APPLYING NEURAL NETWORKS FOR TIRE PRESSURE MONITORING SYSTEMS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Alex Kost

March 2018

COMMITTEE MEMBERSHIP

TITLE:	Applying Neural Networks for Tire Pressure Monitoring Systems
AUTHOR:	Alex Kost
DATE SUBMITTED:	March 2018

COMMITTEE CHAIR:	Mohammad Noori, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Xiao-Hua Yu, Ph.D. Professor of Electrical Engineering
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	William Murray, Ph.D. Professor of Mechanical Engineering

ABSTRACT

Applying Neural Networks for Tire Pressure Monitoring Systems

Alex Kost

A proof-of-concept indirect tire-pressure monitoring system is developed using neural networks to identify the tire pressure of a vehicle tire. A quarter-car model was developed with Matlab and Simulink to generate simulated accelerometer output data. Simulation data are used to train and evaluate a recurrent neural network with long short-term memory blocks (RNN-LSTM) and a convolutional neural network (CNN) developed in Python with Tensorflow. Bayesian Optimization via SigOpt was used to optimize training and model parameters. The predictive accuracy and training speed of the two models with various parameters are compared. Finally, future work and improvements are discussed.

ACKNOWLEDGMENTS

Although I technically started work on this thesis in Fall 2015, I did not begin to make meaningful progress until Winter 2016. I procrastinated and pushed this off for so long that many friends began to ask me *if*—not *when*—I would graduate.

So to my thesis adviser Dr. Mohammad Noori, thank you for your patience, support, and kindness throughout this work. I could not have asked for a more supportive and understanding adviser.

To my friend Han Tran, who encouraged me to work with Professor Noori on a very non-mechanical thesis topic and pushed me to perform to the best of my capabilities. You are the reason for much of my success.

To my fraternity; SPD TID.

And finally to my family, who never failed to remind me that I must graduate by June 2017 "or else." I didn't and everything turned out ok.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Background

It is difficult to understate how important properly pressurized tires are to the performance and safety of a vehicle and its operator, respectively. The National Highway Traffic Safety Administration (NHTSA) estimates that 11,000 tire-related crashes occur annually in the US, with 200 people estimated to be killed in these crashes [28]. Furthermore, under-inflated tires contribute to the following performance issues when driving [37]:

1. Poor fuel economy, wasting an estimated 3.5 million gallons daily and costing drivers as much as 11 cents per gallon in the US.
2. Longer stopping distances and sluggish/ineffective handling, resulting in more dangerous driving conditions.
3. Faster tire wear, reducing the average life of a tire by 4,700 miles.

Tire-pressure monitoring systems (TPMS) became federally mandated in 2000 by the Transportation Recall Enhancement, Accountability, and Documentation Act, where legislators ruled to "require a warning system in new motor vehicles to indicate to the operator when a tire is significantly under inflated" [38]. More specifically, all motor vehicles must have a system that is capable of detecting when one or more of the vehicle's tires, up to all four tires, is 25% or more below the manufacturer's recommended inflation pressure or a minimum activation pressure specified in the standard, whichever is higher [25]. Nonetheless, a study performed in April 2009 showed that 45% of TPMS-enabled vehicles still have under-inflated tires [26].

Therefore, for obvious moral and legal reasons, it is imperative that drivers know that their tires are inflated properly. It is in the individual's and society's best interests to improve safety, performance, and savings while on the road.

1.2 Purpose

The most commonly used TPMS in vehicles today is a simple pressure sensor mounted within the tire to directly measure the pressure of the air within the tire. When the integrated battery dies on these sensors, the sensors must be replaced manually. Time, money, and labor are spent to replace this simple sensor. It would be advantageous if the TPMS architecture was created such that maintenance and repair were not needed.

As advancements in machine learning and deep learning techniques continue, it is no longer a question of *how* or *why* to apply these techniques, but *where* to apply them. In this work, a proof-of-concept TPMS architecture is suggested that uses accelerometer data and deep learning algorithms to determine whether the tires on a vehicle are under, over, or nominally inflated. This work is comprises four chapters:

1. **Chapter 1** introduces the legal and moral motivations behind TPMSs in today's vehicles. This chapter also outlines the content of this work.
2. **Chapter 2** serves as a literature review for this work. Three specific fields of study are defined: a mechanical review of automotive suspension systems and tires, current TPMS frameworks and sensing capability, and current technologies and research in artificial neural networks.
3. **Chapter 3** presents the work done to create a proof-of-concept classifier using simulated data. The simulated model and its limitations are discussed, as well as the architecture and modifications done on the artificial neural network.
4. **Chapter 4** compares the final performance of the implemented classifiers and concludes this work with a discussion on the meaning and limitations of the results. Suggestions for future work are also made.

Chapter 2

THEORY

2.1 Suspension and Tires

Vehicle suspensions and tires are designed to optimize the traction, ride comfort, handling, and fuel consumption of the vehicle. The suspension links the *wheels*—tires mated to rims—to the vehicle chassis and allows relative motion, while the tire transfers energy between the vehicle and the road to allow the vehicle to move [16]. Together, the suspension and tires are the defining aspects of a vehicle's combined stiffness and damping coefficients.

A simplified representation of a vehicle suspension system is used in this work. Known as a *Quarter Car Model*, the representation has only one degree-of-freedom and can only move vertically. The vehicle is rigid; only vibrations transferred from the ground to the tires, axles, and suspension systems are considered. This representation also does not consider any forces or reactions due to the geometry of the vehicle; it is only looking at a single wheel on this "vehicle." The representation is presented in Figure 2.1 [16].

The unsprung mass m_u refers to all masses that are attached to and not supported by the spring, such as the wheels, axles, or brakes. In this representation, the unsprung mass is the weight of the tire and the weight of the air of the tire. In an actual vehicle, suspension stiffness and damping values k_s and c_s are functions of suspension type, tire geometry, tire pressure, vehicle geometry, and vehicle weight. These values should be constant in vehicles without active suspension systems, so the only changing parameter in this model is the unsprung mass's stiffness k_u. Any damping in parallel with k_u is negligible with respect to c_u and is thus not included in the representation.

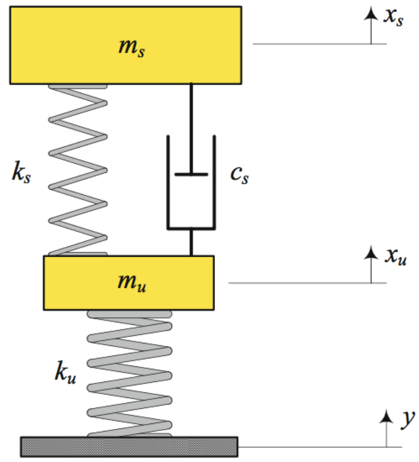


Figure 2.1: A free-body diagram of the quarter-car model. Taken from *Jazar et al.* [16].

2.2 TPMS Architectures

The NHTSA provides vehicle manufacturers three ways to comply with the law: direct, indirect, and hybrid TPMS [27]. Direct TPMS consists normally of pressure sensors located inside each wheel to directly measure the pressure in each tire. Indirect TPMS compares speed data collected from vehicle's anti-lock braking system wheel speed sensors to compare rotational speeds of tires against one another to determine the pressure. Direct systems are more accurate and precise, whereas indirect systems are less hardware-dependent and more robust for each vehicle. The NHTSA leaves the definition of a hybrid TPMS purposefully vague and suggests such a system would use a combination of direct and indirect methods to fulfill the regulatory requirements. Although direct TPMS dominates the method today, indirect TPMS is expected to become the dominant TPMS in the coming years.

A note should be made that not all direct and indirect TPSM are created equal: individual features differ from system to system. As shown in *Kubba and Jiang* [21], various direct TPMS systems use different power sources and sensing solutions.

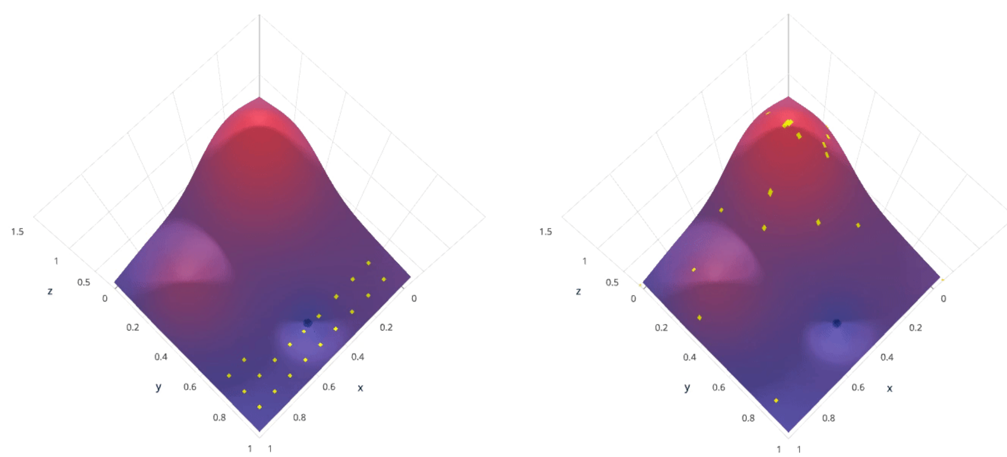
Research of indirect TPMS frameworks has grown and continues to grow because of their perceived advantages over direct TPMS as computing power increases. For example, *Persson, Gustafsson, and Drev* [31] presented in 2002 an indirect TPMS combining vibration and wheel radius analyses was able to detect pressure losses larger than 15% in one, two, three, or four tires and identify the underinflated tire within 1 minute.

2.3 Artificial Neural Networks

An *artificial neural network* (ANN) is a machine learning algorithm used to solve advanced non-linear problems such as handwriting or speech recognition. Neural networks connect computational nodes together to form a singular "network," where each computational node is performing a calculation on its input and outputting the result to all outgoing connections. The output of a node can be the input to at least one other node or to many other nodes. Outputs can be scaled and biased by *weights* and *biases* respectively; think the canonical linear function $y = mx + b$, where y is the original output, m is the weight, x is the new output, and b is the bias. Often, *activation functions* are added to the networks; these further define the output with a linear or non-linear function. As shown by *Ramachandran et. al* [34], the most commonly used activation function in deep learning projects is the *rectified linear unit* (ReLU). In summary, interconnected computational nodes perform linear and non-linear operations on inputs.

At first, all ANN models do not perform well because the weights and biases are not tuned; that is, the model is not *trained*. Neural networks can learn a hierarchical feature representation from raw data automatically [40]; that is, they "learn" or can be trained through example. In this work, we train our models via *supervised learning*—that is, with labeled training data—and compare the model's predictions to the actual labels. By repeatedly minimizing the error between prediction and truth, the model updates the trainable parameters and its accuracy improves. This updating is based on minimizing a *cost* (generally inversely proportional to accuracy) via some optimization strategy. *Gradient Descent* strategies are often implemented; in this work, the *Adaptive Moment Estimation* (Adam) strategy is applied. Adam computes adaptive learning rates for each parameter and takes advantage of the idea of *momentum* to more quickly converge on the global minima with reduced oscillation [20].

Furthermore, models hyperparameters can be *tuned* such that they can more quickly be trained and perform more optimally. *Grid search* tuning is a standard method where an exponentially large grid of possible hyperparameter combinations is systematically searched. Alternatively, *Bayesian Optimization* tuning promises a more intelligently search by learning from prior hyperparameter combinations and their results to intelligently suggest better combinations [6]. Grid searches are exponentially expensive whereas Bayesian optimization are only linearly expensive, as visualized in Figure 2.2. In this work, the software-as-a-service product *SigOpt* is applied to perform Bayesian optimization techniques for quick, intelligent tuning.



(a) (b)

Figure 2.2: (a) Grid Search vs. (b) Bayesian Optimization techniques for tuning, where each yellow dot indicates a model evaluation. Notice that grid searches could be searching along a potentially-coarse grid, whereas Bayesian optimization techniques test any possible combination within the space and intelligently suggests combinations to reach optimal solutions with fewer evaluations.

The type of input data generally defines the type of ANN to be used; in this case, the models are interpreting time series data. As defined by *Dorffner* [8], a *time series* is a sequence of vectors depending on time t such that $\vec{x}(t), t = 0, 1, 2,$ and so on. The components of $\vec{x}$ at each time t (referred to as *datapoints* in this work) are distinct from one another but are not informative enough to extrapolate meaningful information from the time series; instead, each datapoint in a time series must be analyzed in relation to the rest of the time series. We discuss two major model types for interpreting time series data below: the *recurrent neural network* (RNN) and *convolutional neural network* (CNN).

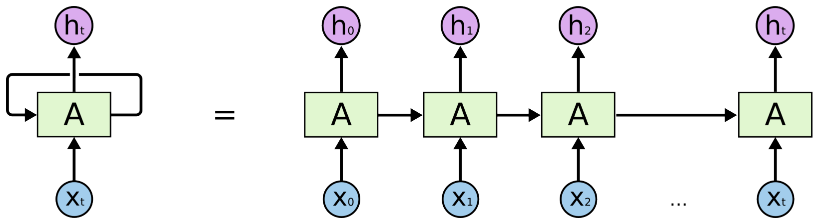


Figure 2.3: A visual representation of a single block in a recurrent neural network (RNN). Taken from [29].

