TensorFlow machine learning for distracted driver detection and assistance using GPU or CPU cluster

by Steve Kommrusch

Problem

In 2015, 391,000 people were injured in motor vehicle crashes involving a distracted driver [1]. As portable electronic devices give us access to information 24 hours per day, and as our lives get more scheduled and busy, there are several forms of distracted driving that are being noticed and analyzed in order to improve driving safety. Visual distraction occurs when the driver takes their eyes off the road; manual distraction occurs when the driver takes their hands off the wheel; and cognitive distraction occurs when the driver takes their mind off driving. The CDC is concerned enough to be tracking these factors and researchers are gathering data to help analyze the situation.

On the Open Science Framework, Malcom Dcosta has posted a simulator study which analyzes 68 volunteers driving through a simulated environment while undergoing various forms of distraction [2]. The raw data being gathered by the breathing and heart rate monitors, external cameras, eye tracking data, automobile driving controls, and thermal facial cameras can exceed 5GB per minute. The total raw data for all 68 drivers is about 1.7TB. Hence, big data techniques are useful in order to fully analyze the total data gathered. By analyzing this data, I hope to explore 2 problems: using the available driver information and how the driver is interacting with the driving course, determine if they are being distracted; and then, using the distraction state along with the driving information, provide driver assist by improving the steering and brake/accelerator controls. I plan to share my results with the original study authors and hope that they may provide insights about how to make driving safer in the future.

Strategy

For tackling the challenge of detecting and assisting distracted drivers, I plan to create 2 neural network models: the first will analyze the data available during a drive to determine if the driver is experiencing distraction; the second will use the driving data for a non-distracted driver to train recommended controls. In each case, I will have a training and inference use model to solve. For training these 2 networks I compared 3 Tensorflow [3] approaches: asynchronous distributed training with CPUs and GPUs, and single-machine GPU training. All 3 cases use an 'Early Fusion' convolution neural network similar to Karpathy et al [4] with the first convolutional layer including 3 or 4 frames of data to evaluate temporal information. Then, using distributed machine learning techniques similar to those discussed by Chu et al [5], Tensorflow will aggregate the updates to the weight matrices during training. To avoid compute resource contention with other students, I used 6 machines in CSB325 for the distributed training as well as for the single GPU system training. I used the single GPU configuration to explore hyperparameters of the network (depth, learning rate, etc). In order to evaluate the hyperparameter choices and final results, I used 80% of my data for training and 20% for testing.

For input, I use the simulated scene the driver sees and driver response data. One of my first tasks in the project was to merge the video data with the numeric driver data values. Because the CNN examples I found expect a square input image and since the actual simulator screen is wider than it is tall, I used the upper part of the input image to visually provide the current and past driver control data as shown in figure 1. This preprocessing code was written in Tensorflow and generates PNG files for use as inputs to the Tensorflow CNN.

Figure 1: A triplet of frames selected to be 0.4 seconds apart from one of the drives. The top green bar shows the driver accelerating through a green light. The yellow bar is indicating a slight steering correction to the left. Color squares on the edges of the image have color intensity related to the control variables. These are the 280x280 pixel images fed into the neural network. (The green dot in the image is an eye tracker indicator).

Figure 2 shows the parameter update models available for distributed learning with TensorFlow [3]. In the synchronous mode, all weight gradients from training for an epoch are reduced into a parameter set which is updated for the next training epoch. In asynchronous mode, devices can receive parameter data even while the data from the previous epoch is still being incrementally included into the model.

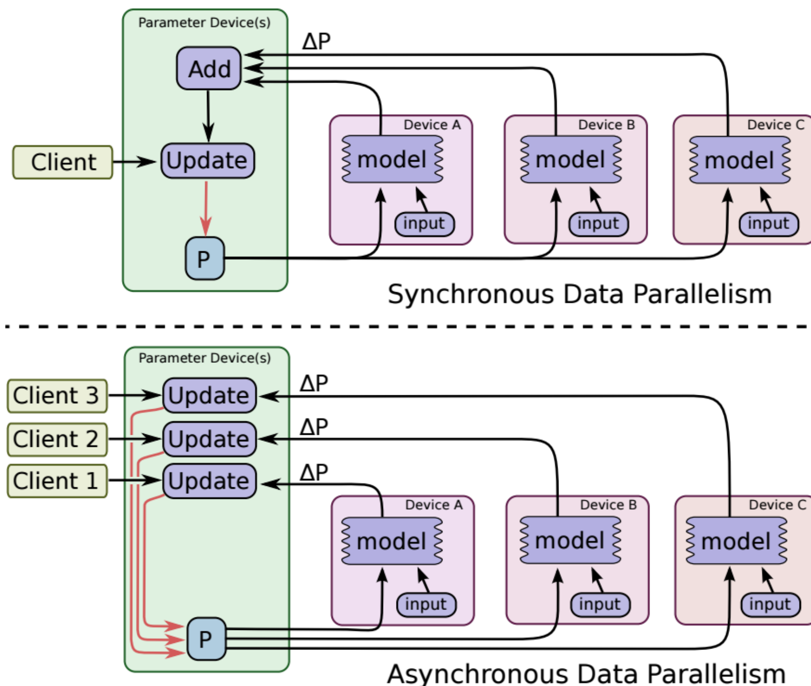

Figure 7: Synchronous and asynchronous data parallel training

Figure 2 taken from Tensorflow whitepaper[3]

