Danijar Hafner

Structuring Your TensorFlow Models

Apr 26, 2016

Defining your models in TensorFlow can easily result in one huge wall of code. How to structure your code in a readable and reusable way? For the inpacient of you, here is the link to a working example gist.

Defining the Compute Graph

It's sensible to start with one class per model. What is the interface of that class? Usually, your model connects to some input *data* and *target* placeholders and provides operations for *training* and *evaluation*.

```
class Model:

    def __init__(self, data, target):
        data_size = int(data.get_shape()[1])
        target_size = int(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(data, weight) + bias
        prediction = tf.nn.softmax(incoming)
        cross_entropy = -tf.reduce_sum(target, tf.log(prediction))
        self._optimize = tf.train.RMSPropOptimizer(0.03).minimize(cross_ent
        mistakes = tf.not_equal(tf.argmax(target, 1), tf.argmax(prediction,
        self._error = tf.reduce_mean(tf.cast(mistakes, tf.float32))

    @property
    def optimize(self):
        return self._optimize

    @property
    def error(self):
```

```
        return self._error
```

This is basically, how models are defined in the TensorFlow codebase. However, there are some problems with it. Most notably, the whole graph is define in a single function, the constructor. This is neither particularly readable nor reusable.

Using Properties

Just splitting the code into functions doesn't work, since every time the functions are called, the graph would be extended by new code. Therefore, we have to ensure that the operations are added to the graph only when the function is called for the first time. This is basically lazy-loading.

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target
        self._prediction = None
        self._optimize = None
        self._error = None

    @property
    def prediction(self):
        if not self._prediction:
            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_siz
            bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
            incoming = tf.matmul(self.data, weight) + bias
            self._prediction = tf.nn.softmax(incoming)
        return self._prediction

    @property
    def optimize(self):
        if not self._optimize:
            cross_entropy = -tf.reduce_sum(self.target, tf.log(self.predict
            optimizer = tf.train.RMSPropOptimizer(0.03)
            self._optimize = optimizer.minimize(cross_entropy)
```

```
        return self._optimize

    @property
    def error(self):
        if not self._error:
            mistakes = tf.not_equal(
                tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
            self._error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
        return self._error
```

This is must better than the first example. Your code now is structured into functions that you can focus on individually. However, the code is still a bit bloated due to the lazy-loading logic. Let's see how we can improve on that.

Lazy Property Decorator

Python is a quite flexible language. So let me show you how to strip out the redundant code from the last example. We will use a decorator that behaves like `@property` but only evaluates the function once. It stores the result in a member named after the decorated function (prepended with an underscore) and returns this value on any subsequent calls. If you haven't used custom decorators yet, you might also want to take a look at this guide.

```
import functools


def lazy_property(function):
    attribute = '_' + 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
```

Using this decorator, our example simplifies to the code below.

```
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))
        optimizer = 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))
```

Note that we mention the properties in the constructor. This way the full graph is ensured to be defined by the time we run `tf.initialize_variables()`.

We can now define models in a structured and compact way. This works well for me. If you have any suggestions or questions, feel free to use the comment section.

Danijar Hafner

Danijar Hafner
mail@danijar.com

danijar
danijarh

I'm a Python and C++ developer from Berlin with strong interest in machine intelligence and cognitive computing. I recently released a neural network library but I like creating new things in general.