EECS 600 final project

Abbreviations

HP = hyper-parameter

TF = TensorFlow

Dataset

A set of 48600 depth scans of 25 different household objects taken at varying positions and orientations. The images measure 34 pixels square and have a few points randomly set to zero. The transformation between the camera and the table on which the objects were placed remained constant. This produced a clearly visible gradient in the resulting greyscale images, as can be seen in the examples.

Figure 1: Rubber ducky

Figure 2: Razor

Images can be extracted from the dataset with `./see_data.py` (which picks one at random each time) and can be seen by running `./serve.sh` and visiting `http://<server-hostname>:8000/<filename>.png` in one's web browser. For this assignment and setup, that expands to `http://o405-u14.case.edu:8000/first_razor.png` for the razor image.

Network architecture and approach

At first, a simple two-layer approach was tried. This is the approach used by the TF introductory example with the MNIST dataset. It had an accuracy of about 20%, which is better than random chance (4%) but worse than acceptable (about 80% or so).

Then, a LeNet-style approach was tried. It has two convolutional layers (both with overlapped max pooling), followed by three fully-connected layers. This can

be found in the function `inference()` in `kdd.py`, which was written to be highly compact and readable. Any confusion in this paragraph should be cleared up by reading it. Each layer uses a ReLU as its activation function (for speed), as well as decaying weights. The values for the decay and learning rate HPs came from the TF example for the CIFAR-10 dataset, although since the machines in the Jennings lab are on TF v0.10.0rc0, the training method used by that example (MonitoredTrainingSession) does not directly work.

The above approach worked quite well, with an acceptable accuracy. Accordingly, a search of the HP space was conducted (see `run.py` for details). Since there were six HPs, only two values were used for each one in order to save on execution time.

Platform

All models were trained on the CPUs in the Jennings lab (via SSH). Specs: 4GB of system memory; Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz (eight threads). Tensorflow v0.10.0rc0 was used to implement the networks. Although a substantial amount of effort was exerted to run TF on a GPU node on the CWRU High-Performance Compute Cluster (HPC) (see last section), this was ultimately unsuccessful.

Results

Model HPs are expressed as a six-tuple with the following elements:

0. Side length of the square kernel used by the first convolution layer.
1. Number of feature maps used by the first convolution layer.
2. Side length of the square kernel used by the second convolution layer.
3. Number of feature maps used by the second convolution layer.
4. Number of elements for the first fully-connected layer.
5. Number of elements for the second fully-connected layer.

These are also the parameters to `kdd.inference()`. Note that the third fully-connected layer is of known dimensions (must match FC2 and number of classes) and is thus not a HP.

Default HPs (5, 64, 5, 128, 192, 96)

These numbers were chosen somewhat arbitrarily to be the default values for structuring the network. They are meant to be reminiscent of LeNet.

This run had a testing set accuracy of 86.4% and a training time of 51 minutes on the Jennings machine. It was hard to beat, although that did happen.

Other parameters

An exploration of HP-space can be done by executing `run.py`. It repeatedly makes calls to the main `fully_connected_feed.py` with different sets of HPs.

The following sets of HPs had test accuracies of better than 89%. All values refer to the testing set on the 2000th step.

- 89.3% (3, 128, 3, 128, 192, 96)
- 89.3% (3, 128, 5, 128, 192, 96)
- 91.8% (3, 128, 5, 128, 192, 48)
- 93.0% (5, 64, 5, 64, 192, 96)
- 95.5% (3, 64, 3, 64, 192, 96)

That last value might have been a fluke, but is impressive nonetheless.

Failed approaches

Here is a partial explanation of why this project took so long.

Hardware

I have access to the following Linux machines:

- Windows Subsystem for Linux on my laptop (or a VM)
- SSHing into the Jennings lab
- SSHing into HPC (access is through employer)

Here are the problems with each:

- GPU is AMD and WSL cannot access it anyways
- No GPU
- No TF (see below for attempts to fix that) and Caffe requires a K40 node, which are all in use. Attempting to run Caffe on a Fermi node (two Tesla M2070s each) resulted in a crash.

The machines in the Jennings lab are more powerful than my laptop, and WSL would be entirely on the command line anyways, so there is no reason to use local over the Jennings lab.

HPC does not have Tensorflow installed. Worse, its libraries are out of date, and (to my knowledge) the administrators do not give out root access. Therefore, the Jennings lab was used for initial development.

TF versions

The machines in the Jennings lab have TF v0.10.0rc0 installed. The CIFAR-10 tutorial/example uses features only found in later versions, such as the current v0.12.0. Attempting to use the new features without knowing this cost some amount of time.

Running TF on HPC

The big time sink. In theory, this should be easy, since TF has a binary distribution. In practice, the following libraries are out of date:

- CUDA is 7.5, need 8.0
- cuDNN is missing, need >=5
- glibc is 2.12, need >=2.17

The first two are relatively easy to solve (local copy) since one must specify the path to them anyways. The third is the time sink. The following methods were tried, all with a Python 3.5.2 virtualenv on gpu002.

Source: various answers to http://stackoverflow.com/q/33655731

- Weakening the dependencies on libc (setting flags) and making own stub functions
  - Only worked for v0.6; for 0.12, too many binaries needed patching (finding locations was easy)
- Stealing the glibc from a CentOS 7 ISO and patching the paths
- Compiling own glibc 2.20 to match binutils on HPC
  - GCC tried to use unavailable instruction (vinserti1, similar to https://forums.gentoo.org/viewtopic-t-1021574-start-0-postdays-0-postorder-asc-highlight-.html)
- Installing TF from sources
  - Bazel uses Java 8, HPC only has 7
  - Bazel does not like being run on compute nodes (caching interferes with PanFS)
  - Configure script (`./configure`) is manual
  - Building protobuf tools requires newer version of glibcxx, the C++ equivalent of glibc.
- Other things that I have forgotten about