Metehan Ozten

CS291K MP2 Report

Implementation

The implementation consists of several crucial components. Initially before the data is even preprocessed, the script checks for the dataset and then if it doesn't exist, it is downloaded and untarred. Once the dataset has been located by the script, the dataset is shuffled and the 'train.bin' file contained is broken up into two files one with 45000 labels and images, and the other with 5000 labels and images. These two files are then used to provide data for the training and validation sets respectively. In particular these two files are named train-split.bin and val-split.bin.

Once the files are broken into their respective subsets, the training data is preprocessed to remove the per-image-mean from each image and adjusting the pixel values by the variance of the pixel values for that specific image, and then the training set is artificially expanded by performing various random distortions to the images, such as random left-right flipping, random 24x24 sub-cropping and and random brightness and contrast distortions.

After the images are preprocessed they are fed through our CNN(the architecture of which will be discussed in the next section).

After the training is performed, three evaluation batches of 10000 examples are performed, one on the training set, one on the validation set and one of the test set. Then a percentage is computed on each of the following

Note: Due to various implementation methods that I've used (such as splitting the train.bin in train-split.bin and val-split.bin) and other file-system interactions this program should be executed with either sudo or in state where the program has full r/w privileges over the directory that the program is contained in AS WELL AS the directory that contains the training/test data (passed in as argv[1] for redo.py)

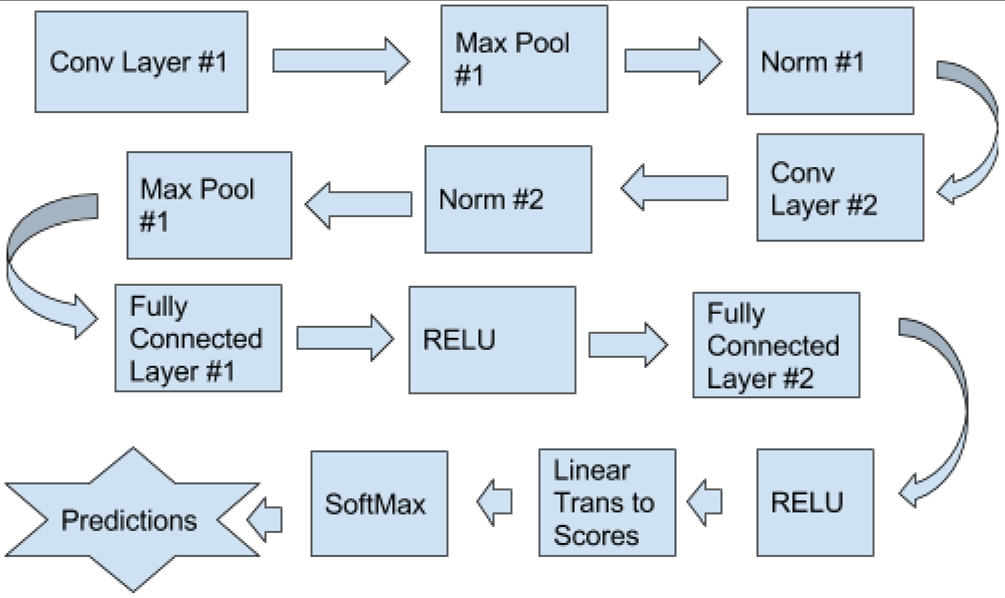
Also, the path that is passed into the first arg of redo.py MUST be either a full path or a relative path (./dataset).

Architecture

The architecture was tested over varying levels of convolutional filter and fully connected layers. Each evaluation of the unique architectures was performed over 2000 iterations, due to time limitations of writing this report. The architecture used in this project is also shown below.

Model Information	Accuracy

Arch. #	#Conv/Pool Layers	1st Conv/Pool Filter Size	2nd Conv/Pool Filter Size	# Fully Connected	First Fully Connected Layer # Neurons	First Fully Connected Layer # Neurons	Final Loss	Train %	Val %	Test %
1	2	5x5/3x3	5x5/3x3	2	384	192	2.98	.235	.235	.236
2	2	5x5/3x3	5x5/3x3	2	768	384	3.33	.338	.341	.328
3	2	5x5/3x3	5x5/3x3	2	192	96	3.23	.051	.049	.050
4	2	7x7/3x3	5x5/3x3	2	768	384	3.49	.358	.350	.344
5	2	5x5/5x5	5x5/3x3	2	768	384	3.31	.362	.360	.353
6	2	5x5/3x3	5x5/3x3	1	768	N/A	5.88	.458	.443	.442



Model Building

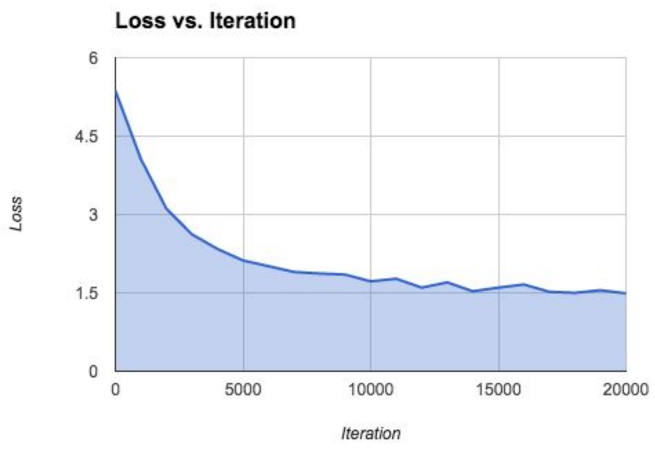
How is training performed?

