DEEP LEARNING FOR SENSOR FUSION

by

SHAUN M. HOWARD

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August, 2017

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Shaun M. Howard

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Wyatt S. Newman

Committee Member

Dr. Murat Cenk Cavusoglu

Committee Member

Dr. Michael S. Lewicki

Date of Defense

May 22nd, 2017

*We also certify that written approval has been obtained

for any proprietary material contained therein.

Contents

List of Tables

List of Figures

List of Acronyms

r^2-metric coefficient of determination.

ADAS advanced driver assistance systems.

ANN artificial neural network.

API application programming interface.

AUC area under the curve.

BLAS Basic Linear Algebra Subprograms.

BP backpropagation.

BPTT backpropagation through time.

CNN convolutional neural network.

FC fully-connected.

GIL global interpreter lock.

GPS global-positioning system.

GPU graphical processing unit.

GRU gated recurrent unit.

hdf5 hierarchical data format 5.

ID identification.

iid independently and identically distributed.

LiDAR Light Detection and Ranging.

LSTM long-short term memory.

MKL multiple kernel learning.

MLP multi-layer perceptron.

ms millisecond.

MSE mean-squared error.

radar RAdio Detection And Ranging.

RMSE root mean-squared error.

RMSProp root mean-squared error propagation.

RNN recurrent neural network.

ROC receiver operating characteristic.

RSS residual sum of squares.

seq2seq sequence-to-sequence.

SGD stochastic gradient descent.

TTC time to collision.

XOR exclusive OR.

Deep Learning for Sensor Fusion

Abstract

by

SHAUN M. HOWARD

The use of multiple sensors in modern day vehicular applications is necessary to provide a complete outlook of surroundings for advanced driver assistance systems (ADAS) and automated driving. The fusion of these sensors provides increased certainty in the recognition, localization and prediction of surroundings. A deep learning-based sensor fusion system is proposed to fuse two independent, multi-modal sensor sources. This system is shown to successfully learn the complex capabilities of an existing state-of-the-art sensor fusion system and generalize well to new sensor fusion datasets. It has high precision and recall with minimal confusion after training on several million examples of labeled multi-modal sensor data. It is robust, has a sustainable training time, and has real-time response capabilities on a deep learning PC with a single NVIDIA GeForce GTX 980Ti graphical processing unit (GPU).

Chapter 1

Introduction

1.1 Motivation

The application of deep learning for sensor fusion is important to the future of self-driving vehicles. Autonomous vehicles must be able to accurately depict and predict their surroundings with enhanced robustness, minimal uncertainty and high reliability. Given the lack of human interaction in these vehicles, they need to be self-aware and resilient in order to deal with their surroundings intelligently and handle errors logically. In order to determine the effectiveness of deep learning for the sensor fusion task, this research seeks to evaluate the applicability, success rate, and performance of novel deep learning techniques in order to learn and provide the complex capabilities of a preexisting state-of-the-art sensor fusion system.

1.2 Sensors

A sensor is a piece of hardware that monitors an environment based on a sensing element. Sensors vary in quality and price. Many modern sensors have on-board processing units to understand the data acquired from the sensing element without a separate computational platform. Remote sensing devices are sensors that can

perceive information without physical contact. Unfortunately, sensors tend to have inherent problems including, but not limited to [Elmenreich, 2001]:

- **Spatial coverage limits**: Each sensor may only cover a certain region of space. For example, a dashboard camera will observe less surrounding region than a camera with a wide-view lens.
- **Temporal coverage limits**: Each sensor may only provide updates over certain periods of time, which may cause uncertainty between updates.
- **Imprecision**: Each sensor has limits to its sensing element.
- **Uncertainty**: Unlike imprecision, uncertainty varies with the object being observed rather than the device making the observation. Uncertainty may be introduced by many environment factors or sensor defects in addition to time delay.
- **Deprivation of sensor**: Sensor element breakdown will cause loss of perception in environment.

Many different types of sensors exist in the world, and each has its own unique application. Four sensors are pertinent to the field of autonomous driving [Levinson et al., 2011]. The four sensors, their descriptions, uses, advantages and disadvantages are mentioned below:

- **GPS**: Global-positioning system (GPS) is a system of satellites and receivers used for global navigation of Earth designed by the U.S. military. GPS sends a signal to any GPS receiver with an unobstructed line of sight to four or more GPS satellites surrounding Earth [gps, 2011]. GPS is useful for finding the exact coordinates of a vehicle when it is in the line of sight of multiple satellites orbiting the Earth.

  - **Advantages**: Precise coordinate measurements, fast, reliable in line of sight, externally-managed satellite systems.
  - **Disadvantages**: Expensive, subject to failure in bad weather conditions, subject to failure in distant locations where satellite coverage is blocked or unavailable, dependent on external data source, subject to hijacking and interference.

- **Radar**: RAdio Detection And Ranging (radar) is a remote sensing device that uses an antenna to scatter radio signals across a region in the direction it is pointing and listens for response signals that are reflected by objects in that area. Radar measures signal time of flight to determine the distance. Radars may use the doppler effect to compute speed based on shift in frequency of scattered waves as an object moves. Radar is useful for detecting obstacles, vehicles and pedestrians around a vehicle [Huang et al., 2016]. Tracking multiple targets at once is a primary use for an automotive radar.
  - **Advantages**: High-bandwidth signals, wide-spread area coverage, independent from external systems, works in multiple weather conditions, light-independent solution.
  - **Disadvantages**: Expensive, subject to interference, easy to corrupt signal with electromagnetic interference, many reflective radio responses make it harder to manage radar signals, algorithms for radar tracking are still imperfect, narrow field-of-view.

- **Camera**: A camera is an optical instrument that utilizes at least one converging or convex lens and a shutter to limit light intake into an enclosed housing for capturing images or recording image sequences [Kodak, 2017]. Video cameras work much like still-image cameras, but instead of simply capturing still images, they record a series of successive still images rapidly at a specific frame rate

[Kodak, 2017]. A camera is useful for acquiring images or video sequences of object pixels in view of the lens in order to help detect, segment, and classify objects based on perceivable object properties like location, color, shape, edges and corners.

  - **Advantages**: Perceives high-level object characteristics like color, shape, and edges, perceives location relative to camera unit, easy to visualize data.
  - **Disadvantages**: Potential slow frame-rate update, image quality may be dependent on light, weather and various other factors, data-intensive processing, all cameras perceive objects differently, typically has a limited range of perception compared to other sensors, may be expensive.

- **LiDAR**: Light Detection and Ranging (LiDAR) is a method of remote sensing that uses light in the form of a pulsed laser to measure distance to an object based on signal time of flight [NOAA, 2012]. LiDAR is useful for perceiving surroundings when 3-dimensional, high-resolution, light-independent images are necessary.

  - **Advantages**: Independent of light, weather and external data sources, fast, accurate, 3-dimensional, high-resolution.
  - **Disadvantages**: Expensive, subject to interference by reflection or lack thereof, incompatible with transparent surfaces, data-intensive processing, less durable than other sensors.

1.3 Sensor Fusion

Sensor fusion is the act of combining data acquired from two or more sensors sources such that the resulting combination of sensory information provides a more certain

description of factors observed by the separate sensors than would be if used individually [Elmenreich, 2001]. Sensor fusion is pertinent in many applications that entail the use of multiple sensors for inference and control. Examples of applications include intelligent and automated systems such as automotive driver assistance systems, autonomous robotics, and manufacturing robotics [Elmenreich, 2007].

Sensor fusion methods aim to solve many of the problems inherently present in sensors. Several important benefits may be derived from sensor fusion systems over single or disparate sensor sources. The benefits of sensor fusion over single source are the following [Elmenreich, 2001]:

- **Reliability**: Using multiple sensor sources introduces more resilience to partial sensor failure, which leads to greater redundancy and reliability.
- **Extended spatial coverage**: Each sensor may cover different areas. Combining the covered areas will lead to a greater overall coverage of surrounding environment and accommodate sensor deprivation.
- **Extended temporal coverage**: Each sensor may update at different time intervals, and thus interpolated sensor updates can be joined for increased temporal coverage and decreased sensor deprivation.
- **Increased Confidence**: Combining sensor data will provide increased confidence by providing measurements resilient to the uncertainties in any particular sensor based on the combined coverage and error mitigation of all sensors.
- **Reduced Uncertainty**: Given the resilience of multiple sensors to the specific uncertainty of any one, the overall uncertainty of the perception system can be drastically reduced using sensor fusion.
- **Robustness against noise**: Multiple sensor sources can be used to determine when any one sensor has encountered noise in order to mitigate influence of

noise in the system.

- **Increased Resolution**: Multiple sensor sources can be used to increase the resolution of measurements by combining all observations.

According to [Rao, 2001], sensor fusion can yield results that outperform the measurements of the single best sensor in the system if the fusion function class satisfies a proposed isolation property based on independently and identically distributed (iid) samples. The fusion function classes outlined that satisfy this isolation property are linear combinations, certain potential functions and feed-forward piecewise linear neural networks based on minimization of empirical error per sample.

1.4 Current Sensor Fusion Systems

Many researchers and companies have developed their own versions of sensor fusion systems for various purposes. Many of these systems are well-known an widely used in practice within the fields of automotive and robotics. In [Steux et al., 2002], a vehicle detection and tracking system using monocular color vision and radar data fusion using a 3-layer belief network was proposed called FADE. The fusion system focused on lower-level fusion and combined 12 different features to generate target position proposals each step and for each target. FADE performed in real-time and yielded good detection results in most cases according to scenarios recorded in a real car.

A fusion system for collision warning using a single camera and radar was applied to detect and track vehicles in [Srinivasa et al., 2003]. The detections were fused using a probabilistic framework in order to compute reliable vehicle depth and azimuth angles. Their system clustered object detections into meta-tracks for each object and fused object tracks between the sensors. They found that the radar had many false positives due to multiple detections on large vehicles, structures, roadway signs and

overhead structures. They also found that the camera had false positive detections on larger vehicles and roadway noise like potholes. Their system worked appropriately for nearby vehicles that were clearly visible by both sensors, but the system failed to detect vehicles more than 100 meters away due to insufficient resolution or vehicle occlusion.

In [Dagan et al., 2004], engineers from Mobileye successfully applied a camera system to compute the time to collision (TTC) course from size and position of vehicles in the image. Although they did not test this theory, they mentioned the future use of radar and camera in a sensor fusion system since the radar would give more accurate range and range-rate measurements while the vision would solve angular accuracy problems of the radar. When the research was conducted, it was suggested that the fusion solution between radar and camera was costly, but since then, costs have decreased.

A collision mitigation fusion system using a laser-scanner and stereo-vision was constructed and tested in [Labayrade et al., 2005]. The combination of the complimentary laser scanner and stereo-vision sensors provided a high detection rate, low false alarm rate, and a system reactive to many obstacle occurrences. They mentioned that the laser-scanner was fast and accurate but could not be used alone due to many false alarms from collisions with the road surface and false detections with laser passes over obstacles. They also mentioned that stereo-vision was useful for modeling road geometry and obstacle detection, but it was not accurate for computing precise velocities or TTC for collision mitigation.

In [Laneurit et al., 2003], a Kalman filter was successfully developed and applied for the purpose of sensor fusion between multiple sensors including GPS, wheel angle sensor, camera and LiDAR. They showed that this system was useful for detection and localization of vehicles on the road, especially when using the wheel angle sensor for detecting changes in vehicle direction. Their results revealed that cooperation

between the positioning sensors for obstacle detection and location paired with LiDAR were able to improve global positioning of vehicles.

A deep learning framework for signal estimation and classification applicable for mobile devices was created and tested in [Yao et al., 2016]. This framework applied convolutional and recurrent layers for regression and classification mobile computing tasks. The framework exploited local interactions of different sensing modalities using convolutional neural network (CNN)s, merged them into a global interaction and extracted temporal relationships via stacked GRU or LSTM layers. Their framework achieved a notable mean absolute error on vehicle tracking regression tasks as compared to existing sensor fusion systems and high accuracy on human activity recognition classification tasks while it remained efficient enough to use on mobile devices like the Google Nexus 5 and Intel Edison.

A multimodal, multi-stream deep learning framework designed to tackle the egocentric activity recognition using data fusion was proposed in [Song et al., 2016b]. To begin, they extended a multi-stream CNN to learn spatial and temporal features from egocentric videos. Then, they proposed a multi-stream LSTM architecture to learn features from multiple sensor streams including accelerometer and gyroscope. Third, they proposed a two-level fusion technique using SoftMax classification layers and different pooling methods to fuse the results of the neural networks in order to classify egocentric activities. The system performed worse than a hand-crafted multi-modal Fisher vector, but it was noted that hand-crafted features tended to perform better on smaller datasets. In review of the research, it seems there were limited amounts of data, flaws in the fusion design with SoftMax combination and flaws in the sensors, such as limited sensing capabilities. These factors all may have led to worse results than hand-crafted features on the utilized dataset.

In [Wu et al., 2015], a multi-stream deep fusion neural network system using convolution neural networks and LSTM layers was applied to classify multi-modal tempo-

ral stream information in videos. Their adaptive multi-stream fusion system achieved an accuracy level much higher than other methods of fusion including averaging, kernel averaging, multiple kernel learning (MKL), and logistic regression fusion methods.

1.5 Neural Networks

1.5.1 Perceptron

The perceptron is classically known as the "fundamental probabilistic model for information storage and organization of the brain" [Rosenblatt, 1958]. It is the key to modern machine learning applications. It was developed by an American psychologist, Frank Rosenblatt, during his time at the Cornell Aeronautical Laboratory in 1957. The perceptron is a simplified form of a hypothetical nervous system designed to emphasize central properties of intelligent systems that could learn without involving the unknown circumstances often portrayed by biological organisms [Rosenblatt, 1958]. The perceptron was developed based on early brain models which responded to stimuli sequences in order to perform certain algorithms using elements of symbolic logic and switching theory. Some of its predecessors were logical calculus models that sought to characterize nervous activity using propositional logic as in [Mcculloch and Pitts, 1943] and a finite set of "universal" elements as in [Minsky, 1956].

The perceptron differed from existing models in the sense that it sought to model deterministic systems like those of [Mcculloch and Pitts, 1943], [Kleene, 1956], and [Minsky, 1956], by the utilization of imperfect neural networks with many random connections, like the brain. The type of logic and algebra used to create the previous models was unsuitable for describing such imperfect networks, and this inability brought the onset of the perceptron which instead applied probability theory to describe such behaviors. The perceptron theory was based on work derived from

[Hebb, 1949] and [Hayek, 1952], who were much more concerned with the learning mechanisms of biological systems.

The theory on which the perceptron was based is summarized in the following:

- Physical nervous system connections used in learning and recognition are not identical between organisms.
- The system of connected cells may change, and the probability that a stimulus applied to one set of cells will cause a response in another set will change over time.
- Exposure to a large sample of similar stimuli will form pathways to similar sets of cells in response. Dissimilar stimuli samples will develop connections to different response cell sets.
- Positive or negative reinforcement may facilitate or hinder in-progress connection formation.
- Similarity is represented by a tendency of activation to the same sets of cells by similar stimuli. The perceptual world will be divided into classes by the system structure and the stimulus-environment ecology.

The organization of a perceptron model is demonstrated in figure 1.1. It was much like the systems described by [Mcculloch and Pitts, 1943] and [Minsky, 1956], but it embraced the randomness of biological systems. In modern times, the perceptron is useful classifying linearly-separable, binary data samples. A binary classifier maps a real-valued vector input x, to an output value $f(x)$, which satisfies the following:

$$f(x) = \begin{cases} 1 & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where w is a real-valued vector of synaptic weights, $w.x$ is the dot product $\sum_{i=1}^{m} w_i x_i$, m is the number of inputs, and b is the bias input. Thus, $f(x)$ classifies the input as a positive or negative instance. The bias is simply designated to shift the boundary of decision-making from the origin and can be negative, but the weighted combination of inputs must produce a value greater than $|b|$ for positive classification.

Linearly-separable data is data that can be separated into classes by at least a single 2-dimensional boundary line or a multi-dimension hyperplane. In binary classification, this means that only one decision boundary is necessary to properly separate data samples in the sample space. One of the main reasons why the perceptron is notably successful with linearly-separable, binary data is that the learning algorithm was designed to model a single decision boundary in the sample space. Additionally, the learning algorithm will not terminate if the training set does not have this property, and thus improper classifications will take place in data with more than 2 classes.

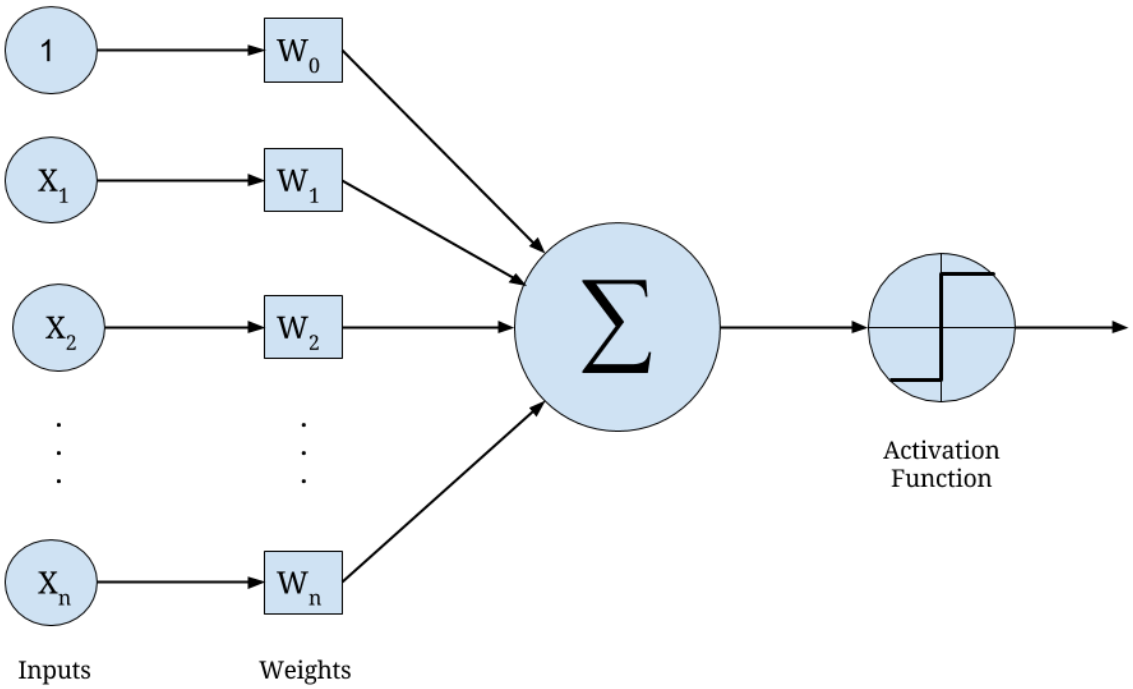

Figure 1.1: A perceptron model with n real-valued inputs each denoted as an element of the vector X along with a bias input of 1. There is a weight parameter along each input connection which leads to the dot product of the combined inputs and weights including the bias term and weight. The weights belong to the vector W, which is adjusted to tune the perceptron network. The output of the dot product is fed to an activation function which determines whether the perceptron fires with a 1 on output or does not fire with a 0 on output.

Given the fact that the perceptron by itself is considered a single element or layer of a neural network, it has limitations to what functions it can model by itself. Since the perceptron was designed to model a single decision boundary, it fails in multi-class classification with greater than two linearly-separable classes. One of the initial drawbacks found in the single-layer perceptron model was that it could not solve the exclusive OR (XOR) logic problem [Minsky and Papert, 1969]. This publication, despite its questionable assumptions, led to a great decrease in neural network research funding and interest until the 1980s. In order to solve this problem, it was found that multiple perceptrons could be chained together such that the outputs of the first perceptron layer could be fed-forward as the inputs of the second perceptron layer in order to solve the problem of multi-class classification. This multi-layer

network configuration, now known as the MLP, was proven to easily solve the XOR problem and even more complex mathematical functions in several works including [Grossberg, 1973] and [Hornik et al., 1989]. A modern MLP is shown in figure 1.2. The MLP came to belong to a class of neural networks called feedforward neural networks. These biologically-inspired perceptron models helped form the foundation of modern machine learning techniques which are successfully used in many practical applications today.

1.5.2 Feedforward Neural Networks

A feedforward neural network is an artificial neural network (ANN) where information flows in one direction and does not contain cycles. The perceptron is a feedforward neural network in the sense that inputs are fed-forward through the network to the output layer without any cycles. Not surprisingly, the multilayer perceptron is also classified as a feedforward neural network for this reason. Feedforward neural networks are useful in time-independent applications such as handwriting recognition, object classification, spectroscopy, protein folding and chromatography [Svozil et al., 1997].

A feed-forward neural network may have at least one hidden layer of neurons, as in the logistic regression case, and each of these neurons has its own activation function. Typically, each layer of neurons will share the same activation function. Each neuron may have multiple incoming and outgoing connections. Each of the connections are weighted, except input connections. Each layer has a bias term connected via a weighted connection to each and every neuron, although modifications have been made to remove the bias. This term can take on many values depending on the application.

Most feedforward neural networks are fully-connected neural networks with multiple layers, where each neuron in any given layer is connected to every other neuron in

the preceding and following layers, if either are present. The input is given at the first visible layer. The output is generated by the last visible layer. The layers in-between are known as hidden layers. A MLP with a single hidden layer is shown in figure 1.2. Layers are called "hidden" because the values used and created by these layers are not provided as input data, but rather they are generated by abstract, weighted mathematical functions contained within the network. More hidden neurons, which often entail more hidden layers, allow the network to generate more abstract representations and avoid over-fitting any specific examples, which is a desirable property for generalizing across a large feature space proportional to the number of training examples [Hornik et al., 1989]. However, too many hidden neurons or layers may slow training time, increase training difficulty given increased metric entropy, or drown out notable features when the learning space or number of training instances is small.

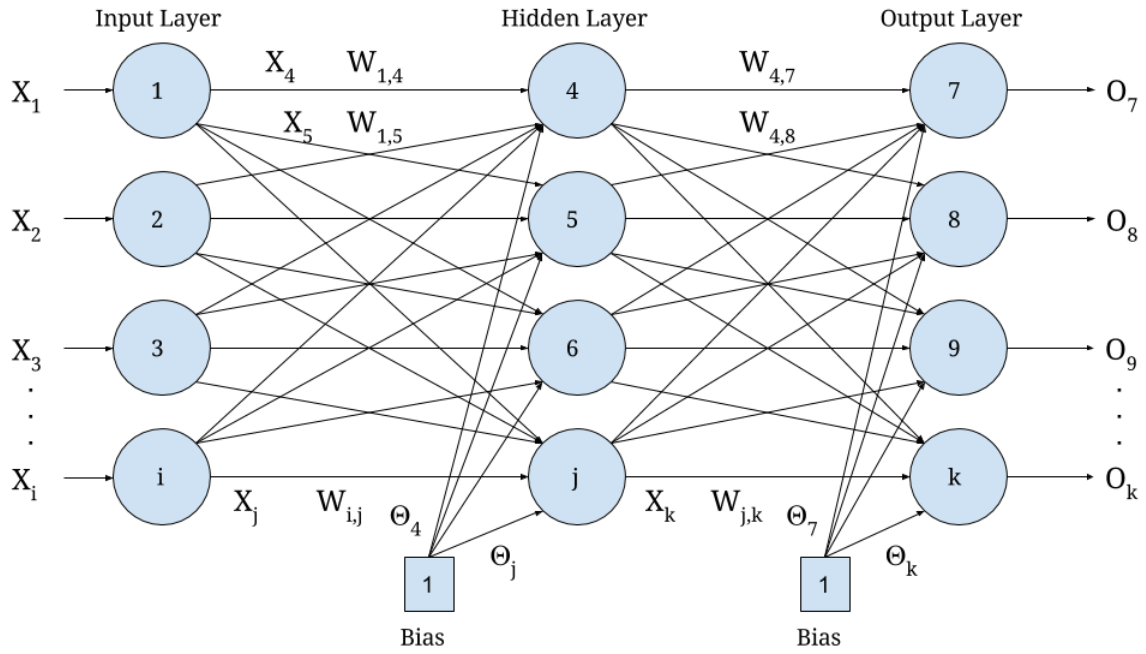

Figure 1.2: A single-hidden layer MLP neural network. Each neuron in each layer has its own activation function. Each layer of neurons will typically have the same activation function. Each neuron in each layer is connected to each of the preceding and following layer neurons by directed, non-cyclic, weighted connections, making the MLP a fully-connected feedforward neural network. From the left, i real-valued inputs from the X_i vector are fed-forward to the input layer. After passing through the input layer neuron activation functions, j outputs included in the X_j vector are fed-forward to the first hidden layer through weighted connections. The weights for each of these connections are kept in the $W_{i,j}$ vector. A weighted bias term is also fed to the hidden layer per each neuron. This term helps the network handle zero-input cases. The hidden layer has j neurons. k outputs from the hidden layer neurons, stored in the X_k vector, are fed-forward through weighted connections to the output layer. The weights for these connections are stored in the $W_{j,k}$ vector. The output layer is also fed a weighted bias term. The output layer has k neurons. The outputs from these neurons, stored in the O vector, are the final outputs generated by the network.

1.5.3 Activation Functions

There are several neuronal activation functions that can be applied in a neural network. Most of these activation functions are non-linear in order to model a target response that varies non-linearly with the explanatory input variables. In this sense, non-linear combinations of functions generate output that cannot be reproduced by a linear combination of the inputs. Some notable activation functions applied in this

work are described below:

- **Rectified Linear Unit (ReLU)**: The rectified linear unit activation function, where x is a real-valued input vector, is the following:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{1.1}$$

  Its derivative is:

$$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{1.2}$$

  It has a range of $[0, \infty)$ and is a monotonic function with a monotonic derivative.

- **Logistic (Soft Step)**: The logistic activation function, where x is a real-valued input vector, is the following:

$$f(x) = \frac{1}{1 + \exp^{-x}} \tag{1.3}$$

  Its derivative is:

$$f'(x) = f(x)(1 - f(x)) \tag{1.4}$$

  It has a range of $(0, 1)$ and is a monotonic function that does not have a monotonic derivative. It is, by characteristic, a sigmoidal function due to its S-shape.

- **Hyperbolic Tangent (tanh)**: The hyperbolic tangent function, also known as $tanh(x)$, where x is a real-valued input vector, is the following:

$$f(x) = tanh(x) = \frac{2}{1 + \exp^{-2x}} - 1 \tag{1.5}$$

Its derivative is:

$$f'(x) = 1 - f(x)^2 \tag{1.6}$$

It has a range of $(-1, 1)$, and it is a monotonic function that does not have a monotonic derivative. It is, by characteristic, a sigmoidal function due to its S-shape.

- **SoftMax**: The softmax activation function, where x is a vector of K real-valued numbers, is the following:

$$\sigma(x)_j = \frac{\exp^{x_j}}{\sum_{k=1}^{K} \exp^{x_k}} \quad \text{for } j = 1, ..., K. \tag{1.7}$$

It is a logistic function that squashes values from x into a categorical probability distribution vector, $\sigma(x)_j$, of K real-valued probabilities that each lie in the range (0,1) and sum to 1. It provides a probability distribution over K unique outcomes, typically necessary in classification problems.

1.5.4 Training with Backpropagation

In order to train feedforward neural networks, the backpropagation algorithm was developed. The backpropagation algorithm depends on a cost or loss function that maps values from one or more variables to a cost associated with a given event such as a network prediction. The loss function should compute the difference between the results of propagating an input example through a neural network and the expected output for that example. There are two assumptions about the loss function utilized in backpropagation. First, the function must be able to be written as an average over the errors for all individual training examples in a given set or batch. Second, the cost function must be able to be written as a function of the outputs of the neural network [Rojas, 1996].

In theory, the algorithm has two phases: propagation and weight update. The phases are outlined below via [Rojas, 1996]:

- **Phase 1: Propagation**: A. Propagate training examples through the neural network to generate network outputs for each example. B. Propagate the network outputs for each example backward through the neural network and use the target output for each example to generate the deltas (differences between output and target) for all layer neurons including output and hidden layers.
- **Phase 2: Weight update**: The following two steps must be followed for each and every weight in the network: A. The output delta is multiplied by the input activation of the weight to find the weight gradient. B. The weight is decremented by a fraction of the weight gradient.

These phases are repeated until the network error is tolerable. In phase 2, the fraction of weight gradient subtracted affects the quality and speed of the learning algorithm. The learning rate is the fraction of weight gradient subtracted from a given weight. The higher the fraction, the higher the learning rate. A general approach is to increase the learning rate until the errors diverge, but more analytical procedures have been developed [Werbos, 1990]. As the learning rate increases, so does the rate at which the neuron learns, but there is an inverse relationship between learning rate and learning quality [Thimm et al., 1996]. As the learning rate increases, the learning quality decreases. The sign of the weight gradient determines how the error varies with respect to the weight. The weight must be updated in the opposite direction in order to minimize the error in the neuron [Werbos, 1990].

Gradient Descent

Gradient descent is an optimization method for computing the minimization of an objective function written as a sum of differentiable functions. The problem of objec-

tive function minimization is frequently posed in machine learning and often results in a summation model such as:

$$O(\theta) = \frac{1}{n} \sum_{i=1}^{n} O_i(\theta) \tag{1.8}$$

where n is the number of observations in vector O, θ is an estimated parameter to minimize $O(\theta)$, and the i-th observation from the set or batch used in training is associated with O_i [Bottou, 2010]. Batch gradient descent performs the following operation over multiple epochs until error minimization in order to minimize the objective function in equation 1.8, where η is the step size [Bottou, 2010], [Ruder, 2016]:

$$\theta := \theta - \eta \sum_{i=1}^{n} \frac{\nabla O_i(\theta)}{n} \tag{1.9}$$

Stochastic gradient descent (SGD) is a technique based on the gradient descent optimization method for stochastically approximating the minimization of an objective function formed by a sum of differentiable functions. The iterative SGD algorithm computes updates on a per-example basis by approximating the gradient of $O(\theta)$ using one example as follows [Bottou, 2010], [Ruder, 2016]:

$$\theta := \theta - \eta \nabla O_i(\theta) \tag{1.10}$$

This update is performed over all examples. After all example updates, the training epoch is complete. Upon the start of a new epoch, shuffling of the training examples may occur to prevent repetition, and the process is repeated until the optimal weights are found.

Calculation of the gradient against a "mini-batch" of examples upon each step leads to a balance between computing the gradient per a single example as in SGD and computing the gradient across a large batch. Vectorization libraries tend to

optimize this method based on the vectorized batch update process. Essentially, the mini-batch gradient descent algorithm is equation 1.9 computed on batches of smaller sizes, b, than the total training dataset size of n [Ruder, 2016]. For example, if there were $n = 50$ examples in the training set and a mini-batch size of $b = 5$ was used for training, then there would be 10 mini-batch training updates to the weight vector per each training epoch for the amount of epochs needed until error minimization.

AdaGrad

The AdaGrad algorithm is an adaptive subgradient method used for gradient-based optimization by adapting the learning rate to the parameters [Duchi et al., 2011]. It adapts the learning rate for each parameter individually through division by the L_2-norm of all previous gradients and thus, it applies a different learning rate for every parameter at every time step [Nervana Systems, 2017b]. AdaGrad performs smaller updates for frequent parameters and larger updates for infrequent parameters [Ruder, 2016]. It is appropriate for general training in addition to "needle-in-the-haystack" problems that have examples with sparse distributions, as it adapts to the geometry of the error surface [Nervana Systems, 2017b].

The AdaGrad algorithm, according to [Duchi et al., 2011] and [Nervana Systems, 2017b], is the following:

$$\begin{aligned} G' &= G + (\nabla J)^2 \\ \theta' &= \theta - \frac{\alpha}{\sqrt{G' + \epsilon}} \nabla J \end{aligned} \tag{1.11}$$

where G is the accumulating norm vector, ∇J is the gradient vector, θ is the parameter vector, α is the learning rate, and ϵ is the smoothing factor to prevent from dividing by zero. A notably resilient learning rate for the AdaGrad algorithm used in many implementations is 0.01 [Ruder, 2016], [Nervana Systems, 2017b].

RMSProp

Rprop is the resilient backpropagation algorithm for performing supervised batch learning in neural networks [Riedmiller, 1994]. The goal of rprop is to eliminate the negative influence of the partial derivative size on the weight step in the normal backpropagation algorithm. In rprop, each weight has an evolving update-value, δ_{ij}, which is applied in a direction based on the sign of the gradient. The size of the weight update is solely based on the weight-specific update-value [Riedmiller, 1994]. Rprop can be applied in a mini-batch manner, but the problem with this is that a different number is used as the divisor for each mini-batch. Rprop used in this fashion causes the weights to largely grow. In short, rprop is equivalent to using the gradient but also considers the size of the gradient by division [Tieleman and Hinton, 2012].

Root mean-squared error propagation (RMSProp) is a mini-batch version of rprop that tracks a moving average of the squared gradient per each weight. The RMSProp algorithm is the following:

$$\begin{aligned} r_t &= (1-\gamma) f'(\theta_t)^2 + \gamma r_{t-1}, \\ v_{t+1} &= \frac{\alpha}{\sqrt{r_t}} f'(\theta_t), \\ \theta_{t+1} &= \theta_t - v_{t+1} \end{aligned} \tag{1.12}$$

where r_t is the mean-squared error at time t, θ_t is the vector of parameters at time t, $f'(\theta_t)$ is the derivative of the loss with respect to the parameters at time t, α is the step or learning rate, and γ is the decay term [Technische Universitaet Muenchen, 2013], [Tieleman and Hinton, 2012]. RMSProp can deal with stochastic objectives nicely, making it useful for mini-batch learning and training recurrent neural networks to avoid exploding or vanishing gradients [Technische Universitaet Muenchen, 2013].

1.5.5 Recurrent Neural Networks

A RNN is an ANN where information flows in many directions and has at least one feed-back connection between neurons to form a directed cycle. This type of network demonstrates dynamic temporal behavior because it forms an internal state of memory useful for learning time-dependent sequences. One of the simplest forms of recurrent neural networks is a MLP with outputs from one hidden layer feeding back into the same hidden layer over a discretized time scale set by a delay unit. A simple RNN, recognized as the Elman RNN proposed in [Elman, 1990], is pictured in figure 1.3. The weights along the edges from hidden units to context units are held constant at 1, while the context unit outputs feed back into the same hidden units along standard, weighted edges. Fixed-weight recurrent edges laid the foundation for future recurrent networks such as the LSTM [Lipton et al., 2015]. Additionally, it has been shown that RNNs outperform MLPs at time-series forecasting of stock market outcomes [Oancea and Ciucu, 2014]. It was shown that the RNNs were able to provide significantly greater predictive power then MLPs in the time-series realm given their ability to model temporal context.

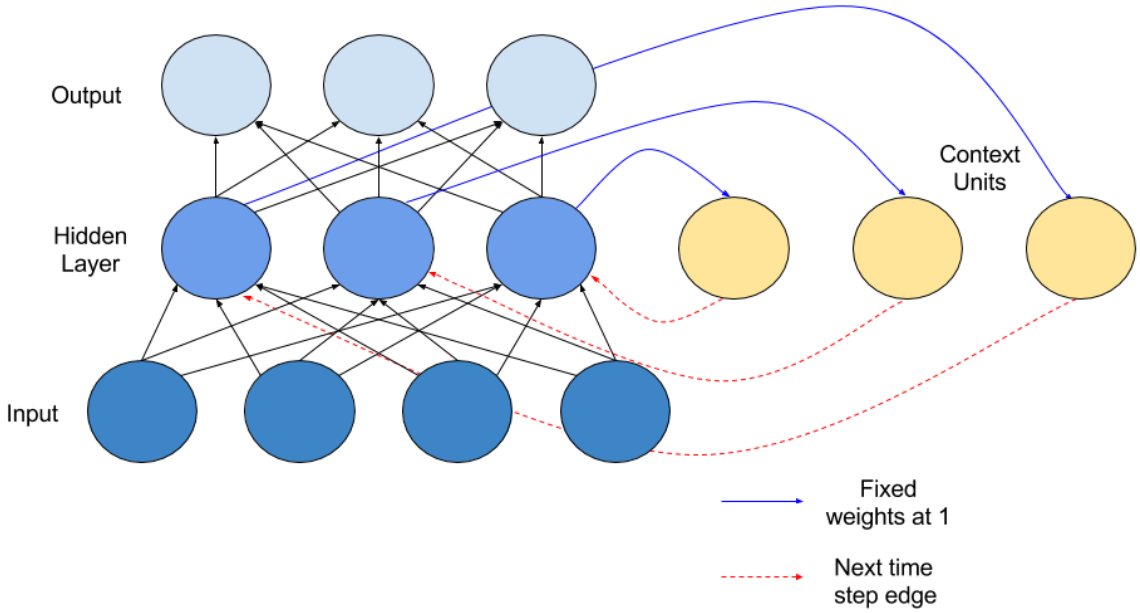

Figure 1.3: A simple Elman recurrent neural network with a single hidden layer as proposed in [Elman, 1990] and reviewed in [Lipton et al., 2015]. The hidden units feed into context units, which feed back into the hidden units in the following time step.

Despite the usefulness of simple RNN units, they tend to utilize typical activation functions that have trouble capturing long-term dependencies due to vanishing or exploding gradients. These gradients tend to mask the importance of long-term dependencies with exponentially larger short-term dependencies. Two notable approaches have been applied to handle these gradient problems which include gradient clipping, where the norm of the gradient vector is clipped, and second-order derivative methods where the second derivative is assumed less susceptible to the gradient problems than the first derivative [Chung et al., 2014]. Unfortunately, these solutions are not generally applicable to all learning problems and have their own flaws.

The activation functions of the simple RNN units are not as robust for modeling long-term dependencies as other methods which track important occurrences over time. The LSTM introduced an intermediate storage to simple RNN units by the addition of a memory cell. The memory cell of the LSTM was designed to maintain a state of long and short term dependencies using gated logic. The memory

cell was built from simpler cells into a composite unit including gate and multiplicative nodes. The elements of the LSTM are described below [Lipton et al., 2015], [Gers et al., 2000]:

- **Input Node**: This unit is labeled as g_c. It takes the activation from the input layer $x_{(t)}$ at the current time step in the standard way and along recurrent edges from the hidden layer at previous time step $h_{(t-1)}$. A squashing function is applied to the summed weight input.