I discuss hyperparameter testing in the next section, and here is the neural network model which I settled on for training to recognize distraction state:

Layer	Description
0	Input will be: 280x280 3-color with 3 frames each 2/5 second apart (similar to Karpathy et al [4]) In the upper section of the image, there are pixel values set based on the driver action data (steering, braking, acceleration).
1	64 11x11x3x3 filters (3 colors, 3 time samples) with ReLU activation function
2	2x2 max pooling layer
3	32 5x5 filters with ReLU activation functions with ReLU activation function
4	2x2 max pooling layer
5	Dense layer (fully connected) with 64 neurons and ReLU activation function
6	Dense layer with 32 neurons and ReLU activation function
7	Dropout layer with 90% chance to keep connection
8	Dense layer with 1 neuron (distraction true/false) and either no activation function (linear) OR sigmoid

The distracted driving simulations include a file identifying when during the drive the distraction was applied (typically one 2-minute distraction starts 90 seconds in, and another starts 6 minutes in). That information is used for setting the target distraction state to learn.

The network for predicting proper driver response is very similar to the distracted stated detector, except that it has 3 outputs (brake, accelerator, and steering) and is shown in figure 3.

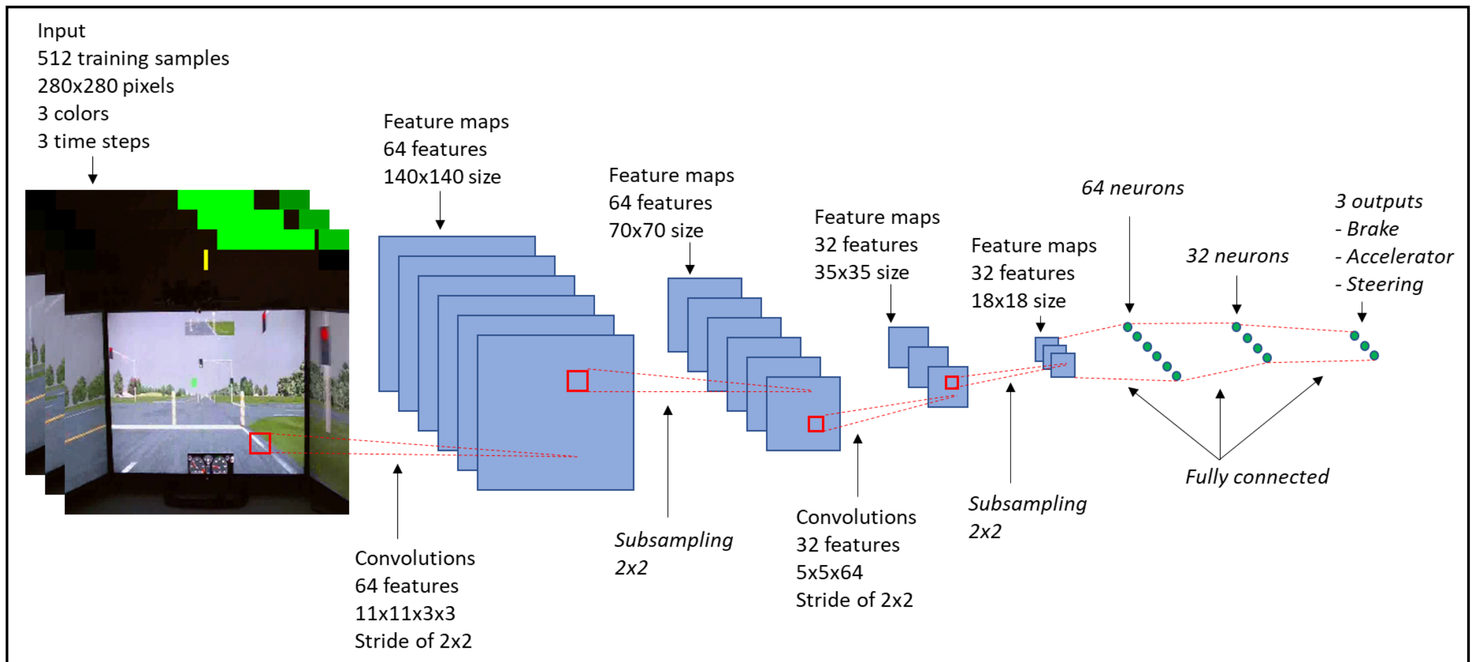

Figure 3: Convolutional neural network used for driver response prediction

The driver response prediction model is trained using non-distracted driver data and is conceptually learning to drive the car on the simulated course. The use model for these networks would be to detect when the driver is distracted and in that case only apply an average between the driver's control settings and the network model's to the car.

Dataset description

The dataset which I use has been published on the Open Science Framework web page as a public project by Malcolm Dcosta [2]. The entire dataset consists of 1.54TB of thermal video data, and 57.5GB of compressed normal videos and sensor data. The thermal video data is in S-interface format, and the other videos are in avi format. All the sensor data are CSV and XLSX formatted.

