SEBASTIAN RASCHKA

Introduction to Artificial Neural Networks and Deep Learning

A Practical Guide
with Applications in Python

Introduction to Artificial Neural Networks and Deep Learning

A Practical Guide with Applications in Python

Sebastian Raschka

This book is for sale at http://leanpub.com/ann-and-deeplearning

This version was published on 2017-02-25

Contents

Website

Please visit the GitHub repository[1] to download code examples used in this book.

If you like the content, please consider supporting the work by buying a copy of the book on Leanpub[2].

I would appreciate hearing your opinion and feedback about the book! Also, if you have any questions about the contents, please don't hesitate to get in touch with me via mail@sebastianraschka.com or join the mailing list[3].

Happy learning!

Sebastian Raschka

[1]https://github.com/rasbt/deep-learning-book
[2]https://leanpub.com/ann-and-deeplearning
[3]https://groups.google.com/forum/#!forum/ann-and-dl-book

Acknowledgments

I would like to give my special thanks to the readers, who caught various typos and errors and offered suggestions for clarifying my writing.

- Appendix A: Artem Sobolev, Ryan Sun
- Appendix B: Brett Miller, Ryan Sun
- Appendix D: Marcel Blattner, Ignacio Campabadal, Ryan Sun

Appendix G - TensorFlow Basics

This appendix offers a brief overview of TensorFlow, an open-source library for numerical computation and deep learning. This section is intended for readers who want to gain a basic overview of this library before progressing through the hands-on sections that are concluding the main chapters.

The majority of *hands-on* sections in this book focus on TensorFlow and its Python API, assuming that you have TensorFlow >=1.0 installed if you are planning to execute the code sections shown in this book.

In addition to glancing over this appendix, I recommend the following resources from TensorFlow's official documentation for a more in-depth coverage on using TensorFlow:

- **Download and setup instructions**[4]
- **Python API documentation**[5]
- **Tutorials**[6]
- **TensorBoard, an optional tool for visualizing learning**[7]

TensorFlow in a Nutshell

At its core, TensorFlow is a library for efficient multidimensional array operations with a focus on deep learning. Developed by the Google Brain Team, TensorFlow was open-sourced on November 9th, 2015. And augmented by its convenient Python API layer, TensorFlow has gained much popularity and wide-spread adoption in industry as well as academia.

TensorFlow shares some similarities with NumPy, such as providing data structures and computations based on multidimensional arrays. What makes TensorFlow particularly suitable for deep learning, though, are its primitives for defining functions on tensors, the ability of parallelizing tensor operations, and convenience tools such as automatic differentiation.

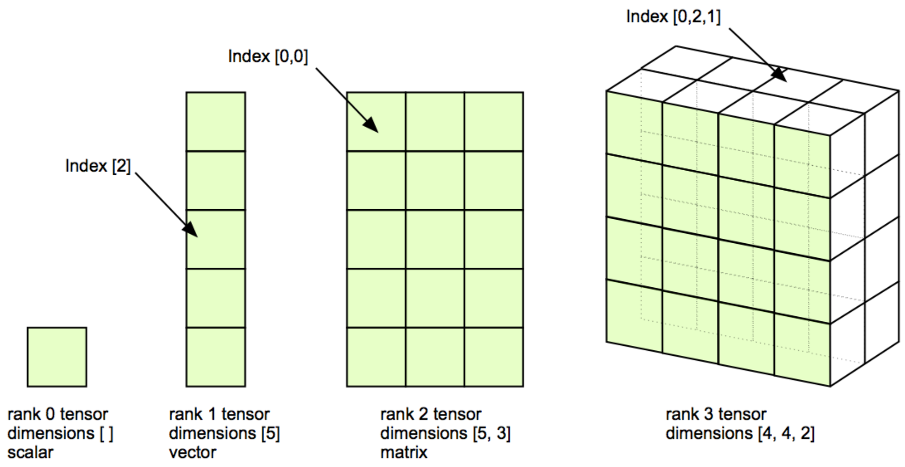
[4] https://www.tensorflow.org/get_started/os_setup
[5] https://www.tensorflow.org/api_docs/python/
[6] https://www.tensorflow.org/tutorials/
[7] https://www.tensorflow.org/how_tos/summaries_and_tensorboard/

While TensorFlow can be run entirely on a CPU or multiple CPUs, one of the core strength of this library is its support of GPUs (Graphical Processing Units) that are very efficient at performing highly parallelized numerical computations. In addition, TensorFlow also supports distributed systems as well as mobile computing platforms, including Android and Apple's iOS.