Recurrent neural networks (RNNs) interpret time-series data successfully by adding feedback loops to the standard ANN network architecture [22] [9]. Some RNNs use more complex computational nodes known as *long short-term memory* (LSTM) blocks to mitigate an issue common in RNNs known as the *vanishing gradient problem* [9].

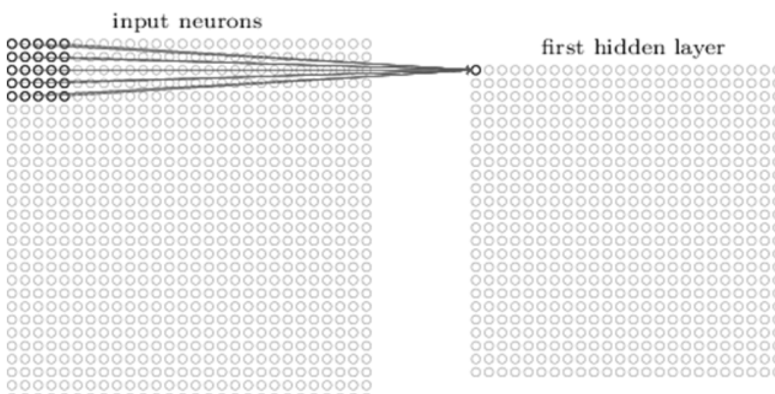


Figure 2.4: Visualization of a 5x5 filter convolving around an input volume and producing an output. Taken from [5].

Convolutional neural networks (CNNs) interpret clusters of datapoints (e.g. time-series, images, sentences, sound recordings, so on) together to preserve spatial or temporal relationships. CNNs apply *kernels* or *filters*—i.e. a weight matrices—to recognize and extract features or patterns [19].

The first few layers of a typical ANN act as *feature extractors*; that is, they are responsible for extracting meaningful information from the input data. For example, RNNs build an internal memory and CNNs use pattern matching. This meaningful information is then fed into a *classifier*. Classifiers are generally *fully-connected layers* (each node is connected to one another; see Figure 2.5) with n outputs, where n is the number of classes in the input data.

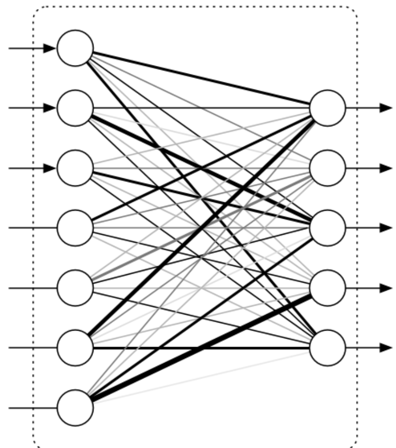


Figure 2.5: Visualization of a fully-connected layer. Taken from *Hollemans et al.* [13].

ANNs have been applied in the automotive industry for decades. In 1990, *Wiggins* presented a neural network that could identify engine faults based on the vehicle's engine controller data [39]. Neural networks were used to control the air-to-fuel ratio in fuel injection systems as shown by *Alippi et al.* in 2003 [2]. More recently, ANNs have driven advances in automated vehicle control ("self-driving") that can detect, identify, and respond to objects and pedestrians on the road in real time. While Tesla, Mercedes-Benz, and BMW were first introduce these features to consumer vehicles, the technology is becoming increasingly ubiquitous [17]. A NHTSA investigation conducted in January 2017 found crash rates Tesla crash rates have dropped by almost 40% since enabling self-driving capabilities in 2015 [12].

Applying ANNs to automobiles requires dedicated software and hardware on the vehicle. Unlike data centers, portable implementations are limited primarily by the size, energy, and computational power of the device they are operating on [33]. Size is generally not a constraint for automotive manufacturers. Energy and computational power are proportional: therefore, research has been focused on improving microprocessing architectures to minimize energy draw (hardware) or improving the efficiency of the algorithm to reduce computational load (software). With respect to hardware, in February 2016, researchers presented a convolutional neural network accelerator chip that uses 10X less power and requires 4.7x fewer DRAM accesses per pixel than a mobile GPU [4]. Similarly, with respect to software, AlphaGo, a Google project, demonstrated that integrating classification trees with neural networks significantly reduces the computational burden, making what people once thought impossible—a computer defeating a world-champion Go player in real time—possible [35]. Many more examples like these can be found.

Chapter 3

WORK

With the desire to explore alternative indirect TPMS frameworks and inspired by deep learning is seemingly infinite applications, this work explores a deep learning framework that analyzes vehicle suspension acceleration data to classify the vehicle tires as under-inflated, nominally inflated, or over-inflated. To validate this idea, work was broken into the following sections:

1. **Collecting Data.** The accuracy and capability of the ANN is largely dependent on the size of our data—ANNs tend to improve when there is more data for training. In this work, data was simulated by a quarter-car model written in Matlab and Simulink. The data serves as the training, validation, and test sets for the ANN.

2. **Creating the Algorithm.** Using the data from the prior step, an RNN-LSTM and CNN are developed in Python with Google's open-source TensorFlow API. Tuning model and training parameters are done using Bayesian Optimization via SigOpt.

3.1 Collecting Data

A Matlab model for the quarter-car representation as shown in Figure 2.1 was run at various tire pressures and step-sizes to generate simulated examples of a vehicle suspension system experiencing a step response (in an attempt to be analogous to a pothole or speed bump). The simulation solves the system of ordinary differential equations for every time step for the position, velocity, and accelerations of the sprung mass m_s and unsprung mass m_u. The simulation inputs are presented below in Table 3.1 and their accompanying derivations are presented in Appendix A.

Table 3.2: Inflation classifications, pressures, and labels.

Inflation Classification	**Pressure Range (psi)**	**Label (int)**
Under	26–30	0
Nominal	30–34	1
Over	34–38	2

Table 3.1: Simulation input variables.

Variable	**Description**	**Value**	**[Units]**
p_u	Tire pressure	Varies	[psi]
y	Step size	Varies	[m]
m_s	Sprung mass	277.25	[kg]
m_u	Unsprung mass	34.69	[kg]
k_s	Sprung stiffness	557.97	[kPa]
c_s	Sprung damping	6218.35	[Pa-sec]
k_u	Unsprung stiffness	Varies	[kPa]
g	Gravity	9.81	[m/sec^2]

The simulation was performed for $p_u = 25.5, 26, 26.5, 27, ..., 38.5$ and for $y = 0.10, 0.15, 0.2, ...$ 2.0, generating 633 total examples. Every 1.5-second-long run is composed of 1500 data points and labeled according to the inflation classifications as defined by Table 3.2. These classifications are 10% of 32 psi, well within the 25% specification as defined by the TREAD Act. The label of the simulation and the sprung's mass acceleration $\ddot{x}_s$ are saved in individual .csv files to be parsed by the algorithm. An example of the generated data is presented below in Table 3.3 (note that the first row is only shown here for clarification and is not included in the raw output).

Table 3.3: Example of simulated data: `Sim_35.5psi_0.75m.csv`**.**

label	$\ddot{x}_s, t = 0.000s$	$\ddot{x}_s, t = 0.001s$	...	$\ddot{x}_s, t = 0.420s$	$\ddot{x}_s, t = 0.421s$
2	-0.00073852	-0.00067152	...	-1.3974	-1.2822

Plots were developed of x_s vs. time as a quick sanity check. The plots make intuitive sense–higher pressure correlates with greater stiffness, which then increases the natural frequency, slows the settling speed of the mass, and reduces the maximum amplitude. The simulation is sound.

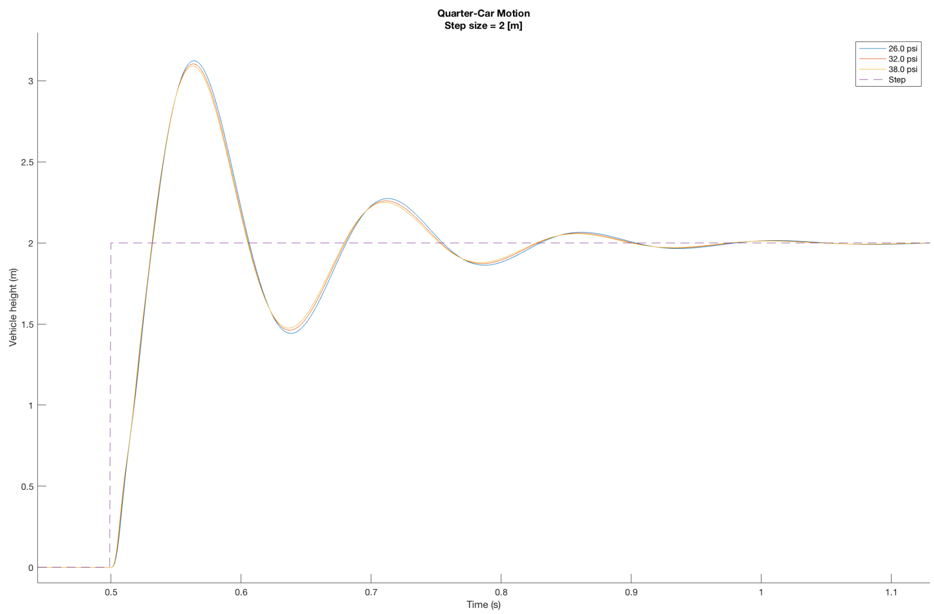


Figure 3.1: Sprung mass vs. time for $p_u = 26, 32, 38$ psi for step size $y = 2$ m.

3.2 Building the Algorithm

3.2.1 Specifications

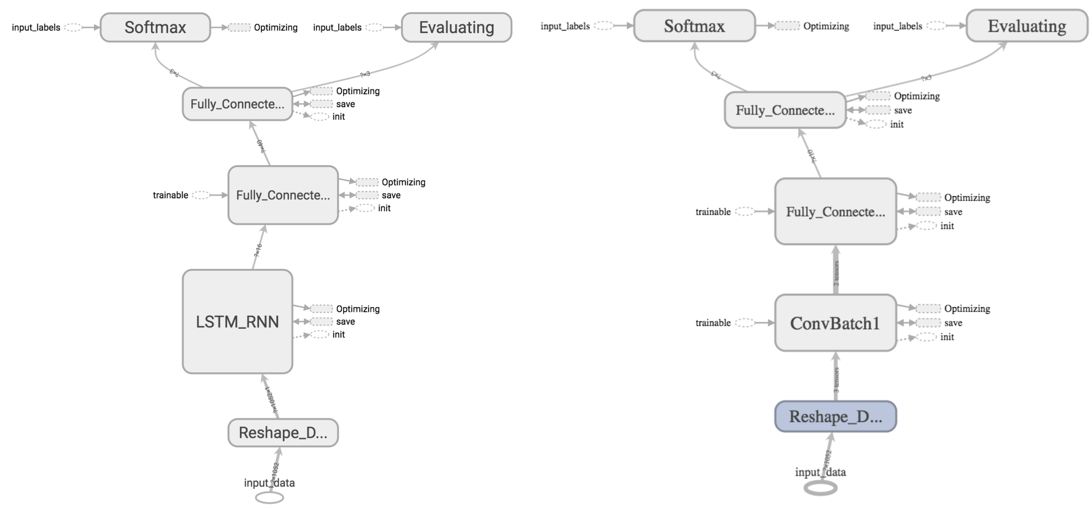
For this work, TensorFlow was used to build, train, and evaluate a RNN and a CNN. Development of each neural network followed the same specifications as listed below.

1. Import the simulated data into the Python environment.
   (a) The input data shall be shuffled randomly.
   (b) The input data shall be split into a training set (60%), validation set (20%), and test set (20%).

2. Build the model of the neural network.
   (a) The model shall be fed labeled input data and output predicted labels.
   (b) The input data should be fed in batches to minimize computational load between parameter updates. Generally, the recommended starting batch size is 32 [3]
   (c) The model shall prevent overfitting by applying dropout to the outputs of at least one fully-connected layer [32].
   (d) *Batch normalization* shall be applied after various layers to reduce the internal covariate shift within the model [15].
   (e) Model logits shall be converted to classification predictions using the softmax activation function.
3. Evaluate the predictive capabilities and training speed of the model.
   (a) The cost shall be calculated using the cross-entropy function between the input data labels and model predictions [24].
   (b) The accuracy shall be calculated by comparing the model's predicted labels to the input data labels.
   (c) The training speed shall be minimized by tuning the model hyperparameters.
4. Train the model parameters.
   (a) The training shall end after a predefined number of epochs and not be stopped early to observe any overfitting in the model.
   (b) The training method shall minimize the batch's average cross-entropy loss using *Adam Optimization* strategy [20].
   (c) The learning rate shall be static or exponentially decaying.

3.2.2 Development

The RNN-LSTM and CNN models are self-contained in `RNNModel` and `CNNModel` respectively. Both models are similar except for the feature extraction near the input layer of the model.



(a) (b)

Figure 3.2: (a) RNN-LSTM and (b) CNN model visual graphs as created by TensorBoard.

