Together

You have just seen how you can easily create your first neural network model in Keras.

Let's tie it all together into a complete code example.

```
# Create your first MLP in Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-d
# split into input (X) and output (Y) v
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activa
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'
# Compile model
model.compile(loss='binary_crossentropy
# Fit the model
model.fit(X, Y, epochs=150, batch_size=
# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_n
```

Running this example, you should see a message for each of the 150 epochs printing the loss and accuracy for each, followed by the final evaluation of the trained model on the training dataset.

It takes about 10 seconds to execute on my workstation running on the CPU with a Theano backend.

```
...
Epoch 145/150
768/768 [==============================
Epoch 146/150
768/768 [==============================
Epoch 147/150
768/768 [==============================
Epoch 148/150
768/768 [==============================
Epoch 149/150
768/768 [==============================
Epoch 150/150
768/768 [==============================
 32/768 [>.............................
acc: 78.26%
```

Note: If you try running this example in an IPython or Jupyter notebook you may get an error. The reason is the output progress bars during training. You can easily turn these off by setting **verbose=0** in the call to **model.fit()**.

Note, the skill of your model may vary.

Neural networks are a stochastic algorithm, meaning that the same algorithm on the same data can train a different model with different skill. This is a feature, not a bug. You can learn more about this in the post:

- Embrace Randomness in Machine Learning

We did try to fix the random seed to ensure that you and I get the same model and therefore the same results, but this does not always work on all systems. I write more about the problem of reproducing results with Keras models here.

7. Bonus: Make Predictions

The number one question I get asked is:

> *After I train my model, how can I use it to make predictions on new data?*

Great question.

We can adapt the above example and use it to generate predictions on the training dataset, pretending it is a new dataset we have not seen before.

Making predictions is as easy as calling **model.predict()**. We are using a sigmoid activation function on the output layer, so the predictions will be in the range between 0 and 1. We can easily convert them into a crisp binary prediction for this classification task by rounding them.

The complete example that makes predictions for each record in the training data is listed below.

```
# Create first network with Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-d
# split into input (X) and output (Y) v
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='
model.add(Dense(8, init='uniform', acti
model.add(Dense(1, init='uniform', acti
# Compile model
model.compile(loss='binary_crossentropy
# Fit the model
model.fit(X, Y, epochs=150, batch_size=
# calculate predictions
predictions = model.predict(X)
# round predictions
rounded = [round(x[0]) for x in predict
print(rounded)
```

Running this modified example now prints the predictions for each input pattern. We could use these predictions directly in our application if needed.

```
[1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0,
```

If you have more questions about making predictions with trained models, see this post:

- How to Make Predictions with Keras

Summary

In this post, you discovered how to create your first neural network model using the powerful Keras Python library for deep learning.