But what is a *tensor*? In simplifying terms, we can think of tensors as multidimensional arrays of numbers, as a generalization of scalars, vectors, and matrices.

1. Scalar: $\mathbb{R}$
2. Vector: $\mathbb{R}^n$
3. Matrix: $\mathbb{R}^n \times \mathbb{R}^m$
4. 3-Tensor: $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p$
5. …

When we describe tensors, we refer to its "dimensions" as the *rank* (or *order*) of a tensor, which is not to be confused with the dimensions of a matrix. For instance, an $m \times n$ matrix, where m is the number of rows and n is the number of columns, would be a special case of a rank-2 tensor. A visual explanation of tensors and their ranks is given is the figure below.



Tensors

Installation

Code conventions in this book follow the Python 3.x syntax, and while the code examples should be backward compatible to Python 2.7, I highly recommend the use of Python >=3.5.

Once you have your Python Environment set up (Appendix - Python Setup), the most convenient ways for installing TensorFlow are via `pip` or `conda` – the latter only applies if you have the Anaconda/Miniconda Python distribution installed, which I prefer and recommend.

Since TensorFlow is under active development, I recommend you to consult the official "Download and Setup[8]" documentation for detailed installation instructions to install TensorFlow on you operating system, macOS, Linux, or Windows.

Computation Graphs

In contrast to other tools such as NumPy, the numerical computations in TensorFlow can be categorized into two steps: a construction step and an execution step. Consequently, the typical workflow in TensorFlow can be summarized as follows:

- Build a computational graph
- Start a new *session* to evaluate the graph
  - Initialize variables
  - Execute the operations in the compiled graph

Note that the computation graph has no numerical values before we initialize and evaluate it. To see how this looks like in practice, let us set up a new graph for computing the column sums of a matrix, which we define as a constant tensor (`reduce_sum` is the TensorFlow equivalent of NumPy's `sum` function).

In [1]:

[8] https://www.tensorflow.org/get_started/os_setup

```python
import tensorflow as tf

g = tf.Graph()

with g.as_default() as g:
    tf_x = tf.constant([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    col_sum = tf.reduce_sum(tf_x, axis=0)

print('tf_x:\n', tf_x)
print('\ncol_sum:\n', col_sum)
```

Out [1]:

```
tf_x:
 Tensor("Const:0", shape=(3, 2), dtype=float32)

col_sum:
 Tensor("Sum:0", shape=(2,), dtype=float32)
```

As we can see from the output above, the operations in the graph are represented as `Tensor` objects that require an explicit evaluation before the `tf_x` matrix is populated with numerical values and its column sum gets computed.

Now, we pass the graph that we created earlier to a new, active *session*, where the graph gets compiled and evaluated:

In [2]:

```python
with tf.Session(graph=g) as sess:
    mat, csum = sess.run([tf_x, col_sum])

print('mat:\n', mat)
print('\ncsum:\n', csum)
```

Out [2]:

```
mat:
 [[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]]

csum:
 [  9.  12.]
```

Note that if we are only interested in the result of a particular operation, we don't need to `run` its dependencies – TensorFlow will automatically take care of that. For instance, we can directly fetch the numerical values of `col_sum_times_2` in the active session without explicitly passing `col_sum` to `sess.run(...)` as the following example illustrates:

In [3]:

```
g = tf.Graph()

with g.as_default() as g:
    tf_x = tf.constant([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    col_sum = tf.reduce_sum(tf_x, axis=0)
    col_sum_times_2 = col_sum * 2


with tf.Session(graph=g) as sess:
    csum_2 = sess.run(col_sum_times_2)

print('csum_2:\n', csum_2)
```

Out [3]:

```
csum_2:
 [array([ 18.,  24.], dtype=float32)]
```

Variables

Variables are constructs in TensorFlow that allows us to store and update parameters of our models in the current session during training. To define a "variable" tensor, we use TensorFlow's `Variable()` constructor, which looks similar to the use of `constant` that we used to create a matrix previously. However, to execute a computational graph that contains variables, we must initialize all variables in the active session first (using `tf.global_variables_initializer()`), as illustrated in the example below.