The data was collected during a controlled experiment where the subject drove in a driving simulator for about 12 minutes. 68 subjects drove the same road under different types of distractions, and multiple response variables were measured. Cognitive distraction involved answering math and analytic questions during the drive. Emotional distraction involved answering emotional questions during the drive. Sensorimotor distraction involved texting during the drive. Each distraction type has a full 12 minute simulated drive while the subject experienced the given distraction for 2 two-minute intervals. The full dataset includes different kinds of biosignals/videos, and input to the vehicle. Not all of the data is used for this project, but because I uncompress and sample the AVI files there is still be significant information to process. Below is a brief description of each data group from [2] and whether this project uses the data.

1. Biographic Data - General information about the subject such as genter, and age. (not used)
2. Psychometric Data - NASA TLX scores to measure perceived Mental Demand, Physical Demand, Temporal Demand, Performance, Effort, and Frustration. (not used)
3. Trend Psychometric Data - The (State-) Trait Anxiety Inventory (STAI) score and the Type A/B Personality score. (not used)
4. Breathing Rate Signal (not used)
5. Heart Rate Signal (not used)
6. Palm Electrodermal Activity Signal (not used)
7. Perinasal Electrodermal Activity Signal (not used)
8. Performance Response Variables - Speed, Acceleration, Brake Force, Steering. (XLS format, used)
9. Perinasal Region of Interest Thermal Video (not used)
10. Facial Video (not used)
11. Operational Theater Video - The first person view of the subject with eye tracking. (AVI format, used)
12. FACS Signals - Subject's emotional state using Facial Action Coding System. (XLS format, reviewed)
13. Stimulus file - time and type of distraction stimulus applied during the drive (XLS format, used)

Table 1 shows example information about the driver control state. I use only the speed, acceleration, braking, and steering information as inputs by encoding this information into the video image data as discussed previously.

Simulator Outputs

Frame	Time	Speed	Acceleration	Braking	Steering	Lane Offset	Lane Position
4664	111.022	40.75272	0	1	0.105842	0.515395	4.959605
4665	111.039	40.74243	5.495911	1	0.105075	0.515395	4.959605
4666	111.072	40.73562	5.627747	1	0.105075	0.540551	4.934449
4667	111.089	40.73369	7.036743	1	0.103541	0.566335	4.908665

Table 1: Sample data from performance response spreadsheet

Table 2 shows example Facial Action Coding System (FACS) data. This data is not used as model input, but I reviewed the data in relation to the distraction stimulus information and confirmed that the distraction stimulus affected these factors.

Facial Expressions									
Frame#	Time	Anger	Contempt	Disgust	Fear	Joy	Sad	Surprise	Neutral
3	0.12	0.12204	0.330137	0.116368	0.002267	0.078749	0.058911	0.000007	0.291523
4	0.16	0.091568	0.27484	0.187858	0.00361	0.062536	0.074601	0.000009	0.304978
5	0.2	0.086424	0.391132	0.092647	0.003435	0.051811	0.078862	0.000006	0.295683

Table 2: Sample data from facial action coding system (FACS) file

After processing the AVI and XLSX source files, my input PNG files for 5 drivers under various distraction modes total 150,000 files with a total file size of 8GB. As each file decompresses to a 280x280x3 floating point tensor, these 150,000 files represent 140GB of source input data for the CNNs I utilize. The PNG files were stored on a shared Linux drive such that they were available to the distributed Tensorflow through standard Linux file access routines.

Software functionality

This section provides an overview of the code used for this project. For optimal understanding of the code, the code can be viewed simultaneously with reading this section.

Linux commands

- ffmpeg: used to create about 7,500 PNG image files from a single AVI input file.
- ssconvert: used to quickly convert XLSX format files to CSV files for Tensorflow input.
- start.sh file: set up environment variables and activate tensorflow environment for GPU training.
- stcpu.sh file: set up environment variables and activate tensorflow environment for CPU training.

imgprocess[123].py

- Processes raw video PNG files into CNN input PNG files. Uses python CSV file processing and tensorflow image methods to create 280x280 pixel images with driver control behaviors added to the image. Aligns timestamp data in CSV files to correct image from AVI-to-PNG process.

tfgpu.py

- The single GPU convolutional neural network model for driver control.
- The file includes these key sections:
  - Import driver control CSV file into 'control[26000][8]' numpy array.
  - Import 640 PNG files from non-distracted video of the 3rd driving subject into the network input 'video[640][280][280][9]' numpy array. The size 9 dimension includes the 3 colors times 3 timestamps which are 0.4 seconds apart.
  - Store the target driver responses in 'responses[640][3]' for training the desired brake, accelerator, and steering output of the network (targets are 0.4 seconds into the future from the most recent of the 3 input time samples)
  - Store the 'null' driver response in 'nullresp[640][3]'. This is the same value embedded into the visual image of the most recent of the 3 image samples. Used for CNN evaluation later.
  - Set up network in tensorflow. Define functions for weights, bias, convolution, and max pooling. Set up training placeholder variables 'x' and 'y_', as well as evaluation 'y_null' variable. Build network with 2 convolution layers and 3 dense layers as described previously. Use mean squared error as function to minimize for the 3 outputs. Include a dropout layer in the final layer to help avoid overfitting [7].
  - Train the network for 10,001 epochs using 128 sample batches randomly selected from the input image triplets. Train on 512 of the 640 original triplets. Every 50 epochs, output the training mean squared error and the test error for the 128 of 640 triplets that are not in the training set.