A `DataProcessor` class was written to provide methods to scan a directory for all files and perform various preprocessing operations. In this work, `DataProcessor` scans the simulated data directory; generates lists of all files found across all labels; shuffles and splits the filenames across test, validation, and training sets; and loads the feature data and label data found in each files from each set into member variables to be used for training.

The training class `TrainModel` is the entry point to train the model. Instantiating `TrainModel` builds the desired model with a provided learning rate `learning_rate` and dropout rate `dropout_rate`. Calling `train_model` trains the model for a desired number of epochs `n_epochs` using feature and label data inherited from `DataProcessor`. Every $\frac{1}{\texttt{n\_checks}}$, the model's accuracy and cost are evaluated across the entire training and validation datasets and reported to TensorBoard for visualization. The test set accuracy is evaluated before and after training.

3.2.3 Tuning

The model parameters were tuned via SigOpt to identify optimal values for various model hyperparameters. Tuning classes `GridSearchTune` and `SigOptTune` were developed to perform a grid search or connect to SigOpt to perform a Bayesian search respectively. It was estimated that a grid search over the entire model space would take over two weeks of computations per model, whereas SigOpt's more-intelligent Bayesian search strategy would take days instead. Thus, only `SigOptTune` was used in this work.

Two SigOpt experiments were run for each model to optimize the training speed and accuracy respectively. The parameters under investigation are listed below in Table 3.4.

Table 3.4: Parameters optimized via SigOpt Bayesian optimization. * denotes that the parameter is related to Adam optimization strategy

Name	Description	RNN-LSTM	CNN
`dropout_rate`	Dropout rate	X	X
`learning_rate`*	Learning rate	X	X
`beta1`*	1st moment estimates exponential decay rate	X	X
`beta2`*	2nd moment estimates exponential decay rate	X	X
`epsilon`*	Numerical stability constant	X	X
`num_filt_1`	Number of filters in convolutional layer		X
`kernel_size`	Kernel size in convolutional layer		X
`num_fc_1`	Number of neurons in first fully-connected layer	X	X
`n_layers`	Number of hidden layers in model	X	
`n_hidden`	Number of features per hidden layer in LSTM	X	

All source code is available in Appendix B.

Chapter 4

RESULTS AND CONCLUSIONS

4.1 Initial Results

Tuning the Adam-specific hyperparameters gave insight in a recurring issue with the LSTM-RNN: The model would not improve in performance after 200 steps (40 epochs with `batch_size` = 128). Figure 4.1 shows multiple training curves with various values for `learning_rate`, `beta_1`, `beta_2`, and `epsilon`. where the cross-validation accuracy would remain at 33.3%, or the same accuracy as randomly guessing.

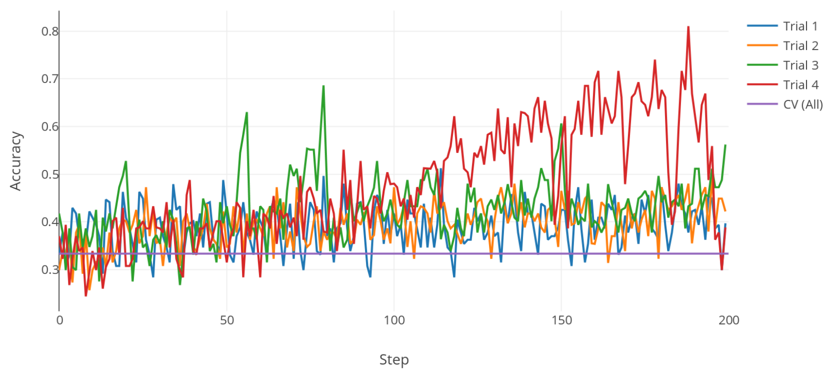


Figure 4.1: RNN-LSTM: Training classification accuracy for various Adam optimization strategy optimization parameters `learning_rate`, `beta_1`, `beta_2`, and `epsilon`.

These results can be from the RNN-LSTM's inability to identify any meaningful features after 40 epochs of the 633 training examples. The same results were seen when the model was trained for 200 epochs: The RNN-LSTM underfit the simulated data every time. Therefore, all model hyperparameters were increased. The resulting models successfully fit the input data and achieved significantly better accuracy when classifying the test set data. Further hyperparameter tuning showed that increasing the number of layers to be greater than 1 results in the model fitting the data appropriately. After 100 observations, Sigopt reported the RNN achieved 96.2% accuracy.

The CNN did not require much hyperparameter tuning. The CNN achieved near state-of-the-art success (accuracy > 95%) on the first try. The CNN achieved 100% accuracy after 15 optimization evaluations with SigOpt.

The final model hyperparameters were based on the first evaluation that classified the test set with 100% accuracy. These values are shown in Table 4.1. Similarly, the final performances are shown below in Figure 4.2.

Table 4.1: Final hyperparameters chosen for both models.

Name	RNN-LSTM	CNN
`dropout_rate`	0.672	0.309
`learning_rate`	0.00001	0.033
`beta1`	0.9	0.684
`beta2`	0.999	0.845
`epsilon`	1e-08	0.282
`num_filt_1`	-	16
`kernel_size`	-	4
`num_fc_1`	31	6
`n_layers`	4	-
`n_hidden`	22	-

4.2 Final Results and Discussion

Overall, both CNN and RNN models achieved above 90% accuracy on the validation and test dataset given sufficient time. Figure 4.2 depicts the accuracy curves during training across the training and validation datasets.

Different training parameters and hyperparameters were defined for each model to achieve these results. The training parameters of both models saw a change in the batch size `batch_size` and number of epochs `n_epochs`. The batch size was increased from 32 to 256 so each model update would better represent the dataset. The models

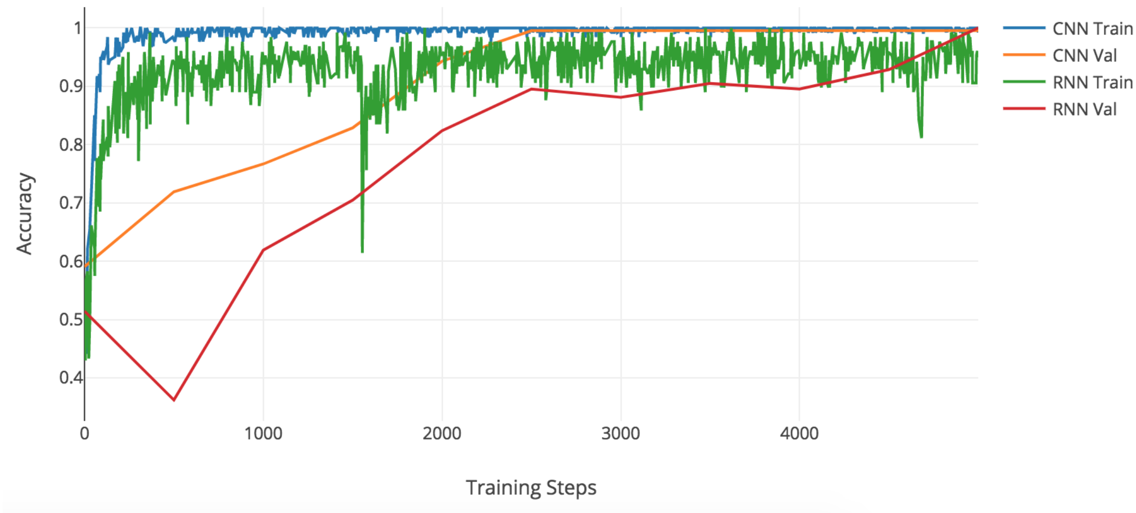


Figure 4.2: Classification Accuracy During Training

were ran until a validation dataset accuracy above 90% was observed, hence the final value `n_epochs` = 1000.

The CNN requires significantly less time to train than the RNN-LSTM. This can be explained by looking at the mathematics behind the architectures. At each time step t, a RNN-LSTM must perform the following computations:

$$
\begin{aligned}
\mathbf{g}^u &= \sigma(\mathbf{W}^u\mathbf{h}_{t-1} + \mathbf{I}^u\mathbf{x}_t + \mathbf{b_u}) \\
\mathbf{g}^f &= \sigma(\mathbf{W}^f\mathbf{h}_{t-1} + \mathbf{I}^f\mathbf{x}_t + \mathbf{b_f}) \\
\mathbf{g}^o &= \sigma(\mathbf{W}^o\mathbf{h}_{t-1} + \mathbf{I}^o\mathbf{x}_t + \mathbf{b_o}) \\
\mathbf{g}^c &= \tanh(\mathbf{W}^c\mathbf{h}_{t-1} + \mathbf{I}^c\mathbf{x}_t + \mathbf{b_c}) \\
\mathbf{m}_t &= \mathbf{g}^f \odot \mathbf{m}_{t-1} + \mathbf{g}^u \odot \mathbf{g}^c \\
\mathbf{h}_t &= \tanh\left(\mathbf{g}^o \odot \mathbf{m}_t\right)
\end{aligned}
\tag{4.1}
$$

where σ is the logistic sigmoid function, $\odot$ represents elementwise multiplication, $\mathbf{W}^u, \mathbf{W}^f, \mathbf{W}^o, \mathbf{W}^c$ are recurrent weight matrices, $\mathbf{I}^u, \mathbf{I}^f, \mathbf{I}^o, \mathbf{I}^c$ are projection matrices, $\mathbf{b}$ is the bias vector, and $\mathbf{h}$ and $\mathbf{m}$ are hidden and memory vectors responsible for controlling state updates and outputs [18]. On the other hand, the input to some unit x_i^l in layer l is the sum of the previous layer's cells contributions y multiplied by a filter ω with size m [11]. More clearly,

$$x_i^l = \sum_{a=0}^{m} \omega_a y_{i+a}^{l-1} + b_i \tag{4.2}$$

Compared directly against the fundamental equations behind a 1D convolution layer, one can see a stark contrast in complexity. Even if the filter or the number of previous-layer inputs are large in size, the CNN model is significantly simpler than the RNN-LSTM model and thus is easier and faster to train.

The CNN also outperformed the RNN-LSTM model in classification capability. The RNN-LSTM model feeds the hidden layer from the previous layer from the previous step into the next step to provide information for tasks requiring long-range contextual information, but the input data here is based on short, simulated step responses. The additional computations aren't needed for classifying the data in this work; in fact, the RNN-LSTM is incorrectly biased on the built-up memory. The CNN is looking for specific patterns within windows of time within the time-series data. The clean, short simulated data does not vary in sequence length and has repeatable patterns within the data so the CNN is able to quickly train and accurately classify input data.

4.3 Future Work

This work laid down a foundation to explore an ANN-based TPMS, but much more work needs to be done before this technology can be applied. Future work should attempt to address the following aspects not covered here.

1. **Improve the simulated data**. In this work, all data was generated from a quarter-car model simulation. The simulation made many assumptions and is not representative of a real car model. A better simulation can be made by using a half-car or full-car model instead of a quarter-car model or making generally less assumptions.

2. **Collect experimental data**. Even better than simulated data is real exper-

imental data. Collecting and analyzing real data can result in a better, more generalized classifier with no issues arising from training on simulated data. Furthermore, the data should be generalized away from a step function profile to the acceleration profile of general driving such that the TPMS can identify underpressurized tires at all times.

3. **Develop the hardware**. Instead of assuming the computational and electrical power required for the system exists, a more-thorough investigation should be performed to determine the validity of the claim. A theoretical system with the properly specified requirements would bring this work one step closer to reality.

4. **Improve the algorithm**. Further fine-tuning the training parameters and hyperparameters as well as adding and removing layers and features from the model architecture may result in more efficient and effective models.

4.4 Conclusion

Considering the various limitations of the work, these ANN-based TPMSs are far away from being applied across the automotive industry. Nonetheless, this work showed that both a CNN and RNN-LSTM model can be developed and trained on simulated training data to accurately classify unseen simulation data. This proves the algorithm's ability to identify unique patterns across each class and sort accordingly, all without any explicit instruction on the mechanical principles behind the data. With better data and appropriate hardware, vehicles may one day be equipped with ANN-based TPMS.

BIBLIOGRAPHY

[1] A. Abdelghaffar, A. Hendy, O. Desouky, Y. Badr, S. Abdulla, and R. Tafreshi. Effects of Different Tire Pressures on Vibrational Transmissibility in Cars. In *Mechanical Engineering and Mechatrnoics: 3rd International Conference, ICMEM 2014, Proceedings*, 8 2014.

[2] C. Alippi, C. de Russis, and V. Piuri. A Neural-Network Based Control Solution to Air-Fuel Ratio Control for Automotive Fuel-Injection Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 33(2):259–268, July 2003. doi: 10.1109/TSMCC.2003.814035.

[3] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[4] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks, Feb. 2016. doi: 10.4271/2002-01-1250.

[5] A. Deshpande. A Beginner's Guide to Understanding Convolutional Neural Networks, 8 2015.

[6] I. Dewancker, M. McCourt, S. Clark, P. Hayes, A. Johnson, and G. Ke. A Stratified Analysis of Bayesian Optimization Methods. *CoRR*, abs/1603.09441, 2016.

[7] J. Dixon. *The Shock Absorber Handbook*. Wiley-PEPublishing Series. John Wiley & Sons, 2007.

[8] G. Dorffner. Neural Networks for Time Series Processing. *Neural Network World*, 6:447–468, 1996.