In [4]:

```
g = tf.Graph()

with g.as_default() as g:
    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    x = tf.constant(1., dtype=tf.float32)

    # add a constant to the matrix:
    tf_x = tf_x + x

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(tf_x)

print(result)
```

Out [4]:

```
[[ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]]
```

Now, let us do an experiment and evaluate the same graph twice:

In [5]:

```
with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(tf_x)
    result = sess.run(tf_x)
```

Out [5]:

```
[[ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]]
```

As we can see, the result of running the computation twice did not affect the numerical values fetched from the graph. To update or to assign new values to a variable, we use TensorFlow's `assign` operation. The function syntax of `assign` is `assign(ref, val, ...)`, where '`ref`' is updated by assigning '`value`' to it:

In [6]:

```
g = tf.Graph()

with g.as_default() as g:
    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    x = tf.constant(1., dtype=tf.float32)

    update_tf_x = tf.assign(tf_x, tf_x + x)


with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(update_tf_x)
    result = sess.run(update_tf_x)

print(result)
```

Out [6]:

```
[[ 3.  4.]
[ 5.  6.]
[ 7.  8.]]
```

As we can see, the contents of the variable `tf_x` were successfully updated twice now; in the active session we

- initialized the variable `tf_x`

- added a constant scalar `1.` to `tf_x` matrix via `assign`
- added a constant scalar `1.` to the previously updated `tf_x` matrix via `assign`

Although the example above is kept simple for illustrative purposes, variables are an important concept in TensorFlow, and we will see throughout the chapters, they are not only useful for updating model parameters but also for saving and loading variables for reuse.

Placeholder Variables

Another important concept in TensorFlow is the use of placeholder variables, which allow us to feed the computational graph with numerical values in an active session at runtime.

In the following example, we will define a computational graph that performs a simple matrix multiplication operation. First, we define a placeholder variable that can hold 3x2-dimensional matrices. And after initializing the placeholder variable in the active session, we will use a dictionary, `feed_dict` we feed a NumPy array to the graph, which then evaluates the matrix multiplication operation.

In [7]:

```
import numpy as np

g = tf.Graph()

with g.as_default() as g:
    tf_x = tf.placeholder(dtype=tf.float32,
                          shape=(3, 2))

    output = tf.matmul(tf_x, tf.transpose(tf_x))


with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    np_ary = np.array([[3., 4.],
                       [5., 6.],
                       [7., 8.]])
    feed_dict = {tf_x: np_ary}
    print(sess.run(output,
                   feed_dict=feed_dict))
```

Out [7]:

```
[[  25.   39.   53.]
 [  39.   61.   83.]
 [  53.   83.  113.]]
```

Throughout the main chapters, we will make heavy use of placeholder variables, which allow us to pass our datasets to various learning algorithms in the computational graphs.

Saving and Restoring Models

Training deep neural networks requires a lot of computations and computational resources, and in practice, it would be infeasible to retrain our model each time we start a new TensorFlow session before we can use it to make predictions. In this section, we will go over the basics of saving and re-using the results of our TensorFlow models.

The most convenient way to store the main components of our model is to use TensorFlows `Saver` class (`tf.train.Saver()`). To see how it works, let us reuse the simple example from the Variables section, where we added a constant `1.` to all elements in a 3x2 matrix:

In [8]:

```
g = tf.Graph()

with g.as_default() as g:

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    x = tf.constant(1., dtype=tf.float32)

    update_tf_x = tf.assign(tf_x, tf_x + x)

    # initialize a Saver, which gets all variables
    # within this computation graph context
    saver = tf.train.Saver()
```

Now, after we initialized the graph above, let us execute its operations in a new session:

In [9]:

```python
with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(update_tf_x)

    # save the model
    saver.save(sess, save_path='./my-model.ckpt')
```

Notice the `saver.save` call above, which saves all variables in the graph to "checkpoint" files bearing the prefix `my-model.ckpt` in our local directory (`'./'`). Since we didn't specify which variables we wanted to save when we instantiated a `tf.train.Saver()`, it saved all variables in the graph by default – here, we only have one variable, `tf_x`. Alternatively, if we are only interested in keeping particular variables, we can specify this by feeding `tf.train.Saver()` a dictionary or list of these variables upon instantiation. For example, if our graph contained more than one variable, but we were only interested in saving `tf_x`, we could instantiate a `saver` object as `tf.train.Saver([tf_x])`.

