NAVIGATION WITH ARTIFICIAL NEURAL NETWORKS

DISSERTATION

Joseph A. Curro II, Captain, USAF

AFIT-ENG-DS-18-S-007

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

AFIT-ENG-DS-18-S-007

NAVIGATION WITH ARTIFICIAL NEURAL NETWORKS

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Degree of Doctor of Philosophy

Joseph A. Curro II, B.S.E.E., M.S.E.E.

Captain, USAF

September 2018

AFIT-ENG-DS-18-S-007

NAVIGATION WITH ARTIFICIAL NEURAL NETWORKS

DISSERTATION

Joseph A. Curro II, B.S.E.E., M.S.E.E.
Captain, USAF

Committee Membership:

John Raquet, PhD
Chair

Brett Borghetti, PhD
Member

Matthew Fickus, PhD
Member

Adedeji B. Badiru, PhD
Dean, Graduate School of Engineering and Management

AFIT-ENG-DS-18-S-007

Abstract

The objective of this dissertation is to explore the applications for Artificial Neural Networks (ANNs) in the field of Navigation. The state of the art for ANNs has improved significantly so now they can rival and even surpass humans in problems once thought impossible. We present different methods to augment, combine, or replace existing Navigation filters with ANN. The main focus of these methods is to use as much existing knowledge as possible then use ANNs to extend the current knowledge base. Next, improvements are made for a class of Artificial Neural Network (ANN)s which provide covariance called Mixture Density Network (MDN)s. MDNs are necessary since covariance is required for navigation problems. Finally the improvements and framework are demonstrated in a Very Low Frequency (VLF) signals navigation problem. Without ANNs, our VLF signals navigation problem would be very difficult.

We conduct two VLF navigation experiments with an indoor and outdoor environment. The ANNs used for these problems provide confidence with probabilistic estimates of position either through class probabilities or probability distributions parameterized by the output of MDNs. ANNs need a measure of confidence in their estimates to work with the filters since navigation filters require a confidence of their estimates. In our problems we achieve an indoor localization accuracy of 86.7% for 50 discrete locations, and a 2D RMS error of 63m for a 1km^2 area of navigation.

Table of Contents

Page

List of Figures

Figure Page

Figure Page

List of Tables

List of Symbols

Symbol	Definition
a	scalar value
$\hat{a}$	estimate of sclar value a
$\tilde{a}$	error in estimate of $a = \hat{a} + \tilde{a}$
$\mathbf{a}$	vector of scalar values
$\hat{\mathbf{a}}$	estimate of vector $\mathbf{a}$
$\tilde{\mathbf{a}}$	error in estimate of vector $\mathbf{a} = \hat{\mathbf{a}} + \tilde{\mathbf{a}}$
c	Speed of Light
$\mathbf{x}$	State of filter
$\mathbf{y}$	True Output
$\hat{\mathbf{y}}$	Estimate of Output
$\mathbf{z}$	Measurement
$\mathbf{h}^{NN}$	State of Neural Network
$\boldsymbol{\theta}$	Parameters for a function or model
$I[f_n]$	True Error of model f_n
$I_s[f_n]$	Empirical Error of model f_n
G	Generalization Error
V	Loss Function
$\rho()$	Probability Function
f_a	Noise Figure
p_n	Available Noise Power
k	Boltzmann's Constant
t_0	Reference Temperature

Symbol	Definition
b	Bandwidth
S	Receiver Power
G_t	Transmitter Gain
A_e	Effective Area
R	Radius
s_n	Desired Signal to Noise Ratio (SNR)
δf	Frequency Difference
δv	Velocity Difference
f_0	Base Frequency

Function	Output	Definition
$a(c, d \| \theta)$	$\rightarrow b$	Function a with inputs c, d parameterized by θ returns output b
$a^{NN}(c, d \| \theta)$	$\rightarrow \hat{b}$	Neural Network a with inputs c, d parameterized by θ returns estimate
$\text{FILTER}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k \| \boldsymbol{\theta})$	$\rightarrow \hat{\mathbf{y}}_k$	Overall filter function
$\mathbf{s}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k \| \boldsymbol{\theta}^X)$	$\rightarrow \hat{\mathbf{x}}_k$	Overall filter function without output
$\mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \boldsymbol{\theta}^F)$	$\rightarrow \hat{\mathbf{x}}_{k\|k-1}$	System state predict function propagates state in time
$\mathbf{h}(\hat{\mathbf{x}}_{k\|k-1} \| \boldsymbol{\theta}^H)$	$\rightarrow \hat{\mathbf{z}}_k$	Measurement predict function
$\mathbf{mu}(\hat{\mathbf{x}}_{k\|k-1}, \mathbf{z}_k \| \boldsymbol{\theta}^{MU})$	$\rightarrow \hat{\mathbf{x}}_k$	Measurement update function
$\mathbf{sf}(\hat{\mathbf{x}}_{k\|k-1}, \mathbf{z}_k, \hat{\mathbf{z}}_k \| \boldsymbol{\theta}^{SF})$	$\rightarrow \hat{\mathbf{x}}_k$	State filter function updates state with measurements
$\mathbf{k}(\hat{\mathbf{x}}_{k\|k-1} \| \boldsymbol{\theta}^K)$	$\rightarrow \mathbf{K}_k$	Gain function returns gain to apply to residual for state update
$\mathbf{o}(\hat{\mathbf{x}}_k \| \boldsymbol{\theta}^O)$	$\rightarrow \hat{\mathbf{y}}_k$	Output function returns output based on state
$\text{FILTER}_c^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k \| \boldsymbol{\theta}^C)$	$\rightarrow \tilde{\mathbf{y}}_k$	Correction to filter function output
$\mathbf{s}_c^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k \| \boldsymbol{\theta}^C)$	$\rightarrow \tilde{\mathbf{x}}_k$	Correction to filter function state
$f_c^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k \| \boldsymbol{\theta}^C)$	$\rightarrow \tilde{\mathbf{x}}_{k\|k-1}$	Correction to system state predict function
$\mathbf{m}^{NN}(\mathbf{z}_k \| \boldsymbol{\theta}^M)$	$\rightarrow \mathbf{z}_k^n$	Estimate new measurements from measurements

Symbol	Definition
$\boldsymbol{\theta}^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k \vert \boldsymbol{\theta}^D) \rightarrow \boldsymbol{\theta}_k$	Estimate parameters of filter

List of Acronyms

Acronym	Definition
AFIT	Air Force Institute of Technology
ANN	Artificial Neural Network
ANOVA	Analysis of Variance
BP	Back Propagation
BPTT	Back Progatation Through Time
DFT	Discrete Fourier Transform
DPF	Discrete Particle Filter
ECEF	Earth Centered Earth Fixed
EKF	Extended Kalman Filter
FOGM	First Order Gauss Markov Model
FFT	Fast Fourier Transform
GCS	Gradient Correlation and Scaling
GF	Global Feedback
GMM	Gaussian Mixture Model
GN	Gradient Normalization
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GRU	Gated Recurrent Unit
ITU	International Telecommunication Union
LLA	Latitude Longitude Altitude
LSTM	Long Short Term Memory
LTI	Linear Time Invariant
MDN	Mixture Density Network

Acronym	Definition
MLP	Multi-Layer Perceptron
MSE	Mean Square Error
NaN	Not a Number
NCE	Normalized Covariance Error
NIST	National Institute of Standards and Technology
PCA	Principal Component Analysis
PDF	Probability Density Function
PS	Probability Simplification
RMS	Root Mean Square
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SLAM	Simultaneous Localization And Mapping
SMC	Sequential Monte Carlo
SNR	Signal to Noise Ratio
SoOP	Signals of Opportunity
TOA	Time of Arrival
TF	Teacher Forcing
VLF	Very Low Frequency
WAV	Waveform Audio File Format

NAVIGATION WITH ARTIFICIAL NEURAL NETWORKS

I. Introduction

Introduction

The goal for this dissertation is to utilize the advances in machine learning to improve navigation. Navigation is the problem of traversing from one location to another. Specifically we are interested in estimating navigation states such as position in order to facilitate navigation. Methods that process information into navigation states are referred to as filters. In order to improve navigation filters and thus obtain better state estimates, we will leverage machine learning, specifically recent improvements to ANNs.

Recent advances in machine learning have led to ANNs which are much easier to train and have much more complexity compared to their ancestors even a few years ago. One of the biggest advancements is the Long Short Term Memory (LSTM) architecture [35]. The LSTM architecture learns from sequences of information and retains memory of the sequence in order to recognize patterns in the sequence. In navigation, a filter can be described as a function of a sequence of system inputs and measurements to determine a sequence of state estimates. Thus, LSTM ANNs can behave like a filter, but are created by training on empirical data, not tailoring filters with a priori knowledge. This dissertation explores the different roles ANNs can play in improving a navigation filter.

This dissertation provides a framework for improving navigation filters by integrating ANNs into filters. There are two different general approaches for integrating ANNs in filters. The first approach allows an ANN to replace an entire filter or functions of a filter. There are many different possible configurations for this type of ANN integration. We use non-linear filtering theory as a guideline to promote a theoretical backing for a certain configuration. This breakdown allows ANNs to perform common filter operations in a framework which is more observable compared to one large ANN. The second approach integrates ANNs and filters by augmenting an existing filter. The augmentations include providing corrections, adding measurements, and adapting filter parameters. The second approach has the advantage of maintaining the operations of a trusted filter while still gaining a performance boost from ANN augmentation.

One contribution of this work is to provide a method to incorporate more a priori knowledge into filters. Traditionally a user has a problem, such as navigation, and starts researching and experimenting to discover knowledge about the problem. For example, the user might read textbooks on the topic, research journal articles on the topic, or even run their own experiments. All of these methods produce more knowledge for the user to incorporate into a filter. The key is that the user must understand the meaning of this knowledge before they can incorporate the knowledge into the filter. Books, articles, and even experiments provide meaning to the user with their knowledge. The user understands why the knowledge relates to their problem and how to use the knowledge in the filter. However, there is one rich source of data that is currently missing in this example – empirical data. Empirical data is collected data about the problem. For example, a test of a system may record data about the experiment including sensor data, filter outputs, and truth data. This empirical data contains knowledge about the problem, but may be difficult for the user to understand. The difference between the truth and filter output reveals the errors committed by the filter. To a human this error may look purely random and thus contain no useful information. However, a learning algorithm may unveil meaningful relationships between sensor data and this filter error. The learning algorithm can exploit this relationship to improve the filter. ANNs are one example of a method to derive meaning from knowledge. This now increases the a priori knowledge on the problem, thus improving filter performance.

Integrating ANNs into filters allows the filter to incorporate learning knowledge about the problem. In the extreme case, new data with no a prior knowledge can be used in a filter by learning the relationship between states, system input, and measurements. Empirical data contains knowledge about the system, but a designer may not know the meaning of this data. ANNs learn meaning from the empirical data through training. Thus, ANNs bridge the gap between the knowledge in the empirical data and the meaning required to design a filter. In some problems there would be no known relationships between states, system inputs, and measurements without empirically derived meaning.

Another key aspect of using filters for state estimation is confidence. Confidence is how much trust the user should give the state estimate. Without confidence, mean point estimates are not useful for certain applications, such as navigation. Some types of ANNs can provide confidence in the estimates with covariance information. One of these ANN classes is called MDNs. MDNs output probability parameters instead of a discrete probability distribution or continuous values. Most commonly, MDN output parameters describe a Gaussian Mixture Model (GMM). GMMs are the summation of a fixed number of weighted Gaussian distributions. This dissertation provides techniques and methods that make GMM MDNs easy to train and improve the performance compared to previous implementation of GMM MDNs. ANNs that provide covariance information are essential for navigation, otherwise the user will have no trust in the ANN output.

We use VLF navigation as a specific example to demonstrate methods to integrate ANNs and classic filters. Our VLF signals navigation problem uses ANNs to determine a position because our VLF navigation problem includes very little a prior knowledge of VLF signals collected. The information that is not known includes a description of the signals from the VLF transmitters and where the transmitters are located. However, the problem does have empirically recorded VLF signal samples with labeled positions. The ANNs use the recored labeled data to build functions to relate VLF signals to position. These functions are a result of the ANNs learning the relationship between the VLF signals and position. Machine learning techniques for supervised learning are applied to encourage the ANN to learn salient features of the VLF signals which will persist over time and through noise.

The VLF navigation problem uses a small loop antenna to record the VLF signals while moving. These signals are recorded by an iPhone 5c which also records position either automatically with Global Positioning System (GPS) or manually by recording discrete locations on the phone. The VLF signals are preprocessed into frequency power bins using a Fast Fourier Transform (FFT). An ANN is trained with the FFT features to predict the location. This ANN is integrated with other filters to show how an ANN can aid a filter. Other machine learning techniques are applied to the signal in order to train the ANN.

Goals and Contributions

The goal of this dissertation is to define a framework for integrating ANNs and filters. This framework encompasses existing deigns of ANN and filter integration. We use non-linear filtering theory as a basis for the framework where other existing designs are more ad hoc [77]. Additionally we improve methods to obtain confidence from ANNs. The filter framework requires confidence and would be useless without the ability to assign confidence to estimates. Specifically we introduce new methods and techniques for working with MDNs with GMM output. As an example, the framework is applied to a VLF navigation problem to showcase the strengths of the framework and MDN improvements. The VLF navigation problem in this paper would difficult without ANNs due to lack of a prior knowledge of VLF signals.

The following is a list of research contributions:

- Developed a framework for integrating ANNs into filters based on non-linear filtering theory

- The framework has two major sections for ANNs replacing filter components and for ANNs augmenting existing filters
- Showed how existing ANN filter implementations fit into our framework
- Improved training and performance of GMM MDN
  - Defined a new series of loss functions for GMM MDNs which improves performance over unmodified GMM loss
  - Outlined methods to avoid common problems when training GMM MDNs
- Demonstrated this filter framework and MDN improvements with VLF signals navigation

Document Organization

This dissertation is organized as follows:

Chapter II is a review of relevant background and publications in this research area. The background includes an overview of state estimation and control theory. Next the background focuses on machine learning techniques and tools. Finally a brief overview of the VLF electromagnetic band is given.

Chapter III outlines the ANN and filter integration framework. The chapter outlines the framework's basis in non-linear filtering theory in relation to existing ad hoc implementations. Next, the chapter discusses how the framework encompasses existing implementations of ANN and filter integration.

Chapter IV introduces methods and techniques to improve training and performance of GMM MDNs which are vital to obtain confidence from ANNs. This includes a new type of MDN loss function that achieves better performance compared to the standard MDN loss function and ANNs without confidence.

Chapter V applies the ANN filter framework to a position estimation problem using VLF signals. First the problem is defined along with the relevant collected data. Next, the structure of the ANN and techniques used in training are outlined. This provides the tools and methods necessary to use ANNs in the VLF navigation scenario. Finally the chapter presents results using different implementations of the framework for the VLF signals problem.

Chapter VI summarizes the results and covers possible future work as a result of this research.

II. Background & Literature Review

This chapter outlines previously known discoveries, research, and experiments that relate to this dissertation. Four main topics of importance will be summarized. The first topic is a background on state estimation outlining the different approaches with respect to navigation. The second topic area is an overview of machine learning and ANNs with a short description of the background and the models that are useful for the problem in this dissertation. The third section outlines properties of the VLF spectrum exploited in this dissertation. The final section reviews the state of the art for ANN and state estimation integration.

Estimation

Estimation theory outlines the estimation or prediction of hidden variables or states, from observed variables or states [54, 55]. Estimation describes the tools and methods to make these estimations based on the given assumptions.

Estimation theory can have two different state classes. The first is continuous scalar states like the range to a target. In this case, the states are continuous variables or vectors of variables determined by our model. The second class of states is classification which estimates discrete states. In classification, discrete states are usually classes or labels without an ordering. An example of discrete estimation is a target detection problem. In target detection, the states may be “target present” or “no target present”. To expand this to a multi-class problem, we could change the question to which target is present and add a null class for no target present.

States are hidden or unknown either because they cannot be observed or observing those states directly is too expensive. Consider using a laser ranger sensor to determine the distance from a wall. In this example the hidden state is the actual distance to the wall. The observations are the measurements from the laser ranger. The measurements may be corrupted or biased by noise, so the measurements are not the true distance. Estimation theory makes estimations or predictions of the true distance based on the measured distance. In this example we had a priori knowledge about the problem. We had a meaningful physics model representation of our laser ranger device. We used this physics model representation to understand the nature of the measurements from the device. An example of our model could be simplified as:

$$x = d + n + b \tag{1}$$

where d is our true distance and x is the measured distance which is corrupted by some kind of noise n and a bias b. Our a priori knowledge about the device informed us the measurement would have noise and could also be biased. Our a priori knowledge could have been from the device documentation which outlined the expected noise and bias for the device. Our model is thus shaped by our a priori knowledge. If for example according to the documentation the device had no bias, we would make a new model:

$$x = d + n \tag{2}$$

which has no bias. In short we have a meaningful model based on a priori knowledge derived from physics principles that governed the device. This is the basis for much of state estimation and navigation. A model is created based on a priori knowledge of the world and/or problem. This a priori knowledge has meaning to the user which allows them to construct a model. Sometimes a priori knowledge is available but the meaning is difficult to understand for humans. For example large amount of empirical measurements from our laser ranger may contain information about the noise and bias. This dissertation shows how to use a priori knowledge from empirical collected data to determine meaning for state estimation.

Estimation with Respect to Navigation.

Navigation is the problem of traversing from one location to another. In general estimate of position at all times is not required to achieve navigation since we are only interested in arriving at the end location. However estimating navigation states such as position and time can facilitate certain navigation algorithms.

Navigation in general exploits a well known system dynamics and sensor models to estimate the desired hidden states based on observable states. Many times humans use a priori knowledge about the world to design navigation systems or algorithms to estimate the desired hidden states. For example, the construction of an accelerometer is done based on known physics principles to give specific force measurements. These measurements are then related to position by known physics principles relating force to acceleration to velocity to position. The GPS system was designed with a specific algorithms and physics principles in mind to allow position to be determined by the signal from the satellites via Time of Arrival (TOA) methods [67].

Other systems were not created specifically for navigation, but a priori knowledge about the system was used to create a navigation algorithm. An example of this is Signals of Opportunity (SoOP) navigation which uses available non-navigation signals for navigation purposes. For example, emissions from communication radio towers can be used to determine a position [82]. A priori knowledge about the properties of the antenna such as antenna location, carrier frequency or signal structure are used to create a navigation algorithm.

In navigation, the system dynamics and measurement functions are carefully derived from a priori knowledge. The study of creating these functions spans many different disciplines. In particular, from the control theory perspective, creating these relationships is called state observation.

State Observation

State observation is a subset of the controls problem [75]. The controls problem in general attempts to solve for the system of inputs to command a plant, in order to achieve a desired state. In this context, a plant is any dynamic system whose state we wish to control. One part of the controls problem is creating a state observer which estimates the states from the system inputs, measurements and current state. In this view of the problem the states are the values which can be predicted from the system inputs, measurements, and current state. This is usually done by estimating the next state value $\mathbf{x}_{k+1}$. System inputs $\mathbf{u}$ are values which can be driven by the controller. This dissertation will adopt the convention that system inputs are inputs to a plant or other dynamic system and inputs in general are arguments for functions or algorithms. For example, the thrust of an aircraft is a system input controlled by the pilot. Thus the control problem of achieving a certain airspeed is done by adjusting the thrust. Measurements are values which are observed by sensors but cannot be directly controlled. In the aircraft example, a sensor can measure airspeed of the aircraft which can aid the pilot in maintaining a certain airspeed, but the pilot cannot directly set the airspeed with a thrust controller. To continue this example, a state of the aircraft may be its position in latitude and longitude. There is no way to observe or change this state directly, but it can be estimated or indirectly changed with knowledge of the input thrust, measured airspeed, and heading.

Control problems are many times simplified to a Linear Time Invariant (LTI) state space representation of the plant and measurement equation. The LTI assumption means that the plant and measurement equations are purely linear and thus can be represented by matrix operations. Also, the phrase “time invariant” means these matrices do not change over time. In general the discrete representation can be calculated from the continuous equation using a transformation (which is beyond the scope of this dissertation) or determined directly. The LTI system and measurement equations are

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{w}_k \tag{3}$$

$$\mathbf{z}_k = \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k + \mathbf{v}_k \tag{4}$$

where **x** is the state vector, **u** is the system input vector, **z** is the measurement vector, **w** is a noise process that corrupts the model of the plant, and **v** is a noise process that corrupts the model of the measurement observation. Here, **A** is the matrix that represents the contribution of the current state to the future state, **B** is the matrix that shows the contribution of the input directly to the state, **C** is the matrix that relates the state to the measurements and **D** is the feed forward matrix that shows how the system inputs are directly related to the measurements. The subscript on each vector denotes the time index of that vector value. For example, $\mathbf{x}_k$ is the state at time index k where we assume a fixed time interval.

Equation 3 is called the plant, system dynamics, propagation, or state predict equation. The state predict equation determines how to propagate or predict the state from one time step to the next. This equation can be used as an open loop estimate to determine the state from a given set of initial states and a time series of the future inputs. However, the system dynamics estimate will quickly diverge from the true estimate as the process noise corrupts the next state. Thus the perfect system dynamics solution differs from the real world situation due to noise. To limit this divergence, Equation 4 (the observation equation) makes a measurement $\mathbf{z}_k$ of some observable value. The observed value is related to a predicted value from the states using Equation 4. A residual is the difference between the observed measurement and the estimated measurement. The residual is used to adjust the states to make the estimated measurement closer to the observed measurement. The residual can then correct some of the error introduced by plant noise. However, the correction will not be perfect either, because measurements are corrupted by measurement noise and modeling errors in the measurement equation. Thus a trade off must be made to determine how much to correct based on measurements. The Kalman filter is an example of how to optimally make this trade off given confidence in states and measurements.

Kalman Filter.

A popular method of state observation for navigation is the Kalman Filter [32]. In this sense a filter can be thought of as reducing (filtering) all the incoming data and distilling it down to only the desired output. A Kalman filter assumes that the system is a linear system so that all the dynamics and measurement updates can be expressed as differential linear equations in the continuous version, and difference equations in the discrete version. The uncertainty in the dynamics and measurement update is assumed to be purely zero-mean white Gaussian noise. The Kalman filter is the optimal estimator when the dynamics are purely linear and all noise is zero-mean white Gaussian noise. The discrete dynamics equation of the filter is written as:

$$\mathbf{x}_{k+1} = \mathbf{F}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{w}_k \tag{5}$$

where $\mathbf{x}$ is the state vector for the values to be estimated, $\mathbf{F}$ is the matrix which determines the change in state based on the state, $\mathbf{B}$ is the matrix which determines the change in the state based on the input, $\mathbf{w}$ is an instance of a Gaussian noise random variable distributed as $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k)$, and $\mathbf{u}$ is the system input. When this equation is turned into time variant instead of time invariant, each matrix will get a subscript of the time step it relates to:

$$\mathbf{x}_k = \mathbf{F}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k \tag{6}$$

where the matrices $\mathbf{F}_k$ and $\mathbf{B}_k$ are now subscripted because at different time steps the values in the matrices can change. This change can be driven by functions of time alone or some other external information. The discrete matrix $\mathbf{F}_k$ is written as $\mathbf{\Phi}$ in the literature [65, 66]. The measurement update equation is only in discrete time as shown:

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{7}$$

where $\mathbf{z}_k$ is a measurement which relates to the value of the state $\mathbf{x}_k$ through a matrix $\mathbf{H}_k$. The vector $\mathbf{v}_k$ is an instance of a white Gaussian noise random variable distributed as $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R}_k)$ which corrupts the measurement.

The Kalman filter uses these equations and starting conditions to provide an average estimate of the state $\mathbf{x}_k$ and a covariance of that estimate $\mathbf{P}_k$. The initial mean estimate is $\mathbf{x}_0$ and the initial covariance of the mean estimate is the covariance matrix $\mathbf{P}_0$. The filter then propagates the state using the dynamics equation. The state-predict equation of the Kalman filter is given as:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} \tag{8}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{Q}_k \tag{9}$$

where $\hat{\mathbf{x}}_{k|k-1}$ is the estimate for $\mathbf{x}$ at time step k given time step $k-1$. When a measurement is made, the state estimate and covariance are updated to reflect the new measurement. The Kalman filter update equations are given as:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \left(\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k \right)^{-1} \tag{10}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \left(\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1} \right) \tag{11}$$

$$\mathbf{P}_{k|k} = \left(I - \mathbf{K}_k \mathbf{H}_k \right) \mathbf{P}_{k|k-1} \tag{12}$$

where $\mathbf{K}_k$ is the Kalman gain which can be thought of as how much to change the current state, such that the estimated measurement is closer to the observed measurements. The change is based on the trust in the measurement compared to the trust in the current state estimate. Lower Kalman gain represents less trust in the measurement. For example, when the gain is close to all zeros, the update equation will largely ignore the measurements. With these equations, a Kalman filter can process measurements to update the state estimate. The Kalman filter provides the optimal estimate, but only for linear problems corrupted by white Gaussian noise.

Particle Filter.

A particle filter is a Sequential Monte Carlo (SMC) Method to estimate a non-linear, non-Gaussian, Markovian state space. A particle filter can be used in non-linear problems with non-Gaussian noise where a Kalman filter would perform poorly. The problem starts with a known probability of states $p(\mathbf{x}_0)$, a method to predict the states forward in time $p(\mathbf{x}_t|\mathbf{x}_{t-1})$, and a measurement prediction equation that determines the probability of a measurement $\mathbf{y}$ from the current states $p(\mathbf{y}_t|\mathbf{x}_t)$ [17]. From these functions the propagation equation is [17]:

$$p(\mathbf{x}_t|\mathbf{y}_{1:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{y}_{1:t-1})\, d\mathbf{x}_{t-1} \tag{13}$$

where the probability of the state at the next time step $\mathbf{x}_t$ is calculated given all the previous measurements $\mathbf{y}_{1:t-1}$. Next we use the update equation to incorporate the current measurement $\mathbf{y}_t$ [17]:

$$p(\mathbf{x}_t|\mathbf{y}_{1:t}) = \frac{p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{y}_{1:t-1})}{\int p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{y}_{1:t-1})\, d\mathbf{x}t} \tag{14}$$

where we have updated our probability from the last time step to the current time step using the latest measurement $\mathbf{y}_t$. While these equations look simple, the marginal probabilities are difficult to compute since we are integrating over our entire continuous state space $d\mathbf{x}_t$. To approximate the distribution, Monte Carlo methods that track a finite number of states or particles are used. These Monte Carlo algorithms track N particles which represent possible state values. Each particle also has a weight which determines the probability of that particle compared to the other particles. These weighted particles represent the probability of the states at each time step $p(\mathbf{x}_t)$.

While it may be difficult in general to integrate over a continuous state space, a discrete state space is much simpler since the integral is now a sum. Thus, discrete state problems are much simpler to model as a particle filter. In a discrete particle filter, each particle represents one of the discrete states and the weight is the probability of that state. Consequently, all particle weights sum to one to provide a valid probability distribution over all states. In a discrete particle filter our propagate equation turns into:

$$p(x_t^k|\mathbf{y}_{1:t-1}) = \sum_{i=0}^{N-1} p(x_t^k|x_{t-1}^i)p(x_{t-1}^i|\mathbf{y}_{1:t-1}) \tag{15}$$

where we calculated the probability of state x_t^k at time t by summing the probability that any i particle or state will transition to the k particle or state. Similarly the measurement equation becomes:

$$p(x_t^k|\mathbf{y}_{1:t}) = \frac{p(\mathbf{y}_t|x_t^k)p(x_t^k|\mathbf{y}_{1:t-1})}{\sum_{i=0}^{N-1} p(\mathbf{y}_t|x_t^i)p(x_t^i|\mathbf{y}_{1:t-1})} \tag{16}$$

where we update the probability of each particle or state by multiplying by the probability that state produced the given measurement. Also, the state probabilities are normalized to make a valid probability distribution. To reiterate, a discrete particle filter is a simplification of a general particle filter, where a general particle filter approximates continuous state distributions.

Particle filters are useful in non-linear state estimation when the distributions are non-Gaussian and the system is highly non-linear. A problem with particle filters is that the particle filters require exponentially many more particles to represent high dimensional state spaces. Thus while the estimate will not degrade in high dimensionality, the computation required to make the estimate may be unreasonable, depending on the constraints of the problem.

Non-linear State Observation.

This section outlines the general framework of non-linear filtering theory. An upfront visual block diagram of the entire filter process is show in Figure 1. While many plants and measurement equations are well represented as LTI systems, many are not. If we assume that at a minimum the noise is additive then we can define a state predict and measurement function [74, 50]:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \tag{17}$$

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{v}_k \tag{18}$$

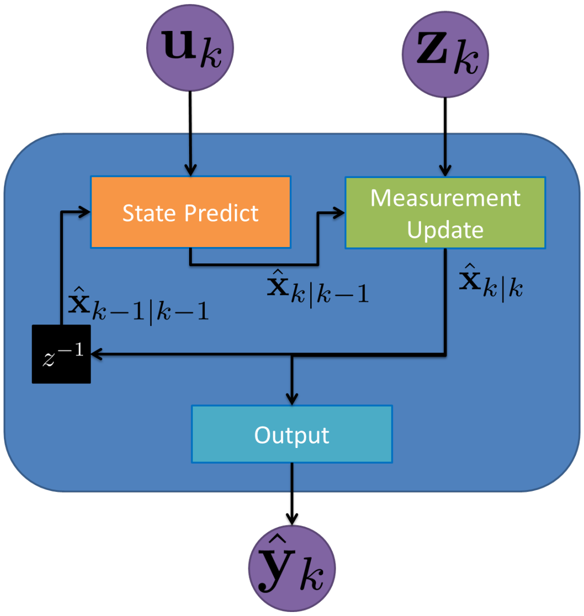

Figure 1: Block diagram of a generic standard filter. The blue rounded block represents the filter operation FILTER(). The overall filter has three components the state predict (orange block), measurement update (green block) and output (light blue block).

where $\mathbf{f}$ is a function that determines the next state based on the current state and system input. Likewise $\mathbf{h}$ is a function that determines the measurement based on the current state and system inputs, and $\mathbf{w}_k$ and $\mathbf{v}_k$ are independent white noise processes. From these equations we can estimate the next state and measurement assuming the noise values are zero mean:

$$\hat{\mathbf{x}}_{k+1} = \mathbf{f}(\hat{\mathbf{x}}_k, \mathbf{u}_k) \tag{19}$$

$$\hat{\mathbf{z}}_k = \mathbf{h}(\hat{\mathbf{x}}_k, \mathbf{u}_k) \tag{20}$$

A general filter uses the knowledge in the state predict and measurement equations to estimate the state. First the filter propagates the state to the time of a measurement $\hat{\mathbf{x}}_{k|k-1}$. Next the filter makes an estimate of the measurement $\hat{\mathbf{z}}_{k|k-1}$ given the propagated state. The filter can then update the state using the error between the actual measurement and the estimated measurement $\tilde{\mathbf{z}}_k = \mathbf{z}_k - \hat{\mathbf{z}}_{k|k-1}$. This step is called the measurement update. A filter in general may have output that is not simply the state values. In this case, an output function that takes only the states is used to transform the estimate of the state to the estimated output:

$$\hat{\mathbf{y}}_k = \mathbf{o}(\hat{\mathbf{x}}_k) \tag{21}$$

where the final output vector is $\hat{\mathbf{y}}$. For example if the state is a Earth Centered Earth Fixed (ECEF) coordinate the user may desire a Latitude Longitude Altitude (LLA) output. Thus, the output function makes the necessary conversion of coordinate frames. A complete filter combining the state predict, measurement estimate, measurement update, and output is denoted as:

$$\hat{\mathbf{y}}_k = \text{FILTER}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k|\boldsymbol{\theta}) \tag{22}$$

where the current output is based on the current input and measurement and last state value, and θ represents parameters of the filter function. The filter function will internally update the state variable from time step $k-1$ to k while it computes the output. In order to determine how to update the states based on the non-linear equations of the system the Kushner equation is required as described in the next section.

Kushner Equation.

The Kushner equation is a conditional probability equation for the state of a stochastic non-linear system [3, 5, 45]. The Kushner equation is conditioned on noisy measurements of outputs which reveal observability of the states. Thus the Kushner equation is a solution to the non-linear filtering problem in estimation theory. The Kushner equation assumes the states propagate according to the following Ito stochastic equations for an N state dynamics and M measurements:

$$d\mathbf{x} = \mathbf{f}(\mathbf{x}, t)\, dt + \mathbf{G}(\mathbf{x}, t)\, d\mathbf{w} \tag{23}$$

$$d\mathbf{z} = \mathbf{h}(\mathbf{x}, t)\, dt + d\mathbf{v} \tag{24}$$

where $\mathbf{x}$ is the state vector of size N, $\mathbf{z}$ is the measurement vector of size M, $d\mathbf{x}$ and $d\mathbf{z}$ are incremental changes in the state and measurements. The vector $\mathbf{f}(\mathbf{x}, t)$ is of size N and relates the current state to the next state, $\mathbf{G}(\mathbf{x}, t)$ is a $N \times Q$ matrix which describes the impact of noise on the states, and $\mathbf{h}(\mathbf{x}, t)$ is a vector of size M which relates the states to the measurements. Together, these two vectors and single matrix are functions of the current state and time. Further, $\mathbf{w}$ and $\mathbf{v}$ are independent Brownian motion processes with $\mathbf{E}[d\mathbf{w}(t)d\mathbf{w}(t)^T] = \mathbf{Q}(t)$ and $\mathbf{E}[d\mathbf{v}(t)d\mathbf{v}(t)^T] = \mathbf{R}(t)$. Thus, $d\mathbf{w}$ and $d\mathbf{v}$ are incremental changes in the Brownian motion process which makes $d\mathbf{w}$ and $d\mathbf{v}$ each a white Gaussian noise process. With this state and measurement equation the Kushner equation for the change in probability $dp(\mathbf{x}, t|\theta_k, \mathbf{Z}_t)$ of state $\mathbf{x}$ at time t is [45]:

$$dp = \mathcal{L}(p)\, dt + (\mathbf{h} - \hat{\mathbf{h}})^T \mathbf{R}^{-1}(t)(d\mathbf{z} - \hat{\mathbf{h}}dt)p \tag{25}$$

$$\theta_k = \{\mathbf{f}(\mathbf{x}, t), \mathbf{G}(\mathbf{x}, t), \mathbf{h}(\mathbf{x}, t), \mathbf{R}(t)\} \tag{26}$$

$$\mathcal{L}(p) = \frac{\partial p}{\partial t} = -\sum_{i=1}^{N} \frac{\partial}{\partial x_i}[pf_i] + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \frac{\partial^2}{\partial x_i \partial x_j}[p(\mathbf{GQG}^T)_{ij}] \tag{27}$$

$$\hat{\mathbf{h}} = \mathbf{E}_t[\mathbf{h}(\mathbf{x}, t)] = \int \mathbf{h}(\mathbf{x}, t)p(\mathbf{x}, t)dx$$

where θ_k is a given set of parameters describing the system, $\mathcal{L}$ is the Kolmogorov forward equation which propagates the probability into the future based on the current probability and state dynamics, $\hat{\mathbf{h}}$ is the mean estimate of the measurements given the state probability at the current time, and $\mathbf{Z}_t$ is the time history of measurements up to time t which is used to determine the incremental changes in measurements $d\mathbf{z}$. In this equation $(d\mathbf{z} - \hat{\mathbf{h}}dt)$ is the innovation term, or the residual–the difference between the actual measurement $d\mathbf{z}$ and expected measurement based on the current state $\hat{\mathbf{h}}dt$. Thus, the Kushner equation shows that for a non-linear system with white Gaussian noise processes corrupting the plant and measurements, there is a form for filtering the updates to improve the estimate of the states. This form specifically has a gain and a residual which is added to the current state estimate propagated by $\mathcal{L}(p)$. This gain value is determined by the system measurement equation $\mathbf{h}(\mathbf{x}, t)$, the measurement noise $\mathbf{R}(t)$, and the current states.

Later in Section III, this Kushner equation form will be exploited to determine the roles of ANN in filters. This approach will use empirical data to derive previously unknown relationships between the states, measurements, and output. The empirical information will then shape the ANNs to become the functions used in a filter. In order to go beyond current a priori knowledge and bring meaning to empirically collected states, measurements, and output data, a different approach is needed. The study of learning from empirical data is called machine learning. Machine learning allows us to learn new functions without explicitly knowing the system parameters $\{\mathbf{f}(\mathbf{x}, t), \mathbf{G}(\mathbf{x}, t), \mathbf{h}(\mathbf{x}, t), \mathbf{R}(t)\}$.

Machine Learning

Machine Learning are methods and techniques to calculate models when the form of the model is unknown [43]. Earlier in this paper, the background of estimation theory generally dealt with parametric models. A priori knowledge enabled us to know the form of the models, but some possible parameters in the model needed to be determined from external information. For example our laser ranger was missing a bias term which could be determined by a calibration. Our bias value may change, but the form of the model is always the same. Machine Learning uses models without a predetermined form to estimate hidden states from empirical data. The key difference is that the meaning of the parameters in the resulting machine learning models is not know to the user. The user does not understand what each parameter means in the machine learning created models. For example, we knew exactly the meaning of our bias term in the laser ranger problem, namely the bias in our sensor. In general, we will not know the meaning of the parameters in the machine learning created model. These models are trained with data beforehand in order learn how to replicate the empirical data. This empirical data contains input output pairs which we want our model to map. The learning process then develops a model that will generalize and map new input and output pairs. Learning of this form is called supervised learning because our model is trained to map these input to output pairs given example input output pairs.

A simple machine learning model is a linear regression model. This model multiplies each of the inputs by a coefficient and adds a bias term to determine a scalar output [43]:

$$\mathbf{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon \tag{28}$$

In this equation $\mathbf{x}$ is our vector outputs, and each β_p is a different coefficient for each of the input features or predictors x_p, except for β_0 which is the bias or intercept term. The model is trained to determine coefficients and bias that reduce the error as much as possible on the training data in order to accurately estimate future data. Now the form of our model is not determined by expert information based on a priori knowledge or system design. Instead we train and test this model to determine if it performs well enough for our problem. We never make the assumption that the true nature of the problem is explained by the model. We may also never understand the meaning behind the coefficients and bias.

The form of the linear regression model is the same for all linear regression models. The only difference is the values and the number of the coefficients and biases. These values are learned from empirical data. In our laser ranger example, our model would eventually learn to give the measurement coefficient β_1 a value near 1 and the bias coefficient β_0 close to the true calibration bias of the sensor. Thus, our model will learn the model we previously defined. In this case we happened to pick a model that was the same class as the one we defined before using our a priori knowledge. Machine learning becomes much more useful when we have no a priori information and cannot determine the form of the model beforehand.

There are many different kinds of machine learning models. What the models have in common is that they learn representations that are made to fit our observed training data. The models never have defined parameters to fill in such as calibration biases or physics constants. Machine learning created model have parameters which are determined form training data. These parameters usually have little meaning to the user especially as the models become larger and more complex. An example of a more complex class of machine learning model used by this dissertation is the ANN class of models.

Artificial Neural Networks

ANNs are a powerful class of machine learning models which can approximate any continuous function [38] given enough perceptrons. Perceptrons are the basic units of an ANN. A perceptron has many inputs and one output as shown in Figure 2. The inputs are either the actual inputs to the model or outputs of other perceptrons. The equation for a perceptron is:

$$y = f\left(b + \sum_{n=0}^{N-1} x_n w_n\right) \tag{29}$$

where N is the number of inputs to the perceptron, y is the output of the perceptron, x_n is the n^{th} input to the perceptron, w_n is the n^{th} weight, b is the bias, and f is some function called the activation function. The inputs to a perceptron are multiplied by a set of weights summed together along with bias. The bias can be thought of adding an extra input to the perceptron that is fixed at one and multiplying it by a weight as shown

$$y = f\left(\sum_{n=0}^{N-1} x_n w_n\right) \tag{30}$$

where x_0 is now forced to have a value of 1. We use this shorthand to refer to all the weights and biases as simply the weights. This is mathematically a dot product between the inputs to the perceptron and the weights. Next, an activation function is applied to the dot product result.