[9] J. C. B. Gamboa. Deep learning for time-series analysis. *CoRR*, abs/1701.01887, 2017.

[10] M. Giaraffa. Springs & Dampers, Part Three, 2017.

[11] A. Gibiansky. Math [Code], Feb 2014.

[12] K. Habib. PE 16-007. Technical report, NHTSA, Jan. 2017.

[13] M. Hollemans. Convolutional Neural Networks on the iPhone with VGGnet, 8 2016.

[14] D. V. III and D. Vomhof. Individual vehicle data search service. Technical report, 4N6XPRT Systems, La Mesa CA 91942, 5 2012.

[15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[16] R. Jazar. *Vehicle Dynamics: Theory and Application*. Springer-Verlag New York, 2 edition, 2014.

[17] T. Jiang, S. Petrovic, U. Ayyer, A. Tolani, and S. Husain. Self-Driving Cars: Disruptive or Incremental? *Applied Innovation Review*, 4(1):3–22, June 2015.

[18] F. Karim, S. Majumdar, H. Darabi, and S. Chen. LSTM Fully Convolutional Networks for Time Series Classification. *CoRR*, abs/1709.05206, 2017.

[19] U. Karn. An intuitive explanation of convolutional neural networks, 8 2016.

[20] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[21] A. Kubba and K. Jiang. A Comprehensive Study on Technologies of Tyre Monitoring Systems and Possible Energy Solutions. *Sensors*, 14(6):10306–10345, June 2014. doi: 10.3390/s140610306.

[22] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[23] MathWorks. Automotive Suspension, May 2016.

[24] J. Mccaffrey. Why You Should Use Cross-Entropy Error Instead of Classification Error or Mean Squared Error for Neural Network Classifier Training, 11 2013.

[25] NHTSA. Docket No. NHTSA 2205-20586, 2005.

[26] NHTSA. Tire Pressure Maintenance - A Statistical Investigation, Apr. 2009.

[27] NHTSA. IV. Tire Pressure Monitoring Systems, 2016.

[28] NHTSA. Tires, 2016.

[29] C. Olah. Understanding LSTM Networks, 8 2015.

[30] J. Overton, B. Mills, and C. Ashley. The Vertical Response Characteristics of the Non-Rolling Tyre. In *Proceedings of The Institution of Mechanical Engineers, Automobile Division 1947-1970*, volume 184, pages 25–40, 6 1969.

[31] N. Persson, F. Gustafsson, and M. Drevo. Indirect Tire Pressure Monitoring Using Sensor Fusion. In *SAE Technical Paper", publisher = SAE International, journal = SAE, year = 2002, month = Mar, note = doi: 10.4271/2002-01-1250, url = https://doi.org/10.4271/2002-01-1250, doi = 10.4271/2002-01-1250.*

[32] E. Phaisangittisagul. An analysis of the regularization between l2 and dropout in single hidden layer neural network. In *2016 7th International Conference on Intelligent Systems, Modelling and Simulation (ISMS)*, pages 174–179, 1 2016.

[33] A. Plieninger. Deep Learning Neural Networks on Mobile Platforms. Master's thesis, Technische Universitat Munchen, Oct. 2015.

[34] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for Activation Functions. *CoRR*, abs/1710.05941, 2017.

[35] D. Silver and D. Hassabis. Alphago: Mastering the ancient game of Go with Machine Learning, 1 2016.

[36] R. Tayler, L. Bashford, and M. Shcrock. Methods for Measuring Vertical Tire Stiffness. *Transactions of the ASAE*, 43(6):1415–1419, 2000. doi: 10.13031/2013.3039.

[37] TireWise. Tire Maintenance, 2016.

[38] United States Senate and House of Representatives. Transportation Recall Enhancement, Accountability, and Documentation (TREAD) Act, Nov. 2000.

[39] V. Wiggins, S. Engquist, and L. Looper. Neural Network Applications: A Literature Review. Technical report, Air Force, Nov. 1992.

[40] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. Zhao. Time Series Classification Using Multi-Channels Deep Convolutional Neural Networks. In *Web-Age Information Management: 15th International Conference, WAIM 2014, Proceedings*, pages 298–310, 6 2014.

APPENDICES

Appendix A

DERIVATIONS OF SIMULATION INPUT PARAMETERS

All constants used as simulation input variables are derived as follows. Except for identifying m_u, all of these calculations are performed in `CalculateTireStiffness.m` and
`CalculateSuspensionStiffnessDamping.m`.

A.1 m_s

m_s is simply taken from [14] and divided by 4 to account for the quarter-car model.

$$m_s = 1109\,\mathrm{kg}/4 = 277.25\,\mathrm{kg} \tag{A.1}$$

A.2 k_s

Assuming that the tire in use across all vehicles is a radial-ply 165x13 tire (a very common tire size found on most passenger vehicles), a linear model for static stiffness based on tire inflation pressure can be used [30]. The model is graphically presented in Figure A.1 and expressed by equation A.2. The model is only accurate above 15 psi—an acceptable limitation as 15 psi is well below the threshold for "under-pressurized."

$$k_s = 30.185 p_u + 46.375 \tag{A.2}$$

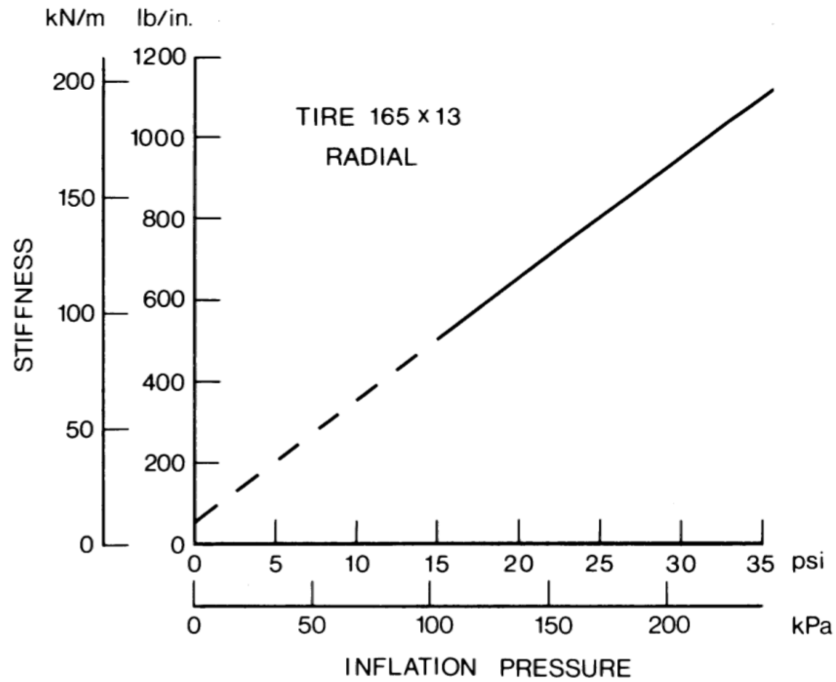


Figure A.1: Static stiffness vs. inflation pressure for a radial-ply car tire. [30]

A.3 m_u

Utilizing the average quarter-car ratio of the sprung to unsprung masses and the one can identify the expected value for m_u [16]:

$$\begin{aligned} \varepsilon = \frac{m_s}{m_u} = 8 \Rightarrow m_u &= \frac{m_s}{\varepsilon} \\ &= \frac{277.25\,\text{kg}}{8} \\ &= 34.69\,\text{kg} \end{aligned} \tag{A.3}$$

It should be noted that m_u should vary with tire pressure due to the additional air inside the tires. However, the mass of the air is insignificant relative to the rest of the unsprung mass (($< 0.1\%$). Nonetheless, the mass of the air is calculated and included in the unsprung mass for these simulations. The calculations are performed in `CalculateTireWeight.m`.

A.4 k_s and c_s

To identify the suspension's stiffness and damping coefficients, assume that the suspension is tuned for a properly-inflated tire. With $p_u = 32$ psi, equations A.2 and A.3 give the $k_u = 6979.53\,\mathrm{kPa}$ and $m_u = 34.69\,\mathrm{kg}$ respectively. With these values, one can find the natural frequency of the unsprung mass ω_u:

$$\begin{aligned} \omega_u &= \sqrt{\frac{k_u}{m_u}} \\ &= \sqrt{\frac{6979.53\,\mathrm{kPa}}{34.69\,\mathrm{kg}}} \\ &= 448.612\,\mathrm{Hz} \end{aligned} \tag{A.4}$$

The average quarter-car ratio for sprung and unsprung natural frequences is used to identify the sprung mass's natural frequency ω_s.

$$\begin{aligned} \alpha = \frac{\omega_s}{\omega_u} = 0.1 \Rightarrow \omega_s &= \alpha\omega_u \\ &= (0.1)(448.61\,\mathrm{Hz}) \\ &= 44.86\,\mathrm{Hz} \end{aligned} \tag{A.5}$$

We already know that $m_s = 277.25\,\mathrm{kg}$, so identifying k_s is trivial.

$$\begin{aligned} \omega_s = \sqrt{\frac{k_s}{m_s}} \Rightarrow k_s &= {\omega_s}^2 m_s \\ &= (44.86\,\mathrm{Hz})^2(277.25\,\mathrm{kg}) \\ &= 557.97\,\mathrm{kPa} \end{aligned} \tag{A.6}$$

To calculate c_s, we can use the relationship between ω_s and the damping ratio $\zeta = \frac{c_s}{c}$, where c is the critical damping coefficient. Numerous sources suggest the proper damping ratio in passenger vehicles to be between 0.2 and 0.3 [7] [10]. For this work, we define $\zeta = 0.25$.

$$
\begin{aligned}
\zeta = \frac{c_s}{2m_s\omega_s} \Rightarrow c_s &= 2\zeta m_s\omega_s \\
&= 2(0.25)(277.25\,\mathrm{kg})(44.86\,\mathrm{Hz}) \\
&= 6218.8\,\mathrm{Pa-s}
\end{aligned}
\tag{A.7}
$$

