Semantic Segmentation Project

This is the elective project involving semantic segmentation for Udacity's self-driving car program.

Running the Project

The project can be run with ease by invoking `python main.py`. The way it is set up in this repo, using a GTX 1060 it takes about 15 minutes to train.

Initialization and Tests

When run, the project invokes some tests around building block functions that help ensure tensors of the right size and description are being used. I updated test method test_safe() to report the name of the function invoked since this is a bit more useful with the tests. I also cleaned up the test code and allowed for printing rather than suppressing it, as this aids in ad hoc debugging. The project has been restructured in a class to facilitate better passing of hyperparameters for training; more on this in a later section.

Once the tests are complete, the code checks for the existance of a base trained VGG16 model. If it is not found locally it downloads it. Once it is available, it is used as the basis for being reconnected with skip layers and recapping the model with an alternative top end for semantic segmentation.

Defining Skip Layers in VGG16

VGG16 is a good start for an image semantic segmentation mechanism. Simply replacing the classifier with an alternative classifier based on flattened image logits is a good start. However, this tends to not have as good performance in terms of resolution as we would desire. The reason for this is the structure of the model with the reduction of resolution. One way of improve this performance is to merge the outputs of the layers, scaled to the right resolution. This works by producing cummulative layer activations and the ability, due to the convolutional kernels, to spread this activation locally to neighboring pixels. This change in layer connectivity is called `skip layers` and the result is more accurate segmentation.

In this exercise we are focusing on layers 3, 4 and 7. First, each of these layers have a 1x1 convolution layer added. Next, layers 7 and 4 are merged, with layer 7 being upsampled using the `conv2d_transpose()` function. The result of this cummulative layer was added to layer 3, the cummulative layer being upsampled first. The final result is upsampled again. What this basically accomplishes is creating a skip-layer encoder-decoder network. The critical part, the encoder, is the pre-trained VGG16 model, and it is our task to train the smaller decoder aspect to accomplish the semantic segmentation that we are trying to achieve.

Optimization

The network is trained using cross entropy loss as the metric to minimize. The way this is accomplished is essentially to flatten both the logits from the last decoder layer into a single dimensional label vector. The cross entropy loss is computed against the correct label image, itself also flattened. I used an Adam optimizer to minimize this.

One thing I had wanted to try was to use intersection over union as a metric to minimize. Rather, the negative of it since that is the proper quantity to minimize. Unfortunately, the TensorFlow intersection over union function is not one that is compatible with minimization.

Training

At this point the network is trained. This is pretty normal, though I added `tqdm` so that there is a little nicer output reporting while going through batches and epochs. This can be seen in the `train_nn()` function. You will notice that some of the hyperparameters are conveyed as properties. As with other methods, some of the passed in arguments are actually tensors. Using the properties made for a nicer way of grouping hyperparameters; more information noted later. The basic training process is simply going through the defined epochs, batching and training. The loss reported is an average over the returned losses for all batches in an epoch.

Once the training is complete, the test images are processed using the provided helper function. Not much to say about this, the code is the expected one liner. The model is also saved later in the process. This is accomplished by using a TensorFlow `Saver()`. The metadata and the graph definition are written out. Reason for this is because we can use an optimizer on the graph for use in faster inference applications. This wasn't directly germane to the goal of the exercise, but it is useful to know how to do.

Hyperparameter Selection

There are several parameters that were relevat to expose easily as hyperparameters:

- Learning rate
- Dropout
- Epochs
- Batch size
- Standard deviation of convolution layer initializer

The learning rate that was found to work well was pretty standard, `0.0001`. Dropout in the range of about `0.5` through `0.8` worked pretty well, though I felt that there was more rapid convergence with the lower values. I chose a batch size of `10` because this seemed to facilitate better generalization and more rapid reduction of loss values. There was no constraint regarding training time that was meaningful as a reason for using a larger batch size. The standard deviation used in the initializer tended to need to be small, but not too small. `0.01` worked well. Values larger tended to increase the initial loss and training time. The number of epochs was selected based on watching the convergence of the loss. After about 15 epochs the loss didn't go down consistently. I used 20 epochs.

Results

The results are *surprisingly good*. The image at the top is some of the output. Of the hundreds of test images, there are very few that do not have fairly adequate road coverage. Places where it seems to fail most include:

- Small regions, such as between cars or around bicycles
- Wide expanses of road with poorly defined boundaries
- Road forks with dominant lane lines

Surprisingly, it works very well at distinguishing between roads and intersecting railroad tracks. This is probably related to why it does not segment the road at intersections with dominant lane lines as well, as the high contrast parallel lines have few examples in the training set.

Code Style

The way the base project was structured with the expectation of testing lent itself to some messy code, specifically regarding injection of training parameters. The tests are a great way to ensure the model is appropriate, but they tend to enforce an interface, and that initially made coordination of injecting parameters fairly ugly (e.g. leaving `learning_rate` and `keep_prob` in the `train_nn()` function). This is further complicated by the fact that TensorFlow in general takes a lot of arguments to functions by its nature, lending itself to somewhat inelegant code.

The way I solved this was by encapsulating my code in a class, and writing an initializer that would take a dictionary argument and create properties from them. This way I could pass all of the relevant hyperparameters in the constructor. One ramification of this was creating a method that explicitly runs the tests rather than having them located after each function declaration. This makes for a fairly nice API using the class where the tests are run after instantiation, followed by the full training run. Parameters that were not hyperparameters were simply added as static class variables.