- **Input Gate**: A gate is a sigmoidal unit that takes activation from the current data point $x_{(t)}$ and hidden layer at the previous time step. The gate value is multiplied with another node's value. The value of the gate determines the flow of information and its continuation. For instance, if the gate value is 0, the flow from another node is discontinued whereas if the gate value is 1, the flow is continually passed. The input node value is multiplied by the input gate, i_c.

- **Internal State**: Each memory cell has a s_c node with linear activation, which is called the internal state by the original authors. This state has a recurrent, self-connected edge with a fixed weight of 1. Due to the constant weight, the error flows across multiple time steps without exploding or vanishing. The update for the internal state is outlined in figure 1.4, which also includes influence from the forget gate, f_c.

- **Forget Gate**: These gates were introduced by [Gers et al., 2000] and provide the network ability to flush the internal state contents when necessary. These are useful for networks that continuously run. The update equation for the internal state in figure 1.4 shows the influence of the forget gate in the update of the internal state.

- **Output Gate**: The output gate, o_c, multiplies the internal state value, s_c, to generate the output value, v_c. Sometimes, the value of the internal state

is run through a non-linear activation function to give the output values the same range as ordinary hidden units with squashing functions, but this may be omitted depending on the scenario.

A LSTM unit with a forget gate in the style described in [Gers et al., 2000] is presented in figure 1.4. LSTM-based neural networks have been shown to significantly outperform feedforward neural networks for speech recognition tasks [Sundermeyer et al., 2013]. In [Sundermeyer et al., 2013], the LSTM network was able to achieve perplexities similar to those of the state-of-the-art speech recognition model, even though the LSTM was trained on much less data. Multiple LSTMs used in combination yielded even better performance on the test data, however, it was found that standard speech recognition technology limited the potential capabilities of the LSTM given the limits of preexisting speech recognition models.

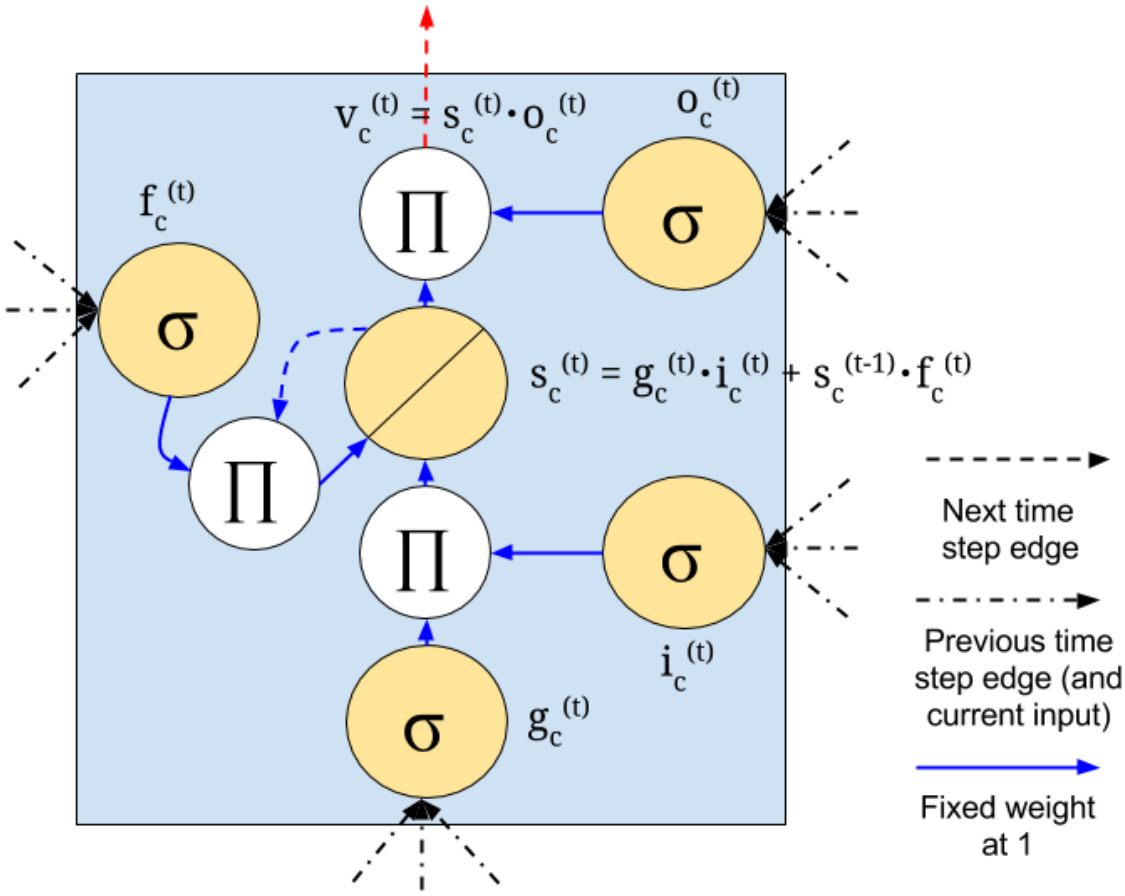

Figure 1.4: A LSTM unit with a forget gate as proposed in [Gers et al., 2000] and reviewed in [Lipton et al., 2015].

A recurrent unit simpler than the LSTM, called the GRU, was proposed by [Cho et al., 2014]. This unit, like the LSTM, modulates the flow of information inside the unit but is simpler because it does not have separate memory cells. The GRU, unlike the LSTM, exposes the whole state at each time step because it does not have a way to control the degree of the state that is exposed. Both the LSTM and GRU add to their internal content between updates from one time step to another, where the RNN unit typically replaces the previous activation with a new one.

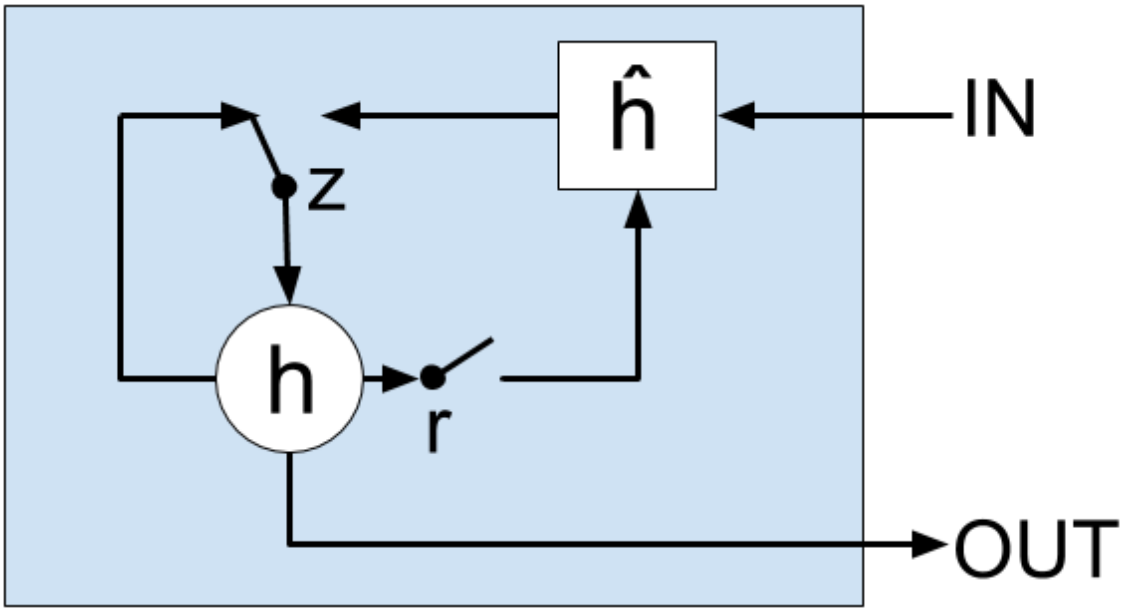

Figure 1.5: The GRU cell proposed in [Chung et al., 2014]. In this diagram, r is the reset gate, z is the update gate, h is the activation, and $\hat{h}$ is the candidate activation.

In the GRU, the activation h_t^j of the GRU at time t is a linear interpolation between the previous activation h_{t-1}^j and the candidate activation $\hat{h}_t^j$, where j is the current unit. The update gate, z_t^j, determines how much the unit activation is updated. The candidate activation computation is similar to that of traditional RNN units, which entails the reset gate to provide the ability to forget the previous state computation like the LSTM. The reset gate computation, r_t^j, is similar to that of the update gate, which both utilize linear sums and element-wise multiplication.

1.5.6 Training with Backpropagation through time

RNNs are harder to train than feedforward neural networks due to their cyclic nature. In order to deal with the complex, cyclic nature of such a network, it must first be unfolded through time into a deep feedforward neural network [Lipton et al., 2015]. The deep unfolded network has k instances of a feedforward network called f [Werbos, 1990]. After unfolding, the training proceeds in a manner similar to backpropagation as described in section 1.5.4. One difference is that the training patterns are visited in

sequential order. The input actions over k time steps are necessary to include in the training patterns because the unfolded network requires inputs at each of the k unfolded levels. After a given training pattern is presented to the unfolded network, weight updates in each instance of f are summed together and applied to each instance of f [Werbos, 1990]. The training is usually initialized with a zero-vector. In theory, the algorithm has eight steps, which are listed below via [Werbos, 1990]:

1. Unfold the recurrent network into k instances of f.
2. Initialize the current input context to the zero-vector.
3. Propagate inputs forward over the entire unfolded network.
4. Compute the error between the predicted and target outputs.
5. Propagate the error backward over the entire unfolded network.
6. Throughout all k instances of f, sum the weight changes.
7. Revise all weights in f.
8. Compute the next time step context.

Steps 2 through 8 are repeated until the network error is satisfactory.

1.6 Deep Learning

Deep learning is a large part of modern day intelligent applications. Deep learning involves the use of deep neural networks for the purpose of advanced machine learning techniques that leverage high performance computational architectures for training. These deep neural networks consist of many processing layers arranged to learn data representations with varying levels of abstraction [LeCun et al., 2015]. The more layers in the deep neural network, the more abstract learned representations become.

Previous machine learning techniques worked well on carefully engineered features extracted from the data using heavy preprocessing and data transformation. However, given the inherent variations in real-world data, these methods failed at being robust because they required a disentanglement and disregard for variations that were undesirable. Given the variability and unpredictable nature of real data, such methods of machine learning became unfeasible as we advanced into the future.

In order to solve the difficulties of learning the best representation to distinguish data samples, representation learning is applied. Representation learning allows a computer provided with raw data the ability to discover features useful for learning on its own. Deep learning is a form of representation learning that aims to express complicated data representations by using other, simpler representations [Ian Goodfellow and Courville, 2016]. Deep learning techniques are able to understand features using a composite of several layers each with unique mathematical transforms to generate abstract representations that better distinguish high-level features in the data for enhanced separation and understanding of true form.

A tremendous breakthrough allowing deep neural networks to train and evaluate on GPUs caused rapid growth into the research area. NVIDIA is a pioneer in GPU computing and has many graphics cards which can be used to enhance the deep learning experience. NVIDIA has a span of application programming interface (API)s to program their branded GPUs in order to handle mathematical operations as those involved in neural network computations in parallel. Due to the highly parallel and asynchronous architectures of GPUs, they can be applied to parallelize and speed up math operations in many diverse ways. NVIDIA CUDA is a parallel computing platform and API model created by NVIDIA for the purpose of CUDA-enabled GPU programming [Nvidia, 2017]. It was proven that network training and evaluation on the GPU is quicker and more power efficient than on CPU [Nvidia, 2015].

Many of the neural network layers described in sections 1.5.2 and 1.5.5 can be

applied in deep, multi-layer structures. Typically, more layers lead to more abstract features learned by the network. Not all combinations of layers or depth of layers work properly though, at least not without specific weight initialization, regularization, training parameters or gradient clipping techniques [Zaremba et al., 2014].

It has been proven that stacking multiple types of layers in a heterogeneous mixture can outperform a homogeneous mixture of layers [Plahl et al., 2013]. One of the main reasons that stacking layers can improve network performance is that recurrent layers generate features with temporal context while feedforward networks generate features with static context. When the recurrent, temporal features are provided to a feedforward, static learning layer, generated outcomes may have a greater certainty than when generated either by homogeneous mixtures of recurrent or feedforward layers. The reasoning is that recurrent networks are not always certain of outcomes given their greater complexity and multi-path cyclic network while feedforward networks are more certain of outcomes but have lower complexity and less predictive power. Combining these two types of layers, in example, can greatly increase the predictive power of the overall neural network.

Multi-stream deep neural networks were created to deal with multi-modal data. These networks have multiple input layers each with potentially different input formats and independent modules of multiple layers that run in parallel. After forward propagation of each stream network, the resultant features are merged into a single module with multiple additional layers combined for joint inference using all independently-generated features. Multi-stream neural networks are useful generating predictions from multi-modal data where each data stream is important to the overall joint inference generated by the network. Multi-stream approaches have been shown successful for multi-modal data fusion in [Yao et al., 2016], [Singh et al., 2016], [Wu et al., 2015], [Xing and Qiao, 2016], [Ngiam et al., 2011], and [Song et al., 2016b].

Overall, deep neural networks have been applied successfully in multiple applica-

tions such as neural machine translation in [Bahdanau et al., 2014] and [Cho et al., 2014], time-series sensor data fusion [Yao et al., 2016], handwriting recognition [Xing and Qiao, 2016], speech recognition [Graves et al., 2013], sequence labeling [Graves, 2012], multiple kernel fusion [Song et al., 2016a], egocentric activity recognition [Song et al., 2016b], video classification [Wu et al., 2015], action detection [Singh et al., 2016], and multimodal deep learning between noisy and accurate streams [Ngiam et al., 2011].

1.7 Network Tuning and Evaluation Techniques

A deep neural network weight initialization scheme was developed that brings quicker convergence for training called GlorotUniform weight initialization [Glorot and Bengio, 2010]. This initialization procedure takes into account the variance of the inputs to determine an optimal uniformly-distributed weight initialization strategy for the provided training examples. This weight initialization scheme takes much of the guesswork out of network initialization.

The optimal number of hidden nodes for feedforward networks is usually between the size of the input and size of the output layers. Deep feedforward and recurrent neural networks may be hard to train. Feedforward networks tend to over-fit on training data and perform poorly on held-out test datasets [Hinton et al., 2012]. Recurrent networks in deep configurations tend to have the problems of exploding or vanishing gradient. These problems have to do with the inability of the backpropagation through time (BPTT) algorithm to properly derive the error gradient for training the weights between unit connections in the RNNs. One way to solve this problem is with dropout [Zaremba et al., 2014]. Dropout is a technique for reducing network overfitting by randomly omitting a percentage of the feature detectors per each training case [Hinton et al., 2012]. In the case of feedforward neural networks, the feature detectors would be connections to random neurons while in the recur-

rent case they would be random connections to either the same unit or another unit. Dropout can be applied in many cases to regularize the network generalization performance. It has been proven effective in [Hinton et al., 2012], [Srivastava, 2013], and [Srivastava et al., 2014].

Bootstrapping is a comprehensive approach for modification of the prediction objective in order to consistently manage random fluctuations in and absence of accurate labeling [Reed et al., 2014]. Many deep neural networks are trained with supervised learning where datasets are labeled and the networks learn to map the inputs to the labeled outputs. However, in many cases, not all possible examples are present in the dataset and many labels may be missing. Thus, the labeling for each dataset is subjective. Bootstrapping is a way to determine if a given network architecture, specifically a deep network, is able to generalize on a variety of data sampled from a sample distribution. This method can be applied for enhanced training and regularization by sampling random training examples with replacement from a large dataset useful for modeling the population distribution of the overall data. This allows the network to approximately learn the population data distribution without over-fitting on specific examples.

1.8 Regression Metrics

Metrics used to evaluate the predictive performance of regression-oriented machine learning models must compare a vector of target outputs to a vector of predicted outputs generated by a trained model. These metrics must consider the error of all pairs of outputs between the target and predicted vectors in order to provide a useful measure of the amount of error or success in the trained model. Three very useful metrics for measuring the regression error between target and predicted output vectors are residual sum of squares (RSS), mean-squared error (MSE) and root mean-

squared error (RMSE). In these error metrics, the smallest possible value is 0 and it happens when the predicted values exactly match the target values. The largest possible value is unbounded and depends greatly on the values being compared.

The single sample RSS metric for computing the residual sum of squares in a regression prediction is the following:

$$\text{RSS}(Y, \hat{Y}) = \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2 \tag{1.13}$$

where Y is 1-dimensional vector of N target values and $\hat{Y}$ is a 1-dimensional vector of N predicted values. Y and $\hat{Y}$ must have the same number of values.

In most cases dealing with large amounts of data, the target and predicted vectors will be stored in matrix representation to simplify representation and vectorize mathematical calculations. In this representation, the number of rows in the matrix are the number of samples and the number of columns are the number of outputs per sample. The MSE and RMSE are two metrics that can easily be vectorized using the matrix representation.

The vectorized MSE metric is the following:

$$\text{MSE}(Y, \hat{Y}) = \frac{\sum_{i=1}^{M} \left(\sum_{j=1}^{N} (Y_{i,j} - \hat{Y}_{i,j})^2 \right)}{MN} \tag{1.14}$$

where Y is 2-dimensional matrix of the target values, $\hat{Y}$ is a 2-dimensional matrix of predicted values, M is the number of samples and N is the number of outputs per sample. Both Y and $\hat{Y}$ must have M rows and N columns.

The vectorized RMSE error metric is the following:

$$\text{RMSE}(Y, \hat{Y}) = \sqrt[2]{\text{MSE}(Y, \hat{Y})} \tag{1.15}$$

where Y is 2-dimensional matrix of the target values, $\hat{Y}$ is a 2-dimensional matrix of predicted values. Both Y and $\hat{Y}$ must have the same dimensions as in equation 1.14.

A useful regression metric for measuring how well a model is likely to predict future samples is called the r^2-metric. The range for this function is [0, 1]. When it outputs 1, the dependent variable can be predicted without error from the independent variable. When it outputs 0, the dependent variable cannot be predicted from the independent variable. A value in its range indicates the extent of predictability between the variables. The r^2-metric is the following:

$$R^2(Y, \hat{Y}) = \frac{\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{N}(Y_i - \bar{Y})^2} \tag{1.16}$$

where N is the number of samples, Y is the target vector, $\hat{Y}$ is the predicted vector, and $\bar{Y}$ is the following:

$$\bar{Y} = \frac{\sum_{i=1}^{N} Y_i}{N} \tag{1.17}$$

1.9 Classification Metrics

Metrics used to evaluate the predictive performance of classification-oriented machine learning models must compare a target output class to a vector of output class confidences the length of the number of trained classes as generated by the trained classification model. In most cases, the classes of specific data samples are represented by integers starting from 0 and increasing to the number of classes minus one, as if indexed in a list. When the output confidence for a specific class is higher than all

other output confidences for a given input sample, then the model predicts that the input sample belongs to that class. Typically, these confidence vectors are converted into what is called a binary one-hot representation, where the list index for the class with the highest confidence is converted to a 1 while all other classes are converted to 0s.

An often used metric for evaluating the error of a binary classifier is the binary cross-entropy error. The binary cross-entropy cost function is the following:

$$C = \sum -tlog(y) - (1-t)log(1-y) \tag{1.18}$$

where t is the target class label and y is the network output [Nervana Systems, 2017c]. When this cost metric is applied on a neural network with a SoftMax classifier for the output layer, backpropagation (BP) training takes advantage of a shortcut in the derivative that saves computation.

Another valuable metric for classification is accuracy. Accuracy is the number of correct predictions made over the total number of predictions made as shown below:

$$ACC = \frac{TP + TN}{P + N} \tag{1.19}$$

where TP is the number of true positive predictions, TN is the number of true negative predictions, P is the total number of positive predictions and N is the total number of negative predictions [Asch, 2013].

Although this metric is a good indicator of how well a classifier performs in early stages, it is not necessarily an accurate metric when the number of samples per class are imbalanced. It can lead to a false representation of the model performance on an imbalanced dataset. Misclassification is another way to view the classification accuracy of the model, as it is $1 - ACC$. This metric highlights the amount of incorrect predictions made by the model as it is the number of false predictions over

the total number of predictions made.

A better pair of metrics for evaluating the performance of a classifier are precision and recall. Both of these metrics focus on measuring the relevance of predictions. Precision, also known as the positive predictive value (PPV), is the fraction of instances retrieved that are relevant. Precision is represented numerically as the following:

$$PPV = \frac{TP}{TP + FP} \tag{1.20}$$

where TP is the number of true positive predictions and FP is the number of false positive predictions [Asch, 2013].

Recall, also known as the sensitivity or true positive rate (TPR), is the fraction of relevant instances retrieved. Recall is represented numerically as the following:

$$TPR = \frac{TP}{TP + FN} \tag{1.21}$$

where TP is the number of true positive predictions and FN is the number of false negative predictions [Asch, 2013]. Both of these metrics do not necessarily correspond which each other. For example, while recall may decrease, precision may not. Typically for cross-validation in the binary classification task, both of these metrics are combined in a plot where the precision is the independent variable and the recall is the dependent variable. When the curve is completely horizontal, the classifier has perfect precision and recall. An individual curve can be plotted for each class to show the predictive performance on a per-class basis. These quantities may also be combined to compute the F1 score, which is the harmonic mean of precision and recall. This score may not necessarily lie between the precision and recall scores. The F1 score is the following:

$$F1 = \frac{2 * P * R}{P + R} \tag{1.22}$$

where P is the precision and R is the recall [Asch, 2013]. Another way to visually evaluate the performance of a classifier is through a receiver operating characteristic (ROC) curve. This curve is created by plotting the true positive rate as the independent variable and the false positive rate as the dependent variable. The ROC curve is thus the sensitivity of prediction as a function of false alarm rate [Fawcett, 2006]. The true positive rate is synonymous to the recall shown in equation 1.21 while the false positive rate (FPR) is the rate of false alarm. The FPR is shown numerically below:

$$FPR = \frac{FP}{FP + TN} \tag{1.23}$$

where FP is the number of false positive predictions and TN is the number of true negative predictions. When the ROC curve is a perfect horizontal line, the classifier is perfect according to this metric. The area under the precision-recall curve or the ROC curve can be calculated. This area is referred to as the area under the curve (AUC) score. The larger the area, the better the classifier performance.

The scores can be averaged to determine a single number representative of the classifier performance. There are several ways to average the scores, a few of which are the following:

- **Micro-Average**: Given equal weight to each per-sample classification decision. Large classes dominate smaller classes in this averaging method.
- **Macro-Average**: Give equal weight to each class. Classifier effectiveness on smaller classes can be achieved using this averaging method.

According to [Asch, 2013], each of these scores has its own pitfalls based on label imbalance. However, many of the averaging methods perform similarly when labels are balanced. Thus, it is a good idea to achieve balanced datasets when possible.

Chapter 2

Methods

2.1 Sensor Data

Time-series sensor data was collected from multiple vehicles with forward-facing sensor systems as provided by supplier A. The systems consisted of radar and camera mounted on the front of vehicles. Each sensor was able to track up to 10 objects in front of the vehicle at once. Per each sensor, the important objects were provided with their own track of multiple informative data streams yielding properties observed by the sensor. Synchronized sensor data was collected from vehicles at a 40 millisecond (ms) sample rate while driving along various roadways with varying light, weather and traffic conditions.

The data used was sampled from week-long spans of synchronized multi-track sensor data acquired from both sensors. The sampled data captured a multitude of scenarios useful for data analysis, training and evaluation of a robust sensor fusion system involving deep learning. The fusion methodology presented was developed in order to fuse synchronized pairs of the acquired types of radar and camera sensor track data based on a preexisting state-of-the-art sensor fusion system with demonstrated success on similar data.

2.2 Camera

The camera was provided by vendor A. The camera could track up to ten detectable vehicles at once within its field-of-view. These vehicles were each arbitrarily assigned to one of ten object tracks. Unique vehicle identification (ID) labels were provided by the camera in order to identify each tracked vehicle at any given time. The camera was noisy in its approximation of various attributes like longitudinal distance and relative velocity with respect to the mounted observer vehicle. Two notable camera advantages were its wide field-of-view and unique vehicle recognition and tracking abilities. On the other hand, two disadvantages of the camera were its noisy approximation of depth and susceptibility to noise from varying conditions. Figure 2.1 shows the camera field-of-view with respect to the observer vehicle and other moving vehicles.

2.3 Radar

The radar was provided by vendor B. The radar could track up to ten targets at once within its field-of-view. A target is synonymous to a detected object in front of the observer vehicle. The radar was able to provide the same information as the camera along with more accurate tracking of distance and relative velocity with respect to the observer vehicle. Unlike the camera, however, targets tracked by the radar were arbitrarily identified and did not necessarily represent vehicles on the road. There was no unique ID assigned to any one target at any time. Targets would abruptly be re-assigned to new tracks over time. This arbitrary target track re-assignment made it difficult to determine fusion between both sensors for a long period of time. For this reason, the radar data required preprocessing in order to uniquely segment targets with the same consistent properties over time. Two notable radar advantages were its deeper depth-of-field for target detection and tracking as well as smooth signals. Two

disadvantages of the radar were its narrow field-of-view and spontaneous target track re-assignment. Figure 2.1 shows the radar field-of-view with respect to the observer vehicle and other moving vehicles.

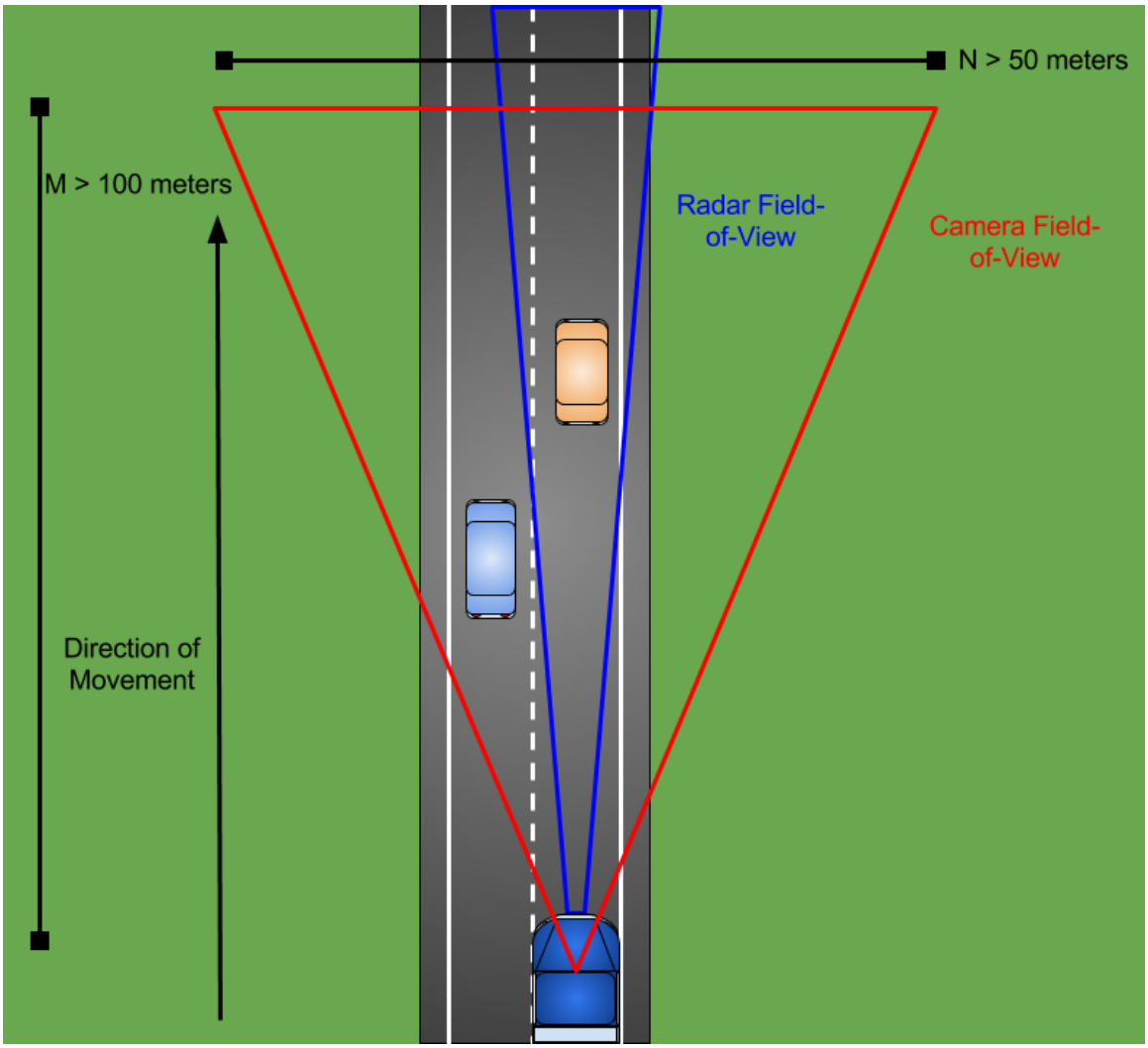

Figure 2.1: A 2-dimensional overview of an observer vehicle following two vehicles on a 2-lane roadway. All vehicles are moving forward. The figure demonstrates the field-of-view and depth-of-field of the camera and radar sensors. The camera field-of-view, shown in red, is wider than the radar but its depth-of-field is shallower. The radar field-of-view, shown in blue, is narrower than that of the camera but its depth-of-field is deeper.

2.4 Utilized Sensor Streams

By visual analysis with plotting tools, it was determined that four specific streams of data per each object track would be useful to fuse between the sensors. While addressing the sensor fusion problem, these four streams were found reliable and consistent between sensors when overlap occurred without object re-assignment. Given the formulation of a machine learning solution to address sensor fusion between the two sensors, it was necessary to find correlated time-series data streams between any two selected pairs of object tracks in order to train and evaluate the fusion models constructed. Such data would have to be separable in order to distinguish between fused and non-fused examples. The four chosen streams that seemed to best correlate visually were the following:

- **Longitudinal (Range) Distance**: This data represented the distance of the observer vehicle from each detected object on the road measured with respect to the longitudinal axis of the observer vehicle. The units were meters.
- **Lateral Position**: This data represented the lateral position of the object relative to the lateral axis of the observer vehicle. Left of the vehicle was a distance increase and right of the vehicle was a distance decrease, where the center of the vehicle was the origin. This data was equivalent, but not limited to, tracking the lane occupied by the object in focus according to the observer vehicle. The units were meters.
- **Relative Velocity**: This data represented the velocity of the tracked object relative to the observer vehicle. The units were meters per second.
- **Width**: This data represented the width of the object tracked by the observer vehicle. The units were meters.

The radar already provided the chosen data streams. The camera data required

preprocessing to compute the lateral position based on azimuth angles and longitudinal distance. The units of the azimuth angles were in radians per second. The following formula was used to compute the lateral position of each vehicle with respect to the observer vehicle based on the azimuth angles and longitudinal distance:

$$0.5 * dist * tan(min(angles)) + tan(max(angles)) \tag{2.1}$$

where *tan* was the tangent function, *min* was the minimum value function of a collection, *max* was the maximum value function of a collection, *dist* was the longitudinal distance in meters of the tracked object from the vehicle and *angles* were the left and right azimuth angles of the tracked object with respect to the center of the mounted camera in radians per second. Figure 2.2 shows the measurements provided by each of the four chosen sensor streams.

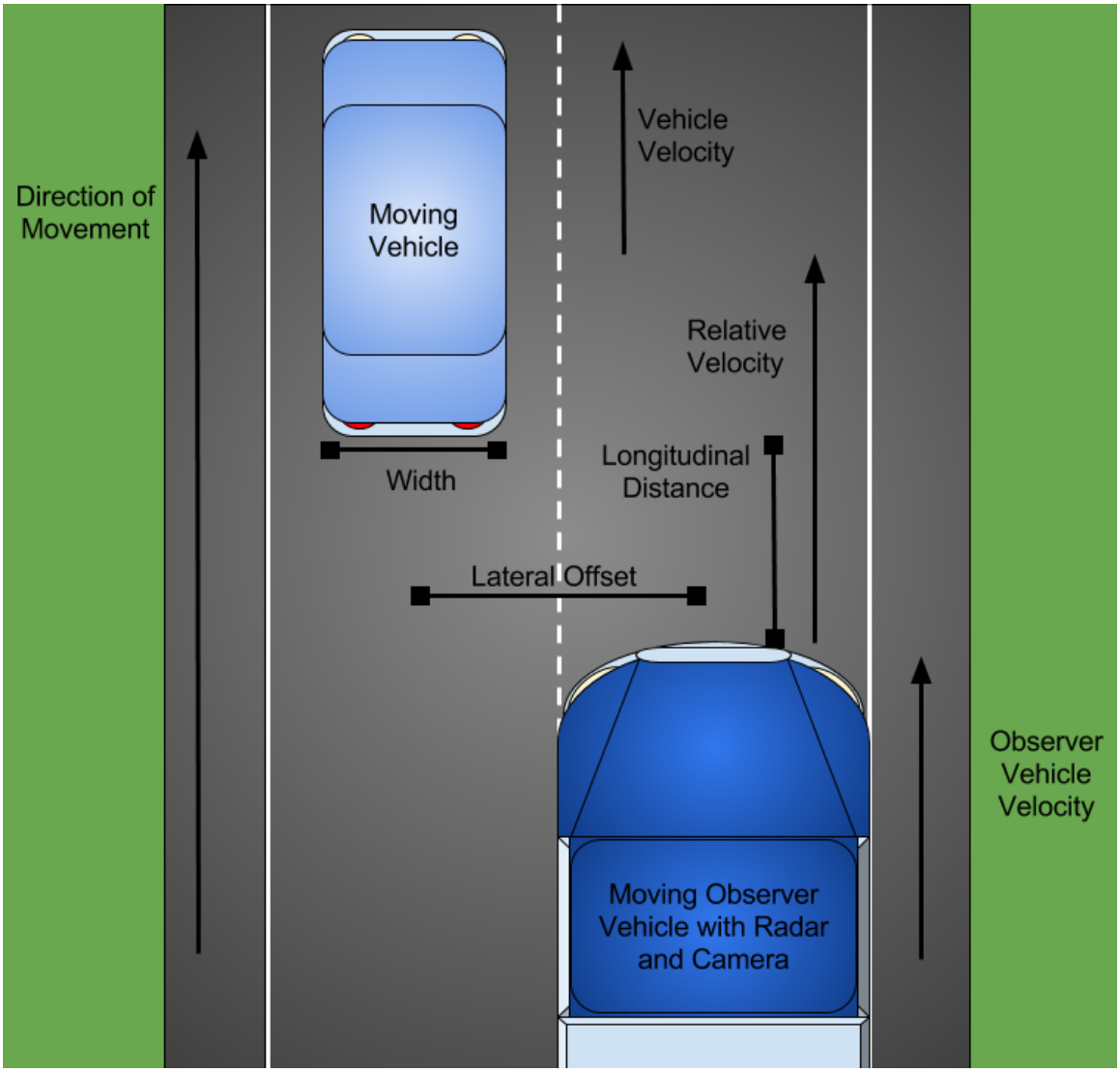

Figure 2.2: A 2-dimensional overview of an observer vehicle following another vehicle on a 2-lane roadway. The measurements represented by each of the four chosen sensor streams are outlined.

2.5 Problem Formulation

2.5.1 Correlation of Consistent Sensor Tracks

In order to determine the strength of correlation between the four chosen object track streams reconstructed from both sensors where fusion was implied, two metrics were computed including the coefficient of determination, referred to as r^2-metric,

and a normalized, discretized temporal cross-correlation using the product of the convolution where the two signal streams overlapped completely. The samples of sensor data selected for comparison were those fused based on the existing state-of-the-art fusion system discussed in section 2.6.4. The r^2-metric was utilized because it was a direct and general way to compare the object track data samples from the disparate sensor sources. The normalized, discretized temporal cross-correlation formula was used to compute a more in-depth score between the synchronized time-series vectors of data samples from the two sensors based on displacement overlap.

The coefficient of determination formula used was covered in equation 1.16section of 1.8. The discretized temporal cross-correlation formula via [Rabiner and Gold, 1975] and [Rabiner and Schafer, 1978] was the following:

$$xcorr(f, g) = (f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m]g[m+n] \tag{2.2}$$

where f and g were aligned temporal sequences, f^* was the complex conjugate of f, m was the current point in time, and n was the displacement or time lag between data points being compared. There were several modes of this algorithm that could have been used depending on the type of convolution desired. It was chosen to use the convolution mode for cross-correlation only where the signal vectors overlapped completely.

In order to compute the normalized, discretized cross-correlation, the auto-correlation of each vector was taken using equation 2.2 by providing the same vector as both inputs. The selected stream vector was then divided by the square-root of the auto-correlation results for that vector to obtain the normalized auto-correlation per each vector. These normalized auto-correlation vectors were subsequently fed to equation 2.2 to compute the normalized cross correlation. Hence, the normalized cross-correlation between a given pair of reconstructed, fused streams, according to the

state-of-the-art fusion system, was computed according to the following equation:

$$X = xcorr(\frac{f}{\sqrt{xcorr(f,f)}}, \frac{g}{\sqrt{xcorr(g,g)}}) \tag{2.3}$$

where X is the normalized cross-correlation that uses the convolution product where the signals completely overlap, f is the first input stream vector and g is the second input stream vector, $xcorr$ is equation 2.2 and $sqrt$ is the vectorized square-root operation. This function had a range of [0, 1], where 1 was the highest possible correlation between any two vectors.

An additional similarity metric used to determine the error between fused sensor tracks was the Euclidean distance. The 1-dimensional Euclidean distance provided a multi-modal distribution of errors with physical units useful for understanding the state-of-the-art fusion system based on the distances between fused sensor streams. The vectorized 1-dimensional Euclidean distance formula is the following:

$$dist(X,Y) = |X - Y| \tag{2.4}$$

where X is one real-valued time-series sensor stream sample vector from radar and Y is another real-valued time-series sensor stream sample vector from camera. For proper comparison, both samples were time-synchronized and the streams compared were of the same type as listed in section 2.4. In this formula, the closest possible correlation was represented by zero.