- Output the mean squared error of using the ‘nullresp’ values to predict the correct targets 0.4 seconds in the future (this is the error resulting from holding the brake, accelerator, and steering static for 0.4 seconds).
- Set up triplet samples and control targets for emotional, cognitive, and sensorimotor distraction periods. Use trained network to predict desired car controls and compare mean squared error with recorded controls from distracted driver (gives sense of driver deviation from normal driver behavior).

distgpu.py

- The single GPU convolutional neural network model for distraction prediction
- The file differs from tfgpu.py in these areas:
  - Read distraction stimulus file to set up ‘control[5]’ array with transition times for distraction.
  - Build ‘distracted[64,1]’ numpy array with distraction state for the 640 image triplets.
  - Build network with only 1 output – the distracted state.
  - Using trained network, which is trained to output 1.0 for distraction and 0.0 for no distraction, sweep a detection threshold from 0.01 to 0.99 in 99 steps and print out true positive rate and true negative rate for use in evaluation of the result.

tfmcpu.py

- The multiple compute unit convolutional neural network model for driver response prediction. Changing the ‘CUDA_VISABLE_DEVICES’ Linux environment variable before running code allows to switch from training with multiple CPUS or multiple GPUS.
- The file differs from tfgpu.py in these areas:
  - Set up asynchronous distributed tensorflow parameter server and workers using tf.train.ClusterSpec and associated methods. Parameter servers handle the state of the weight and bias variables. The workers use the weights and biases to perform forward propagation, error computations, and back propagation to compute the weight and bias updates.
  - Setup ‘control’ and ‘video’ numpy arrays to have 640 frames of data from 1 driver per worker (5 driver’s data is trained for this model with 5 workers).

tfsync.py

- The synchronous distributed training multiple compute unit convolutional neural network model for driver response prediction.
- The file differs from tfmcpu.py in this area:
  - Adds use of tf.train.SyncReplicasOptimizer and associated methods to synchronize parameter updates for each training step (tfmcpu.py allows workers to run asynchronously, which can result in workers using slightly stale parameters during batch processing).

Testing for correctness

During code development, I checked for correctness with directed testing at various steps and used judgment on the final result behaviors to guide model updates (this qualitative testing is discussed further in the results section).

For the ffmpeg, ssconvert, and imgprocess commands, I carefully viewed the source AVI file and source XLS driver behavior files. In some videos the driver had to stop at a red light and wait for green; this gave me a good way to insure my driver control data (brake, steer, and accelerator) were being aligned correctly from the XLS file with timestamps into the files from the AVI-to-PNG step.

For the distracted driving data, I viewed the distraction stimulus XLS-to-CSV files while looking at the FACS XLS file also. The FACS file showed recognizable agitated states occurring during the application of

distraction. The agitated state tended to elevate about 10 seconds after the distraction is started (perhaps the first question for the 'math question' distraction had to be asked before the actual effect occurred). Also, agitated state persisted for about 10 seconds after the distraction stimulus ends (perhaps it takes that time for the driver to 'calm down' after being distracted). I accounted for those 10 second delays when setting up the expected distracted state training target and test values in the distgpu.py file.

For the single GPU files tfgpu.py and distgpu.py, I relied on the training, test, and null behavior mean squared error (MSE) results to convince myself that the network was training and functioning as desired. The large performance difference between CPU and GPU modes, along with output messages from tensorflow, confirmed that the GPU was being used as desired by the GPU training mode.

For the multiple CPU and multiple GPU testing of tfmcpu.py, I used 'top' to confirm that CPUs were highly active for the CPU tests; 'top' also confirmed when my input data sets were too large and caused swapping (discussed in results section). I used the MSE rates to confirm that the 5 worker GPU machines were indeed improving the results for the shared model compared to the results a single GPU could achieve.

The synchronous distributed tensorflow code in tfsync.py never worked well for me. I spent some time reading and attempting to replicate examples from the internet for this mode, but in the end I could not get all 5 worker nodes to synchronize their training correctly. The output logs never showed more than 1 worker actually executing training steps before the execution shut down. It was clear from the paucity of examples as well as a couple bugs against Tensorflow that most distributed Tensorflow programmers prefer the asynchronous mode which functions correctly for this project using tfmcpu.py. I would expect that relative to asynchronous training, synchronous training would have reduced MSE a bit faster per training step, but would not reduce MSE as quickly per wall clock minute of distributed execution. This theory is supported by [6], which discusses the problem of stragglers impacting the execution performance of synchronous distributed training.

Results and evaluation

While I had not planned on extensive hyperparameter testing, I did do some such testing in order to settle on a fairly accurate network. For these tests, the network trains on a randomly ordered 80% of the total input triplets and the remaining 20% of the data is used as a test set. Table 3 summarizes hyperparameter experiment results. For these runs, 'Trial test MSE' is the mean squared error on the test set used to compare that parameter's options.