Since the training was rather CPU intensive (and due to the fact that for some reason I could not get TensorFlow working on my laptop), all training was performed on the cloud (in particular on Google Compute Engine). I had three VM instances setup, one with 16 CPUs and two others with 4 CPUS each (Google capped me at 24 CPUS). Once I had my implementation completed and working on my 16 CPU cloud box, I would run modifications of the architecture with different parameters on the other boxes.

Num Iters

To determine the optimal number of iterations, I first set the number of iterations to a relatively high value (20000) and then graphed the loss as a function of the number of iterations to determine the point at which the loss converges. This took a relatively long time, but not that long on the 16 CPU box (~1 hour).

Below is a graph showing the loss as a function of the current iteration running on architecture #1.



Convergence occurs essentially at 10000 iterations. This particular run used architecture #1.

Number of layers

I believe that I am at the optimal amount of layers, the fewer the fully connected layers, the less training required, however the worse performance occurs once the loss as converged. One fully connected layer is too little and three layers would take too long to train for this project time limit. So I came to the conclusion that 2 FC layers would be optimal.

In terms of conv/max pooling layers, more than two of these would result in data loss as the model would begin to lose important info about non-max values. While only one, would miss patterns in the images, and would allow too much noise through.

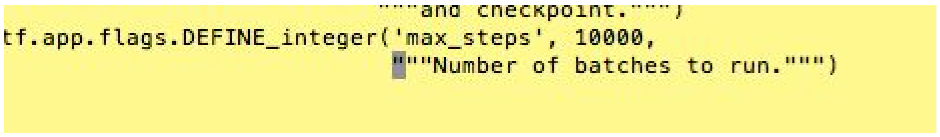
Regularization/Initial Learning Rate

In my experience(and the experience of those on stack overflow) a regularization constant on the order 10^(-3) is optimal, so a value of .004 was used.

The initial learning rate of 1 would be too high, the gradient descent would skip from local minima to local minima, while an initial learning rate of anything less than .01 would fail to converge in a reasonable number of iterations

Results

With 20000 iterations, architecture #1 performs extremely well with a final loss of 1.64 and a train/val/test accuracy of .746, .642, and .641 respectively. This model took 2 hours and 27 minutes to train on a 16 CPU cloud virtual machine with 32gb of RAM. Because I don't think that I would be able to recreate this running on CSIL (because it will probably take longer than 3 hours), I will probably end up lowering the iterations to 10000 (since that was when the loss converged). If this takes more than 3 hours it would be easy to reduce the num-iterations by modifying the code shown below on like 29 in conv_net.py, which I'm saying incase the TA/grade of this assignment finds that it takes longer than 3 hours, they can simply reduce the number of iterations until it fits in the 3 hour limit, but in its current state this *SHOULD* work in less than 3 hours.



```
                              """and checkpoint.""")
tf.app.flags.DEFINE_integer('max_steps', 10000,
                            """Number of batches to run.""")
```

Challenges

Learning to use TensorFlow itself was the greatest challenge of this project. Although TensorFlow provides a lot of helper functions, libraries and objects in order to aid user convenience, the overhead of learning how to use the TensorFlow structure was larger because of this. Also, I was limited on time (partly my own fault because I started the project late) but was not able to tune my hyper parameters to the level (in respect to the fineness of the tuning, my parameters had to be adjusted in exponential increments because I wanted to tune all the parameters and I ran out of time). Most of the challenges I encountered while working on this project were time-constraints (mostly due to my own error), however the CPU-intensive processes required to train the algorithm limited my ability to figure out the optimum architecture and hyper parameters.

Extra Credit

Different Filter Sizes of Convolution Layer	See Arch #2 and Arch #4
Different Filter Sizes of Pooling Layer	See Arch #2 and Arch #5
Different Numbers of Neurons in Fully Connected Layers	See Arch #1 and Arch #2

Possible Improvements

There are a variety of ways that I could have improved this model. The first way that comes to mind is the implementation of dropout, as well as the implementation of momentum. Another way that I could have improved this model, is if I had more time I could tune the hyperparameters to a more fine level. I also didn't have time to train and test more complex architectures with more than 2 conv/pooling layers.