Understanding the noise distribution and tracking error between the sensors with reference to the preexisting fusion system was necessary to develop alternative sensor fusion datasets. The absolute differences acquired over a variety of conditions were subsequently used to represent allowable error tolerances between the two sensors in a variety of generated datasets. Furthermore, it was conjectured that comparable sets of data streams from both sensors where the objects were consistent and close with

no sudden track or ID reassignments by either sensor would lead to highly correlated data useful for sensor fusion purposes. Figure 2.3 demonstrates an example of sensor fusion between a single radar track and camera object.

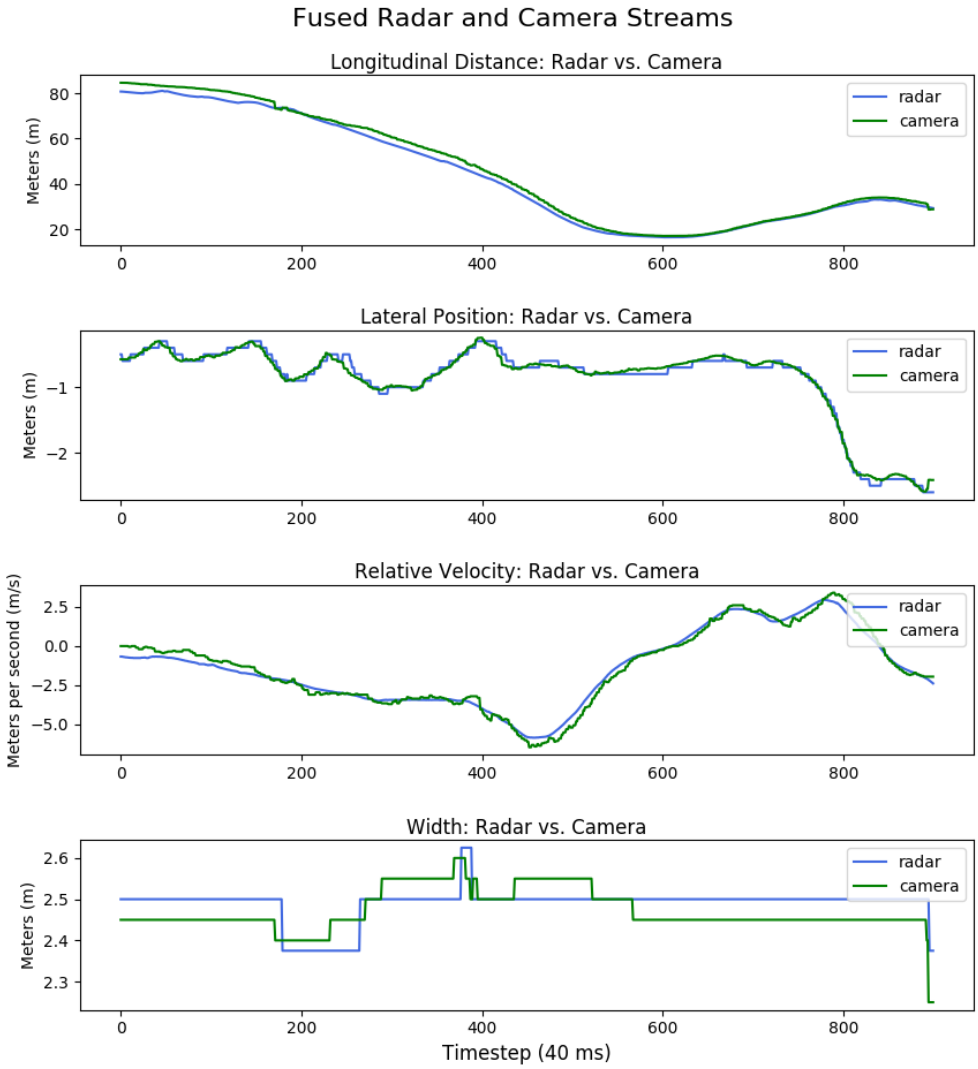

Figure 2.3: An example of consistent sensor fusion between a single radar track and camera object for approximately 36 seconds. The fusion is apparent through correlation of all four selected sensor streams.

2.5.2 Notion of Sensor Data Consistency

The notion of sensor data consistency was developed based on the inability to fuse object track data directly between both sensors due to spontaneous reassignment of

an object to a new track by either sensor. It was found that the radar tended to swap target information between two tracks over time or even reassign a target to a new track spontaneously. The camera, however, had less predictable track reassignment. Once object re-assignment took place on either fused sensor track, the fusion correlation between those particular tracks was lost. Figure 2.4 reveals the swapping of target information between two radar tracks over time. Such data swapping would break the correlation between fused sensor tracks without inherent knowledge of the swap.

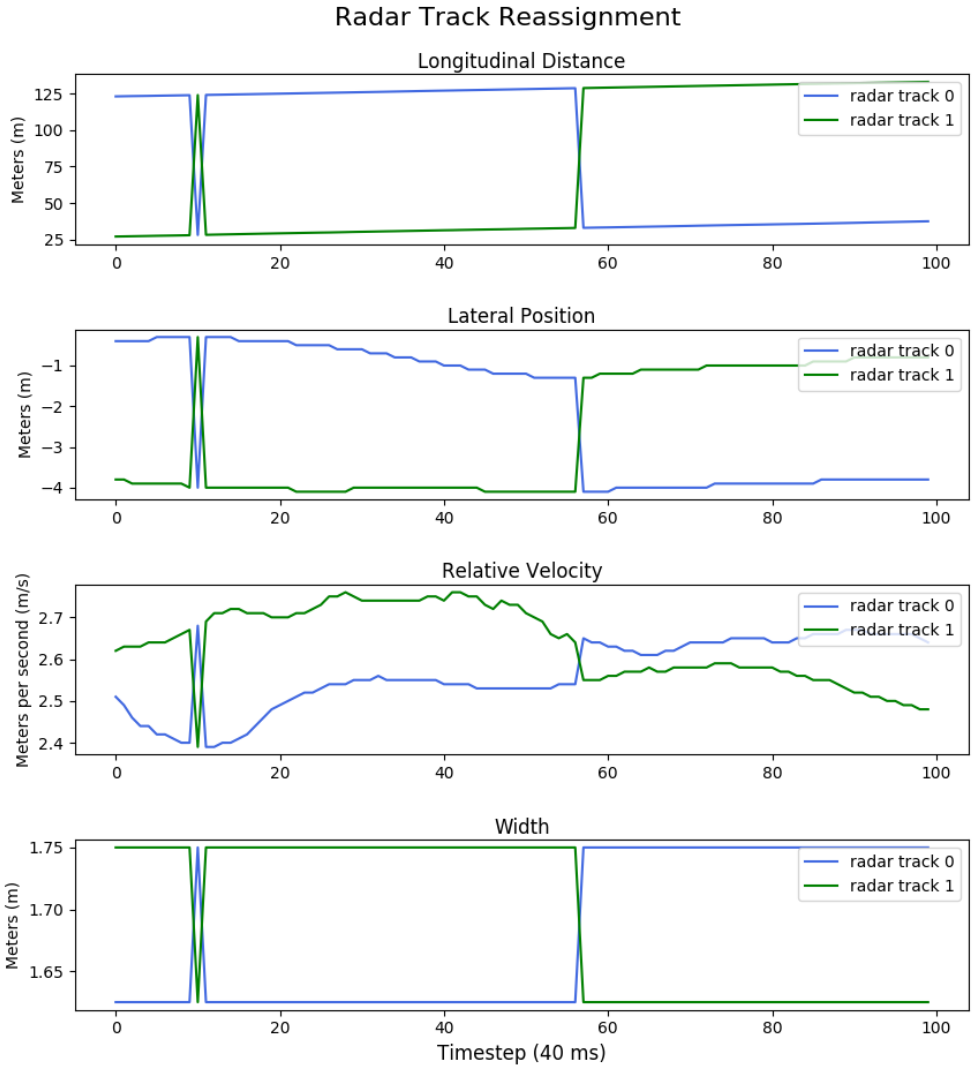

Figure 2.4: A demonstration of target information swapping between two radar tracks over a 4-second time period.

To solve this problem, it was determined that fusion would be considered only after the consistency of all object tracks per each sensor was marked per each time step in the time-series sample. Thus, if a given track held information about the same object consistently, it would be marked as consistent for each possible time step. If a given track started providing information about a new object or unknown data, that particular track would be marked as inconsistent for each time a drastic change occurred. Finally, if a given track began providing unknown data as in empty space, very noisy data, or zeros, then the track would be marked unknown for that amount of time. These three classifications of data consistency were necessary for building a robust fusion system.

2.5.3 Notion of Sensor Stream Comparison for Fusion

The state-of-the-art sensor fusion system was able to provide one means of labeling sensor fusion to create a dataset. Based on these labeled fusion occurrences, an error distribution was created for the absolute differences between each of the four sensor streams. Each sensor stream type had its own independent distribution of acceptable tracking error with respect to the other sensor. Maximum allowable error thresholds per each stream were considered when labeling fusion occurrences between any two sensor tracks for further generated sensor fusion datasets.

2.6 Data Consumption, Preprocessing and Labeling

2.6.1 Reading and Processing of Big Data

In order to experiment with the idea of fusion between two sensor sources and create a robust system, several gigabytes of data was acquired from real vehicles. This

amount of data was equivalent to several million samples of input sensor data. Given the fact that there were millions of data samples, a fast, robust programming language was necessary for data processing and analysis. Given the involvement of machine learning, it was inherently necessary to involve the GPU for model training and evaluation. The tools used had to be compatible with both CPU and GPU in this case.

There was also the idea of building a embeddable fusion system, so the language had to be compatible with multiple platforms and architectures such as Intel x86-64 CPU, NVIDIA GPU, and ARM 32-bit and 64-bit. Given the fact that the project had many aspects requiring rapid development across multiple areas, a general purpose language possessing the necessary traits was chosen. The choice was to use Python 2.7.12, given its versatility, easy-to-use interface, compatibility across multiple systems with multiple architectures and operating systems, breadth of available open-source libraries, and speed when properly implemented [Python Software Foundation, 2017]. Python has many open-source libraries in the scientific computing and machine learning realms that are maintained by industry experts. Python also has interfaces to other languages like C, for tremendous speedup. Cython is a python interface to C that provides enhanced performance for python scripts and libraries [Behnel et al., 2011].

The acquired database was read, stored and manipulated using pandas and numpy. Pandas is a python package that provides fast, flexible, and expressive data structures designed to facilitate work on data that is relational [McKinney, 2010]. Numpy is a fundamental library for scientific computing in python that provides vectorized, high-speed data structures and directly interfaces with cython [van der Walt et al., 2011]. While pandas made it easy to manipulate relational data with multiple named rows and columns, numpy made it easy to quickly manipulate matrices of numerical data for vectorized math operations. Many of the tools used relied heavily on numpy

due to its tremendous speedup over typical Python data structures. Optimized Basic Linear Algebra Subprograms (BLAS) libraries, like OpenBLAS, were also used to tremendously enhance the speed of applications by means of vectorized numpy functions.

In order efficiently compute metrics and statistics on the data, scipy and scikit-learn were used in addition to built-in functions from numpy. Scipy is an open-source library for scientific computing in Python that relies on numpy [Jones et al., 01]. It provides many functions for signal processing and data analysis. Scikit-learn is an open-source Python library that provides implementations for a series of data preprocessing, machine learning, cross-validation and visualization algorithms for unified data science applications [Pedregosa et al., 2011]. Scikit-learn also has a heavy dependence on numpy, like scipy. Most notably, scikit-learn provides evaluation metrics for regression and classification problems that are versatile, utilize scipy and numpy for high performance, and are tested by many involved in machine learning development.

Neon is Intel Nervana Systems' open-source deep learning framework for designing, training and evaluating deep neural networks [Nervana Systems, 2017a]. Neon has many of the important building blocks necessary to create feedforward, recurrent and convolutional neural networks or load a pre-trained deep neural network. Neon has a fast and robust implementation of most network layer types, but more importantly, it is easier to implement than the other machine learning frameworks in Python including TensorFlow and Theano. For machine learning purposes, it has been shown that the Intel Nervana Neon Deep Learning Framework is one of the fastest performers in Python alongside Theano [Bahrampour et al., 2015]. However, the abilities of fast prototyping and ease-of-use outweighed the slight performance gains available using Theano. Given the advantages of using Neon, it was the selected option for implementing and evaluating deep neural networks in this research.

Parallelization

In order to design algorithms with high performance for rapid prototyping, high productivity and greater scalability, it was necessary to parallelize multiple aspects of the data read/write, preprocessing, labeling, post-processing, machine learning, statistical analysis and deep learning fusion system scripts. In order to parallelize these aspects, multiple libraries were used and compared. The libraries that were used included the "threading," "multiprocessing," "joblib," and "PyCuda" libraries in Python. The "threading" library in Python provides multi-threading ability but has the limitation of the global interpreter lock (GIL). Thus, threading in Python is not nearly as fast or scalable as it would be in a language like C or C++, however, it was used for experimentation with runtimes in the deep learning fusion system.

To unlock the GIL in Python, multiprocess programming was used. The multiprocessing module in Python provided the ability to fork multiple processes and share memory between processes in addition to providing control for all child processes from the main parent process. The joblib module had an implementation of "embarrassingly parallel" utilities for parallelizing "for" loops and encapsulating threading and multiprocess modules from an abstract, straight-forward interface. The multiprocessing and joblib libraries also allowed the creation of process pools for reusing processes for multiple jobs to amortize costs. Multiprocessing was used to parallelize just about every aspect possible in all the mentioned portions of this project.

Parallelization for most script aspects took place on the CPU via multiprocessing or joblib. However, training and evaluation of neural networks often took place on the GPU via NVIDIA CUDA 8 through the PyCuda module. PyCuda is a python library designed for interfacing with the NVIDIA CUDA parallel computation API [Klockner, 2017]. NVIDIA CUDA 8 is a version of CUDA that supports the latest Pascal architecture as of 2017. CuBLAS is an implementation of BLAS for CUDA by NVIDIA that greatly enhances the speed of vectorized GPU functions during linear

algebra and batch operations, like dot-products across large matrices. It was also applied for faster neural network training and evaluation.

2.6.2 Preprocessing Data for Neural Networks

In typical neural networks, there are a fixed number of inputs provided. The neural networks used for this project consisted of a fixed number of inputs to simplify the overall problem. Given the 40 ms update rate for the sensor signals and a need for the number of inputs to represent a consistent sensor signal, it was determined that the use of the past 25 time steps of acquired data points would suffice in order to provide enough data to determine whether two tracked objects were fused. The chosen number of time steps was equivalent to using the past one second of sensor data in order to determine whether a pair of independent sensor tracks were fused between the two sensors at the current time step.

2.6.3 Labeling of Data and Dataset Creation for Consistency

Given the need to train a neural network for generating the consistency sequence of any given sample of four synchronized input stream samples for a given radar track, as discussed in section 2.4, it was determined that integer sequences would be the easiest to learn for the neural network. This determination was based on the idea that simplifying the separation of learned representations for the neural network would lead to better recognition outcomes in the sequence generation respect. The camera would also need the notion of consistency sequences, but those sequences would be generated based on the provided video IDs given the camera vehicle recognition algorithm.

Since three consistency categories were already necessary to best determine when fusion was applicable between any two sensor object tracks, these three categories were mapped to integers as in a classification problem per each time step in the sample series. Thus, due to the time-series nature of the data, each time-step had

to have its own consistency classification, which would require the consistency neural network to generate a numerical consistency sequence per the four input streams.

Following were the three consistency labeling categories:

- **Consistent (2)**: For each time step that a sensor track was considered consistent as determined by the tracking of a single object at any given time according to all four applicable streams, that particular sensor track was marked with a 2. Consistent objects are determined after two consistent time steps of the same data coming from any given sensor track.

- **Inconsistent (1)**: For each time step that a sensor track was considered inconsistent as determined by a change in tracking from any of the following, according to all four applicable streams, that particular sensor track was marked with a 1:

  - Unknown data to a consistent object
  - One consistent object to another consistent object
  - One consistent object to unknown data

- **Unknown (0)**: For each time step that a sensor track was considered unknown as determined by tracking very noisy, unrecognizable or zero data, according to all four applicable streams, that particular sensor track was marked with a 0.

A labeling algorithm was developed in order to properly generate the consistency sequences corresponding to each sensor track. The inputs to the algorithm were the sensor tracks for a given driving instance and the outputs were consistency sequence labels per each input sensor track, useful to determine when any sensor track had unknown, inconsistent or consistent data. These sequences also proved useful for segmenting the sensor data for neural network training and evaluation as well as for determining the time steps when fusion was applicable between any pair of sensor

tracks. Sample consistency neural network inputs and outputs, which showcase an inconsistent radar track with its generated consistency sequence, are shown in figure 2.5.

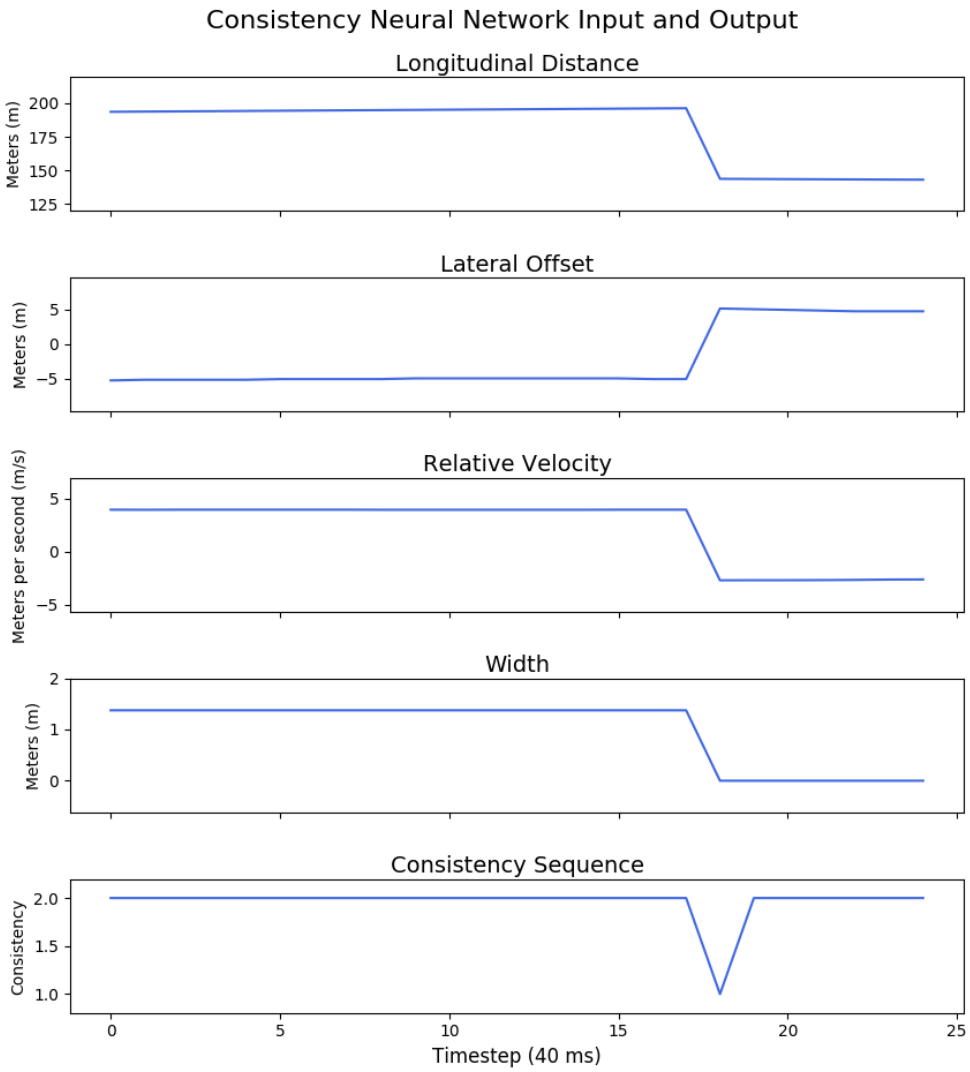

Figure 2.5: A time-series sample of consistency neural network input and output, which consist of synchronized radar track streams over 25 consecutive 40 ms time steps and a corresponding consistency sequence. In this case, the radar track streams reveal an inconsistent radar track. The consistency sequence is 2 when the track follows the same target and 1 when the track begins to follow a new target.

The consistency dataset was generated by sequencing random samples of 25 consecutive time steps of radar track streams of the 4 chosen types discussed in section 2.4 along with the consistency sequences corresponding to those random samples as

generated by the labeling algorithm. The input values were the radar track streams and the target output values for model training and validation were the consistency sequences.

2.6.4 Labeling of Data and Dataset Creation for Sensor Fusion

In order to create fusion datasets for training, testing and validating the fusion neural networks, examples of where two sensor tracks were fused or not fused were extracted and labeled. The examples consisted of 25 time steps each of the absolute differences between the two paired sensor track streams, where the data was either consistent and fused across both sensor tracks for fused cases or had mixed consistency and was not fused for non-fused cases. Thus, each example consisted of 100 time-series data points, 25 for each of the four stream differences between the two sensor tracks. Integer classes were used to represented the binary fusion cases. The fused cases were mapped to the class represented by a one while the non-fused cases were mapped to the class represented by a zero.

Five different fusion datasets were constructed to test the fusion neural networks using the same sensor data. One dataset consisted of fusion examples from the state-of-the-art sensor fusion system applied to the data. Four other datasets were created based on a hand-engineered fusion labeling algorithm which was constructed to fuse sensor tracks based on configured error thresholds between each of the four chosen synchronized sensor stream track pairs. The error thresholds were determined using statistics from fusion occurrences based on the state-of-the-art fusion system. Sample fusion neural network inputs and outputs, which showcase a fused pair of camera and radar track absolute differences with their corresponding binary fusion label sequence, are shown in figure 2.6.

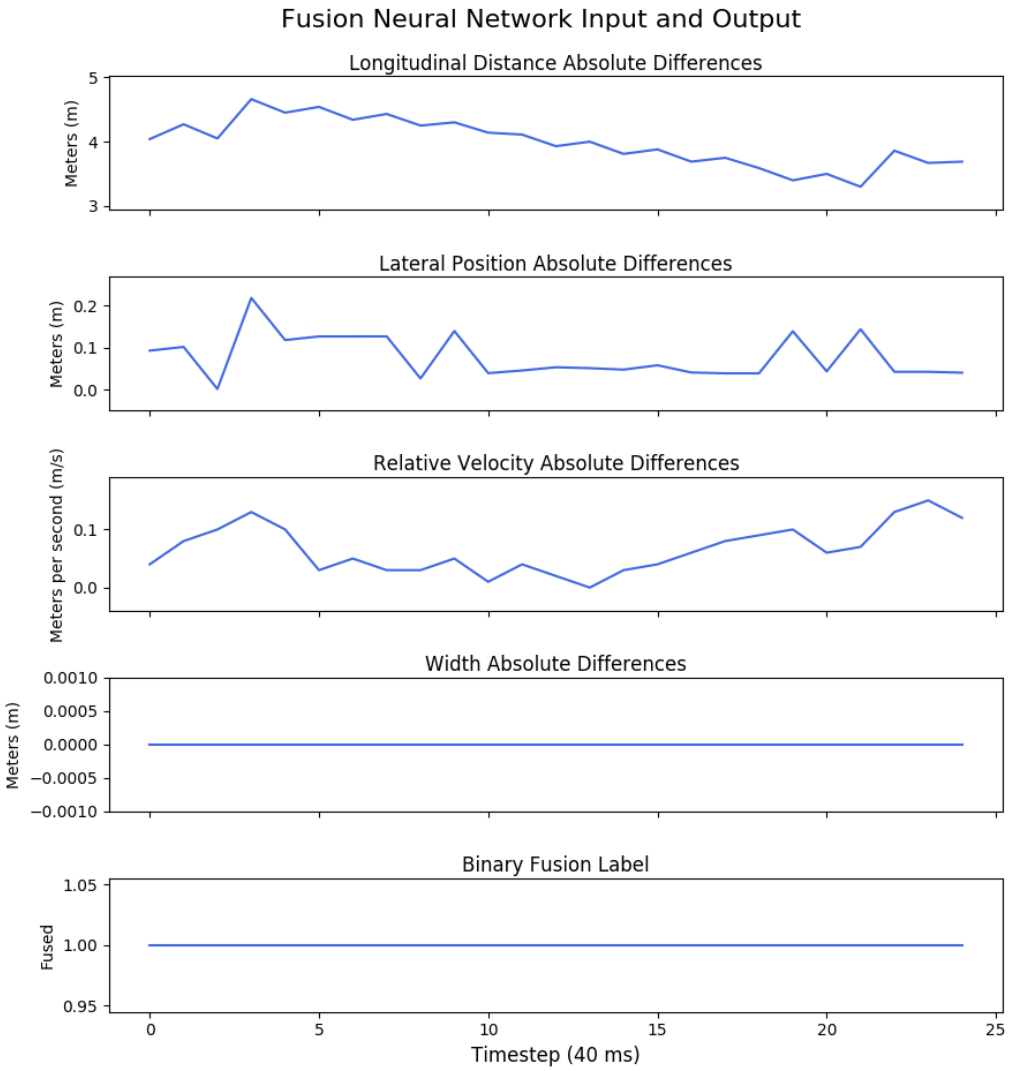

Figure 2.6: A time-series sample of fusion neural network input and output, which consist of the absolute differences between fused radar and camera track streams belonging to one consistent moving object over 25 consecutive 40 ms time steps and a corresponding binary fusion label sequence.

State of the Art Sensor Fusion System Dataset

One labeled fusion dataset was extracted from the state-of-the-art fusion system with outliers filtered out. The state-of-the-art fusion system occurrences were generated based on the provided sensor data database by black-box sensor fusion system known to perform well in real-world scenarios. This preexisting sensor fusion system used more than the four selected streams discussed in section 2.4 to determine fusion,

which most likely included actual video data from the camera sensor. The fusion results generated by this system were verified to be accurate within their own means based on calibrated sensor errors and noise learned from the actual vehicles. The scenarios were gathered from many cases of real-world driving across multiple road, lighting, and weather conditions. All fusion occurrences from this system, excluding outliers, were selected for analysis to determine reasonable error thresholds between the utilized sensor track streams for generating other datasets.

Hand-engineered Sensor Fusion Algorithm Datasets

The hand-engineered labeling algorithm was created in order generate customized fusion datasets with adjustable error criteria between any two pairs of synchronized object track samples consisting of 25 consecutive time steps across the entire sensor database. The entire database was searched for track pair sequences of 25 consecutive time steps where both tracks were consistent and fused or had mixed consistency and were not fused.

For any pair of tracks between radar and camera, the absolute differences between each of the four selected streams of each track were used to determine the error between the two tracks. The maximum absolute difference for each stream type was matched to a threshold criteria for fusion. If the absolute error for any given stream type exceeded the set error threshold for that stream type, then the track data sample was considered non-fused. If all errors between the two tracks met the fusion threshold criteria, the sample was considered fused.

The error thresholds provided to the algorithm had the original units of the stream type, whether it was meters or meters per second. Given that fusion was binary, the sample pairs that were not considered fused were considered non-fused and vice-versa. The hand-engineered algorithm allowed one to adjust the maximum threshold of error between any two radar and camera streams of the same type to be considered fused.

This algorithm allowed for each stream error threshold to be set individually, so that the user could decide their own separate stream error thresholds based on a selected level of noise found in each sensor stream as well as a selected level of error between the two chosen sensor streams when fusion occurred. This method had no guarantee that the fused cases would be true fusions, but it left the discretion of how to label fusion cases up to the user. The user would measure the typical error in sensor streams between the two sensor tracks when fused by the state-of-the-art fusion system and consider the typical noise of each sensor stream in order to best determine allowable error thresholds for fused samples.

The errors considered when labeling fused and non-fused example cases between any two pair of object tracks were the following:

- Absolute error in longitudinal distance streams per both sensor tracks
- Absolute error in lateral position streams per both sensor tracks
- Absolute error in relative velocity streams per both sensor tracks
- Absolute error in width streams per both sensor tracks

The fused example cases in the hand-engineered algorithm were determined by whether the maximum value of the absolute differences per pair of track streams were less than or equal to the maximum error threshold for that stream type. The non-fused example cases, on the other hand, were determined when the maximum of the absolute differences between any pair of sensor streams was greater than the set error threshold for that given stream type.

The four chosen sets of fusion error thresholds used when generating the four fusion datasets for network experimentation were the following:

- **Most restrictive**: Error thresholds determined by subtracting half the standard deviation from the mean of the absolute differences of each stream across samples of fusion matches found by the state-of-the-art system.

- **Mid-restrictive**: Error thresholds determined by adding half the standard deviation to the the mean of the absolute differences of each stream across samples fusion matches found by the state-of-art system.
- **Least restrictive**: Error thresholds determined by taking one-third of the maximum of the absolute differences of each stream across all fusion matches found by the state-of-art system.
- **Maximum threshold**: Error thresholds determined by taking one-half of the maximum of the absolute differences of each stream across all fusion matches found by the state-of-art system and adding observed sensor error.

Multiple fusion datasets were generated to showcase the fusion network performance across multiple variations of noise and notions of "fusion" between the two sensors. All sensors are different and have different noise distributions and assumptions. The idea was that depending on the two sensors and their noise amounts, the dataset generation algorithm could be tuned to generate a binary fusion dataset with "fused" examples that allowed for the assumed amount of error between the selected sensor streams up to a certain threshold in any of the four streams. Once the absolute error was beyond any of these tuned thresholds in any of the four streams between the two sensor tracks, the examples were deemed "non-fused" in the dataset. Given that each dataset allowed for different amounts of absolute error between the two sensors, the training and testing of the neural network versions on each of these datasets showcased their ability to perform on multiple variations of sensor noise fusion assumptions.

2.7 Sensor Consistency Deep Neural Network

The sensor consistency neural network was a model created for the purpose of predicting the consistency of each time step of a 25 time step sample of a radar target

track. The network was trained to accept 4 streams of 25 time step samples of radar track data in order to generate a 25 time step sequence of consistency values in the range [0, 2]. The consistency labels were discussed in section 2.6.3.

2.7.1 Sensor Consistency Deep Neural Network Design

The consistency network was a multi-stream neural network. It accepted 100 inputs and generated 25 outputs. It consisted of four pathways, each of which accepted 25 inputs. Each pathway was designated for one of the four chosen streams of radar input data consisting of 25 consecutive time steps from a given target track. Each pathway of the network consisted of the same sequential neural network model, but they were all trained to learn different weights between the connections. The pathway model was developed based on [Cho et al., 2014], which applied an RNN encoder-decoder model as a seq2seq network for translating one language sequence into another. This method was shown to perform well on short sequences. Their model accepted a sequence in one language which passed through an encoder RNN layer and a decoder RNN layer, subsequently yielding a translated output sequence. A follow-up to their work showcasing an advanced version of this model was developed in [Bahdanau et al., 2014], which performed even better on longer sequences.

The chosen pathway model had an encoder layer and a decoder layer as in [Cho et al., 2014] and [Bahdanau et al., 2014], but it also had a fully-connected (FC) central embedding layer. This model was very similar to an auto-encoder in nature, except that it understood temporal context and its training was supervised. It has been shown that stacking network layers, especially recurrent with FC layers, improves network predictive performance by providing temporal context for sequence generation [Plahl et al., 2013]. The encoder and decoder layers each had 25 units and their own sigmoidal activation functions. Each of these layers had its own GlorotUniform random weight initialization scheme and was followed by a dropout layer. In training,

the dropout layer was necessary for network regularization in order to prevent over-fit of certain training examples [Hinton et al., 2012], [Srivastava et al., 2014]. A central embedding layer with non-linear activation functions and random weight initialization connected the encoder and decoder layers. Experiments were conducted with four versions of simultaneously paired encoder and decoder layers in the multi-stream network. The four chosen alternatives of encoder and decoder layers are listed in table 2.1.

Table 2.1: Sensor Consistency Neural Network Encoder and Decoder Layer Types

Network Version	Layer Type
1	Fully-Connected
2	Recurrent
3	Gated Recurrent Unit
4	Long-Short Term Memory

Each of these pathway networks yielded a series of features with potentially temporal context, which were concatenated into a single vector of features and fed to a MLP network. The MLP consisted of one hidden layer and an output layer. The neurons of both layers had non-linear activation functions. The last layer generated the consistency sequence of the multi-modal input sample consisting of real-valued numbers in the range [0, 2]. These numbers represented where data was unknown, inconsistent, or consistent. The general model structure is shown in figure 2.7.

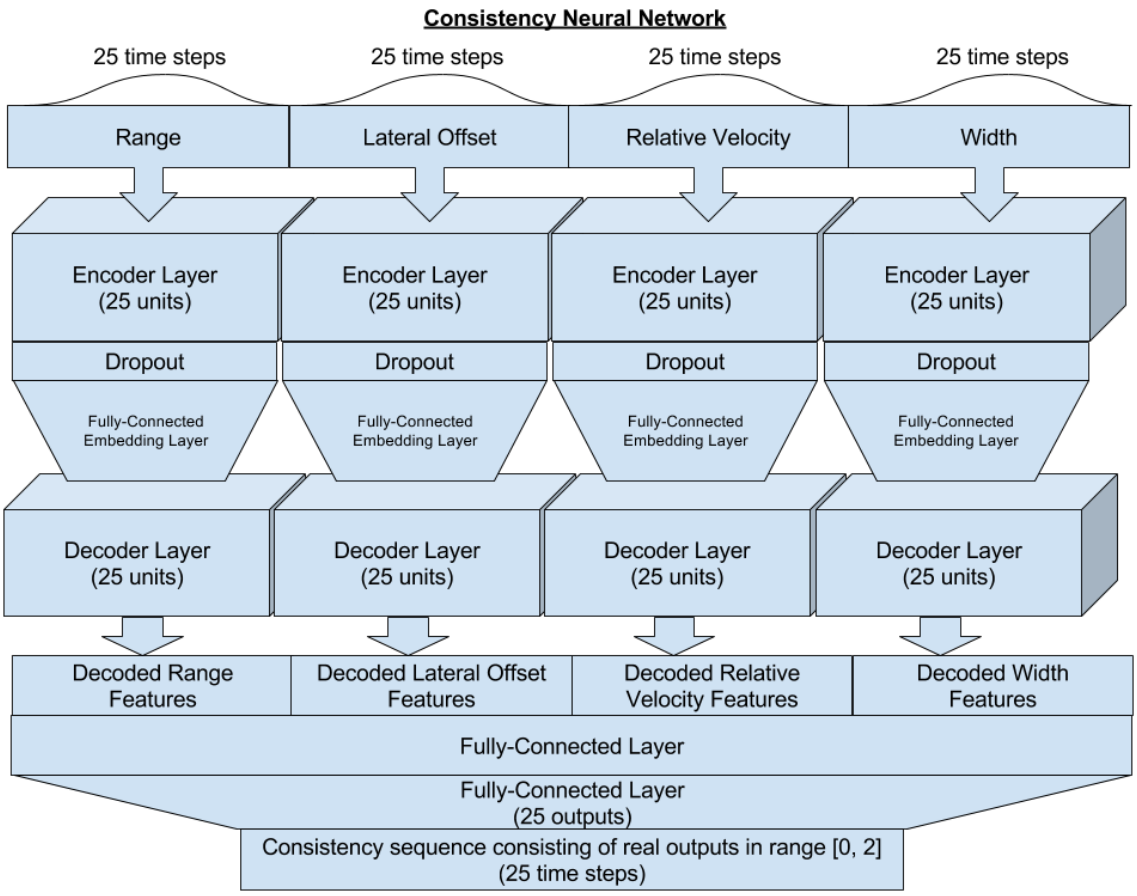

Figure 2.7: The multi-stream seq2seq consistency sequence deep neural network. In experiments, the encoder and decoder layers were selected from table 2.1.

2.7.2 Sensor Consistency Network Training Method

For training the seq2seq neural network, the RMSProp optimization algorithm as described in section 1.5.4 was utilized with either BP or BPTT. BP was applied when the network was feedforward, while BPTT was applied when the network was recurrent. Batch training was applied with a batch size of 10,000 in order to increase speed on large datasets and accommodate nearly one million examples across train and test partitions. A learning rate of 0.002 and a state decay rate of 0.95 were used for the problem. These parameters provided a balance of learning speed and quality. The cost function used to evaluate and minimize the error of outputs from the network was the RSS cost function. The RSS cost function was chosen because it

was best able to minimize the error of sequences generated by the network across all components of the output sequence rather than focusing on the mean of the squared errors as in the MSE function.

The RMSProp algorithm with RSS was proven effective for training seq2seq recurrent neural networks as in the work of [Graves, 2014]. In [Graves, 2014], an LSTM was trained to generate cursive handwriting sequences with a high level of realism. In the case of sensor fusion, the smallest errors needed to be acknowledged when training the neural network due to the inherent reliability required in such a system. Given the susceptibility of the RSS to outliers and the lack thereof in the MSE cost function, the RSS was needed for training this particular network in order to minimize error even in rare cases.

2.7.3 Sensor Consistency Network Training Cycles and Data Partitioning

Each of the four consistency network versions constructed based on table 2.1 were trained and tested to completion over 9 bootstrapping cycles to determine the average regression performance of the selected network version. Bootstrapping was utilized to provide statistical certainty that a given network version was appropriate across a sample distribution over all available consistency sequence data.

In each bootstrapping cycle, the labeled consistency network data was randomly partitioned into two separate categories of 70 percent train and 30 percent test data sampled from the overall set of millions. Per each bootstrapping cycle, there were hundreds of thousands of samples per train and test partition.

Upon each training epoch, the network was trained using RMSProp with RSS on the train data partition. After each training epoch, the network was tested using the same metric on the test data partition. The test data was never exposed to the network during training, thus it was used to evaluate the general regression performance

of the network. Once the network reached a plateau in cost between both train and test partitions, the network training was ceased. The number of selected train epochs for each network was determined through observation of the train and test cost trends characteristic of the network version across multiple random samples of the data.