After we executed the previous code example, we should find the three `my-model.ckpt` files (in binary format) in our local directory:

- `my-model.ckpt.data-00000-of-00001`
- `my-model.ckpt.index`
- `my-model.ckpt.meta`

The file `my-model.ckpt.data-00000-of-00001` saves our main variable values, the `.index` file keeps track of the data structures, and the `.meta` file describes the structure of our computational graph that we executed.

Note that in our simple example above, we just saved our variable one single time. However, in real-world applications, we typically train models over multiple iterations or epochs, and it is useful to create intermediate checkpoint files during training so that we can pick up where we left off in case we need to interrupt our session or encounter unforeseen technical difficulties. For instance, by using the `global_step` parameter, we could save our results after each 10th iteration by making the following modification to our code:

In [10]:

```python
g = tf.Graph()

with g.as_default() as g:

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    x = tf.constant(1., dtype=tf.float32)

    update_tf_x = tf.assign(tf_x, tf_x + x)

    # initialize a Saver, which gets all variables
    # within this computation graph context
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(100):
        result = sess.run(update_tf_x)
        if not epoch % 10:
            saver.save(sess,
                       save_path='./my-model-multiple_ckpts.ckpt',
                       global_step=epoch)
```

After we executed this code we find five `my-model.ckpt` files in our local directory:

- `my-model.ckpt-50 {.data-00000-of-00001, .ckpt.index, .ckpt.meta}`
- `my-model.ckpt-60 {.data-00000-of-00001, .ckpt.index, .ckpt.meta}`
- `my-model.ckpt-70 {.data-00000-of-00001, .ckpt.index, .ckpt.meta}`
- `my-model.ckpt-80 {.data-00000-of-00001, .ckpt.index, .ckpt.meta}`
- `my-model.ckpt-90 {.data-00000-of-00001, .ckpt.index, .ckpt.meta}`

Although we saved our variables ten times, the `saver` only keeps the five most recent checkpoints by default to save storage space. However, if we want to keep more than five recent checkpoint files, we can provide an optional argument `max_to_keep=n` when we initialize the `saver`, where `n` is an integer specifying the number of the most recent checkpoint files we want to keep.

Now that we learned how to save TensorFlow `Variables`, let us see how we can restore them. Assuming that we started a fresh computational session, we need to specify the graph first. Then, we can use the `saver`'s `restore` method to restore our variables as shown below:

In [11]:

```
g = tf.Graph()

with g.as_default() as g:

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)
    x = tf.constant(1., dtype=tf.float32)

    update_tf_x = tf.assign(tf_x, tf_x + x)

    # initialize a Saver, which gets all variables
    # within this computation graph context
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    saver.restore(sess, save_path='./my-model.ckpt')
    result = sess.run(update_tf_x)
    print(result)
```

Out [11]:

```
[[ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]
```

Notice that the returned values of the `tf_x` `Variable` are now increased by a constant of two, compared to the values in the computational graph. The reason is that we ran the graph one time before we saved the variable,

```python
with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(update_tf_x)

    # save the model
    saver.save(sess, save_path='./my-model.ckpt')
```

and we ran it a second time when after we restored the session.

Similar to the example above, we can reload one of our checkpoint files by providing the desired checkpoint suffix (here: `-90`, which is the index of our last checkpoint):

In [12]:

```python
with tf.Session(graph=g) as sess:
    saver.restore(sess, save_path='./my-model-multiple_ckpts.ckpt-90')
    result = sess.run(update_tf_x)
    print(result)
```

Out [12]:

```
[[ 93.  94.]
 [ 95.  96.]
 [ 97.  98.]]
```

In this section, we merely covered the basics of saving and restoring TensorFlow models. If you want to learn more, please take a look at the official API documentation[9] of TensorFlow's `Saver` class.