Appendix B

SOURCE CODE

B.1 cnn_model.py

```
"""Created on 24 June 2017.
@author: Alex Kost
@description: Main python code file for Applying CNN as a TPMS.
"""

# Basic Python
import logging

# Extended Python
import tensorflow as tf

# Alex Python
from data_processor import SIM_LENGTH_SEQ


class CNNModel(object):
    """
    CNNModel is a class that builds and trains a CNN Model.

    Attributes:
        accuracy (TensorFlow operation): step accuracy (predictions vs. labels)
        beta1 (float): exponential decay rate for the 1st moment estimates
        beta2 (float): exponential decay rate for the 2nd moment estimates
        cost (TensorFlow operation): cross entropy loss
        dropout_rate (float): dropout rate; 0.1 == 10% of input units drop out
        epsilon (float): a small constant for numerical stability
        kernel_size (int): kernel size in conv layer
        learning_rate (float): learning rate, used for optimizing
        logger (logger object): logging object to write to stream/file
        n_classes (int): number of classifications: under, nominal, over pressure
        n_features (int): number of features in input feature data: sprung_accel
        num_fc_1 (int): number of neurons in first fully connected layer
        num_filt_1 (int): number of filters in conv layer
```

```python
        optimizer (TensorFlow operation): AdamOptimizer operation used to train the model
        summary_op (TensorFlow operation): summary operation of all tf.summary objects
        trainable (TensorFlow placeholder): boolean flag to separate training/evaluating
        x (TensorFlow placeholder): input feature data
        y (TensorFlow placeholder): input label data
    """

    def __init__(self):
        """Constructor."""
        # HYPERPARAMETERS
        self.num_filt_1 = 16                        # number of filters in conv layer
        self.kernel_size = 5                        # kernel size in conv layer
        self.num_fc_1 = 30                          # number of neurons in first fully connected layer
        self.dropout_rate = 0.2                     # dropout rate; 0.1 == 10% of input units drop out
        self.learning_rate = 0.001                  # learning rate, used for optimizing
        self.beta1 = 0.9                            # exponential decay rate for the 1st moment estimates
        self.beta2 = 0.999                          # exponential decay rate for the 2nd moment estimates
        self.epsilon = 1e-08                        # a small constant for numerical stability

        # CONSTANT
        self.n_features = 1                         # sprung_accel
        self.n_classes = 3                          # classifications: under, nominal, over pressure
        self.logger = logging.getLogger(__name__)   # get the logger!

        # MODEL MEMBER VARIABLES
        self.x = None                               # input data
        self.y = None                               # input label
        self.cost = None                            # cross entropy loss
        self.accuracy = None                        # step accuracy (predictions vs. labels)
        self.optimizer = None                       # optimizing operation
        self.trainable = tf.placeholder(tf.bool, name='trainable')  # flag to separate training/evaluating
```

```python
        self.summary_op = None                      # summary operation to write data

    def build_model(self):
        """Build the CNN Model."""
        input_shape = [None, SIM_LENGTH_SEQ, self.n_features] if self.n_features > 1 else [None, SIM_LENGTH_SEQ]
        self.x = tf.placeholder(tf.float32, shape=input_shape, name='input_data')
        self.y = tf.placeholder(tf.int64, shape=[None], name='input_labels')

        with tf.variable_scope("Reshape_Data"):
            # tf.nn.conv2d requires inputs to be shaped as follows:
            # [batch, in_height, in_width, in_channels]
            # so -1 = batch size, should adapt accordingly
            # in_height = "height" of the image (so one dimension)
            # in_width = "width" of image
            x_reshaped = tf.reshape(self.x, [-1, SIM_LENGTH_SEQ, self.n_features])
            self.logger.debug('Input dims: {}'.format(x_reshaped.get_shape()))

        with tf.variable_scope("ConvBatch1"):
            x_bn = tf.contrib.layers.batch_norm(inputs=x_reshaped,
                                                is_training=self.trainable,
                                                updates_collections=None)

            conv1 = tf.layers.conv1d(inputs=x_bn,
                                     filters=self.num_filt_1,
                                     kernel_size=[self.kernel_size])
            self.logger.debug('Conv1 output dims: {}'.format(conv1.get_shape()))

        with tf.variable_scope("Fully_Connected1"):
            conv2_flatten = tf.layers.flatten(conv1, name='Flatten')
            fc1 = tf.contrib.layers.fully_connected(inputs=conv2_flatten,
                                                    num_outputs=self.num_fc_1,
                                                    weights_initializer=tf.contrib.layers.xa
                                                    biases_initializer=tf.constant_initializ
                                                    normalizer_fn=tf.contrib.layers.batch_no
                                                    normalizer_params={'is_training': self.trainable,
                                                                       'updates_collections' None})
```

```
            fc1 = tf.layers.dropout(inputs=fc1, rate=self.dropout_rate, training=self.trainable)
            self.logger.debug('FCon1 output dims: {}'.format(fc1.get_shape()))

        with tf.variable_scope("Fully_Connected2"):
            pred = tf.contrib.layers.fully_connected(inputs=fc1,
                                                     num_outputs=self.n_classes,
                                                     weights_initializer=tf.contrib.layers.x
                                                     biases_initializer=tf.constant_initiali
            self.logger.debug('FCon2 output dims: {}'.format(pred.get_shape()))
            tf.summary.histogram('pred', pred)

        # MEASURE MODEL ERROR
        # Cross-Entropy: "measuring how inefficient our predictions are for describing the truth"
        #    http://colah.github.io/posts/2015-09-Visual-Information/
        #    https://stackoverflow.com/questions/41689451/valueerror-no-gradients-provided-for-an
        #    Use sparse softmax because we have mutually exclusive classes
        #    logits must be [batch_size, num_classes], label must be [batch_size]
        # tf.reduce_mean = reduces tensor to mean scalar value of tensor
        with tf.variable_scope("Softmax"):
            cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=pred, labels=self.y)
            self.cost = tf.reduce_mean(cross_entropy, name='cost')
            tf.summary.scalar('cross_entropy_loss', self.cost)

        # EVALUATE OUR MODEL
        # tf.argmax = returns index of the highest entry in a tensor along some axis.
        #     Predictions are probabilities corresponding to class (ex. [0.7 0.2 0.1])
        #     tf.argmax returns the most probable label (ex. 0)
        # tf.equal = compares prediction to truth, returns list of bools (T if correct, F if not)
        # tf.reduce_mean = reduces tensor to mean scalar value of tensor
        # tf.cast = convert bools to 1 and 0
        with tf.variable_scope("Evaluating"):
            correct_pred = tf.equal(tf.argmax(pred, 1), self.y)
            self.accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
            tf.summary.scalar('accuracy', self.accuracy)

```

```
        # OPTIMIZE OUR MODEL
        with tf.variable_scope("Optimizing"):
            self.optimizer = tf.train.AdamOptimizer(self.learning_rate,
                                                    beta1=self.beta1,
                                                    beta2=self.beta2,
                                                    
                                                     ↪   epsilon=self.epsilon).minimize(self.cost
```

B.2 rnn_model.py

```python
"""Created on 24 June 2017.
@author: Alex Kost
@description: Main python code file for Applying RNN as a TPMS.
"""

# Basic Python
import logging

# Extended Python
import tensorflow as tf

# Alex Python
from data_processor import SIM_LENGTH_SEQ


class RNNModel(object):
    """
    RNNModel is a class that builds and trains a RNN model with LSTM cells.

    Attributes:
        accuracy (TensorFlow operation): step accuracy (predictions vs. labels)
        beta1 (float): exponential decay rate for the 1st moment estimates
        beta2 (float): exponential decay rate for the 2nd moment estimates
        cost (TensorFlow operation): cross entropy loss
        dropout_rate (float): dropout rate; 0.1 == 10% of input units drop out
        epsilon (float): a small constant for numerical stability
        learning_rate (float): learning rate, used for optimizing
        logger (logger object): logging object to write to stream/file
        n_classes (int): number of classifications: under, nominal, over pressure
        n_features (int): number of features in input feature data: sprung_accel
        n_hidden (int): number of features per hidden layer in RNN
        n_layers (int): number of hidden layers in model
        num_fc_1 (int): number of neurons in first fully connected layer
        optimizer (TensorFlow operation): AdamOptimizer operation used to train the model
        summary_op (TensorFlow operation): summary operation of all tf.summary objects
        trainable (TensorFlow placeholder): boolean flag to separate training/evaluating
        x (TensorFlow placeholder): input feature data
        y (TensorFlow placeholder): input label data
```

```python
        """

    def __init__(self):
        """Constructor."""
        # HYPERPARAMETERS
        self.n_hidden = 8                           # number of features per hidden layer in LSTM
        self.num_fc_1 = 16                          # number of neurons in first fully connected layer
        self.n_layers = 2                           # number of hidden layers in model
        self.dropout_rate = 0.5                     # dropout rate; 0.1 == 10% of input units drop out
        self.learning_rate = 0.0001                 # learning rate, used for optimizing
        self.beta1 = 0.9                            # exponential decay rate for the 1st moment estimates
        self.beta2 = 0.999                          # exponential decay rate for the 2nd moment estimates
        self.epsilon = 1e-08                        # a small constant for numerical stability

        # CONSTANT
        self.n_features = 1                         # sprung_accel
        self.n_classes = 3                          # classifications: under, nominal, over pressure
        self.logger = logging.getLogger(__name__)   # get the logger!

        # MODEL MEMBER VARIABLES
        self.x = None                               # input data
        self.y = None                               # input label
        self.cost = None                            # cross entropy loss
        self.accuracy = None                        # step accuracy (predictions vs. labels)
        self.optimizer = None                       # optimizing operation
        self.trainable = tf.placeholder(tf.bool, name='trainable')  # flag to separate training/evaluating
        self.summary_op = None                      # summary operation to write data

    def build_model(self):
        """Build the RNN model."""
        input_shape = [None, SIM_LENGTH_SEQ, self.n_features] if self.n_features > 1 else [None, SIM_LENGTH_SEQ]
        self.x = tf.placeholder(tf.float32, shape=input_shape, name='input_data')
```

```python
        self.y = tf.placeholder(tf.int64, shape=[None], name='input_labels')

        if input_shape == [None, SIM_LENGTH_SEQ]:
            with tf.variable_scope("Reshape_Data"):
                # tf.nn.conv2d requires inputs to be shaped as follows:
                # [batch_size, max_time, ...]
                # so -1 = batch size, should adapt accordingly
                # max_time = SIM_LENGTH_SEQ
                # ... = self.n_features
                x_reshaped = tf.reshape(self.x, [-1, SIM_LENGTH_SEQ, self.n_features])
                self.logger.debug('Input dims: {}'.format(x_reshaped.get_shape()))

        with tf.variable_scope("LSTM_RNN"):
            # add stacked layers if more than one layer
            if self.n_layers > 1:
                cell = tf.contrib.rnn.MultiRNNCell([self._setup_lstm_cell() for _ in range(self.n_layers)],
                                                   state_is_tuple=True)
            else:
                cell = self._setup_lstm_cell()

            # outputs = [batch_size, max_time, cell.output_size]
            #   outputs contains the output of the last layer for each time-step
            outputs, _ = tf.nn.dynamic_rnn(cell=cell,
                                           inputs=x_reshaped,
                                           dtype=tf.float32)

            self.logger.debug('dynamic_rnn output dims: {}'.format(outputs.get_shape()))

            # We transpose the output to switch batch size with sequence size - http://monik.in/a-noobs-guide-to-implementing-rnn-lstm-using-tensorflow/
            outputs = tf.transpose(outputs, [1, 0, 2])      # Now shape = [max_time, batch_size, cell.output_size]
            last = outputs[-1]                              # Last slice is of shape [batch_size, cell.output_size]
            self.logger.debug('last output dims: {}'.format(last.get_shape()))

        with tf.variable_scope("Fully_Connected1"):
            fc1 = tf.contrib.layers.fully_connected(inputs=last,
                                                    num_outputs=self.num_fc_1,
                                                    weights_initializer=tf.contrib.layers.xa
```

```
                                                 biases_initializer=tf.constant_initializ
                                                 normalizer_fn=tf.contrib.layers.batch_nc
                                                 normalizer_params={'is_training':
                                                 self.trainable,
                                                                   'updates_collections'
                                                                   None})

            fc1 = tf.layers.dropout(inputs=fc1, rate=self.dropout_rate,
            training=self.trainable)
            self.logger.debug('FCon1 output dims: {}'.format(fc1.get_shape()))

        with tf.variable_scope("Fully_Connected2"):
            pred = tf.contrib.layers.fully_connected(inputs=fc1,
                                                 num_outputs=self.n_classes,
                                                  weights_initializer=tf.contrib.layers.x
                                                  biases_initializer=tf.constant_initiali
            self.logger.debug('FCon2 output dims: {}'.format(pred.get_shape()))
            tf.summary.histogram('pred', pred)

        # MEASURE MODEL ERROR
        # Cross-Entropy: "measuring how inefficient our predictions are for
        describing the truth"
        #    http://colah.github.io/posts/2015-09-Visual-Information/
        #
        https://stackoverflow.com/questions/41689451/valueerror-no-gradients-provided-for-ar
        #    Use sparse softmax because we have mutually exclusive classes
        #    logits must be [batch_size, num_classes], label must be [batch_size]
        # tf.reduce_mean = reduces tensor to mean scalar value of tensor
        with tf.variable_scope("Softmax"):
            cross_entropy =
            tf.nn.sparse_softmax_cross_entropy_with_logits(logits=pred,
            labels=self.y)
            self.cost = tf.reduce_mean(cross_entropy, name='total')
            tf.summary.scalar('cross_entropy_loss', self.cost)

        # EVALUATE OUR MODEL
        # tf.argmax = returns index of the highest entry in a tensor along some
        axis.
```

```
        #     Predictions are probabilities corresponding to class (ex. [0.7 0.2 0.1])
        #     tf.argmax returns the most probable label (ex. 0)
        # tf.equal = compares prediction to truth, returns list of bools (T if correct, F if not)
        # tf.reduce_mean = reduces tensor to mean scalar value of tensor
        # tf.cast = convert bools to 1 and 0
        with tf.variable_scope("Evaluating"):
            correct_pred = tf.equal(tf.argmax(pred, 1), self.y)
            self.accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
            tf.summary.scalar('accuracy', self.accuracy)

        # OPTIMIZE OUR MODEL
        with tf.variable_scope("Optimizing"):
            self.optimizer = tf.train.AdamOptimizer(self.learning_rate,
                                                    beta1=self.beta1,
                                                    beta2=self.beta2,
                                                    epsilon=self.epsilon).minimize(self.cost

    """ Helper Functions """
    def _setup_lstm_cell(self):
        """Creates an LSTM Cell to be unrolled.

        There's a bug in tf.contrib.rnn.MultiRNNCell that requires we create
        new cells every time we want to a mult-layered RNN. So we use this
        helper function to create a LSTM cell. See more here:
        https://github.com/udacity/deep-learning/issues/132#issuecomment-325158949

        Returns:
            cell (BasicLSTMCell): BasicLSTM Cell
        """
        # forget_bias set to 1.0 b/c http://proceedings.mlr.press/v37/jozefowicz15.pdf
        cell = tf.nn.rnn_cell.BasicLSTMCell(self.n_hidden, forget_bias=1.0, state_is_tuple=True)

        return cell
```