Parameter	Value tested	Trial test MSE	Comments
Time steps of input	3	0.041	3 time steps 0.4s apart reduced MSE relative to 4 time steps. 3 time steps is sufficient to see accelerating objects in the scene.
	4	0.068	
Fully connected layers	2	0.140	Going from 64->3 to 64->32->3 reduced MSE.
	3	0.052	
Learning rate with 1,000 epochs	0.001	0.044	With 1,000 epochs for training, weight learning rate 0.0002 looks best.
	0.0002	0.043	
	0.0001	0.052	
Learning rate w/10,000 epochs	0.0002	0.045	With 10,000 epochs for training, weight learning rate 0.0001 looks best.
	0.0001	0.042	
Keep probability of dropout layer	0.99	0.043	This parameter avoids overfitting by randomly dropping some connections in final layer during training. Example networks on internet often use 0.5 dropout, but 0.9 seemed best for this network.
	0.9	0.035	
	0.8	0.058	
Neuron counts	32->64	0.035	More neurons in 2 of the layers (last convolution and 1st fully connected) increased the test MSE
	64->128	0.089	

Table 3: CNN hyperparameter testing for driver response prediction network

After hyperparameter testing, I settled on the network shown previously in figure 3 for the driver response training. Figure 4 shows the results of training this network to predict the actual driver response 0.4 seconds into the future. Leaving aside the details of how the outputs are scaled for proper training, one can see the reduction in mean squared error that occurs over the 10,000 iterations. The 'Null Response' line shows the mean squared error of comparing the driver response from time T with the desired target response 0.4 seconds into the future.

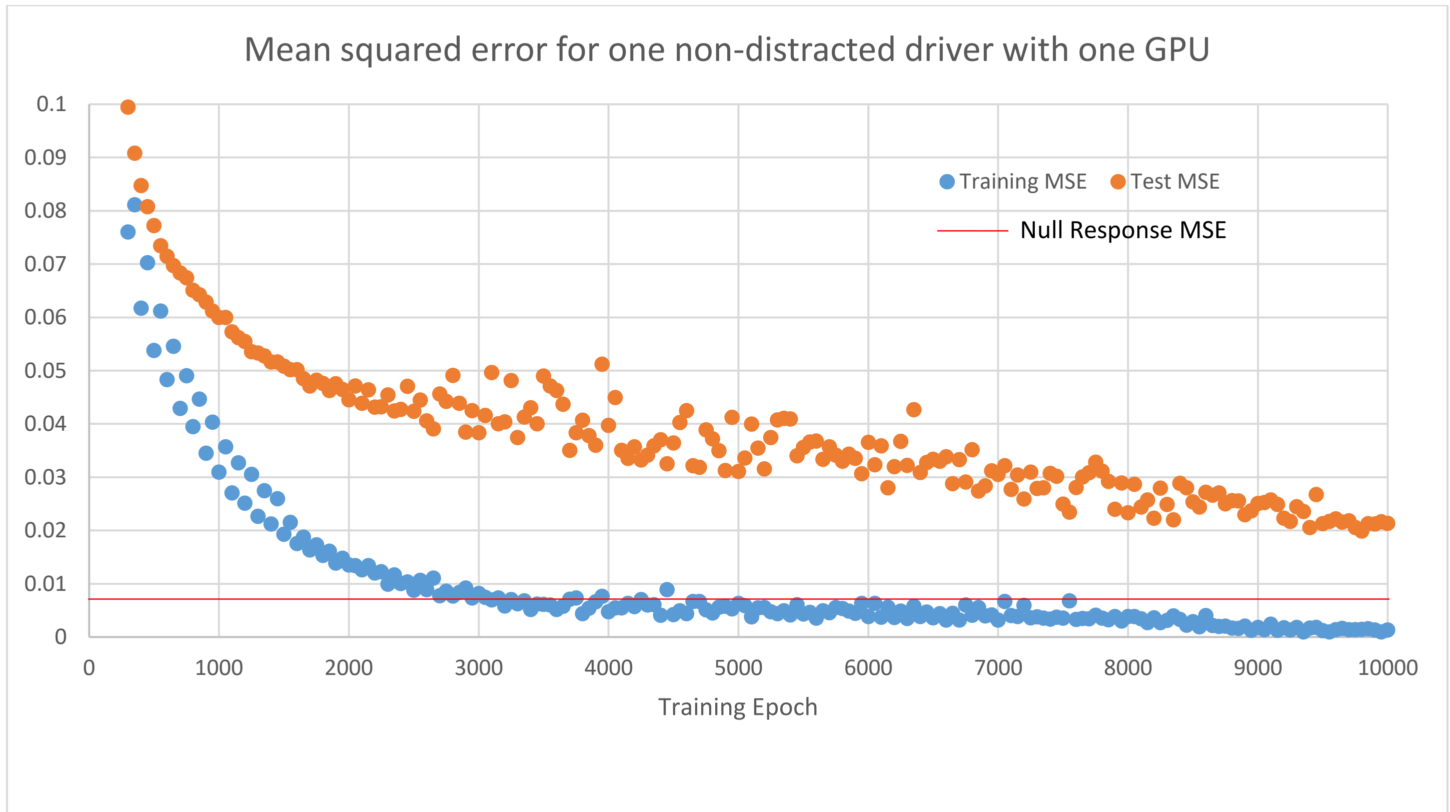

Figure 4: Training, test, and null response MSE for driver response prediction

The network trained in figure 4 was trained on the non-distracted simulated driver for one of the drivers (driver #3 in this case). Table 4 compares the prediction of the trained network with the driver response for the non-distracted drive against the 3 distraction types simulated with the same driver. As can be seen, the driver when texting has the largest deviation from the trained network, which can be interpreted as showing that of the distraction types analyzed, texting has the most impact on driving performance.

