My Journey into DeepLearning using Keras

Parul Pandey Follow
Nov 2, 2018 · 11 min read

An introduction to building and training a neural network from scratch in Keras.

PC: Felipe Lopez on Unsplash

"Don't believe the short-term hype, but do believe in the long-term vision. It may take a while for AI to be deployed to its true potential—a potential the full extent of which no one has yet dared to dream—but

> *AI is coming, and it will transform our world in a fantastic way"—Francois Chollet*

Deep learning is a fascinating field of machine learning and also a promising one. It has fared remarkably well especially in areas of perceptual problems like seeing and hearing which were once impossible for machines to tackle. My first tryst with Deep Learning occurred a couple of years ago when I was working on problems in Kaggle. It was here that I was introduced to Keras and I thought of exploring it in detail. Out of all the other deep learning libraries that I have come across, I found Keras extremely easy, flexible and straightforward to use. The documentation and examples were also convenient and easy to follow.

For having a stronger grasp over the subject, I have been following a book called "**Deep Learning with Python**" by **Francois Chollet** who also happens to be the author of Keras**.** It has already become one of my most favored book on the topic, and I have learned quite a lot from it. Like I always say, I have a habit of creating notes while reading any book, so here are my notes from the **Deep learning with Python** book. I have tried to summarise the essential concepts and presented them in the form of an article. For anyone who is thinking to start or has just begun their journey into the Deep Learning world will find these notes useful.

. . .

Objective

The objective of this article is to get acquainted with the world of Keras, its installation, deployment and how it can be used to solve a Machine Learning problem. This article does not deal with advanced deep learning concepts like LSTM, Convets, etc. I intend to cover those topics in my next series of articles.

What should you know

Since you have ventured into the field of Deep learning and you are particularly interested in Keras, it is assumed that you:

- have basic programming experience
- have a fair idea of working in Python and

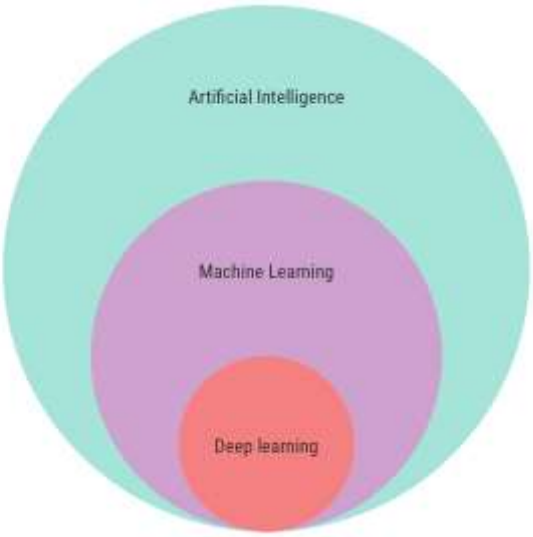
- are aware of fundamental Machine learning concepts.

. . .

Table of Contents

. . .

1. Deep learning: A Primer



Deep learning is a specific subfield of machine learning. Deep Learning emphasizes learning successive *layers* of increasingly meaningful representations. The ***deep*** in deep learning stands *for the* idea of using successive layers of representations. These layered representations are learned through models called neural networks which are structured in literal layers stacked on top of each other.

Why Deep learning now?

In general, three technical forces are driving major advances in machine learning today

- Hardware
- Datasets and benchmarks and
- Algorithmic advances

It is due to abundance and improvements in all these three fields today that Deep learning has gained momentum.

What Deep learning has achieved so far

Some of the major successes that deep learning has achieved are:

- Near-human-level image classification and speech recognition.
- Near-human-level handwriting transcription
- Improved machine translation
- Improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Ability to answer natural-language questions

. . .

2. Keras: The Python Deep Learning library

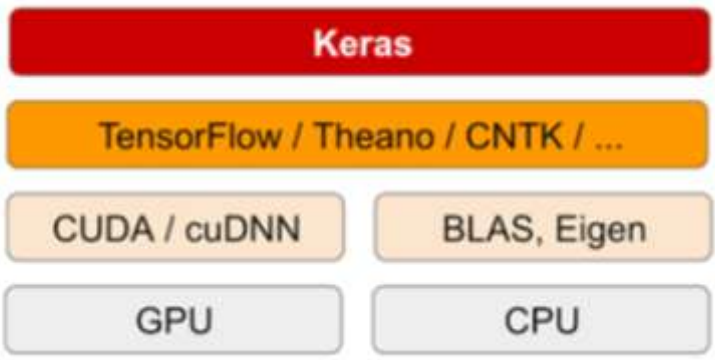
source

Keras (κέρας) means *horn* in Greek. It is a reference to a literary image from ancient Greek and Latin literature, first found in the *Odyssey*. Keras was initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating

System). Keras is a framework for building deep neural networks with Python. Keras enables us to build state-of-the-art, deep learning systems just like those used at Google and Facebook, with little complexity and also with a few lines of code. Some of its **key features** are:

- User-friendly API
- Built-In support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on.

