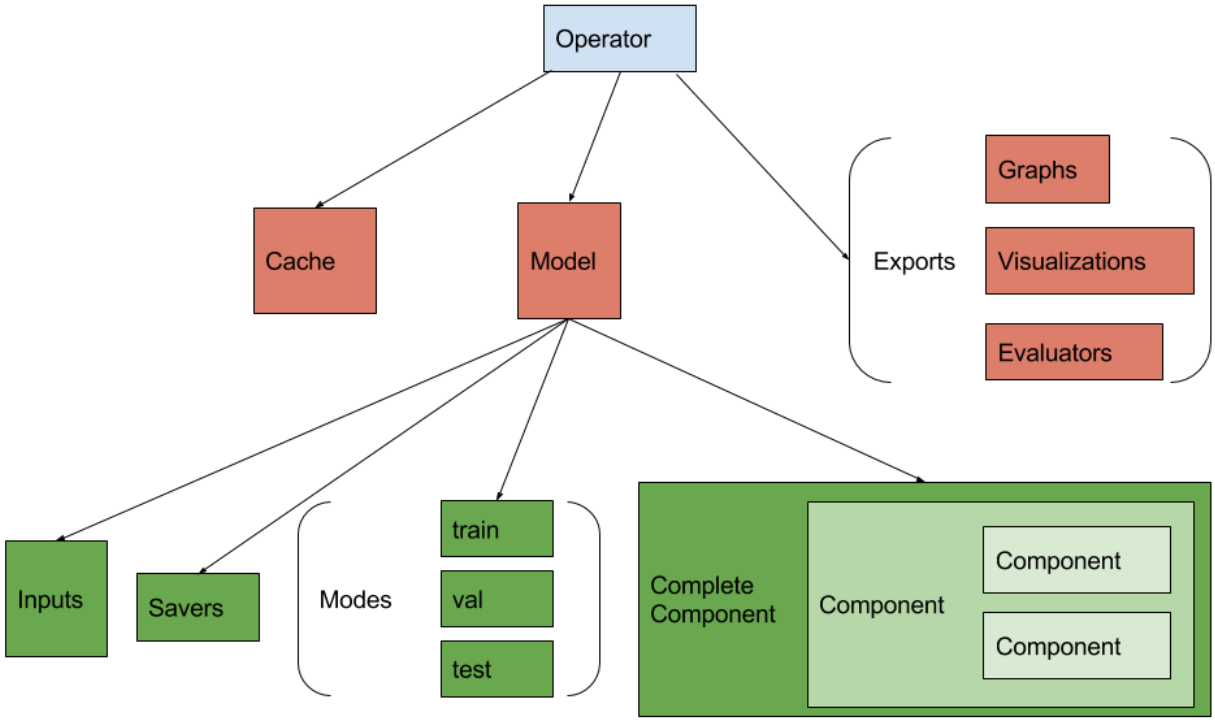
There are usually a lot of moving parts in a brain (deep learning) experiment, which, if you're trying to run experiments, can quickly lead to spaghetti code. Anyway, this is an introduction to a general framework I wrote and use for all my experiments.

Generally, an experiment requires that you define some neural networks, and then train them. A neural network can have many parts, but the most important parts are usually the inputs, the outputs, and the learned parameters.

Once you have your neural network, you generally want to *do* things to it. This includes training it, saving it, loading it from a save file, adding new parts, removing parts, visualizing the outputs, and so on.



The ultimate "controller" in my setup is the Operator class. It's responsible for

1. The actual logic of how to run the neural network
    a. Contains the loop which runs the training
    b. Contains the loop which runs the evaluation and testing
    c. Periodically saves the learned parameters
    d. Responsible for taking outputs of the network and *exporting* them via Exports objects

    e. Handles all the overhead of setting up the computing environment
2. The Exports abstract class deals with processing the outputs of the model
    a. TB is an exports class which graphs the loss curves of the model
    b. Visualizations writes images to disk
    c. Evaluators are exports which objectively evaluate the results by computing some metrics
3. The cache can be used by the operator to speed up access to input data
4. The Model object is the most important object owned by the Operator. It contains three parts
    a. The Modes, which are basically instructions on how to run the model. For example, if the operator is told to run in “train” mode, the train Mode object would contain information on where to get the correct inputs, and which computation nodes to run.
    b. The Inputs object, which just contains the logic for preprocessing and proving inputs.
    c. The Savers object, which contains all the logic for saving and loading different parts of the neural network to different locations on disk.
    d. A Component object, which contains the neural network definitions
        i. Tensorflow provides functionality to create a large number of “tensor” objects, which can be run. A component is an abstraction for grouping useful tensors together. For example, a component may contain a “predictions” group, or a “inputs” group of tensors. (I also refer to groups as endpoints).
        ii. Components can be nested -- This isn’t built-in, but there’s nothing which forbids a component object from using another other components, and it’s a useful pattern
        iii. Components need to follow a Spec (not pictured), which an object specifying which groups of tensors a component must have. The component is then checked against its spec using some code I wrote. Specs can be nested! This allows easy combination and recombination of components, since the spec provides a clear definition of the *endpoints* that a component supports.
        iv. Though it’s not formally defined as code yet, there’s a concept of “Complete Component”. Basically, usually the outermost component owned by the model needs to support a fairly large number of endpoints. On the other hand, the inner components may have only one or two endpoints. So it’s a useful pattern to have a thin wrapper Component which does nothing but assembles all the endpoints of the other components into a complete set of endpoints. This increases the ease of changing things around.

Now I’ll provide an example of how this framework helps. Suppose you had a neural network which predicted the speed of an object. Now, you want to add a second network to your model

-- one which predicts the category of that object, then feeds that prediction into the speed network.

Here's the list of changes I'd probably have to make if I wrote the code without using this framework

1. Define the new categorization network
2. Define a new speed network which can take as input the output of the categorization prediction
3. Add a boolean switch to pick between the using the previous speed network, or the two new networks
4. Modify the saving logic to save the new network to a different location -- but. This logic also needs to take into account the boolean switch from step 3, since the two speed networks are probably incompatible now.
5. If the entire neural network part is defined in a function (and it usually is with spaghetti code like this), then modify the output of the function so instead of returning like 20 outputs, it returns 21 -- the rest, plus the new categorization prediction
6. Modify the loss function / training objective to include the new prediction
7. Pass the output -- item 21 of the output tuple -- to the function which does visualization. That function needs to be modified so it takes 21 items instead of 20 now.
8. Etc. There's a lot of stuff I skipped over/forgot here

As you can see, this is pretty crazy and prone to mistakes. Hopefully, this framework would streamline the process in the following ways:

1. If you define the new speed and categorization networks as different Components, and then register them using a function built-in to the framework, then the saver will automatically be able to save them correctly without further modifications to the code
2. Instead of modifying all your function signatures to accommodate the additional output, you just need to modify or add an additional endpoint to your component, and the corresponding spec. Since endpoints can be nested or composed, this is less changes. In addition, because of the Specs, if you make a mistake here, it would be quickly caught at compile-time.
3. It doesn't reduce the amount of code written -- in fact the overhead is a bit higher, but there is something to be said about knowing which class you need to go to in and modify in order to get something working, instead of looking through a huge pile of code. For example, the fact that plotting statistics, visualizing outputs, and computing metrics for evaluation all have the same interface makes it easy to add on a new handler which processed the network outputs.