Driving condition	MSE of actual driver versus network recommendation
Normal driver test dataset	0.021
Emotional Distraction (emotional questions)	0.036
Cognitive Distraction (math and analytic questions)	0.038
Sensorimotor distraction (texting)	0.066

Table 4: Distraction type variation relative to trained network.

Figure 5 shows the results of using 5 machines (each with a GPU) to train for 50,000 epochs of 128 sample batches from 5 different drivers. The added training epochs and variation in driver behaviors combine to allow the network to nearly reach the 'Null Response' line for the MSE of the testing data.

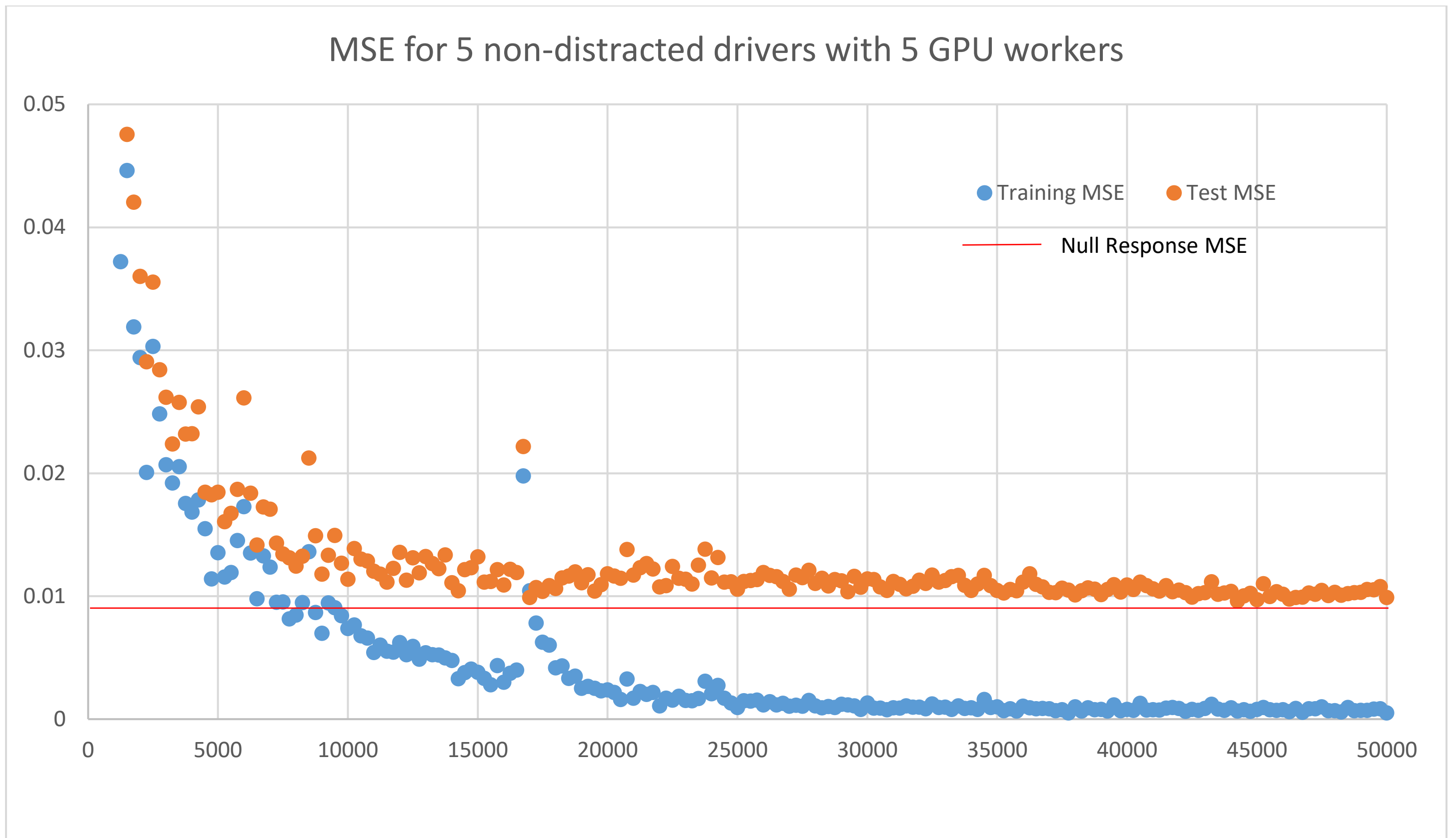

Figure 5: Distributed Tensorflow training results for driver response

For the single-output distracted driver network, figure 6 shows the training results. For this network, I tested with both a linear output and the same network with a sigmoid function as an output. The sigmoid function worked significantly better both during training for MSE reduction and for the receiver operating characteristics shown in figure 7. Ultimately, the sigmoid output AUROC (area under the receiver operating characteristics curve) was 0.997. Given that the target result for detection is 0 or 1, I expect the sigmoid function allows the network to clamp certain 'obvious' conditions to 0 or all 1, whereas a linear prediction would result in -0.5 or 1.5 output values being seen as errors instead of a clear indication of distraction state. However, the fact that the sigmoid function did so well is suspicious to me; since the simulation drive is rather deterministic, the network could have learned, for example, that 'any time there is a farm house on the right, the person is distracted'. A better distraction classifier could probably be trained with less predictable driving paths, which were not available from the dataset.