Keras is a front-end layer written in Python that runs on top of other popular deep learning toolkits like **TensorFlow, Theano** and **Microsoft Cognitive Toolkit (CNTK)**. Any piece of code that you write with Keras can be run with any of these backends without having to change anything in the code. Via TensorFlow/Theano /CNTK, Keras can run seamlessly on both CPUs and GPUs.



The deep-learning software and hardware stack

Installation

Before installing Keras, we need to install one of its backend engines, i.e., either of the three: TensorFlow, Theano, or CNTK. In this article, we shall be working with the TensorFlow backend. Read the detailed instructions here to install Tensorflow.

There are two ways to install Keras:

- **Install Keras from PyPI (recommended):**

```
sudo pip install keras
```

If you are using a virtualenv, you may want to avoid using sudo:

```
pip install keras
```

- **Alternatively: Install Keras from the GitHub source:**

First, clone Keras using `git`:

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

Then, `cd` to the Keras folder and run the install command:

```
cd keras
sudo python setup.py install
```

Please refer to the Keras official documentation for any installation related queries.

Hardware requirements: setting up the workstation.

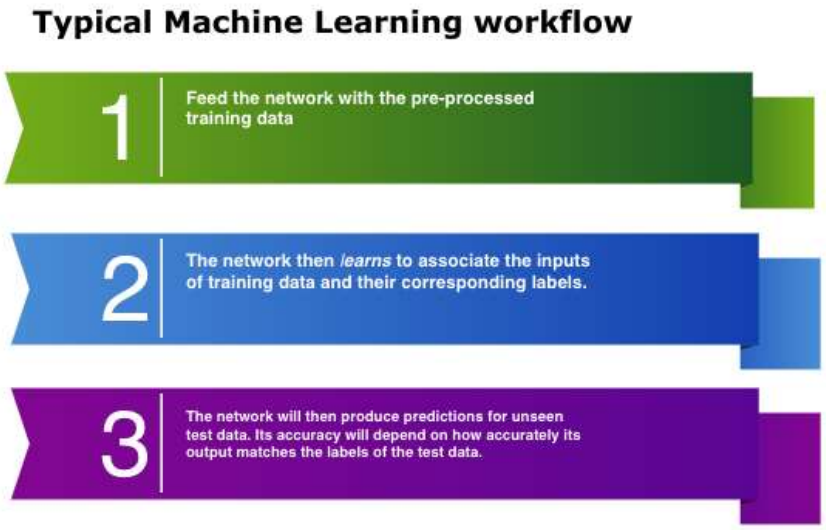
The author recommends (although not strictly necessary) that you run deep-learning code on a modern NVIDIA GPU. Alternatively, you can consider running your experiments on an AWS EC2 GPU instance or on Google Cloud Platform(this gets expensive over time).

In this article, I shall be using **Jupyter notebooks** to get started with Keras, although that isn't a requirement: you can also run standalone Python scripts or run code from within an IDE such as PyCharm.

• • •

3. Keras Workflow

A typical machine learning workflow is as follows:



A Keras workflow is no different and to completely understand it; we will be working with the example of a neural network that uses Keras to classify handwritten digits. It is the famous **MNIST** database problem also called the "**Hello World**" of deep learning. The problem is to classify grayscale images of handwritten digits (28 × 28 pixels) into their ten categories (0 through 9).

MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges

The MNIST database was constructed from NIST Special Database 3 and Special Database 1 which...

yann.lecun.com

> The MNIST dataset comes preloaded in Keras in the form of a set of four Numpy arrays.

Steps

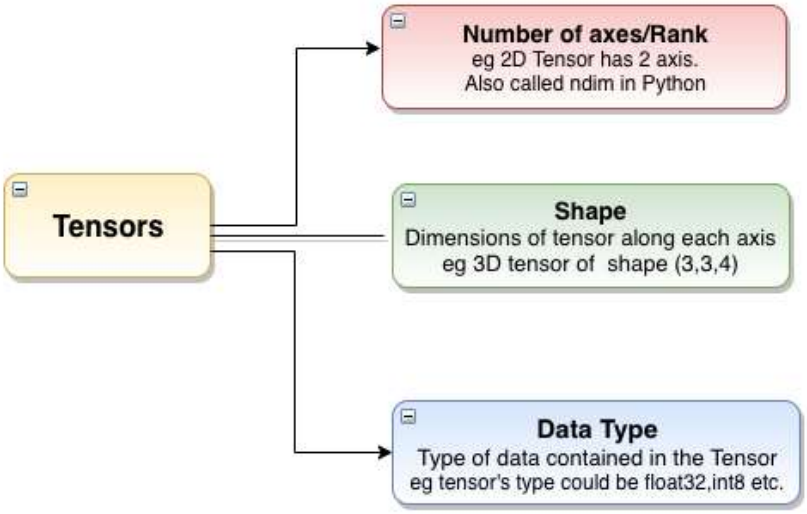
Let us underline the steps that we will undertake to solve this problem using Keras.