The most basic activation function is a linear function $f(x) = x$. However the power of an ANN is realized when a non-linear activation function is used. If only linear activation functions are used then the ANN can only represent linear functions. Non-linear activation functions are required for the ANN to model non-linear functions. Some common non-linear activation functions are the sigmoid function $f(x) = \sigma(x) = \frac{1}{1-e^{-x}}$ or the hyperbolic tangent function $f(x) = \tanh x = \frac{e^{2x}-1}{e^{2x}+1}$ [52] . In general, an activation function maps a scalar to another scalar. This allows the activation functions to operate element-wise on a vector. However some activation functions operate on vectors instead of scalars. For example, the softmax function transforms a vector into a valid probability distribution which sums to one

$$\text{softmax}(\mathbf{x})_j = \frac{\exp(x_j)}{\sum_{k=1}^{K} \exp(x_k)} \text{for} j = 1, \ldots, K \tag{31}$$

where $\mathbf{x}$ is some argument vector with size K. This function is often used as the activation function of the last layer of an ANN which is performing classification. Perceptrons are grouped together and connected to describe more complicated phenomenon many times in a layered format as described in the next section.

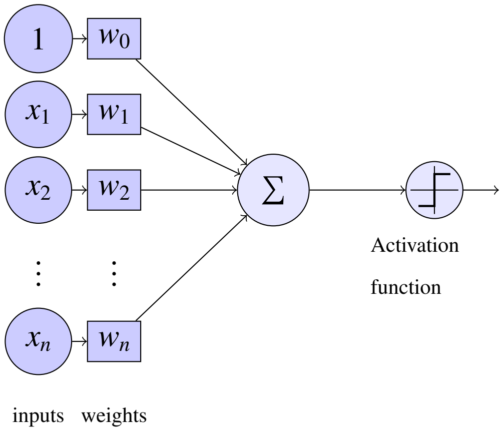

Figure 2: Basic Perceptron design. The inputs to the perceptron are x_n and the single scalar output is after the activation function. Each input x_n is multiplied by a weight w_n then summed together. The activation function performs a possibly non-linear operation on the sum.

Multi-Layer Perceptron.

Using perceptron outputs as the inputs of other perceptrons can forms graphs of perceptrons. Perceptrons in general can be connected in any way conceivable, but most ANNs use a specific architecture that has been proven to work [38, 58]. The simplest architecture is the fully connected layer architecture which makes Multi-Layer Perceptron (MLP) networks. In an MLP architecture, perceptrons are arranged in layers as shown in Figure 3. Each perceptron in a layer has as its input all the outputs of the previous layer. The perceptron then in turn passes its output to each perceptron in the next layer as shown in Figure 3. The middle layers in a MLP network are called the hidden layers. We denote the size of a layer as the number of perceptrons in that layer. Capacity is the expressive ability of the overall network which comes from the number of perceptrons in each layer and total number of layers. A network with larger capacity has more layers with more perceptrons in each layer. With a non-linear activation function and enough perceptrons, a one layer MLP can approximate any function [38, 36]. This is similar to how any signal can be reconstructed from the frequency components that make up the signal given enough frequency components. The output of each perceptron is a different kind of non-linear function which can be combined enough times to approximate any arbitrary function. The ability to approximate any function is an important property of the MLP architecture. This property means that as long as there is some underlying function to the data, it is possible for an ANN with enough capacity to learn to approximate that function. The real problem is how to train the network to represent the desired arbitrary function.

Training

Training an ANN involves learning the values for all the weights of each perceptron. The most common way to achieve this is to perform a version of Stochastic Gradient Descent (SGD) [58] such as RMS-Prop [15] or Adam [56]. SGD methods take the

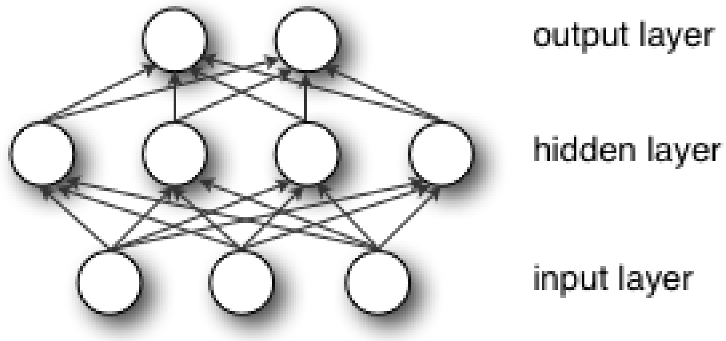

Figure 3: Basic MLP architecture for a one hidden layer neural network. The hidden layer and output layer circles are each perceptrons. The input values are passed to the hidden layer as inputs to each perceptron.

derivative of a loss function with respect to all the weights in the network. The loss function gives a value for the SGD optimizer to minimize. Next, the optimizer updates the weights in the negative direction of the gradient in order to minimize the loss function. This process is referred to as Back Propagation (BP), because the errors in the loss function propagate backwards to the weights in order to make weight updates. A learning rate is multiplied by the gradient to control magnitude of the weight change for each update. The BP algorithm [58] is an efficient algorithm that applies the derivative chain rule for all the weights while remembering previously calculate values. The BP algorithm allows efficient derivative calculation and makes training ANNs much faster. This is the same concept as how the FFT is much faster compared to a Discrete Fourier Transform (DFT) even though they preform the same task. Different variants of SGD exist to change the learning rate depending on different factors such as the momentum from the previous gradients [72, 87, 56].

Batch Update.

ANN training is done in batches of samples when using SGD. The loss function is evaluated for many input-output sample pairs. Next an overall weight update is determined from the current batch of input-output pairs. The batch update allows the training average over noise in one sample and make weight updates move more directly to the minimum. This is at the cost of computation since batch updates require more computation per weight update compared to a single sample update. Thus, updating weights for each pair individually allows the same magnitude weight change for less computation [104]. However, with parallelization the batch update can computed much faster than simply naively computing the updates in sequence, since the weight update calculation from each sample is independent from other samples. Calculating the weight updates in parallel reduces the overall computation time compared to calculating the weight updates sequentially. Parallelization now allows a batch weight update to be made in approximately the same time as a single weight update. Batch updates are done off line when all the sample data is available at once. If training is done on each sample as it is recorded, then it is called online learning.

Online learning.

Online learning is when the weights of an ANN are adjusted in real time as the data is recorded. Thus, the weights are updated for each new sample as the sample is recorded. One popular online learning method in the state estimation field is the Extended Kalman Filter (EKF) optimizer [32] [39] [51] detailed in Section II. The filter includes the weights of the ANN as its states. This allows the filter to use measurement updates to modify the weights of the ANN. The EKF optimizer works well as online learning where each measurement is slowly changing the weights of the ANN to represent the current environment. Online learning is most effective if applied after the ANN has accurate starting point for its weights. The online updates then correct small changes in the ANN to reduce the residual error.

Loss Functions

When training an ANN, a loss function is required to determine how well an estimate matches a target. This loss function is the target of an optimizer to minimize. In general, a loss function $\mathcal{L}$ only uses the output from the ANN and some target output value to return a metric which is a smaller value when the ANN output is more desirable. Different loss functions are used depending on the goal of the ANN. The loss functions will differ depending on the whether the output is discrete classification or regression.

If the goal is classification, then a loss function is usually the cross entropy between the true class distribution (usually only one class has a probability of 1) and the estimated class distribution. The loss function for cross entropy starts with the discrete cross entropy equation [87]:

$$H(p, q) = \mathbf{E}_p[-\log(q)] \tag{32}$$

where H is the cross entropy between the true distribution p, and the proposed distribution q (calculated by the ANN). The smaller the value of H the closer the proposed distribution is to the true distribution. If we assume the probability distributions p and q are vectors that give the probability for each class x of X classes we can make the cross entropy into a summation:

$$H(p, q) = -\sum_{x=0}^{X-1} p(x)\log(q(x)) \tag{33}$$

where p, q must sum to one to be valid probability distributions over all classes, x is the label for our discrete classification which we will treat as an integer from $0, 1, \ldots X-1$ where X is the total number of labels. Thus, $q(x)$ is the probability that class x is the true class. We have summed the cross entropy for all X possible class labels of our probability distributions p, q. Now we can use this definition of cross entropy to make a loss function $\mathcal{L}$ for all our samples:

$$\mathcal{L} = -\sum_{n=0}^{N-1}\sum_{x=0}^{X-1} p(x)\log(q(x)) \tag{34}$$

where p is the target probability vector and q is the predicted probability vector from the ANN. The loss function $\mathcal{L}$ sums up all the training examples from all N samples. We can simplify this equation when we restrict our p distribution to a vector that contains zeros except for one entry which is one, referred to as a one hot encoding or a vector of support one. This now makes the equation:

$$p(x) = \begin{cases} 0 & \text{if } x \neq x_t \\ \\ 1 & \text{if } x = x_t \end{cases}$$

$$H(p, q) = -\log(q(x_t)) \tag{35}$$

where x_t is the correct class label which has $p(x_t) = 1$. This simplification makes training an ANN much faster because the number of classes is not a factor for how long the loss function will take to complete. This loss function is calculated for each training sample and summed together to get a total loss for a given batch of N samples. Thus, the optimizer will attempt to minimize the loss by changing the output distributions $q(x)$ in order to more closely match $p(x)$. In order to change the distribution $q(x)$, the optimizer changes the weights of the ANN. However, since our $p(x)$ distributions are one hot vectors, the end result will be to attempt to maximize the probability of the correct class $q(x_t)$, which will drive our cross entropy loss function down (since $\log(1) = 0$). The optimizer will not just try to make $q(x_t)$ the highest value compared to the rest of the x's but it will keep being rewarded for increasing the true probability. This property allows a SGD algorithm follow the gradient of the loss function even when the correct class is the most probable.

If the goal of the ANN is regression, then Mean Square Error (MSE) is typically used as a loss function [87]. MSE compares the true output vector $\mathbf{y}$ to the estimated output vector $\hat{\mathbf{y}}$ for each training example.

$$\text{MSE}(\mathbf{Y}, \hat{\mathbf{Y}}) = 1/N \sum_{n=0}^{N-1} \left(\sum_{k=0}^{K-1} (y_k^n - \hat{y}_k^n)^2 \right)$$

where y_k^n is nth training example vector $\mathbf{y}^n$, and k represents the k^{th} index of the vector.

Mean Squared Error (MSE) is a popular loss function to use in machine learning due to a number of convenient properties [101]. MSE is a simple metric that requires only a few computations to compute, and has a clear relation to distance in two and three dimensions. MSE also has desirable optimization properties such as convexity, symmetry and differentiability. MSE also assumes the ordering of the dimensions and the signs or direction of the error are both irrelevant.

Stability and Generalization

Generalization error describes how accurately a model can perform on data which it has never seen before. In a learning algorithm, data which the model has never seen is data which was withheld from training and validation. This withheld data is usually called testing data, but can also be any future samples from the same distribution as the training and validation data. Training and validation data are any data which was used to determine the current state of the model including weights, architecture, and type. Usually training data is used for changing weights, and validation data is used for parameter selection. Parameter selection can include properties of the ANN (like the number of layers or layer size), as well as training parameters (such as the λ of L1 regularization explained in Section II).

Low generalization error means models will perform well on predicting future data. When generalization error is low, the model has generalized well to the underlying problem and has not over fit to the available training and validation data. The true model error can be expressed in terms of the input space $\mathbf{x}$ and output space $\mathbf{y}$ where every possible combination $\mathbf{z} = \{\mathbf{x}, \mathbf{y}\}$ is contained in set $z \in \mathbf{S}$ and one learned function $f_n(\mathbf{x}) = \hat{\mathbf{y}}$ is [2]:

$$I[f_n] = \int_{\mathbf{S}} V(f_n(\mathbf{x}), \mathbf{y})\rho(\mathbf{x}, \mathbf{y})d\mathbf{x}d\mathbf{y} \tag{36}$$

where V is a loss function describing the error between estimate $\hat{\mathbf{y}} = f_n(\mathbf{x})$ and the true output $\mathbf{y}$, and $\rho(\{\mathbf{x}, \mathbf{y}\})$ is the probability of obtaining that combination of input and output. This equation describes the total error over all possible input output combination in $\mathbf{S}$, but the probability of all those combinations ρ is usually unknown or difficult to estimate. Thus, a sample of the true probability is made by taking measurements to obtain a discrete sample set $\mathbf{S_i}$ of size N. With this sample set, the empirical error can be calculated by:

$$I_S[f_n] = \frac{1}{N}\sum_{i=i}^{N} V(f_n(\mathbf{x}), \mathbf{y}) \tag{37}$$

Thus the generalization error G is the difference between the empirical error (which is usually used to train a model) and the true model error (which is usually unknown):

$$G = I[f_n] - I_s[f_n] \tag{38}$$

While the generalization error usually cannot be computed directly, there are methods to estimate and bound the error. One measure of generalization is stability. Stability characterizes how small changes in the sample space $\mathbf{S_i}$ will change the model f_n. Two stability criteria can be used to determine how stable a model is and thus estimate its generalization error. The first stability criteria is called "leave one out cross validation". This criteria means that the empirical error must not change by some small values ϵ when one sample is removed from the training of f_n. In simple terms, one sample point must not have enough weight to significantly change the entire model f_n if it is removed. The other stability criteria is called "expected leave one out error stability". This criteria is met if the prediction of a sample is within some small value ϵ when the sample was used in training compared to if the sample was removed from training. In short, the algorithm should predict the same value for one point if it was included in training or not. [8]

These stability criteria can reveal possible over fitting by a model. When a model over fits, it has lowered the error on the training set by creating highly varying changes in its predictions that match only the training samples. An example of over fitting is fitting a polynomial to points on a line where the polynomial passes through all the points but does not accurately represent other points on the line. In this case over fitting occurs because the polynomial had enough capacity to represent all the sample points but represented the sample points by misrepresenting the underlying function. In order to generalize better, the model must have smoother curves to interpolate between training samples. The stability criteria are geared to stop this exact problem. For example, leave one out cross validation can reveal that the model over fits to the training points. The left out samples usually have poor prediction accuracy when the model is not stable. The poor accuracy occurred because the model was not smoothly interpolating between the training samples.

Thus, while generalization error is difficult to explicitly calculate, enforcing stability in the model can lower generalization error. Stability reduces generalization error by disallowing a model to over fit to training samples, which will generally allow better performance on samples not included in training. Regularization can enforce stability by penalizing models for properties that lead to instability.

Regularization

Regularization is methods and techniques that attempt to lower test error and thus generalize better by possibly increasing the training error. Regularization attempts to have the training error match the test error, which means the model is likely to generalize to unseen future data [43]. ANNs use many different forms of regularization to lower test error. This section outlines a few methods used in this dissertation.

A simple form of regularization is adding a loss term that is proportional to the size of the weights [43, 38]. Larger weights increase the penalty thus the model will try to reduce the magnitude of the weights while still trying to minimize the original loss function. This prevents the model from putting too much weight on a specific input path to predict the output. An L1 regularization function adds loss based on the absolute value of the weights:

$$L1_{reg} = \lambda_1 \sum_{n=0}^{N-1} |w_n| \tag{39}$$

where $|w_n|$ is the absolute value of the n^{th} weight, and λ_1 is a constant to weight the impact of this regularization term compared to the overall error. An L2 regularization function adds loss based on the square of the weights:

$$L2_{reg} = \lambda_2 \sum_{n=0}^{N-1} w_n^2$$

where λ_2 is a constant to weight the impact of this regularization term compared to the overall error.

Early stopping is allowing the training to stop early based on some criteria [43, 38]. Early stopping can be based on a validation set error to prevent the model from memorizing too many noise details about the training data that will not generalize. In this context, the training will stop when validation error has not decreased after a number of weight updates.

Dropout is a form of regularization used during training to force a model to find different ways of predicting the output [38, 92]. Dropout sets some weights in a neural network to zero during a training batch. All the other weights that were not dropped out are proportionally increased to keep the same magnitude of the final output vector. For example, if half the weights are set to zero the remaining weights are doubled to make up for the loss of the other weights. Dropout is performed on a probabilistic basis where each weight has a probability of being zero. The weights chosen to be zero are changed after each batch of samples are used to calculate a weight update. This prevents the model from relying on specific inputs as the only way to predict an output. When weights are dropped out, the current weight update will find a way to predict output without the input paths that had their weights dropped out. Thus the ANN will find alternative methods to use the input paths that were not dropped out to calculate the output. This makes the ANN more robust to learning only one input-output relation which may not always be true. After training all weights are no longer dropped out for prediction.

Other more complex form of regularization exist but they all attempt to limit the capacity of a model in an attempt to have lower generalization error [38, 43].

Principal Component Analysis

Principal Component Analysis (PCA) is the decomposition of multidimensional data into components by variability [43, 73, 109]. The multidimensional data consists of samples of vectors. Each index of the vector is a different dimension. A covariance matrix Σ between each dimension can be calculated as:

$$\Sigma = \frac{1}{N} \sum_{n=0}^{N-1} (\mathbf{x}_n - \boldsymbol{\mu})\,(\mathbf{x} - \boldsymbol{\mu})^T \tag{40}$$

where $\mathbf{x}_n$ is the n^{th} sample of vectors, and $\boldsymbol{\mu} = E[\mathbf{x}]$ is the mean vector of all the samples. PCA is a linear transform that makes the covariance matrix between the different dimensions a diagonal matrix. Thus, PCA determines the matrix **W** to transform each vector:

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{41}$$

where **y** is the transformed version of **x**. The PCA transform makes the underlying variability of the data orthogonal. However, the underlying data in each vector is the same. PCA also usually orders the new indices of the vectors by variability beginning with the largest variability. This is the same as ordering the covariance matrix to have the larger covariances be on the upper left.

An additional part of the PCA transform can be to whiten the data. Whitening forces all the variances to be the same magnitude. This addition to the transform makes the covariance matrix contain all ones as well as be diagonal. PCA can be used as an unsupervised learning technique to reduce dimensionality of data while maintaining variability [43]. PCA applied to the input data also makes training easier for ANNs. When all the variability in the input is made orthogonal, an ANN can more quickly determine what changes in the input are related to the output. Thus PCA can be applied to input as a preprocessing step to speed up learning for certain machine learning problems [43].

k-Means Clustering

k-Means clustering is an algorithm that attempts to classify n observations of some d dimensional space into k groups [30, 29, 43]. Each group k is centered on a point which serves as a prototype for that cluster k. All the observations are assigned a cluster by determining the closest prototype to that observation using some distance metric. The equation for the Euclidean distance metric is:

$$\underset{S}{\arg\min} \sum_{i=0}^{k} \sum_{\mathbf{x} \in S_i} ||\mathbf{x} - \mu_i||^2 \tag{42}$$

where S is the set of cluster prototypes for the k clusters, $\mathbf{x}$ is a vector in one of the clusters, and μ_i is the mean vector of the i^{th} cluster. The cluster prototypes are chosen in order to minimize the distances between the observations and their closest prototype. The minimization of the distance between cluster prototypes and the members of the clusters is usually done with an iterative algorithm. First, cluster prototypes are randomly initialized. Next, the members of each cluster are determined by the closest cluster prototype to the sample vector. Then the cluster prototype is moved to the mean of the current members of the cluster. Next, the cluster members are determined again with the new cluster prototype. Calculating the new mean, moving the cluster prototype, and determining members repeats until the cluster members do not change between iterations. However, the final cluster prototypes may be a local minimum. Thus, the algorithm repeats the overall algorithm starting with determining random cluster prototypes. The overall algorithm is repeated and the final set is the final cluster prototype set that occurred the most often.

The cluster prototypes partition can be represented by Voronoi cells using the center points of each cluster [88] (or polygon in 2D space). This dissertation uses k-Means clustering to turn a continuous output into a discrete output. The k-Means algorithm is applied to the output position space and labels each output point as belonging to one of the clusters. k-Means has now transformed the output position space from continuous position to discrete position bins. The discrete binned output may make training easier for certain data sets on certain machine learning problems [43].

Recurrent Neural Networks

In the simple form of a MLP ANN, the network is strictly a feed forward network [38, 24, 62]. Thus, feed forward networks make directed acyclical graphs. The MLP information must flow from the input layer to the output layer. A Recurrent Neural Network (RNN) removes this restriction and allows networks to make connections which will create cycles as shown in Figure 4. With cycles it is possible to return to a starting point after traversing other perceptrons. This introduces a memory or state to the neural network. For example if a perceptron has as an input its own output, then it must remember the last output in order to use it as an input. This output value from the last time step must be stored in memory for the next time step. Thus the stored value will change the output even with the same inputs to the overall RNN. The previous memory or state of the ANN is denoted as $\mathbf{h}_k^{NN}$ at time step k. The previous state makes the ANN equation:

$$\mathbf{y}_k = \mathrm{a}^{NN}(\mathbf{x}_k, \mathbf{h}_{k-1}^{NN} | \boldsymbol{\theta}) \tag{43}$$

This is the equation for general ANN with function a^{NN} which is parameterized by $\boldsymbol{\theta}$. Note that he inputs $\mathbf{x}_k$ and outputs $\mathbf{y}_k$ now have time indices k because the order in which the inputs are given will change the outputs. The output equation for this one layer would be:

$$\mathbf{y}_k = \mathbf{W}_l \mathbf{x}_k + \mathbf{b} + \mathbf{W}_r \mathbf{h}^N N_{k-1} \tag{44}$$

where the RNN layer has weights $\mathbf{W}_l, \mathbf{W}_r$ and biases $\mathbf{b}$, and $\mathbf{h}^N N_{k-1}$ is the output of the layer at the previous time step. While arranging RNN into layers is common, RNNs can take on many more architectures because they are not forced to be feed forward networks. Keeping a state now makes every output dependent on the entire history of inputs which are compacted into the previous state of the RNN. This allows a RNN with enough perceptrons

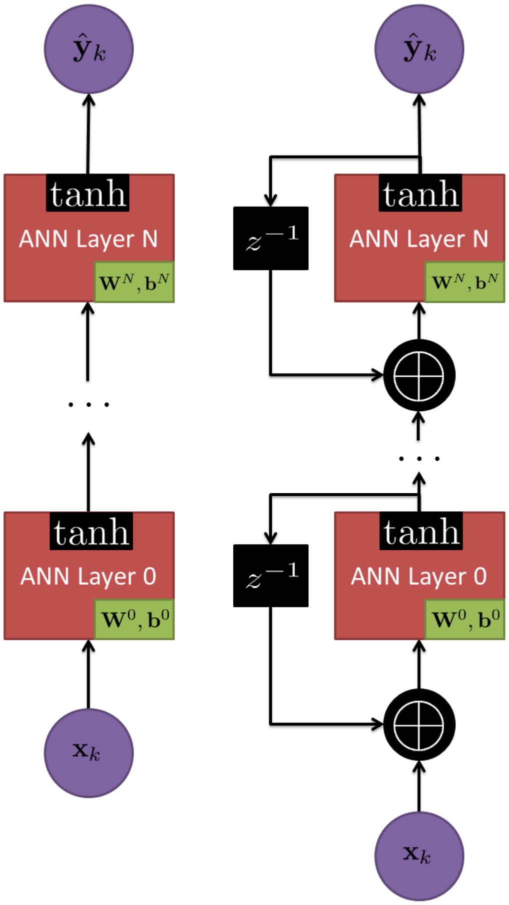

Figure 4: Left – the architecture for a feed forward neural network given for reference. Right — a recurrent neural network network. Note a cycle is created from the concatenation of the output of the layer and the input of the same layer.

or capacity to become a function which maps from any given input sequence to output sequence. This is analogous to how a feed forward ANN could become a function to map any input to any output [38]. Once again the real problem is how to train such a network to be the function we desire.

RNNs have an added complication now because they have a time dependence. When applying a derivative to the network with respect to the weights, the chain rule must be applied back in time to determine how the previous inputs changed the current output. Applying the chain rule through the full time dependence to update weights is referred to as Back Progatation Through Time (BPTT). This repeated chain rule causes one of two problems, the vanishing gradient problem [35, 38] and the exploding gradients problem [26, 60]. While these problems do exist in feed forward networks, the problem is amplified in RNNs. In feed forward networks the problems arose because the back propagation algorithm had to propagate through many layers. In the RNN, the back propagation algorithm has to flow through layers and time steps. The more time steps the network operates on, the more likely the gradient is to vanish or explode. This is because the chain rule will compute longer and longer products depending on how far back the in time the chain rule has been applied. The final value of these products has a tendency to vanish to zero if the weights are less than one or explode to very large numbers if the weights are greater than one. The exploding gradient problem can be mitigated by making a maximum gradient value and clipping all larger gradients to this maximum number [95]. However, if the gradient vanishes then it cannot be recovered. This leads to a central problem in the supposed power of RNN which is their ability to learn output sequences on the full history of inputs. If state information is lost over time, then the RNN has lost the ability to relate past inputs to current inputs when the time separation is too large [24]. In order to solve this problem, specific architectures of RNNs have been developed such as Long Short Term Memory.

Long Short Term Memory

LSTM is a type of RNN architecture created to overcome the vanishing gradient problem and achieve very long term dependencies in parsing input sequences [35, 84]. The key component of LSTM networks is the addition of gates to modify data stored in LSTM memory cells. A gate is an MLP layer with a sigmoid output. This forces all the gate output to between 0 and 1. The gates are used to determine what memory cell data should be passed through the gate and what data should be erased. Gate operations are done by an element wise multiplication of the memory cells and the gate output. The LSTM also has a standard MLP layer called the input MLP which determines what to add to the memory cells. We denote the size of the LSTM layer as the number of the memory cells which matches the sizes of the input MLP and gate ANNs. For example, if the LSTM has size of 10, then there will be 10 memory cells, and thus the gates and input MLP will output 10 values, one for each memory cell. The standard LSTM architecture has three of these

gates called the forget, input, and output gates as shown in Figure 5. Each gate and the input MLP take as an input the current LSTM layer input $\mathbf{x}_k$ and the output $\mathbf{y}_{k-1}$ of the last time step. The forget gate determines to what amount $\mathbf{f}_k$ the LSTM should forget the value of the memory cells. The second gate is the input gate. The input gate produces a filter $\mathbf{i}_k$ to filter the output of the input MLP $\tilde{\mathbf{c}}$. This filtered output from the input MLP is then added to the memory cells. The memory cell is then put through a tanh to become the output of the LSTM. However, before the output is passed outside the LSTM block, the output gate filters the memory cells. This is now the final output of the LSTM which is the passed on to the next layer and fed back as input to the gates and input MLP. The full equations for the LSTM are:

$$\mathbf{f}_k = \sigma(\mathbf{W}^f[\mathbf{x}_k \oplus \mathbf{y}_{k-1}] + \mathbf{b}^f) \tag{45}$$

$$\mathbf{i}_k = \sigma(\mathbf{W}^i[\mathbf{x}_k \oplus \mathbf{y}_{k-1}] + \mathbf{b}^i) \tag{46}$$

$$\mathbf{o}_k = \sigma(\mathbf{W}^o[\mathbf{x}_k \oplus \mathbf{y}_{k-1}] + \mathbf{b}^o) \tag{47}$$

$$\tilde{\mathbf{c}} = \tanh(\mathbf{W}^c[\mathbf{x}_k \oplus \mathbf{y}_{k-1}] + \mathbf{b}^c) \tag{48}$$

$$\mathbf{c}_k = (\mathbf{c}_{k-1} \odot \mathbf{f}_k) + (\mathbf{i}_k \odot \tilde{\mathbf{c}}_k) \tag{49}$$

$$\mathbf{y}_k = \tanh(\mathbf{c}_k) \odot \mathbf{o}_k \tag{50}$$

In these equations, $\mathbf{x}_k, \mathbf{y}_k$ are the input and output vectors of the LSTM at time step k, $\mathbf{f}_k, \mathbf{i}_k, \mathbf{o}_k$ are the outputs of the forget, input and output gates at time step k,

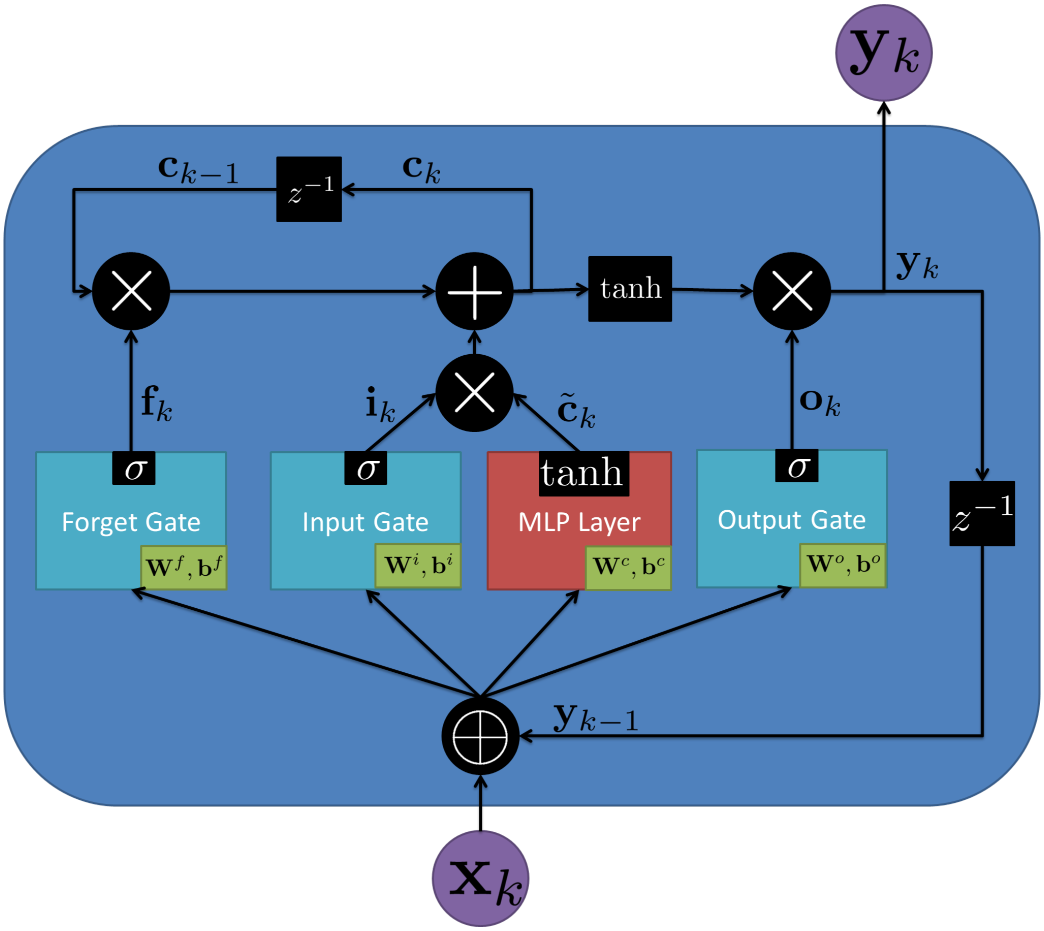

Figure 5: One LSTM layer broken into its component parts. The overall blue rounded rectangle is the LSTM layer. The blue rectangles are the forget, input and output gates. The red rectangle is a standard MLP layer. The + and × symbols denote element-wise addition and multiplication while the ⊕ symbol denotes vector concatenation. The z^{-1} denotes a delay of the value between time steps.

$\mathbf{W}^f, \mathbf{W}^i, \mathbf{W}^o, \mathbf{b}^f, \mathbf{b}^i, \mathbf{b}^o$ are the weights and biases of the forget, input and output gates, $\tilde{\mathbf{c}}$ is the output of the input MLP, $\mathbf{W}^c, \mathbf{b}^c$ are the weights and biases of the input MLP, $\mathbf{c}_k$ is the state of the memory cell at time step k, $\oplus$ is vector concatenation, $\odot$ is element wise multiplication, and σ is the sigmoid function.

One of the most important changes to the architecture is how the previous state and the input are combined. In a general RNN architecture, the previous state would be multiplied by weights before replacing the old state value. The LSTM architecture only multiplies the state memory cells by gate outputs which are limited to between 0 and 1. The LSTM can only increase the state memory cells value with addition. This addition prevents the vanishing gradient problem from shrinking or exploding the gradient, because there is no longer a product to be calculated. As Back Progatation Through Time follows the state value from one time step to the last, there is no longer an ever expanding product but a simple summation. This is one of the key advancements that enabled training to make the LSTM architecture remember states for many time steps [38].

Other types of LSTM architectures exist with various slight changes [13, 22, 107]. One variation is to change the inputs to the gates by adding the current cell state as an input [22]. These are called peephole connections which allow the gates to determine what to forget, input, or output based on the current cell state. Other variations combine the input and forget gates to make what is called a Gated Recurrent Unit (GRU) architecture [13]. The idea with GRU is that forget and input need to work together to forget something and then put something else in its place. The input MLP activation function of the LSTM is not required to be a tanh function. However, the gates must have a sigmoid activation functions in order to maintain an output range between zero and one. While many types of LSTM architectures exist, the core elements stay the same. The gates forget or keep information, and the current state is updated with addition. This core idea allows the LSTM architecture to connect state information over many time steps to achieve high performance in sequence processing compared to simple RNN layers.

Residual Connections

Residual connections used in residual neural networks or ResNets were first introduced by [34] and later improved by [33]. The basic idea of a residual connection is to add skip connection that bypass a layer or layers or an ANN and skip to a later part in the network. The skip connection is combined with this output from a later layer usually with some kind of addition. This allows the layers that were skipped to learn the residual values between the original input. Simply put the concept is that it may be easier to learn the change from some input to an output instead of the output directly. The advantage of this is that the skip connection allows the gradient to easily reach layers far from the output layer and still contain useful information alleviating the problem of vanishing gradients. This enables training very deep networks much easier compared to standard networks with no residual connections. In the simple case where the skip connection matches the output in size the equation would be:

$$\mathbf{y}_l = \mathbf{x}_l + \mathbf{f}^{NN}(\mathbf{x}_l, \boldsymbol{\theta}) \tag{51}$$

$$\mathbf{x}_{l+1} = \mathbf{y}_l \tag{52}$$

where $\mathbf{y}_l$ is the output of layer l, $\mathbf{x}_l$ is the input to layer l and $\mathbf{f}^{NN}(\mathbf{x}_l, \boldsymbol{\theta})$ is some layer of an ANN with weights $\boldsymbol{\theta}$. Thus the output of the ANN layer is added to the input to obtain the final output which is the input to the next layer. Note that if the size of $\mathbf{x}_l$ is not the same size as $\mathbf{f}^{NN}(\mathbf{x}_l, \boldsymbol{\theta})$ then some method will be required to combine the two vectors into an output vector $\mathbf{y}_l$. Different methods exist from padding the smaller vector to the larger vector size, or introducing a dense MLP layer or 1D convolution layer to resize one of the vectors. In summary residual connection allows training much deeper ANN compared to standard feed forward ANNs.

Teacher Forcing

When training a RNN there is a class of problems where the output at the last time step is used as an input for the next time step. An example of this is an equation of the form:

$$\mathbf{y}_k = \mathbf{f}(\mathbf{y}_{k-1}, \mathbf{x}_k | \boldsymbol{\theta}) \tag{53}$$

where the output $\mathbf{y}_k$ at time step k is a function of the last output $\mathbf{y}_{k-1}$ and some input $\mathbf{x}_k$ at time step k. When output estimates are used to predict future output in an ANN this equation becomes:

$$\hat{\mathbf{y}}_k = \mathbf{f}^{NN}(\hat{\mathbf{y}}_{k-1}, \mathbf{x}_k|\boldsymbol{\theta}) \tag{54}$$

where the output is replaced by the estimates of the output. In this case, an ANN will be trained to determine the function $\mathbf{f}^{NN}$ using an optimizer to compare the true output with the estimated output. The ANN will be trained with data from the true output values $\mathbf{y}$ and inputs $\mathbf{x}$. Clearly, the estimate at future time steps $k_1 > k_0$ depends on past estimates from time $k_1 \leq k_0$. When the ANN has a poor estimate of $\mathbf{f}^{NN}$, the future estimates of $\mathbf{y}$ will be very poor. The poor estimates of $\mathbf{y}$ will make it very difficult for the ANN to train using all

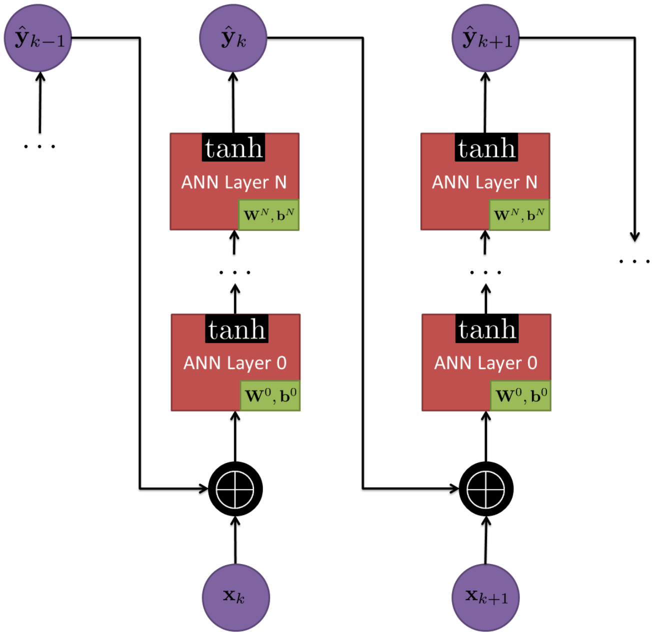

Figure 6: Global Feedback method of training an ANN. Note that the previous estimate from the last time step is concatenated with the current input to predict the output at the current time.

the available supervised output. This is because we do not expect the ANN to have such poor estimates of **y** for all the training data points when training is complete. Rather, we expect that when the ANN is fully trained, our estimates are very good, and we want to train using these better estimates. Using the last estimated state in this manner is called Global Feedback (GF) [77], as shown in Figure 6. Teacher Forcing (TF) uses the last true output

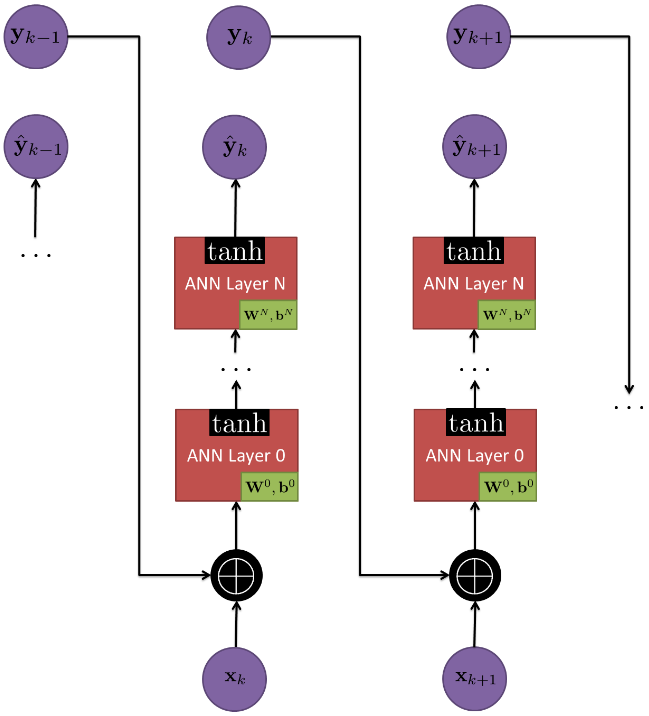

Figure 7: Teacher Forcing method of training an ANN. Note that while an estimate of the last time step is made the true output at the last time step is concatenated to the current input to predict the current output.

$\mathbf{y}_{k-1}$ as an input, instead of the last estimated output $\hat{\mathbf{y}}_{k-1}$, as shown in Figure 7 [38, 46, 32]. This allows the ANN to train much more quickly, because all the training samples now have good inputs which are close to the true values. This will make a much better current output estimate especially early in training when the estimate is poor. However, this can create problems with the training. For example, if the ANN is stable with teacher forcing, it may not be stable when the ANN is used in practice and global feedback is used instead. Another problem is if the ANN pathologically uses only the true last output and no other input to determine the next output. In our example, this would be like ignoring the inputs **x** and relying only on the output **y**. This will create problems when the truth is replaced with the estimates, and the ANN drifts from the truth and cannot correct the drift.