B.3 data_processor.py

```python
"""Created on 17 December 2017.
@author: Alex Kost
@description: Main python code file for preprocessing data

Attributes:
    SIM_DATA_PATH (str): Local simulation data output folder path
    SIM_LENGTH_FIX (int): bias to datapoint length due to slicing ops in Matlab,
     ↪ datapoints
    SIM_LENGTH_SEQ (int): simulation length, datapoints
    SIM_LENGTH_TIME (float): simulation time, sec
    SIM_RESOLUTION (float): simulation resolution, sec/datapoint
"""

# Basic Python
import logging
import os

# Extended Python
import numpy as np

# Simulation Constants
SIM_LENGTH_TIME = 1.5 - .45
SIM_RESOLUTION = .001
SIM_LENGTH_FIX = 2
SIM_LENGTH_SEQ = int(SIM_LENGTH_TIME / SIM_RESOLUTION) + SIM_LENGTH_FIX
SIM_DATA_PATH = 'Data/simulated_labeled'


class DataProcessor(object):
    """
    DataProcessor is a class that processes datasets.

    Attributes:
        logger (logger object): logging object to write to stream/file
        n_classes (int): number of classifications: under, nominal, over pressure
        n_features (int): number of features in input feature data: sprung_accel
        test_data (np.array): loaded data from test dataset
        test_files (list of strings): list of filenames in test dataset
        train_data (np.array): loaded data from training dataset
        train_files (list of strings): list of filenames in training dataset
        val_data (np.array): loaded data from validation dataset
```

```python
            val_files (list of strings): list of filenames in validation dataset
        """
    def __init__(self, n_classes, n_features):
        """Constructor

        Args:
            n_classes (int): label classifications
            n_features (int): features per example
        """
        # assign input variables
        self.n_classes = n_classes
        self.n_features = n_features

        # FILENAME LISTS
        self.train_files = []
        self.val_files = []
        self.test_files = []

        # LOADED DATA
        self.train_data = None
        self.val_data = None
        self.test_data = None

        self.logger = logging.getLogger(__name__)   # get the logger!

    def preprocess_all_data(self):
        """Shuffle all data and then preprocess the files."""
        all_files = self._create_filename_list(SIM_DATA_PATH)
        np.random.shuffle(all_files)

        train_val_test_files = self._split_datafiles(all_files)    # train_set,
        ↪  val_set, test_set
        self.train_files = train_val_test_files[0]
        self.val_files = train_val_test_files[1]
        self.test_files = train_val_test_files[2]

        # Report sizes and load all datasets
        self.logger.info('Train set size: %d', len(self.train_files))
        self.logger.info('Validation set size: %d', len(self.val_files))
        self.logger.info('Test set size: %d', len(self.test_files))
        self._load_all_datasets()

    def preprocess_data_by_label(self):
        """Simulation data is organized by label. This method mixes and splits up
        ↪  the data."""
```

```
        for i in range(self.n_classes):
            modified_data_path = os.path.join(SIM_DATA_PATH, str(i))
            class_files = self._create_filename_list(modified_data_path)

            # get files for each thing
            result = self._split_datafiles(class_files)    # train_set, val_set, test_set
            self.train_files.extend(result[0])
            self.val_files.extend(result[1])
            self.test_files.extend(result[2])
            self.logger.debug('%d/%d/%d added to train/val/test set from class %d.',
                              len(result[0]), len(result[1]),
                              len(result[2]), i)

        # Shuffle data
        np.random.shuffle(self.train_files)
        np.random.shuffle(self.val_files)
        np.random.shuffle(self.test_files)

        # Report sizes and load all datasets
        self.logger.info('Train set size: %d', len(self.train_files))
        self.logger.info('Validation set size: %d', len(self.val_files))
        self.logger.info('Test set size: %d', len(self.test_files))
        self._load_all_datasets()

    """ Helper Functions """
    def _load_all_datasets(self):
        """Assign class member variables after processing filenames."""
        self.train_data = self._load_data(self.train_files)   # features, labels
        self.val_data = self._load_data(self.val_files)         # features, labels
        self.test_data = self._load_data(self.test_files)       # features, labels

    @staticmethod
    def _create_filename_list(data_dir):
        """Identify the list of CSV files based on a given data_dir.

        Args:
            data_dir (string): local path to where the data is saved.

        Returns:
            filenames (list of strings): a list of CSV files found in the data directory
```

```
        """
        filenames = []
        for root, _, files in os.walk(data_dir):
            for filename in files:
                if filename.endswith(".csv"):
                    rel_filepath = os.path.join(root, filename)
                    abs_filepath = os.path.abspath(rel_filepath)
                    filenames.append(abs_filepath)

        return filenames

    @staticmethod
    def _split_datafiles(data, val_size=0.2, test_size=0.2):
        """Spit all the data we have into training, validating, and test sets.

        By default, 60/20/20 split
        Credit:
↪   https://www.slideshare.net/TaegyunJeon1/electricity-price-forecasting-with-recurrent-neural-

        Args:
            data (list): list of filenames
            val_size (float, optional): Percentage of data to be used for
↪   validation set
            test_size (float, optional): Percentage to data set to be used for
↪   test set

        Returns:
            train_set (list): list of training example filenames
            val_set (list): list of validation example filenames
            test_set (list): list of test example filenames
        """
        val_length = int(len(data) * val_size)
        test_length = int(len(data) * test_size)

        val_set = data[:val_length]
        test_set = data[val_length:val_length + test_length]
        train_set = data[val_length + test_length:]

        return train_set, val_set, test_set

    def _load_data(self, filenames):
        """Load data from the filenames

        Args:
            filenames (list of strings): filenames
```

```python

        Returns:
            features, labels (np.array, np.array): loaded features and labels
        """
        # Get features and labels from dataset
        features, labels = [], []
        for example_file in filenames:
            example_data = np.loadtxt(example_file, delimiter=',')

            ex_label = example_data[0, 0] if self.n_features > 1 else
             ↪  example_data[0]
            ex_feature = example_data[:, 1:] if self.n_features > 1 else
             ↪  example_data[1:]

            features.append(ex_feature)
            labels.append(ex_label)

        # stack features
        features = np.vstack(features)

        return features, labels
```

B.4 train.py

```python
"""Created on 24 June 2017.
@author: Alex Kost
@description: Training class for CNN and RNN models

Attributes:
    DEFAULT_FORMAT (str): Logging format
    LOGFILE_NAME (str): Logging file name
    OUTPUT_DIR (str): TensorBoard output directory
"""

# Basic Python
import logging
import os
from time import strftime
from math import ceil

# Extended Python
import progressbar
import tensorflow as tf

# Alex Python
from data_processor import DataProcessor
from rnn_model import RNNModel      # RNN MODEL
from cnn_model import CNNModel      # CNN MODEL

# Progressbar config
progressbar.streams.wrap_stderr()

# Constants
DEFAULT_FORMAT = '%(asctime)s: %(levelname)s: %(message)s'
LOGFILE_NAME = 'train.log'
OUTPUT_DIR = 'output'


class TrainModel(DataProcessor):
    """
    TrainModel is a class that builds and trains a provided model.

    Attributes:
        batch_size (int): number of examples in a single batch
        dropout_rate (float): dropout rate; 0.1 == 10% of input units drop out
```

```
        learning_rate (float): learning rate, used for optimizing
        logger (logger object): logging object to write to stream/file
        model (TensorFlow model object): Model to train and evaluate
        n_checks (int): number of times to check performance while training
        n_epochs (int): number of times we go through all data
        summary_op (TensorFlow operation): summary operation of all tf.summary objects
    """
    def __init__(self, model, n_epochs=20, batch_size=32):
        """Constructor.

        Args:
            model (TensorFlow model object): Model to train and evaluate
            n_epochs (int, optional): number of times we go through all data
            batch_size (int, optional): number of examples in a single batch
        """
        # TRAINING PARAMETERS
        self.n_epochs = n_epochs
        self.batch_size = batch_size

        # CONSTANT
        self.model = model
        self.summary_op = None
        self.logger = logging.getLogger(__name__)
        self.n_checks = 5

        # INPUT DATA/LABELS
        super(TrainModel, self).__init__(self.model.n_classes, self.model.n_features)
        self.preprocess_data_by_label()

        # HELPER VARIABLES
        self._ex_per_epoch = None
        self._steps_per_epoch = None
        self._train_length_ex = None
        self._train_length_steps = None
        self.calculate_helpers()

    def calculate_helpers(self):
        """Calculate helper variables for training length."""
        self._ex_per_epoch = len(self.train_files)
        self._steps_per_epoch = int(ceil(self._ex_per_epoch / float(self.batch_size)))
        self._train_length_ex = self._ex_per_epoch * self.n_epochs
        self._train_length_steps = self._steps_per_epoch * self.n_epochs
```

```python

        self.logger.debug('self._ex_per_epoch: %d', self._ex_per_epoch)
        self.logger.debug('self._steps_per_epoch: %d', self._steps_per_epoch)
        self.logger.debug('self._train_length_ex: %d', self._train_length_ex)
        self.logger.debug('self._train_length_steps: %d',
         ↪  self._train_length_steps)

    def train_model(self, use_tensorboard=True):
        """Train the model.

        Args:
            use_tensorboard (bool, optional): Description

        Returns:
            TYPE: Description
        """

        # SETUP TENSORBOARD FOR NEW RUN
        if use_tensorboard:
            checkpoint_prefix, run_dir = self._setup_tensorboard_directories()
            saver = tf.train.Saver(tf.global_variables())
        else:
            self.logger.info('*** NEW RUN ***')
        self._log_training_and_model_params()
        self.summary_op = tf.summary.merge_all()

        # TRAIN
        with tf.Session() as sess:
            # Initialization
            progress_bar =
             ↪  progressbar.ProgressBar(max_value=self._train_length_steps)
            sess.run(tf.global_variables_initializer())
            if use_tensorboard:
                train_writer = tf.summary.FileWriter(run_dir + '/train',
                 ↪  sess.graph)
                val_writer = tf.summary.FileWriter(run_dir + '/val')
            batch_idx = 0
            progress_bar.start()
            progress_bar.update(0)

            self.logger.info("The training shall begin.")
            try:
                _, acc_test_before, _ = self.evaluate_model_on_data(sess, 'test')
                for step in range(self._train_length_steps):
                    # Reset/increment batch_idx
```

```
if step % self._steps_per_epoch == 0:
    batch_idx = 0
else:
    batch_idx += 1

if use_tensorboard:
    do_full_eval = step % ceil(self._train_length_steps / float(self.n_checks)) == 0
    do_full_eval = do_full_eval or (step == self._train_length_steps - 1)
    if do_full_eval:
        # Check training and validation performance
        cost_train, acc_train, _ = self.evaluate_model_on_data(sess, 'train')
        cost_val, acc_val, summary = self.evaluate_model_on_data(sess, 'val')

        # Report information to user
        self.logger.info('%d epochs elapsed.', step / self._steps_per_epoch)
        self.logger.info('COST:     Train: %5.3f / Val: %5.3f', cost_train, cost_val)
        self.logger.info('ACCURACY: Train: %5.3f / Val: %5.3f', acc_train, acc_val)

        # Save to Tensorboard
        val_writer.add_summary(summary, step)
        saver.save(sess, checkpoint_prefix, global_step=step)

        # # If model is not learning immediately, break out of training
        # if acc_val == acc_test_before and step > 100:
        #     self.logger.info('Stuck on value: %d', acc_val)
        #     break

# Training step
x_batch, y_batch = self._generate_batch(batch_idx)
_, summary = sess.run([self.model.optimizer, self.summary_op],
                      feed_dict={self.model.x: x_batch,
                                 self.model.y: y_batch,
                                 self.model.trainable: True})

# Save to Tensorboard, update progress bar
if use_tensorboard:
    train_writer.add_summary(summary, step)
```

```
                    progress_bar.update(step)
            except KeyboardInterrupt:
                self.logger.info('Keyboard Interrupt? Gracefully quitting.')
            finally:
                progress_bar.finish()
                _, acc_test_after, _ = self.evaluate_model_on_data(sess, 'test')
                self.logger.info("The training is done.")
                self.logger.info('Test accuracy before training: %.3f.', acc_test_before)
                self.logger.info('Test accuracy after training: %.3f.', acc_test_after)
                if use_tensorboard:
                    train_writer.close()
                    val_writer.close()

        return acc_test_after

    def evaluate_model_on_data(self, sess, dataset_label):
        """Evaluate the model on the entire training data.

        Args:
            sess (tf.Session object): active session object
            dataset_label (string): dataset label

        Returns:
            float, float: the cost and accuracy of the model based on the dataset.
        """
        try:
            dataset_dict = {'test': self.test_data,
                            'train': self.test_data,
                            'val': self.val_data}
            dataset = dataset_dict[dataset_label]
        except KeyError:
            raise '"dataset" arg must be in dataset dict: {}'.format(dataset_dict.keys())

        cost, acc, summary = sess.run([self.model.cost, self.model.accuracy, self.summary_op],
                                      feed_dict={self.model.x: dataset[0],
                                                 self.model.y: dataset[1],
                                                 self.model.trainable: False})

        return cost, acc, summary

    @staticmethod
```

```python
    def reset_model():
        """Reset the model to prepare for next run."""
        tf.reset_default_graph()

    """ Helper Functions """
    def _setup_tensorboard_directories(self):
        """Set up TensorBoard directories.

        Returns:
            checkpoint_prefix, run_dir (string, string): checkpoint prefix, output root folder
        """
        timestamp = str(strftime("%Y.%m.%d-%H.%M.%S"))
        model_type = self.model.__class__.__name__.replace('Model', '')
        model_name = timestamp + '_' + model_type
        out_dir = os.path.abspath(os.path.join(os.path.curdir, OUTPUT_DIR))
        run_dir = os.path.abspath(os.path.join(out_dir, model_name))
        checkpoint_dir = os.path.abspath(os.path.join(run_dir, "checkpoints"))
        checkpoint_prefix = os.path.join(checkpoint_dir, "model")
        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)

        # Logging the Run
        self.logger.info('*** NEW RUN ***')
        self.logger.info('filename: %s', model_name)

        return checkpoint_prefix, run_dir

    def _log_training_and_model_params(self):
        """Record new run details."""
        model_type = self.model.__class__.__name__

        self.logger.info('  *** TRAINING ***')
        self.logger.info('    n_epochs: %d', self.n_epochs)
        self.logger.info('    batch_size: %d', self.batch_size)
        self.logger.info('  *** MODEL ***')
        if 'CNN' in model_type:
            self.logger.info('    num_filt_1: %d', self.model.num_filt_1)
            self.logger.info('    kernel_size: %d', self.model.kernel_size)
            self.logger.info('    num_fc_1: %d', self.model.num_fc_1)
        elif 'RNN' in model_type:
            self.logger.info('    n_hidden: %d', self.model.n_hidden)
            self.logger.info('    num_fc_1: %d', self.model.num_fc_1)
            self.logger.info('    n_layers: %d', self.model.n_layers)