i) Loading the MNIST dataset in Keras

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
```

The images are encoded as Numpy arrays, and the labels are an array of digits, ranging from 0 to 9. The multidimensional Numpy arrays, in which data is stored are called **Tensors**. You can think of tensors as a container for data. To be more specific tensors are a generalization of matrices to an arbitrary number of dimensions. Tensors can be defined by three key features:



Let's have a look at the training and testing data.

```
#Training Data
train_images.shape
(60000, 28, 28)
len(train_labels) # Total no. of training images
60000
train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)

#Testing Data
test_images.shape
(10000, 28, 28)
len(test_labels)
```

```
10000
test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

So what we have here is a 3D tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

ii) Preprocessing the Data

i) **Preparing the image data**

Before feeding in the inputs, we will need to preprocess the data. This is done by reshaping it into a shape that is expected by the network. We will also scale all the values between the range(0,1).

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

So, we have transformed out training images from an array of shape `(6000, 28, 28), type=uint8` into a `float32` array of shape `(60000, 28 * 28)` with values between 0 and 1.

ii) **Encoding the labels**

The labels will also need to be categorically encoded since we cannot feed the lists of integers into a neural network.

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

iii) Building the Network architecture

A Deep learning network essentially consists of models and layers.

Layers

A **layer** is a core building block of a neural network. It acts as a kind of a data processing module. Layers extract ***representations*** out of the input data, that is fed into them. Inherently, Deep learning consists of stacking up these layers to form a model.

> *Layers are the LEGO bricks of deep learning*

Model

A model is a linear stack of layers. It is like a sieve for data processing made of a succession of increasing refined data filters called layers. The simplest model in Keras is **sequential**, which is built by stacking layers sequentially.

Model Building

We will be building our very first model in Keras with only two layers. Here, we're passing the expected shape of the input data to the first layer.