Different techniques can be used to combat the problem of over fitting to the true data. The first is to only use teacher forcing for some beginning time steps and then move to global feedback. This allows the ANN to jump start with perfect input and then have to correct itself to work with its estimated outputs [77].

In short, teacher forcing is a training technique used when the inputs to an ANN contain some of the outputs it estimates. This type of ANN will show up when the ANN mimics certain filter functions that use estimates as input such as state prediction or propagation functions.

Stratified Training

One problem of some datasets is when class samples are very unequally distributed. When training using a loss function the optimizer will exploit the classes that occur less and not prioritize their accuracy in order to focus on classes that occur more frequently. This leads to an undesirable property where classes that occur less have an artificially low accuracy at the cost of high accuracy in other classes.

Another problem that occurs more acutely in online training but also in off line training is recency. Recency is when the optimizer make large weight changes in response to new data at the cost of older data. In this case the recent data has good performance but older data now has worse performance.

In order to alleviate these two problems, the training samples are often stratified so the class samples appear about equally in a batch [44]. In RNNs each training sample is a sequence of samples. Thus each batch should be stratified across sequences. Sometimes there many be only a few but long sequences. This creates a stratification problem where each batch can be skewed towards a certain class. To combat this problem, each sequence is reordered to guarantee class diversity over a batch. This approach in effect can shuffle the sequences since they may no longer occur in their natural order. This is similar to Multistream in [32] where the sequences were randomly shuffled but no guaranteed to be have diverse output.

Mixture Density Networks

Mixture Density Networks are a special kind of ANN that outputs the parameters for a mixture of distributions to create a probability distribution [7, 103, 86, 25]. The basic form of the MDN is as follows:

$$\mathbf{z}^i = f(\mathbf{x}^i, \mathbf{w}) \tag{55}$$

Where f is the MDN function evaluated with weights $\mathbf{w}$ at the i^{th} input sample $\mathbf{x}$. The vector $\mathbf{z}^i$ represents the parameters of the distribution for the i^{th} sample calculated by the MDN. An MDN minimizes the negative log likelihood of the probability of the output distribution at the target output over all the targets as shown:

$$\mathcal{L} = \sum_{i=0}^{N-1} - \ln\left(p(\mathbf{y}^i|\mathbf{z}^i)\right) \tag{56}$$

where $\mathcal{L}$ is the loss function, N is the total number of samples and $\mathbf{y}^i$ is the true target output for the i^{th} sample. Thus, the MDN is attempting to maximize the probability of each target for its given parameters across all samples. Since our parameters are a function of the inputs, this in effect will allow the MDN to approximate the probability of the output target given the input denote as $p(\mathbf{y}|\mathbf{x})$. However, this loss function in Equation 56 is more general than required because we are specifically interested in *mixture* density networks, so we can rewrite Equation 56 with mixtures:

$$\mathcal{L} = \sum_{i=0}^{N-1} - \ln\left(\sum_{j=0}^{M-1} w_j p(\mathbf{y}^i|\mathbf{z}^i_j)\right) \tag{57}$$

where we have now explicitly called out the weight parameter w_j of each set of M parameters calculated by the MDN. The sum of the weight parameters must be one and is usually enforced by a softmax function. The parameters from the MDN are broken into parameters for each j^{th} mixture as $\mathbf{z}^i_j$. Thus, for each mixture distribution, the probability of that set of parameters is calculated and then weighted for that specific mixture. This is now general enough to allow any number of mixtures of any type of distribution. However, this dissertation focuses specifically on MDNs that output Gaussian or normal distribution parameters.

Gaussian Mixture Models

GMMs provide a parameterized method to define distributions which can take on different shapes such as multi-modal distributions. GMMs use a number of different Gaussian distributions added together to represent one full distribution. The equation for the probability of a Gaussian Mixture Model is as follows [21, 27, 83]:

$$p(\mathbf{y}|\{w_{0 \ldots M-1}, \boldsymbol{\mu}_{0 \ldots M-1}, \boldsymbol{\Sigma}_{0 \ldots M-1}\}) = \sum_{j=0}^{M-1} w_j \mathcal{N}(\mathbf{y}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{58}$$

$$\sum_{j=0}^{M-1} w_j = 1 \tag{59}$$

$$\mathcal{N}(\mathbf{y}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{y} - \boldsymbol{\mu})\right)}{(2\pi)^{D/2} |\boldsymbol{\Sigma}|^{1/2}} \tag{60}$$

where D is the number of dimensions estimated by each of the M Gaussian mixtures, $\boldsymbol{\mu}_j$ is a vector of the means in each dimension, and $\boldsymbol{\Sigma}_j$ is the positive semi-definite covariance matrix between each dimension. The subscript j refers to the mean or covariance of the j^{th} mixture. The target vector $\mathbf{y}$ is the point whose probability will be evaluated. Each Gaussian distribution is given a weight w_j with the condition that all the weights sum to one as shown in Equation 59. This allows the summation of the probability from all the Gaussian distributions to be one valid probability distribution. To calculate the probability of an output, each Gaussian distribution would return its own probability using its own mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, then each of those probabilities would be weighted by the weights.

The MDN will output these parameters to parameterize the GMM:

$$\mathbf{z} = \begin{bmatrix} \boldsymbol{\mu}_0 & \boldsymbol{\Sigma}_0 & w_0 & \boldsymbol{\mu}_1 & \boldsymbol{\Sigma}_1 & w_1 & \dots & \boldsymbol{\mu}_{M-1} & \boldsymbol{\Sigma}_{M-1} & w_{M-1} \end{bmatrix} \tag{61}$$

where all the parameters of the Gaussian distribution have been flattened into a single vector which is outputted by the MDN. This now leads to a general GMM loss function of:

$$\mathcal{L}_{GMM} = \sum_{i=0}^{N-1} - \ln\left(\sum_{j=0}^{M-1} w_j^i p(\mathbf{y}^i | \boldsymbol{\mu}_j^i, \boldsymbol{\Sigma}_j^i)\right) \tag{62}$$

In general the covariance matrix $\boldsymbol{\Sigma}$ has dependent terms and thus can be expressed with fewer terms than the full matrix. This reduces the total number of terms in $\mathbf{z}$ as follows. For each distribution in the GMM there are D means, $D(D + 1)/2$ independent terms in the covariance matrix (due to symmetry), and one weight for a total of $D(D + 3)/2 + 1$ parameters. Thus, the entire distribution has a total of $MD(D + 3)/2 + M$ parameters.

This dissertation focuses specifically on MDNs that output GMM parameters. While the parameters for a GMM are above, the MDN may not determine those parameters explicitly but an equivalent representation that makes computation easier. Thus, the actual parameters determined by the MDN are dependent on the loss function used by the MDN to evaluate the target output for that probability distribution. For example, Graves et al. [24] does not estimate $\boldsymbol{\Sigma}$ directly but instead uses a parameterization where the MDN determines standard deviation σ_i for each i dimension of the possible D dimensions as well as the correlation coefficient $\rho_{i,j}$ between each combination of the i^{th} and j^{th} dimension. Finally, these parameters are repeated M times for all M mixtures. As stated above, to parameterize the distribution we require a covariance matrix $\boldsymbol{\Sigma}$ which can be determined by the parameters σ_i and $\rho_{i,j}$ as follows:

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_0^2 & \sigma_0\sigma_1\rho_{0,1} & \sigma_0\sigma_2\rho_{0,2} & ... & \sigma_0\sigma_j\rho_{0,j} \\ \sigma_1\sigma_0\rho_{1,0} & \sigma_1^2 & \sigma_1\sigma_2\rho_{1,2} & ... & \sigma_1\sigma_j\rho_{1,j} \\ \vdots & \sigma_2\sigma_1\rho_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \sigma_i^2 & \sigma_i\sigma_j\rho_{i,j} \\ \sigma_i\sigma_0\rho_{i,0} & \sigma_i\sigma_1\rho_{i,1} & ... & \sigma_i\sigma_j\rho_{i,j} & \sigma_i^2 \end{bmatrix} \tag{63}$$

where i, j is the row and column index respectively and $\rho_{i,j} = \rho_{j,i}$ and each ρ is limited to the bounds of $[-1, 1]$, also each σ is restricted to be strictly positive.

These restrictions are enforced by an activation functions at the last layer of the MDN. The ρ values are put through a tanh function and the σ value are put through an exp function. Also the weights are put through a *softmax* function [9] to ensure the weights sum to one. These activation functions force the GMM parameters to produce a valid probability distribution that will integrate to one.

This form from Graves is straightforward to understand and directly calculates values of interest. However, this form may have numerical computation difficulties when implemented. For example, the probability equation for a GMM uses the inverse of the covariance matrix Σ^{-1}. Thus Σ must be inverted for each parameter set for every calculation of probability.

Inverting the covariance matrix is a costly calculation. Thus, in this dissertation we use an inverse covariance form in [103] to represent $\mathbf{\Sigma}$. This form uses an upper diagonal matrix $\mathbf{A}$ to hold the independent values used to calculate $\mathbf{\Sigma}$. The $\mathbf{A}$ matrix is multiplied by its transpose to determine directly an inverse covariance matrix as follows:

$$\mathbf{\Sigma}^{-1} = \mathbf{A}^T\mathbf{A} \tag{64}$$

where the diagonals of $\mathbf{A}$ are forced to be strictly positive by applying the exp function to each diagonal as the final activation of the MDN. The resulting $\mathbf{\Sigma}^{-1}$ matrix is guaranteed to be positive semi-definite and thus have an inverse [103] (as long as none of the diagonals of $\mathbf{\Sigma}^{-1}$ become exactly zero). Thus we can calculate $\mathbf{\Sigma}$ if desired. The advantage is that $\mathbf{\Sigma}^{-1}$ can be used directly to determine probabilities for the GMM in Equation 58 which reduces the computation required to invert the matrix for every probability calculation. If the diagonal is guaranteed to be strictly greater than zero, then $\mathbf{\Sigma}^{-1}$ is guaranteed to be positive definite. However, numerical precision may round $\exp x$ to zero for large negative x and thus create a positive semi-definite matrix.

With the ability to construct valid covariance matrices, an ANN can be trained to develop a full parameterization of a GMM. This allows an ANN to output different probability distribution for a multidimensional random variable. This covariance training can also be used to determine covariance matrices for other problems, for example the covariance matrix of a Kalman filter.

State of the Art ANN Integration

This section outlines the history of integrating ANNs into filters. State estimation and filtering theory have incorporated ANNs for many different purposes since ANN interest started returning in the later 1980s and early 1990s. These different integration methods will be directly fit into the overall architecture outline in Chapter III. At the end of Chapter III the history will be revisited to show exactly how previous integration fits into the overall architecture.

The initial use of ANNs for state estimation and filtering theory was interested in system identification as found in Chen et al. [11] and Narendra et al. [70]. By system identification, the research sought to model the system dynamics or state predict equation with ANNs. In equation form this is:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) \rightarrow \hat{\mathbf{x}}_{k|k-1} = \mathbf{f}^{NN}(\mathbf{x}_{k-1}, \mathbf{u}_k) \tag{65}$$

where $\mathbf{f}()$ is some informed system dynamics based of prior knowledge, and $\mathbf{f}^{NN}()$ is some ANN trained on empirical data to minimize the propagated state error. This continues to be a topic of interest as shown by various implementation in Dreyfus et al. [19], Ljung et al. [63], Elanayar et al. [20], Puskorius et al. [79], Sastry et al. [85], Szilas et al. [96], Moon et al. [68], Ngia et al. [72], Mandic et al. [64], Rajesh et al. [81], Lee et al. [59], and Almonacid et al. [4]. Many different kinds of ANNs such as RNNs [59] and are used with different learning techniques like online and offline learning. The goal of these methods is to achieve better performance modeling the system dynamics with an ANN.

Shortly after modeling the system dynamics was started, modeling of the entire filter with an ANN began. Early filter ANN modeling started with Nerrand et al. [71], followed by Ting-Ho Lo [99], and then Teissier et al. [98]. In equation form this distinction is:

$$\hat{\mathbf{x}}_k = \mathbf{s}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k) \rightarrow \hat{\mathbf{x}}_k = \mathbf{s}^{NN}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k) \tag{66}$$

where $\mathbf{s}()$ is some informed filter model based on prior knowledge, and $\mathbf{s}^{NN}()$ is some ANN trained on empirical data to minimize the filtered state error. Recent interest from the state estimation and filtering communities is shown in Wilson et al. [105] and Draganov [18]. Recent interest in modeling filters as ANNs exploits the work done by the machine learning community that has made RNN and ANNs in general much easier to implement and train. Usually the goal of these methods has been to improve the final filter performance by modeling the filter as an ANN [99]. However, Draganov [18] used an ANN to condense a detailed particle filter with many states into a more lightweight implementation.

Along the same idea of modeling the system dynamics or filter as an ANN, Parlos et al. [77, 76] developed a decomposed ANN filter model where ANNs model functional components of filters. While the early research on system identification continued to refine developing propagation equation, Parlos et al. also replaced the measurement update equations with ANNs. In equation form this looks like

$$\hat{\mathbf{x}}_k = \mathbf{mu}(\mathbf{x}_{k|k-1}, \mathbf{z}_k) \rightarrow \hat{\mathbf{x}}_k = \mathbf{sf}^{NN}(\mathbf{x}_{k|k-1}, \mathbf{h}^{NN}(\mathbf{x}_{k|k-1}), \mathbf{z}_k) \tag{67}$$

where $\mathbf{mu}()$ is some informed filter measurement update based on a known filter, $\mathbf{h}^{NN}()$ is an ANN trained to predict measurements from the current state and $\mathbf{sf}^{NN}()$ is an ANN trained to filter the state based on the current state, measurements, and predicted measurements from $\mathbf{h}^{NN}()$. This decomposition allowed the ANN to determine how to filter the measurements in order to improve the current state estimate. Further research implemented this idea for other purposes such as Rajagopal et al. [80] and Srinivasan et al. [91]. However, this dissertation in Section III will further break down the network put forth by Parlos et al. using non-linear filtering theory. The objective of the decomposition is to gain a theoretical backing for a specific architecture. Also, the ANNs in the decomposed architecture should be easier to train because the ANN goals are as specific as possible. While research continued in modeling functions with ANN equivalent functions, other research sought to maintain standard filters and allow ANN to only provide corrections.

Instead of replacing standard filter functions with ANNs, the filtering community started to augment the filters with ANNs. In contrast to the early work which let the ANN become the system dynamics, neural network aided filters start with known equations and aid or correct the state estimate with ANNs. Early work in aiding the filters added a correction term to the filter state output. This allows the user to have a standard filter base with small corrections applied by an ANN to improve performance. In equation form this is:

$$\hat{\mathbf{x}}_k = \text{FILTER}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k) \rightarrow \hat{\mathbf{x}}_k = \text{FILTER}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k) + \text{FILTER}_c^{NN}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k) \qquad (68)$$

where FILTER() is some informed filter model based on prior knowledge, and $\text{FILTER}_c^{NN}()$ is some ANN trained to minimize the error between the estimated state and the true state. Chin et al. [12] contributed some of the early research in this approach to improve filters for multi target tracking. This type of integration is also shown in Vaidehi et al. [100], Bhattacharya et al. [6], Jwo et al. [47], Yu et al. [108], Li et al. [61], and Katayama et al. [53]. While the filter state is corrected at each time step, all the research listed does not use the corrected estimate as the starting point of filter for the next time step. This means that the ANN corrected state is only given to the user. The filter only works with the uncorrected state in the state dynamics and measurement update functions. Some papers are unclear if the correction is only applied to the output, but no paper explicitly shows the corrected state as the initial state of the next time step for the filter.

Another example of augmenting a filter with an ANN is to correct the state predict equation of the filter. This narrows down the scope of what the ANN can correct and can be applied to state estimation where filters are not used. In equation form this is:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) \rightarrow \hat{\mathbf{x}}_{k|k-1} = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{f}_c^{NN}(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k) \tag{69}$$

where $\mathbf{f}()$ is some informed system dynamics based on prior knowledge and $\mathbf{f}^{NN}()$ is an ANN trained to aid or correct the errors from $\mathbf{f}()$ to minimize the error between the state estimate and true state. Stubberud et al. [93] contributed early research into this type of ANN integration. Interest in this topic continued with many different publications such as Horton et al. [37], Zhan et al. [110], Choi et al. [14], Xu et al. [106], Talebi et al. [97], and Kang et al. [52]. There is a nuanced difference between the filter output correction and state predict correction. Both corrections change the overall filter output but in slightly different ways. Correcting the filter output can correct errors introduced by the measurement update, while the system dynamics correction must still be updated by measurements. Also, correcting the filter output is usually not fed back into the filter. However, system dynamics corrections do improve the internal filter state.

The next method outlined in this dissertation is to process measurements with an ANN. In general this applies to any measurements that are processed by ANNs before being used in a filter. However, this dissertation is specifically interested in situations where an ANN transforms measurements a filter cannot incorporate into measurements the filter can incorporate. In equation form this would look like:

$$\mathbf{z}_k = \mathbf{z}_k^f \rightarrow \mathbf{z}_k = \mathbf{z}_k^f \oplus \mathbf{m}^{NN}(\mathbf{z}_k^u) \tag{70}$$

where $\mathbf{z}_k^f$ are measurements a filter understands how to incorporate, and $\mathbf{m}^{NN}()$ is an ANN trained to transform measurements the filter cannot incorporate $\mathbf{z}_k^u$, into measurements a filter can incorporate. The filter now has more measurements to update the state estimate. An example of this is in Haarnoja et al. [28](2016) where ANNs use differenced images from a camera to provide velocity estimates to a filter. The filter used by Haarnoja et al. could not incorporate differenced images but, the filter could incorporate velocity measurements. While using ANNs to determine velocity from images has a long history such as in Hatsopoulos et al. [31], it has only been recently that the ANNs have been trained specifically to transform measurements filters cannot incorporate into measurements filters can incorporate.

Another type of ANN aiding is in adaptive filtering. Adaptive filtering tunes parameters of a filter as measurements are taken [102]. Different optimizing schemes can be used to tunes parameters, one of which is with ANNs. Stubberud et al. [94] introduced a Neural Extended Kalman Filter for general purpose filtering. An application of ANN adaptive filtering is from Jwo et al. [49][48] who used adaptive filtering for GPS positioning. Krishnan et al. [57] also applied a deep Kalman filter to improving health care.

Another integration method popular in the state estimation community is the EKF optimizer introduced by Singhal et al. [89]. In this EKF the weights of the ANN are treated like states in the filter. The system equations are as follows [32]:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{q}_k^w \tag{71}$$

$$\mathbf{z} = \mathbf{h}_k^{NN}(\mathbf{w}_k, \mathbf{u}_k, \mathbf{h}_{k-1}^{NN}) + \mathbf{v}_k \tag{72}$$

where $\mathbf{w}$ is the weights of the ANN, $\mathbf{q}$ is white noise corrupting the weights with covariance $E[\mathbf{q}_k^w \mathbf{q}_k^{wT}] = \mathbf{Q}_k$, $\mathbf{z}$ is a measurement of the desired output, $\mathbf{h}^{NN}()$ is the ANN function that predicts the output, $\mathbf{h}^{NN}()$ is the state of the ANN, and $\mathbf{v}$ is white noise corrupting the measurement prediction with covariance $E[\mathbf{v}_k \mathbf{v}_k^T] = \mathbf{R}$. This system filters incoming measurements of the output $\mathbf{z}$ with an EKF. The covariance of the state is estimated in the matrix $\mathbf{P}_k^w$. The Jacobian $\mathbf{H}$ matrix is calculated by using back propagation on each output value in $\mathbf{h}_k^{NN}()$ with respect to each weight in $\mathbf{w}$. This allows the EKF to calculate the Kalman gain $\mathbf{K}$ and thus update the weights and covariances with Equation 10. This optimizer becomes more useful to filtering theory when the state vector includes weights as well as other states of interest such as position. When non-weight states are included, the algorithm estimates the propagation of those states as an ANN:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{q}_k \tag{73}$$

$$\mathbf{x}_{k+1} = \mathbf{f}_k^{NN}(\mathbf{x}, \mathbf{u}_k, \mathbf{w}_k) + \mathbf{q}_k^x \tag{74}$$

$$\mathbf{z} = \mathbf{h}_k^{NN}(\mathbf{x}_k, \mathbf{w}_k, \mathbf{u}_k, \mathbf{h}_{k-1}^{NN}) + \mathbf{v}_k \tag{75}$$

where $\mathbf{f}^{NN}()$ is the state predict equation for the states, and $\mathbf{u}$ is the system input. There is also a new noise term $\mathbf{q}_k^x$ where $E[\mathbf{q}_k^x \mathbf{q}_k^{xT}] = \mathbf{Q}_k^x$. The weights are propagated in the same manner as before, but a new state predict equation is trained to predict future values of the states. The addition of the new states also means a new covariance matrix is required for the states $\mathbf{P}_k^x$. While the covariance propagation equation for the weights was simply $\mathbf{P}_k^w = \mathbf{P}_{k-1}^w + \mathbf{Q}$, the propagation update for the states now has to linearize the propagation equation:

$$\mathbf{P}_k^x = \mathbf{F}\mathbf{P}_{k-1}^x\mathbf{F}^T + \mathbf{Q}_k^x \tag{76}$$

where $\mathbf{F} = \frac{\partial \mathbf{f}^{NN}}{\partial \mathbf{x}}$ is the Jacobian of $\mathbf{f}^{NN}()$. In this form there are essentially two different Kalman filters running. One filter optimizes the weights for $\mathbf{f}^{NN}()$ and $\mathbf{h}^{NN}()$ while the other filter estimates the states. If interaction between the weight and states is desired, the two filters can be combined:

$$[\mathbf{x}_{k+1} \oplus \mathbf{w}] = \mathbf{f}_k^{NN}(\mathbf{x}, \mathbf{u}_k, \mathbf{w}_k) + \mathbf{q}_k \tag{77}$$

$$\mathbf{z} = \mathbf{h}_k^{NN}(\mathbf{x}_k, \mathbf{w}_k, \mathbf{u}_k, \mathbf{h}_{k-1}^{NN}) + \mathbf{v}_k \tag{78}$$

where $\mathbf{f}_k^{NN}()$ now predicts the weights and states and $\mathbf{q}_k$ contains the noise terms for the weights and states. Also there is now only one covariance matrix $\mathbf{P}_k$ for both the weights and states. The combined covariance matrix can now expresses how the weights and states interact. The filter propagates and updates the weights and states using measurements **z**. The EKF optimizer has found many applications such as in Iiguni et al. [39], Puskorius et al. [79], Stubberd et al. [93, 94], Zhan et al. [110], Rajesh et al. [81], Kang et al. [51], and Lee et al. [59].

This dissertation brings the different research implementations of integrating ANNs and filters into one common framework. This makes it simpler to understand the ways that ANNs can work with filters. This also makes it easier to compare different implementations without delving into implementation details. There may be other nuanced ways that ANNs are used for filtering but this section outlined the major research thrusts.

VLF Spectrum

The framework and methodology described in Chapter III are general enough to apply to any time varying signal. The specific case of VLF electromagnetic spectrum was chosen for unique properties which can showcase the strength of the framework and methodology. This section will outline properties of the VLF spectrum which play key roles in position estimation. The first property is that at VLF frequencies the noise in the VLF band is much larger compared to higher frequencies. There are many noise sources in the VLF spectrum as noted in [90]:

1. Radiation from lightning discharges including local weather and weather around the world
2. Unintentional man made radiation from electrical equipment, electrical and electronic equipment, power transmission lines, or internal combustion engine ignition
3. Radiation from atmospheric gases and hydrometeors (water in the air as rain or water vapor)
4. Radiation from celestial radio sources

This increased noise means a transmitter must have a much higher signal power to overcome the noise in the VLF spectrum compared to transmitters at higher frequencies. This increased power in the received signal will in most cases be much larger then the Johnson noise in the receiver [41]. This allows receive antennas at VLF frequencies to have low gain, since the transmitted signal is already at high power. The power is actually high enough to be processed by a receiver without the internal receiver noise significantly degrading the signal to noise ratio. The International Telecommunication Union (ITU) has compiled noise figure measurements at low frequencies for communications equipment design [42]. The noise figure measures the loss of signal due to a the signal to noise ratio. Higher noise figure indicates more noise sources which will contaminate a transmitted signal. The noise figure measurements are broken up into two different bands: frequencies below 10 kHz (Figure 8) and above 10 kHz (Figure 9).

In both bands the noise figure increases as the frequencies get lower. The noise figure allows a calculation of the expected noise power at a given frequency:

$$f_a = \frac{p_n}{k t_0 b} \tag{79}$$

where f_a is the noise figure, p_n is the available noise power, k $=1.38 \times 10^{-23}$ J/K is the Boltzmann's constant, $t_0 = 290$ K is the reference temperature in Kelvin, and b is the noise bandwidth of the receiver. With this equation we can determine the power at a given noise bandwidth.

The high noise power at VLF frequencies also makes it resistant to noise jamming. Noise jamming attempts to raise the noise floor of a frequency band to drown out a transmitted signal. In order to transmit this jamming signal at a power high enough to compete with the noise, a high power antenna with large gain is required since the noise in the spectrum is so high already. For example, the National Institute of Standards and

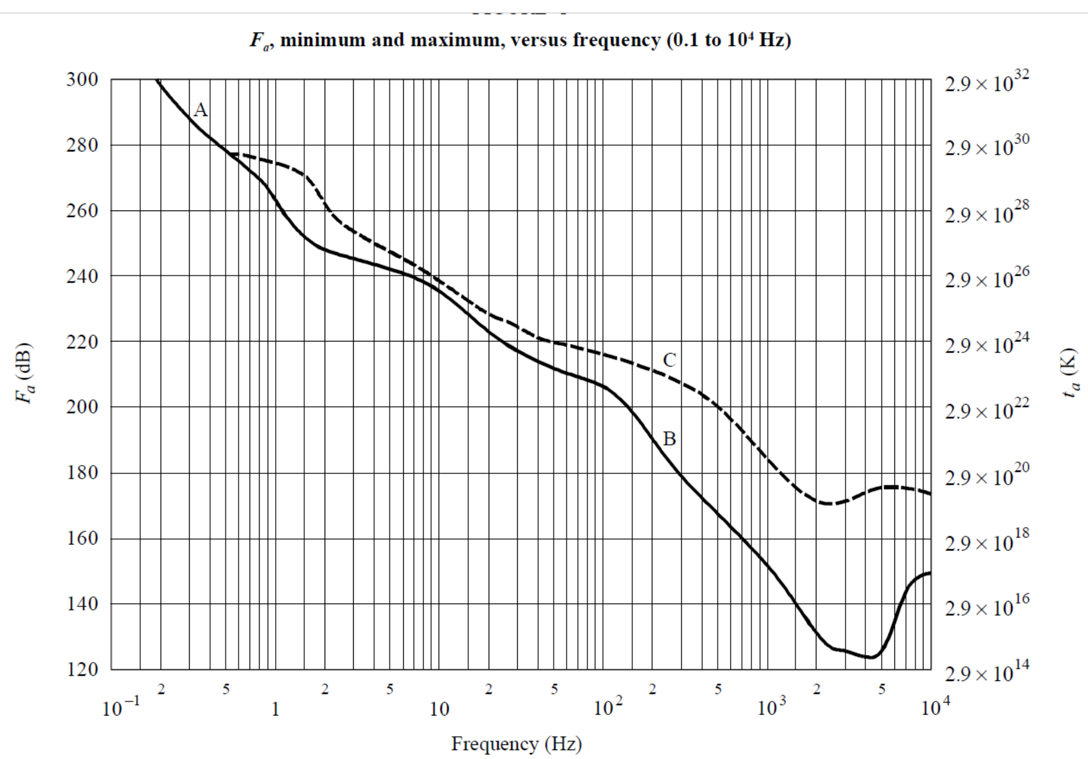

Figure 8: Noise Figure F_a measurements from the ITU at frequencies below 10 kHz. Note the downward trend as frequency increases. A is from micro pulsations, B is the minimum values expected of atmospheric noise and C is the maximum value expected of atmospheric noise. Figure found in [42]

Technology (NIST) WWVB antenna transmits a 60 kHz signal across the United States from Colorado. However the antenna consists of four 122 m towers spaced 857 m apart to suspend a T-antenna. The antenna also only transmits in a small frequency band around 60 kHz.

Next, a transmitter range equation will be evaluated based on this noise to show the short range of low power transmitters at VLF frequencies. First we give the equation for electromagnetic propagation [69]:

$$S = \frac{P_t G_t A_e}{4\pi R^2} \tag{80}$$

FIGURE 2

F_a versus frequency (10^4 to 10^8 Hz)

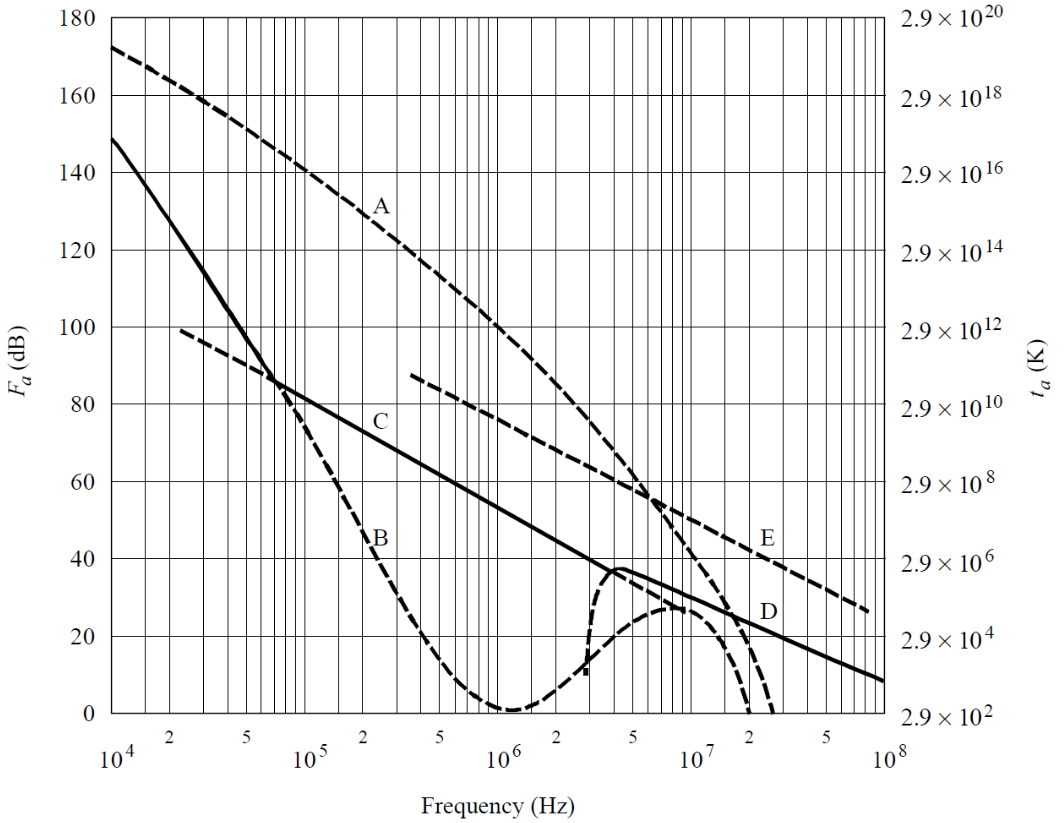

Figure 9: Noise Figure F_a measurements from the ITU at frequencies above 10 kHz. Note the downward trend as frequency increases. A is atmospheric noise values exceeded 0.5% of the time, B is atmospheric noise value exceeded 99.5% of time, C is man made noise at a quiet receiving site, D is galactic noise, E is the median for man made noise in a business area. The solid line is the minimum noise expected. Figure found in [42]

where S is the power at the receiver input, P_t is the power of the transmitter, G_t is the antenna gain of the transmitter, A_e is the effective area of our antenna with the wavelength taken into consideration, and R is the distance from the transmitter to the receiver. A_e is how much area of the radiated power our antenna will be able to absorb. This can be thought of how large electronically our antenna is. Larger antennas will in general be larger electronically but that is not always true. S is the power that will be compared to p_n in Equation 79. When a desired SNR s_n is set, the maximum distance R_m at which the signal will still be acceptable over the noise figure can be calculated by substituting the receiver power S with minimum signal power above noise $p_n s_n$:

$$R_m = \sqrt{\frac{P_t G_t A_e}{f_a k t_0 b s_n 4\pi}} \tag{81}$$

In this equation, R_m is the maximum range at which the SNR will still be at least s_n. Using the noise figure numbers from Figure 9 and 8 and setting power, gain, and signal to noise ratio a plot of resulting maximum range is shown in Figure 10.

From Figure 10 the range expected from transmitters with power of 10 W and 10 Hz bandwidth and unity transmission antenna gain and a receive antenna electronic size of 1 m^2 is on the order of tens of meters to hundreds of meters depending on frequency. The power, gain and antenna constants were chosen based on engineering judgment to be plausible and are not empirically derived from any real transmitter. This range is based on the stated power of the transmitters gains and noise figure measurements from the ITU. This dissertation assumes that local transmitters are close and powerful enough to be picked up by the antenna used in the experiments.

Literature Review Summary

In summary this literature review addressed four main topics of importance. The first topic provided a background on state estimation outlined the different approaches with respect to navigation. The second topic presented an overview of machine learning and ANNs with a short description of the background and the models that are useful for the problem in this research. The next section reviewed the state of the art for ANN integration with state estimation and filters. The last section provided a background on the VLF spectrum exploited for the experiments in Chapter V.

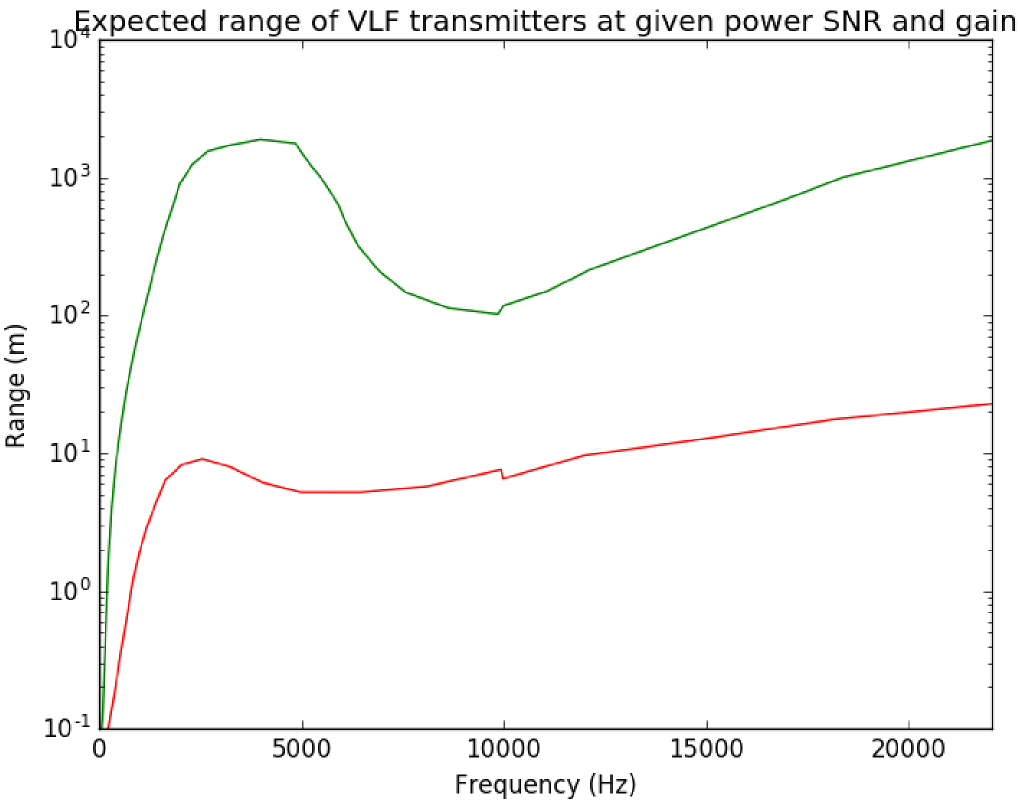

Figure 10: Range of transmitters with $P_t = 10$ W,$b = 10$ Hz, $G_t = 1$, $A_e = 1$ m^2. Green line is the range based on the lower bound of the noise figure and the red line is based on the higher bound of the noise figure.

III. ANN Filter Architecture

This chapter proposes how standard filters and ANNs can work together to improve performance and understanding. The first two sections outline syntax of symbols and notation. The next section outlines how to use ANNs to replace existing filters and filter parts. Next, methods to use ANNs to augment existing filters is outlined. Finally, a combined framework is given which incorporates the ideas from the previous two sections into one coherent filter framework.

Unknown System Input and Measurements

This section describes the nomenclature used in this dissertation to differentiate system inputs and measurements. This is important, because when ANNs are used to work with unknown data (such as VLF features), it is not obvious if the recorded data should be used as system inputs or as measurements. Unknown data is information which is difficult to relate to other values of interest, such as a state. The user has very little a priori knowledge about unknown data. However is is likely that the unknown data contains information about other values of interest. This section will outline the difference between system inputs and measurements and how this dissertation will classify unknown data.

In control theory, system inputs are strictly values which are controllable by the user and measurements are observed values. The purpose of the state predict function is to propagate the state estimates forward in time until observations are made to correct the states. Thus, the state predict equations are functions of states and system inputs $\hat{\mathbf{x}}_k = \mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k)$. This is because the user can control system inputs in the future and future states can be determine from the previous state. This allows predictions to be made indefinitely into the future. The observed values or measurements are used to adjust the state estimate to correct errors from the system dynamics.

In navigation, control is not a priority, and the state observer is much more important. From this perspective, system inputs are values which naturally propagate the state forward in time, and measurements are values which the state can naturally predict. It is not required that the system inputs are controllable by the user, because navigation only provides the state estimates and allows a separate entity to control the state. For example, in an aircraft navigation problem it may be easier to use a fuel rate as a system input instead of a measurement. This may be because fuel rate is easily incorporated into a state predict equation. However, the fuel rate is actually measured by some sensor and is not directly controlled by the navigation algorithm. This would be a problem from the control theory standpoint, because fuel rate cannot be used in the state predict equation since it is not controllable. The navigation algorithm, however, only wants to estimate the states and will use the fuel rate as a system input in the state predict equation, because it is easier to predict states with fuel rate than it is to correct the states with the fuel rate.

In traditional navigation problems this distinction is unnecessary, because it does not matter if observed data is called system inputs or measurements. The user will determine where the data is more easily incorporated. However, when an ANN is used with navigation filters, the distinction becomes more important. When the user has data with little a priori information, it may be useful to process that information with an ANN. However, lack of a priori information may make it difficult for the user to label the data as either a system input or a measurement. It is not apparent to the user if the data is more useful in predicting the state or relating directly to the states.

Determining if unknown data is a system input or measurement is a engineering problem dependent on how the ANN is incorporated into a filter. Testing may be required to see where the unknown data fits best. One rule of thumb is that the unknown system inputs and measurements should be mutually exclusive. If data is used in two different locations in the filter, then the filter may believe that it received two independent updates and be overconfident in its state estimate.

When integrating a filter and an ANN, this dissertation uses the following syntax to denote how data is to labeled. System inputs are denoted as $\mathbf{u}_k$ and measurements are denoted as $\mathbf{z}_k$ where k denotes the time step. The system inputs and measurements can be broken into two groups, system inputs and measurements which have enough a priori information to be incorporated by a filter, and system inputs and measurements which lack a priori information and cannot be used in a filter. The full set of system inputs and measurements will be denoted as:

$$\mathbf{u}_k = \mathbf{u}_k^f \oplus \mathbf{u}_k^u \tag{82}$$

$$\mathbf{z}_k = \mathbf{z}_k^f \oplus \mathbf{z}_k^u \tag{83}$$

where the superscript f denotes system inputs and measurements which can be incorporated by a filter to make estimates. The superscript u represent unknown system inputs and measurements that lack a priori information and therefore cannot be readily incorporated into a filter to make estimates as shown in Figure 11. When a filter cannot incorporate data, an ANNs can learn additional information that can represent the data in a form a filter can understand as shown in Figure 12. The unknown data can now be incorporated into the filter via the ANN in order to improve performance.