2.8 Sensor Consistency Network Validation Cycles and Metrics

At the end of each bootstrapped training cycle, the trained consistency network model was validated using the randomly-selected test dataset with multiple metrics in inference mode to determine how well the trained network model performed on the test set after training was complete. The metrics used to evaluate the consistency seq2seq network after training were the MSE, RMSE, and r^2-metric. The implementation and visualizations for each of these metrics were constructed via scikit-learn and matplotlib.

2.9 Sensor Fusion Deep Neural Network

The sensor fusion neural network was a model created for the purpose of predicting whether two pairs of synchronized sensor track samples, each consisting of 25 consecutive time steps, were fused, based on the absolute differences of the 4 chosen streams between the two tracks. The fusion dataset generation was discussed in section 2.6.4. The network was trained to accept 4 difference sequences between the 4 streams of the paired sensor tracks in order to classify whether the two sensor tracks were fused via binary classification.

2.9.1 Sensor Fusion Deep Neural Network Design

The fusion network, much like the consistency network, was a multi-stream neural network. It accepted 100 inputs and generates 25 outputs. It consisted of four pathways, each of which accepted 25 inputs. Each pathway was designated for the absolute difference sequence computed between one of the four chosen stream types of two synchronized, paired sensor tracks each consisting of 25 consecutive time steps. Each pathway of the network consisted of the same sequential neural network model, but they were all trained to learn different weights between the connections. This pathway model was also developed based on [Cho et al., 2014] and [Bahdanau et al., 2014], which applied a RNN encoder-decoder network for seq2seq translation from one language sequence into another.

The chosen pathway model had an encoder layer and a decoder layer as in [Cho et al., 2014] and [Bahdanau et al., 2014], but it also had a FC central embedding layer. This model was very similar to an auto-encoder in nature, except that it understood temporal context and its training was supervised. It has been shown that stacking network layers, especially recurrent with FC layers, improves network predictive performance by providing temporal context for sequence generation [Plahl et al., 2013]. The encoder and decoder layers each had 25 units and their own sigmoidal activation functions. Each of these layers had its own GlorotUniform random weight initialization scheme and was followed by a dropout layer. In training, the dropout layer was necessary for network regularization in order to prevent over-fit of certain training examples [Hinton et al., 2012], [Srivastava et al., 2014]. A central embedding layer with non-linear activation functions and random weight initialization connected the encoder and decoder layers. Experiments were conducted with four versions of simultaneously paired encoder and decoder layers in the multi-stream network. The four chosen alternatives of encoder and decoder layers are listed in table 2.2.

Table 2.2: Sensor Fusion Neural Network Encoder and Decoder Layer Types

Network Version	Layer Type
1	Fully-Connected
2	Recurrent
3	Gated Recurrent Unit
4	Long-Short Term Memory

Each of these pathway networks yielded a series of features with potentially temporal context, which were concatenated into a single vector of features and fed to a MLP network. The MLP consisted of two hidden layers and an output SoftMax classification layer. The neurons in the hidden layers had non-linear activation functions. The output layer applied the SoftMax activation function with two classes to generate confidences in whether the samples were fused via binary classification. The general model structure is shown in figure 2.8.

2.9.2 Sensor Fusion Network Training Method

The fusion neural network was trained via the AdaGrad optimization algorithm as described in section 1.5.4 with either BP or BPTT. BP was applied when the network was feedforward, while BPTT was applied when the network was recurrent. Batch training was applied with a batch size of 10,000 to increase speed on large datasets and accommodate millions of examples across train and test partitions. The cost function used to evaluate and minimize error in network classification was the cross-entropy binary cost function. This cost function was useful for minimizing error in the binary classification problem. A learning rate of 0.01 for AdaGrad was applied for the problem in order to provide a balance between learning speed and quality.

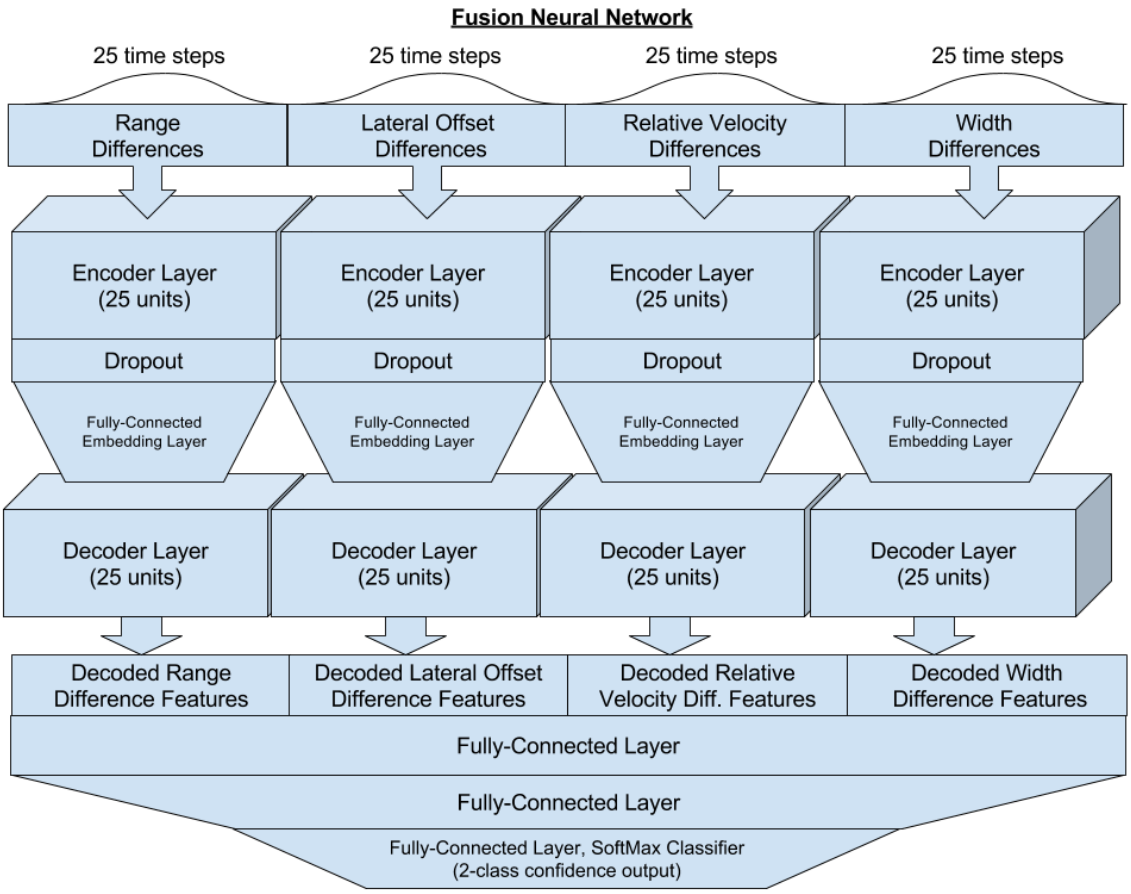

Figure 2.8: The multi-stream sensor fusion classification deep neural network. In experiments, the encoder and decoder layers were selected from table 2.2.

2.9.3 Sensor Fusion Network Training Cycles and Data Partitioning

Each of the four fusion network versions constructed based on table 2.2 were trained and tested to completion over 9 bootstrapping cycles per each of the five fusion dataset types to determine the average classification performance of the selected network version per the selected dataset type. Bootstrapping was utilized to provide statistical certainty that for a given dataset with specific fusion error thresholds, a given network version was appropriate to apply by training and testing across a sample distribution of the given dataset over all available fusion data for that dataset.

In each bootstrapping cycle, per each network and dataset, the labeled fusion

network data was randomly partitioned into two separate categories of 70 percent train and 30 percent test data randomly sampled from the overall specifically-labeled dataset of millions of examples. Per each bootstrapping cycle, there were millions of training and testing samples.

Upon each training epoch, the network was trained using AdaGrad with the cross-entropy binary cost function on the train data partition. After each training epoch, the network was tested using the same metric on the test data partition. The test data was never exposed to the network during training, thus it was used to evaluate the general classification performance of the network. Once the network reached a plateau in cost between both train and test partitions, the network training was ceased. The number of selected train epochs for each network was determined through observation of the train and test cost trends characteristic of the network version across multiple random samples of the selected dataset.

2.10 Sensor Fusion Network Validation Cycles and Metrics

At the end of each bootstrapped training cycle, the trained fusion network model was validated on the randomly-selected test dataset with multiple metrics in inference mode to determine how well the trained network model performed on the test set after training was complete. The metrics used to evaluate the fusion network after training were accuracy, misclassification, precision, recall, F1 weight macro, ROC AUC micro average and ROC AUC macro average. The implementation and visualizations for each of these metrics were constructed via scikit-learn and matplotlib.

2.11 Neural Network Training Results Visualization

The training results for all networks were stored in the hierarchical data format 5 (hdf5) format. This was a format developed to store huge amounts of numerical data in binary, hierarchical format for easy manipulation and fast access [hdf, 2017]. The library used for reading data of this type in python was h5py. The h5py library allowed one to store and manipulate data in this format as if it were a numpy array [Collette, 2013]. The results were visualized using the Nervana Systems nvis script designed for easy training results visualization using h5py and Bokeh to generate html-embedded Bokeh plots [Nervana Systems, 2017d]. Bokeh is an interactive visualization library designed around modern web browsing for high-performance interactivity over large datasets much like the popular D3.js [Continuum Analytics, 2015].

2.12 Deep Learning Sensor Fusion System

High-Level System Description

The deep learning fusion system was designed much in the style of fusion dataset generation algorithm with the use of trained deep neural networks instead of static error thresholds for measuring similarity. The best version of the consistency network was employed to generate radar track consistency sequences based on the latest 25 time steps of radar track data. The vehicle IDs provided by the camera software were utilized to determine the consistency sequences of the latest 25 time steps of camera track data. These sequences were binarized into bins of consistent and unknown data. Then, for each radar track and each camera track, the sequences were logically AND'ed to determine where the tracks were both consistent. The absolute differences were taken between the given track pairs, and these differences were masked based

on the latest point of consistency for the given track pairing.

From the point of the latest consistent data between both tracks, the differences were interpolated backward for each stream pair to cover the masked stream space and fit the data to the input dimension of the fusion deep neural network. If both tracks were consistent through the latest 25 time steps, then the true absolute differences would be used without any masking. The differences for each combination of tracks were then propagated through the best version of the fusion network to determine the probability of fusion between the two sensor tracks. All combinations of tracks were compared and the best matches were taken. Any fusion ties with high confidence were broken based on the track ranking number, where smaller was higher priority. Then, each radar track was associated with a given camera vehicle ID, when possible, based on the fusion confidence. Pairings with confidence below 0.5 were considered non-fused. At times, multiple radar tracks could be fused to the same camera track given overlap in radar coverage. The end result was that for each 40 ms time step, each radar track was fused with a camera track and its associated vehicle ID when possible.

Chapter 3

Results

3.1 Correlation of Consistent Sensor Tracks

According to the state-of-the-art fusion system, when any two fused sensor tracks were simultaneously consistent, with no sudden track or ID reassignments over a selected period of time, there was a strong correlation between the data streams of those two tracks. The correlations in these cases were measured with the r^2-metric (R2-Metric), the normalized cross-correlation (Norm-XCorr), and the median 1-dimensional Euclidean distance (Median 1-D Euclid) between each of the four stream pairs. The correlations found between selected samples of fused, consistent sensor track pairs are shown in table 3.1.

Table 3.1: Sample Correlation Metrics between Consistent, Fused Sensor Tracks according to State-of-the-art Fusion System.

Stream	R2-Metric	Norm-XCorr	Median 1-D Euclid
Longitudinal Distance	0.9911	0.9992	4.960 +/- 3.554 m
Lateral Position	0.9968	0.9968	0.077 + 0.199 m
Relative Velocity	0.968	0.9804	0.270 + 0.470 m/s
Width	0.986	0.9998	0.025 +/- 0.018 m

3.2 Sensor Consistency Neural Network Training

3.2.1 Training Errors vs. Time

All consistency neural networks were trained using the NVIDIA GTX 980Ti GPU with CUDA 8. Each network was trained over 9 bootstrapping trials on over one million samples. For each trial, there were hundreds of thousands of randomly-sampled training examples, amounting to 70 percent of the dataset, and hundreds of thousands of test examples, derived from the remainder of the dataset. For each training trial, the sum-squared error according to equation 1.13 was measured between the predicted outputs of the network and the target outputs for both the train and test sets of data. Each of the following plots showcases an example of a training trial for each of the four consistency neural network versions from random weight initialization to training completion once minimal regression error was met. Plot 3.1 showcases the regression error over time for a train/test trial of the MLP version of the consistency neural network.

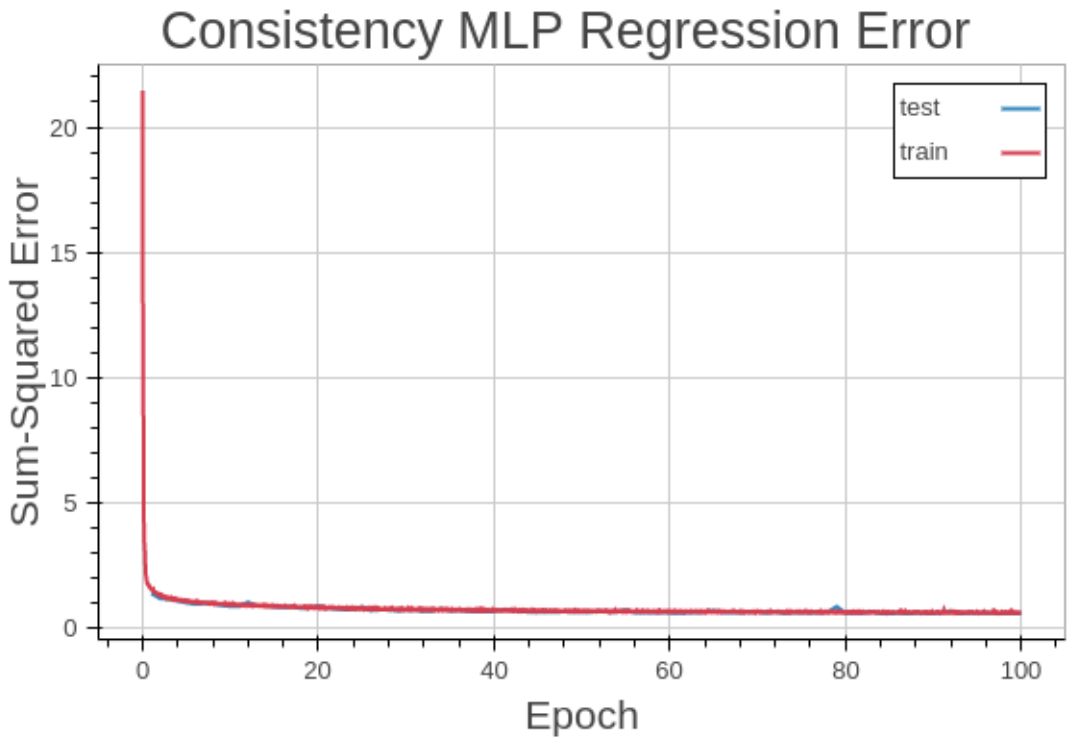

Figure 3.1: An example of the train and test error over time per one complete cycle of consistency MLP network training.

According to plot 3.1, the MLP version of the consistency network took 100 epochs to train until the minimal error amount was reached. Each training epoch took 1.05 seconds over 85 batches, and each evaluation cycle took 0.1 seconds on average. Plot 3.2 showcases the regression error over time for a train/test trial of the RNN version of the consistency neural network.

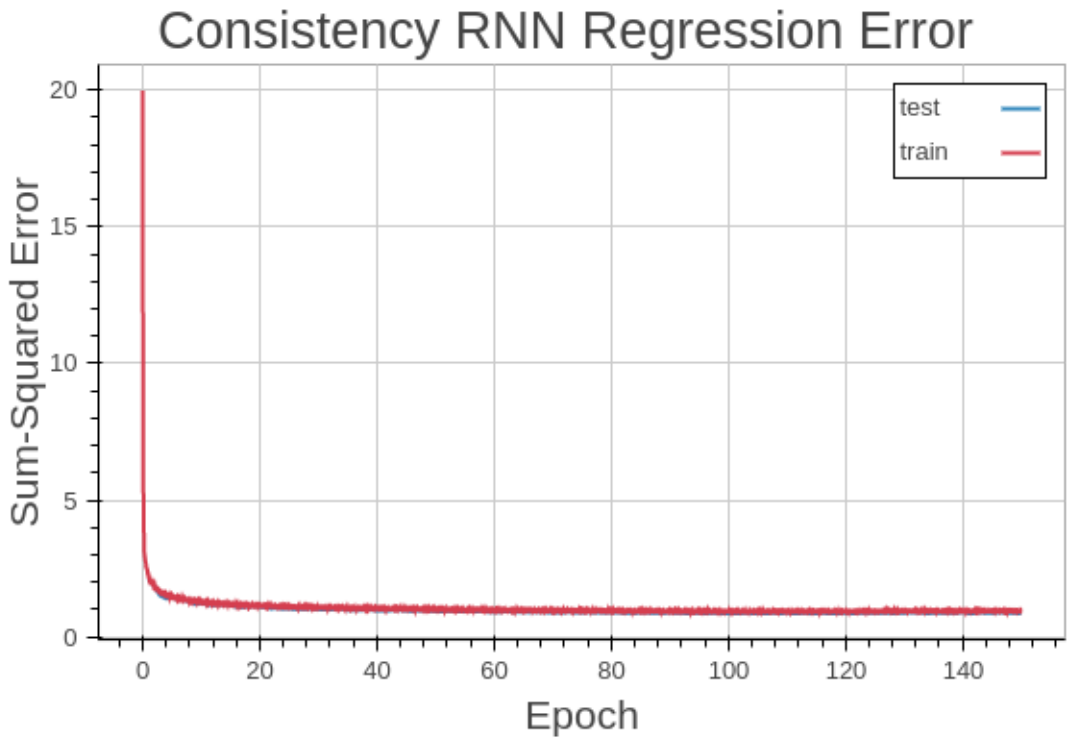

Figure 3.2: An example of the train and test error over time per one complete cycle of consistency RNN network training.

According to plot 3.2, the RNN version of the consistency network took 150 epochs to train until the minimal error amount was reached. Each training epoch took 1.2 seconds over 85 batches, and each evaluation cycle took 0.12 seconds on average. Plot 3.3 showcases the regression error over time for a train/test trial of the GRU version of the consistency neural network.

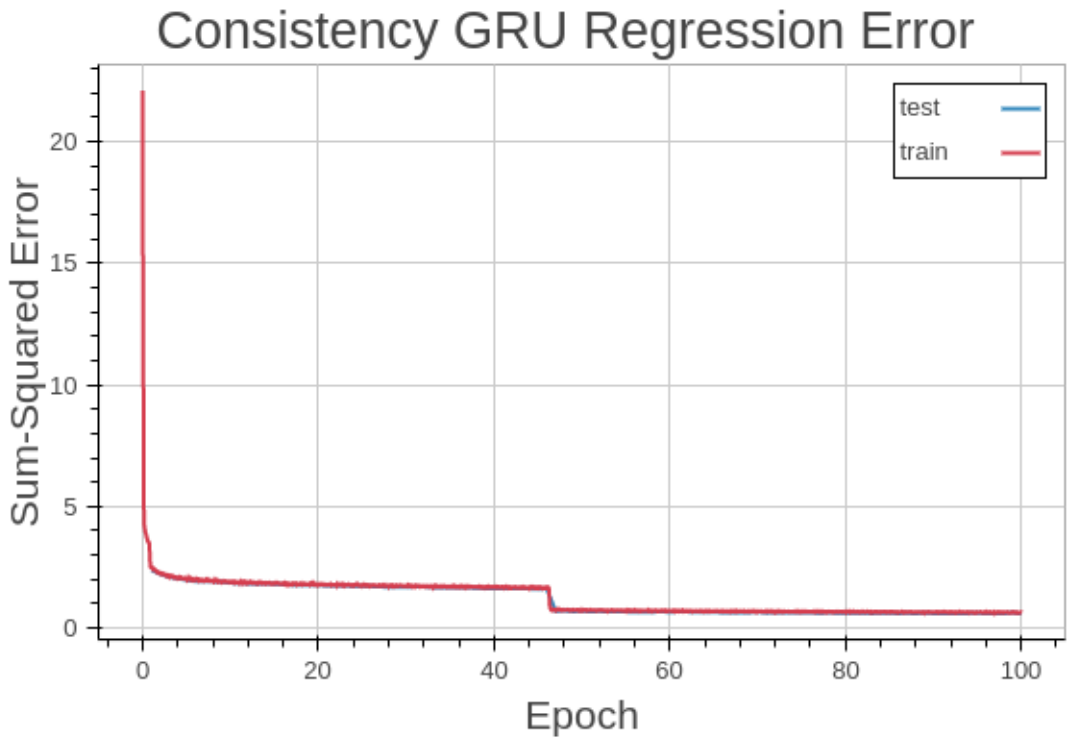

Figure 3.3: An example of the train and test error over time per one complete cycle of consistency GRU network training.

According to plot 3.3, the GRU version of the consistency network took 100 epochs to train until the minimal error amount was reached. Each training epoch took 1.8 seconds over 85 batches, and each evaluation cycle took 0.25 seconds on average. Plot 3.4 showcases the regression error over time for a train/test trial of the LSTM version of the consistency neural network.

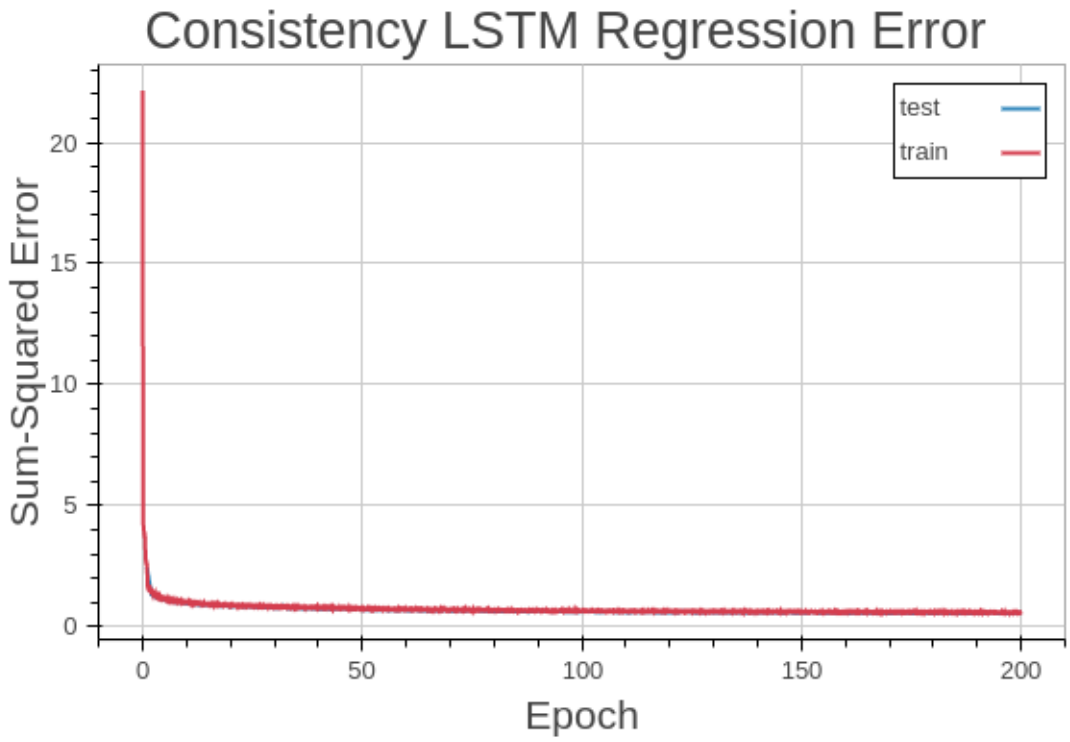

Figure 3.4: An example of the train and test error over time per one complete cycle of consistency LSTM network training.

According to plot 3.4, the LSTM version of the consistency network took 200 epochs to train until the minimal error amount was reached. Each training epoch took 1.8 seconds over 85 batches, and each evaluation cycle took 0.25 seconds on average.

3.3 Sensor Consistency Neural Network Validation Results

Figures 3.5, 3.6, 3.7, and 3.8 showcase the validation performance of the consistency neural network variations on randomly-selected test datasets generated from actual sensor data using the MSE, RMSE, and r^2-metric regression metrics across 9 bootstrapping trials. For none of the evaluations were the neural networks exposed to the test dataset during the training cycle.

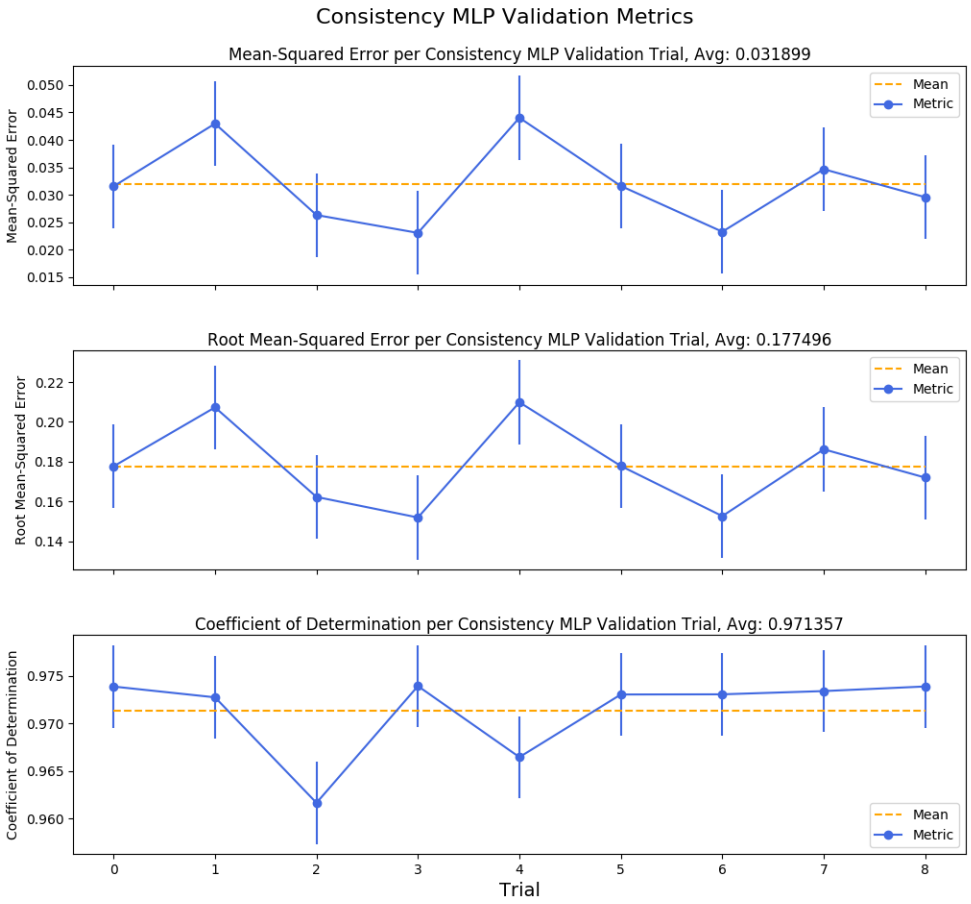

Figure 3.5: MSE, RMSE and r^2-metric performance for the consistency MLP neural network on the test dataset across 9 bootstrapping train/test trials from over 1 million randomly-selected samples. The average score and its standard deviation are included in the title of each subplot.

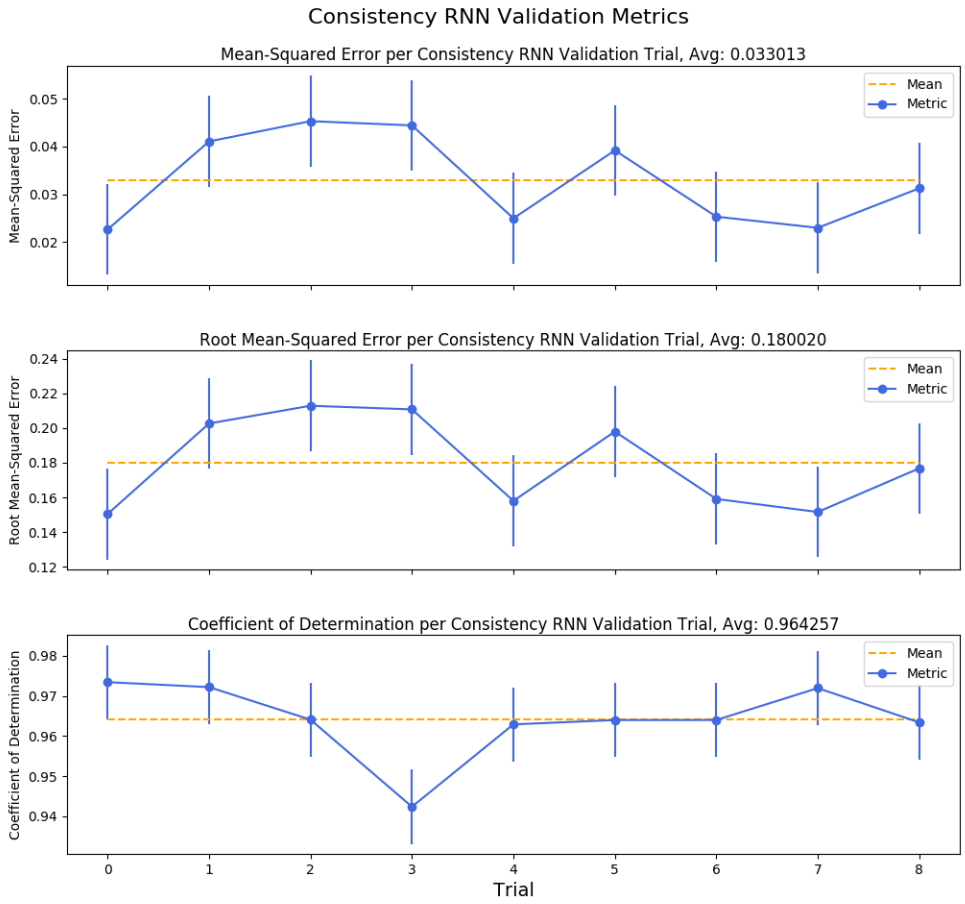

Figure 3.6: MSE, RMSE and r^2-metric performance for the consistency RNN neural network on the test dataset across 9 bootstrapping train/test trials from over 1 million randomly-selected samples. The average score and its standard deviation are included in the title of each subplot.

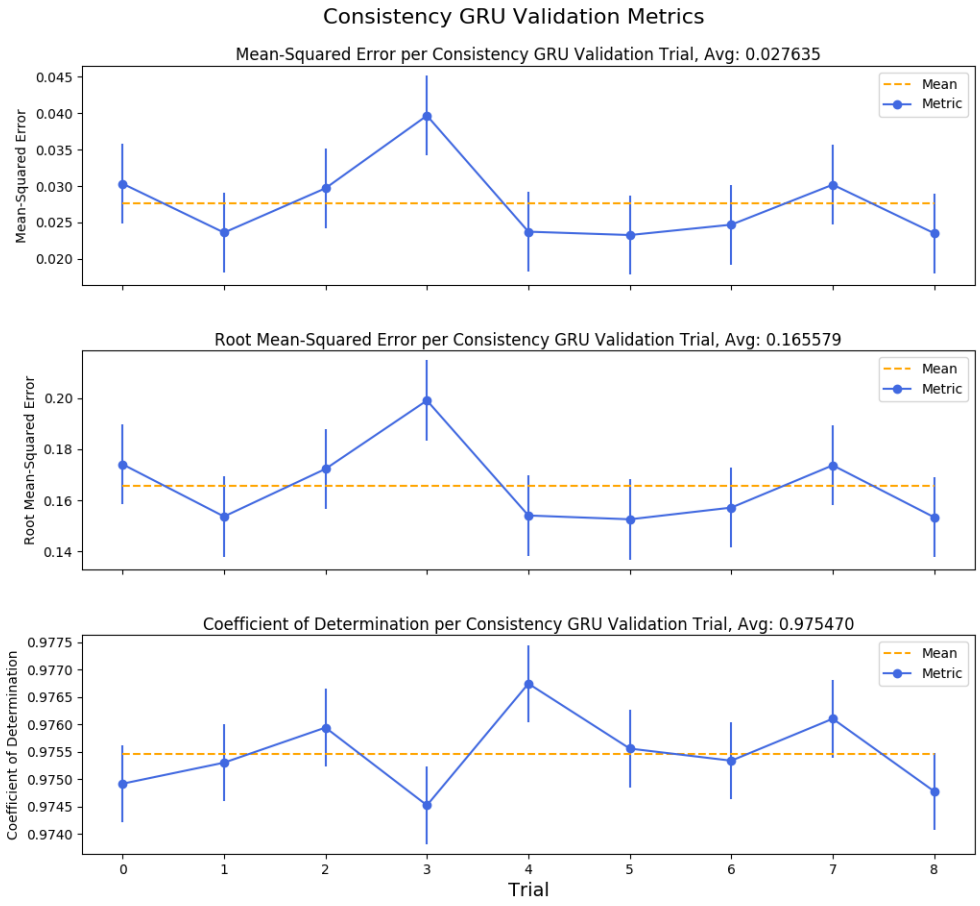

Figure 3.7: MSE, RMSE and r^2-metric performance for the consistency GRU neural network on the test dataset across 9 bootstrapping train/test trials from over 1 million randomly-selected samples. The average score and its standard deviation are included in the title of each subplot.

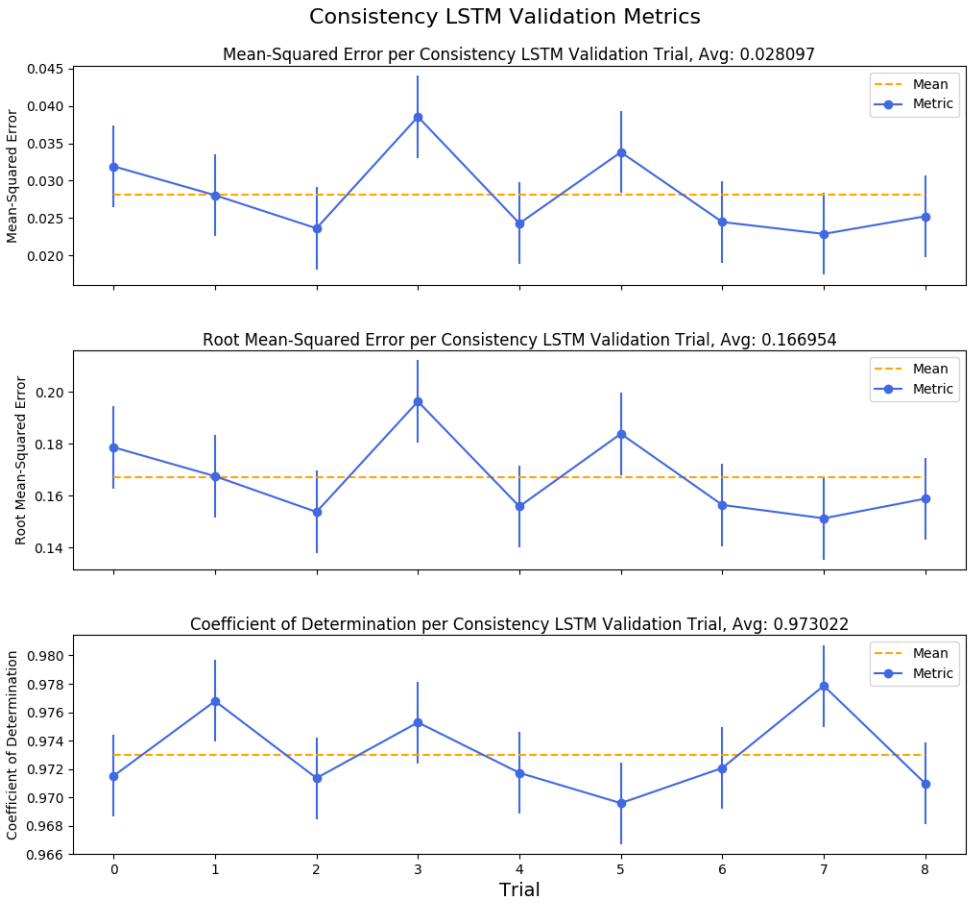

Figure 3.8: MSE, RMSE and r^2-metric performance for the consistency LSTM neural network on the test dataset across 9 bootstrapping train/test trials from over 1 million randomly-selected samples. The average score and its standard deviation are included in the title of each subplot.

3.4 Sensor Fusion Dataset Labels

3.4.1 State-of-the-art Sensor Fusion Labels Statistics

Statistics between fused camera and radar sensor tracks were computed to determine thresholds for the allowable error between each of the four sensor stream types when paired. The statistics were computed based millions of randomly-sampled fusion occurrences between radar and camera from the state-of-the-art fusion system with outliers removed. These statistics shaped the fused examples found in the first fusion

dataset based on the state-of-the-art fusion system. The statistics are shown in table 3.2.

Table 3.2: State-of-the-art fusion system error statistics between fused sensor track streams.

Stream	Mean	Max	Standard Deviation
Longitudinal Distance (m)	3.38	23.4	2.83
Lateral Position (m)	0.16	2.8	0.17
Relative Velocity (m/s)	0.52	5.4	0.57
Width (m)	0.03	1.88	0.020627

The amount of allowable error between the sensors depended directly on the sensor noise present in either sensor track. The amount of sensor noise depended on multiple factors, including time of day, weather, calibration and mounting point of sensor. Per each vehicle, the sensors were slightly different in regards to all of these factors and thus the sensor noise behaved differently depending on the vehicle the data was acquired from. These statistics were useful to determine multiple levels of allowable error thresholds for creating binary fusion datasets via the hand-engineered fusion labeling algorithm.