```

```python
        self.logger.info('    dropout_rate: %f', self.model.dropout_rate)
        self.logger.info('    learning_rate: %f', self.model.learning_rate)
        self.logger.info('    beta1: %f', self.model.beta1)
        self.logger.info('    beta2: %f', self.model.beta2)
        self.logger.info('    epsilon: %f', self.model.epsilon)

    def _generate_batch(self, batch_idx):
        """Generate a batch and increment the sliding batch window within the
        ↪  data."""
        features = self.train_data[0]
        labels = self.train_data[1]

        start_idx = batch_idx * self.batch_size
        end_idx = start_idx + self.batch_size - 1

        # Error handling for if sliding window goes beyond data list length
        if end_idx > self._ex_per_epoch:
            end_idx = self._ex_per_epoch

        if self.n_features > 1:
            x_batch = features[:, start_idx:end_idx]
        else:
            x_batch = features[start_idx:end_idx]

        y_batch = labels[start_idx:end_idx]
        self.logger.debug('batch_idx: %d', batch_idx)
        self.logger.debug('Got training examples %d to %d', start_idx, end_idx)

        return x_batch, y_batch


def main():
    """Sup Main!"""
    models = [CNNModel(), RNNModel()]
    for model in models:
        model.build_model()
        train = TrainModel(model, n_epochs=200, batch_size=128)
        train.train_model()
        train.reset_model()


if __name__ == '__main__':
    # create logger with 'spam_application'
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)
```

```
    # create file handler which logs even debug messages
    fh = logging.FileHandler(LOGFILE_NAME)
    fh.setLevel(logging.INFO)
    # create console handler with a higher log level
    ch = logging.StreamHandler()
    ch.setLevel(logging.INFO)
    # create formatter and add it to the handlers
    formatter = logging.Formatter(DEFAULT_FORMAT)
    fh.setFormatter(formatter)
    ch.setFormatter(formatter)
    # add the handlers to the logger
    logger.addHandler(fh)
    logger.addHandler(ch)

    main()
```

B.5 tune.py

```
"""Created on 6 Jan 2017.
@author: Alex Kost
@description: mastermind tuning script for model

Attributes:
    DEFAULT_FORMAT (str): Logging format
    LOGFILE_NAME (str): Logging file name
    OUTPUT_DIR (str): TensorBoard output directory
"""

# Basic Python
import logging

# Extended Python
from sigopt import Connection

# Alex Python
from train import TrainModel
from rnn_model import RNNModel     # RNN MODEL
from cnn_model import CNNModel     # CNN MODEL

# Constants
DEFAULT_FORMAT = '%(asctime)s: %(levelname)s: %(message)s'
LOGFILE_NAME = 'tune.log'
#EXPERIMENT_ID = 34189           # CNNModel Accuracy v1
#EXPERIMENT_ID = 34205           # CNNModel Accuracy v2
#EXPERIMENT_ID = 34424           # CNNModel Accuracy v3

EXPERIMENT_ID = 34631            # RNNModel Accuracy v1


class SigOptTune(object):
    def __init__(self):
        """Constructor."""
        self.logger = logging.getLogger(__name__)   # get the logger!

        self.conn = Connection(client_token="XWCROUDALHMNJFABTLYVXBUHISZQKKACUGULCENHPSZNQPSD")
        self.conn.set_api_url("https://api.sigopt.com")
        self.experiment = None
        self.suggestion = None
```

```

        self.model = None
        self.acc = None

    def create_cnn_experiment(self):
        """Create experiment. Modify as needed."""
        self.experiment = self.conn.experiments().create(
            name="CNNModel Accuracy v3",
            parameters=[dict(name="learning_rate",
                             bounds=dict(min=0.00001, max=0.1),
                             type="double"),
                        dict(name="dropout_rate",
                             bounds=dict(min=0.2, max=0.9),
                             type="double"),
                        dict(name="beta1",
                             bounds=dict(min=0.0001, max=0.999),
                             type="double"),
                        dict(name="beta2",
                             bounds=dict(min=0.0001, max=0.999),
                             type="double"),
                        dict(name="epsilon",
                             bounds=dict(min=1e-8, max=1.0),
                             type="double"),
                        dict(name="num_filt_1",
                             bounds=dict(min=1, max=40),
                             type="int"),
                        dict(name="kernel_size",
                             bounds=dict(min=1, max=10),
                             type="int"),
                        dict(name="num_fc_1",
                             bounds=dict(min=1, max=40),
                             type="int")
                        ])

        self.logger.info('Experiment created! ID %d.', self.experiment.id)

    def create_rnn_experiment(self):
        """Create experiment. Modify as needed."""
        self.experiment = self.conn.experiments().create(
            name="RNNModel Accuracy v1",
            parameters=[dict(name="learning_rate",
                             bounds=dict(min=0.00001, max=0.1),
                             type="double"),
                        dict(name="dropout_rate",
                             bounds=dict(min=0.2, max=0.9),
```

```
                             type="double"),
                        dict(name="beta1",
                             bounds=dict(min=0.0001, max=0.999),
                             type="double"),
                        dict(name="beta2",
                             bounds=dict(min=0.0001, max=0.999),
                             type="double"),
                        dict(name="epsilon",
                             bounds=dict(min=1e-8, max=1.0),
                             type="double"),
                        dict(name="n_hidden",
                             bounds=dict(min=1, max=40),
                             type="int"),
                        dict(name="num_fc_1",
                             bounds=dict(min=1, max=40),
                             type="int"),
                        dict(name="n_layers",
                             bounds=dict(min=1, max=10),
                             type="int")
                        ])

        self.logger.info('Experiment created! ID %d.', self.experiment.id)

    def get_suggestions(self):
        """Create suggestions for next iteration."""
        try:
            self.suggestion = self.conn.experiments(EXPERIMENT_ID).suggestions().create()
            logger.info('Created new suggestions.')
        except:
            self.conn.experiments(EXPERIMENT_ID).suggestions().delete(state="open")
            self.suggestion = self.conn.experiments(EXPERIMENT_ID).suggestions().create()
            logger.info('Deleted old and created new suggestions.')

    def update_parameters(self):
        """Update model parameters with suggestions."""
        #model_type = self.model.__class__.__name__.replace('Model', '')

        params = self.suggestion.assignments
        # if model_type == 'CNN':
        #     self.model.num_filt_1 = int(params['num_filt_1'])
        #     self.model.kernel_size = int(params['kernel_size'])
        #     self.model.num_fc_1 = int(params['num_fc_1'])
```

```
        # elif model_type == 'RNN':
        #     self.model.n_hidden = int(params['n_hidden'])
        #     self.model.num_fc_1 = int(params['num_fc_1'])
        #     self.model.n_layers = int(params['n_layers'])

        #self.model.dropout_rate = params['dropout_rate']
        self.model.learning_rate = params['learning_rate']
        self.model.beta1 = params['beta1']
        self.model.beta2 = params['beta2']
        self.model.epsilon = params['epsilon']

    def report_observation(self):
        """Report observation to SigOpt."""
        self.conn.experiments(EXPERIMENT_ID).observations().create(
                suggestion=self.suggestion.id,
                value=float(self.acc),
                value_stddev=0.05)

    def optimization_loop(self, model):
        """Optimize the parameters based on suggestions."""
        for i in range(100):
            self.logger.info('Optimization Loop Count: %d', i)

            # assign suggestions to parameters and hyperparameters
            self.get_suggestions()

            # update model class
            self.model = model()
            self.update_parameters()
            self.model.build_model()

            # update training class
            train = TrainModel(self.model, n_epochs=200, batch_size=128)

            # run the training stuff
            self.acc = train.train_model()
            train.reset_model()

            # report to SigOpt
            self.report_observation()


class GridSearchTune(object):
    def __init__(self):
        """Constructor."""
```

```python
        self.logger = logging.getLogger(__name__)   # get the logger!

    def tune_cnn_with_gridsearch():
        """Grid search to identify best hyperparameters for CNN model."""
        cnn_model_values = []
        n_epoch_list = [100, 200, 300, 400, 500]                         # 5
        batch_size_list = [16, 32, 64, 128, 256]                         # 5
        learning_rate_list = [.0001, .0005, .00001, .00005]              # 4
        dropout_rate_list = [0.2, 0.5, 0.7]                              # 3

        try:
            for n_epoch in n_epoch_list:
                for batch_size in batch_size_list:
                    for learning_rate in learning_rate_list:
                        for dropout_rate in dropout_rate_list:
                            for num_filt_1 in [8, 16, 32]:             # CNN ONLY # 3
                                for num_filt_2 in [10, 20, 30, 40]:      # CNN ONLY # 4
                                    for num_fc_1 in [10, 20, 30, 40]:   # CNN ONLY # 4
                                        CNN = TrainModel(CNNModel, n_epoch, batch_size, learning_rate, dropout_rate)
                                        CNN.model.num_filt_1 = num_filt_1
                                        CNN.model.num_filt_2 = num_filt_2
                                        CNN.model.num_fc_1 = num_fc_1
                                        CNN.model.build_model()
                                        CNN.calculate_helpers()
                                        acc = CNN.train_model()
                                        CNN.reset_model()

                                        results = [acc, n_epoch, batch_size, learning_rate, dropout_rate, num_filt_1, num_filt_2, num_fc_1]
                                        cnn_model_values.append(results)
        except:
            pass
        finally:
            best_cnn_run = max(cnn_model_values, key=lambda x: x[0])
            logger.info('Best CNN run: {}'.format(best_cnn_run))
```

```python
            logger.info('All CNN runs: {}'.format(cnn_model_values))

    def tune_rnn_with_gridsearch():
        """Grid search to identify best hyperparameters for RNN."""
        rnn_model_values = []
        n_epoch_list = [200, 400, 600, 800, 1000]                        # 5
        batch_size_list = [16, 32, 64, 128, 256]                         # 5
        learning_rate_list = [.001, .005, .0001, .0005]                  # 4
        dropout_rate_list = [0.2, 0.5, 0.7]                              # 3

        for n_epoch in n_epoch_list:
            for batch_size in batch_size_list:
                for learning_rate in learning_rate_list:
                    for dropout_rate in dropout_rate_list:
                        for n_hidden in [8, 16, 32]:               # RNN ONLY
                            for num_fc_1 in [10, 20, 30, 40]:      # RNN ONLY
                                for n_layers in [1, 2, 3]:         # RNN ONLY
                                    RNN = TrainModel(RNNModel, n_epoch, batch_size, learning_rate, dropout_rate)
                                    RNN.model.n_hidden = n_hidden
                                    RNN.model.num_fc_1 = num_fc_1
                                    RNN.model.n_layers = n_layers

                                    RNN.model.build_model()
                                    RNN.calculate_helpers()
                                    acc = RNN.train_model()
                                    RNN.reset_model()

                                    rnn_model_values.append([acc, n_epoch, batch_size, learning_rate, dropout_rate, n_hidden, num_fc_1, n_layers])

            best_rnn_run = max(rnn_model_values, key=lambda x: x[0])
            logger.info('Best RNN run: {}'.format(best_rnn_run))
            logger.info('All RNN runs: {}'.format(rnn_model_values))


def main():
    """Sup Main!"""
    tune = SigOptTune()
    #tune.create_cnn_experiment()
```

```
    #tune.optimization_loop(CNNModel)
    #tune.create_rnn_experiment()
    tune.optimization_loop(RNNModel)

if __name__ == '__main__':
    # create logger with 'spam_application'
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    # create file handler which logs even debug messages
    fh = logging.FileHandler(LOGFILE_NAME)
    fh.setLevel(logging.INFO)
    # create console handler with a higher log level
    ch = logging.StreamHandler()
    ch.setLevel(logging.INFO)
    # create formatter and add it to the handlers
    formatter = logging.Formatter(DEFAULT_FORMAT)
    fh.setFormatter(formatter)
    ch.setFormatter(formatter)
    # add the handlers to the logger
    logger.addHandler(fh)
    logger.addHandler(ch)

    main()
```