States

This dissertation will in general define a state as any possible value that can be predicted into the future based on system inputs, measurements and/or previous state measurements. In order to to exclude states which are irrelevant to the problem, we place a restriction so that the state value must relate to an output of interest or another state which relates to an output. The output function is defined as $\mathbf{y} = \mathbf{o}(\mathbf{x})$. This distinction is important, because when an ANN models some functions of the filter, the ANN may

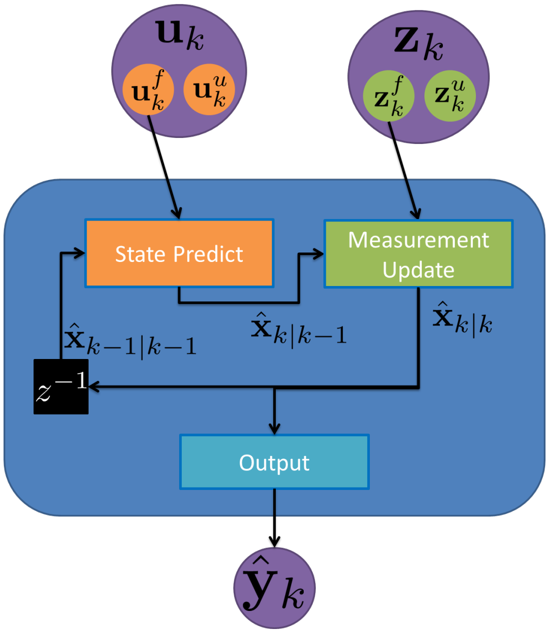

Figure 11: Filter with data which cannot be incorporated into the state predict or measurement update functions.

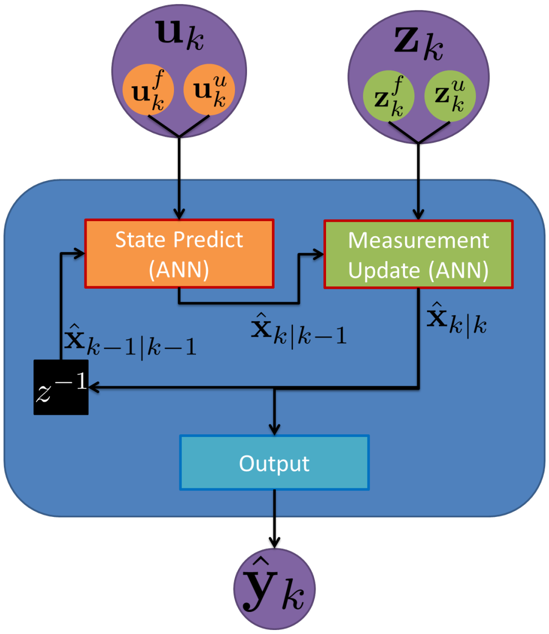

Figure 12: Filter with data which could not be incorporated until the state predict and measurement update functions were changed to ANN representations (red box outline).

develop its own states. However, these internal ANN states may not have clear meaning to the user. This definition excludes ANN states as filter states since the relation to other desirable states is unknown.

The estimate of the states usually requires a probability distribution, since the future value of the states is probabilistically uncertain. When the true state is $\mathbf{x}$, the estimate of that state will be written as $\hat{\mathbf{x}}$, where the hat denotes an estimate of the true value. The estimate may be with respect to different criteria. If a single value is desired, the estimate may be the mean of the value. For example, a Kalman filter assumes the states have Gaussian distributions which are fully defined by their mean and covariance. Thus the Kalman filter calculates both a mean and a covariance matrix for state estimates.

Instead of thinking of the Kalman filter providing the mean and covariance for each time step, we can think of the Kalman filter as determining the parameters of a multivariate Gaussian distribution with N dimensions. In this case we could adopt the syntax that $\hat{\mathbf{x}} = \boldsymbol{\mu} \oplus \boldsymbol{\Sigma}$ where $\boldsymbol{\mu}$ is the mean of the state estimates commonly written in Kalman filter notation as $\hat{\mathbf{x}}$, and $\boldsymbol{\Sigma}$ is the covariance between the states commonly written in Kalman filter notation as $\mathbf{P}$. Our concatenated vector $\hat{\mathbf{x}}$ contains both the means and covariances as one vector, where the first N values would be the means of the multivariate Gaussian. The next N^2 values of the vector would represent indices of the covariance matrix for the multivariate Gaussian. The estimate of the states is no longer a single value but a parameterized probability distribution. This representation does require explicitly declaring what type of distribution $\hat{\mathbf{x}}$ parameterizes, but allows for a more general representation of the distribution of the states $\mathbf{x}$.

This dissertation will in general consider $\hat{\mathbf{x}}$ as some list of parameters that describes the distribution of the states $\mathbf{x}$. This means that some function written as $\hat{\mathbf{x}}_k = \mathbf{f}(\hat{\mathbf{x}}_{k-1})$ is taking the parameters that represent the distribution of $\mathbf{x}$ at time step $k - 1$ and returning the parameters that represent the distribution of $\mathbf{x}$ at time step k. This function alone does not specify what distribution the parameters represent. A distribution (such as a Gaussian distribution or a Gaussian Mixture Model) must be explicitly declared someplace else. The type of distribution then determines the meaning of the parameters in $\hat{\mathbf{x}}$.

ANN Filters

This section shows different approaches for using ANNs as a filter or filter parts in a navigation solution. First, a method to make one large ANN to perform all filter operations is defined. Next the large ANN is decomposed into different parts to perform different functions of standard filters. The Kushner equation is used to provide guidance on the breakdown of the large ANN filter. Finally an overview of the ANN as filter parts is given.

ANN as Filter.

The most straightforward way to make an ANN into a filter is to give the ANN all the system inputs and measurements a standard filter would receive along with the initial state estimate $\hat{\mathbf{x}}_0$ [71, 99, 98, 105, 18]. The ANN then predicts the state at the next time step given this initial information. At future time steps, the ANN would use its own prediction of the state at the last time step instead of the initial state estimate. A block diagram of this model is shown in Figure 13. In equation form this filter is written as:

$$\hat{\mathbf{x}}_k = \mathbf{s}^{NN}(\mathbf{u}_k, \mathbf{z}_k, \hat{\mathbf{x}}_{k-1}, \mathbf{h}^{NN}_{k-1}|\boldsymbol{\theta}^X) \tag{84}$$

$$\hat{\mathbf{y}}_k = \mathbf{o}(\hat{\mathbf{x}}_k|\boldsymbol{\theta}^O) \tag{85}$$

where $\hat{\mathbf{x}}_k$ is the estimate of the state $\mathbf{x}$ at time step k, $\mathbf{u}_k, \mathbf{z}_k$ are the system inputs and measurements at time step k, $\mathbf{h}^{NN}_{k-1}$ is the internal state of the ANN at the previous time step (since this is a RNN), $\boldsymbol{\theta}^X$ is the parameter weights of our ANN, and $\hat{\mathbf{y}}_k$ is the final output which is a function of the state only through some function $\mathbf{o}()$. If we assume the output is the same as the states then the equation reduces to:

$$\hat{\mathbf{y}}_k = \text{FILTER}^{NN}(\mathbf{u}_k, \mathbf{z}_k, \hat{\mathbf{y}}_{k-1}, \mathbf{h}^{NN}_{k-1}|\boldsymbol{\theta}^F) \tag{86}$$

where FILTER^{NN}() is one large ANN that determines the output directly based on the inputs, measurements, and previous output.

This type of ANN may be trained using some form teacher forcing since it uses its previous estimate as inputs. While determining the architecture of the ANN is simple, training the ANN is much more complicated. In order to achieve low error in the final output estimates, the ANN may require high capacity which will then require many training examples and regularization. This is compounded by the fact that when the ANN is trained with teacher forcing another level of complication is added to prevent the ANN from ignoring system inputs and measurements and using only previous estimates. However

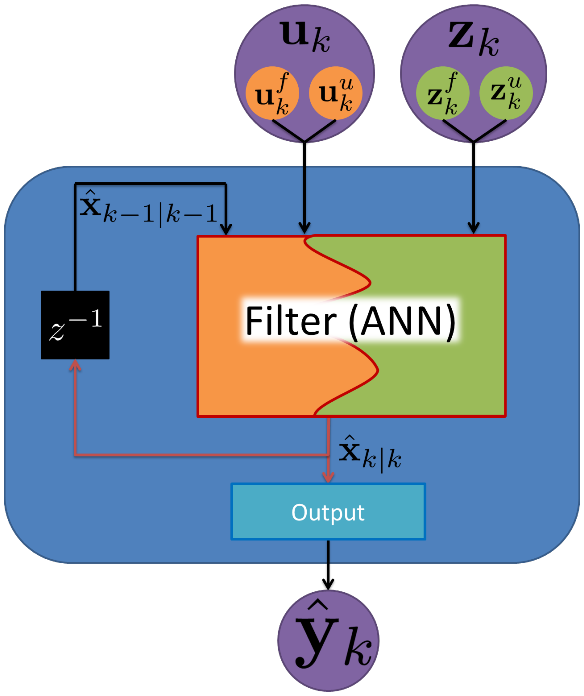

Figure 13: ANN used as the entire filter. The ANN takes all the system inputs and make a state estimate based on the previous estimate and system inputs and measurements.

this filter architecture does allow the full power of the ANN to do whatever it takes to reduce the error if trained properly.

One drawback of this method is that now the entire filter is one large black box. Inference on how the ANN combined system inputs, measurements, and previous states is completely lost. Additionally, this model does not take advantage of any a priori information the user had about the known system inputs, measurements, and states. While the filter can be trained to output covariances and probabilities of the states, we will not know how well the ANN generalized. Lack of generalization could degrade the performance of our final estimates when using the filter in a new environment. Thus, the filter may have high confidence in its state estimate, but in reality the state may be incorrect and the ANN may not have generalized well to new combinations of system inputs, measurements, and states. This motivates the filter to be decomposed into different parts so the parts may be monitored individually in order to improve the level of trust in the ANN filter. Smaller ANN components may be easier to train compared to a single large ANN filter.

ANN as Filter Parts.

To make the ANN filter less of a black box, the ANN filter can be decomposed into standard filter parts represented by multiple ANNs [77, 76, 80, 91]. This allows the user to exploit research on replacing filter parts with ANN (such as the state predict function [11, 70, 19, 63, 20, 79, 85, 96, 68, 72, 64, 81, 59, 4]) This breakdown also makes it easier to understand what the filter does with the system inputs, measurements, and state to determine output. There are many different possible ways to break up the filter into ANN parts. The main research contribution into how to perform the breakdown was provided by Parlos et al. [77]. Parlos et al. used an ad hoc method to determine what roles the ANNs would play and what inputs they take. In their paper, Parlos et al. specifically mention how they could have chosen different inputs to the ANNs functions. A specific instance of this was the state filter which they wrote could include the actual measurement or the estimated measurement, because the residual allowed the ANN to determine the other value. This dissertation looks to the Kushner equation from Section II in order to build a theoretical backing for the roles of the ANNs and their inputs. First we shall explain the Kushner equation then breakout parts of the overall filter

Kushner Equation Revisited.

For reference the Kushner equation solves the filtering problem for the system equations:

$$d\mathbf{x} = \mathbf{f}(\mathbf{x}, t)dt + \mathbf{G}(\mathbf{x}, t)d\mathbf{w} \qquad \text{(23 revisited)}$$

$$d\mathbf{z} = \mathbf{h}(\mathbf{x}, t)dt + d\mathbf{v}$$

where measurements **z** are filtered to determine corrections for states **x**. By grouping terms of the Kushner equation as approximate function representations, the Kushner equation 25 can be simplified as the change in probability $dp(\mathbf{x}, t|\theta_{\text{FILTER}}, \mathbf{Z})$:

$$dp = \mathbf{f}^{NN} + \left(\mathbf{k}^{NN}\right)^T \left(d\mathbf{z} - \mathbf{h}^{NN}\, dt\right) \tag{87}$$

$$\theta_{\text{FILTER}} = \{\mathbf{f}^{NN}, \mathbf{k}^{NN}, \mathbf{h}^{NN}\}$$

$$\begin{aligned}
&\mathbf{f}^{NN}(p) \approx \mathcal{L}(p)dt \\
&\mathbf{h}^{NN}(p) \approx \hat{\mathbf{h}} = \int \mathbf{h}(\mathbf{x}, t)p(\mathbf{x}, t)dx \\
&\mathbf{k}^{NN}(p) \approx (\mathbf{h} - \hat{\mathbf{h}})^T \mathbf{R}^{-1}(t)p
\end{aligned}$$

where the Kolmogorov forward equation $\mathcal{L}(p)dt$ has been replaced with an approximate state predict function $\mathbf{f}^{NN}()$ to estimate the future state. The terms added to the Kolmogorov forward equation were simplified as well. For example, the measurement estimate has been replaced with an approximate function $\mathbf{h}^{NN}()$ which makes the new residual $(d\mathbf{z}{-}\mathbf{h}^{NN}(\mathbf{x}, t)dt)$. This residual is now multiplied by a gain $\mathbf{k}^{NN}()$ which is an approximation of the previous gain value. This simplistic view is an approximation for the Kushner equation. This view approximates the system equations containing the parameters θ_k from Equation 26, into the parameters θ_{FILTER} reducing the number of functions. The functions $\mathbf{f}^{NN}(), \mathbf{h}^{NN}(), \mathbf{k}^{NN}()$ are trained to determine the probability of the states. By training these functions to estimate the probability, they will approximate the Kushner equation true vales. By approximating the Kushner equation, the ANN functions become a general nonlinear differential filter. These ANN functions are useful for approximating the Kushner equation when the system equations are unknown. For example, when some combination of $\mathbf{f}(), \mathbf{G}(), \mathbf{h}(), \mathbf{Q}(), \mathbf{R}()$ from the original Kushner equation are unknown, but the form of the system almost surely follows the form for the system in Equation 23. In practice, the ANN functions will likely not become the optimal solution expressed in the Kushner equation, but it is theoretically possible since ANNs can in general become any function (including the Kushner equation functions) given enough capacity [38], [36].

We now make a further simplification by changing the output of the Kushner equation from a change in probability to a full probability as discrete time equation. First we will discretize time by changing continuous values like dt into discrete changes $\Delta t = t_1 - t_0$. We use the same notation for the simplified functions as before but realize that they will be different from their continuous time versions:

$$\Delta p_{t_1|t_1}(\mathbf{x}, t_1, t_0) = \Delta p_{t_1|t_0} + \mathbf{k}^T(\Delta \mathbf{z} - \hat{\mathbf{z}}) \tag{88}$$

$$\begin{aligned}
\Delta p_{t_1|t_0} &= \mathbf{f}^{NN}(p_{k_0|k_0})\Delta t \\
\mathbf{k} &= \mathbf{k}^{NN}(p_{t_1|t_0}) \\
\hat{\mathbf{z}} &= \mathbf{h}^{NN}(p_{k_1|k_0})\Delta t \\
\Delta t &= t_1 - t_0 \\
\Delta \mathbf{z} &= \mathbf{z}_{t_1} - \mathbf{z}_{t_0}
\end{aligned}$$

where $p_{t_1|t_0}$ means the probability at discrete time t_1 given all the measurements up to time t_0. This probability is a function of both times as well as the state vector. This discrete representation will allow us to represent this approximated version of the Kushner equation in discrete time.

We will next assume that the time between our time steps are now fixed to some Δt where we now have discrete equally spaced time indices $t_k = k\Delta t, k = 0, 1, \ldots, N$. We now can refer to time indices k or $k - 1$ instead of specific times t_0, t_n. Next we will express this equation as the full probability instead of a change in probability. To accomplish this we add $p_{k-1|k-1}$ to both sides and express our residual as not a difference in measurements but as the measurement at time step k:

$$p_{k|k}(\mathbf{x}) = p_{k|k-1} + \mathbf{k}^T(\mathbf{z}_k - \hat{\mathbf{z}}_k) \tag{89}$$

$$p_{k|k-1} = \mathbf{f}^{NN}(p_{k-1|k-1})$$

$$\mathbf{k} = \mathbf{k}^{NN}(p_{k|k-1})$$

$$\hat{\mathbf{z}}_k = \mathbf{h}^{NN}(p_{k|k-1})$$

Once again the functions are slightly different now that we are estimating the probability at time step k and not the change in probability. The equation now looks similar to the Kalman filtering equations outlined in Section II.

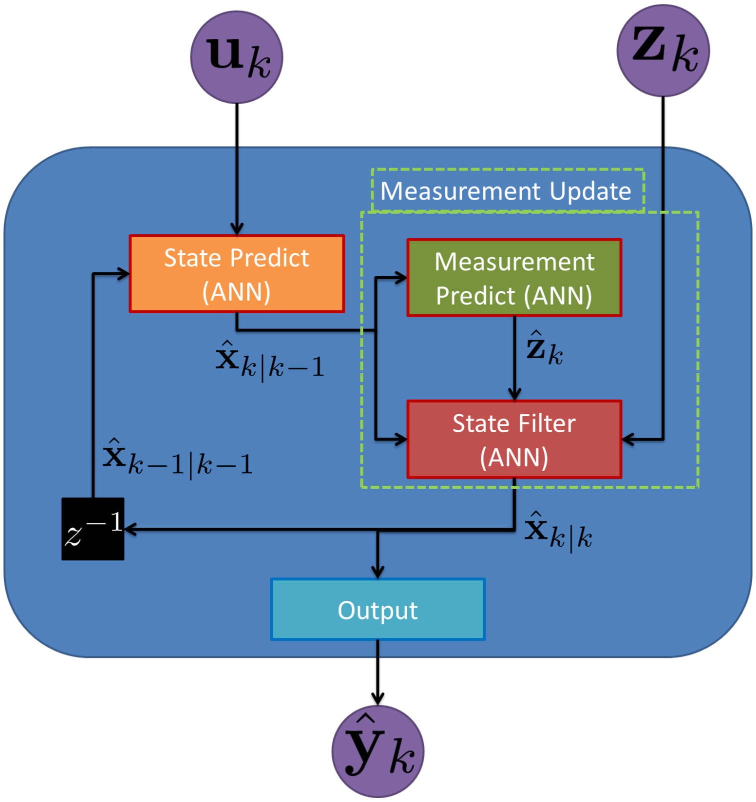

Figure 14: Standard Filter with the Measurement update function changed to explicitly estimate the measurements based on state.

ANN Breakdown.

If we revisit our standard filter diagram from Section II we can now break the diagram up into parts which will represent our approximate functions. A block representation of the first decomposition is shown in Figure 14. Our decomposition breaks the measurement update function **mu**() into two different sub functions **h**() and **sf**(). The equations for the diagram can be written with the measurement update function as:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{f}^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k|\boldsymbol{\theta}^F) \tag{90}$$

$$\hat{\mathbf{x}}_k = \mathbf{mu}^{NN}(\hat{\mathbf{x}}_{k|k-1}, \mathbf{z}_k|\boldsymbol{\theta}^{MU}) \tag{91}$$

Next we write a new equation to obtain $\hat{\mathbf{x}}_k$ where we break up the measurement update equation into two parts to explicitly use an estimate of the measurements:

$$\hat{\mathbf{z}}_k = \mathbf{h}^{NN}(\hat{\mathbf{x}}_{k|k-1}|\boldsymbol{\theta}^H) \tag{92}$$

$$\hat{\mathbf{x}}_k = \mathbf{sf}^{NN}(\hat{\mathbf{x}}_{k|k-1}, \mathbf{z}_k, \hat{\mathbf{z}}_k|\boldsymbol{\theta}^{SF}) \tag{93}$$

where the measurement update function has been replaced with the state filter function $\mathbf{sf}^{NN}()$ which takes as and additional input the predicted measurement from the measurement predict function $\mathbf{h}^{NN}()$. The function $\mathbf{h}^{NN}()$ explicitly predicts measurements based only on the current state. The function $\mathbf{sf}^{NN}()$ is a state filter which takes as input the measurement estimate from the first function, the actual measurement, and current state estimate. Thus, this state filter function uses measurements to correct the predicted state dynamics. Note that the Kushner equation did not include system inputs **u**. However, this paper will include the system inputs in the system dynamics function **f**(). Also, the estimate of the probabilistic distribution p_k of states **x** is represented by the parameters $\hat{\mathbf{x}}$ of some probability density function as stated in Section III. With this we have decomposed the measurement update equation to a form similar to the form in Parlos et al. [77]. However we will continue to decompose the state filter function further using insights from our Kushner simplification in the previous section.

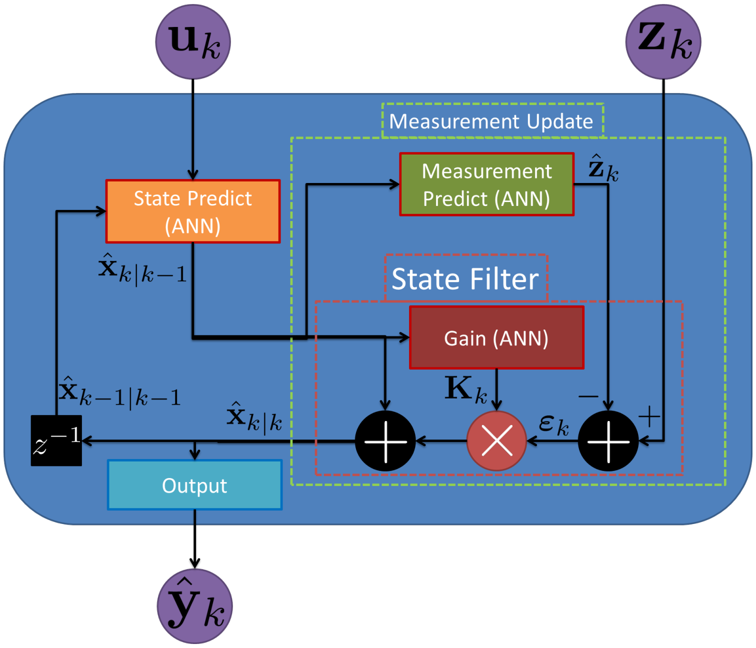

Figure 15: Standard Filter with the Measurement update block changed to explicitly estimate the measurements based on state. The black plus signs are element wise addition of two vectors. The red multiply symbol is a matrix vector multiplication and not an element wise operation.

From the Kushner equation simplification Equation 89, we note that the state filter function is not a general function of the state, measurements, and measurement estimate but a correction to the state estimate based on the residual. A block representation of the Kushner equation break down is shown in Figure 15. This block diagram approximates the Kushner equation which describes how to filter states from measurements, where the system dynamics and update equations are non-linear and is corrupted with white Gaussian noise. In equation form, the state filter function is now defined as a function of the residual between measurements and estimated measurements, correcting the current propagated state estimate:

$$\mathbf{sf}^{NN}(\hat{\mathbf{x}}_{k|k-1}, \mathbf{z}_k, \hat{\mathbf{z}}_k|\boldsymbol{\theta}^{SF}) = \hat{\mathbf{x}}_{k|k-1} + \mathbf{k}^{NN}(\hat{\mathbf{x}}_{k|k-1}, \hat{\mathbf{z}}_k|\boldsymbol{\theta}^{K})\boldsymbol{\epsilon}_k \tag{94}$$

$$\boldsymbol{\epsilon}_k = (\mathbf{z}_k - \hat{\mathbf{z}}_k)$$

where the function $\mathbf{k}^{NN}()$ estimates a $N \times M$ matrix gain to apply to the residual vector $\boldsymbol{\epsilon}_k$ in order to correct the propagated state estimate $\hat{\mathbf{x}}_{k|k-1}$. Note that $\mathbf{k}^{NN}()$ is the same form as the function in Equation 89.

With this decomposition the state predict, measurement predict, and gain functions can each be represented by an ANN. In general we will assume that the output function is known. We also assume that a true or high fidelity estimate of the state and measurements are available. Now we describe how to train the ANN components.

We train the state predict function or $\mathbf{f}^{NN}()$ to predict the future state based on system inputs and previous states. Next, the measurement predict function or $\mathbf{h}^{NN}()$ function is trained to predict measurements with the true state. In order to train the $\mathbf{k}^{NN}()$ function, the estimates from the state predict $\mathbf{f}^{NN}()$ and measurement predict $\mathbf{h}^{NN}()$ are required. If teacher forcing is used to train the ANN, then some method to prevent the ANN from learning solutions that simply pass on the true state.

Also, without using the previously created $\mathbf{f}^{NN}(), \mathbf{h}^{NN}()$ functions, the gain function will not be able to learn the types of error committed and how to apply gain in order to correct the estimate. The target of training the gain function $\mathbf{k}^{NN}()$ is the true state after the matrix multiply and element wise addition are completed. Thus, the training will try to estimate a gain which makes a correction to the current state estimate in order to minimize a loss function between the estimate and the truth.

Finally, when all three functions are trained, an overall fine tuning step can be done starting with teacher forcing and moving to global feedback [77, 28]. This fine tuning step trains all three functions at the same time in order for them to work together to calculate the best final state estimate. If only overall performance is required, then the loss function only involves the final state estimate $\hat{\mathbf{x}}_{k|k}$ and the truth. If it is desired to still have the state predict become more accurate, then the loss function can involve the state predict output $\hat{\mathbf{x}}_{k|k-1}$ as well as the state estimate after the measurement update $\hat{\mathbf{x}}_{k|k}$. Also, the measurement predict can be further trained to still have accurate measurement estimates, and $\hat{\mathbf{z}}_k$ can be included in the loss function as well. Either way, fine tuning is done to train each function to work together to minimize error on the loss function.

Overall this section outlined a method to use only empirical data to train ANN(s) to become a filter. Filtering theory provided a guideline to partition a large ANN filter into components. This section requires truth data to train the ANN(s) but does not require any a priori knowledge of the system dynamics or measurements. The meaning of the system dynamics and measurements is learned from empirical data by training the ANN(s). The next section explores how to incorporate a priori information into a filter.

ANN as Filter Parts Overview.

This section defines an algorithm for the decisions to make in designing a filter with ANN parts. The previous section went through a decomposition using the Kushner equation and assumed all the functions would be replaced with ANNs. However, any of the different breakdowns discussed while determining the final breakdown are valid configurations. Also, any function at any level of the breakdown can either be an a priori function or an ANN function. A flowchart of the algorithm is shown in Figure 16. The flowchart outlines the decision to be made in determining the filter. First the flowchart includes research of the current problem and obtains the current stat of the art knowledge. The decisions in the flowchart are based on this knowledge to determine where ANNs will be useful and where a priori methods are still acceptable.

The flowchart can produce any of the three break downs provided by Figures 1, 14 and 15. For example, by taking decisions left–left–right, we can produce the filter configuration in Figure 1. This configuration has a state predict function and a measurement update

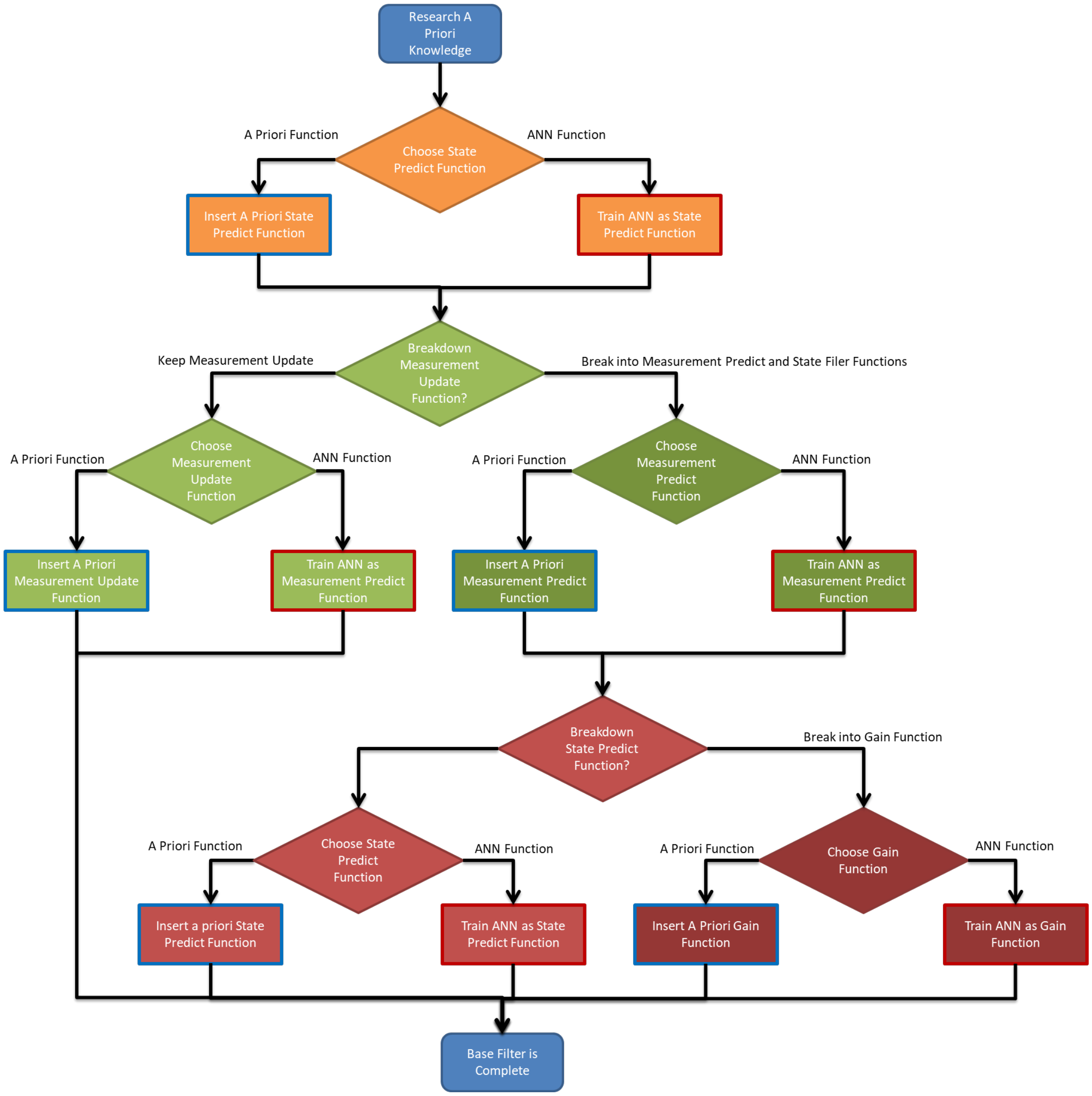
Research A Priori Knowledge
A Priori Function
Choose State Predict Function
ANN Function
Insert A Priori State Predict Function
Train ANN as State Predict Function
Keep Measurement Update
Breakdown Measurement Update Function?
Break into Measurement Predict and State Filer Functions
A Priori Function
Choose Measurement Update Function
ANN Function
A Priori Function
Choose Measurement Predict Function
ANN Function
Insert A Priori Measurement Update Function
Train ANN as Measurement Predict Function
Insert A Priori Measurement Predict Function
Train ANN as Measurement Predict Function
Breakdown State Predict Function?
Break into Gain Function
A Priori Function
Choose State Predict Function
ANN Function
A Priori Function
Choose Gain Function
ANN Function
Insert a priori State Predict Function
Train ANN as State Predict Function
Insert A Priori Gain Function
Train ANN as Gain Function
Base Filter is Complete

Figure 16: Flowchart showing the different steps to configure ANNs as filter parts. Orange blocks relate to the State Predict function, green blocks relate to the Measurement Update and Predict function, and red relates to the State Filter and Gain functions. A red outlined block is a ANN, while a blue outlined rectangle is some a priori function.

function. However, the measurement update function has been replaced by an ANN while leaving the a priori state predict function intact. Alternatively, we could take options left–right–right–left to create the filter configuration of Figure 14. This configuration separated the measurement update function into a measurement predict function and state filter function. In this case, we choose options to replace the measurement predict function by an ANN while leaving the a priori state predict and state filter functions. In another example, we could choose options right–right–right–right, which creates the most specific model in Figure 15. These choices further broke apart the state filter into a gain function based on the Kushner Equation from the previous section. This configuration replaces all three functions–state predict, measurement predict, and gain functions with ANNs as discussed in the previous section. In summary, the flowchart gives steps to determine what shape of the filter is possible and what functions can be replaced by ANNs.

When deciding which option to take in the flowchart there are a few guidelines to aid in decision making. These guidelines are consolidated with guidelines from other integration types in Section III. Next integration will be viewed not as replacing functions with ANN but augments existing functions with ANN.

ANN Filter Integration

This section outlines how to augment current filters with ANNs trained with empirical data. This section describes three different ways standard filters can be integrated with an ANN to produce state estimates. The first method allows ANNs to correct estimates provided by the filter. The second method allows ANNs to provide measurements for the filter. The third method allows ANNs to change filter parameters. This integration idea is motivated by the fact that if information about the system is already known, ignoring it could hinder performance of an ANN-only filter described in Section III. The other motivation is to provide a method that can use learned non-linear functions provided by ANN while keeping a proven filter. The filters used in this section are standard filters that already contain their own state predict and measurement update equations. This section only augments the standard filters with ANNs and does not change any existing state predict or measurement update functions. If trust is lost in the ANNs for any reason the ANNs can be removed and the original filter remains intact.

ANN Corrections for Filter.

A general method for combining a filter and an ANN is to correct the filter with an ANN. In this sense, correcting means adding a vector, element wise, to a current estimate in order to reduce the error. An ANN can correct three different estimates–the filter output, the filter state estimate, and the filter state predict estimate. The simplest case is to just correct the filter final output. Extra training considerations must be made when correcting the state estimate and state predict estimate.

ANN Correcting Filter Output.

The first correction method uses the filter to first generate an output estimate. An ANN is then trained to correct the error in the output of the filter [12, 100, 6, 47, 108, 61, 53]. The ANN is trained with the true error as determined by some high fidelity truth source. The ANN correction is then added to the filter output to make a more accurate corrected output. This corrected state estimate is then presented to the user as the final output. This correction method has the benefit of producing a standard filter estimate at each time step in case the ANN is believed to be untrustworthy for any reason. This can alleviate concerns about using a "black box" function for filtering since, the regular filter output is still a proven algorithm. The overall diagram to correct filter output is shown in Figure 17.

The filter output in equation form is written as:

$$\hat{\mathbf{y}}'_k = \text{FILTER}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}^f_k, \mathbf{z}^f_k | \boldsymbol{\theta}) \tag{95}$$

where FILTER is the filter function which gives and estimate for the output $\hat{\mathbf{y}}'_k$ at time step k. The prime is added to differentiate the filter output estimate from the final output estimate

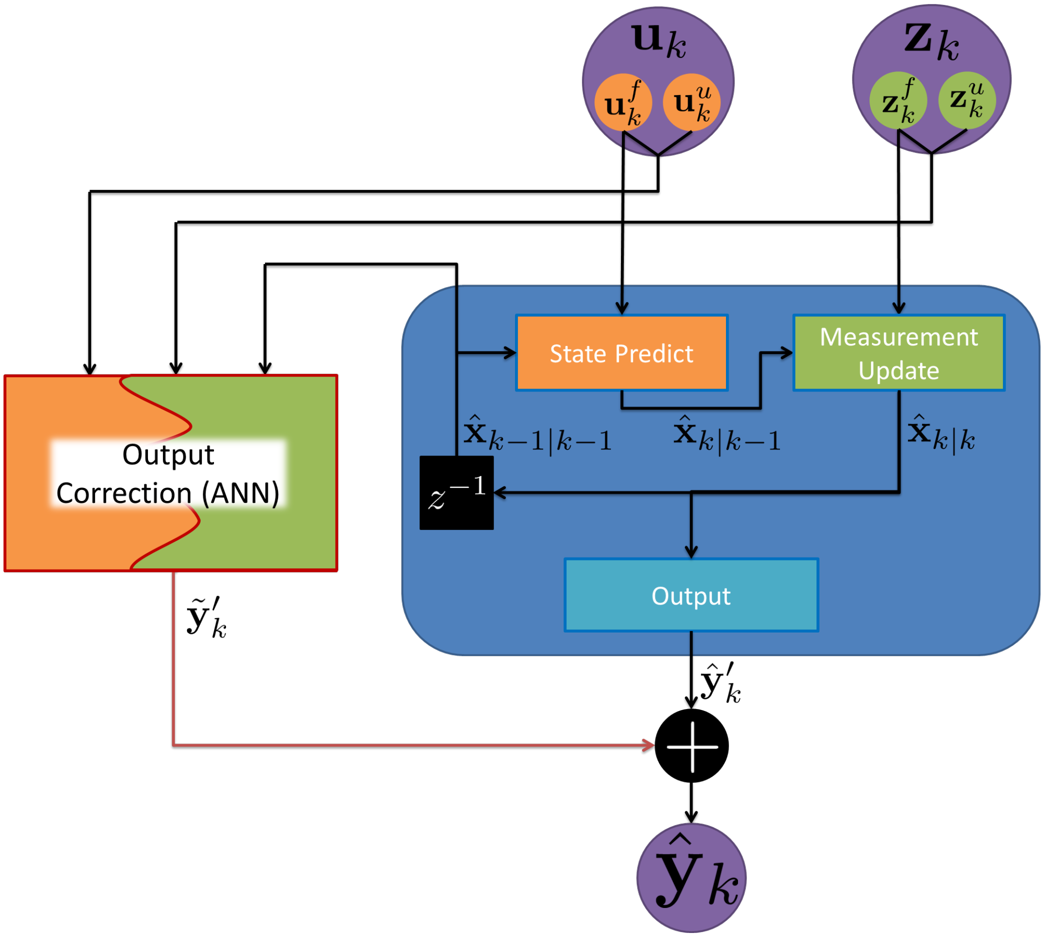

Figure 17: Block diagram of a generic standard filter with state predict correction provided by an ANN. The ANN uses the previous state estimate, the system inputs and measurements to correct the filter output. The red line denotes the ANN provided correction which if removed falls back to the original filter.

$\hat{\mathbf{y}}_k$. The output error committed by the filter is:

$$\tilde{\mathbf{y}}_k = \mathbf{y}_k - \hat{\mathbf{y}}'_k \tag{96}$$

$$\tilde{\mathbf{y}}'_k = \hat{\mathbf{y}}_k - \hat{\mathbf{y}}'_k \tag{97}$$

where $\tilde{\mathbf{y}}_k$ is the true error committed by the FILTER() function and $\tilde{\mathbf{y}}'_k$ is the estimate of that error. This true error is derived from the high fidelity truth source. The ANN is trained to estimate the true error of the filter $\tilde{\mathbf{y}}_k$ calculated from:

$$\tilde{\mathbf{y}}'_k = \mathrm{FILTER}_c^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k, \mathbf{h}_{k-1}^{NN}|\boldsymbol{\theta}^C) \tag{98}$$

where the correction term provided by the ANN $\tilde{\mathbf{y}}'_k$, attempts to match the true error $\tilde{\mathbf{y}}_k$. However the ANN estimate is not perfect and creates our final estimate $\hat{\mathbf{y}}'_k$. Note that the ANN function FILTER_c^{NN} includes as input not only the system inputs $\mathbf{u}_k$ and measurements $\mathbf{z}_k$ but also the state of the filter $\hat{\mathbf{x}}_{k-1}$. This allows the ANN to predict what the filter output will be. The ANN uses this prediction of the filter output to calculate a correction.

The ANN uses the full system inputs and measurements $\mathbf{u}_k, \mathbf{z}_k$. However, the filter is restricted to only the system inputs and measurements it knows how to incorporate $\mathbf{u}_k^f, \mathbf{z}_k^f$. Thus, the ANN correction is determined using more information then the original filter. The ANN attempts to use this extra information to close the gap between the filter output estimate and the true output as shown:

$$\hat{\mathbf{y}}_k = \hat{\mathbf{y}}'_k + \tilde{\mathbf{y}}'_k \tag{99}$$

where $\hat{\mathbf{y}}_k$ is the final output of the integrated ANN/filter combination which is the sum of the filter output estimate $\hat{\mathbf{y}}'_k$ and the correction from the ANN $\tilde{\mathbf{y}}'_k$. The final corrected filter equation is then:

$$\hat{\mathbf{y}}_k = \mathrm{FILTER}(\mathbf{x}_{k-1}, \mathbf{u}_k^f, \mathbf{z}_k^f|\boldsymbol{\theta}) + \mathrm{FILTER}_c^{NN}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k, \mathbf{h}_{k-1}^{NN}|\boldsymbol{\theta}^C) \tag{100}$$

where FILTER parameterized by $\boldsymbol{\theta}$ is the original filter equation which is corrected by the ANN function FILTER_c^{NN} parameterized by $\boldsymbol{\theta}^C$.