3.4.2 Generated Sensor Fusion Labels Statistics

Four binary fusion datasets were generated based on the computed statistics from the state-of-the-art fusion system. The error thresholds for each generated dataset version, as derived from the computed fusion statistics, were modeled according to table 3.3.

Table 3.3: Sensor Fusion Dataset Error Threshold Models.

Most Restrictive	Mid-Restrictive	Least Restrictive	Max Threshold
$mean - 0.5 * std$	$mean + 0.5 * std$	$1/3 * max$	$1/2 * max + noise$

The tolerated error thresholds between fused camera and radar tracks for each of the four streams for each of the four fusion datasets were computed based on the state-of-the-art fusion system statistics. These error thresholds are listed in table 3.4, where stream 1 is longitudinal distance, stream 2 is lateral position, stream 3 is relative velocity, and stream 4 is width.

Table 3.4: Tolerated Fusion Dataset Error Thresholds between Fused Sensor Streams.

Dataset	Stream 1 Error	Stream 2 Error	Stream 3 Error	Stream 4 Error
Most Restrictive	1.965 m	0.075 m	0.227 m/s	0.02 m
Mid-Restrictive	4.8 m	0.25 m	0.793 m/s	0.04 m
Least Restrictive	7.8 m	.93 m	1.8 m/s	0.62 m
Max Threshold	12 m	1.5 m	4 m/s	0.9 m

3.5 Sensor Fusion Neural Network Training

3.5.1 Training Errors vs. Time

All fusion neural networks were trained using the NVIDIA GTX 980Ti GPU with CUDA 8. Each network was trained over 9 bootstrapping trials on over two million samples per each fusion dataset. For each trial, there were 2.5 million randomly-sampled training examples, amounting to 70 percent of the dataset, and over 1 million test examples, derived from the remainder of the dataset. For each training trial, the cross-entropy error according to equation 1.18 was measured between the predicted outputs of the network and the target outputs for both the train and test sets of data. Each of the following plots showcases an example of a training trial for each of the four fusion neural network versions from random weight initialization to training completion once minimal classification error was met. Example plots are used for brevity since most of the plots per network version contained very similar outcomes.

Plot 3.9 showcases the classification error over time for a train/test trial of the MLP version of the fusion neural network.

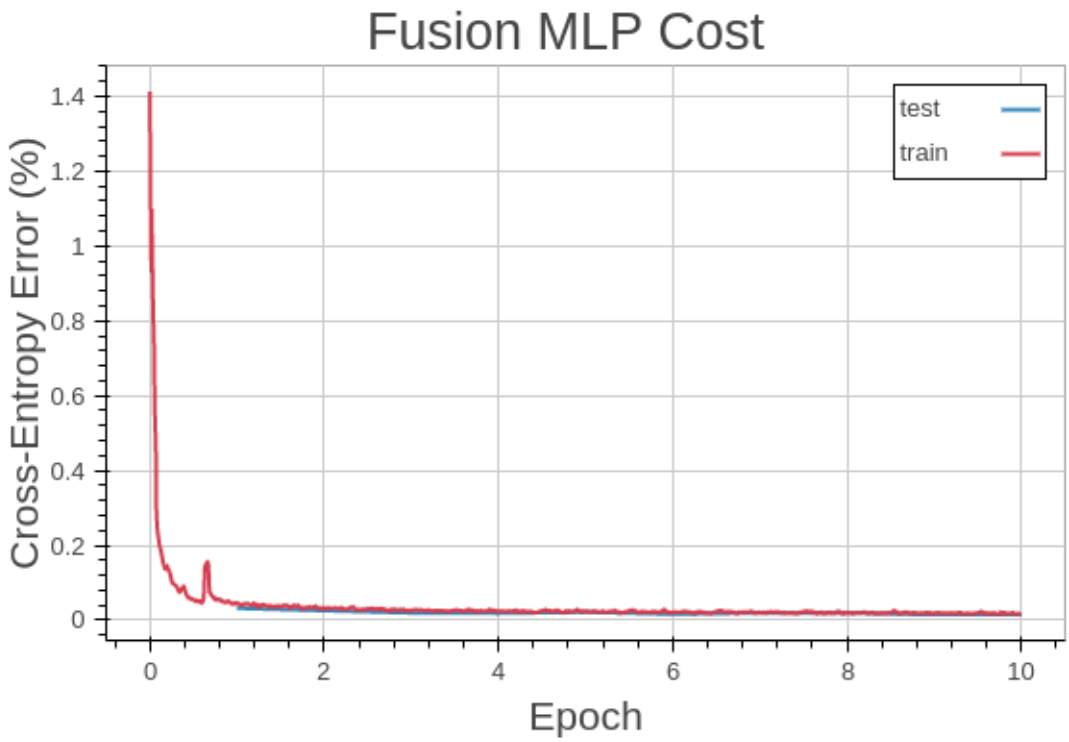

Figure 3.9: An example of the train and test error over time per one complete cycle of fusion MLP network training on the most restrictive fusion dataset.

The MLP version of the fusion neural network took 10 epochs to train per each fusion dataset. Each training epoch took 3.4 seconds over 254 batches, and each evaluation cycle took 0.33 seconds on average. Plot 3.10 showcases the classification error over time for a train/test trial of the RNN version of the fusion neural network.

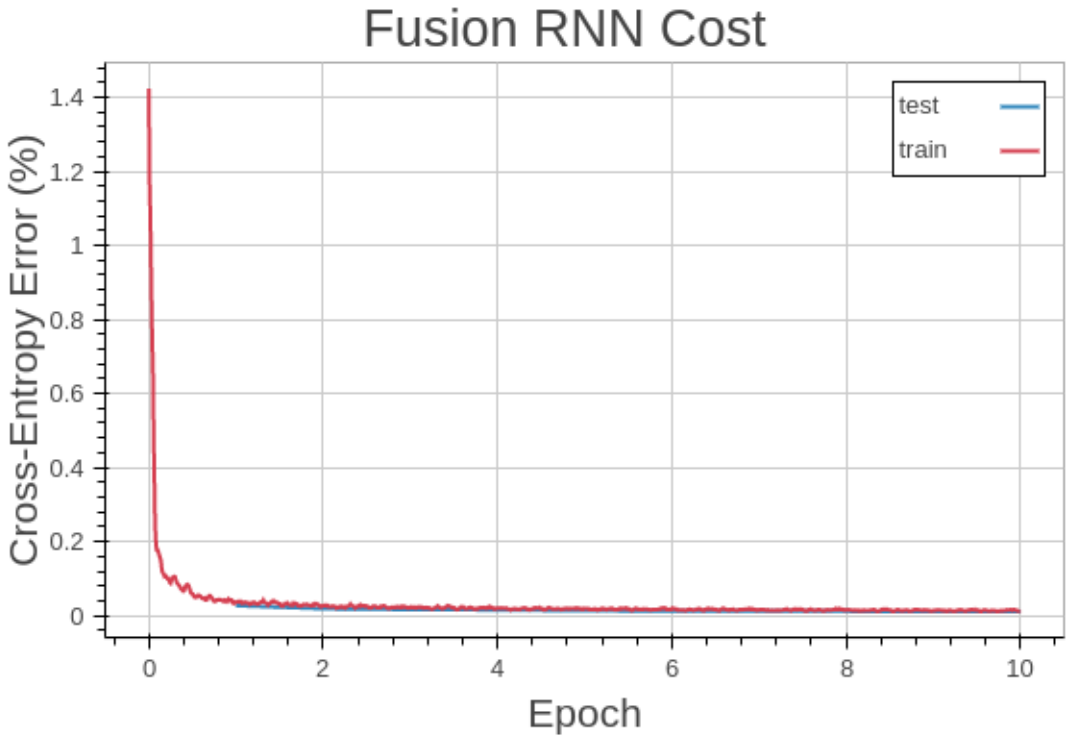

Figure 3.10: An example of the train and test error over time per one complete cycle of fusion RNN network training on the most restrictive fusion dataset.

The RNN version of the fusion neural network took 10 epochs to train per each fusion dataset. Each training epoch took 3.5 seconds over 250 batches, and each evaluation cycle took 0.4 seconds on average. Plot 3.11 showcases the classification error over time for a train/test trial of the GRU version of the fusion neural network.

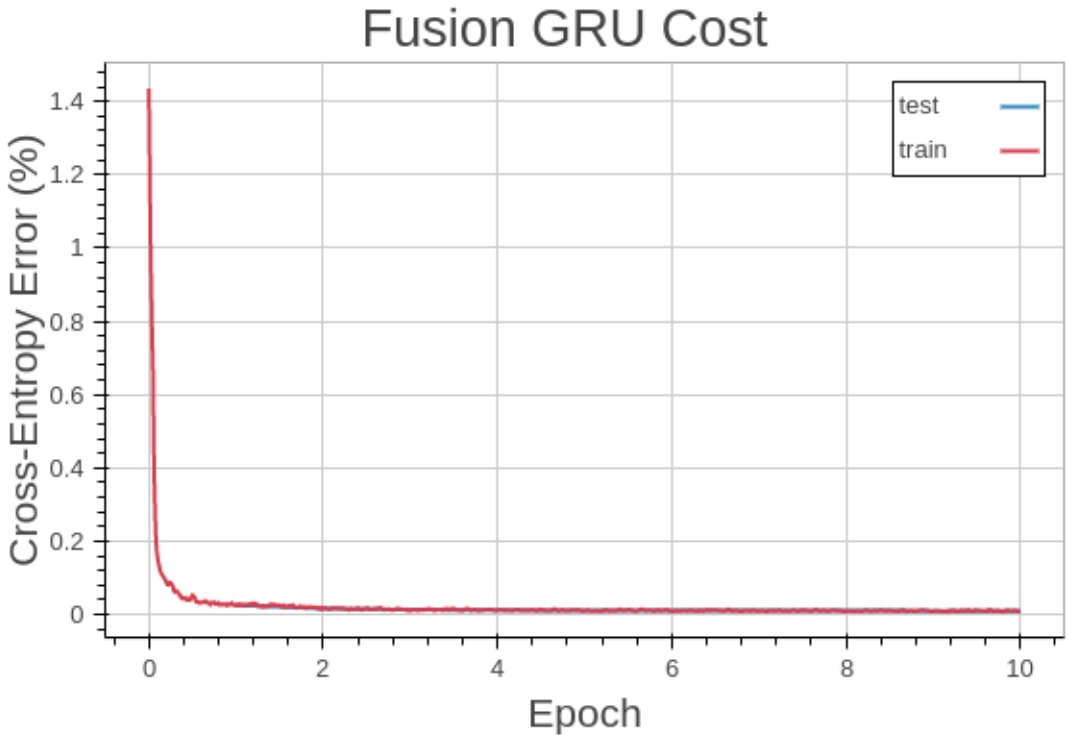

Figure 3.11: An example of the train and test error over time per one complete cycle of fusion GRU network training on the most restrictive fusion dataset.

The GRU version of the fusion neural network took 10 epochs to train per each fusion dataset. Each training epoch took 5 seconds over 250 batches, and each evaluation cycle took 0.7 seconds on average. Plot 3.12 showcases the classification error over time for a train/test trial of the LSTM version of the fusion neural network.

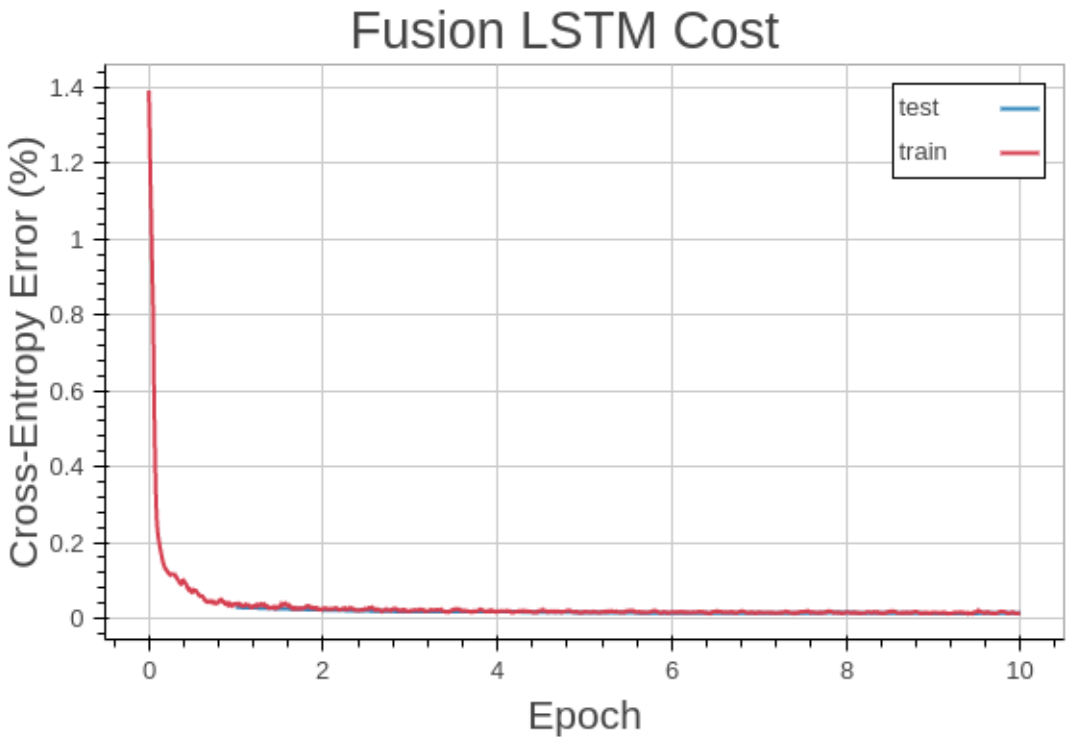

Figure 3.12: An example of the train and test error over time per one complete cycle of fusion LSTM network training on the most restrictive fusion dataset.

The LSTM version of the fusion neural network took 10 epochs to train per each fusion dataset. Each training epoch took 6 seconds over 285 batches, and each evaluation cycle took 0.8 seconds on average.

3.6 Sensor Fusion Neural Network Validation Results

Four different figures follow which showcase the validation performance of the four fusion neural network variations on the state-of-the-art sensor fusion system dataset. The networks were validated with the accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro, and ROC AUC macro metrics along 9 bootstrapping trials of random train/test data selection. For none of the evaluations were the neural networks exposed to the test dataset during the training cycle. Validation

results for each of the four fusion network variations trained on the generated fusion datasets based on the hand-engineered fusion algorithm described in section 2.6.4 are included in the appendices. The neural network validation figures in the appendix are divided into sections based on the fusion dataset types outlined in table 3.4.

3.6.1 State-of-the-art Fusion Dataset Validation Results

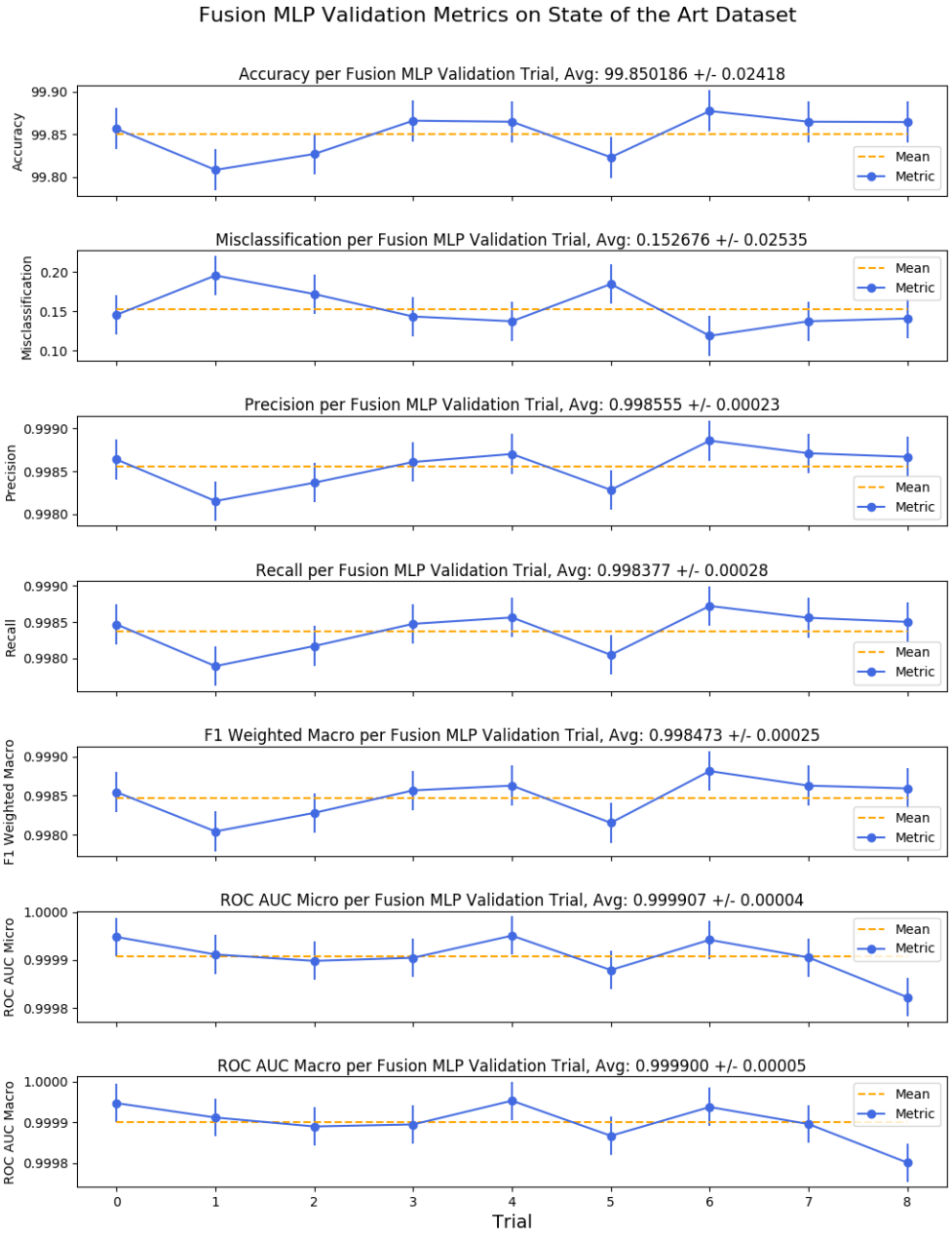

Figure 3.13: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the MLP fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the state-of-the-art fusion dataset. The average score and its standard deviation are included in the title of each subplot.

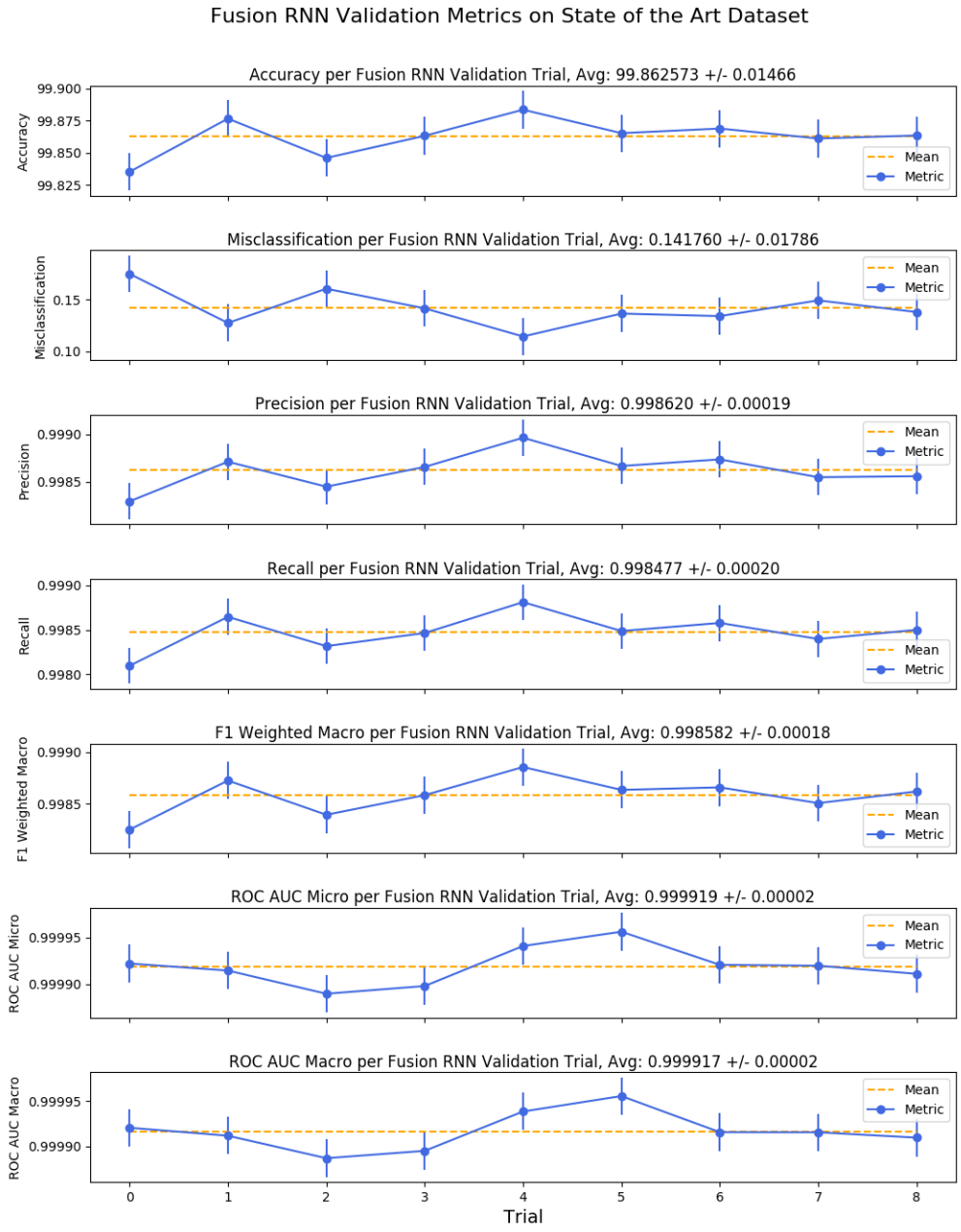

Figure 3.14: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the RNN fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the state-of-the-art fusion dataset. The average score and its standard deviation are included in the title of each subplot.

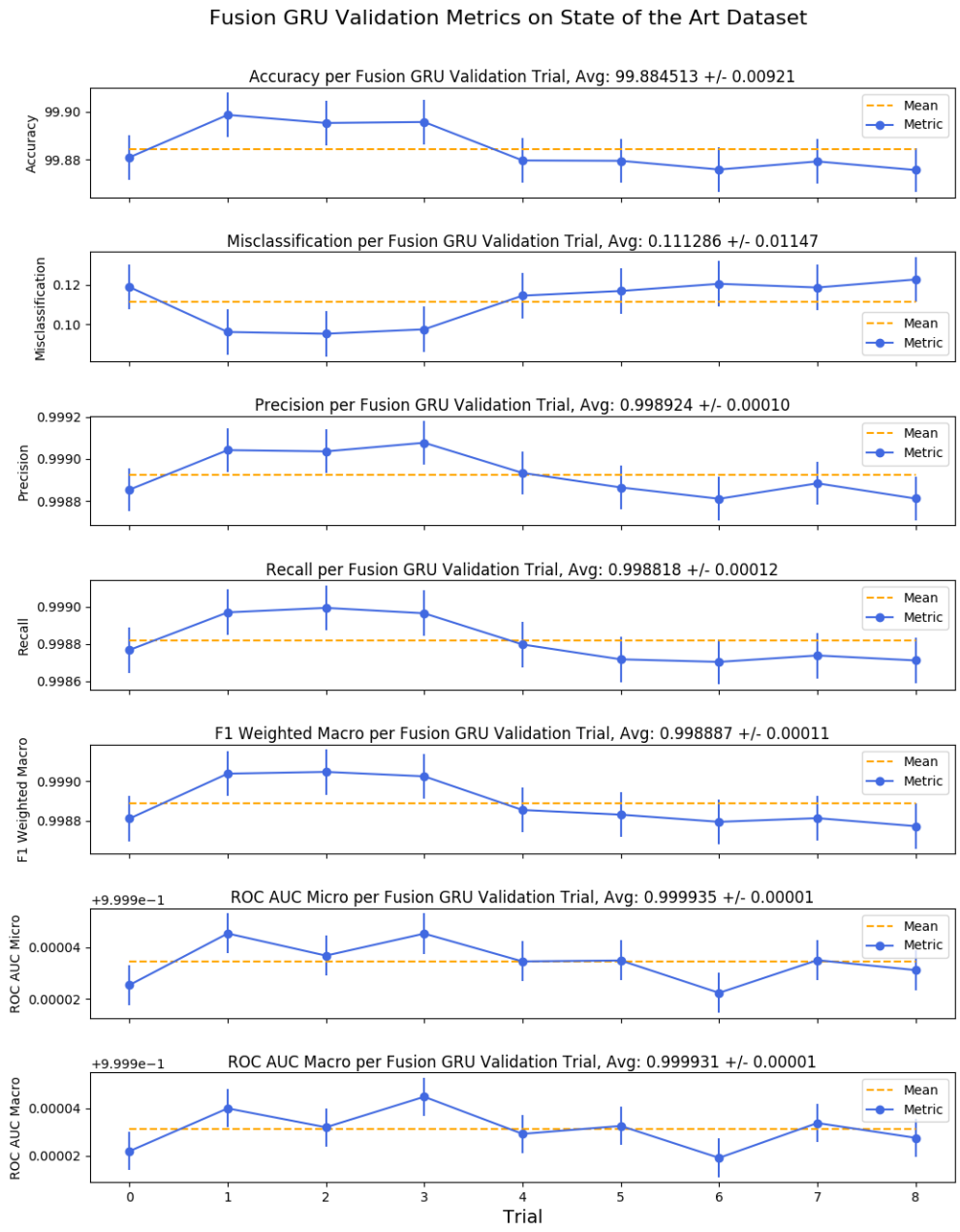

Figure 3.15: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the GRU fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the state-of-the-art fusion dataset. The average score and its standard deviation are included in the title of each subplot.

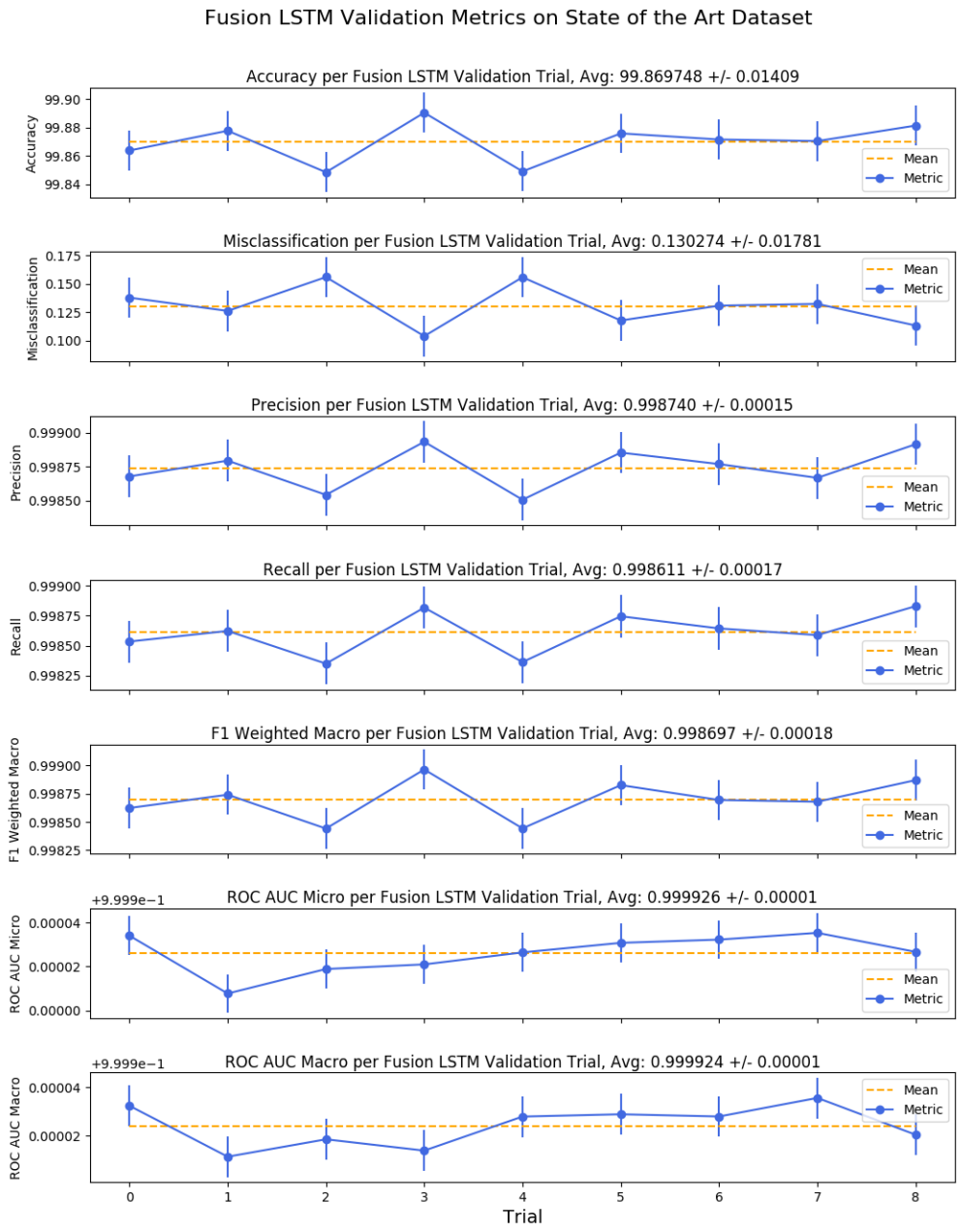

Figure 3.16: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the LSTM fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the state-of-the-art fusion dataset. The average score and its standard deviation are included in the title of each subplot.

3.6.2 Sensor Fusion Neural Network Precision and Recall Validation Results

Following are sample precision and recall curve plots including micro-averaged precision and recall curves for each of the fusion neural network versions. Only samples are included for brevity. These plots echo samples of the information in the previous figures but are still worthy to visualize. Additionally, the plots per network version were very similar per dataset type.

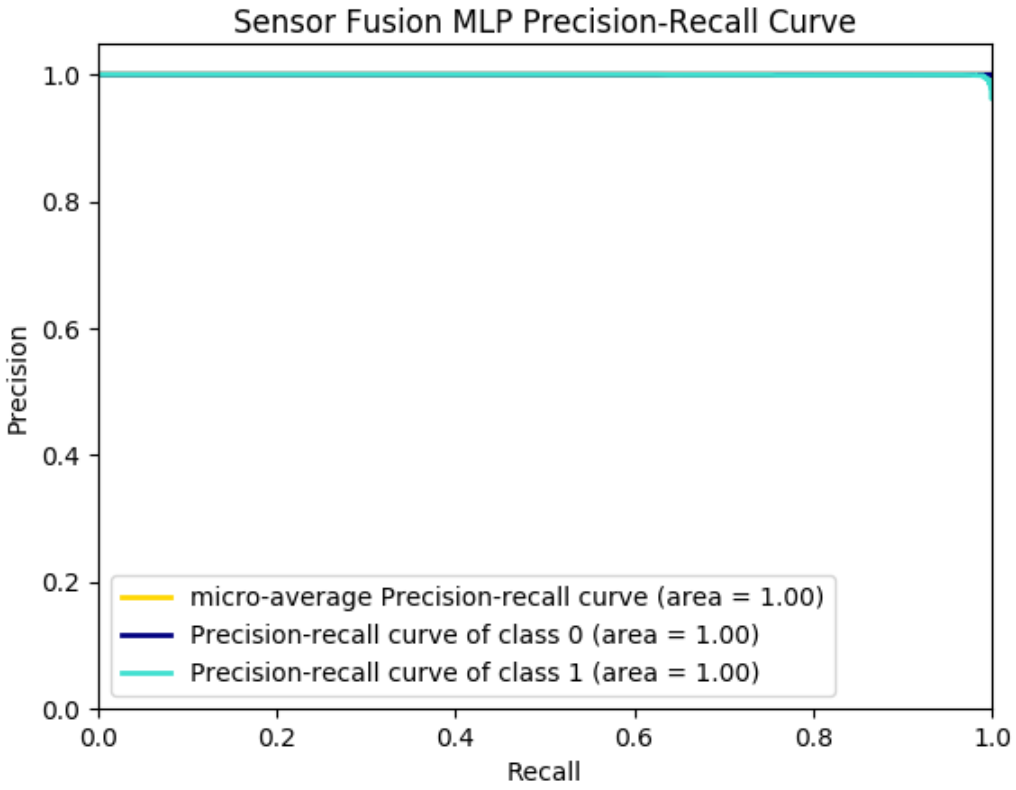

Figure 3.17: Sample precision, recall and micro-averaged precision and recall curves for the fusion MLP network.

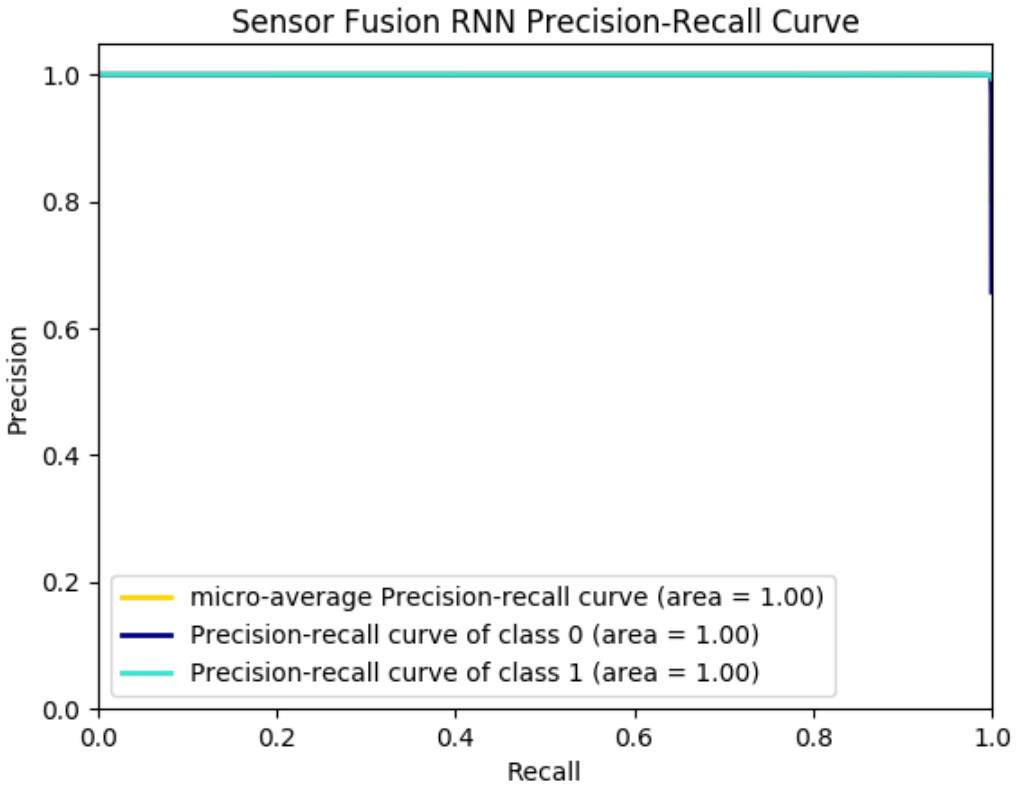

Figure 3.18: Sample precision, recall and micro-averaged precision and recall curves for the fusion RNN network.

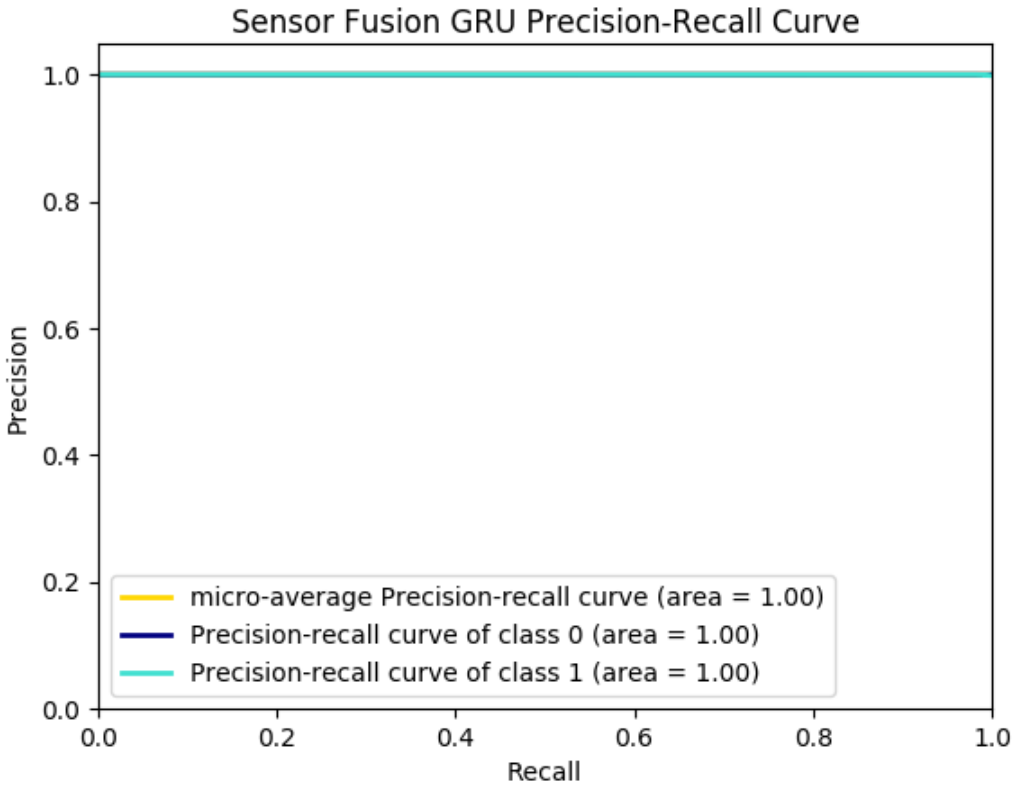

Figure 3.19: Sample precision, recall and micro-averaged precision and recall curves for the fusion GRU network.