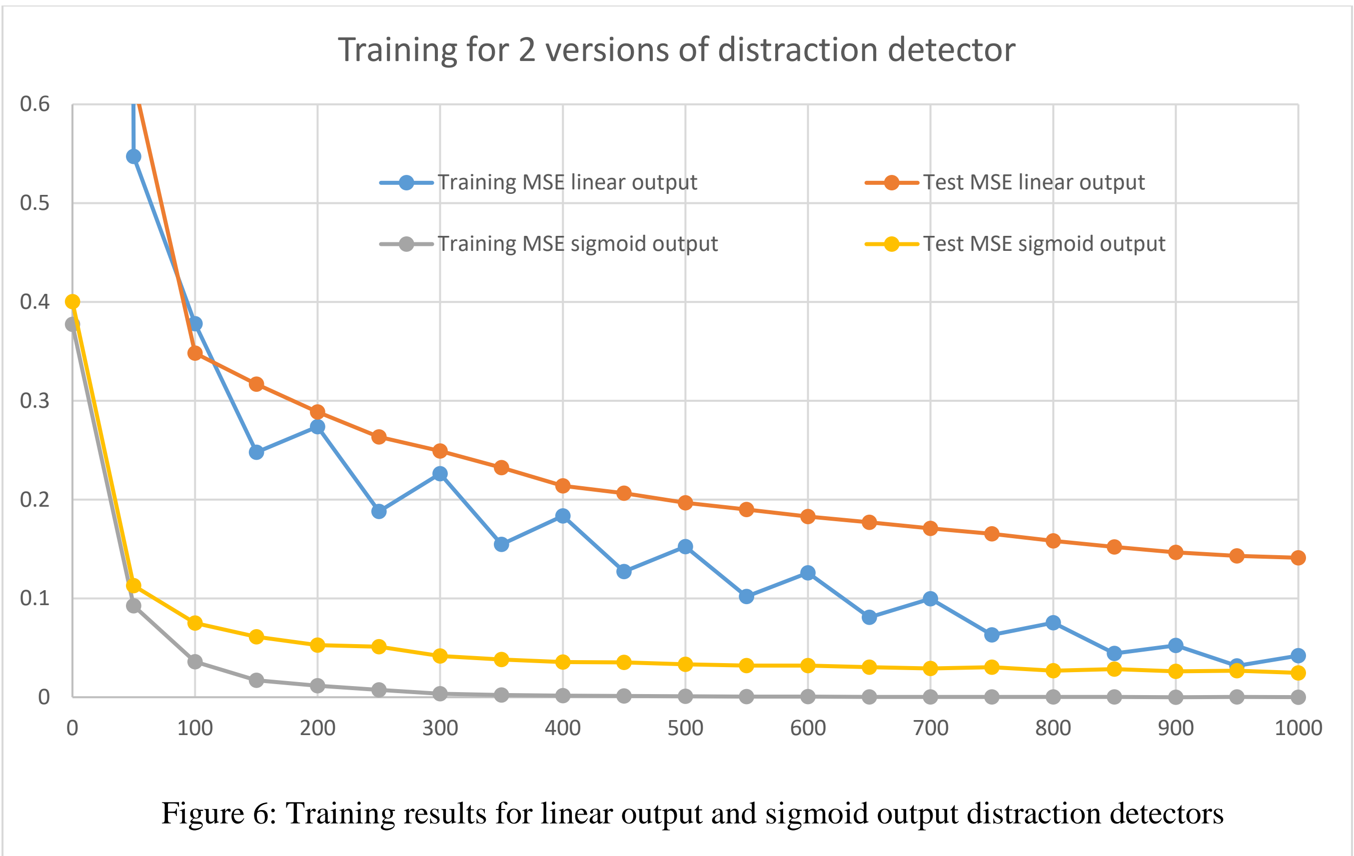

Figure 6: Training results for linear output and sigmoid output distraction detectors