This type of correction works well when the filter already has a stable output which does not drift. The purpose of the ANN in this stable filter is only used to make slight corrections. The corrections are learned by the ANN by observing what error occurs with what combinations of system inputs, measurements, and state.

A problem can arise if the filter estimate is poor and the output is unstable. An example of this would be a state in the filter that is corrupted by process noise and not corrected by measurements. If this state is directly related to an output then the output will drift as well. The uncorrected filter estimate of this output will quickly become inaccurate and drift far from the truth. However, the ANN correction may be able to correct the output enough to limit the error and stabilize the output. In this case, the ANN correction can stabilize the output presented to the user but the internal filter state will still drift. Since the filter output is grossly inaccurate, any a priori known relation between the drifted output and other outputs will be useless. We are now wasting a priori information we programmed into our filter. In this case the drifted filter output is only a nuisance to the ANN output correction which essentially has to relearn the filter known relationship between the drifted output and the other outputs. Also, if the ANN correction was removed the output would return to the drifted filter output. Depending on how long the filter output has drifted, the output estimate could be grossly inaccurate. To prevent losing a priori information when states drift, we must stabilize the actual filter state. To do this, we ANN must correct the internal filter state instead of the output as outlined in the next section.

ANN Correcting Filter State Estimate.

The second way to correct the filter is to provide corrections to the state estimates. This requires correcting the internal filter state and not just the output presented to the user. The internal filter state is the value used by the filter at the next time step unlike the previous section where the correction was only make to the output presented to the user. The equations from Section III are modified to correct the internal filter state instead of the state output. An overview of the final diagram is shown in Figure 18.

To show this in equation form, first we need to break up the filter from its output function to get to the state estimates:

$$\hat{\mathbf{x}}'_{k|k} = \mathbf{s}\left(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}^f_k, \mathbf{z}^f_k | \boldsymbol{\theta}^X\right) \tag{101}$$

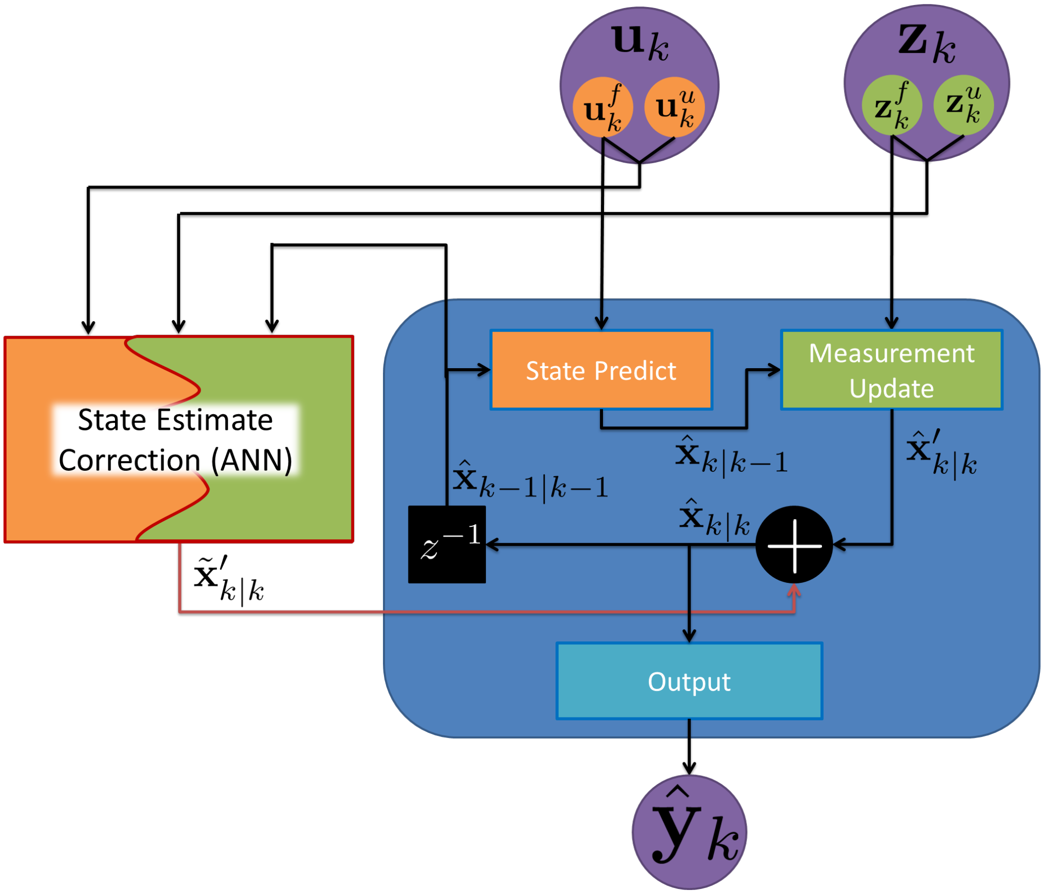

Figure 18: Block diagram of a generic standard filter with state estimate correction provided by an ANN. The state estimate correction ANN takes all the system inputs and measurements in order to correct the updated state estimate from the filter. The red line denotes the ANN provided correction which if removed falls back to the original filter.

where the filter function $\mathbf{s}$ gives an estimate for the state $\hat{\mathbf{x}}'_{k|k}$ based on all information up to and including time step k. The prime is added to differentiate the filter state estimate from the final state estimate $\hat{\mathbf{x}}_k$. To correct $\mathbf{s}$ we must define the state error committed by the filter:

$$\tilde{\mathbf{x}}_k = \mathbf{x}_k - \hat{\mathbf{x}}'_k \tag{102}$$

$$\tilde{\mathbf{x}}'_k = \hat{\mathbf{x}}_k - \hat{\mathbf{x}}'_k \tag{103}$$

where $\tilde{\mathbf{x}}_k$ is the true error committed by the state estimate of the filter $\hat{\mathbf{x}}'_k$ based on the truth $\mathbf{x}_k$ which is derived from some high fidelity source. An ANN will now be trained on the true error $\tilde{\mathbf{x}}_k$ to provide a correction:

$$\tilde{\mathbf{x}}'_k = \mathbf{s}_c^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{z}_k, \mathbf{h}_{k-1}^{NN} | \boldsymbol{\theta}^C) \tag{104}$$

where $\tilde{\mathbf{x}}'_k$ is the estimate of the true error committed by the filter $\tilde{\mathbf{x}}_k$. The ANN uses the full system inputs and measurements $\mathbf{u}_k, \mathbf{z}_k$ while the filter is restricted to only the system inputs and measurements it knows how to incorporate $\mathbf{u}_k^f, \mathbf{z}_k^f$. The ANN correction is then applied to correct the state estimate of the filter to close the gap between the filter and the truth:

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}'_k + \tilde{\mathbf{x}}'_k \tag{105}$$

where $\hat{\mathbf{x}}_k$ is now the ANN corrected best estimate of the state. This corrected estimate will be used as input to the $\mathbf{s}()$ function in Equation 101. This corrected state can then be used to estimate the output:

$$\hat{\mathbf{y}}_k = \mathbf{o}(\hat{\mathbf{x}}_k | \boldsymbol{\theta}^O) \tag{106}$$

where $\hat{\mathbf{y}}_k$ is the final output from the corrected state filter. The output as one equations is then:

$$\hat{\mathbf{y}}_k = \mathbf{o}\left(\mathbf{s}(\mathbf{x}_{k-1}, \mathbf{u}_k^f, \mathbf{z}_k^f, \boldsymbol{\theta}^X) + \mathbf{s}_c^{NN}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}_k, \mathbf{h}_{k-1}^{NN}, \boldsymbol{\theta}^C) | \boldsymbol{\theta}^O\right) \tag{107}$$

With this change from correcting filter output to correcting internal filter state the ANN can now improve the state estimate made by the filter in a way not possible in Section III. This state correction has the benefit of stabilizing a filter state that could have been potentially unstable. For example, there may have been a state that was not updated by measurements which caused the filter state to drift. However, when the ANN corrected this state it may have become stable and stopped drifting. If the filter becomes stable, the ANN may have used unknown measurements $\mathbf{z}_k^u$ to provide the stability. Unlike in the previous section, the state correction is made *before* the state is given to the next time step or the output function.

This corrected state is internal to the filter which allows relationships in the filter equations to benefit form this more accurate state. If the ANN correction is removed the state will begin to drift from its corrected starting point.

This type of combination may make training an ANN more difficult. In this type of correction, the filter state at the next time step is a function of the ANN estimate. The simple approach to train the state correction ANN would be to generate estimates from the filter at every time step and present the error at each time step to the ANN as the supervised output target. The ANN would then learn the errors committed by the filter at each time step. However, using this simple approach, at runtime the ANN will overestimate the error committed by the filter. This is because the filter state estimates contain less error since the filter uses corrected state estimates at the start of each time step. To correctly estimate the error at runtime, the ANN output target must include this feedback at every time step or else the ANN will overestimate the errors the filter makes. To correct this problem, after N weight updates the corrected filter output must be recalculated using the updated version of the ANN. This will then change the output targets for the next training epoch of the ANN. When this process is repeated, the ANN optimizer will eventually converge on a local minimum or saddle point resulting in the same corrections for each epoch. Thus, the output target for the next update will not change significantly. This extra step of recalculating the target output for the ANN may make training more difficult. However, if the optimizer does train the ANN, then the ANN will correctly estimate the errors committed by the filter when it uses corrected states to update at runtime.

ANN Correcting Filter Propagate with ANN.

Another method an ANN can correct a filter is specifically in the state predict step [93, 37, 110, 14, 106, 97, 52]. This placement aims to help only the state predict function with errors in the state dynamics equation. A block diagram of this entire process is shown in Figure 19.

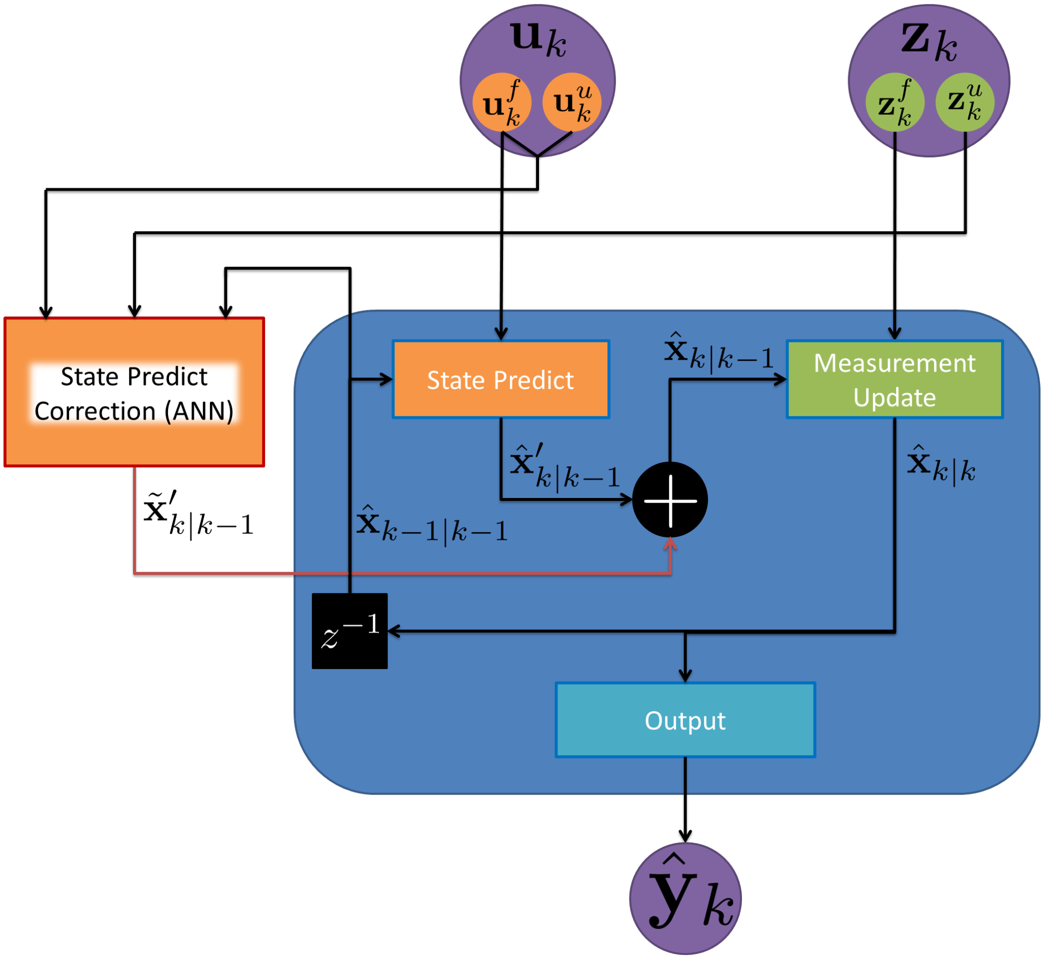

Figure 19: Block diagram of a generic standard filter with state predict correction provided by an ANN. The state predict correction ANN only takes the system inputs and unknown measurements as input. The objective of this ANN is only to improve the state predict block. The red line denotes the ANN provided correction which if removed falls back to the original filter.

The ANN will once again attempt to estimate the errors committed by the a priori state predict function and add in a correction. The state estimate from the filter before a measurement update is:

$$\hat{\mathbf{x}}'_{k|k-1} = \mathbf{f}(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k^f | \boldsymbol{\theta}^F)$$

where the filter uses an a priori state predict equation $\mathbf{f}$ to predict the state at time $k-1$ to the state at time step k. The filter uses the previous state estimate $\hat{\mathbf{x}}_{k-1|k-1}$ and the current system input it can incorporate $\mathbf{u}_k^f$. This estimate contains error from the true solution according to the following two equations:

$$\tilde{\mathbf{x}}_{k|k-1} = \mathbf{x}_{k|k-1} - \hat{\mathbf{x}}'_{k|k-1} \tag{108}$$

$$\tilde{\mathbf{x}}'_{k|k-1} = \hat{\mathbf{x}}_{k|k-1} - \hat{\mathbf{x}}'_{k|k-1} \tag{109}$$

where $\tilde{\mathbf{x}}_{k|k-1}$ is the true error committed by the state estimate $\hat{\mathbf{x}}'_{k|k-1}$ of the $\mathbf{f}()$ function based on the truth $\mathbf{x}_{k|k-1}$. An ANN will now be trained to estimate the true error $\tilde{\mathbf{x}}_{k|k-1}$:

$$\tilde{\mathbf{x}}'_{k|k-1} = \mathbf{f}_c^{NN}(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k, \mathbf{z}_k^u, \mathbf{h}_{k-1}^{NN}|\boldsymbol{\theta}^C) \tag{110}$$

where the ANN uses all information at the current time including the previous state estimate $\hat{\mathbf{x}}_{k-1|k-1}$, all the system inputs $\mathbf{u}_k$, and the measurements the filter cannot incorporate $\mathbf{z}_k^u$. The only information not used by the ANN is the set of measurements the filter knows how to incorporate $\mathbf{z}_k^f$. This is different compared to Sections III and III where the ANN was given all available data. The state predict correction ANN is not given the measurements the filter will use to perform the measurement update. The measurement update step is left alone and keeps using its same measurements $\mathbf{z}_k^f$. If the known measurements $\mathbf{z}_k^f$ are used in the state predict correction ANN, the ANN may start to do some of the calculations performed by the measurement update step. This would make the filter overconfident in the state, because the state predict and measurement update no longer use independent data. However, we allow the ANN to use unknown measurements $\mathbf{z}_k^u$, because the divide between system inputs and measurements can be problem specific as stated in Section III. In general, the design does not have to use unknown measurements $\mathbf{z}_k^u$ if the user desired to actually control this system or make prediction into the future. After the ANN estimates the error committed by the filter, the correction is added to the filter estimate to make the final state predict estimate:

$$\hat{\mathbf{x}}_{k|k-1} = \hat{\mathbf{x}}'_{k|k-1} + \tilde{\mathbf{x}}'_{k|k-1} \tag{111}$$

where $\hat{\mathbf{x}}_{k|k-1}$ is the corrected state estimate for time step k given measurements up to and including $k-1$. This state estimate is now updated by the measurement update function to include the measurements at time step k:

$$\hat{\mathbf{x}}_{k|k} = \mathbf{mu}(\hat{\mathbf{x}}_{k|k-1}, \mathbf{z}_k^f|\boldsymbol{\theta}^{MU}) \tag{112}$$

where the filter uses the measurements $\mathbf{z}_k^f$ to update the current propagated state from time $k-1$ to k. Finally this state estimate is used to calculate the output estimate at the current time:

$$\hat{\mathbf{y}}_k = \mathbf{o}(\hat{\mathbf{x}}_{k|k}|\boldsymbol{\theta}^O) \tag{113}$$

As with the state estimate correction in III the state predict correction ANN must also be trained with the a priori state predict function in the training loop. Since the corrections provide a better estimate for the state predict at the next time step, the estimate at the next time step will also be more accurate. If the ANN is trained on a static training set of predictions by the filter, the ANN will begin to overestimate errors committed by the filter. Thus, when training is performed, the a priori state predict function FILTER$_P$ must be incorporated into training to ensure the ANN is estimating the correct error committed by the filter.

ANN Measurements For Filters.

The second general method to integrate an a priori filter and an ANN is to have the ANN provide measurements for the filter [28]. A block diagram of this type of integration is shown in Figure 20. This integration is accomplished by training the ANN to produce measurements which can be included in the a priori filter. The ANN input includes measurements which the filter could not incorporate. Incorporating the measurements from the ANN gives the filter information from these previously unusable measurements. The functional representation of this system is thus:

$$\hat{\mathbf{z}}_k^n = \mathbf{m}^{NN}(\mathbf{z}_k, \mathbf{h}_{k-1}^{NN}|\boldsymbol{\theta}^M) \tag{114}$$

$$\mathbf{z}'_k = \left[\mathbf{z}_k^f \oplus \hat{\mathbf{z}}_k^n\right] \tag{115}$$

$$\hat{\mathbf{y}}_k = \text{FILTER}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{z}'_k|\boldsymbol{\theta}) \tag{116}$$

where $\hat{\mathbf{y}}_k$ is the estimate of the output at time step k, $\mathbf{x}_k$ is the state of the filter at time step k, $\mathbf{u}_k$ is system inputs at time step k, FILTER() is the a priori filter function, and $\mathbf{z}_k$ is the measurements at time step k. The function $\mathbf{m}^{NN}()$ determines new measurements $\hat{\mathbf{z}}_k^n$ from existing measurements $\mathbf{z}_k$. The measurements are broken into $\mathbf{z}_k^f$ and $\mathbf{z}_k^u$. The measurement $\mathbf{z}_k^f$ is the part of the measurements from $\mathbf{z}_k$ which the filter naturally incorporates. The measurement $\mathbf{z}_k^u$ is the part of the measurements from $\mathbf{z}_k$ which the filter cannot incorporate.

New measurements $\mathbf{z}_k^n$ are measurements which the filter can incorporate and are available during training, but are not available during run time. The ANN provides an estimate of these new measurements $\hat{\mathbf{z}}_k^n$ at runtime. The total measurements to the filter are then $\mathbf{z}'_k = \mathbf{z}_k^f \oplus \hat{\mathbf{z}}_k^n$. The measurements to the filter has been replaced with an concatenation of the raw measurements the filter could incorporate $\mathbf{z}_k^f$, and the new measurement estimates determined the ANN $\hat{\mathbf{z}}_k^n$. Note that the estimated measurements by the ANN must be values which the filter can incorporate. In this instance the filter could incorporate measurements $\mathbf{z}_k^f \oplus \mathbf{z}_k^n$. However, the true measurements for $\mathbf{z}_k^n$ are only available during training, not at runtime. At runtime the true measurements are replaced by an estimate provided by the ANN $\hat{\mathbf{z}}_k^n$. The advantage to this approach is that the ANN can use the unknown measurements $\mathbf{z}_k^u$ to aid in determining the new measurements $\hat{\mathbf{z}}_k^n$.

The measurement ANN can also use the known measurements in order to estimate the new measurements. This synergy may allow a better estimate of the new measurements. However, using known measurements may also make the new measurements highly correlated with the known measurements. The filter must be informed of this dependence

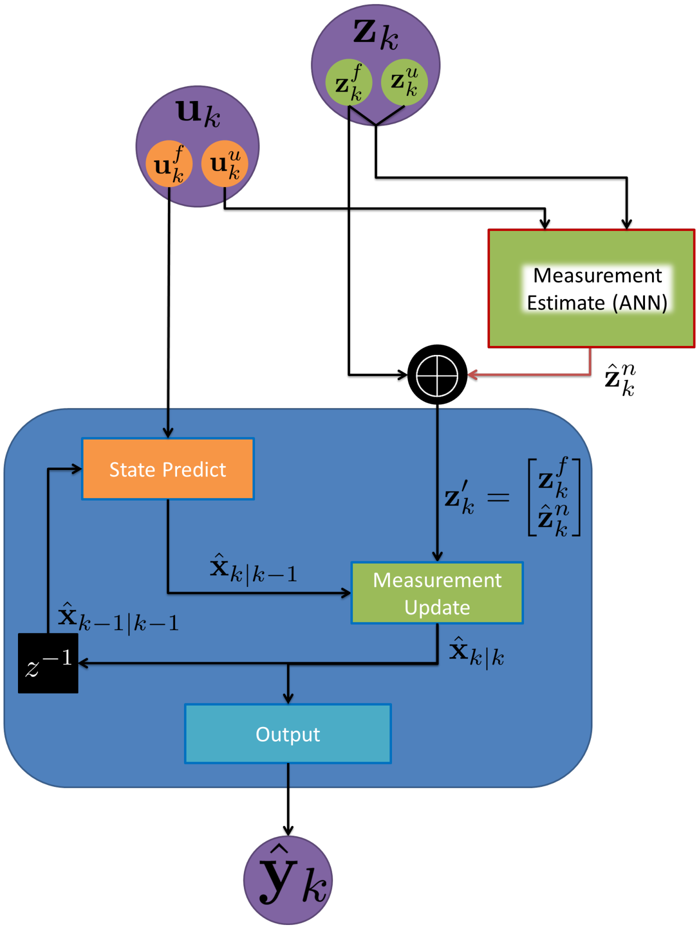

Figure 20: Block diagram of a generic standard filter with measurements provided in part by an ANN. The blue rounded block represents the filter operation FILTER(). The overall filter has three components the state predict, measurement update, and output.

so that the filter does not become overconfident in the state estimates. For example, assume that the new measurements are a high fidelity GPS receiver and known measurements are a low grade GPS receiver. The measurement ANN then uses the known GPS measurements and unknown measurements to provide a close estimate of the new measurements from the high fidelity GPS receiver. The filter would include information that the two GPS measurements are correlated. The method to incorporate the dependence of the measurements would be the same process if we had always included two GPS receivers as measurements or other similar correlated measurements.

The measurement ANN method allows incorporating a priori system dynamics and measurement update knowledge into the filter while being flexible enough to use measurements which have have little a priori information. This method can also estimate measurements which the filter may be able to incorporate but the relationship is very complex, and simplification is difficult or degrades performance. In this dissertation, the VLF measurements have little a prior information and cannot be related to position. There is no known way to use the raw VLF features to estimate position. Also, it is very costly to determine the relationship between the VLF measurements and position manually. Determining the relationship manually would involve locating and characterizing all the VLF transmitters in a given area and determining a measurement update for each transmitter. Instead the ANN is trained to estimate position using the FFT features. This turns measurements which we did not know how to incorporate (VLF measurements) into measurements which we can incorporate (position).

The measurement ANN allows a standard filter to make state estimates while incorporating extra information with very little a priori information. All existing filter techniques still apply such as state predict, measurement update, outlier rejection, and anything else performed by the a priori filter. This approach essentially allows the ANN output to be treated as new sensor information.

ANN Adaptive Filter.

Another method to integrate ANNs and standard filters is to allow the ANN to change filter parameters [94, 49, 48, 57]. A block diagram of this is shown in Figure 21. This style of integration maintains the functional form of the standard filter but allows an ANN to modify the filter parameters in response to changes in system input, measurements, and current state. In equation form this is:

$$\hat{\mathbf{y}}'_k = \text{FILTER}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}^f_k, \mathbf{z}^f_k | \boldsymbol{\theta}_k) \tag{117}$$

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}^{NN}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}^f_k, \mathbf{z}^f_k | \boldsymbol{\theta}^D) \tag{118}$$

where $\mathbf{x}_k, \mathbf{u}_k, \mathbf{z}_k$ are the state, system inputs, and measurements respectively. FILTER() is the standard filter function parameterized by $\boldsymbol{\theta}_k$. However, the filter parameters are actually a function of the state, system inputs, and measurements. This allows the a priori filter to adapt to changes in the system. For example one such change could be a range dependent range error. In such a case, the **R** matrix in a Kalman filter must change depending on the measurement. The introduction of the adaptive parameter ANN allows the changes in parameters to be learned by empirical data. To train the ANN, the filter would calculate state estimates, and the ANN would tune parameters to minimize the filter state error. Adaptive filtering can also be done on filter functions defined by ANNs. ANNs also have parameter weights which could be adapted based on the state, system inputs, and measurements. This dissertation will not explore this option, because the machine learning literature was sparse on how to train such an ANN however, there was no reason why it could not be done.

ANN Filter Integration Overview.

The previous sections outlined five different methods to augment an a priori filters with ANNs. This section outlines an overall algorithm in order to combine different methods to augment some a priori filter. This algorithm process is outlined as a flowchart in Figure 22.

The first step is to obtain some a priori base filter to augment. This filter must be able to create final state estimates before any integration is done. The next step involves three decisions relating to adding correction ANNs to the base filter. The first decision is if state predict corrections should be added. State predict correction should be added when the current state predict function has poor performance and/or has room for improvement.

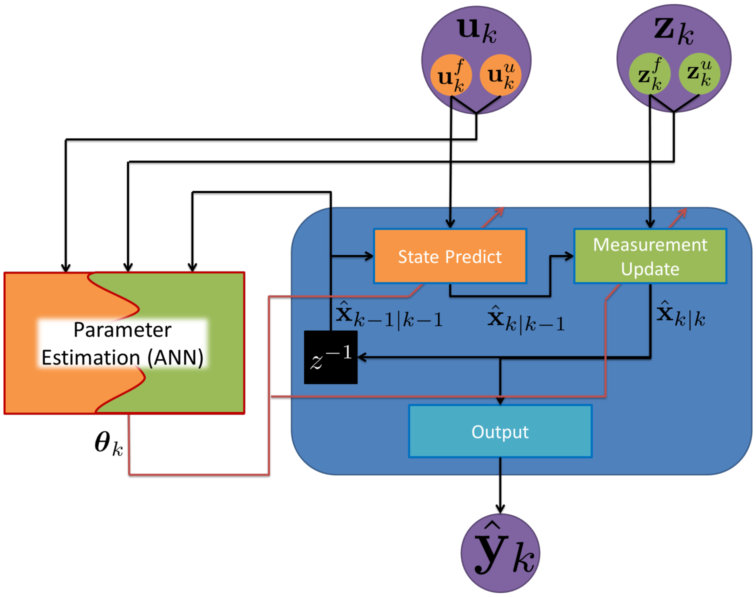

Figure 21: Block diagram of a standard filter with parameters provided by ANNs.

For example, if the state predict function is a simplification of something that is know to be more complex, then a correction may improve the estimate of the final propagated state estimate.

The second correction type decision is if state estimate corrections should be added after the measurement update. State estimate corrections should be added when the current measurement update function has poor performance and/or has room for improvement. This correction may also make corrections related to state propagation especially if there is no state predict correction. However, this correction should not be chosen to specifically correct problems with the state predict function.

The third correction type decision is if filter output correction should be added. This correction should be added for two possible reasons. The first reason is if the user does not want to modify the internal filter estimates due to training difficulties or trust in ANNs. The second reason is if the user believes the internal state is difficult to correct, and the output correction may provide better performance. While theoretically, the state

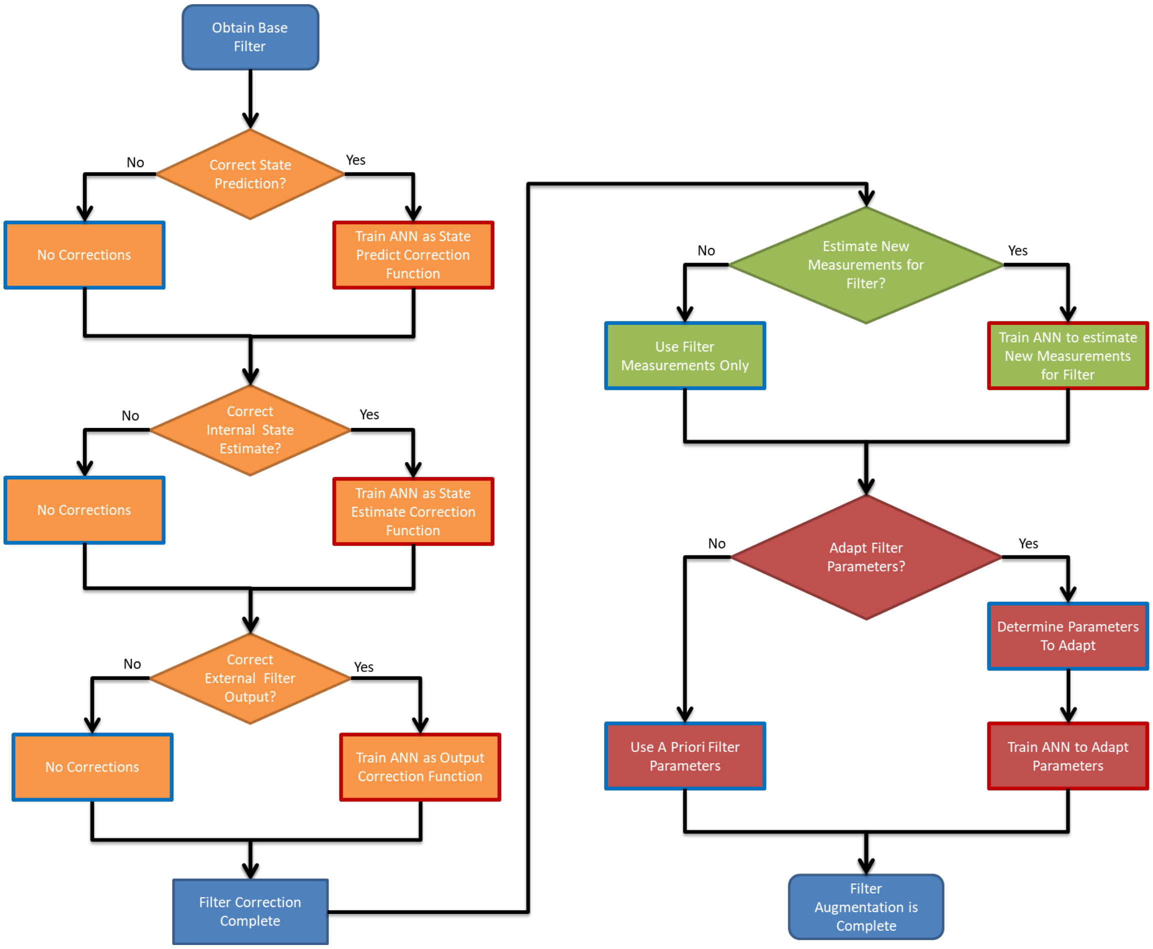
Obtain Base Filter
Correct State Prediction?
No
Yes
No Corrections
Train ANN as State Predict Correction Function
Correct Internal State Estimate?
No
Yes
No Corrections
Train ANN as State Estimate Correction Function
Correct External Filter Output?
No
Yes
No Corrections
Train ANN as Output Correction Function
Filter Correction Complete
Estimate New Measurements for Filter?
No
Yes
Use Filter Measurements Only
Train ANN to estimate New Measurements for Filter
Adapt Filter Parameters?
No
Yes
Determine Parameters To Adapt
Use A Priori Filter Parameters
Train ANN to Adapt Parameters
Filter Augmentation is Complete

Figure 22: Flowchart for determining which integrations to include. Orange blocks relate to base filter corrections. Green blocks relate to providing new measurements to the base filter. Red blocks relate to adapting parameters of the base filter.

estimate correction in the previous paragraph can provide the same corrections as the output correction, in practice this may not be true. For example, if there are many states compared to outputs, and the output function is complex, it may be easier to train an ANN with the smaller set of inputs. The complex output function may make it difficult for the state estimate to operate on the states. However, the output correction function corrects the output directly which may be simpler to correct. Thus, the output correction may sometimes in practice make better corrections compared to the state estimate.

The next section of the flowchart involves adding a measurement ANN. A measurement ANN should be added for two possible reasons. The first reason is if the user believes there are unknown measurements that contain state information, but the base filter cannot incorporate the unknown measurements. The measurement ANN would bridge the gap between these unknown measurements and the states by estimating new measurements the base filter can incorporate. The second reason would be if the user believes that the measurement update function(s) do not adequately represent the relation between the measurements and states. For example, if the measurement update functions were a simplification of some complex phenomenon. Thus, by estimating the state with known measurements, more information about the relationship between state and measurements could be gained compared to using the measurement update function(s) alone. As always when adding a measurement ANN, (but especially if done for this reason), the dependence of the measurements must be given to the filter to prevent the filter from producing overconfident estimates.

The final section of the flowchart involves adjusting parameters using an ANN. If the user decides to adjust parameters, the first step is to determine which parameters to adjust, since not all the parameters of the base filter must be adjusted. The parameters that should be adjusted are the parameters the user believes change over time based on the current system inputs, measurements, and state. Adjusting too many parameters may make training the adaptive ANN difficult, and filter performance could suffer.

After following the full flowchart, the base filter will be augmented with all the different types of ANNs augmentations applicable to this problem. However there are some guidelines to follow during this process to make the integration a success. As with the ANN as filter parts integration method, these guidelines are outlined in the next Section III.

Combined Framework Overview

This chapter so far has given two different classes of how to combine ANNs and filters. One class viewed integration as ANNs substituting filter functions. The other class viewed integration as ANNs augmenting existing a priori filters. This section outlines how to combine these two different methods into one overall framework and gives overall guidelines for integrating ANNs into filters.

Combining the two different classes of integration is a straightforward process of following the two flowcharts in order. The process starts by following the ANN as parts flowchart found in Figure 16. This algorithm produces a base filter which may or may not have ANN components. The next step is to follow the ANN filter integration flowchart found in Figure 22. However, this flowchart begins with the base filter determined in the first step. Thus, the algorithm will augment the base filter (which may have ANN components) with other ANNs. The final result is an algorithm for an overall framework to combine ANNs and filter. However, there are some guidelines to follow when using this algorithm.

One guideline to follow is to limit the ANN inputs to only what is necessary. In general this chapter has given ANNs all possible inputs in order to show the full extent of inputs. However, inputs for the ANN will require more supervised training data to train the ANN. Thus, a priori knowledge and experimentation maybe necessary to trim the inputs of the ANN in order to allow adequate training with the available training data.

The next guideline is that high capacity ANNs with more inputs are more difficult to train compared to smaller low capacity ANNs with few inputs. Thus, while making decisions to break up a function or not, the user must realize that if the the function is kept whole and will be an ANN, it may be more difficult to train compared to multiple simpler

ANNs with less inputs. This will also make the overall algorithm more transparent as the output passed between these smaller ANNs can be observed. For example, monitoring the output of the ANNs may shed light on which sections of the problem are more difficult.

Another guideline is that replacing a priori functions with ANN should have a reason. If the performance is adequate with a priori functions it may be a waste of time to develop and ANN solution that may increase performance. The user should favor a simpler filter and only add more ANNs based on final performance goals. However, if there is no a priori function, or the performance is poor, then using an ANN can remove a weak link in the filter.

The last guideline is to use as much a priori knowledge as possible. While the reason for using an ANNs is that the current a priori knowledge is only part of the complete picture, a priori knowledge should be incorporated into the filter where possible. This is the entire reason for the second class of integrating ANNs with filters. If there is knowledge about the system, an a priori function should incorporate the existing information if possible. The ANN should fill in the gaps where the current a priori knowledge is incomplete or inaccurate. For example, if it is known that a position state is the derivative of a velocity state, then explicitly describing this relationship in the state predict function will guarantee that the filter knows this relationship. It is possible an ANN will learn this relationship from empirical data, but the purpose of the ANN is to learn relationships that are unknown, not relearn known relationships.

In conclusion, the flowchart provides guidance on how to build a filter, but the user must also follow some general guidelines to use their engineering judgment and a priori knowledge to create a successful filter.

State of the Art ANN Integration Revisited

This section revisits the previous research into ANN integration in order to fit previous work into the framework of this chapter. Throughout this Chapter references were given where each kind of integration has been found in the literature except two types of integration. The first type not found in the literature is the state estimate correction ANN outlined in Section III. This type of integration was never explicitly found in the literature. However, research implementations of correction the filter output outlined in Section III were not always explicit if the corrected output was used as the starting estimate for the next time step. It is possible one of these implementations did implement a state estimate correction ANN.

The other type of integration found in literature, but not explicitly outlined in the framework, is the EKF optimizer [89, 32]. The EKF optimizer, as outlined in Section II, is unique because the algorithm is first and foremost an optimizer, not a filter for state estimation. The EKF optimizer only becomes a filter in the traditional sense when the state vector includes states that are not weights of the ANN as shown in Equations 73 and 77. At this point, the filter predicts and updates both the weights and states. In an offline application this would be equivalent to training the state predict function and measurement predict function using an EKF optimizer, then running the ANN integrated EKF filter with these weights. This is an example of ANN as filter parts from Section III. In an online application we allow the parameters of the ANN to be inputs by modifying the ANN equation from:

$$\mathrm{a}^{NN}(\mathbf{x}, \mathbf{h}^{NN} | \boldsymbol{\theta}^{A}) \rightarrow \mathrm{a}^{NN}(\mathbf{x}, \mathbf{h}^{NN}, \boldsymbol{\theta}^{A}) \tag{119}$$

where we changed the ANN from having the parameters or weights given $\boldsymbol{\theta}^{A}$, to having the parameters or weights as an input $\boldsymbol{\theta}^{A}$. This once again fits into the ANN as filter parts in Section III where we also use an adaptive filter from Section III.

Thus the previous history of ANN integration fits into the architecture given in Sections III, and III.

IV. Neural Network Confidence Improvements

No matter how the user will integrate ANNs and filters, there will be one common thread with the ANN output estimates – they must provide covariance. Estimates without covariance are useless to a filter for navigation. It is possible to use an average covariance determined after the model is trained by evaluating the variance of the error. However, providing a covariance per sample improves covariance performance, since it takes into account the current ANN inputs. A popular type of ANN that provides covariance estimate is an MDN outlined in Section II. This chapter defines some improvements over existing methods that apply to MDN which output GMM parameters. The first two sections outline improvements that allow easier training and better performance compared to classical MDN training. The last section defines methods to measure and quantify the errors in estimating covariance made by the MDN.

Gaussian Mixture Model Custom Loss Function

This section defines a method to improve estimates made by standard GMM loss functions defined in Section II by separating the loss function into two parts. Each part is the loss for just covariance or mean estimates. The GMM loss function is the standard method to determine covariance from ANN, thus improving this loss function will produce better results for applications that require covariance estimates. This method also allows to optimize the means estimates with a loss function that is not implicitly tied to the distribution of the MDN.