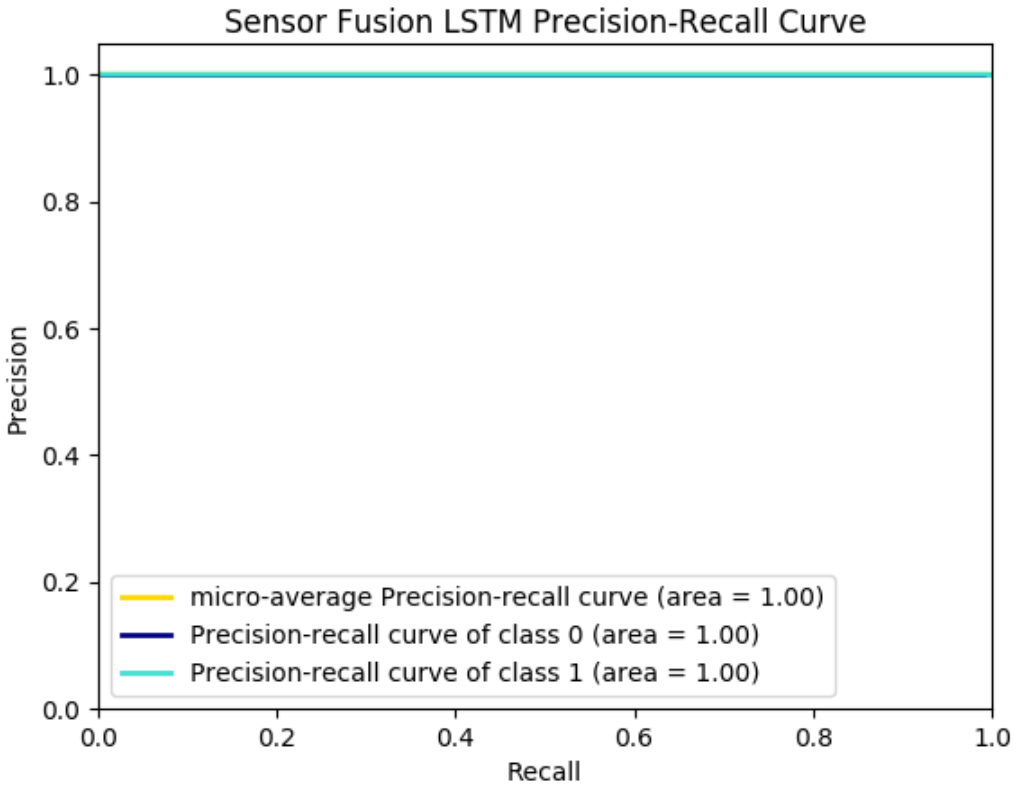

Figure 3.20: Sample precision, recall and micro-averaged precision and recall curves for the fusion LSTM network.

3.7 Deep Learning Sensor Fusion System Evaluation

3.7.1 Agreement between Deep Learning Fusion System and State-of-the-art Fusion System

The agreement between the deep learning sensor fusion system and the state-of-the-art fusion system outputs was evaluated. The deep learning sensor fusion system with both consistency and fusion neural networks was used for the evaluations. The LSTM version of the consistency neural network was combined with each of the four versions of the fusion neural network trained on the state-of-the-art fusion system

labels for each separate evaluation. The outputs between the state-of-the-art fusion system and deep learning fusion systems were compared in order to determine the percentage agreement between the two systems.

The confusion matrices that follow correspond to the fusion results between the two most populated radar tracks with multiple camera objects over 1 hour and 15 minutes of never-before-seen test sensor data. Over the course of that time period, approximately 542 unique cars were visible to both radar and camera sensors. In each confusion matrix, the diagonal elements imply agreement between the two fusion systems while the off-diagonal elements imply disagreement between the two fusion systems. An ideal agreement between the two would be a diagonal matrix where the diagonal elements sum to 1. A template confusion matrix is shown in figure 3.5.

Table 3.5: Confusion Matrix Template Between Deep Learning Fusion System and State-of-the-art (SOA) Fusion System.

	NN Fused	NN Non-Fused
SOA Fused	% True Positive	% False Negative
SOA Non-Fused	% False Positive	% True Negative

The upper left cell in each matrix represents the percentage of true positive fusion agreements between both fusion systems, shown by example in figure 3.21. The bottom left cell of each matrix represents the percentage of false positive fusion disagreements, as depicted in figure 3.22. In these cases, the state-of-the-art fusion system found fusion but the deep learning fusion system did not. The bottom right cell of each matrix represents the percentage of true negative fusion agreements, where neither system found fusion. This occurrence is shown by example in figure 3.23. Lastly, the upper right cell in each matrix represents the percentage of false negative fusion disagreements between the two fusion systems, shown by example in figure 3.24. In these cases, the deep learning fusion system found fusion but the state-of-the-art fusion system did not.

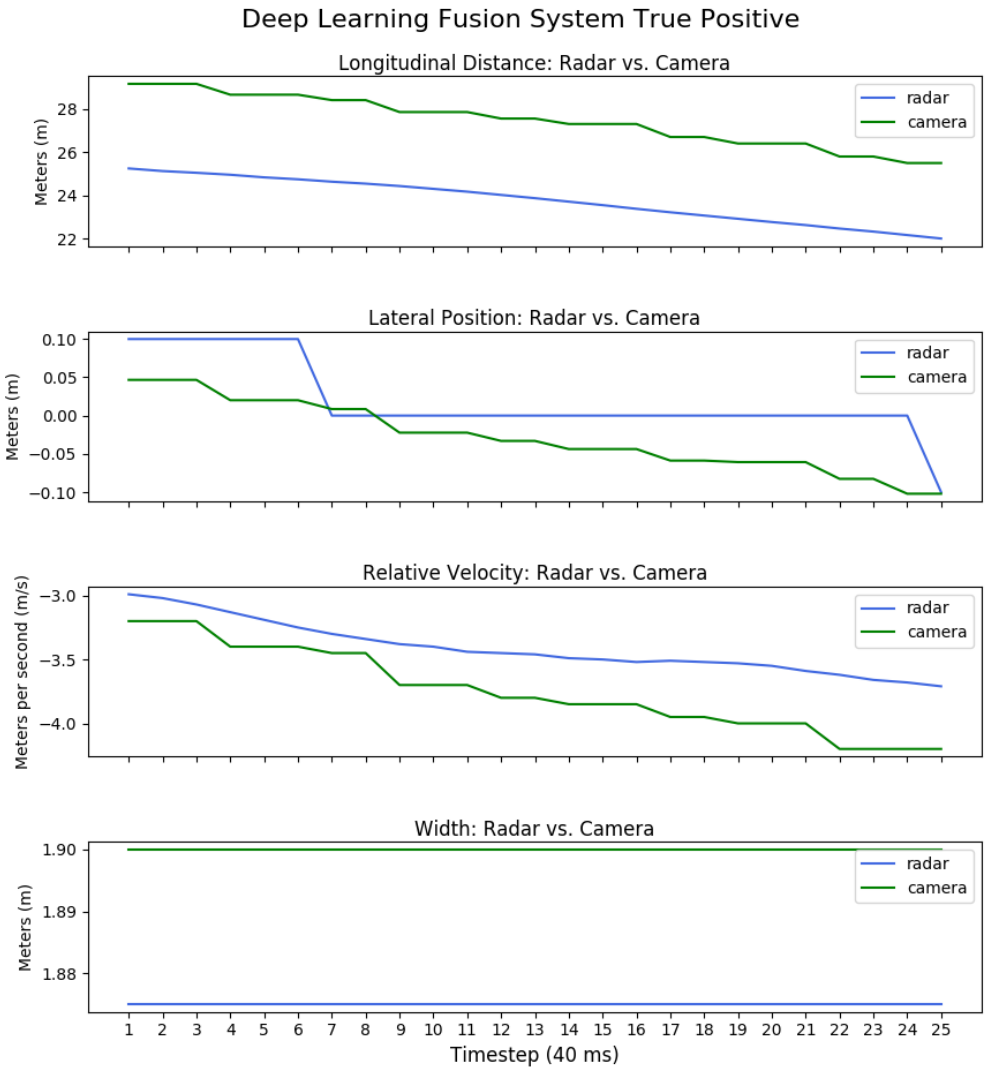

Figure 3.21: A sample true positive fusion agreement between the deep learning sensor fusion system and the state-of-the-art sensor fusion system over a one-second period. The fused radar and camera track streams are plotted against each other to demonstrate the fusion correlation.

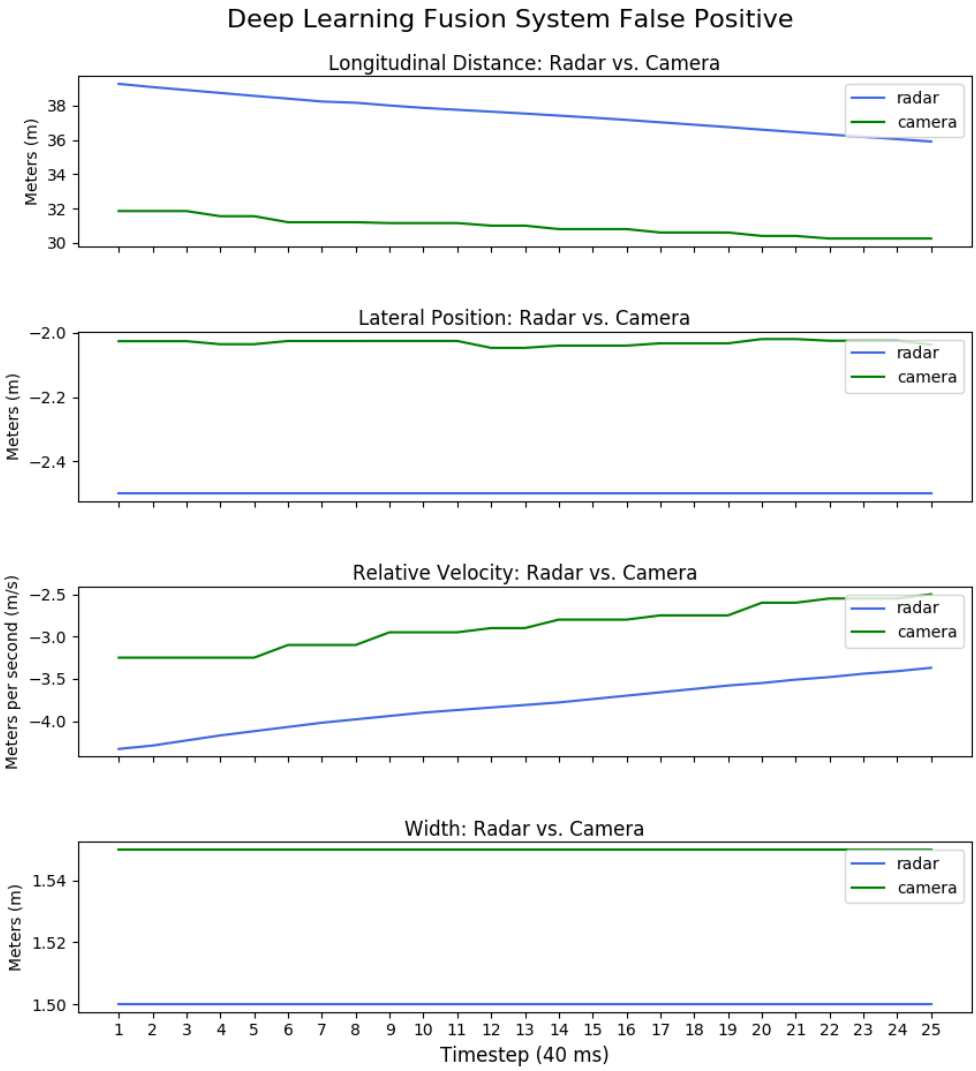

Figure 3.22: A sample false positive fusion disagreement between the deep learning sensor fusion system and the state-of-the-art sensor fusion system over a one-second period. The radar and camera track streams are plotted against each other to demonstrate the potential fusion correlation where the systems disagreed.

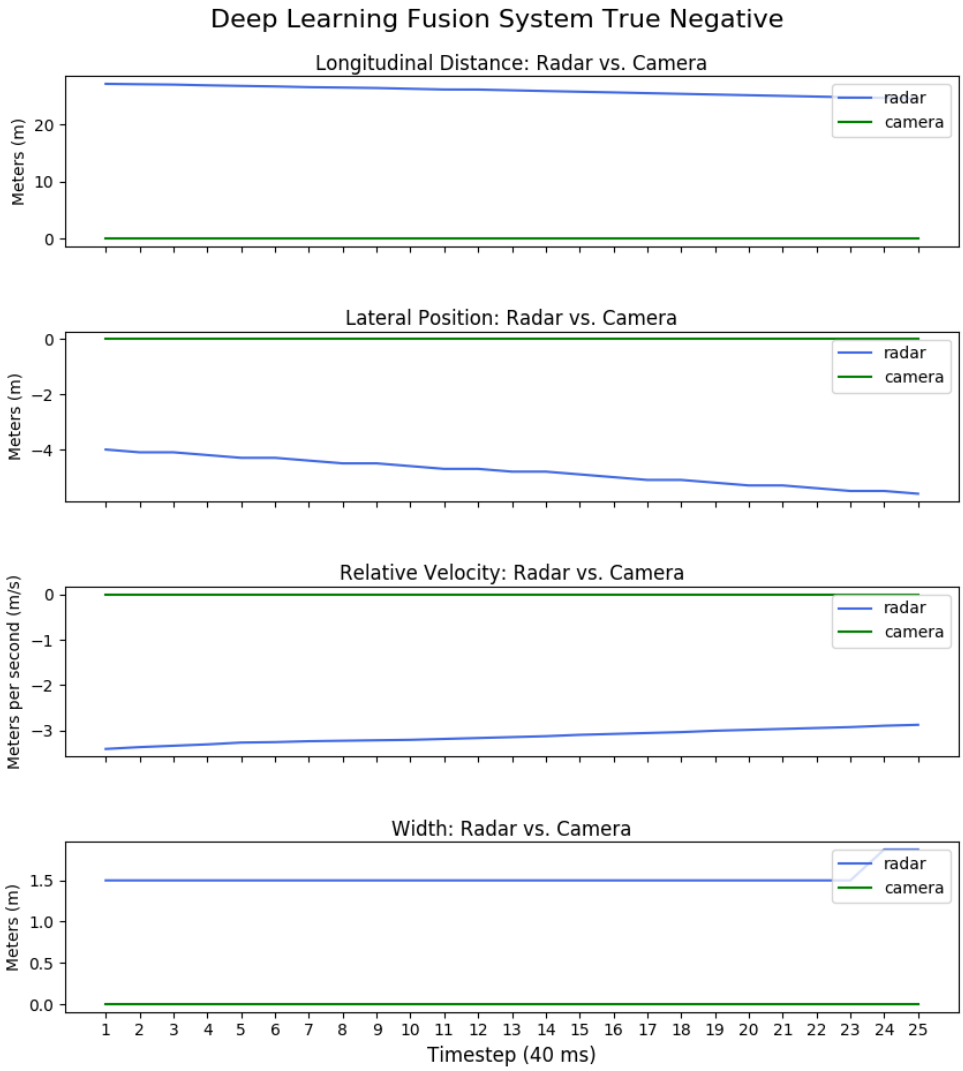

Figure 3.23: A sample true negative fusion agreement between the deep learning sensor fusion system and the state-of-the-art sensor fusion system over a one-second period. The non-fused radar and camera track streams are plotted against each other to demonstrate the absence of valid data in the camera track while the radar track is consistently populated thus leading to non-fusion.

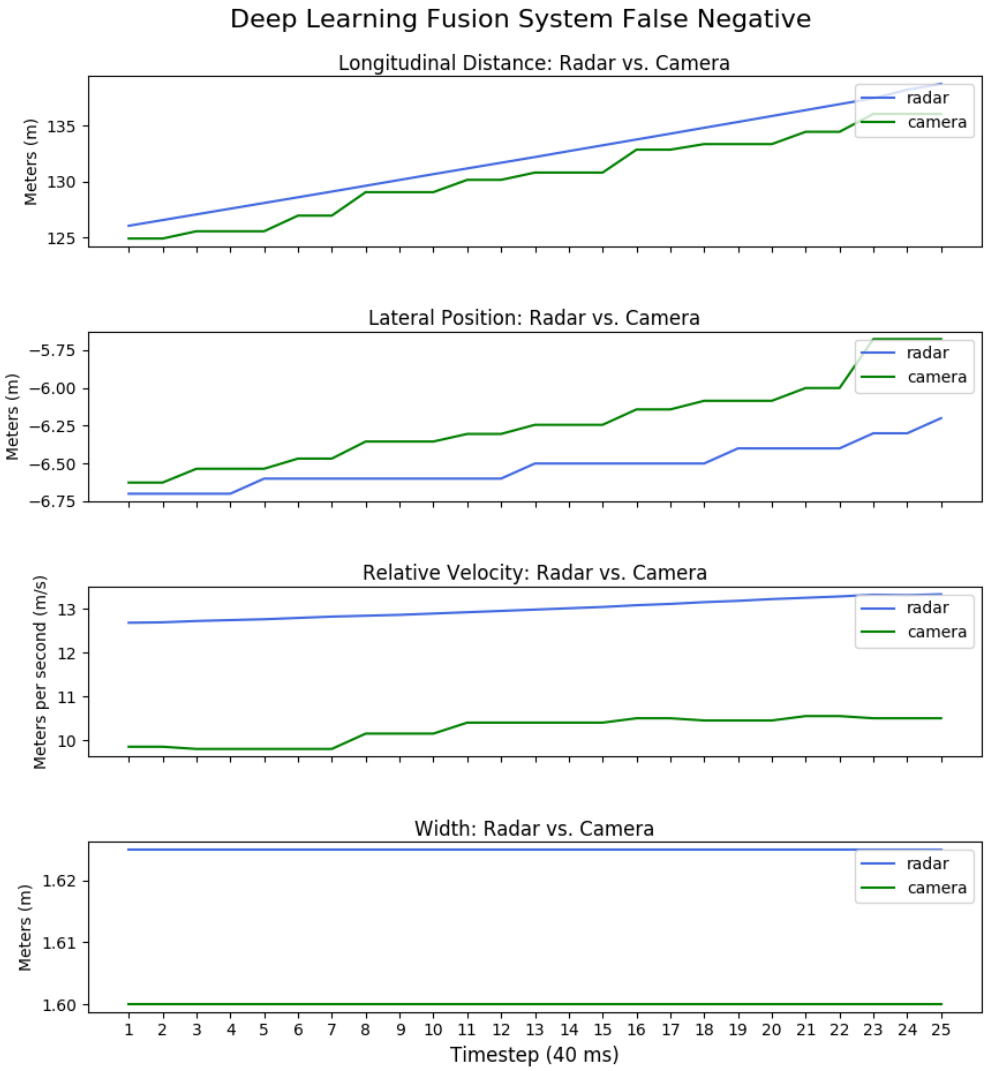

Figure 3.24: A sample false negative fusion disagreement between the deep learning sensor fusion system and the state-of-the-art sensor fusion system over a one-second period. The radar and camera track streams are plotted against each other to demonstrate the potential fusion correlation where the systems disagreed.

The confusion matrices for the fusion MLP are shown in tables 3.6 and 3.7. The first matrix showcases a 95.08% agreement between the deep learning fusion system and the state-of-the-art fusion system for the most populated radar track. The second matrix showcases a 93.88% agreement between the two fusion systems for the second most populated track. The agreement per each matrix was determined by summing the diagonal elements of the matrix.

Table 3.6: Confusion Matrix Between Deep Learning Fusion System with Fusion MLP and State-of-the-art (SOA) Fusion System on Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	53.6% True Positive	3.35% False Negative
SOA Non-Fused	1.57% False Positive	41.48% True Negative

Table 3.7: Confusion Matrix Between Deep Learning Fusion System with Fusion MLP and State-of-the-art (SOA) Fusion System on Second Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	24.88% True Positive	3.7% False Negative
SOA Non-Fused	2.42% False Positive	69% True Negative

The confusion matrices for the fusion RNN are shown in tables 3.8 and 3.9. The first matrix showcases a 95.22% agreement between the deep learning fusion system and the state-of-the-art fusion system for the most populated radar track. The second matrix showcases a 94.1% agreement between the two fusion systems for the second most populated track.

Table 3.8: Confusion Matrix Between Deep Learning Fusion System with Fusion RNN and State-of-the-art (SOA) Fusion System on Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	53.68% True Positive	3.27% False Negative
SOA Non-Fused	1.5% False Positive	41.54% True Negative

Table 3.9: Confusion Matrix Between Deep Learning Fusion System with Fusion RNN and State-of-the-art (SOA) Fusion System on Second Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	24.75% True Positive	3.84% False Negative
SOA Non-Fused	2.07% False Positive	69.35% True Negative

The confusion matrices for the fusion GRU are shown in tables 3.10 and 3.11. The first matrix showcases a 95.6% agreement between the deep learning fusion system and the state-of-the-art fusion system for the most populated radar track. The second matrix showcases a 94.39% agreement between the two fusion systems for the second most populated radar track.

Table 3.10: Confusion Matrix Between Deep Learning Fusion System with Fusion GRU and State-of-the-art (SOA) Fusion System on Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	53.69% True Positive	3.26% False Negative
SOA Non-Fused	1.13% False Positive	41.91% True Negative

Table 3.11: Confusion Matrix Between Deep Learning Fusion System with Fusion GRU and State-of-the-art (SOA) Fusion System on Second Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	24.47% True Positive	4.12% False Negative
SOA Non-Fused	1.49% False Positive	69.92% True Negative

The confusion matrices for the fusion LSTM are shown in tables 3.12 and 3.13. The first matrix showcases a 95.2% agreement between the deep learning fusion system and the state-of-the-art fusion system for the most populated radar track. The second matrix showcases a 94.13% agreement between the two fusion systems for the second most populated radar track.

Table 3.12: Confusion Matrix Between Deep Learning Fusion System with Fusion LSTM and State-of-the-art (SOA) Fusion System on Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	53.56% True Positive	3.39% False Negative
SOA Non-Fused	1.41% False Positive	41.64% True Negative

Table 3.13: Confusion Matrix Between Deep Learning Fusion System with Fusion LSTM and State-of-the-art (SOA) Fusion System on Second Most Populated Radar Track.

	NN Fused	NN Non-Fused
SOA Fused	24.6% True Positive	3.98% False Negative
SOA Non-Fused	1.89% False Positive	69.53% True Negative

3.8 Deep Learning Sensor Fusion System Runtime Results

The deep learning sensor fusion system was run on multiple computer systems to evaluate the response of the system in terms of real-time updates, which were provided at a 40 ms sample rate. The fusion system was evaluated with the LSTM versions of both consistency and fusion neural networks because these were inherently the slowest deep neural networks constructed given their complexity. The two notable computer systems that the deep learning fusion system was evaluated on were the following:

- **Deep Learning PC** with the following specifications:
  - Intel i7 6-core CPU running at 3.5 GHz
  - 32 GB DDR4 RAM
  - 256 GB Solid State Drive
  - NVIDIA GTX 980Ti 6GB GPU
  - OS: Ubuntu Linux 16.04 with CUDA 8 and CuBLAS
- **RaspberryPi 3** embedded system with the following specifications:
  - 1.2GHz 64-bit quad-core ARMv8 CPU
  - 1GB DDR2 RAM

- 32GB Extreme Speed Micro SD card
- OS: Ubuntu MATE 16.04

On the deep learning PC, the fusion system was evaluated using both CPU and GPU, while on the embedded system, the fusion system was evaluated using only CPU. The runtime results using CPU on the deep learning PC with multiple configurations involving serial and parallel computing with batch mode were as stated in table 3.14. The runtime results using CPU and GPU on the deep learning PC with multiple configurations in serial, parallel and batch mode were as stated in table 3.15. The runtime results using CPU on the embedded system with multiple configurations in both serial and parallel were as stated in table 3.16.

Table 3.14: Deep Learning Fusion System Runtime on Intel CPU with Serial, Parallel and Batch Processing.

Version	Mean (s)	Std. Dev. (s)
Serial	0.347	0.008
Batch	0.106	0.01
Multi-thread	0.577	0.118
Multi-process	0.113	0.021

Table 3.15: Deep Learning Fusion System Runtime on with Intel CPU and NVIDIA GPU with Serial, Parallel and Batch Processing.

Version	Mean (s)	Std. Dev. (s)
Serial GPU	0.678	0.016
Batch GPU	0.081	0.01
Multi-thread GPU	0.715	0.152
Multi-process w/ Batch GPU	0.034	0.01

Table 3.16: Deep Learning Fusion System Runtime on RaspberryPi 3 CPU with Serial, Parallel and Batch Processing.

Version	Mean (s)	Std. Dev. (s)
Serial	3.23	0.42
Batch	0.68	0.014
Multi-thread	2.14	0.52
Multi-process	1.38	0.242

Chapter 4

Discussion

4.1 Sensor Data Consistency and Correlation

The sample correlation metrics computed between each of the four streams of consistent, fused sensor tracks according to the state-of-the-art fusion system in section 3.1, reveal that the correlation is strong between two camera and radar sensor tracks when both tracks are consistent. Between all four streams, the most reliable streams for fusion are the longitudinal distance, lateral position, and width due to their strong correlations between both sensors, while the relative velocity had the weakest correlation between both sensors. The stream rankings in terms of correlation strength from strongest to weakest with the r^2-metric are the following:

1. Lateral Position, 0.9968
2. Longitudinal Distance, 0.9911
3. Width, 0.986
4. Relative Velocity, 0.968

These rankings reveal the reliability order with which a dependent variable, such as the fusion between two sensor tracks, could be predicted from any of the four se-

lected streams based on the proportion of variance found between the fusion results of the state-of-the-art fusion system. The lateral position was revealed to be the most reliable stream for predicting a dependent variable based on its variance. The longitudinal distance was second in terms of reliable prediction based on its variance, off by a marginal 0.0057 from the lateral position in terms of the correlation metric. The width was shown to be less reliable than the previous two streams under this metric, and the relative velocity was the least reliable stream for predicting a dependent variable based on its proportion of variance according to this metric. Despite the comparison shown, all the streams utilized for fusion were very correlative when fused as compared to other potential data streams acquired from the sensors.

The stream rankings in terms of correlation strength from strongest to weakest with the normalized cross correlation metric where the signals completely overlapped are the following:

1. Width, 0.9998
2. Longitudinal Distance, 0.9992
3. Lateral Position, 0.9968
4. Relative Velocity, 0.9804

These rankings reveal the reliability order of the streams upon which a displacement-based comparison strategy, such as a convolution, could predict fusion between the two sensors based on the fusion results from the state-of-the-art fusion system. The width was revealed to be the most reliable stream for fusing the two sensor tracks based on a correlation of displacement. This expresses that the width stream provides the most reliable displacement correlation between the two sensors, and when both width streams have strong correlation with direct overlap, there is a higher chance that the compared sensor tracks are fused. The longitudinal distance was second to

the width in terms of reliability based on displacement correlation. The difference between the two with this metric was a mere $6 * 10^{-4}$. This reveals that both the width and longitudinal distance are strong indicators of fusion between the two sensors. The lateral position ranked third in reliability for correlation as a function of displacement. The top three streams for this metric were inherently close in correlation when compared directly against each other where the signals overlapped. The relative velocity, as with the r^2-metric, ranked last in terms of reliability based on displacement correlation. It has been shown according to the two reviewed correlation metrics that the relative velocity has the weakest correlation between both sensors in the state-of-the-art fusion system results.

The stream rankings in terms of 1-D Euclidean distance median and standard deviation from smallest to largest are the following:

1. Width, 0.025 +/- 0.018 m
2. Lateral Position, 0.077 + 0.199 m
3. Relative Velocity, 0.27 + 0.47 m/s
4. Longitudinal Distance, 4.96 +/- 3.554 m

These rankings reveal that the width is the strongest indicator of when two sensor tracks are fused between radar and camera based on the distance between the two track streams. The width shows strong agreement with minimal error and deviation between the two sensors. The rankings also reveal that the longitudinal distance has a larger error tolerance and variance than all other streams between the radar and camera when any two tracks are fused. This larger error tolerance may be due to many factors involving the sensors including sensor position, calibration, and approximation of distance. The camera must approximate the distance of any given vehicle using various algorithms, and this distance approximation may be the cause of a larger

acceptable error between longitudinal distance streams of both sensors. The larger error tolerance in the longitudinal distance is, however, proportionate to the standard deviation in the error. As shown in previous correlation rankings, the lateral position still requires a minimal error tolerance between both sensors while the relative velocity requires a higher error tolerance. Based on this ranking order, the two streams that agree with minimal error between the two sensors are the width and lateral position while the relative velocity and longitudinal distance streams must have a higher error tolerance between the two sensors due to sensor-specific reasons.

The computed correlation results uphold the ideas that the four selected sensor streams, when consistent, are highly correlative between the radar and camera sensors according to various correlation metrics. These streams thus proved to be very useful for the determination of sensor fusion once they were rearranged or masked to be consistent with no track reassignments.

4.2 Sensor Consistency Neural Network Training Results

The train/test error versus time results showcased in section 3.2.1 reveal the convergence of each consistency neural network to minimal error using the RMSProp training algorithm with the RSS cost metric. Each network took a different number of epochs to reach minimal regression error. The network versions are ranked in terms of training time in epochs from least to greatest below:

1. Multilayer Perceptron, 100 epochs
2. Gated Recurrent Unit, 100 epochs
3. Recurrent, 150 epochs
4. Long-short Term Memory, 200 epochs

Interestingly, the MLP-based consistency neural network took the least amount of time to train. The GRU-based network was tied with the MLP in terms of training epoch time, but the errors observed over time in GRU consistency training error plot in figure 3.3 were greater than the errors observed in the MLP training plot in figure 3.1. An interesting phenomenon took place in the GRU training error plot where there was a large sum-squared error for train and test samples until epoch 45, where the error dropped below half of what it was spontaneously. This sudden decrease in error was not something that took place in any of the other consistency training error plots, but it could have been based on a decay in the learning rate due to the RMSProp optimization strategy. The RNN-based network ranked third in training time. Surprisingly, the LSTM-based neural network took the longest amount of time to train. Typically, the RNN-based network would take the most time to train due to its lack of gated logic to modulate data flow, but in this case the LSTM network took a longer time to train.

The wall-clock time required per training and evaluation epoch grew as the consistency network complexity increased as described in section 3.2.1. The required times per training epoch over 85 batches are ranked from least to greatest below:

1. Multilayer Perceptron, 1.05 seconds per epoch
2. Recurrent, 1.2 seconds per epoch
3. Gated Recurrent Unit, 1.8 seconds per epoch
4. Long-Short Term Memory, 1.8 seconds per epoch

Based on the wall-clock time ranking list, it is shown that the MLP-based network takes the least amount of time for forward and backward propagation while the LSTM-based network takes the most time for said propagations. These results support the idea that the increased network complexity incurs additional runtime overhead.

These results demonstrate that the consistency neural networks were able to learn when data was consistent, inconsistent, or unknown quickly with minimal error. The observation that each training epoch took longer as the networks grew more complex shows that the additional network complexity incurred extra overhead runtime for forward and backward propagation of input samples.

4.3 Sensor Consistency Neural Network Validation Results

The consistency neural networks were evaluated over 9 bootstrapping trials on the test datasets with three regression validation metrics including the MSE, RMSE, and r^2-metric. The validation results are shown in section 3.3. The mean validation results for each network are summarized in table 4.1.

Table 4.1: Consistency Neural Networks Validation Results

Version	MSE	RMSE	r^2-metric
MLP	0.0319	0.1775	0.9714
RNN	0.033	0.182	0.9643
GRU	0.0276	0.1656	0.9755
LSTM	0.0281	0.167	0.973

According to the table, the best performing network across all 9 bootstrapping trials was the GRU-based solution. The LSTM network was very close in comparison to the GRU, and the difference in MSE was a mere $5 * 10^{-4}$. The MLP network was the third best performer, and the RNN-based solution was the worst performer. All networks were able to provide substantial generalization power for the consistency sequence regression problem based on the four synchronized input streams of radar track data.

4.4 Sensor Fusion Neural Network Training Results

The train/test error versus time results showcased in section 3.5.1 display quick convergence to minimal classification error for each trained fusion network version using the AdaGrad optimization algorithm with the cross-entropy binary cost metric. Each network took 10 epochs to fully train. The train and test errors in all plots looked very similar. The train error was able to grow very small after only one epoch of training for each network. The next 9 epochs were useful for learning edge cases that were not absorbed in the first epoch. The test error was small for all versions of the network throughout all training epochs. The train and test errors for each network were able to converge to the range [0, 0.01]. The large batch size and number of samples were most likely large contributors to the speed of the learning process.

The wall-clock time required per training and evaluation epoch grew as the fusion network complexity increased as described in section 3.5.1. The required times per training epoch over a range of batch sizes from 250 to 285 batches are ranked from least to greatest below:

1. Multilayer Perceptron, 3.4 seconds per epoch
2. Recurrent, 3.5 seconds per epoch
3. Gated Recurrent Unit, 5 seconds per epoch
4. Long-Short Term Memory, 6 seconds per epoch

Based on the wall-clock time ranking list, it is shown that the MLP-based network takes the least amount of time for forward and backward propagation while the LSTM-based network takes the most time for said propagations. These results support the idea that the increased network complexity incurs additional runtime overhead.

These results support the application of the designed fusion neural network structure for classifying fusion of the four streams of absolute differences between two radar and camera tracks. The networks were able to learn the complex nature of the fusion scenarios labeled according to the state-of-the-art sensor fusion system quickly with minimal classification error. These networks also learned the nature of the hand-engineered fusion label generation algorithm just as fast with just as minimal error. These similar results across all fusion datasets provide substantial evidence that all designed versions of the fusion neural networks are well-suited for learning and providing the complex capabilities of the preexisting state-of-the-art fusion system as well as other alternative sensor fusion algorithms. The observation that each training epoch took longer as the networks grew more complex shows that the additional network complexity incurred extra overhead runtime for forward and backward propagation of input samples.

4.5 Sensor Fusion Neural Network Validation Results

The fusion neural networks were evaluated over 9 bootstrapping trials on the test datasets with seven classification validation metrics including accuracy (ACC), misclassification (1-ACC), precision (PPV), recall (TPR), F1 weight macro (F1), ROC AUC micro (ROC_micro), and ROC AUC macro (ROC_macro). The validation results are shown in section 3.6. The mean validation results for each network trained and evaluated on the state-of-the-art sensor fusion system dataset, as acquired from 3.6.1, are summarized in table 4.2.

According to the table, the best performing network across all 9 bootstrapping trials was the GRU-based solution. The LSTM network was very close in comparison to the GRU, and the difference in precision was a mere $2 * 10^{-4}$ while the difference

Table 4.2: Sensor Fusion Neural Networks Validation Results Based on State-of-the-art Fusion Dataset

Version	ACC	1-ACC	PPV	TPR	F1	ROC_micro	ROC_macro
MLP	99.85	0.15	0.9985	0.9984	0.9985	0.9999	0.9999
RNN	99.86	0.14	0.9986	0.9985	0.9986	0.9999	0.9999
GRU	99.88	0.12	0.9989	0.9989	0.9989	0.9999	0.9999
LSTM	99.87	0.13	0.9987	0.9986	0.9987	0.9999	0.9999

in recall was a mere $3 * 10^{-4}$. The RNN network was the third best performer, and the MLP-based solution was the worst performer. All networks were able to provide substantial generalization power for the sensor fusion classification problem based on the absolute differences between the four synchronized input streams of camera and radar track data. These networks were shown to perform exceptionally well on fusion datasets based on the state-of-the-art sensor fusion system. The precision and recall plots in section 3.6.2 show that all fusion networks were able to achieve near perfect precision and recall on the fusion test datasets, notably those based on the state-of-the-art fusion system.

4.6 Agreement between Deep Learning Fusion System and State-of-the-art Fusion System

The agreement and confusion of the deep learning sensor fusion system as compared to the state-of-the-art sensor fusion system was evaluated in section 3.7.1. Examples of the four possible types of agreement and disagreement were shown. The confusion matrices for the fusion results between both systems were computed for the two most populated radar tracks over the course of 1.25 hours of test data. Four variants of the deep learning fusion system were evaluated against the state-of-the-art fusion system for these scenarios with the consistency LSTM held constant and each of the four fusion network versions variated. The fusion agreement percentages between the constructed and preexisting fusion systems for all four variants of the deep learning fusion

system along with the corresponding average agreement percentages are summarized in table 4.3.

Table 4.3: Agreement Percentages Between Deep Learning Fusion System and State-of-the-art Fusion System for Two Most Populated Radar Tracks (RT0) and (RT1)

Fusion Network	RT0 Agreement	RT1 Agreement	Avg. Agreement
MLP	95.08%	93.88%	94.48%
RNN	95.22%	94.1%	94.66%
GRU	95.6%	94.39%	95%
LSTM	95.2%	94.13%	94.67%

According to the table, the GRU-based fusion neural network performed the best out of all versions applied in the deep learning fusion system. The MLP-based fusion neural network performed the worst of all versions applied in the fusion system. Surprisingly, the LSTM and the RNN average agreement percentages differed by a mere 0.01%, with the LSTM performing slightly better than the RNN. The results reveal how closely the deep learning fusion system agreed with the state-of-the-art sensor fusion system no matter the fusion neural network version applied. It was shown that overall, all tested variants of the deep learning fusion system agreed with the state-of-the-art fusion system at least 94.4% of the time.

Based on the agreement percentages, there was a disagreement percentage of approximately 5% between the deep learning and state-of-the-art sensor fusion systems. Examples of false positive and false negative fusion disagreements between the systems are visualized in figures 3.22 and 3.24, respectively. The visualized examples were primary cases where the systems disagreed. Cases like those shown were hard to distinguish and could be clarified using video data captured by the camera in order to truly determine if the two sensors were fused during those disagreement occurrences.