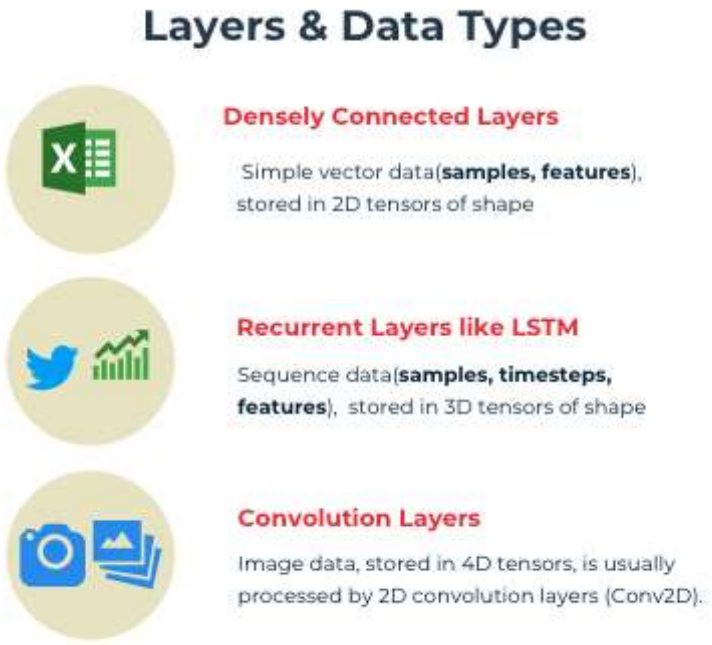
```
from keras import models
from keras import layers


network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Our network consists of a sequence of two **layers**, which are **densely connected** (also called *fully connected*) neural layers.

- The first layer comprising of 512 units taking an input of a 784-dimensional array converted from the 28×28 image. The activation function used is ReLU (rectified linear unit). A **rectified linear unit** has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input.
- The second (which is also the last here) layer is a 10-way ***softmax*** layer, which means it will return an array of 10 probability scores (summing to 1). Each score will be a probability that the current digit image belongs to one of our 10 digit classes. You can read more about the **softmax** function below.

Different layers are best suited for different types of data processing. The **general** rule(not a thumb rule) is :

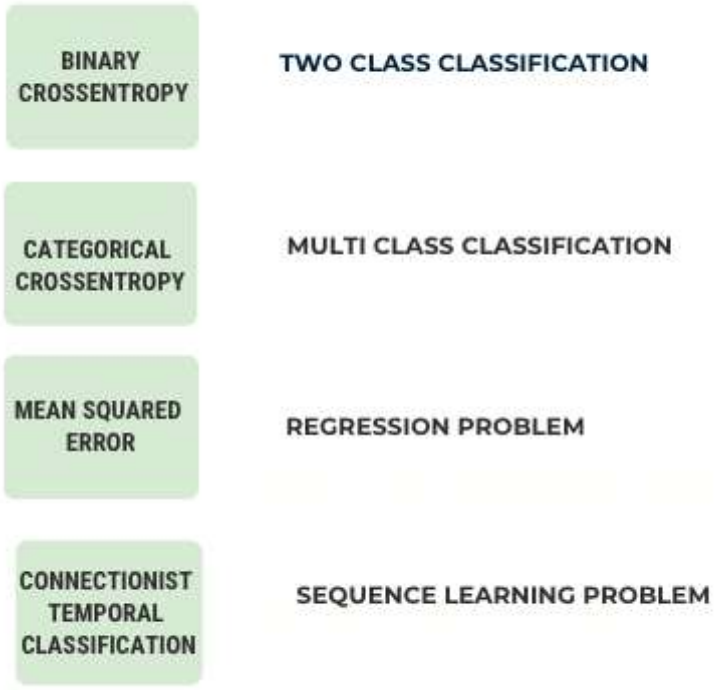


iv) Network Compilation

We now know that our network consists of two dense layers each of which is applying some tensor operations on the input data. The operations involve weight tensors, and it is at these weights that the *knowledge* of the network is present.

After the model has been built, we enter the compilation phase, which primarily consists of three essential elements:

- **Loss Function(Objective Function)**: *loss* (Predicted—Actual value) is the quantity that we try to minimize during training of a neural network. Thus, measuring the Loss function is a measure of success for the task we are trying to solve. It is imperative to choose the right objective function for a problem. See the diagram below for commonly used loss functions.



Commonly used Loss functions

- **Optimizer**: It specifies the exact way in which the gradient of the loss will be used to update parameters. In other words, it determines how the network will be updated based on the loss function. Optimizers could be the RMSProp optimizer, SGD with momentum, and so on.
- **Metrics**: to measure the accuracy of the model. In this case, we will use accuracy.

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',

metrics=['accuracy'])
```

Essentially, the reduction of the loss happens via mini- batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the rmsprop optimizer passed as the first argument.

v) Training the network

We're now ready to train the network, which in Keras is done via a call to the network's **fit method**—we *fit* the model to its training data:

```
network.fit(train_images, train_labels,
epochs=5,batch_size=128)
```

```
Epoch 1/5
60000/60000 [==============================] - 5s 90us/step
- loss: 0.2567 - acc: 0.9264
Epoch 2/5
60000/60000 [==============================] - 5s 88us/step
- loss: 0.1032 - acc: 0.9694
Epoch 3/5
60000/60000 [==============================] - 5s 89us/step
- loss: 0.0679 - acc: 0.9798
Epoch 4/5
60000/60000 [==============================] - 6s 92us/step
- loss: 0.0490 - acc: 0.9856
Epoch 5/5
60000/60000 [==============================] - 5s 89us/step
- loss: 0.0375 - acc: 0.9885
```

The network will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an ***epoch***). At each iteration, the network will compute the gradients of the weights about the loss on the batch, and update the weights

Two quantities are displayed during training:

- **Loss** of the network over the training data, and
- **Accuracy** of the network over the training data.

The accuracy on the training data comes out to be around `98.9%` Now let's check how the model performs on the test set.

vi) Testing on Test Data

```
test_loss, test_acc = network.evaluate(test_images,
test_labels)
10000/10000 [==============================] - 1s 66us/step

print('test_acc:', test_acc)
test_acc: 0.9783
```

The test-set accuracy turns out to be `97.8%` —that's quite a bit lower than the training set accuracy. But then that is the issue famously known as **overfitting** in ML terminology. You can avoid it by creating a

validation sample of about 10,000 images from the original training data and setting it aside. The testing can be done on this sample since it has not been exposed to the training before. The entire code can be accessed from here.

. . .

Conclusion

So that's pretty much it. We have worked through the whole MNIST digit classification problem in detail with the help of Keras. We just built and trained a neural network from scratch. However, theory is one aspect of learning, and its practical application is another. So get up and get going. Utilise your new found knowledge by working on some real datasets. I will see you in the next part where we will learn to utilise Keras for some advanced problems. Till then happy reading.