During training, to minimize a GMM loss function as shown in Equation 62, the optimizer will make changes to three different values to lower the loss – means, covariances, and weights. At the beginning of training it is likely that the covariance is a poor estimate of the error between the means and targets. When the covariance is too large there are two values that can be changed to correct this inaccuracy – covariance or means. Intuitively if the covariance is decreased it will better match the error. However, if the error is increased it may also make the covariance more accurate and thus decrease the loss in the GMM loss function. In order to disallow increasing error to improve covariance accuracy, we must separate the loss function into two parts where means and covariance do not interact. We shall do this be adding a new loss function to our current GMM loss as shown:

$$\mathcal{L}_{GMM^*} = \mathcal{L}_{GMM} + \beta \mathcal{L}_{custom} \tag{120}$$

where β is a scaling factor to compare the loss functions and our new loss function, or custom loss function, which is defined as:

$$\mathcal{L}_{custom} = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{M-1} w^j custom(\mathbf{y}^i, \boldsymbol{\mu}^i_j) \right) \tag{121}$$

where *custom()* is simply a function to compare a target vector **y** to a predicted output. The same target vector **y** is compared for every j^{th} mean $\boldsymbol{\mu}_j$ from the M mixtures of Gaussians predicted at the i^{th} sample. The loss from each target and mean combination is then scaled by the weight of that Gaussian w_j. The custom loss function only depends on the means, weights and output targets.

Next we must remove the dependence of means from the standard GMM loss function $\mathcal{L}_{GMM}$. To remove the means from the standard GMM loss we will alter the derivative with respect to the means of the standard GMM loss. We can alter the gradient without changing the function by utilizing new software that make gradient changes a simple process.

New symbolic math software automatically computes the gradients of the operations described in the ANN architecture. This process makes algorithms such as back propagation [38] simple to implement. Recent software like TensorFlow [1] allows this automatic process to be manually edited by the user. The user can now let the software perform the majority of the back propagation algorithm but add a few extra changes. This dissertation uses these extra changes to remove the MSE component from the GMM loss function commonly used in MDNs. Next we outline the method to implement the gradient change.

GMM Custom Loss Function Method.

Our goal in this section is to describe how to alter the gradient of the GMM loss in order to make a GMM Loss function $\mathcal{L}_{GMM^*}$ which does not optimize the means with the implicit optimization of the distribution. In our case, Gaussian distributions implicitly use a MSE loss function $\mathcal{L}_{MSE}$ when incorporated in MDN [7] and [103]. Instead we wish to use our own custom loss function $\mathcal{L}_{custom}(\mathbf{y}, \boldsymbol{\mu}_{0...M-1})$ to optimize the means of the GMM parameters. The first step in understanding how to alter the gradient is to know what the current gradient is.

As explained in detail in [7] and [103], the GMM loss function will optimize the means in a similar fashion to MSE. The MSE similarity comes from the fact that the derivative of the GMM loss function with respect to our MDN output will be of a similar form to the derivative of the MSE loss function with respect to the ANN output. To show this similarity, we first start with the MSE loss function for one sample:

$$\mathcal{L}_{MSE} = \frac{1}{D} \sum_{d=0}^{D-1} (\mu_d - y_d)^2 \tag{122}$$

where μ_d is the estimate from the ANN of target y_d in the d^{th} dimension. We take the derivative of the MSE loss function with respect to the ANN output at each dimension for one sample is as follows:

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \mu_d} = \frac{2}{D} (\mu_d - y_d) \tag{123}$$

In vector form this equation is:

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \boldsymbol{\mu}} = \frac{2}{D} (\boldsymbol{\mu} - \mathbf{y}) \tag{124}$$

From this equation we note that the derivative of the loss function with respect to the output is proportional to the error between the output and target with the same scale factor for all dimensions and samples.

Next, we determine the derivative of one of the Gaussian distributions in the GMM loss function with respect to the MDN output for one sample (see Appendix Section A for steps).

$$\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - \mathbf{y}) \tag{125}$$

From the matrix form it is clear that the derivative of each dimension is now correlated with other dimensions through the covariance information in $\boldsymbol{\Sigma}$. However, the derivatives are still proportional to the error between the predicted output and the targets. Next, comparing Equation 124 and Equation 125 we see that the derivative of MSE loss applies a constant scale factor to each dimension, while the derivative of GMM loss mixes the error through the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$.

When the dimensions in GMM are independent, the covariance matrix becomes a diagonal matrix and becomes closer to MSE loss. The independent GMM loss applies only a possibly different scaling factor to each dimension. If the variances were further assumed to be a constant, $2/D$ then all dimensions would have the same scale factor and become the same as the derivative of MSE loss. Thus, when the assumptions of MSE loss are applied to the GMM loss, the derivative functions will become similar.

Equation 125 is where we can see how large error with appropriate covariance receives a smaller gradient, and thus smaller weight updates, through the interaction of means and covariance. For example, assume the covariance matrix is diagonal, and thus the inverse covariance matrix is diagonal, with each diagonal being the inverse of the diagonal in the covariance matrix. Each of these diagonals in the inverse covariance matrix will be multiplied by the error between the mean and target in the same dimension. The result of this multiply will be multiplied by the gradient which determines how large the weight updates will be. Thus, if one dimension has large error, the gradient update will be large as well. However, the error is also multiplied by the inverse covariance. If the covariance is large the inverse will be small and thus shrink the error and resulting gradient update. This is how accurate large covariance estimates will deter reducing the error in order to maintain accurate covariance.

From our observation of the derivatives of the loss functions with respect to the ANN and MDN outputs, we conclude that modifying the partial derivative $\frac{\partial \mathcal{L}_{GMM}}{\partial \mu}$ of the GMM loss function would alter the underlying loss function to optimize the means of the GMM parameters.

We wish to alter this derivative in order to remove the interaction of the means and covariance. To accomplish this, we simply remove the derivative of the GMM loss with respect to the means. To remove this component of the derivative we set $\frac{\partial \mathcal{L}}{\partial \mu} = 0$ when calculating derivatives. Now the standard GMM loss is unable to make weight updates to optimize the means. However we still need to optimize the means to minimize some kind of loss. This is why we introduced our own derivative by adding our custom loss function to the GMM loss function to create a new overall loss function from Equation 120.

Now our derivative with respect to the means will be:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial \boldsymbol{\mu}} = \frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} + \frac{\partial \mathcal{L}_{custom}}{\partial \boldsymbol{\mu}} \tag{126}$$

however we have set $\frac{\partial \mathcal{L}_{GMM}}{\partial \mu} = 0$ so our final partial derivative is:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial \boldsymbol{\mu}} = \frac{\partial \mathcal{L}_{custom}}{\partial \boldsymbol{\mu}} \tag{127}$$

where now the means are optimized in accordance to our new custom loss function only since we have removed the contributions from the GMM loss function. However, we have left the derivatives with respect to the weights and covariance terms untouched. Thus, the GMM loss function will still attempt to optimize the covariances and weights of the GMM in order to express the uncertainty of the means (which are being optimized by our custom loss function). Thus our derivative with respect to the covariances will be:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial \boldsymbol{\Sigma}} = \frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\Sigma}} + \frac{\partial \mathcal{L}_{custom}}{\partial \boldsymbol{\Sigma}} \tag{128}$$

since the custom loss function is not a function of Σ we know the derivative will be zero $\frac{\partial \mathcal{L}_{custom}}{\partial \mathbf{\Sigma}} = 0$ so our final partial derivative is:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial \mathbf{\Sigma}} = \frac{\partial \mathcal{L}_{GMM}}{\partial \mathbf{\Sigma}}. \tag{129}$$

In conclusion we have shown a method to remove the interaction between covariance and means in the GMM loss function in order to prevent the optimizer from increasing error to match covariance. In addition, this method allows a custom loss function to determine the means of the distribution while still providing covariance.

Gradient Correlation and Scaling.

In our method of GMM custom loss function, the custom loss function has no dependence on the covariance between dimensions. This was done on purpose to remove the interaction between means and covariance. However, in some cases it may be desirable to reintroduce this correlation, for instance, if the original goal was to introduce scaling or a custom loss function but not remove mean and covariance interaction. To bring this correlation back to our custom loss function, we once again will modify the derivative function to incorporate this new information. We define a new derivative of the custom loss function with respect to the means, based on the custom loss derivative and the original derivative Equation 125:

$$\frac{\partial \mathcal{L}_{custom}}{\partial \boldsymbol{\mu}}^{*} = \sum_{j=0}^{M-1} \mathbf{\Sigma}_j^{-1} \frac{\partial \mathcal{L}_{custom}}{\partial \boldsymbol{\mu}_j} \tag{130}$$

where this equation is for one sample of the N total samples and $\mathbf{\Sigma}_j^{-1}$ is the inverse covariance matrix of the j^{th} mixture out of M mixtures. Now we have reintroduced the covariance information from the GMM loss derivative. This new derivative now correlates the dimensions of the output so the optimizer can modify the correlated states in unison. This method creates a step in between standard GMM loss and our custom loss function by allowing mean/covariance interaction but removing the implicit MSE loss and introducing a scaling factor between mean and covariance.

Gradient Normalization for Multiple Loss Functions.

With the introduction of multiple loss functions in Section IV we introduced a scaling factor β to weight the relative differences between the loss functions. The scaling factor increases or decreases the relative loss attributed to each loss function. However, our main goal was to change the *gradient* to encourage the optimizer to adjust weights in order to improve one of the loss functions more compared to the other. The larger scale factor will make the loss larger, and subsequently the gradients from that loss should follow. However, the magnitude of the gradients does not have to be correlated with the magnitude of the loss. For example, a function that is evaluated with a high magnitude may have a very small derivative.

We actually want to scale the gradients of the loss functions relative to each other. To achieve gradient scaling, we normalize the magnitude of the gradients with respect to each function. Then we will apply a scaling to the normalized gradient. Scaling the gradient assures a relative scaling of the impact on the weights from the two loss functions, since the gradient directly impact the weight updates. To demonstrate gradient scaling we write the gradient of the loss function with respect to the ANN output parameters z:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial z} = \frac{\partial \mathcal{L}_{GMM}}{\partial z} + \beta \frac{\partial \mathcal{L}_{custom}}{\partial z} \tag{131}$$

Next we will normalize the gradients and apply our β scaling directly to the normalized custom loss function gradient:

$$\frac{\partial \mathcal{L}_{GMM^*}}{\partial z} = \frac{\frac{\partial \mathcal{L}_{GMM}}{\partial z}}{\left\| \frac{\partial \mathcal{L}_{GMM}}{\partial z} \right\|_2} + \beta \frac{\frac{\partial \mathcal{L}_{custom}}{\partial z}}{\left\| \frac{\partial \mathcal{L}_{custom}}{\partial z} \right\|_2} \tag{132}$$

This normalization enforces the gradient magnitudes of each function to always be in a fixed relative scaling to each other. Note that we have changed the original magnitude of the gradients by forcing one of the magnitudes to always be one. Due to this change it may be necessary to change the learning rate and learning rate scheduler to account for this change in gradient. The end result is that our optimizer will always make weights updates with a fixed relative scaling between the loss functions.

Mixture Density Network Training Methods

Implementing an MDN produces non-trivial practical issues that require special attention in order to train models without running into numerical computation problems, which are covered in the subsections that follow. The first practical issue is how to scale the output parameters when training to a scaled version of the output. The second practical issue is reducing the *mixture* of Gaussians into a single Gaussian to use with applications that require a single distribution. The next section describes how to fix a numerical computation problem in determining gradients when probabilities are rounded to exactly zero due to underflow. The last section defines a method to simplify calculating Gaussian log likelihoods which avoids numerical precision problems.

Output Scaling.

This section outlines how to scale GMM parameters when training to a scaled output. Training to a scaled output is done because training to large output values will be slow when using standard initialization algorithms. The standard initialization algorithms usually try to maintain layer outputs that are zero mean with a variance of one [23]. Thus if the target output is much larger than one, the optimizer will spend many weight updates to scale up the ANN output to the appropriate target output range. To avoid this problem, we may scale the target outputs using the equation:

$$\mathbf{y}' = (\mathbf{y} + \mathbf{b}) \odot \mathbf{s} \tag{133}$$

where $\mathbf{y}'$ is the scaled version of $\mathbf{y}$ and $\odot$ is the element-wise multiplication operator. The original target output is thus biased by $\mathbf{b}$ and scaled element-wise by $\mathbf{s}$. The values for $\mathbf{b}$ and $\mathbf{s}$ are chosen so that the resulting targets $\mathbf{y}'$ have zero mean and a standard deviation of one [25]. Our ANN will now estimate the scaled version of our target output $\mathbf{y}'$. This allows the optimizer to spend fewer updates trying to bring the prediction range in alignment with the target range, especially if the magnitudes of the original target range are much larger than one.

However, now that we are training to scaled output our MDN will output, a scaled version of the GMM parameters. To obtain unscaled estimates from the GMM probability distribution, it is more convenient to unscale the parameters and just calculate the unscaled distribution. To unscale the parameters of the GMM with σ and ρ values we use the equation:

$$\mu_i = \frac{\mu'_i - b[i]}{s[i]} \tag{134}$$

$$\sigma_i = \frac{\sigma'_i}{s[i]} \tag{135}$$

where $\boldsymbol{\mu}$ is the original mean and $\boldsymbol{\mu}'$ is the scaled mean determined by the MDN. The index i is for each dimension of the Gaussian and $s[i], b[i]$ are the i^{th} components of the scaling and biasing vectors, respectively, from Equation 133. If we were using the inverse covariance form with the upper diagonal $\mathbf{A}$ we would need to scale $\mathbf{A}$ as such:

$$A_{i,j} = A'_{i,j} s[j] \tag{136}$$

where $A'_{i,j}$ is i^{th} row and j^{th} column of the scaled upper diagonal $\mathbf{A}'$ determined by the MDN. Note that we scale by multiplying and not dividing like in Equation 135, because this is already the inverse form of the covariance matrix. Also note that we scale each column by the same corresponding value of our scaling vector and allow Equation 64 to distribute the scaling across rows. With this method we can train to scaled parameters and still obtain probabilities for the unscaled versions of the GMM distributions.

Single Gaussian Reduction.

While the GMM is useful because it can represent a variety of probability distributions, the multiple distributions can also be a problem when only one Gaussian distribution can be used, such as in a Kalman filter. The naive approach at reduction would be to limit the MDN output to only one Gaussian mixture. However, this creates problems when the middle of the two distributions is actually not likely [7]. For example, if the MDN probability was distributed between two parallel roads it may represent the Probability Density Function (PDF) as a bimodal distribution with two Gaussian distributions centered on each road. To represent this as one Gaussian distribution, the distribution would be centered in between the roads which may or may not be a valid output depending on the situation. A sensible reduction method is to only use the Gaussian distribution from the mixtures with the highest weight. In our road example this would reduce to one road which may not always be correct, but it will be a valid output, in the sense that it would lie on a road.

An extra piece of information which can be used when reducing the GMM is the magnitude of the weights of each mixture. The higher the weight of one Gaussian the more the mixture tends toward a single Gaussian distribution. When the weights are evenly distributed, the mixture will become less Gaussian. In certain applications which assume Gaussian distributions, we may use this property of the weights to reject distributions which are highly non-Gaussian. If these applications used non-Gaussian distributions, it would invalidate assumptions of the application and increase the resulting error.

In fact, when the underlying distribution is Gaussian, our experiments in MDN show that the MDN favors moving all the weight on one of the Gaussian mixtures. We observed that the MDNs favor adding weight to one mixture if possible instead of stacking distributions with similar means. Thus, the MDN will favor reducing the GMM distribution to a single Gaussian distribution if possible. In turn, choosing the highest weighted Gaussian will reduce to the correct underlying Gaussian distribution since the ANN will move all the weight onto one distribution if possible.

In summary, MDNs tend to make one Gaussian distribution have higher weight if possible. Thus, if a single distribution is required, reducing to the highest weighted distribution from multiple mixtures is a good assumption when the largest weight is much higher compared to the rest of the distributions.

Derivative Changes.

Theoretically, Gaussian distributions have probability over the entire input space and can have arbitrarily small covariances. However, when implemented in a computer, it is possible to have probability which underflows to exactly zero probability or other numerical instabilities. The underflows can be especially true early in training when the model has a poor estimate of the distribution or very late when the model increases certainty about estimates. Graves et al. [25] notes that later in training the GMM loss function tends to encounter numerical instability problems.

An example of instability occurs when model initialization causes all the means to be far from the target output and assigns low variance to each mean. This initialization creates a mismatch between the predicted means and the targets which could result in an underflow and create a zero probability estimate. The underflow will occur if all the means of one distribution are very far from the target. Thus, if all the mixtures of one sample have zero probability, we notice from Equation 56 that the loss function will attempt to compute $-\ln(0)$ which results in infinity. This means our loss function will have infinite loss. This in itself is not a problem, until we attempt to back propagate the loss to update the weights as is standard in ANN training [38]. As shown in the Appendix Section A this will lead to Not a Number (NaN)s in the derivative.

If a NaN appears in a derivative with respect to the loss function, then the NaN will propagate through the entire MDN and make all weights NaN via the chain rule of differentiation. However, this NaN was produced by only one sample of possibly many. To correct this problem we ignore the NaN sample and use all the rest of the samples to update the weights. If the sample with NaN is similar to other samples, eventually the NaN sample will return a non-zero probability. For our previous example, in order to ignore the

NaN created by the derivative of $-\ln(0)$ we modified the gradient of our loss function as follows:

$$\frac{\partial \ln(f(x))}{\partial x} = \begin{cases} \frac{f'(x)}{f(x)} & f(x) > 0 \\ 0 & f(x) = 0 \end{cases} \tag{137}$$

where $f'(x) = \frac{\partial f(x)}{\partial x}$. This will now ensure that NaN values will not pollute our weights. Returning a zero also tells the optimizer that there is no information from this target output which is true, since the shape of the distribution in this zero probability region is flat with no gradient to follow.

In general, there may be other cases where the derivatives become NaN due to numerical precision problems, but to determining all the cases may be difficult. To overcome this difficulty we apply a general modification to the gradient at the end of the MDN to catch all the NaNs that could have appeared along our differentiation of the loss function. This general modification is as follows:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \begin{cases} 0, & if\, \frac{\partial \mathcal{L}}{\partial \mathbf{z}} = NaN \\ \frac{\partial \mathcal{L}}{\partial \mathbf{z}}, & otherwise \end{cases} \tag{138}$$

where the derivative of loss function from Equation 56 with respect to the output GMM parameters from the MDN $\mathbf{z}$, is set to zero if the derivative is NaN. This will now catch any problem where numerical precision in our loss function causes unexpected NaN results in the derivative.

GMM Loss Probability Simplification.

When working with the numerical precision of computers we found that calculating exponents of 32 bit floats and then taking the natural log can lead to significant error during training due to floating point precision. Calculating exp with exponents in the hundreds quickly becomes the upper value of a 32 bit floating point number and thus introduces rounding. If we calculate the natural log after this rounding we will not end up with our original exponent. This calculation occurs every time we calculate the negative log likelihood of GMM loss function found in Equation 62, since our probability is from Equation 58 which contains $\exp(x)$ terms. To simplify this step and avoid the numerical precision loss, we can rewrite Equation 58 by making it only an exponent:

$$p(\mathbf{y}|\{w_{0\ldots M-1}, \boldsymbol{\mu}_{0\ldots M-1}, \boldsymbol{\Sigma}_{0\ldots M-1}\}) = \exp\left(-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu}) - \ln\left((2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}\right)\right) \quad (139)$$

If this was not a mixture of Gaussians it would be mathematically correct to have the natural log of the negative log likelihood cancel the exp function and be left only with the exponent. However we are dealing with mixtures and we cannot distribute a log over a summation as shown:

$$x = -\ln(\exp(-x)) \quad (140)$$

$$\sum_{i=0}^{N-1} x_i \neq -\ln\left(\sum_{i=0}^{N-1}\exp(-x_i)\right) \quad (141)$$

We realize that mathematically logs cannot distribution over sums. However, we will explore what happens to the derivative when this assumption is made and show a surprising result. To simplify our math, we will only look at one sample from our GMM loss and assume that the exponent of Equation 139 is:

$$\boldsymbol{e}_j(\mathbf{y}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \left(-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu}) - \ln\left((2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}\right)\right) \quad (142)$$

where j is the index of the mixture. With this substitution of the exponent GMM loss for one sample is:

$$\mathcal{L}_{GMM} = -\ln\left(\sum_{j=0}^{M-1} w_j \exp\left(\boldsymbol{e}_j\right)\right) \quad (143)$$

We will take the derivative of this loss function with respect to the arguments of $\boldsymbol{e}_j$ (namely $\boldsymbol{\mu}, \boldsymbol{\Sigma}$) but for simplicity we will write this as $\partial\boldsymbol{e}$. First, for reference we show the derivative of a single Gaussian mixture (see Appendix Section A for steps):

$$\frac{\partial\mathcal{L}_{GMM}}{\partial\boldsymbol{e}} = -\frac{\partial\boldsymbol{e}_0}{\partial\boldsymbol{e}} \quad (144)$$

where we have indexed the first and only Gaussian distribution at 0. Next, we show the derivative of a mixture of Gaussian distributions (see Appendix Section A for steps):

$$\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} = -\sum_{j=0}^{M-1} \text{softmax}\left(\boldsymbol{e}_j + \ln(w_j)\right) \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}} \tag{145}$$

From Equation 144 and 145 it becomes clear that when we added multiple distributions to make a mixture, the derivative applied a softmax weighting to the single Gaussian derivative from Equation 144. Note that when $M = 1$, Equation 145 reduces to Equation 144. Next, we assume that logs distribute over summations and take the derivative of the resulting loss function (see Appendix Section A for steps):

$$\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} \approx -\sum_{j=0}^{M-1} \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}} \tag{146}$$

By comparing Equations 145 and 146 we see that the difference between the two equations is the softmax weighting function. Thus, by assuming logarithms distribution over sums our derivative loses the softmax weighting. Next, we will investigate what the softmax weighting does when we use the proper loss function.

The softmax function increases relative differences between inputs by applying the exp function to each input then normalizing. Thus, relative differences between the inputs grow much larger. In our example the inputs are the exponents of a Gaussian distribution. The exponents are always negative in order to return valid probabilities between zero and one. The more negative this exponent values, the lower the probability will be for that distribution in the mixture.

In our GMM probability, the exponent will be more negative when the error between the target and mean is large. When taking the softmax of the exponent, large negative errors will result in smaller weights. The softmax function will then magnify this relative difference between weights so larger errors with have much smaller weights compared to the exponents with smaller errors. Thus, the softmax weighting will give more weight to distributions that have smaller errors and reduce weight from distribution with larger errors.

This weighting is directly proportional to the gradient term. The larger the gradient term the larger the impact on the weight update. Thus, distributions with larger error will have less impact on weight update compared to distributions with smaller error. This effect will make it more likely to further reduce error for distributions which already have less error.

By assuming that logarithms distribute over sums and thus removing the softmax, we no longer apply a softmax weighting. Without the softmax weighting, all errors will be weighted only by the magnitude of their error. Without weighting, there will be no runaway effect that attempt to further decrease error for already low error distributions. Thus, assuming logarithms distribution over sums may allow more time early in training for distribution to hone in on the correct mean instead of locking on to the means with the smallest error after initialization. This can lead to more accurate mean estimates because the optimizer can hone in on a better minimum instead of quickly falling into a bad minimum.

Covariance Accuracy

When MDNs determine parameters for a GMM there are two distinct evaluations to make. First is how accurate the actual estimate is. A common metric for this is simply RMSE between the truth and a point estimate. In our case, the point estimate is the maximum likelihood point of the distribution, which we assume is the mean of the mixture with the most weight. The second evaluation is to measure the accuracy of the covariance estimates. In our case the covariance estimate is the covariance of the mixture with the most weight. With this simplification, this section presents a simple method called Normalized Covariance Error to evaluate covariance accuracy.

Normalized Covariance Error Metric.

The metric we use to evaluate covariance accuracy is a custom single value metric called NCE. This method uses the covariance estimate for each point estimate to scale the error to what should be a series of normal distributions. We scale the error between the truth and the mean of the highest weighted Gaussian distribution by the covariance of the highest weighted Gaussian as shown:

$$\boldsymbol{e}_n = \boldsymbol{A}\left(\boldsymbol{\mu} - \mathbf{y}\right) \tag{147}$$

where $\boldsymbol{A}$ is the upper triangular decomposition of the inverse covariance matrix (see Section II) of the highest weighted Gaussian, $\boldsymbol{\mu}$ is the mean estimate of the highest weighted Gaussian, and $\mathbf{y}$ is the true target for this distribution. Thus, our normalized error $\boldsymbol{e}_n$ should be a vector of samples from a standard Gaussian distribution (zero mean, standard deviation of one, all dimension independent) if our covariance estimate was correct. To characterize how close that error is to a true Gaussian distribution, we calculate the empirical covariance matrix of the error across all samples as:

$$\boldsymbol{\Sigma}_{e_n} = \mathrm{E}[\boldsymbol{e}_n\boldsymbol{e}_n^T] - \mathrm{E}[\boldsymbol{e}_n]\mathrm{E}[\boldsymbol{e}_n^t] \tag{148}$$

where E[] denotes the expected value over all samples, and we perform outer products between $\boldsymbol{e}_n$ and its transpose to make a matrix. The final covariance matrix $\boldsymbol{\Sigma}_{e_n}$ should be close to an identity matrix if our error was truly from a standard Gaussian distribution. Our final metric of how accurate this covariance was across all distribution is the Frobenius Norm between $\boldsymbol{\Sigma}_{e_n}$ and the identity matrix:

$$\mathrm{NCE} = \sqrt{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|\Sigma_{e_n}[i,j] - I_N[i,j]\right|^2} \tag{149}$$

where our final normalized covariance error metric is NCE, N is the number of rows/columns in the covariance matrix $\boldsymbol{\Sigma}$, $\Sigma_{e_n}[i,j]$ is the i^{th} row and j^{th} column of the covariance matrix $\boldsymbol{\Sigma}$, and $I_N[i,j]$ is the identity matrix of size N at the i^{th} row and j^{th} column. When the NCE metric is closer to zero, the covariance estimate form the MDN is more accurate. This metric should only be used to compare distributions with the same number of output dimensions N, since as N becomes larger the Frobenius Norm will increase due to the dimensionality, even if the covariance accuracy is about the same. Also, since the metric is a single number it contains no information on how the covariance estimates deviated from a standard normal. A visual metric was explored but eventually not used for this dissertation because the visual metric was difficult to interpret without familiarity with advanced statistics. More information on the visual metric can be found in the Appendix Section A. Thus we have introduced a single value metric which can be used to compare the accuracy of covariance for MDNs with the same number of dimensions.

V. VLF Navigation

This chapter outlines how the methodologies and ANN architecture from Chapter III can be used to navigate with VLF signals. This chapter outlines specific implementations of how ANNs can be used for VLF navigation. First, the problem statement is given for VLF Navigation and the assumptions are outlined. Next, properties of VLF signals relevant to navigation are defined, and includes how ANNs can use this information to determine location. Next the two datasets are defined to include the input and output features of each dataset used for the different types of ANNs. After datasets are outlined, the ANN architecture and training methods are described. Finally, the results from the two different datasets are analyzed for different ANN filter implementations. The results include a section which explores the GMM loss methods from Chapter IV to determine which works best with the datasets. The filter implementations include filtering measurements from ANNs with classical methods, using ANN to replace propagate functions in the classical methods, and using an ANN as a complete filter.

Problem Statement

This chapter aims to perform navigation with VLF signals. Certain details of the VLF signals are assumed to be unknown a priori. In this problem the user does not know what is transmitting the signals, where the transmitters are, and what protocols the transmitters use. The area of navigation was mapped by recording empirical data which was used to train the ANNs. One continuous data recording is referred to as a data collect or a run. The VLF signals were recorded along with metadata such as odometry, pedometry, or accelerometer data. The navigation problem was restricted to 2D, where each position was constrained to a flat 2D plane. Orientation and height were not estimated. The following assumptions were used to constrain the problem further.

Assumptions.

This section lists the assumptions in this navigation problem:

- Signal Type – The signals of interest are in the frequency range from 0 to 22050 Hz. This comes from the sampling frequency of 44100 Hz used by the recording device which is an iPhone 4s or iPhone 5c. This frequency range is mostly the VLF range (3 kHz to 30 kHz) but it also includes other bands below 3kHz. The features used by models in this dissertation to characterize the VLF signals rely on the power of frequency bands determined by an FFT. Phase information is discarded.

- Transmitter Assumptions – It is assumed that enough distinguishable VLF transmitters exist in the environment that attempting to uniquely identify a location is possible. This is analogous in Global Navigation Satellite System (GNSS) navigation to having enough satellites visible to get a position solution. Transmitters are assumed to be distinguishable via either signal pattern, frequency, or power. The desired VLF signals are emitted by stationary predictable transmitters. Any unpredictable or moving transmitter is considered to be noise.

- Map Assumptions – The location estimates are performed on a 2D map with north and east as the x and y directions respectively. The axis has an origin at an arbitrary point inside the recorded measurement locations, usually in the center of the navigation area. This also assumes the area being traversed is generally flat and there is no significant elevation change over the map due to the curvature of the earth. This map may be discretized into location bins if continuous position is not available.

- Markov Models – All the ANN filter models used in this dissertation have a Markov assumption. Each filter and ANN requires only its current state to predict future states. This is important when the ANNs are recurrent which contain a state from time step to time step. These types of ANNs are assumed to contain all the relevant information in their internal state.

- ANN Types – The ANNs in this dissertation are all trained by a form of SGD through either back propagation or back propagation through time. No online learning algorithms are used. This allows all the computation in training an ANN to be done off line. The off line training does require much more computation compared to on line position estimation. However, off line resources can have less restrictions compared to on line position estimation. Thus, the ANN implementation can run on line on more mobile platforms that have resource constraints.

- Supervised Training – This dissertation outlines ANNs training for many different goals of output based on different types of input. All the ANNs in this dissertation use supervised training which requires known target output values. This means that for a general ANN function $\hat{\mathbf{y}} = \mathbf{f}^{NN}(\dots)$, the ANN is trained using true values of $\mathbf{y}$. This is an important restriction on the problems that can use ANNs. If the desired output of the ANN is not known by some truth source, then the ANN cannot be trained to predict those values.

- Truth Source – We assume that some high fidelity truth source is available to train the ANN for the desired output. For a 2D position problem, a high fidelity truth source is some source that gives accurate 2D position. The assumption is that the high fidelity truth source is expensive to use in an operational context due to some restrictions on the truth source such as size, power, weight, processing time, or availability. If the truth source could be used in an operational context, there would be no reason to estimate the truth source (ANN or otherwise). However the input arguments to the ANN can be used in an operational context, because they do not have the same restrictions as the truth source. Thus, the ANN can approximate the high fidelity truth source output with the readily available input arguments. If the truth source is not high fidelity and contains noisy errors, then it is assumed that enough samples are used to average out any noise on the measurements. This also assumes that the noise sources are zero mean in this case. If the noise sources are not zero mean, then the estimates from the ANN may contain a biased estimate of the truth.

Signal Fabric

This section describes a conceptual method of how to determine position from VLF measurements with a measurement ANN. Since it may be difficult to imagine how an ANN can determine position from noisy measurements this section provide a conceptual idea for how the ANN can output position given the noisy measurement input. This idea is only one possible way the ANN could derive position but may not be what the ANN actually does.

The method has two distinct parts. The first part creates a meaningful representation of the raw signal. This representation is called the signal fabric. The signal fabric represents a condensed version of the meaningful information contained in the raw signal. We denote the signal fabric $\mathbf{S_F}$ as the output of some conceptual transform $g(\mathbf{X})$ of the raw signal $\mathbf{X}$. Here, g then returns the signal fabric as $\mathbf{S_F} = g(\mathbf{X})$. Here, g is meant to be a conceptual function and does not have any actual realization. The function g distills the useful information from the raw input to derive a representation for the signal fabric. By distilling the useful information from the signal, g is robust to noisy and/or missing data. The function g is determined by training on the expected types of input signals but has no relation to the expected types of output. This allows g to contain general information for use in problems with different types of output. For example, the same signal fabric transform g could be used for different kinds of navigation problems such as 2D position, 3D position, heading, or orientation assuming the signal fabric provides observability to those states. However g cannot derive information from signals that were not included in the training.

The second part of this method uses the signal fabric to determine a position. The signal fabric space $\mathbf{S_F}$ is transformed to the position output space $\mathbf{Y}$ by some conceptual function f where $\mathbf{Y} = f(\mathbf{S_F})$. f learns how the information contained in the signal fabric relates to position. In general, the dimensionality of $\mathbf{S_F}$ must be equal to or larger then $\mathbf{Y}$ in order to cover all the possible output states unless the final output space lies on a smaller manifold space. f learns how to relate the signal fabric to positions through training samples. If training samples do not exist for certain configurations or positions then f can fail to map those positions even if the information about that position is contained in the signal fabric. f must be retrained on new samples to extend the positions which f can map to.

A concrete example of this method is navigation with cell towers. For example, g could be trained to create an intermediate signal fabric that represents the received power and ID to each unique cell tower. This type of function is useful where ever cell tower signals are present. With enough cell signals, g can learn to give a power and ID of each tower. The second function f could then learn a mapping from the signal fabric to position. In this example, the g function can be the same in all areas since the cell tower signals are the same. However, each new area of cell towers requires a new f function to learn how the expected power of each cell tower in that area relates to position. Thus, new samples must be collected in each new area in order to retrain f so it can map the signal fabric to positions. Note that f made no assumption on signal propagation which can allow for distortion effects such as multi-path to distort the power at each location.

Signal Fabric Properties.

If there did exist some concrete functions g and f, it would be useful to describe how properties of those functions would affect the signal fabric and final output. If the signal fabric to output location relation is turned around and the output is used to index possible signal fabric measurements, then we can derive information about the signal fabric in terms of changes in the output. In general the function f may not be invertible depending on the implementation. In this case a lookup table can be created for each of the training samples. This table would determine signal fabric from each output sample. If multiple output samples were identical the table could output a valid set of signal fabric measurements.

Once an inverse is created, an FFT can be taken for each dimension of the signal fabric with respect to each output dimension. This FFT transform shows how the signal fabric spatial frequency changes with respect to position. If there is high power in the higher frequencies then the signal fabric changes quickly with position. If the signal fabric has mostly low frequency content, then it is much smoother and changes very little with position. The frequency content can show how densely to sample a certain area. If the area is not sampled densely enough then the signal fabric many be aliasing and some high frequency content may have been lost. If the high frequency content is already near the Nyquist frequency then the samples must be resampled at a higher density in order to be sure that there is no aliasing occurring.

Another property based on the inverse is the derivative of the signal fabric with respect to the output. This is done by taking a derivative of the signal fabric with respect to the output. Now the signal fabric change in terms of the output can be examined to determine if the signal fabric change is constant or smooth. When the derivative is changing rapidly between samples, more samples may be required to describe the rapidly changing signal fabric. A rapidly changing signal fabric with respect to position or time requires more samples to describe accurately without missing changes in between samples points. When the signal fabric has many dimensions each dimension can be examined individually to determine how that dimension of the signal fabric changes.

Signal Fabric Implementation.

The functions g and f are conceptual functions that are never defined. However internally an ANN can perform similar functionality as g and f. The ANN can conceptually be broken into two sections which represent g and f. At some point the ANN distills the input signal into the signal fabric which contains the salient features of the input signal. After this point the ANN decodes that information into position. This divide was the reason for using recurrent and feed forward layers in the ANN. The concept was that similar functionality as g is performed by the stateful layers, and functionality similar to f be performed by the non-stateful layers. The exact output of the recurrent layers may not be the signal fabric which is a distilled representation of the signals. Since our ANN does determine position however, at some point internally the ANN does provide some idea of a signal fabric.

If we do assume the signal fabric is the output of the stateful layers, we can start to examine the properties of the signal fabric. In order to determine the properties of the signal fabric we can use the techniques described in the previous Section V. The indexing function f developed from the non-stateful layers then decodes the signal fabric map into a real position based on the training signal fabric. If new combinations of the salient features are given from a test set, the decoding function f may have unexpected results. This can happen when the raw signals are recorded from a new environment or when the true signal fabric created from the current transform g is not predictable from recording signals which are close in output space.

Temporal Stationarity.

Temporal stationarity of the VLF signal fabric is an important property for determining how well a given set of measurements reflects the future of the VLF signal fabric. The temporal stationarity of the VLF signal fabric is addressed in two parts, one for the transmitters and one for the environment.

Transmitters.

Temporal stationarity of the transmitters can be divided in short term and long term stationarity. Short term stationarity is how much the signal changes in a very short time frame like a few seconds or minutes. Short term stationarity is not required as long as the change can still be captured by the given measurements. In fact if many transmitters are on the same frequency it is more difficult to differentiate the transmitters if they are stationary in the short term. That is to say short term transmitter changes can be used to differentiate transmitters. Short term changes could have periodic patterns which can be identified by the ANN. Long term stationarity occurs over much longer time periods such as months or years. Long term changes can enhance or degrade our position solution based on if they are predictable or not. For example, an unpredictable change could be that some transmitter no longer exists. Periodic or systematic long term changes can be modeled if VLF measurements are recorded at the right times or if external metadata is provided. The VLF measurements and metadata can be used to develop a pattern to determine the changes of VLF signal fabric caused by these long term changes. For example, temperature is a long term change which can be measured and accounted for. Thus, predictable long term transmitter changes can be accounted for with external metadata input, but unpredictable changes always degrade location accuracy.

Environment.

The environment consists of obstructions like houses or vehicles, noise from transmitters whose signal cannot be used to predict location, and other properties of the environment such as time of day, temperature, or precipitation. For example, an unpredictably moving transmitter cannot be used to determine location. The environment warps the signal fabric from the transmitters by adding noise or changing signal propagation. Radiation from environmental objects like moving transmitters in other vehicles, or just ambient radiation from space all adds noise to the VLF signal fabric. The changes to the signal caused by the environment can be removed if they can be modeled. Models can take periodic inputs like season or day in order to determine how the environment changes the signal over time. For example diurnal changes can be roughly modeled if the time of day is known. Without a known time of day it may be difficult to generalize the affect diurnal changes have on the environment. Long term measurements can give insight to the long term changes caused by the environment. The model can learn to ignore short term changes caused, for example, by moving cars by obtaining samples when cars are not present.

Temporal Stationarity Conclusion.

The better a model can capture the effects the transmitters and environment have on the VLF signal fabric, the better the model can estimate the VLF signal fabric. This becomes more difficult when the transmitter and environment are non-stationary but extra measurement inputs can be used to model this non-stationarity. Even so some non-stationary effects cannot be modeled and degrade accuracy.

Scenarios

This section describes the two VLF navigation scenarios. This includes how and when the signals were collected.

AFIT Dataset.

Our AFIT indoor dataset consisted of runs collected at the Air Force Institute of Technology (AFIT) campus on Wright Patterson Air Force Base (WPAFB). These collects were performed indoors where GPS service was limited. Due to this restriction, there was no continuous GPS truth data available for supervised training. Instead, the campus was manually partitioned into 50 discrete physical locations. Each room was considered a location, as were hallway segments. A visual image of the discrete locations used is given in Figure 23. A unicycle antenna setup was created so the collection iPhone 5C could be held in one hand while the unicycle was driven with the other hand as shown in Figure 24. During collection, the iPhone 5C was used to manually tag times when the user was in each of the 50 different discrete locations inside AFIT.

Each collection covered most locations, and the collection was rarely temporarily paused. In small rooms, only the perimeter of the room was traversed, and for large rooms, the middle of the room was also traversed. Hallways, lobbies, and atria were collected at about the middle of the hallway in a straight line. In total, data was collected on 118 weekdays spanning the year of 2017 for all months except June and most of July. The first 67 collects from January to April (and one day in May) were evenly distributed to create 24 training sequences, 22 validation sequences, and 21 testing sequences. These collections were mutually exclusive datasets collected in a temporally-interleaved fashion. The next 51 collects were from May to December and were separated into seven extra test sets – one for each month from May to December (excluding June which had no collects). These test sets show for how long the training set from January to April can still obtain similar performance. For a detailed view of the number of runs for each set see Table 1.

The expected VLF sources from our collect include lights, electrical wiring, computers, monitors, HVAC equipment, and background VLF from outside the building. However, these sources may not stay constant across all days. For example, electrical equipment like computers and monitors may be on or off, the HVAC system may be running in either to heating or cooling mode, and physical objects may be moved altering how all the VLF signals interact. The ANN must learn what VLF sources in what bands consistently provide navigation information. While these are the expected VLF sources, the ANN may actually use only some of these VLF sources or other unexpected VLF sources to determine navigation information.