4.7 Deep Learning Sensor Fusion System Runtime

The deep learning sensor fusion system runtimes were benchmarked for multiple configurations of the system with serial, batch and parallel processing across multiple hardware architectures including Intel CPU, NVIDIA GPU and ARMv8 CPU. The results were shown in section 3.8.

Of all deep learning fusion system configurations with serial, batch and parallel processing, only one notable combination of parallelization techniques led the system to a real-time response update rate of 40 ms on the deep learning PC. The winning runtime system configuration involved parallelizing the consistency neural networks over 10 processes, per the 10 radar tracks, and evaluating the fusion neural network in a single process with a batch size of 100 samples on the NVIDIA GPU, per the 100 possible radar and camera track combinations. This successful result was shown in the last row of table 3.15. This result reveals that the deep learning fusion system can respond to real-time sensor updates in the proper parallelization configuration on a computer with capable hardware.

The slowest runtime result benchmarked on the deep learning PC involved the configuration with multi-threaded GPU evaluation of the deep neural networks. This result was shown in the second-to-last row of table 3.15. This result reveals the slowdown involved in using the GPU for small tasks such as consistency neural network evaluation as well as the overhead runtime entailed by the GIL in the Python threading interface. Table 3.14 also supports this claim because the multi-threaded fusion system on the CPU performed the worst of all CPU configurations on the deep learning PC as well.

The slowest runtime results were benchmarked on the RaspberryPi 3 with its ARMv8 CPU. For this embedded system architecture, the fastest runtime of 0.68 seconds per update was found using batch processing for both neural networks. On this system, there were many factors that may have contributed to the slow per-

formance observed including processor speed, RAM speed and amount, and lack of optimized BLAS libraries to enhance linear algebra performance. An additional factor in this slow performance was the lack of a GPU to parallelize multiple neural network computations.

These results support the idea that parallel computing with processes in Python, coupled with the use of CUDA and CuBLAS with batch GPU neural network evaluation over many samples, is necessary in order to achieve high-speed, real-time worthy deep neural network computations. It was demonstrated that the GPU is not always useful for small tasks, but it is very useful for large, equally-distributed tasks. The runtime results acquired with the GPU revealed that evaluating the consistency neural network on the GPU was detrimental to the runtime of the fusion system because the number of samples was small and the overhead of transferring data back and forth from GPU took more time than necessary. However, evaluating the fusion neural network in batch mode on the GPU was beneficial for the runtime of the fusion system because the number of samples was large and the neural network evaluation tasks were equally-distributed.

Chapter 5

Conclusion

Based on the consistency neural network training and validation results, it was observed that all four versions of the consistency neural network performed well on the dataset generated by the consistency labeling algorithm that was developed. These networks were each able to learn to predict the consistency of any 1-second sample of 4 synchronized radar track streams with minimal error. Based on the fusion neural network training and validation results, it was observed that all four versions of the fusion neural network performed well across five different sensor fusion datasets, including a dataset generated based on fusion labels from a preexisting state-of-the-art sensor fusion system. These networks were each able to classify 1-second samples of fused camera and radar track pairs of the absolute differences of any 4 synchronized streams between the two sensors with high accuracy, precision, and recall. The greater the complexity of either deep neural network, the longer it took for forward and backward propagation of any input samples.

According to the results, the consistency neural network structure was able to detect radar track reassignments across synchronized multi-modal track data input. This neural network proved highly accurate at recognizing said reassignments in radar track streams when the training dataset was appropriately designed to map the input

streams to consistency numerical sequences of integers in the range [0, 2]. Using this track reassignment detection and the camera video IDs for a selected camera object track, a masking of absolute difference data between any two paired sensor tracks along with backward interpolation of the most recent consistent difference data was able to provide the fusion neural network with the ability to classify fusion between 1-second camera and radar track samples in the constructed deep learning sensor fusion system.

When combined in the constructed deep learning fusion system, it was shown that the consistency and fusion deep neural networks could be utilized to fuse multi-modal sensor data from two independent sensor sources of camera and radar to within a 95% agreement of a state-of-the-art sensor fusion system. The constructed deep learning sensor fusion system was thus able to mimic the complex fusion capabilities of a preexisting state-of-the-art fusion system to within a 5% error bound.

The runtime of the constructed deep learning sensor fusion system was shown to react to simulated incoming sensor updates at real-time speeds of 40 ms on a deep learning PC with a single NVIDIA GeForce GTX 980Ti GPU with combined parallelization and batch mode deep neural network evaluation. The runtimes were benchmarked using the slowest versions of the deep neural networks constructed in order to acquire an upper-bound runtime, which still demonstrated a real-time response rate of the constructed deep learning sensor fusion system with the proper parallelization and hardware specifications.

The work described herein could be extended in a variety of ways. A physics model could be applied to the sensor data or neural network structures in order to ensure that fusion only took place in truly realistic situations where both sensors tracked the same vehicles over time. The physics model would likely help eliminate disagreement between the deep learning fusion system and the state-of-the-art fusion system as shown in section 3.7.1. The additional analysis of forward-facing video

acquired from vehicles equipped with paired radar and camera sensors could add increased reliability and certainty to the deep learning fusion results. A convolutional neural network, provided with the video as input, could merge with the developed multi-stream neural network structures in order to learn a more accurate representation of sensor fusion between the employed sensors via additive spatial video context. Additionally, the multi-stream deep neural networks could be implemented in an embeddable, high-speed deep learning framework such as Caffe2 for increased efficiency and performance. Lastly, the deep learning fusion system could be tested on a embedded system with a NVIDIA GPU, such as the Jetson TX2, to determine if the system was truly capable of embedded real-time response rate with GPU hardware.

In conclusion, we found that the constructed deep learning sensor fusion system was able to reproduce the complex, dynamic behavior of a state-of-the-art sensor fusion system on behalf of multiple deep neural networks combined in the proposed novel way. These deep neural networks required training datasets with enough breadth and quality to sufficiently capture a wide spectrum of synchronized sensor track reassignment and fusion scenarios between both sensors. The constructed deep learning fusion system was proven versatile and highly generalizable given its ability to learn and mimic both an existing state-of-the-art fusion algorithm and a novel hand-engineered fusion algorithm with minimal confusion. It also demonstrated capabilities of real-time inference with proper parallelization and hardware specifications. Thus, it was shown that deep learning techniques are versatile enough to provide both data preprocessing and data fusion capabilities when trained with enough consistent, high quality examples, and such deep learning techniques may be accelerated for high performance use in a variety of ways. Future work was proposed that could enhance the realism and performance of the constructed deep learning fusion system to further guide it toward embeddable deployment.

Appendices

.1 Most Restrictive Fusion Dataset Validation Results

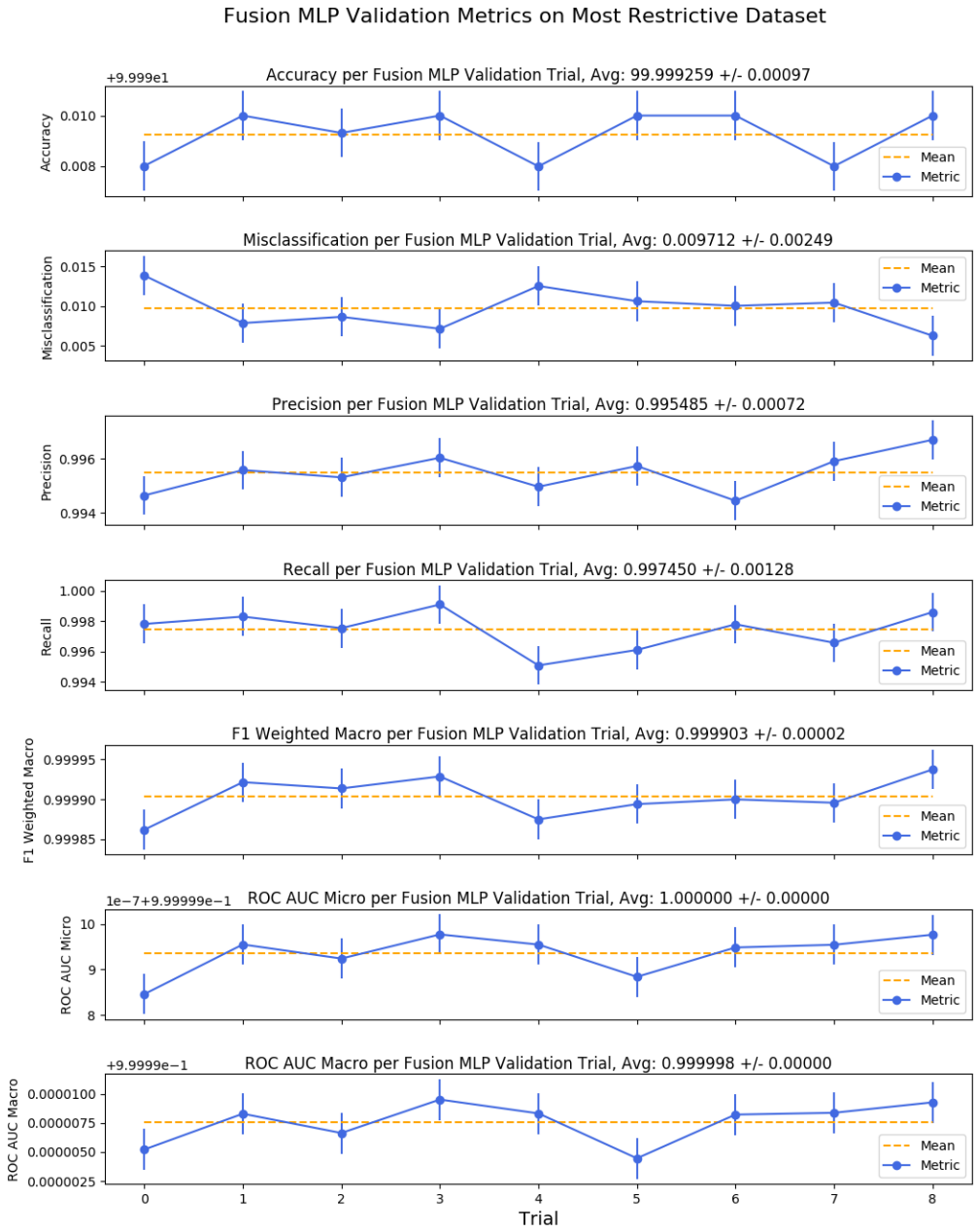

Figure 1: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the MLP fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the most restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

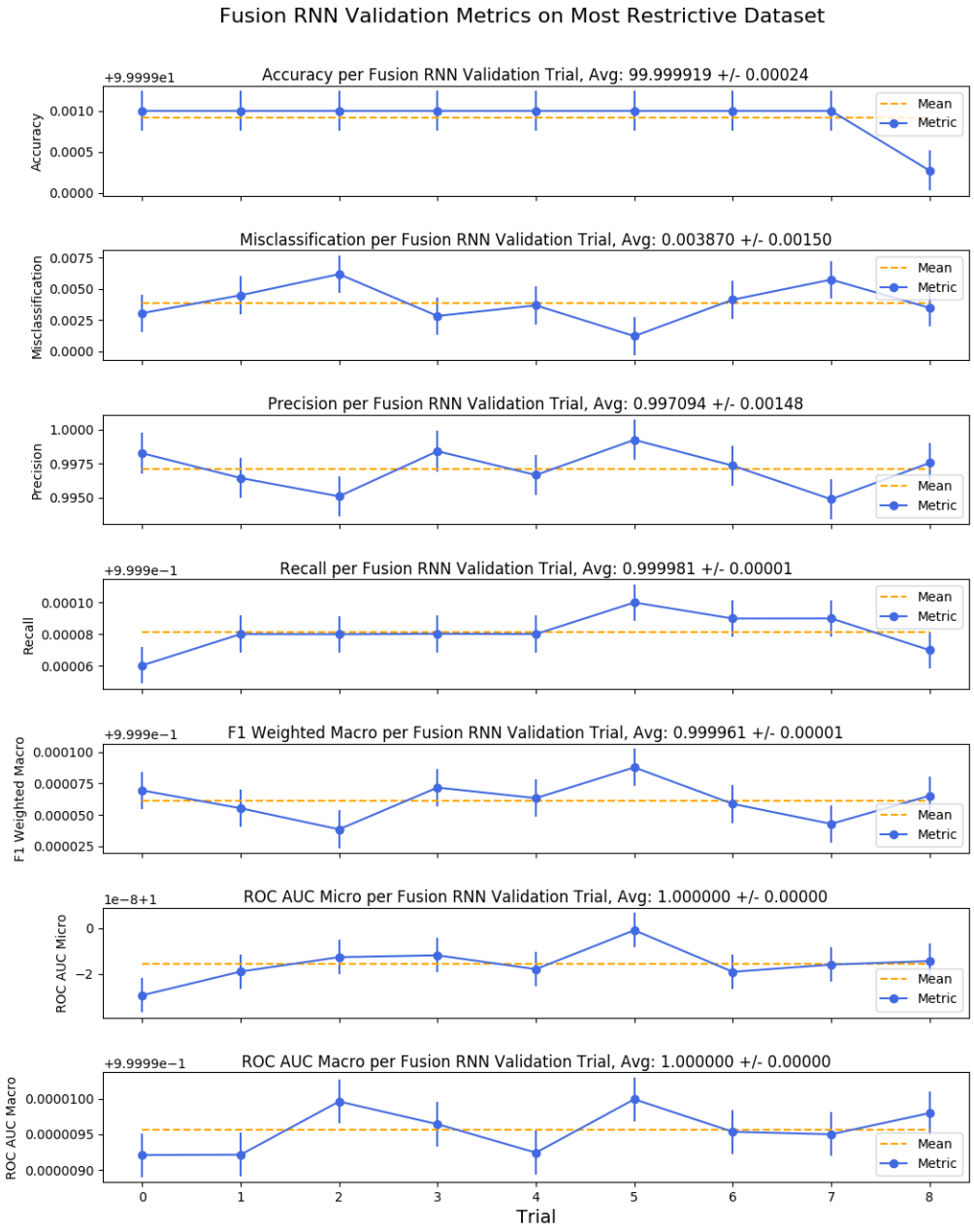

Figure 2: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the RNN fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the most restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

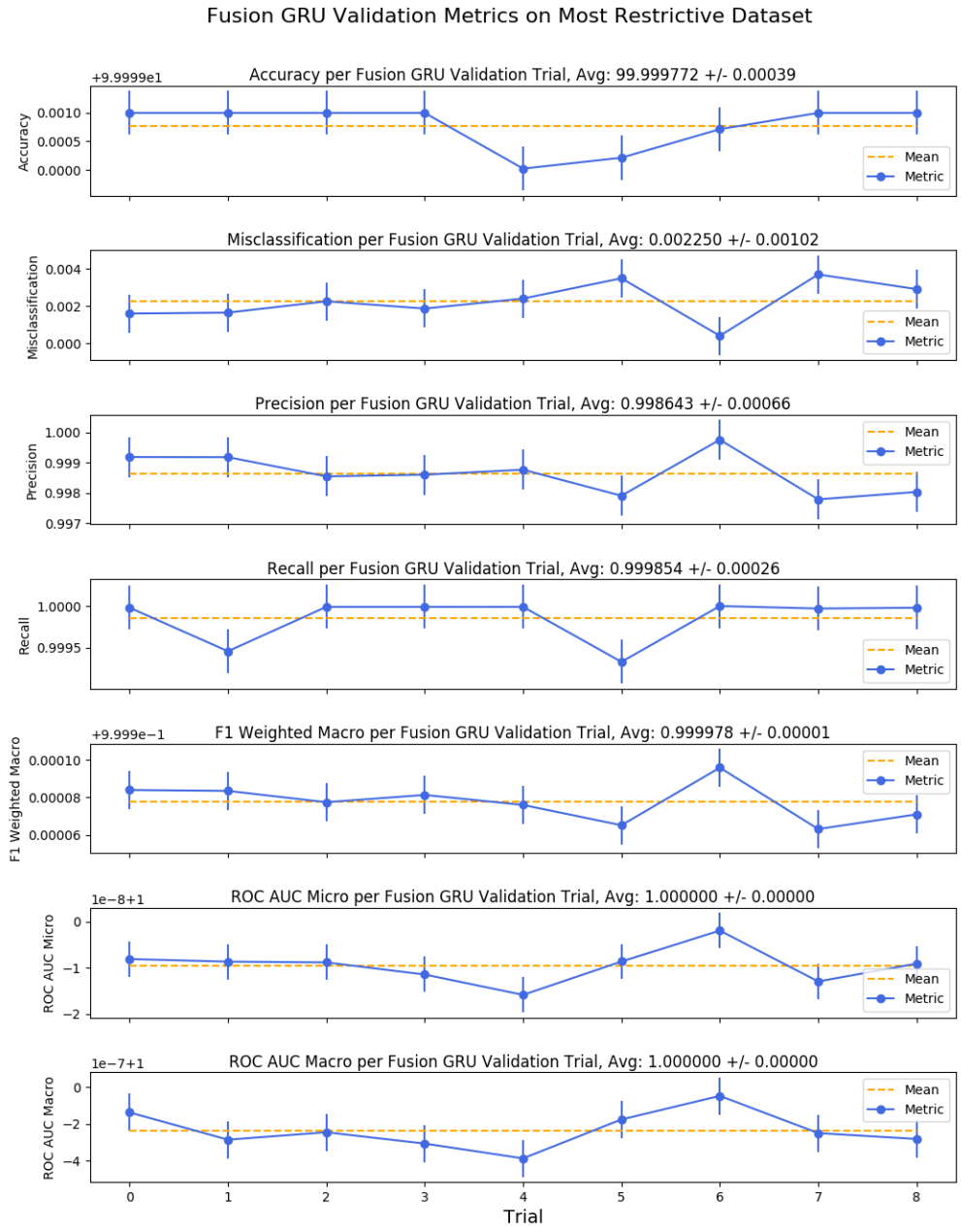

Figure 3: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the GRU fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the most restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

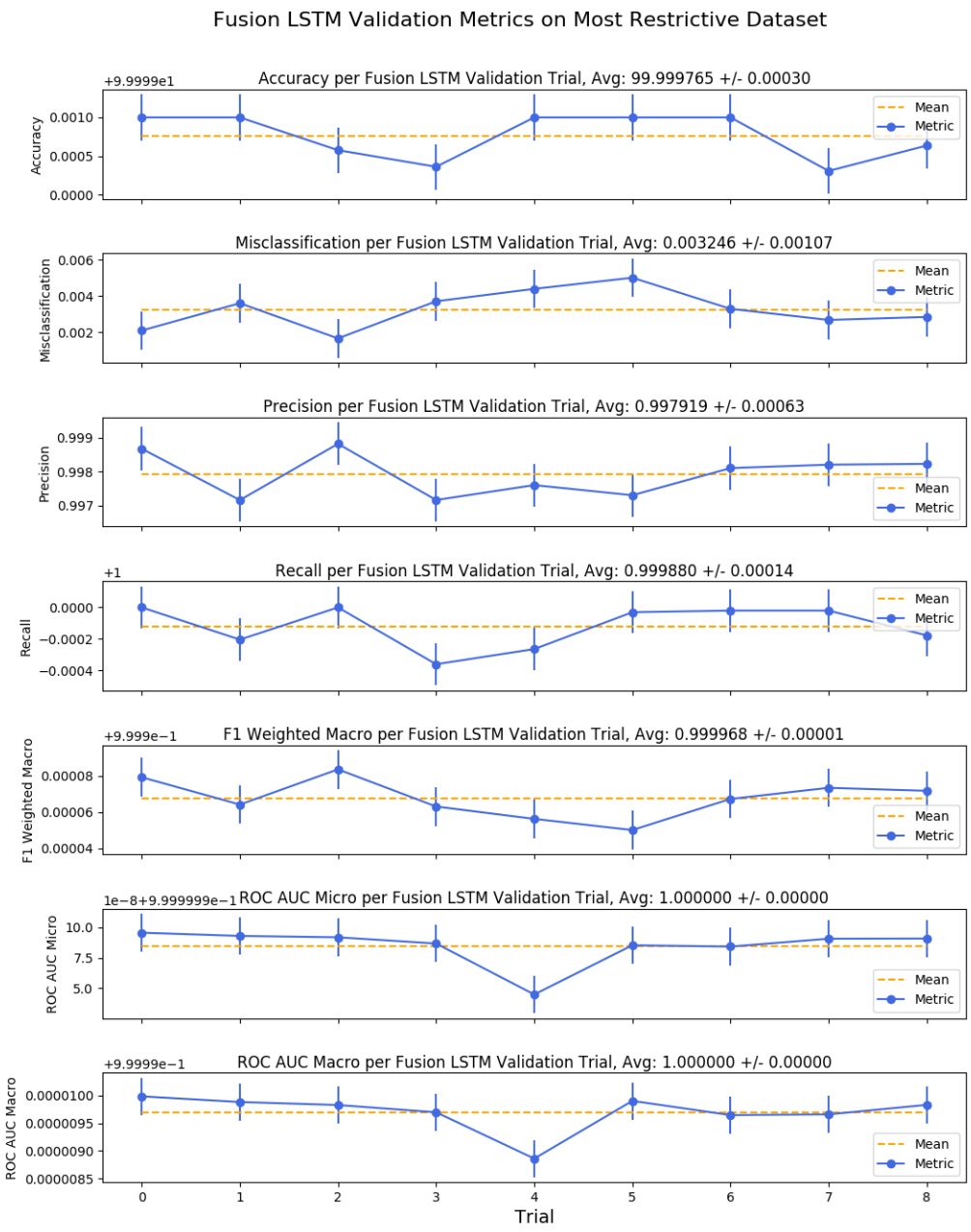

Figure 4: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the LSTM fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the most restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

.2 Mid-Restrictive Fusion Dataset Validation Results

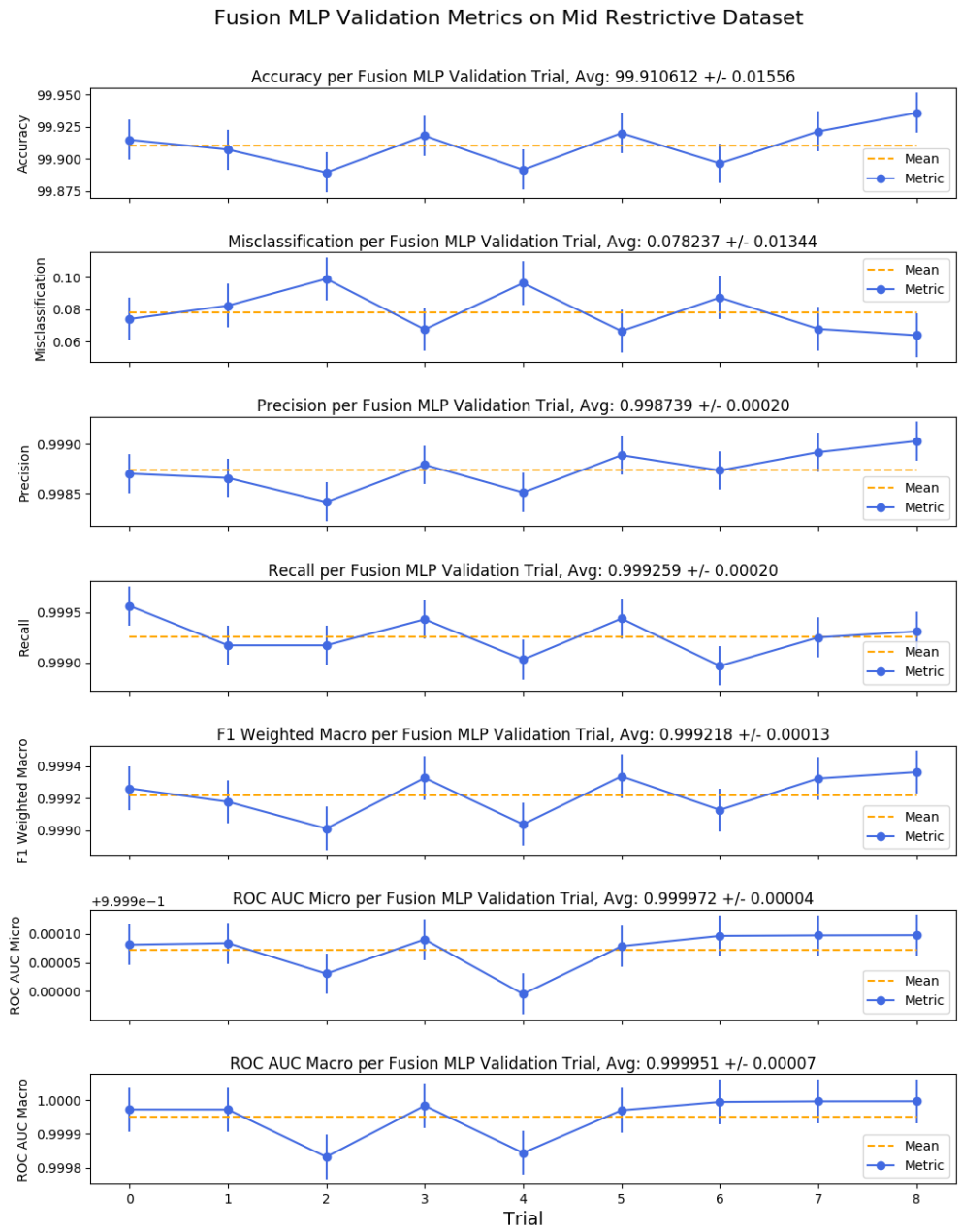

Figure 5: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the MLP fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the mid-restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

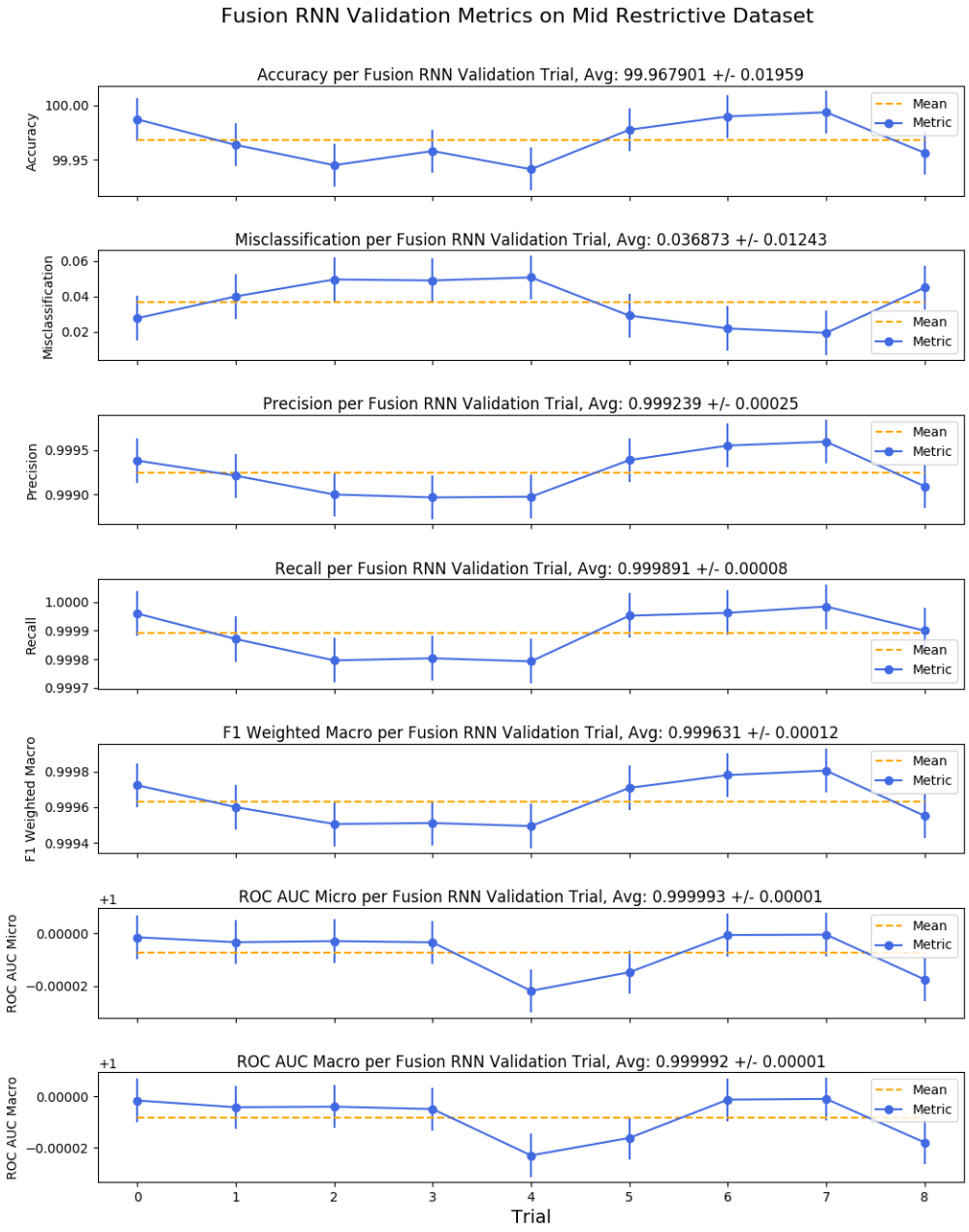

Figure 6: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the RNN fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the mid-restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

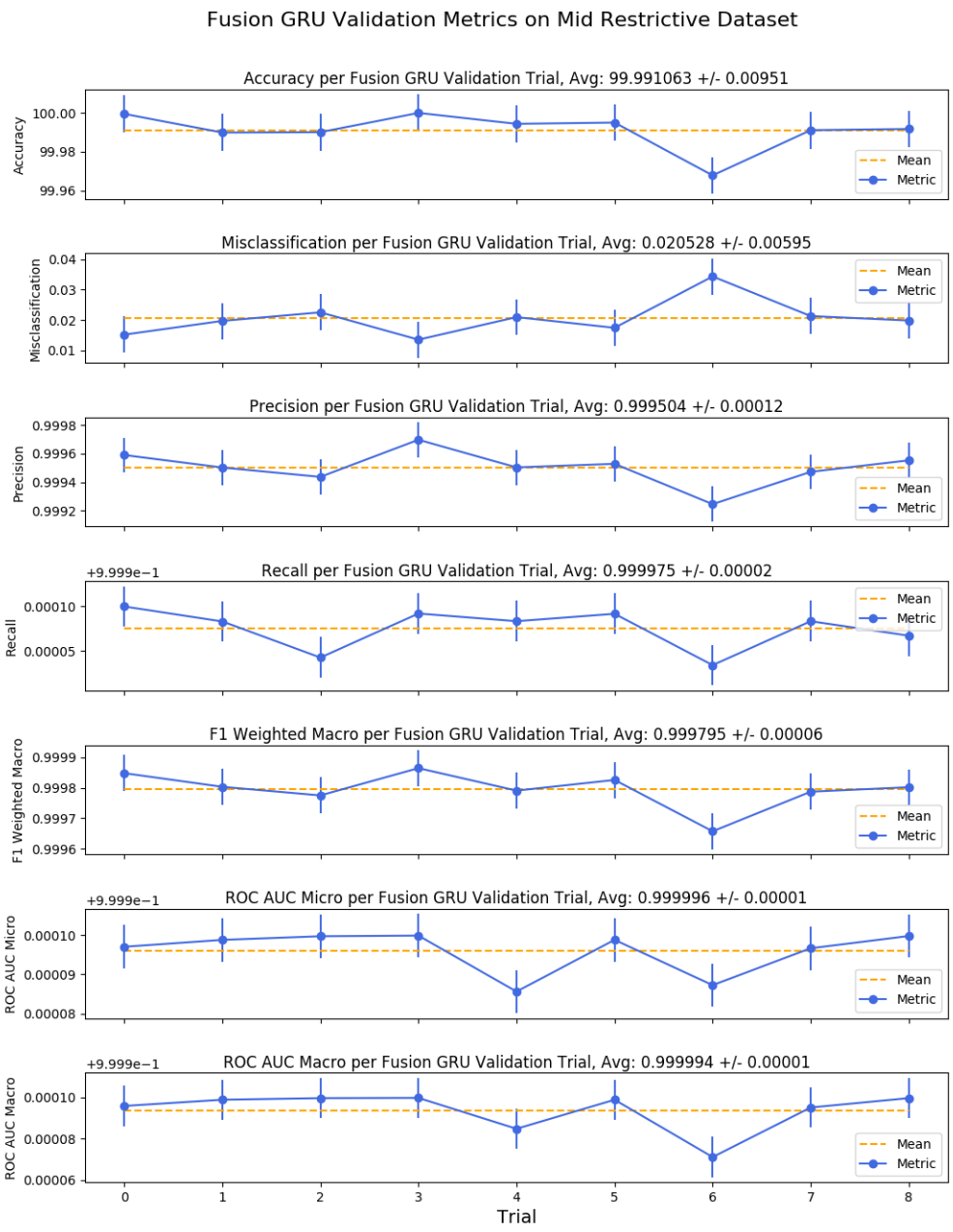

Figure 7: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the GRU fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the mid-restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

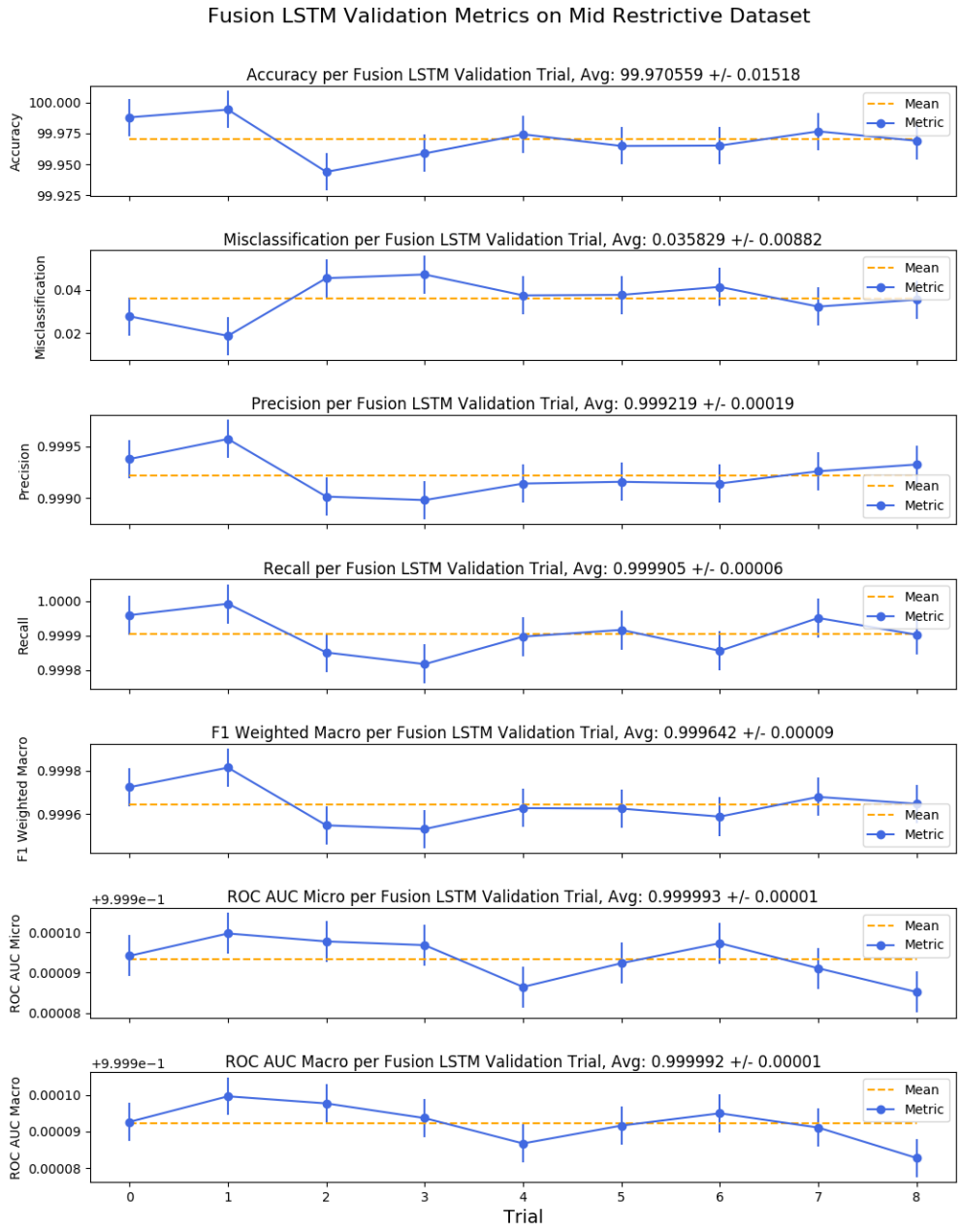

Figure 8: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the LSTM fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the mid-restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