Naming TensorFlow Objects

When we create new TensorFlow objects like `Variables`, we can provide an optional argument for their `name` parameter – for example:

[9]https://www.tensorflow.org/api_docs/python/tf/train/Saver

```
tf_x = tf.Variable([[1., 2.],
                    [3., 4.],
                    [5., 6.]],
                   name='tf_x_0',
                   dtype=tf.float32)
```

Assigning names to `Variables` explicitly is not a requirement, but I personally recommend making it a habit when building (more) complex models. Let us walk through a scenario to illustrate the importance of naming variables, taking the simple example from the previous section and add new variable `tf_y` to the graph:

In [13]:

```
g = tf.Graph()

with g.as_default() as g:

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)

    tf_y = tf.Variable([[7., 8.],
                        [9., 10.],
                        [11., 12.]], dtype=tf.float32)

    x = tf.constant(1., dtype=tf.float32)
    update_tf_x = tf.assign(tf_x, tf_x + x)
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(update_tf_x)

    saver.save(sess, save_path='./my-model.ckpt')
```

The variable `tf_y` does not do anything in the code example above; we added it for illustrative purposes, as we will see in a moment. Now, let us assume we started a new computational session and loaded our saved my-model into the following computational graph:

In [14]:

```
g = tf.Graph()

with g.as_default() as g:

    tf_y = tf.Variable([[7., 8.],
                        [9., 10.],
                        [11., 12.]], dtype=tf.float32)

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]], dtype=tf.float32)

    x = tf.constant(1., dtype=tf.float32)
    update_tf_x = tf.assign(tf_x, tf_x + x)
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    saver.restore(sess, save_path='./my-model.ckpt')
    result = sess.run(update_tf_x)
    print(result)
```

What results do you expect after running the code snippet above?

Out [14]:

```
[[  8.   9.]
 [ 10.  11.]
 [ 12.  13.]]
```

Unless you paid close attention on how we initialized the graph above, this result above surely was not the one you expected. What happened? Intuitively, we expected our session to `print`

```
[[ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]
```

The explanation behind this unexpected `result` is that we reversed the order of `tf_y` and `tf_x` in the graph above. TensorFlow applies a default naming scheme to all operations in the computational graph, unless we use do it explicitly via the `name` parameter – or in other words, we confused TensorFlow by reversing the order of two similar objects, `tf_y` and `tf_x`.

To circumvent this problem, we= could give our variables specific names – for example, 'tf_x_0' and 'tf_y_0':

In [15]:

```
import tensorflow as tf

g = tf.Graph()

with g.as_default() as g:

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]],
                       name='tf_x_0',
                       dtype=tf.float32)

    tf_y = tf.Variable([[7., 8.],
                        [9., 10.,
                         ]],
                       name='tf_y_0',
                       dtype=tf.float32)

    x = tf.constant(1., dtype=tf.float32)
    update_tf_x = tf.assign(tf_x, tf_x + x)
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(update_tf_x)

    saver.save(sess, save_path='./my-model.ckpt')
```

Then, even if we flip the order of these variables in a new computational graph, TensorFlow knows which values to use for each variable when loading our model – assuming we provide the corresponding variable names:

In [16]:

```
g = tf.Graph()

with g.as_default() as g:

    tf_y = tf.Variable([[7., 8.],
                        [9., 10.,
                         ]],
                       name='tf_y_0',
                       dtype=tf.float32)

    tf_x = tf.Variable([[1., 2.],
                        [3., 4.],
                        [5., 6.]],
                       name='tf_x_0',
                       dtype=tf.float32)

    x = tf.constant(1., dtype=tf.float32)
    update_tf_x = tf.assign(tf_x, tf_x + x)
    saver = tf.train.Saver()

with tf.Session(graph=g) as sess:
    saver.restore(sess, save_path='./my-model.ckpt')
    result = sess.run(update_tf_x)
    print(result)
```

Out [16]:

```
[[ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]]
```

CPU and GPU

Please note that all code examples in this book, and all TensorFlow operations in general, can be executed on a CPU. If you have a GPU version of TensorFlow installed, TensorFlow will automatically execute those operations that have GPU support on GPUs and use your machine's CPU, otherwise. However, if you wish to define your computing device manually, for instance, if you have the GPU version installed but want to use the main CPU for prototyping, we can run an active section on a specific device using the `with` context as follows

```python
with tf.Session() as sess:
    with tf.device("/gpu:1"):
```

where

- "/cpu:0": The CPU of your machine.
- "/gpu:0": The GPU of your machine, if you have one.
- "/gpu:1": The second GPU of your machine, etc.
- etc.

You can get a list of all available devices on your machine via

```python
from tensorflow.python.client import device_lib

device_lib.list_local_devices()
```

For more information on using GPUs in TensorFlow, please refer to the GPU documentation at https://www.tensorflow.org/how_tos/using_gpu/.