B.6 main.m

```
%% Quarter Model Simulation MAIN
% Alex Kost
% Thesis
%
% Main file for quarter model simulation procedure.
%
% Arguments (see 'Test Parameters' section):
%   num_psis = num of psis to simulate
%   psi_min = minimum PSI to simulate
%   psi_max = maximum PSI to simulate
%   num_steps = num of step sizes to simulate
%   step_min = minimum step size to simulate
%   step_max = maximum step size to simulate
%   sim_tim = how long to run the simulation
%   snr = signal-to-noise ratio per sample, dB
%   save_path = path to save the simulation data
%
% Simulation data will output as plots and CSVs

%% Reset workspace and hide figures
clc
clear all
close all
set(0, 'DefaultFigureVisible', 'off');
set(0, 'DefaultFigureWindowStyle', 'docked');

%% Test parameters (user-provided)
num_psi = 25;                % number of psis to simulate
psi_min = 25.5;              % minimum psi
psi_max = 38.5;              % maximum psi

num_steps = 39;              % number of step sizes to simulate
step_min = .1;               % minimum step size, m
step_max = 2;                % maximum step size, m

sim_time = 1.5;              % simulation time, s
snr = 0;                     % signal-to-noise ratio per sample, dB

% save data path
save_path = '/Users/alexkost/Dropbox/Grad Life/thesis/Data/simulated_labeled/';
%% Test parameters (predefined)
```

```
% Create a range of PSIs and Steps using defined values above
psi_all = linspace(psi_min, psi_max, num_psi);
steps_all = linspace(step_min, step_max, num_steps);

% ICs for simulations (cannot be nested in functions)
IC = [-1.74412834455962e-12
    -2.44861738501480e-06
    -5.70054231468026e-11
    -7.99748963336152e-05];

%% Run simulations and get outputs (CSVs and plots)
for i=1:num_steps
    step_size = steps_all(i);
    figure(i)
    hold on;
    for j=1:num_psi
        % run simulation
        psi = psi_all(j);
        simout = QuarterModelSimulation(psi, ...
                                        step_size, ...
                                        sim_time);

        % Add white gaussian noise if snr > 0
        if snr > 0
            for k=1:size(simout, 2)
                simout(:,k) = awgn(simout(:, k), snr);
            end
        end

        % interpret simulation outputs
        sprung_pos = simout(:,1);
        %sprung_vel = simout(:,2);
        %unsprung_pos = simout(:,3);
        %unsprung_vel = simout(:,4);
        step = simout(:,5);             % constant every run
        time = simout(:,6);             % constant every run
        sprung_acc = simout(:,7);
        %unsprung_acc = simout(:,8);

        % Plot individual run
        str = strcat(num2str(psi, '%.1f'), ' psi');
        plot(time, sprung_pos, 'DisplayName', str);

        % calculate label value
        if psi < 30
```

```
            label_val = 0;
        elseif psi <= 34
            label_val = 1;
        elseif psi > 34
            label_val = 2;
        end

        % Output to CSV
        % Modifications done for Tensorflow
        %    use sprung acceleration data only (1 feature)
        %    transpose so each row is independent example
        %    remove first .45 seconds of data
        filename = strcat('Sim_', ...
                          num2str(psi, '%.1f'), 'psi_', ...
                          num2str(step_size, '%.2f'), 'm.csv');
        fullfilename = fullfile(save_path, num2str(label_val), filename);
        acc_transposed = [sprung_acc]';
        M = acc_transposed(:,(.45/.001):end);
        label_val_column = ones(size(M, 1),1) * label_val;
        csvwrite(fullfilename, horzcat(label_val_column, M));
    end

    % create figure with step
    plot(time, step,'--','DisplayName','Step');
    hold off;
    title(sprintf('Quarter-Car Motion\nStep size = %g [m]', step_size));
    xlabel('Time (s)');
    ylabel('Vehicle height (m)');
    legend('show');

    % save figure
    filename = sprintf('Plot_step_size_%g.png', step_size);
    fullfilename = fullfile(save_path, filename);
    print(figure(i),fullfilename,'-dpng','-r300');
end
```

B.7 QuarterModelSimulation.m

```matlab
function [ simout ] = QuarterModelSimulation(psi, y, sim_time)
% QuarterModelSimulation runs a Simulink model based on provided PSI
% and outputs the relevant data to be used elsewhere

global m_s m_u c_s k_s k_u g alpha zeta

%% Constants
N_over_lb = 4.448;       % [N / lb]
m_over_in = .0254;       % [m / in]
m_over_mm = .001;        % [m /mm]
Pa_over_psi = 6894.76;   % [Pa / psi]
g = 9.81;                % gravity, m/s^2

%% Vehicle parameters (user-provided)
m_s_full = 1109;                    % full body mass, kg
zeta = .25;                         % dampening ratio
epsilon = 8;                        % sprung/unsprung mass ratio
alpha = .1;                         % natural frequency ratio

%% Vehicle parameters (calculated)
m_s = m_s_full / 4;                 % quarter body mass, kg
m_air = CalculateTireWeight(psi);   % mass of air in tire, kg
m_u = (m_s / epsilon) + m_air;      % quarter unsprung mass, kg

%% Calculate suspension values from ideal conditions (32 psi)
Pa_over_psi = 6894.76;  % [Pa / psi]
k_u_eng = 30.185 * psi + 46.375;            % unsprung stiffness, lb/in
k_u = k_u_eng * Pa_over_psi;                % unsprung stiffness, N/m
omega_u = sqrt(k_u/m_u);                    % unsprung natural freq, Hz
k_s = alpha^2 * m_s * omega_u^2;            % sprung stiffness, N/m
omega_s = sqrt(k_s/m_s);                    % sprung natural freq, Hz
c_s = 2 * zeta * sqrt(k_s * m_s);           % spring damping, N/(m/s)


[ k_s, c_s, omega_s ] = CalculateSuspensionStiffnessDamping(32);

%% Calculate tire stiffness from PSI
% Unsprung mass refers to all masses that are attached to and not supported by the spring, such as wheel, axle, or brakes.
[ k_u, omega_u ] = CalculateTireStiffness(psi);

```

```
%% Check we have all the values we need for the simulation
debug = 0;
if debug
    fprintf('psi = %f [psi]\n', psi);
    fprintf('step_size = %f [m]\n', y);
    fprintf('m_s = %f [kg]\n', m_s);
    fprintf('m_u = %f [kg]\n', m_u);
    fprintf('c_s = %f [N/(m/s)]\n', c_s);
    fprintf('k_s = %f [N/m]\n', k_s);
    fprintf('k_u = %f [N/m]\n', k_u);
    fprintf('g = %f [m/s^2]\n', g);
    % And print out the stuff that we don't need anyways
    fprintf('omega_s = %f [Hz]\n', omega_s);
    fprintf('omega_u = %f [Hz]\n', omega_u);
end

%% run Simulink simulation
sim('QuarterModelMatrix.slx', sim_time);

end
```

B.8 CalculateSuspensionStiffnessDamping.m

```
function [ k_s, c_s, omega_s ] = CalculateSuspensionStiffnessDamping(psi)
% Function to identify stiffness and damping coefficients based on tire psi
% Other important parameters are defined by globals
global m_s m_u alpha zeta

%% Constants
Pa_over_psi = 6894.76;  % [Pa / psi]

%% Calculations
k_u_eng = 30.185*psi + 46.375;              % unsprung stiffness, lb/in
k_u = k_u_eng * Pa_over_psi;                % unsprung stiffness, N/m
omega_u = sqrt(k_u/m_u);                    % unsprung natural freq, Hz
k_s = alpha^2 * m_s * omega_u^2;            % sprung stiffness, N/m
omega_s = sqrt(k_s/m_s);                    % sprung natural freq, Hz
c_s = 2 * zeta * sqrt(k_s * m_s);           % spring damping, N/(m/s)

end
```

B.9 CalculateTireStiffness.m

```
function [ k_u, omega_u ] = CalculateTireStiffness(psi)
% Function to identify stiffness and damping coefficients based on tire psi
% Other important parameters are defined by globals
global m_u

%% Constants
Pa_over_psi = 6894.76;   % [Pa / psi]

%% Calculations
k_u_eng = 30.185*psi + 46.375;              % unsprung stiffness, lb/in
k_u = k_u_eng * Pa_over_psi;                % unsprung stiffness, N/m
omega_u = sqrt(k_u/m_u);                    % unsprung natural freq, Hz

end
```

B.10 SimFunc.m

```
function [ xDD ] = SimFunc(u)
% SimFunc is used in QuarterModelMatrix.slx
% All motions of equation are in matrix form and done here to keep
% the simulink model clean. GUIs can be painful sometimes.

global m_s m_u c_s k_s k_u g

% Reassign Simulink values for readability
x_s = u(1);         % sprung mass height, m
x_s_d = u(2);       % sprung mass velocity, m/s
x_u = u(3);         % unsprung mass height, m
x_u_d = u(4);       % unsprung mass velocity, m/s
y = u(5);           % road height (step input), m

% Assign matrix elements
M11 = m_s;
M12 = 0;
M21 = 0;
M22 = m_u;

C11 = c_s;
C12 = -c_s;
C21 = -c_s;
C22 = c_s;

K11 = k_s;
K12 = -k_s;
K21 = -k_s;
K22 = k_s + k_u;

F11 = m_u*(-g);
F21 = k_u*y + m_s*(-g);

% Assemble matrices
M = [M11 M12;
     M21 M22];

C = [C11 C12;
     C21 C22];

K = [K11 K12;
```

```
     K21 K22];

F = [F11;
     F21];

X_d = [x_s_d;
       x_u_d];

X = [x_s;
     x_u];

% Assemble the matrix form of the equation of motion
A = F - (C*X_d) - (K*X);

% Calculating x_s_ddot and x_u_ddot
% https://www.mathworks.com/help/matlab/ref/mldivide.html
xDD = M\A;

% % Equation form
% F_s = -k_s*(x_s - x_u) - c_s*(x_s_d - x_u_d);
% F_u = k_s*(x_s - x_u) + c_s*(x_s_d - x_u_d) - k_u*(x_u - y);
%
% xDD = [F_s/m_s;
%        F_u/m_u];

end
```

B.11 `QuarterModelMatrix.slx`

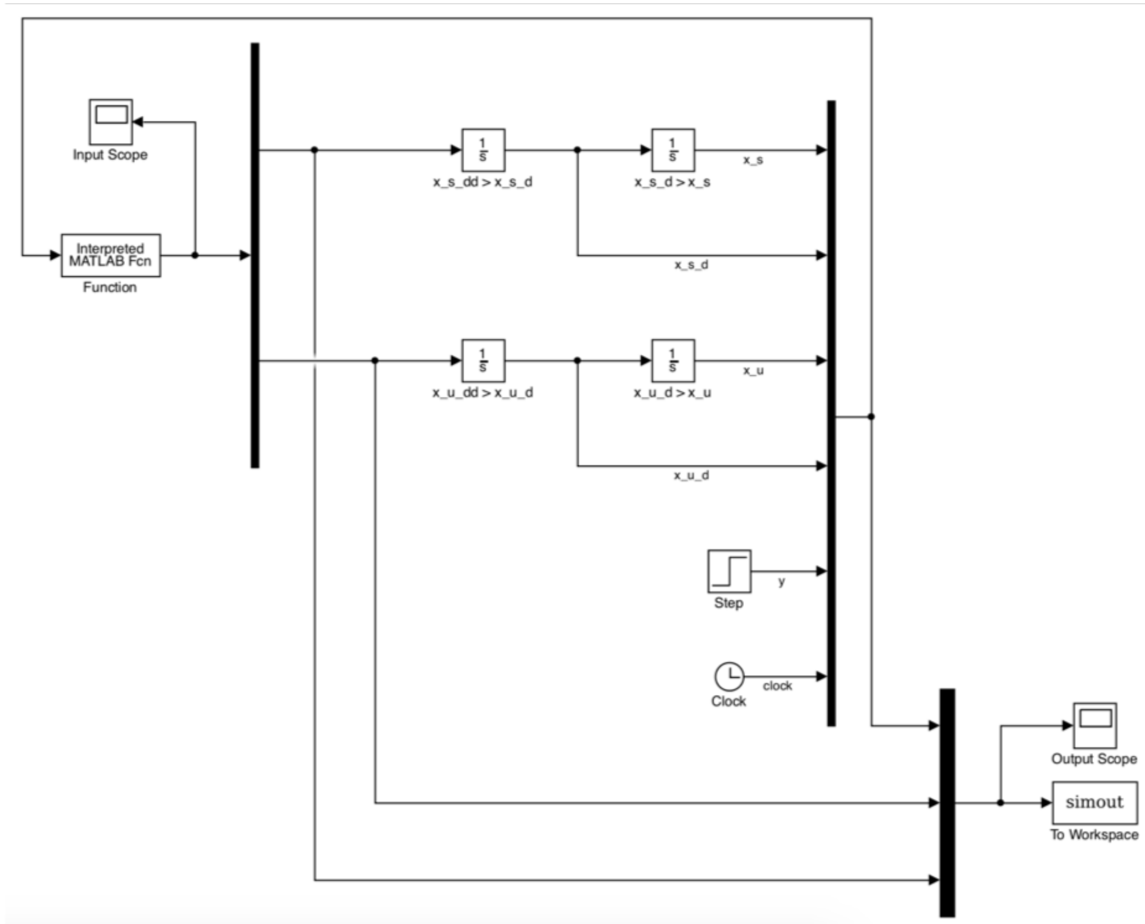


Figure B.1: `QuarterModelMatrix.slx`