**Part 2:**

Our group met to do part 1 of the assignment well before we met to do part 2. In the gap between our meetings we forgot that the tutorial for Deep MNIST was already using a network structure of 2 convolutional units (with max pools and RELUs) followed by a fully connected layer and lastly a softmax. This is a very standard architecture and without revisiting the tutorial for Deep MNIST, we coincidentally coded this layer into the tensorboard tutorial (which was original a simple MNIST). We created the network from scratch, and then went to compare it to the Deep MNIST tutorial at which point we realized our implementation used an identical architecture. As expected, it performed the same. We have included the code in the src folder under “**INSERT**” as it still performs more logging and test evaluation than the original Deep MNIST tutorial does. We then created an entirely new architecture taking some of Forsyth’s hints in the HW description.

The new architecture we ended up with is comprised of 4 smaller convolutional units. Each unit performs a convolution with a depth of 8 (vs. the original 32) and ends with an RELU and a dropout layer but no max pooling. The dropout layers are an easy and simple regularization technique to avoid overfitting. They are usually implemented by disabling neurons with some probability. This introduces some stochastic behavior in the forward pass. Obviously the neurons are all kept active when doing a prediction/testing run. In our case, the dropout layers make it difficult for the network to start up but once a successful direction is found, there is protection against overfitting. This is useful because randomness keeps overfitting from degrading the test accuracy allowing for higher final accuracies at the cost of increases training time (more iterations). After the convolutional units, the network includes a fully connected layer with an RELU followed by a dropout. Lastly a readout layer finishes up the network with a softmax.

We experimented with batch normalization, which often helps in practice… but we were unable to get it working without errors. We did implement an adaptive learning rate by splitting the training iterations into epochs, and reducing the learning rate each epoch. By decreasing the learning rate each epoch the training is likely to make big parameter jumps in the beginning where the training time has not accumulated to a reliable direction, but make smaller jumps towards the end when the direction is assumed to have aggregated towards the minimization of the cost. This network takes a little longer than the previous network to train, mostly because of all the dropouts but it achieved similar if not slightly better performance and will likely continue improving slightly given more iterations. Our implementation logs summaries of training accuracy every 100 iterations and evaluated test accuracy every 1000 iterations. Ideally we would only log the training accuracy and then provide the final evaluated test accuracy on a heldout data set as a single accuracy value. We opted to log the test accuracy at larger intervals to follow the style provided by the tensorboard tutorial, but wanted to note this in the writeup as somewhat poor style. Given the slow start and number of dropout layers, the training is slower but keeps improving. We ran the model for 50000 iterations split into 200 epochs. The final test accuracy changed slightly each time we ran it, but all runs were greater than 99.2%. Images of the training accuracy, test accuracy, overlay of test and train accuracy, and cost are shown below as screen captures from tensorboard.

The following graphs show train accuracy (upper left), test accuracy (upper right), overlay of test + train accuracy (lower left) and cost (lower right).