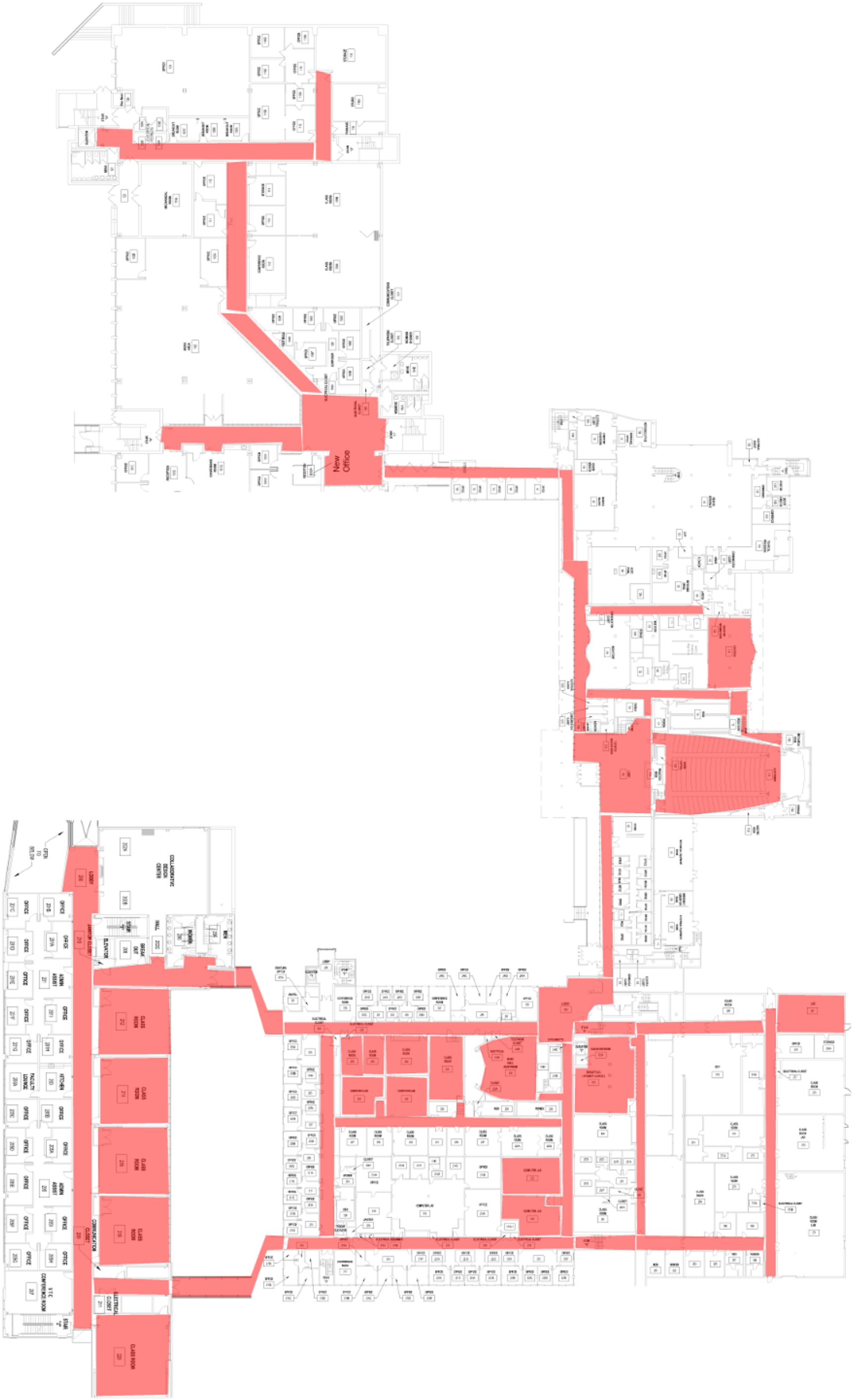

Figure 23: AFIT Dataset map image with named location regions shaded red. Only each of these red shaded regions can be used as an output location.

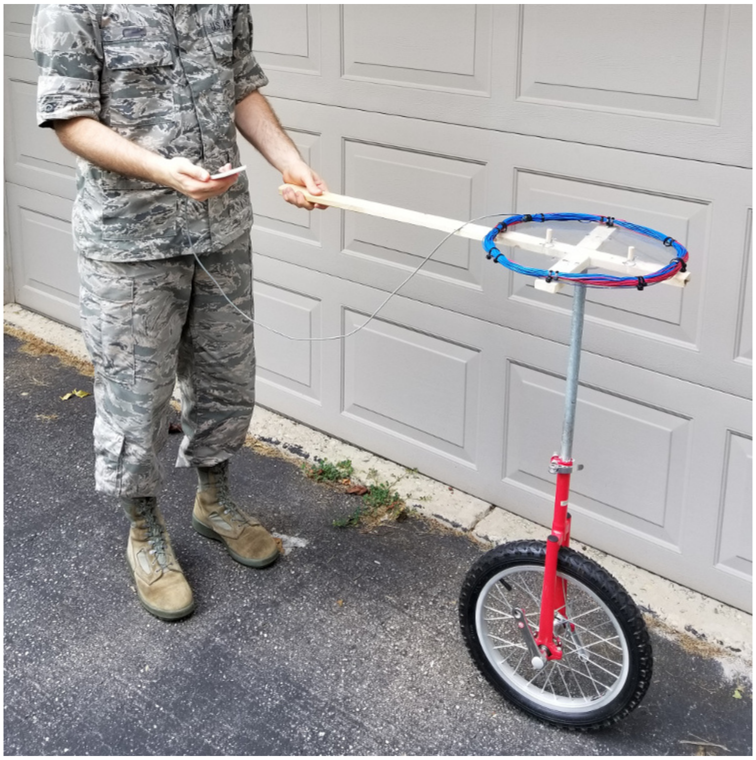

Figure 24: VLF Loop antenna mounted to a unicycle. Note the iPhone 5C is held in order to record locations manually.

Bike Dataset.

The bike dataset was an outdoor dataset. The VLF signals and metadata in Section V were collected in independent runs. A run was a continuous data collect where the bike started at a beginning point and returned to the point at the end of the run. Each run was about 40 minutes long and covered a suburban neighborhood in Beavercreek OH as shown in Figure 25. A total of 17 training runs and 13 validation runs were used to train the model. Of the 30 runs, 19 runs were collected on different days over a 40 day span of time in October and November 2016. The remaining 11 were collected on different days over an 80 day span from mid-August to the end of October 2017. The validation runs were interspersed withing the training runs, alternating between training runs and validation runs. The test set consisted of 5 runs interspersed with the collects in 2017. For a detailed view

Table 1: Number of runs in each of the AFIT datasets. Each run was about 45 minutes long. Note that hardware difficulties at the beginning of the data collection process prevented many early runs from containing metadata.

Dataset	No Metadata	Metadata
Jan-Apr Training	24	19
Jan-Apr Validation	22	19
Jan-Apr Test	21	17
May	6	6
July	2	2
August	9	9
September	9	9
October	9	9
November	9	9
December	7	7

of number of runs per set see Table 2. All runs were collected with the VLF loop antenna described in Section V attached the back of a bicycle as shown in Figure 26. All runs were collected mostly under normal bike motion, staying to the right side of the road and turning only at intersections. Each run made different turns at intersections resulting in different paths for each run, however the same roads were traversed in each training, validation, and test set.

Collected Data and Features

Our two different datasets contained some features that are similar and some that are unique to each set. This section outlines the different features of each set. In general the ANNs had different input features and target outputs depending on their purpose. The features form each dataset may be used as inputs to ANNs or as output targets. One input data source for both datasets consist of transformed time series VLF signals which we

Table 2: The number of data collects for the training validation and test sets split up by year. Each data collect was about 45 minutes long. Note that early collects encountered hardware problems that prevented metadata collection.

Set Name	No Metadata			Metadata		
	2016	2017	Total	2016	2017	Total
Train	10	7	17	10	6	16
Valid	9	4	13	5	4	9
Test	0	5	5	0	5	5

refer to as the VLF features. All other data sources are referred to as metadata. Metadata was collected at the same time as the VLF signals but consisted of many different kinds of features such as GPS location, accelerometer data, odometry data, and other data. Any combination of the VLF features and the metadata may be used as inputs to ANN or as target outputs depending on the goal of the ANN. Different output schemes were required depending on the target output type such as discrete or continuous target output types.

VLF Features.

This section outlines the VLF features used by the ANNs. This section first outlines the collection of the signals, then defines the processing required to turn the raw signals into features.

Loop Antenna.

VLF signal frequencies range from approximately 3kHz to 30kHz [40]. These frequencies afford a relatively compact loop antenna design. Additionally, because these frequencies subsume the human audio frequency range, a standard microphone input can be used to collect most of the bandwidth of the signals received by a VLF antenna.

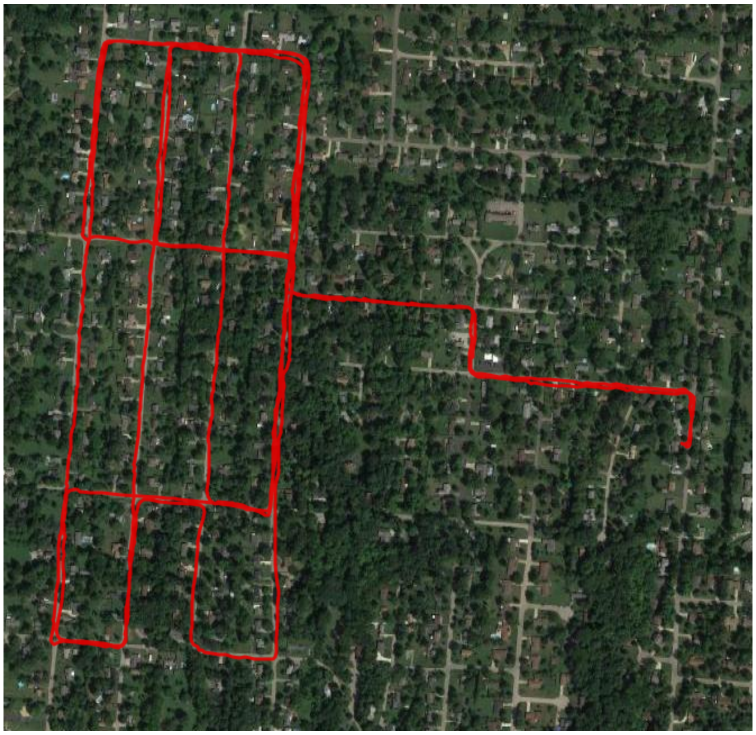

Figure 25: Area traversed by bike run. Note the suburban environment has houses, trees, and power lines near the roads.

The device for our loop antenna was designed by Dr. James Plesa [78] for our VLF navigation experiment as shown in Figure 27. The loop antenna has 41 windings and a diameter of 31.75cm. The loop antenna is attached to the 3.5mm Tip Ring Ring Sleeve (TRRS) microphone input jack of an Apple iPhone 5C for recording. No filtering or amplifying of the signal was performed at the antenna. The VLF signals of interest in this dissertation are in band with human speech and allow the iPhone microphone receiver to record VLF signals from the antenna. The VLF signals are sampled at a rate of 44.1 kHz and recorded in Waveform Audio File Format (WAV) files for later post processing.

A 1.5k resistor was put in series with the loop antenna to achieve the minimum resistance required for the iPhone to recognize a microphone input. The only calibration required was to determine the size of the resistor on the antenna so the iPhone recognized it

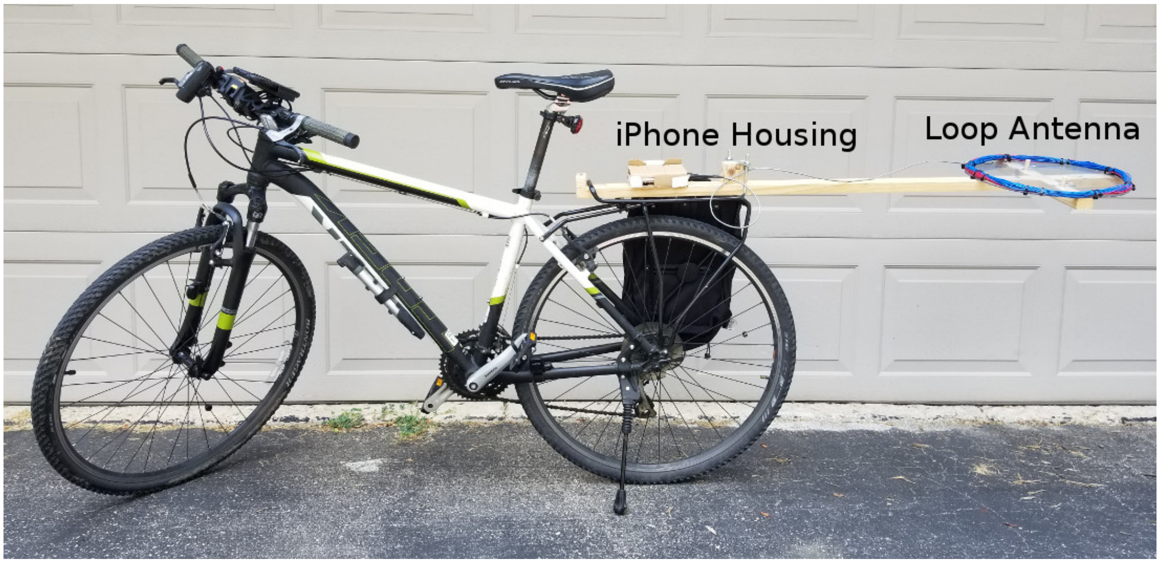

Figure 26: Bicycle with loop antenna attached to the back.

as a valid microphone source. The same antenna was used for all experiments, so anything that was particular to this antenna was present in the entire dataset. This prevented the models from using antenna induced characteristics to differentiate location. During signal collection the antenna loop was aligned with the horizontal plane of the ground.

Signal Preprocessing and Feature Generation.

To prepare the VLF signals for developing a navigation model, the raw signals were preprocessed into samples. The VLF signals were partitioned into 100ms time segments and several frequency-domain features were computed for each segment. Frequency-domain features for each segment were calculated based on a Fast Fourier Transform (FFT) of the signal in that segment. A sample consisted of the set of features of one time segment. Samples were labeled according to the antenna's position where they were collected. To summarize the process:

1. Partition collected signal into 100ms time segments.

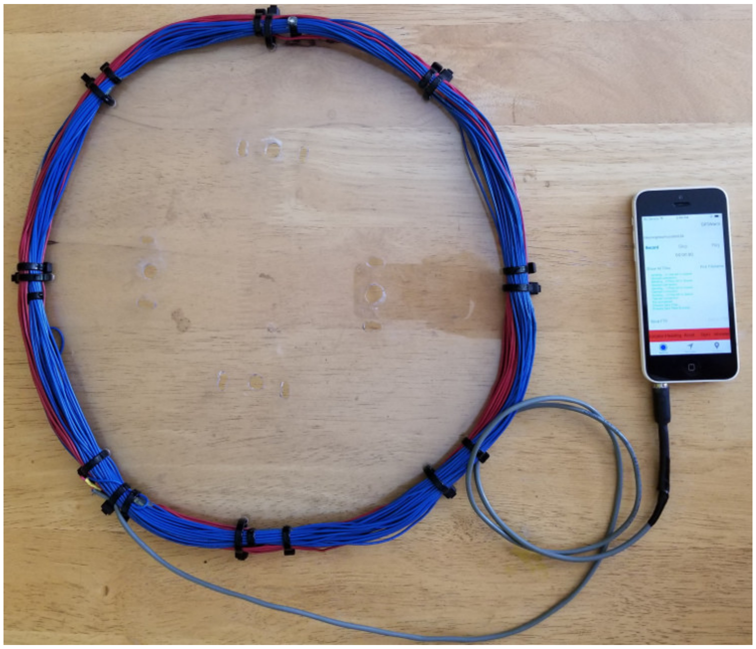

Figure 27: Loop antenna used for data collection. The design of the antenna is attributed to James Plesa [78].

2. Transform the signals from time-series to frequency domain using a Fast Fourier Transform (FFT) on each time segment to determine magnitude in each frequency bin from Equation 150.

3. Stack features from each segment into time-series sequences of frequency domain features.

This process created sample sequences that were used to train a model, and after training, to determine location. The samples were visualized using a tool called a spectrogram. The spectrogram showed how the power in each frequency changes over time. A spectrogram was displayed as a heat map of the signal's spectral power in each frequency bin (vertical axis) over time (horizontal axis), as shown in Figure 28. Note that the features used in for the ANN model are magnitude values while a spectrogram shows

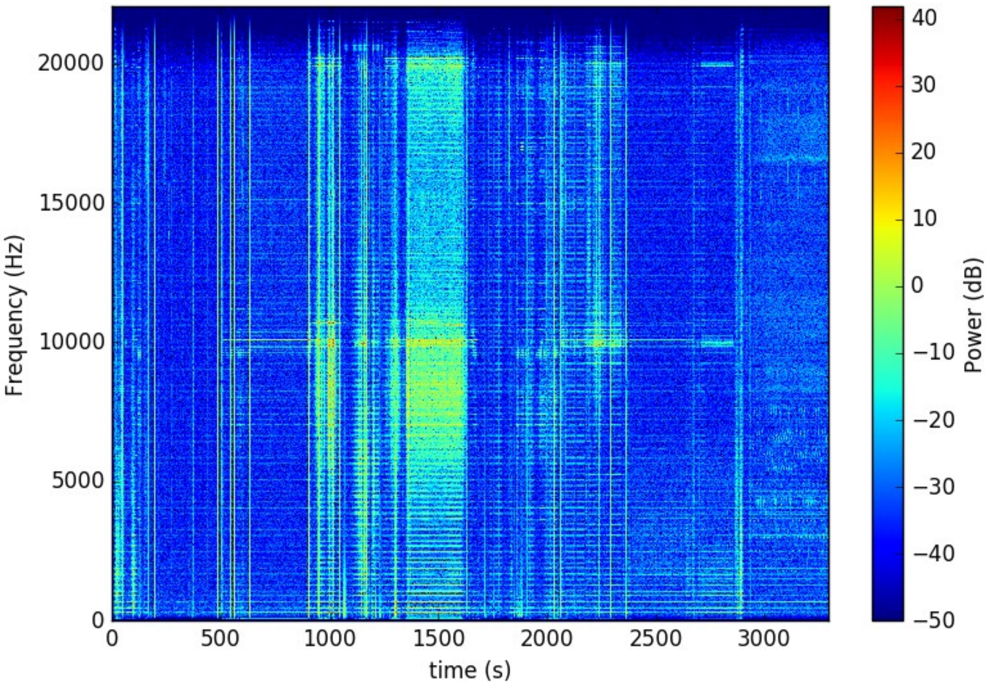

Figure 28: Spectrogram of one data collect. The vertical axis shows power in each of the frequency bins and horizontal axis is time discretized to the segments. Thus, each column is the result of one FFT transform of one time segment and then the time segments are stacked horizontally over time to make the spectrogram.

power, which is simply magnitude squared. Next we describe how each of the spectral magnitude values are computed.

The following equation describes the spectral magnitude features for one time segment:

$$x_k^{t_w} = \left| \sum_{n=p_w t_w - N/2}^{p_w t_w + N/2 - 1} x_n \exp\left(\frac{-i2\pi kn}{N}\right) \right| \qquad \underset{0<=f_l<f_u<N}{k = f_l, \ldots, f_u} \tag{150}$$

where $x_k^{t_w}$ is the magnitude of the frequency bin k for time segment t_w. N is the number of amplitude samples to use for each time segment, which will also determine the segment length when the sampling frequency is known. The scalar x_n is the amplitude value for the n^{th} sample in the VLF signal. The variable p_w is the number of amplitude samples between each segment center. The index of the time segment is t_w. Each time segment was positioned so that amplitude sample number $p_w \times t_w$ was the center sample in the time segment t_w. Thus if $p_w = N$, then the second half of one time segment would overlap with the first half of the next segment. For example, assume time segments contain one second of amplitude samples, and there is one second between time segment centers. In this example, the first one second time segment is centered on the sample at 0.5 seconds, and the next one second time segment is centered on the amplitude sample at 1.0 seconds. This means that the time segments overlap from 0.5 to 1.0 seconds. However our features ended up not using any overlap between time windows. The two variables f_l, f_u set the lower and upper limit for the frequency bounds to be included in the features. Our experiment required to reduce the overall features due to memory and computation limitations. Our VLF features had an upper bound of $f_u = 11.025$ kHz and a lower bound of $f_l = 0$kHz.

One additional data preparation step was taken prior to training. Many machine learning models (including ANNs) are sensitive to differences in the scales of features. Several techniques were used to reduce these differences. Before any samples were used for training they were transformed using the PCA algorithm [43, 73, 109]. PCA is the decomposition of multidimensional data into components, sorted by variability. Our approach also adds whitening to the PCA algorithm which scales all the variances to one. PCA is commonly used to reduce the dimensionality of the data by removing dimensions with low variability [43]. However, our method used all the dimensions because results were adequate without removing dimensions. PCA was used because the models in this experiment trained much faster when PCA was used compared to when PCA was not used.

Metadata Features.

Metadata was collected at the same time as the VLF signals. Two cell phones were used to record the metadata. An iPhone recorded the following metadata: accelerometer (Δv), gyro ($\Delta\theta$), magnetometer heading, true heading (correction of magnetic heading), GPS location, and location name string. A second android cell phone recorded metadata related to workout data. This data included heart rate, bicycle wheel odometer data, bicycle crankshaft odometer data, and pedometer step data. Time synchronization between the two devices was accomplished with cell tower timing protocols. All of this data was time tagged but the update rates for each data stream were not the same. For example, the location name was updated when the location was changed, but wheel odometry data was recorded every time the wheel made a full revolution. Also some data was generated based on the recording time. This metadata included the percent through the current day and a scalar indicating the current season. The current season scalar was determined from a sine curve of unit amplitude that peaked at the summer solstice and had a low at the winter solstice. This allowed datasets collected over a full year to understand seasonal effects. A summary of the different types of metadata used and when they are available is given in Table 3.

Using many different types of update rates created a problem when trying to use all the possible features as input for an ANN. The ANN expected all input at once and did not allow for only partial data. This required all data to be aligned at the same rate. Thus, all of this metadata was further processed for use in the model when the time steps do not match up. All of the metadata was processed to align with the center of the VLF features time segments. The processing gave a unified vector of input for the ANN to process.

Different processing was required for different types of metadata. All the continuous absolute data sources were interpolated between time steps to the center of the time segment. The interpolation function between time steps was linear. For example, GPS position was interpolated to the center position for the time segment.

Incremental measurements were additive and summed up from the beginning of the time segment to the end, and only the time segment edge values were interpolated. For example, the accelerometer Δv is summed up over the entire time segment to evenly distribute the changes in acceleration. For example, if 100 evenly spaced accelerometer

Table 3: Metadata Types. Table showing the different types of metadata used and the properties of each metadata variable.

Metadata Name	Data Type	Synchronization Method	Available
Accelerometer Δv	$\mathbb{R}^3[x, y, z]$	Summation	All
Gyro $\Delta\theta$	$\mathbb{R}^3[\theta_x, \theta_y, \theta_z]$	Summation	All
Magnetometer Heading	$\mathbb{R}[\phi]$	Interpolation	All
True Heading	$\mathbb{R}[\phi]$	Interpolation	All
GPS Location	$\mathbb{R}^3[lat, lon, alt]$	Interpolation	All
Location Name String	String	Nearest	All
Heart Rate	$\mathbb{R}[h_r]$	Interpolation	All
Pedometer walking speed	$\mathbb{R}[s_p]$	Interpolation	Walking Only
Bicycle Wheel speed	$\mathbb{R}[s_w]$	Interpolation	Bike Only
Bicycle Crankshaft speed	$\mathbb{R}[s_c]$	Interpolation	Bike Only
Percent Through Day	$\mathbb{R}[t_h]$	Generated	All
Season Value	$\mathbb{R}[t_s]$	Generated	All

measurements were taken during the time segments, the value for the time segment is the sum of all the acceleration measurements since the last center of the time segment.

Discrete measurements, such as the location name string, were chosen by the discrete value that occurred at the center of the time segment. If the discrete value was a string, the string was changed to one hot representation across all possible strings for input to ANNs. For example if there were 10 possible string values, a vector of size 10 with all zeros except a single one, would represent that single string. If the location name changes during a time segment, the value at the center of the time segment is used for the entire time segment.

The generated time related measurements were calculated at the center of each time segment and required no further processing.

All metadata was normalized before given to any ANNs. The normalization determined a bias and scaling factor to make the metadata zero mean and unit variance for each metadata feature. Unless otherwise stated, the transform is determined using only training data and never validation or test data.

Certain metadata were only available for certain data collects. For example, bicycle wheel odometer data was only available for data collects on a bicycle. Pedometer step data was only available for walking data collects. Also, some collects did not contain metadata due to hardware malfunctions but still contained VLF signals. These missing metadata sets are included in training/validation/testing for non-metadata datasets but are not included for datasets which require metadata. For a detailed view of the number of sets included in each experiment see Tables 1 and 2.

Input Features

The VLF features and the metadata features were available as input features for the ANNs. In general the entire feature set was available for ANN input. However, depending on the purpose of ANN, only certain features may be desired as input for the ANN. For example, the state predict ANN, outlined in Section III, was given only previous state (position) as inputs. After a set of features was determined, the features were concatenated together into one input vector.

Output Features

Output features for many ANNs were some kind of position target. However, the output features differed depending on the dataset. The bike dataset generally had accurate GPS data for truth. The GPS location data provided more options for output feature types. The AFIT Dataset was collected inside buildings where GPS accuracy was poor and thus limiting options for output features. The full set of output features were broken into two types–discrete locations and continuous locations. The bike dataset with high GPS accuracy was the only one that supported both continuous output locations and discrete locations. To obtain discrete locations, continuous locations were binned into discrete locations. However, the AFIT Dataset with poor GPS accuracy only had discrete locations based on the name of the area while recording. The discrete locations were manually recorded during collection. Thus, while the AFIT dataset had discrete output features, it did not have continuous locations because there was no underlying continuous locations. This section outlines the different types of output features used in both datasets.

Local Level Coordinates.

The continuous output features used the latitude and longitude from the GPS metadata to determine location. The GPS latitude and longitude measurements were converted to a local level frame for use as output features. To convert to the local level frame, first the GPS measurement was converted to the GPS Earth Centered Earth Fixed (ECEF) frame. Next, the origin was moved to a fixed position in the middle of the collect which was the same for all datasets in the same area. Finally, the coordinate frame was rotated to make the x direction point north, the y direction point east, and the z direction point down. This created a flat local frame on the surface of the earth near the area of interest that was only valid for the local region due to the curvature of the earth. This local level coordinate system served as the basis for the continuous output features and different types of discrete features. An example of this kind of local level output is shown in Figure 29.

The continuous local level output was biased and scaled to have zero mean and a standard deviation of one like all metadata features. The scaling allowed the ANN in this research to train more quickly. While the actual values given to the ANN were the normalized data, all figures and graphs were converted back to unscaled versions for

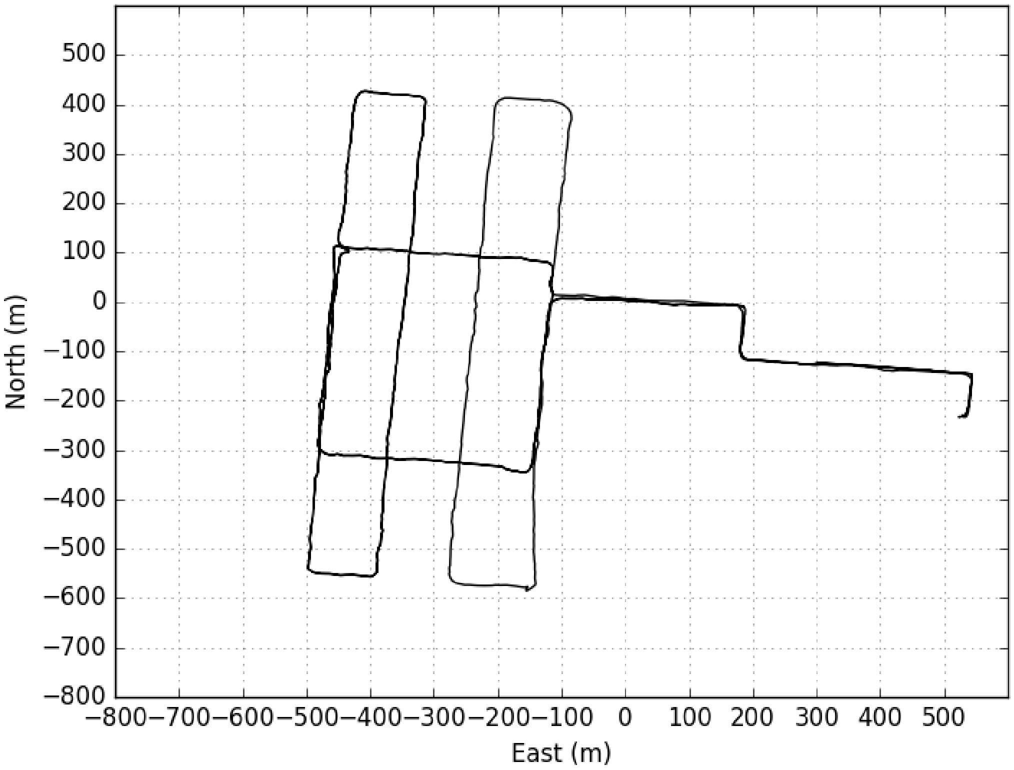

Figure 29: Example of one run with local level coordinates. This is the transformed local level coordinates in meters.

readability. The local level coordinate output can be used as the target output to train regression models as well as probability parameter models such as MDNs.

Discrete Output.

The discrete output locations determine a probability of a location bin. The discrete outputs based on the local level frame were binned by the clustering method described in Section II. The 100 center points of each bin are shown in Figure 30. Each continuous location on the map is classified as the bin whose center is closest to the given point. The clustering algorithm used only the positions from the training dataset in order to determine the center points.

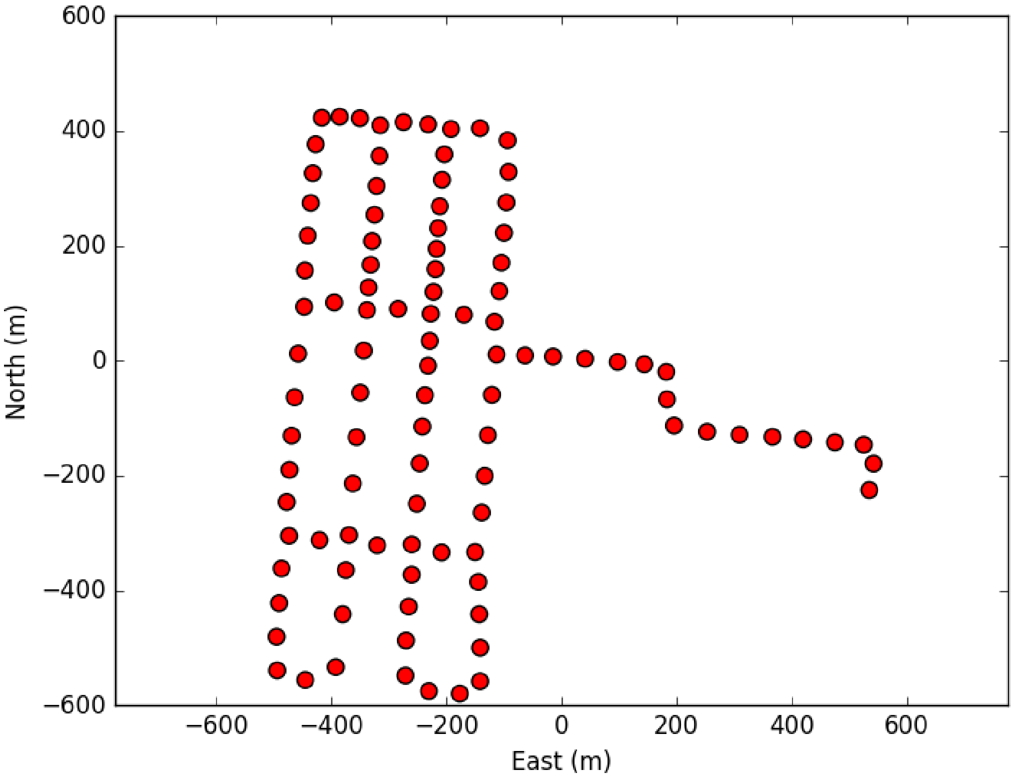

Figure 30: Classification region center points (red circles) determined by a k-Means clustering algorithm with 100 clusters.

The discrete output locations not based on the local level frame were the names of the location from the AFIT dataset. The names of the locations were manually recorded during the collect. The total number of locations for each collect was the total possible locations. The total number of locations for each collect did not change even if some locations were not visited during the collect. In this dissertation, the locations names were regions of a floor plan that were selected for recording. An example of how the regions were partitioned is shown in Figure 23.

Each discrete method had the same style final target output no matter what he underlying classes represent. The final target output data was a binary vector consisting of zeros and ones. The target vector had only one non-zero value of 1 at the index of the correct class. Thus, the target one hot vector was a valid probability distribution with all the probability at the correct class. When the ANN attempted to predict the probability of the input, the ANN tried to match the target one hot probability vector as closely as possible. Thus, the ANN assigned probability to classes in an attempt to match the target probability vector. Using this probability distribution, many different metrics can be used to reject outliers or provide confidence in a position estimate.

Classification Weighted Regression Output.

This section outlines a method to obtain interpolated continuous output from a discrete classification output ANN. This form was motivated because the pure regression output models had been shown by experiments in this dissertation to have poor MSE even when compared to classification. The binned locations had better Root Mean Square (RMS) error compared to regression even while suffering from the loss in precision by binning locations. Next we show how to interpolate between bin centers to improve this RMS error further.

To interpolate the discrete classification, first an ANN was trained to output classes based on locations with known coordinates. Next, an MLP layer with two perceptrons (one for each axis x, y) was added to the end of the pre-trained classification ANN. This MLP layer had its weights initialized to be the center coordinates of the output class labels used to train the ANN. This MLP layer now calculated a weighted average of all the output positions based on probability. The weighted average resulted in smoothly transitioning between classes which reduced error when locations were on the edges of the classes. This combined ANN was also be further trained to fine tune the pre-trained weights and new final output layer to further optimize the output.

Probability Model Output.

The probability model outlined in Section II was used to provide a continuous output while also providing a probability distribution of the position. For this output type, the

ANN determined the parameters of a probability model. The ANN was trained to increase the probability of the correct output position. A probability model provided a probability over the entire output space instead of being required to output only one position. This dissertation used a GMM as the probability model. 2D multivariate Gaussian distributions were used for the 2D position problem. Thus, each Gaussian distribution in the Gaussian Mixture Model had two dimensions as the output space for the 2D position estimate. Unless otherwise stated, the GMM loss functions used the upper triangular $\boldsymbol{A}$ form of the inverse covariance matrix to estimate covariance for GMM. Also the GMM loss functions estimated seven Gaussian distributions per mixture.

Base ANN Model

Now that the problem, inputs, and outputs have been outlined, the actual architecture of the base ANN model is detailed. The ANN given is a concrete form for ANN blocks that represented many different functions in Chapter III such as measurement ANNs, state predict ANNs, complete filter ANNs, and others. The inputs and outputs to each ANN were tailored to the goal of the ANN as stated in Section V. The specific purpose of the ANN determined what inputs are given to the ANN and what the target output type was.

The base ANN model used in this dissertation was an ANN with LSTM and MLP layers. A graphical example image of the model is shown in Figure 31. The parameters (such as number of layers) of the ANN were chosen to reduce the loss on the validation sets. These were not necessarily the optimal parameters but produced favorable results for different ANN goals. The ANN started with two LSTM layers of size 500. The LSTM activation function was a tanh and the inner gate function was a hard sigmoid function. Each of the weight matrices in the LSTM layers had a dropout probability of 0.2 during training. Below the LSTM layers were 21 MLP layers each with a size of 50 hidden units. The MLP layers had a tanh activation function. After each MLP layer there was a layer of

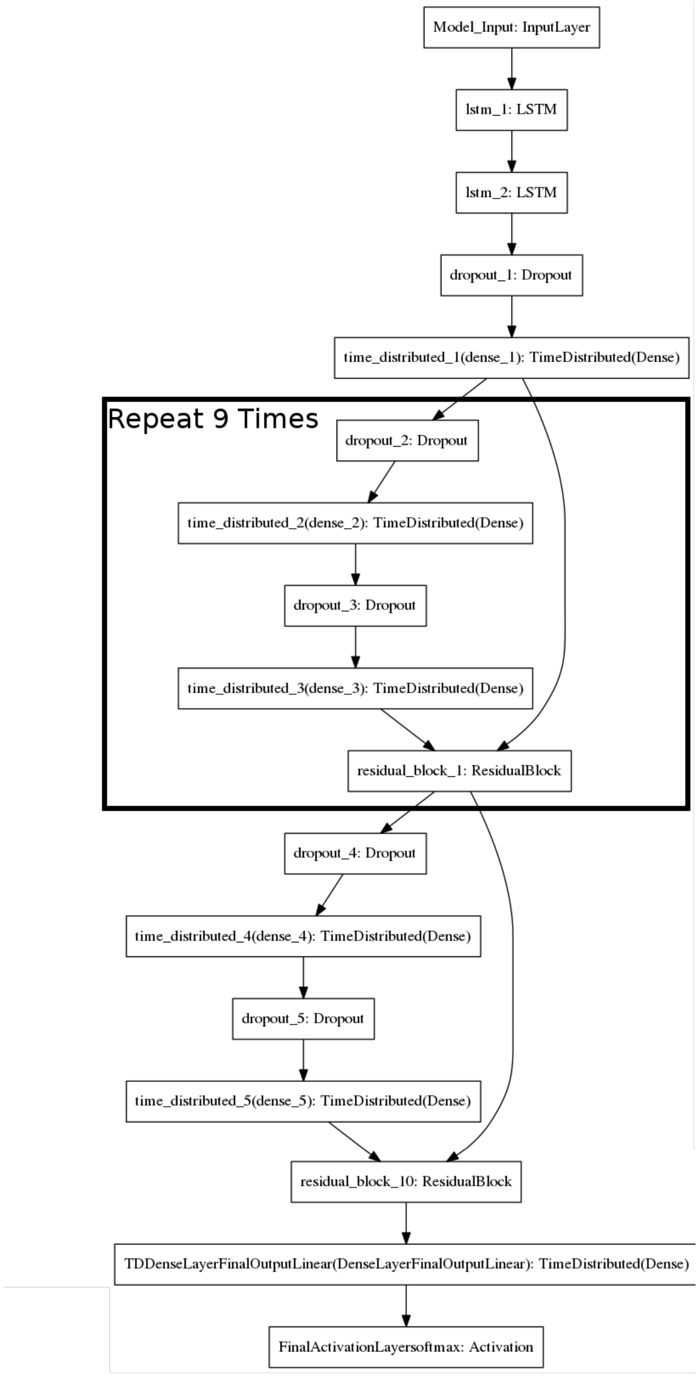
Model_Input: InputLayer
lstm_1: LSTM
lstm_2: LSTM
dropout_1: Dropout
time_distributed_1(dense_1): TimeDistributed(Dense)
Repeat 9 Times
dropout_2: Dropout
time_distributed_2(dense_2): TimeDistributed(Dense)
dropout_3: Dropout
time_distributed_3(dense_3): TimeDistributed(Dense)
residual_block_1: ResidualBlock
dropout_4: Dropout
time_distributed_4(dense_4): TimeDistributed(Dense)
dropout_5: Dropout
time_distributed_5(dense_5): TimeDistributed(Dense)
residual_block_10: ResidualBlock
TDDenseLayerFinalOutputLinear(DenseLayerFinalOutputLinear): TimeDistributed(Dense)
FinalActivationLayersoftmax: Activation

Figure 31: Graphical Image of the model used to implement the approach of this dissertation. Information flows from top to bottom. The input and output layer sizes are changed depending on the dataset.

dropout with a probability of 0.25 during training. Between every two MLP layers there was a residual connection from the first layer to the second layer as defined in Section II. Only the first MLP layer had no residual connection. This first MLP layer resized the output size of the LSTM layers to the MLP layers. The resizing allowed each residual connection to be added element wise with no padding or resizing.