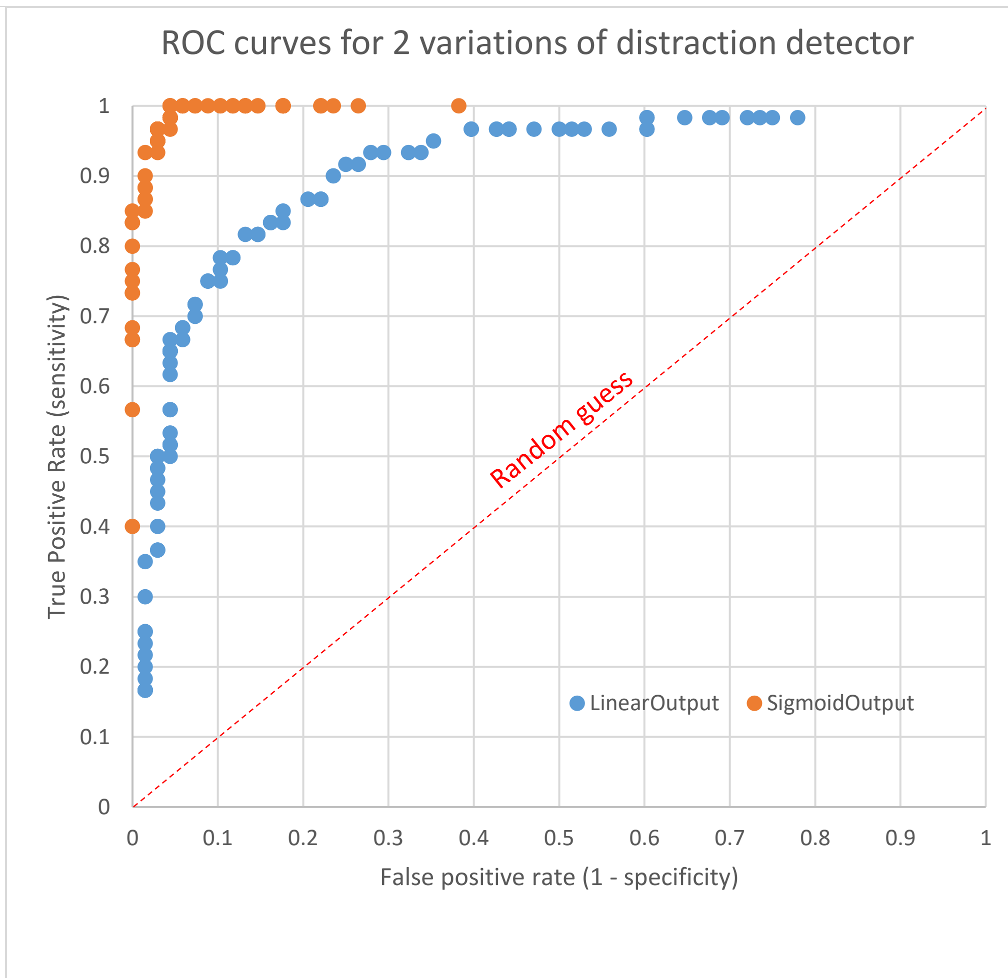

Figure 7: Receiver Output Characteristic for test dataset. The AUC for the sigmoid output is 0.997.

My final results for this project relate to the performance of the CPU, GPU, and distributed Tensorflow programs. All of my performance testing was done with CSB325 systems which have 6 Intel Xeon E5-1650 CPUs (12 threads) at 3.6GHz, an Nvidia GeForce GTX 1060 6GB GPU (1280 CUDA cores at 1.86GHz), and 16GB of RAM. Table 5 shows the performance of the driver response prediction network from figure 3 with various processor configurations. The 5 GPU system is able to achieve 2.9X speedup over the 1 GPU system. The system I built had 1 parameter server and 5 workers; the workers proceed asynchronously to each other as they send updates for the weights and bias variables to the parameter server and then request the newest parameter values for the next batch of 128 samples to be processed. It seems likely that the parameter passing overhead impacted the ability of the 5 GPU system to reach the ideal 5X speedup. The 5 GPU system is 8.4X faster than the 5 CPU system (which runs 12*5 = 60 threads).

Processor config	128-sample epochs	Time to execute	Epochs/minute
1 GPU machine	10,000	157 minutes	63.7
5 GPU machines	50,000	274 minutes	182.5
5 CPU machines	500	23 minutes	21.7

Table 5: Driver response prediction network performance under varying conditions

For both CPU and GPU systems, I found that I could not process more than 640 samples per machine (a 5 machine cluster could process 3200 samples). Table 6 shows that 1280 samples results in horrible CPU utilization as the kswapd process becomes very active. Also seen in Table 6 is the clear indication of CPU utilization for the CPU configuration during training.

Processor type	Size of dataset per machine (80% for training, 20% for test)	CPU utilization	Virtual Memory
GPU	640	52.5%	43.5G
GPU	1280	0.6% (kswapd 2.8%)	64.6G
CPU	640	1041%	47.8G

Table 6: Driver response prediction network CPU and memory usage

Tables 5 and 6 together demonstrate the 2 main advantages of distributed machine learning. The training can process more samples per second thanks to execution parallelism with multiple workers, and the network can train on a larger dataset by sharing the dataset memory requirements over multiple workers.

Bibliography

[1] National Center for Statistics and Analysis. *Distracted Driving, 2015*, in *Traffic Safety Research Notes. DOT HS 812 381*. March 2017, National Highway Traffic Safety Administration: Washington, D.C.

[2] Dcosta, Malcolm. *SIMULATOR STUDY I: A Multimodal Dataset for Various Forms of Distracted Driving.* Open Science Framework, 15 July 2017. Web. https://osf.io/c42cn/

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.* Google Research, 2015. Web. http://download.tensorflow.org/paper/whitepaper2015.pdf

[4] Andrej Karpathy and George Toderici and Sanketh Shetty and Thomas Leung and Rahul Sukthankar and Li Fei-Fei. *Large-scale Video Classification with Convolutional Neural Networks.* CVPR 2014.

[5] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, and Andrew Y. Ng, *Map-Reduce for Machine Learning on Multicore*, NIPS 2006: 281-288

[6] Xinghao Pan*, Jianmin Chen*, Rajat Monga, Samy Bengio, Rafal Jozefowicz. *Revisiting distributed synchronous SGD. arXiv e-print (arXiv:1604.00981)* February 2017.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research, 2017. Pages 1929-1958.

"TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc."