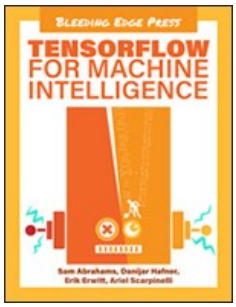

TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli

Chapter 8: Helper Functions, Code Structure, and Classes

Overview

In this short chapter, we provide code snippets and explanations for various helper functions and classes used throughout this book.

Ensure a directory structure

Let's start with a little prerequisite that we need when interacting with the file system. Basically, every time we create files, we have to ensure that the parent directory already exists. Neither our operating system nor Python does this for us automatically, so we use this function that correctly handles the case that some or all of the directories along the path might already exist.

```
import errno
import os

def ensure_directory(directory):
    """
    Create the directories along the provided directory path that do not ex-
ist.
    """
    directory = os.path.expanduser(directory)
    try:
        os.makedirs(directory)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise e
```

Download function

We download several datasets throughout the book. In all cases, there is shared logic and it makes sense to extract that into a function. First, we determine the filename from the URL if not specified. Then, we use the function defined above to ensure that the directory path of the download location exists.

```
import shutil
from urllib.request import urlopen

def download(url, directory, filename=None):
    """
    Download a file and return its filename on the local file system. If the
    file is already there, it will not be downloaded again. The filename is
    derived from the url if not provided. Return the filepath.
    """
    if not filename:
        _, filename = os.path.split(url)
    directory = os.path.expanduser(directory)
    ensure_directory(directory)
    filepath = os.path.join(directory, filename)
    if os.path.isfile(filepath):
        return filepath
    print('Download', filepath)
    with urlopen(url) as response, open(filepath, 'wb') as file_:
        shutil.copyfileobj(response, file_)
    return filepath
```

Before starting the actual download, check if there is already a file with the target name in the download location. If so, skip the download since we do not want to repeat large downloads unnecessarily. Finally, we download the file and return its path. In case you need to repeat a download, just delete the corresponding file on the file system.

Disk caching decorator

In data science and machine learning, we handle large datasets that we don't want to preprocess again every time we make changes to our model. Thus, we want to store intermediate stages of the data processing in a common place on disk. That way, we can check if the file already exists later.

In this section we will introduce a function decorator that takes care of the caching and loading. It uses Python's `pickle` functionality to serialize and deserialize any return values of the decorated function. However, this also means it only works for dataests fitting into main memory. For larger dataests, you probably want to take a look at scientific dataset formats like HDF5.

We can now use this to write the `@disk_cache` decorator. It forwards function arguments to the decorated function. The function arguments are also used to determine whether a cached result exists for this combination of arguments. For this, they get hashed into a single number that we prepend to the filename.

```
import functools
import os
import pickle

def disk_cache(basename, directory, method=False):
    """
    Function decorator for caching pickleable return values on disk. Uses a
    hash computed from the function arguments for invalidation. If 'method',
    skip the first argument, usually being self or cls. The cache filepath is
    'directory/basename-hash.pickle'.
    """
    directory = os.path.expanduser(directory)
    ensure_directory(directory)

    def wrapper(func):
        @functools.wraps(func)
        def wrapped(*args, **kwargs):
            key = (tuple(args), tuple(kwargs.items()))
            # Don't use self or cls for the invalidation hash.
            if method and key:
                key = key[1:]
            filename = '{}-{}.pickle'.format(basename, hash(key))
            filepath = os.path.join(directory, filename)
            if os.path.isfile(filepath):
                with open(filepath, 'rb') as handle:
                    return pickle.load(handle)
            result = func(*args, **kwargs)
            with open(filepath, 'wb') as handle:
                pickle.dump(result, handle)
            return result
        return wrapped

    return wrapper
```

Here is an example usage of the disk cache decorator to save the data processing pipeline.

```
@disk_cache('dataset', '/home/user/dataset/')
def get_dataset(one_hot=True):
    dataset = Dataset('http://example.com/dataset.bz2')
    dataset = Tokenize(dataset)
    if one_hot:
        dataset = OneHotEncoding(dataset)
    return dataset
```

For methods, there is a `method=False` argument that tells the decorator whether to ignore the first argument or not. In methods and class methods, the first argument is the object instance `self` that is different for every program run and thus shouldn't determine if there is a cache available. For static methods and functions outside of classes, this should be `False`.

Attribute Dictionary

This simple class just provides some convenince when working with configuration objects. While you could perfectly well store your configurations in Python dictionaries, it is a bit verbose to access their elements using the `config['key']` syntax.

```
class AttrDict(dict):

    def __getattr__(self, key):
        if key not in self:
            raise AttributeError
        return self[key]

    def __setattr__(self, key, value):
        if key not in self:
            raise AttributeError
        self[key] = value
```

This class, inheriting from the built-in `dict`, allows to access and change existing elemets using the attribute syntax: `config.key` and

`config.key = value`. You can create attribute dictionaries by either passing in a standard dictionary, passing in entries keyword arguments, or using `**locals()`.

```
parmas = AttrDict({
    'key': value,
})

params = AttrDict(
    key=value,
)

def get_params():
    key = value
    return AttrDict(**locals())
```

The `locals()` built-in just returns a mapping from all local variable names in the scope to their values. While some people who are not that familiar with Python might argue that there too much magic going on here, this technique also provides some benefits. Mainly, we can have configuration entries that rely on ealier entries.

```
def get_params():
    learning_rate = 0.003
    optimizer = tf.train.AdamOptimizer(learning_rate)
    return AttrDict(**locals())
```

This function returns an attribute dictionary containing both the `learning_rate` and the `optimizer`. This would not be possible within the declaration of a dictionary. As always, just find a way that works for you (and your colleagues) and use that.

Lazy property decorator

As you learned, our TensorFlow code defines a compute graph rather than performing actual computations. If we want to structure our models in classes, we cannot directly expose its outputs from functions or properties, since this would add new operations to the graph every time. Let's see an example where this becomes a problem:

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target

    @property
    def prediction(self):
        data_size = int(self.data.get_shape()[1])
        target_size = int(self.target.get_shape()[1])
        weight = tf.Variable(tf.truncated_normal([data_size, target_size]))
        bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
        incoming = tf.matmul(self.data, weight) + bias
        prediction = tf.nn.softmax(incoming)
        rediction

    @property
    def optimize(self):
        cross_entropy = -tf.reduce_sum(self.target, tf.log(self.prediction))
        optimizer = tf.train.RMSPropOptimizer(0.03)
        optimize = optimizer.minimize(cross_entropy)
        return optimize

    @property
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
        error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
        return error
```

Using an instance of this from the outside creates a new computation path in the graph when we access `model.optimize`, for example. Moreover, this internally calls `model.prediction` creating new weights and biases. To address this design problem, we introduce the following `@lazy_property` decorator.

```
import functools

def lazy_property(function):
    attribute = '_lazy_' + function.__name__
```

```
    @property
    @functools.wraps(function)
    def wrapper(self):
        if not hasattr(self, attribute):
            setattr(self, attribute, function(self))
        return getattr(self, attribute)
    return wrapper
```

The idea is to define a property that is only evaluated once. The result is stored in a member called like the function with some prefix, for example `_lazy_` here. Subsequent calls to the property name then return the existing node of of the graph. We can now write the above model like this:

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target
        self.prediction
        self.optimize
        self.error

    @lazy_property
    def prediction(self):
        data_size = int(self.data.get_shape()[1])
        target_size = int(self.target.get_shape()[1])
        weight = tf.Variable(tf.truncated_normal([data_size, target_size]))
        bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
        incoming = tf.matmul(self.data, weight) + bias
        return tf.nn.softmax(incoming)

    @lazy_property
    def optimize(self):
        cross_entropy = -tf.reduce_sum(self.target, tf.log(self.prediction))
        optimize = tf.train.RMSPropOptimizer(0.03)
        return optimizer.minimize(cross_entropy)

    @lazy_property
    def error(self):
        mistakes = tf.not_equal(
    tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
return tf.reduce_mean(tf.cast(mistakes, tf.float32))
```

Lazy properties are a nice tool to structure TensorFlow models and decompose them into classes. It is useful for both node that are needed from the outside and to break up internal parts of the computation.

Overwrite Graph Decorator

This function decorator is very useful when you use TensorFlow in an interactive way, for example a Jupyter notebook. Normally, TensorFlow has a default graph that it uses when you don't explicitly tell it to use something else. However, in a Jupyter notebook the interpreter state is kept between runs of a cell. Thus, the initial default graph is still around.

Excecuting a cell that defines graph operations again will try to add them to the graph they are already in. Fortunately, TensorFlow throws an error in this case. A simple workaround is to *restart the kernel and run all cells again*.

However, there is a better way to do it. Just create a custom graph and set it as default. All operations will be added to that graph and if you run the cell again, a new graph will be created. The old graph is automatically cleaned up since there is no reference to it anymore.

```
def main():
    # Define your placeholders, model, etc.
    data = tf.placeholder(...)
    target = tf.placeholder(...)
    model = Model()

with tf.Graph().as_default():
    main()
```

Even more conveniently, put the graph creation in in a decorator like this and decorate your main function with it. This main function should define the whole graph, for example defined the placeholders and calling another function to create the model.

```
import functools
import tensorflow as tf
```

```
def overwrite_graph(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        with tf.Graph().as_default():
            return function(*args, **kwargs)
    return wrapper
```

This makes the example above a bit easier:

```
@overwrite_graph
def main():
    # Define your placeholders, model, etc.
    data = tf.placeholder(...)
    target = tf.placeholder(...)
    model = Model()

main()
```

This is the end of the chapter, but take a look at the next chapter to read our wrapup of the book.