.3 Least Restrictive Fusion Dataset Validation Results

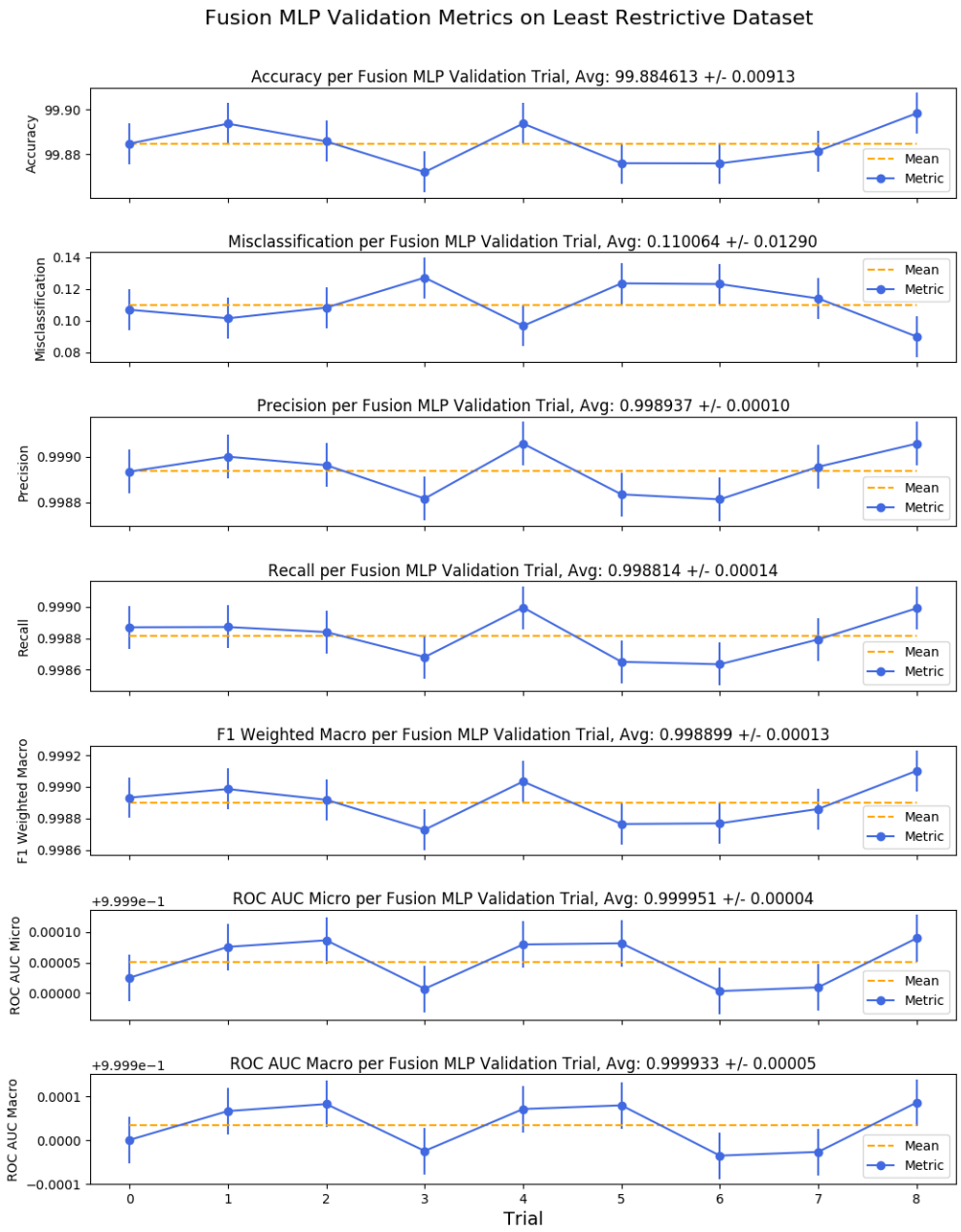

Figure 9: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the MLP fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the least restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

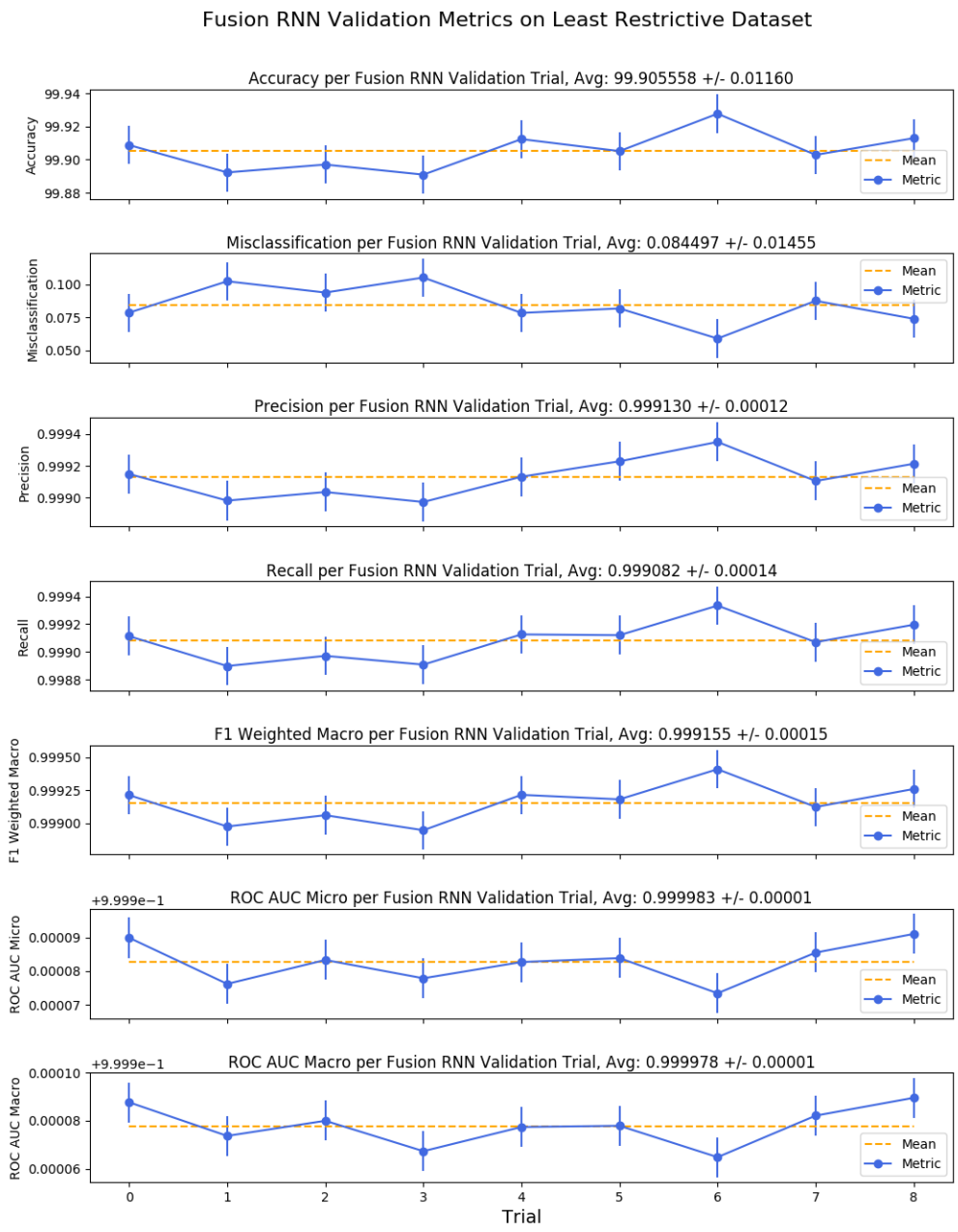

Figure 10: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the RNN fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the least restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

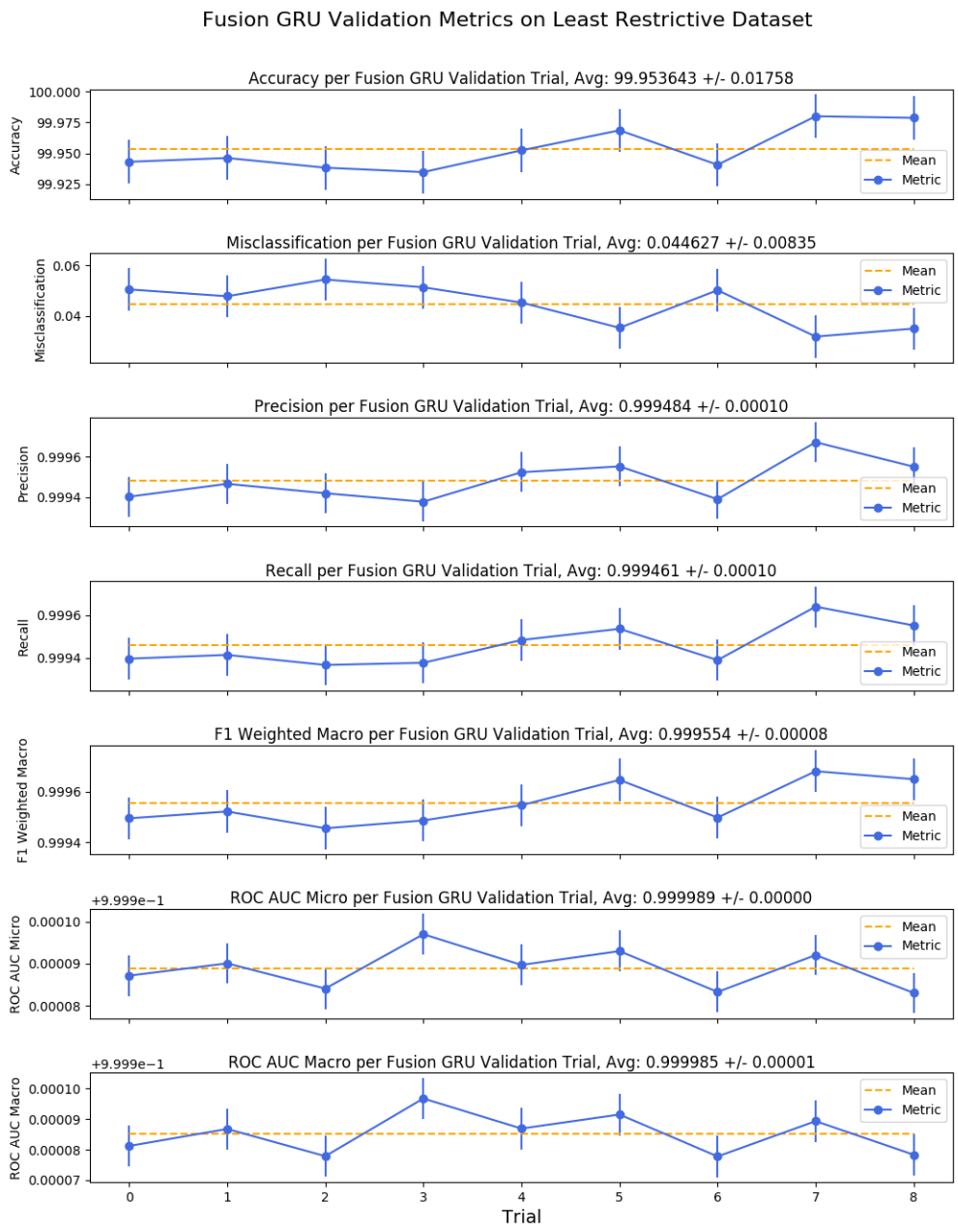

Figure 11: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the GRU fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the least restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

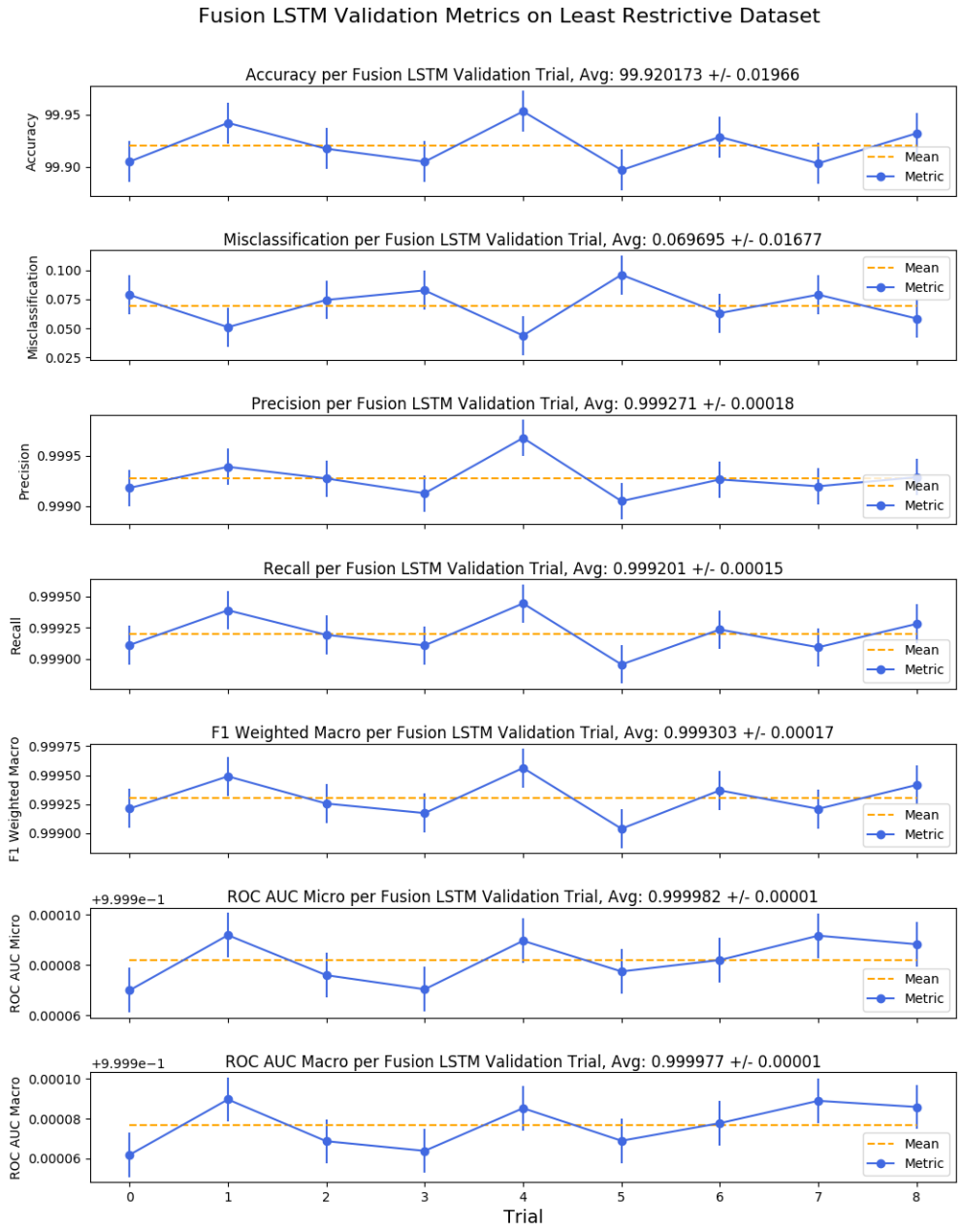

Figure 12: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the LSTM fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the least restrictive fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

.4 Maximum Threshold Fusion Dataset Validation Results

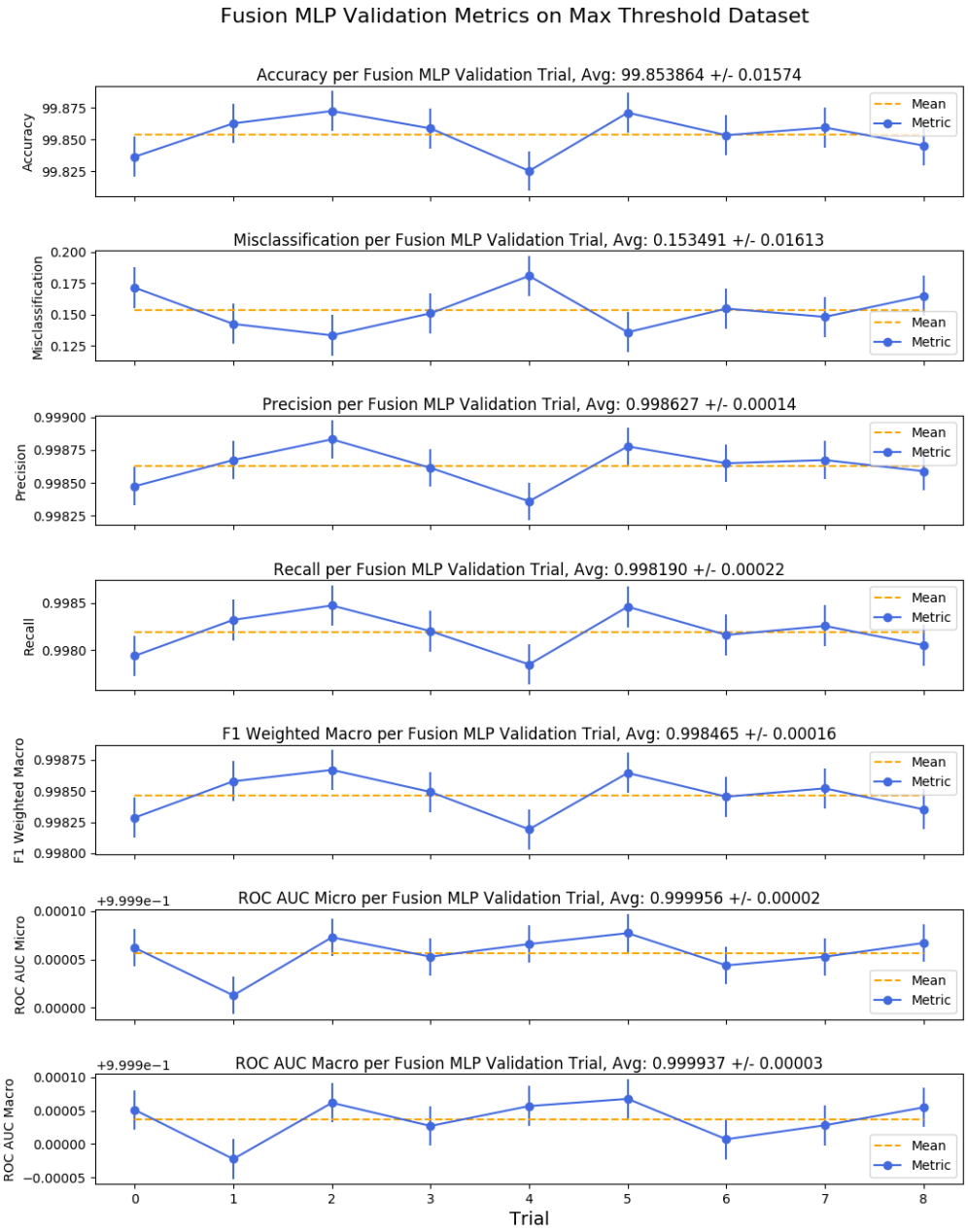

Figure 13: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the MLP fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the max threshold fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

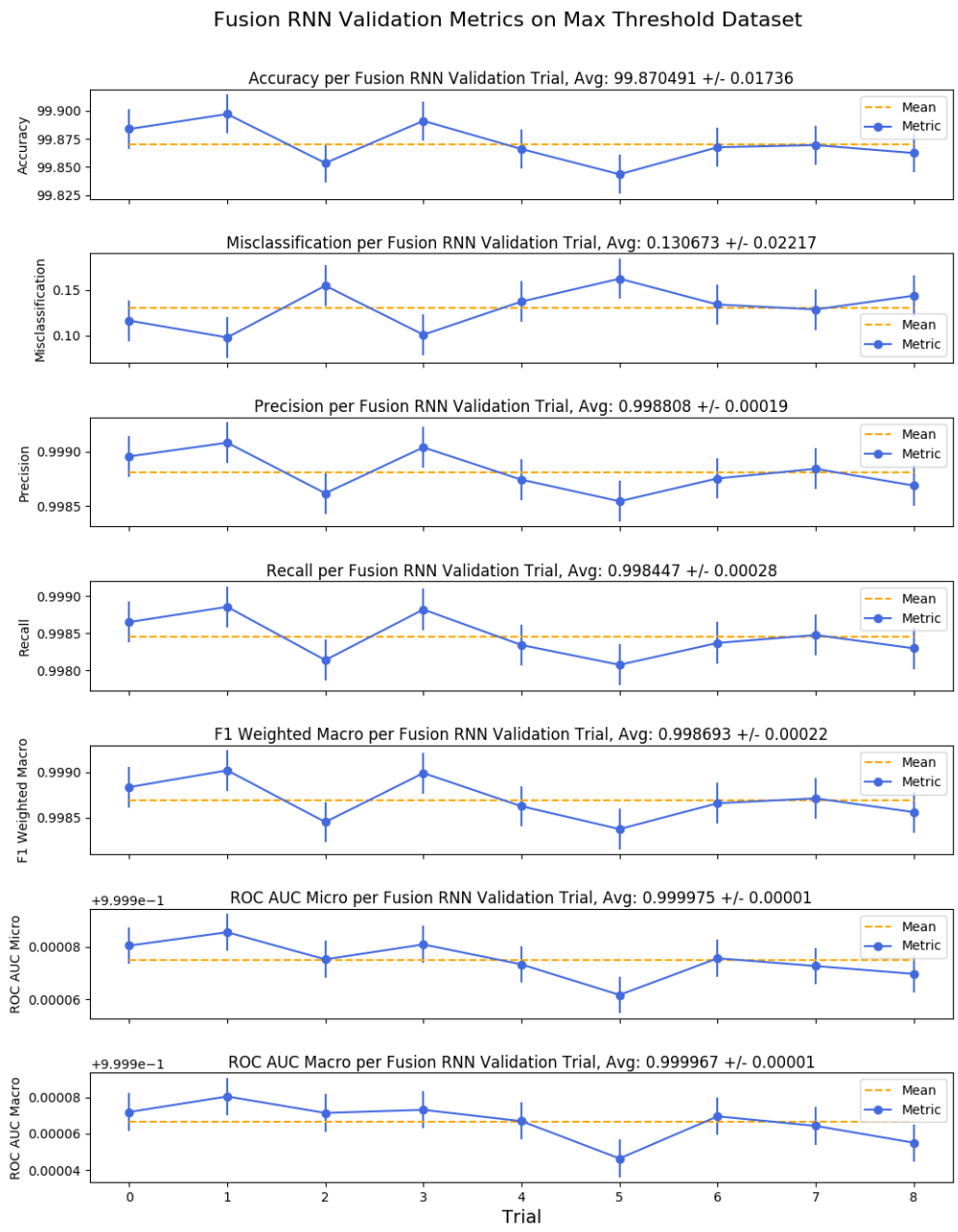

Figure 14: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the RNN fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the max threshold fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

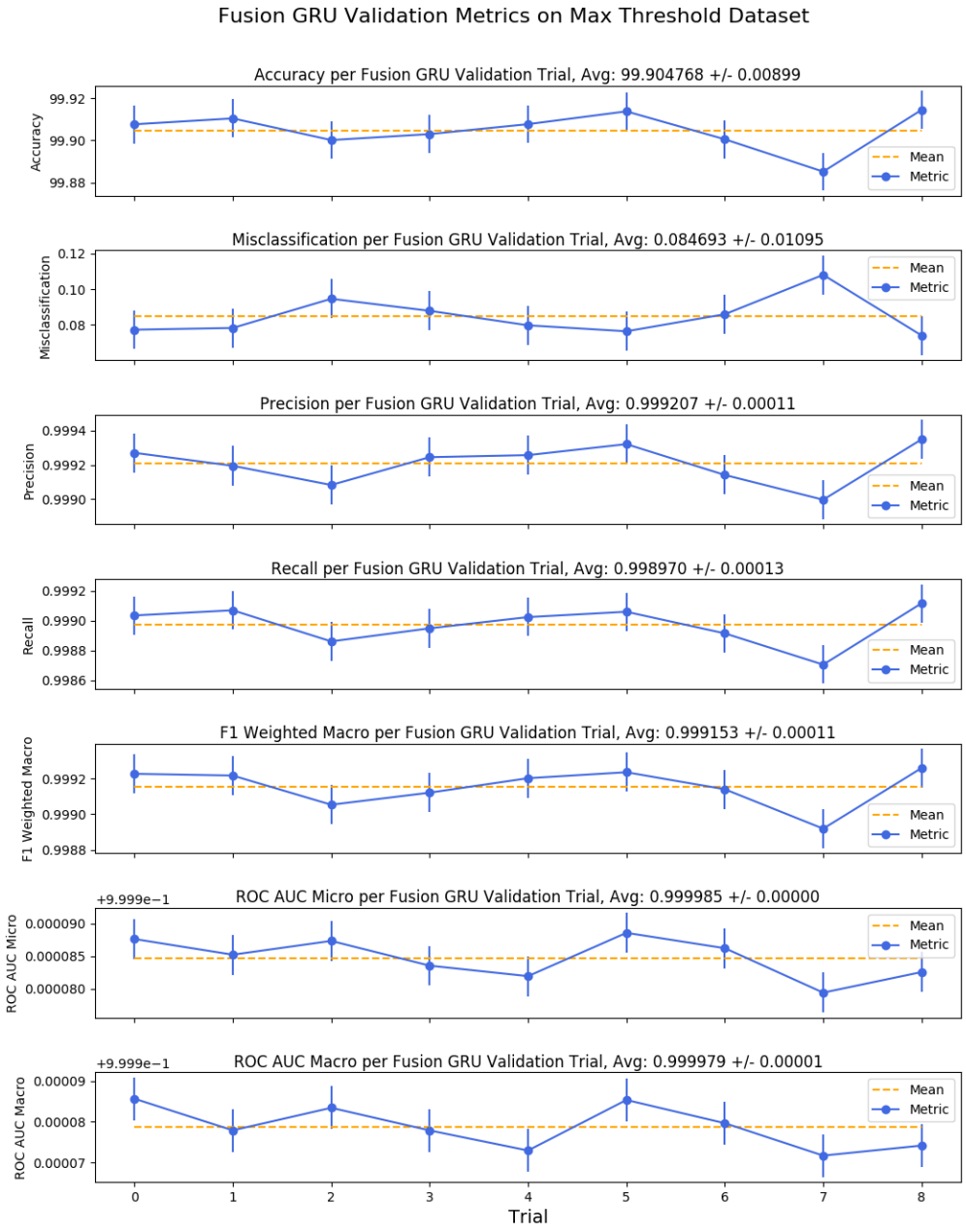

Figure 15: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the GRU fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the max threshold fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

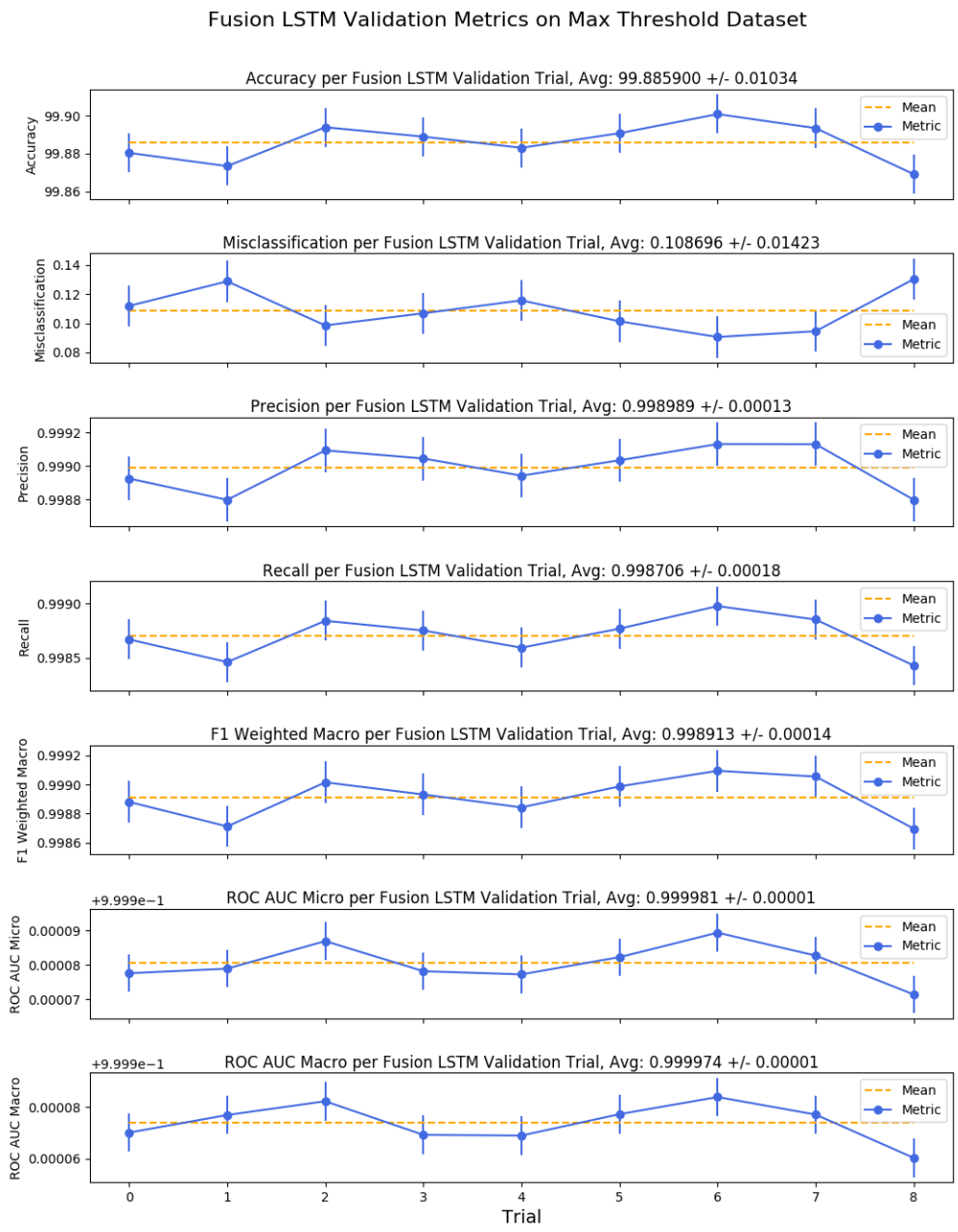

Figure 16: Accuracy, misclassification, precision, recall, F1 weighted macro, ROC AUC micro and ROC AUC macro scores for the LSTM fusion neural network across 9 bootstrapping trials of random train/test data selection from over 1 million randomly-selected samples via the max threshold fusion error dataset. The average score and its standard deviation are included in the title of each subplot.

Bibliography

[gps, 2011] (2011). What is a gps? how does it work?

[hdf, 2017] (2017). The hdf group: Support.

[Asch, 2013] Asch, V. V. (2013). Macro- and micro-averaged evaluation measures.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

[Bahrampour et al., 2015] Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. (2015). Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435.

[Behnel et al., 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39.

[Bottou, 2010] Bottou, L. (2010). *Large-Scale Machine Learning with Stochastic Gradient Descent*, pages 177–186. Physica-Verlag HD, Heidelberg.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

[Chung et al., 2014] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

[Collette, 2013] Collette, A. (2013). *Python and HDF5*. O'Reilly.

[Continuum Analytics, 2015] Continuum Analytics (2015). Bokeh.

[Dagan et al., 2004] Dagan, E., Mano, O., Stein, G. P., and Shashua, A. (2004). Forward collision warning with a single camera. In *IEEE Intelligent Vehicles Symposium, 2004*, pages 37–42.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2).

[Elmenreich, 2001] Elmenreich, W. (2001). An introduction to sensor fusion. Research Report 47/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.

[Elmenreich, 2007] Elmenreich, W. (2007). *A Review on System Architectures for Sensor Fusion Applications*, pages 547–559. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Fawcett, 2006] Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874.

[Gers et al., 2000] Gers, F. A., Schmidhuber, J. A., and Cummins, F. A. (2000). Learning to forget: Continual prediction with lstm. *Neural Comput.*, 12(10):2451–2471.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*.

[Graves, 2012] Graves, A. (2012). Supervised sequence labelling. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 4–103. Springer Berlin Heidelberg.

[Graves, 2014] Graves, A. (2014). Generating sequences with recurrent neural networks.

[Graves et al., 2013] Graves, A., Jaitly, N., and r. Mohamed, A. (2013). Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 273–278.

[Grossberg, 1973] Grossberg, S. (1973). Contour enhancement, short term memory, and constancies in reverberating neural networks. *Studies in Applied Mathematics*, 52(3):213–257.

[Hayek, 1952] Hayek, F. A. (1952). *The sensory order*. University of Chicago Press.

[Hebb, 1949] Hebb, D. O. (1949). *The organization of behavior*. Wiley.

[Hinton et al., 2012] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

[Huang et al., 2016] Huang, L., Chen, H., Yu, Z., and Bai, J. (2016). Multi-target tracking algorithm in the complicated road condition for automotive millimeter-wave radar. In *SAE Technical Paper*. SAE International.

[Ian Goodfellow and Courville, 2016] Ian Goodfellow, Y. B. and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

[Jones et al., 01] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. [Online; accessed ¡today¿].

[Kleene, 1956] Kleene, S. C. (1956).

[Klockner, 2017] Klockner, A. (2017). Pycuda documentation.

[Kodak, 2017] Kodak (2017). Motion picture cameras and lenses. In *The Essential Reference Guide for Filmmakers*, chapter 1, pages 63–70.

[Labayrade et al., 2005] Labayrade, R., Royere, C., and Aubert, D. (2005). A collision mitigation system using laser scanner and stereovision fusion and its assessment. In *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.*, pages 441–446.

[Laneurit et al., 2003] Laneurit, J., Blanc, C., Chapuis, R., and Trassoudaine, L. (2003). Multisensorial data fusion for global vehicle and obstacles absolute positioning. In *IEEE IV2003 Intelligent Vehicles Symposium. Proceedings (Cat. No.03TH8683)*, pages 138–143.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[Levinson et al., 2011] Levinson, J., Askeland, J., Becker, J., Dolson, J., Held, D., Kammel, S., Kolter, J. Z., Langer, D., Pink, O., Pratt, V., Sokolsky, M., Stanek, G., Stavens, D., Teichman, A., Werling, M., and Thrun, S. (2011). Towards fully

autonomous driving: systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*.

[Lipton et al., 2015] Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019.

[Mcculloch and Pitts, 1943] Mcculloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147.

[McKinney, 2010] McKinney, W. (2010). Data structures for statistical computing in python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

[Minsky, 1956] Minsky, M. L. (1956).

[Nervana Systems, 2017a] Nervana Systems (2017a). Neon.

[Nervana Systems, 2017b] Nervana Systems (2017b). neon.optimizers.optimizer.adagrad.

[Nervana Systems, 2017c] Nervana Systems (2017c). neon.transforms.cost.crossentropybinary.

[Nervana Systems, 2017d] Nervana Systems (2017d). Nvis.

[Ngiam et al., 2011] Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A. (2011). *Multimodal deep learning*, pages 689–696.

[NOAA, 2012] NOAA (2012). Lidar 101: An introduction to lidar technology, data, and applications.

[Nvidia, 2015] Nvidia (2015). Gpu-based deep learning inference: A performance and power analysis.

[Nvidia, 2017] Nvidia (2017). Cuda zone: Nvidia developer.

[Oancea and Ciucu, 2014] Oancea, B. and Ciucu, S. C. (2014). Time series forecasting using neural networks. *CoRR*, abs/1401.1333.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830.

[Plahl et al., 2013] Plahl, C., Kozielski, M., Schlter, R., and Ney, H. (2013). Feature combination and stacking of recurrent and non-recurrent neural networks for lvcsr. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6714–6718.

[Python Software Foundation, 2017] Python Software Foundation (2017). Welcome to python.org.

[Rabiner and Schafer, 1978] Rabiner, L. and Schafer, R. (1978). *Digital Processing of Speech Signals. Signal Processing Series.* Prentice Hall.

[Rabiner and Gold, 1975] Rabiner, L. R. and Gold, B. (1975). *Theory and Application of Digital Signal Processing.* Prentice-Hall.

[Rao, 2001] Rao, N. S. V. (2001). On fusers that perform better than best sensor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(8):904–909.

[Reed et al., 2014] Reed, S. E., Lee, H., Anguelov, D., Szegedy, C., Erhan, D., and Rabinovich, A. (2014). Training deep neural networks on noisy labels with bootstrapping. *CoRR*, abs/1412.6596.

[Riedmiller, 1994] Riedmiller, M. (1994). Rprop - description and implementation details.

[Rojas, 1996] Rojas, R. (1996). *The Backpropagation Algorithm*. Springer-Verlag New York, Inc., New York, NY, USA.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 386–408.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms.

[Singh et al., 2016] Singh, B., Marks, T. K., Jones, M., Tuzel, O., and Shao, M. (2016). A multi-stream bi-directional recurrent neural network for fine-grained action detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1961–1970.

[Song et al., 2016a] Song, H., Thiagarajan, J. J., Sattigeri, P., Ramamurthy, K. N., and Spanias, A. (2016a). A deep learning approach to multiple kernel fusion.

[Song et al., 2016b] Song, S., Chandrasekhar, V., Mandal, B., Li, L., Lim, J. H., Babu, G. S., San, P. P., and Cheung, N. M. (2016b). Multimodal multi-stream deep learning for egocentric activity recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 378–385.

[Srinivasa et al., 2003] Srinivasa, N., Chen, Y., and Daniell, C. (2003). A fusion system for real-time forward collision warning in automobiles. In *Proceedings of*

the 2003 IEEE International Conference on Intelligent Transportation Systems, volume 1, pages 457–462 vol.1.

[Srivastava, 2013] Srivastava, N. (2013). *Improving neural networks with dropout*. PhD thesis, University of Toronto.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

[Steux et al., 2002] Steux, B., Laurgeau, C., Salesse, L., and Wautier, D. (2002). Fade: a vehicle detection and tracking system featuring monocular color vision and radar data fusion. In *Intelligent Vehicle Symposium, 2002. IEEE*, volume 2, pages 632–639 vol.2.

[Sundermeyer et al., 2013] Sundermeyer, M., Oparin, I., Gauvain, J. L., Freiberg, B., Schlter, R., and Ney, H. (2013). Comparison of feedforward and recurrent neural network language models. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8430–8434.

[Svozil et al., 1997] Svozil, D., Kvasnicka, V., and Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43 – 62.

[Technische Universitaet Muenchen, 2013] Technische Universitaet Muenchen (2013). rmsprop.

[Thimm et al., 1996] Thimm, G., Moerland, P., and Fiesler, E. (1996). The interchangeability of learning rate and gain in backpropagation neural networks. *Neural Computation*, 8(2):451–460.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. E. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.

[van der Walt et al., 2011] van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[Wu et al., 2015] Wu, Z., Jiang, Y., Wang, X., Ye, H., Xue, X., and Wang, J. (2015). Fusing multi-stream deep networks for video classification. *CoRR*, abs/1509.06086.

[Xing and Qiao, 2016] Xing, L. and Qiao, Y. (2016). Deepwriter: A multi-stream deep CNN for text-independent writer identification. *CoRR*, abs/1606.06472.

[Yao et al., 2016] Yao, S., Hu, S., Zhao, Y., Zhang, A., and Abdelzaher, T. F. (2016). Deepsense: A unified deep learning framework for time-series mobile sensing data processing. *CoRR*, abs/1611.01942.

[Zaremba et al., 2014] Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *CoRR*, abs/1409.2329.