The final layer depended on the type of desired output. If the output is discrete classification, as outlined in Section V, then the last layer has as many outputs as there are output classes. A softmax activation is performed on this final layer to normalize the output values to a probability distribution. The loss function used for discrete classification is categorical cross entropy.

If the final layer was continuous output, as outlined in Section V, a two node dense layer with linear activation was used, where each node represented the north and east estimate. The loss function used for this type of output was MSE loss.

The final layer for a probability model output, as outlined in Section V, was a dense layer with each output representing parameters for the probability model. For probability model output, the number of outputs was determined by the representation of the covariance matrix and number of mixtures in the model. Also, a specific activation function tailored to the representation of the covariance matrix was required to make a valid probability parameter output.

Our base ANN has a general equation:

$$\hat{\mathbf{y}}_k = \mathrm{a}^{NN}(\mathbf{a}_k, \mathbf{b}_k, \ldots, \mathbf{h}^{NN}_{k-1} | \boldsymbol{\theta}^A) \tag{151}$$

where $\mathrm{a}^{NN}()$ is the function for the ANN, and represents the architecture of the ANN. The ANN function can take multiple input vectors $\mathbf{a}_k, \mathbf{b}_k, \ldots$ which are concatenated together as one input vector. $\mathbf{h}^{NN}_{k-1}$ is the previous state of the ANN if a recurrent ANN was used. Also, if the ANN contained state (making the order of the inputs is important), the subscript k refers to the time step of that vector. $\boldsymbol{\theta}^A$ represents the parameters of the ANN, for example the weights and biases of each layer. The number of parameters $\boldsymbol{\theta}^A$ depended on the architecture of the ANN. The given parameters must match the expected number of parameters according to the architecture of the ANN. The output of the ANN is $\hat{\mathbf{y}}_k$. The ANN output is the estimate of the target output, which depends on the type of output desired.

The base ANN also had a slightly different form when the ANN received inputs which were functions of the ANN output. The simplest case was when the output of the last time step was directly used as input. The equation for this type of base ANN is:

$$\hat{\mathbf{y}}_k = \mathrm{a}^{NN}(\mathbf{a}_k, \mathbf{b}_k, \ldots, \hat{\mathbf{y}}_{k-1}, \mathbf{h}^{NN}_{k-1} | \boldsymbol{\theta}^A) \tag{152}$$

The difference between the ANN in Equation 151 and this ANN is that the latter uses its own output $\hat{\mathbf{y}}_{k-1}$ as input. By feeding the output back into the ANN as input, the ANN can learn dynamics of the problem, similar to a state predict function in a standard filter. This approach is useful if for example the purpose of the ANN is to replace a state prediction block in a filter. Besides these considerations, the ANN had the same architecture as the ANN shown in Figure 31.

ANN Training.

The training parameters were chosen to minimize the loss of the validation sets. These parameters (such as learning rate) were not optimized for a specific problem. Rather, they were chosen to work well for many different types of problems. Training was done in batches across all training runs. Each batch contained 64 sample sequences selected randomly from all possible runs. Each sample sequence contained 100 time steps. The weights of the model were updated after each batch. The total number of batches was determined by the total possible sample sequences. Instead of training on all possible sample sequences for each epoch, only as fraction of the total was used for each epoch. We used one hundredth of the total possible samples before claiming an epoch was complete. Using this method, it was possible to repeat overlapping sequences even in a single batch.

However, due to the large number of possible sequences (hundreds of thousands) this repetition was unlikely and did not pose any over fitting problems.

The LSTM layer input and previous output weight matrix had an L1 regularization value of 0.0001. The MLP weight matrices also had an L1 regularization value of 0.0001. The overall learning rate was 0.0001. During training, the learning rate was reduced by a factor of 0.8 if after 10 epochs the loss function was never reduced. The training algorithm then waited at least 10 more epochs before reducing the learning rate again. The number of training epochs was 500 for each model unless otherwise specified.

Also, if the output was fed back as an input, as shown in Equation 152, the previous outputs were precomputed for each batch, since the samples were selected randomly. This was important to provide a realistic training environment where the ANN learned how to use its own output as input. Using true output as input resulted in over fitting to the unrealistically good input and prevented learning real relationships from the rest of the input.

For evaluation purposes the sequences were constructed slightly differently. Unlike training which used small subsequences and started with a fresh ANN state for each subsequence, evaluation was performed on sequences consisting of an entire run. This allowed the internal state of the ANN to maintain the state for the entire run and improved performance compared to the training scores.

AFIT Dataset

This section develops implementations of the ANN and filter integration framework from Chapter III for the AFIT Dataset. First, this section describes how measurement ANNs were trained to output discrete location. Next, the measurement ANNs were integrated into a discrete particle filter. Next, another ANN was trained to predict discrete location based on previous discrete location, called a state predict ANN. Then the state predict ANN was integrated into the discrete particle filter with the measurement ANNs.

Finally, complete filter ANNs were trained to use previous discrete location information as well as measurements to predict the next discrete location.

AFIT Dataset Measurement ANN.

This section analyzes the results of training measurement ANNs using the architecture outlined in Section V trained on the dataset described in Section V. Two measurement ANNs were trained to predict discrete location, one using the VLF signals only and another using VLF signals and Metadata. The metadata included accelerometer, gyroscope, magnetic heading, true heading, heart rate, walking speed according to step tracking, percent through day, and season sine wave from Table 3. Each of the training, validation, and test datasets were evaluated with the final trained models to obtain a set of predicted locations the antenna traversed. The ANNs generated location probability predictions for each dataset. The accuracy ($\frac{correct\ predictions}{all\ predictions}$) of each dataset is shown in the Table 4 along with 95% confidence intervals [44] around the accuracies. The confidence intervals represented the chance that if we sampled from the same distribution as the given data, we would have a sample accuracy inside the error bars 95% of the time.

In addition to the confidence intervals, we calculated the Cohen's Kappa value for each set. The Cohen's Kappa value represents how far from "guessing" the classifier performed [44]. For example, if 90% of the samples came from one class, the classifier could achieve 90% accuracy by only guessing that class. Cohen's Kappa value takes class distribution into account and provides a metric to show how close to guessing based on the class distribution our classifier preformed. Values near and below 0 are considered very poor and indicate that the classifier is not performing much better than guessing. Values above 0.4 are considered to be better and indicate that it is unlikely the classifier is guessing. In our case all the Cohen's Kappa statistics were above 0.4, thus our classifier was most likely not guessing from the class distribution as shown in Table 4. This property shows that our

Table 4: Performance comparison of the AFIT Map measurement ANN without metadata and with metadata.

No Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	93.6%	0.934	2.47%
Jan-Apr Validation	76.6%	0.760	2.06%
Jan-Apr Test	79.6%	0.790	2.59%
May	75.2%	0.746	4.43%
July	64.3%	0.633	9.41%
August	70.2%	0.694	4.41%
September	66.1%	0.651	4.07%
October	68.6%	0.678	4.49%
November	61.5%	0.605	4.47%
December	58.6%	0.575	4.96%

Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	95.0%	0.948	2.72%
Jan-Apr Validation	78.7%	0.782	2.77%
Jan-Apr Test	81.5%	0.810	2.86%
May	75.1%	0.744	5.51%
July	63.3%	0.674	9.41%
August	72.5%	0.718	4.41%
September	71.2%	0.703	4.07%
October	69.9%	0.692	4.49%
November	59.1%	0.580	4.47%
December	57.5%	0.564	4.96%

classifier obtained position information from the VLF signals and was not simply guessing a position.

The models achieved high accuracy on the training and validation sets. For example, the metadata model achieved a validation accuracy of 78.7% with a 95% confidence interval of 2.77%. However we must look at the test sets to draw any conclusion about accuracy since we may have over fit our model to the training and validation datasets. We next evaluate the model on the sequestered test sets as described in the next section.

AFIT Dataset Measurement ANN Test Results.

Eight separate sequestered test sets that were not used for weight training or hyper parameter selection were evaluated against our trained measurement ANN. To evaluate test data collected in the same time frame as the training and validation data, the first test set was interleaved with the training and validation sets (while ensuring the three sets contained mutually exclusive data collection sequences). On this test set, the metadata model obtained an accuracy of 81.6% with a 95% confidence interval of 2.86%. However, our confidence interval only applied to data with the same distribution as this test set. Future sets may vary slightly due to changes in the VLF signals over time.

To shed some light on the stability of model's performance over time, seven other test sets were collected after the three interspersed sets. These test sets consisted of runs that were collected from May 2017 to December 2017. Since the the training and validation sets were collected from January to April 2017, this final test sets shared no temporal overlap with the training or validation data. Thus, the final test sets allowed an evaluation of the model's resilience to changes in the VLF signals months after originally VLF-mapping the navigation area. As shown in Table 4 and Figure 32, the accuracy of the model on the final test sets ranged from 75.2% to 57.5%.

Now we explore the accuracy of the test sets over time using Figure 32 to see if there is any trend. The accuracy of each collect is expected to vary slightly from day to day due to

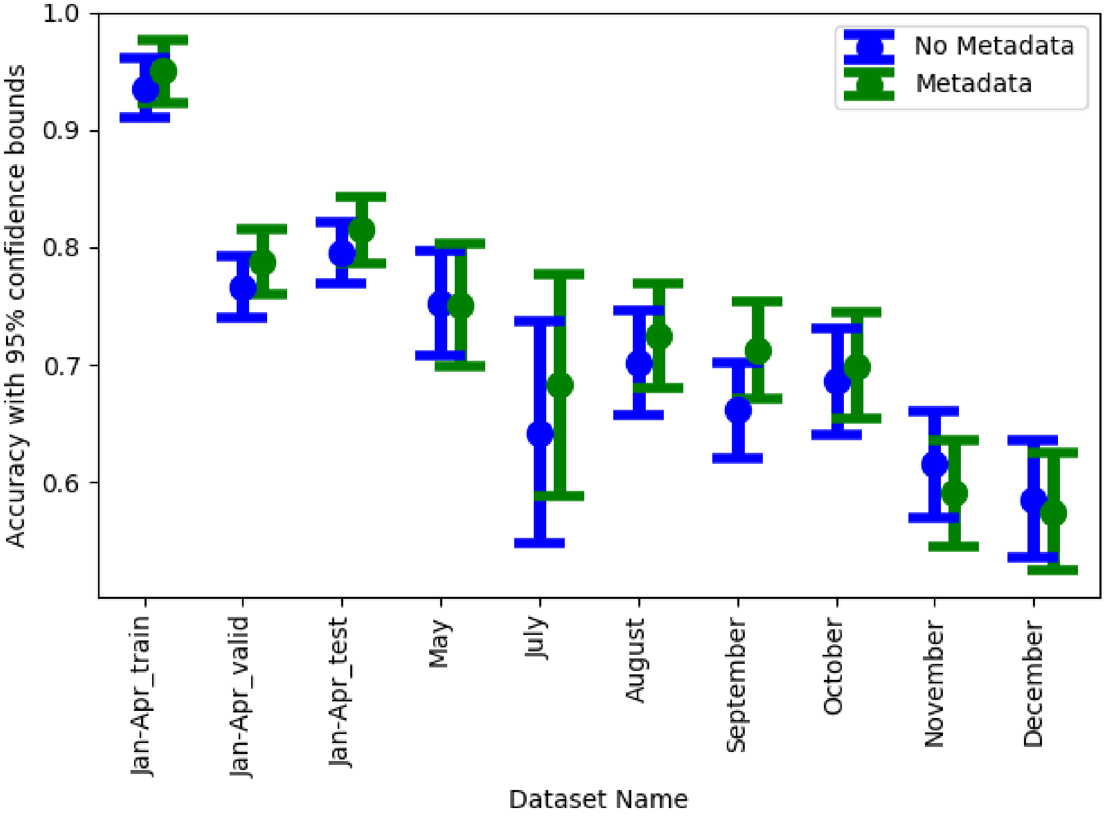

Figure 32: Accuracy of AFIT dataset with no metadata results compared to accuracy of AFIT dataset with metadata. 95% confidence bars are included for all accuracy estimates. Note the July confidence bound is larger due to only 2 runs in July.

changes in the VLF sources in the environment. For example, some electric machinery, like the HVAC system, may only turn on during certain times the day. However, over time our accuracy may degrade due to permanent changes in the environment such as adding new VLF sources or removing existing VLF sources. Our test sets showed that some changes likely did occur over time, since the later sets performed worse compared to the earlier sets. Specifically there was a large drop in November which continued to December. This drop could be due to some permanent change in AFIT that caused previous VLF signals to no longer match the new environment. However, considering this time period was months after the training validation data, the performance is still quite high. Using more recent data for training could allow the ANN to learn the changes over time and improve performance. Also, the performance for July was worse compared to the neighboring months. However, July contained only two runs which increased its confidence bounds to well within the performance of the neighboring months. Thus, the performance may actually have been comparable to the neighboring months if more runs were collected and the confidence bounds were decreased.

Another facet of the results to analyze is the performance of each location region. While the average accuracy was 81.5% in the metadata model interspersed Jan-Apr test set, some classes had much lower accuracy. For example, in Figure 33, where the average accuracy of each class is color coded, it becomes apparent some classes had very poor accuracy. Specifically, a row of rooms on the bottom right had very bad accuracy (well below 50%). Analysis of the confusion matrix showed that these four rooms were actually confused with each other. This may be due to the fact that the four rooms on the bottom have the same layout. Each of these locations regions was a classroom with identical computer and furniture setups. This similarity may make it difficult to distinguish between rooms using VLF features with no additional information. Thus, it is important to note that not all locations may be distinguishable from each other depending on the VLF signals in that environment.

In conclusion the metadata measurement ANNs achieved accurate locations on the dataset. The measurement ANNs can be used to make good predictions months after the collection of the data used to train and validate it. However, as expected, the model can become confused about locations when the VLF signals do not provide enough information to differentiate between different locations.

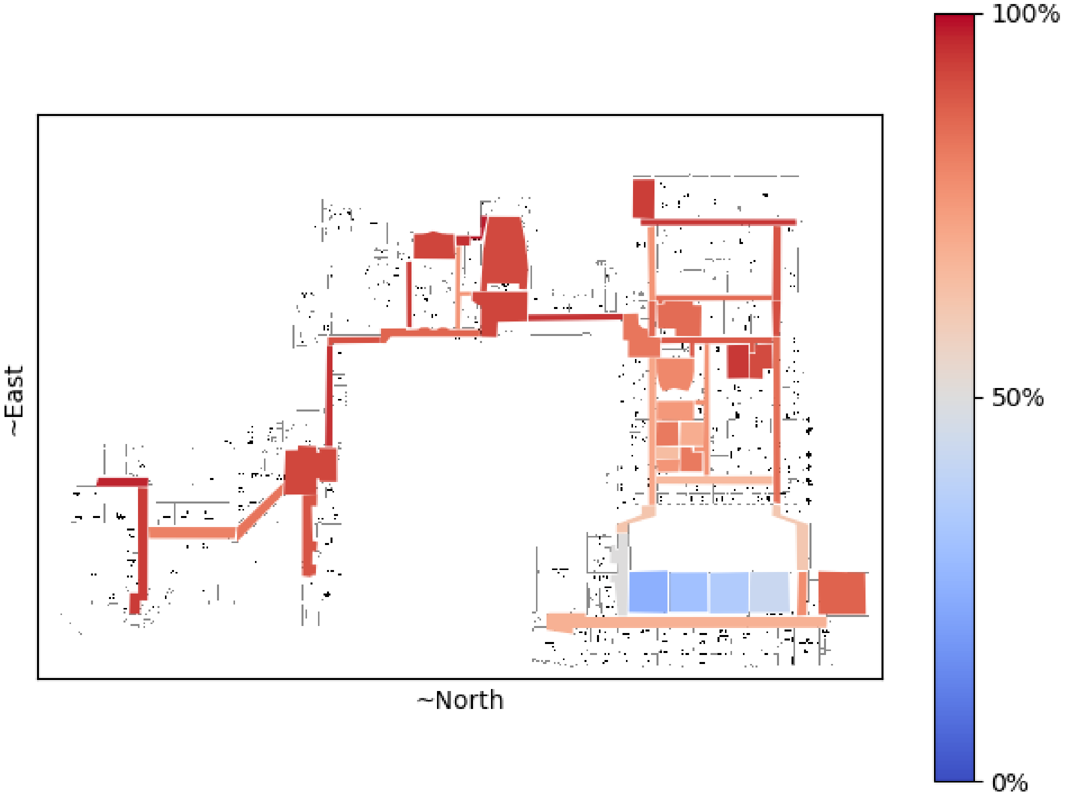

Figure 33: Accuracy of AFIT dataset with metadata results for each location. The accuracy of each room is colored where Red is near 100% and blue is near 0%. Notice the rooms at the bottom with poor accuracy which were confused with each other.

AFIT Dataset with Discrete Particle Filter.

In this section we integrated a discrete particle filter with the two measurement ANNs from Section V. The discrete particle filter was chosen because the AFIT dataset output was discrete. Other filters such as a Kalman filter usually estimate continuous outputs such as position; however, this dataset has discrete output. Thus, a discrete particle filter was chosen to represent discrete locations with particles. Each particle was assigned a discrete location bin and an associated weight. The weight of that particle was changed by the filter to reflect the probability of the user being in that location.

A state transition matrix was used to propagate the particle probability between time steps and before the measurement update. The state transition matrix contained the probability to transition from one location to all other locations. The transition probabilities were calculated from the truth data in the training set from Jan-Apr. The transition matrix was heavy on the diagonal since it calculated transitions for every 100ms and only around 100 transitions occurred in a run out of about 24k time steps. This means less than a 0.5% change of transition per time step. Thus, the user usually stayed in the same location.

To propagate a state probability from one time step to the next, the state probability at each location was simply multiplied by the state transition matrix. To incorporate the measurement probability from the Measurement ANN, the measurement probability was element wise multiplied by the current state probabilities. The result of this operation was normalized to sum to one and become a valid probability distribution. The process then repeated for the next time step as shown:

$$\begin{aligned} \mathbf{x}_{k+1|k} &= \mathbf{\Phi}\mathbf{x}_k \\ \mathbf{x}'_{k+1} &= \mathbf{x}_{k+1|k} \odot \mathbf{m}^{NN}(\mathbf{z}_k, \boldsymbol{\theta}_m) \\ \mathbf{x}_{k+1} &= \frac{\mathbf{x}'_{k+1}}{\left\|\mathbf{x}'_{k+1}\right\|} \end{aligned}$$

where $\mathbf{x}_k$ is our state (location probabilities) at time k, $\mathbf{\Phi}$ is the state transition matrix, $\mathbf{m}^{NN}$ is the trained measurement ANN with weights $\boldsymbol{\theta}_m$, and $\mathbf{z}_k$ is our measurement (VLF features) at time k.

The biggest impact of the discrete particle filter was to smooth out transitions between locations. Near the boundaries of locations the measurement ANN output started to alternate between the two adjacent locations. The discrete particle filter smoothed out the flip flopping by favoring the current location until the next location had much higher probability from the measurement ANN compared to the current location.

The results of integrating the discrete particle filter with the measurement ANNs from Section V in general improved the performance by around 2% for each set compared to the measurement ANN only solution as shown in Figure 34 and 35. A detailed view of the performance is shown in the Appendix Table 17. Thus, by filtering the output of the a measurement ANN we improved performance.

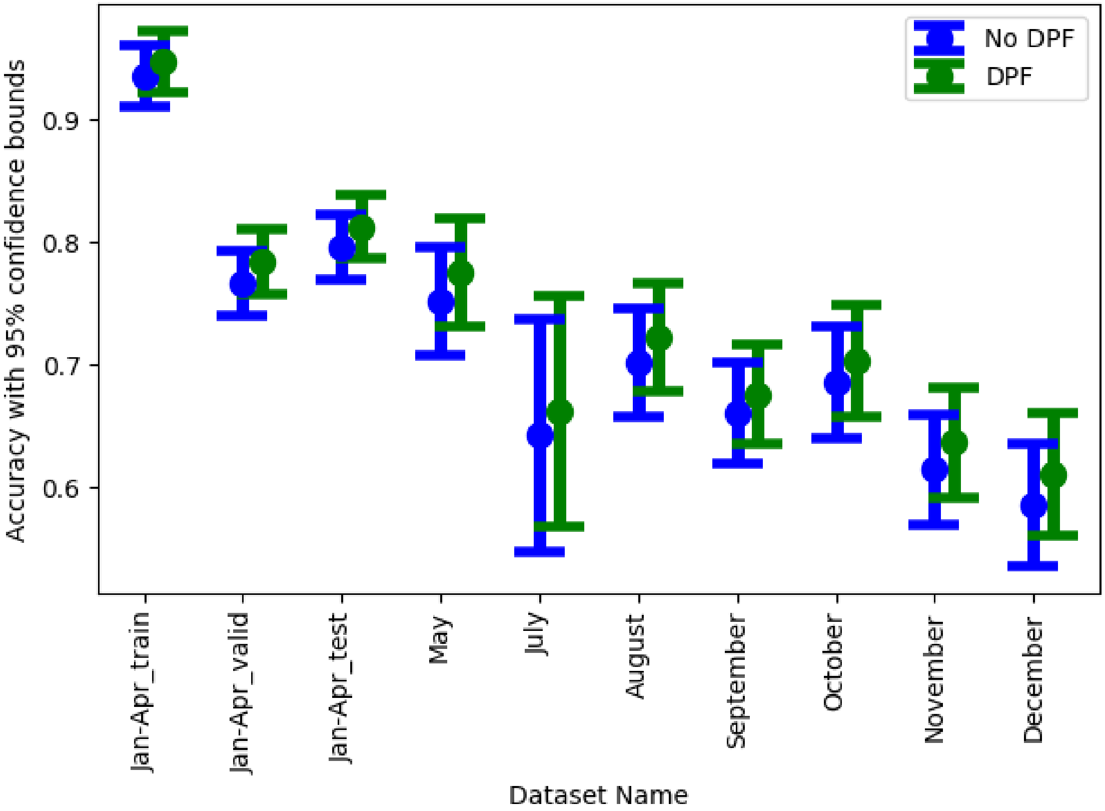

Figure 34: Accuracy of AFIT dataset with measurement ANN no metadata results compared to accuracy of AFIT dataset DPF with state transition matrix integrated with the measurement ANN with no metadata. 95% confidence bars are included for all accuracy estimates.

AFIT Dataset State Predict ANN.

Next we created a state predict ANN for the AFIT dataset. The state predict ANN outputted the probabilities of each discrete location in the same fashion as the output from

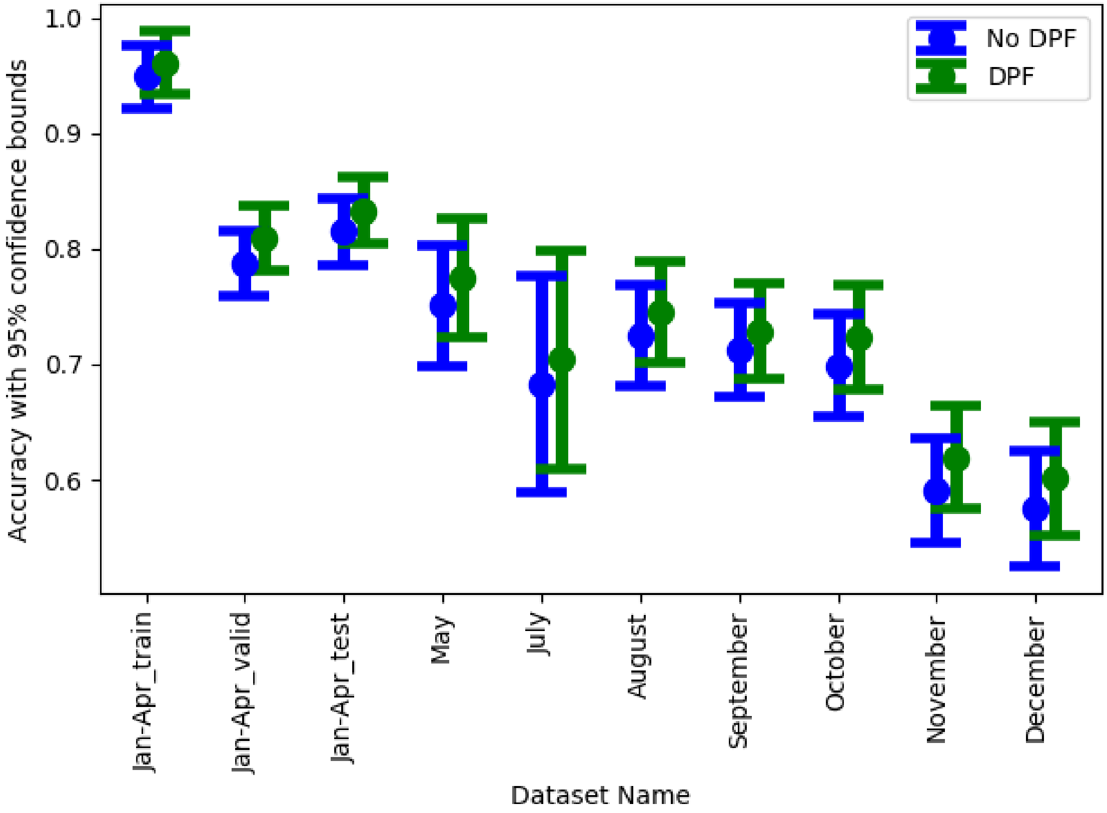

Figure 35: Accuracy of AFIT measurement ANN metadata results compared to accuracy of AFIT dataset DPF with state transition matrix integrated with the measurement ANN with metadata. 95% confidence bars are included for all accuracy estimates.

the measurement ANNs in Section V. However, the state predict ANN used the previous state probabilities as input and not VLF features. Since the position estimates were discrete, the ANN state predict function acted like a state transition model. The state predict ANN remembered previous state transitions in order to predict the next state. For example, the ANN may learn that some rooms are inaccessible from other rooms, or for how long certain rooms are visited. This extra information may make a more accurate state predict function compared to the simple memoryless state transition model used in the previous section.

The state predict ANN was trained in the same method as the measurement ANNs except the input for the state predict ANN was the last true location as a one hot vector instead of VLF features. The measurement ANN was trained on the Jan-Apr training set and validated on the Jan-Apr validation set. The results of the training was almost perfect, since the ANN learned to output the inputted true location as the prediction which was usually correct. These results were expected since, if we input the last *true* location, we expect the the ANN to be able to output the next location with near perfect accuracy. However, inputting the last true location is unrealistic at runtime since our predictions will not be perfect. If the ANN was given the estimated previous location as input, the ANN would quickly become inaccurate as the truth and predictions propagate away from each other. Thus the results are only given when integrated in a discrete particle filter in the following section.

AFIT Dataset Discrete Particle Filter with State Predict.

In this section we integrated a discrete particle filter with the measurement ANNs from Section V and the state predict ANN from the previous section. This integration was similar to the discrete particle filter in Section V but instead of using a state transition matrix, the state predict ANN propagated the state probability from one time step to the next as shown:

$$\begin{aligned}
\mathbf{x}_{k+1|k} &= \mathbf{f}^{NN}(\mathbf{x}_k, \boldsymbol{\theta}_f) \\
\mathbf{x}'_{k+1} &= \mathbf{x}_{k+1|k} \odot \mathbf{m}^{NN}(\mathbf{z}_k, \boldsymbol{\theta}) \\
\mathbf{x}_{k+1} &= \frac{\mathbf{x}'_{k+1}}{\left\|\mathbf{x}'_{k+1}\right\|}
\end{aligned}$$

where $\mathbf{f}^{NN}$ is our trained ANN state predict function with weights $\boldsymbol{\theta}_f$.

To show the results, a comparison of the no metadata DPFs with the state transition matrix filter and the no metadata DPFs with state predict ANN is shown in Figure 36. The full results of the state predict ANN integrations are shown in the Table 18.

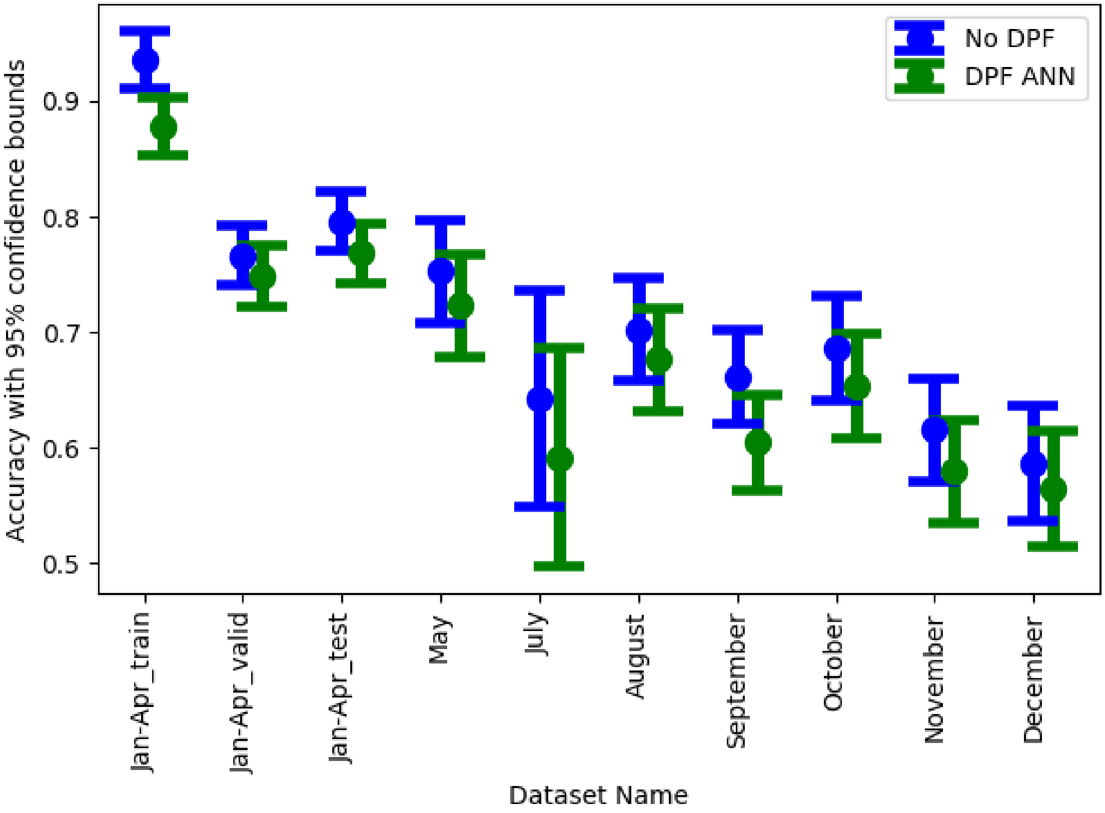

Figure 36: Accuracy of AFIT dataset DPF from data integrated with measurement ANN no metadata results compared to accuracy of AFIT dataset DPF from state predict ANN integrated with the measurement ANN with no metadata. 95% confidence bars are included for all accuracy estimates.

The figure shows that on average the estimates from the state predict ANN integration were around 5% worse compared to the state transition matrix estimates. The poor results revealed that our state predict ANN may have learned to become a poor identity function instead of a propagate function. This may have occurred because most of our time steps remained in the same location. The similarity of input and output was due to a high update rate of 100 ms.

Attempts to encourage the ANN not to become an identity transform proved difficult. Attempts included using different architectures and different update rates, but none proved useful. The entire training algorithm may require modification in order to coax the ANN to learn useful dynamics. However, it was possible for an ANN to learn dynamics as we shall see in the next section when the ANN was trained to be a complete filter.

ANN as Filter for AFIT Dataset.

Finally we trained ANNs to be complete filters for determining position in the AFIT dataset. These filter ANNs were given as input the VLF signal features for the AFIT dataset (like the measurement ANNs) as well as the position estimate from previous time step. The output was again the probability of each discrete location. The complete filter ANNs, unlike the state predict ANN described in Section V, learned extra information from the previous estimate in order to build some system dynamics model.

Two complete filter ANNs were trained with where one version one did not include metadata and the second version did include metadata. Both models included VLF features and the previous location probability estimate. A comparison of the complete filter ANN to the previous best model, the DPF with state transition matrix, is given in Figure 37 and Figure 38. The full results of the complete filter ANN are shown in the Appendix in Table 19.

The figure shows that our filter ANNs outperformed the DPF models in the training, validation and interspersed test set. When comparing on the later test sets, the filter ANN usually performed about the same or better compared to the DPF. This performance increase showed that the complete filter ANNs learned a better dynamics model compared to the filtering with the DPF state transition matrix. The improved dynamics model most likely leveraged the state in the ANN to keep track of state history in order to improve the dynamics model. The internal changing state is in contrast to the DPF which had a static dynamics model that did not use a time history of the states to change location transition probabilities.

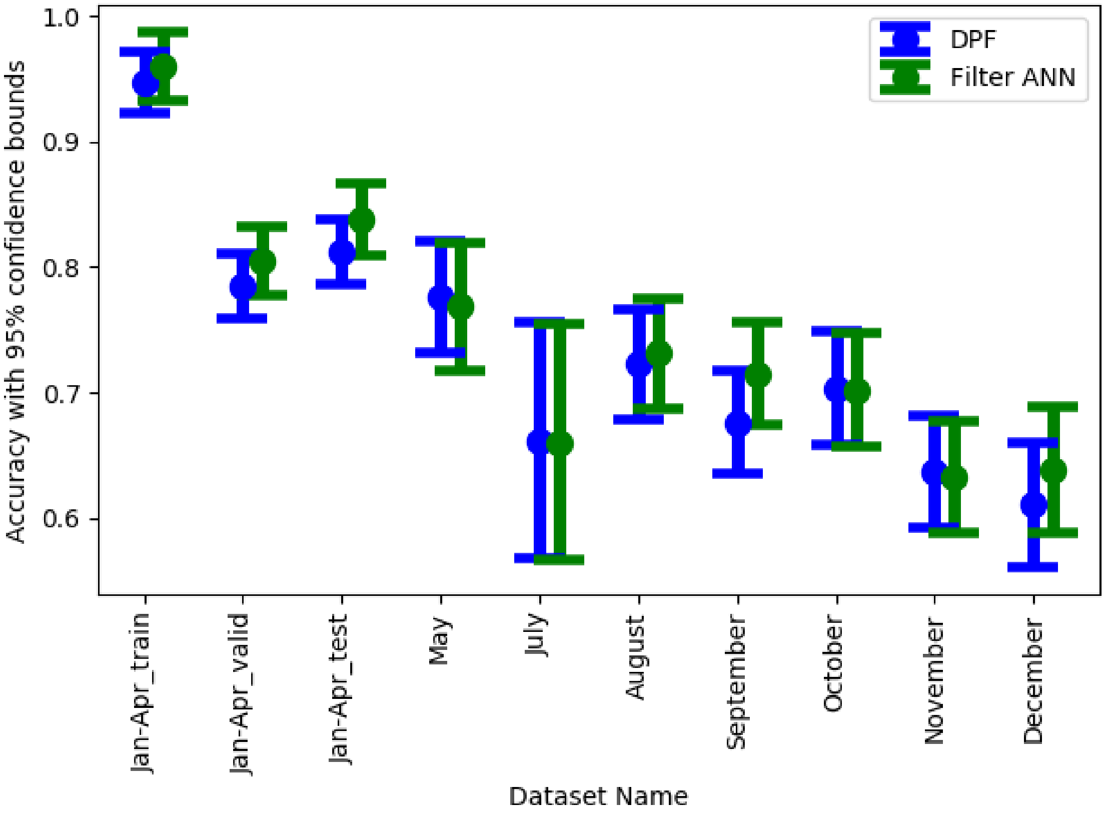

Figure 37: Accuracy of AFIT dataset DPF from data integrated with measurement ANN no metadata results compared to accuracy of AFIT dataset Filter ANN. 95% confidence bars are included for all accuracy estimates.

AFIT Dataset Conclusion.

After all the results from the different kinds of models some trends became clear. The first trend was that models with metadata consistently performed better compared to the same model without metadata. The only exception to this rule was that the last two months (November and December), usually had worse accuracy for the metadata models.

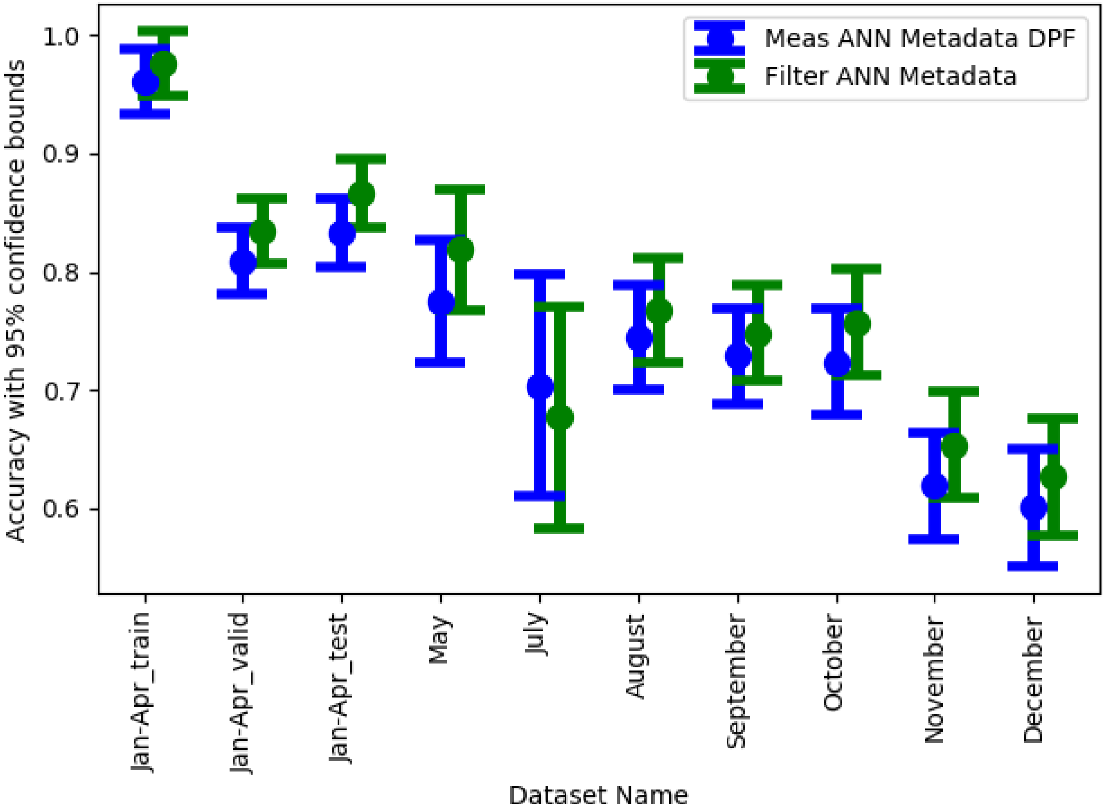

Figure 38: Accuracy of AFIT dataset DPF from data integrated with measurement ANN metadata results compared to accuracy of AFIT dataset Filter ANN with metadata. 95% confidence bars are included for all accuracy estimates.

This seems to suggest that the metadata did not generalize well to that many months in the future. This was unexpected since the metadata should be consistent across time. The VLF features are the features which were expected to degrade over time. The reason for the lack of generalization of metadata is unknown.

Another trend was that including previous position estimates improved overall filter estimates except with the state predict ANN models. As was expected, the DPF and filter ANN models performed better compared to the measurement ANN alone since they used previous position estimates to aid in determining the final estimate. However, the state predict ANN integrated filter had worse performance. As previously stated this was most likely because the state predict ANN did not learn dynamics but instead learned the identity transform due how the state predict ANN was trained.

Following these trends, the model with the best accuracy performance was the filter ANN with metadata model. This model performed better compared to the DPF with metadata. The superiority of the complete filter ANN supports the idea that a properly trained ANN can outperform a simple filter by learning information the simple model does not contain. However, this large ANN solution may be difficult to understand, since it is a complete "black box" solution. In conclusion, ANNs can improve filter solutions by either integrating with an existing filter or performing the filtering itself. Next we move to the bike dataset problem which was a continuous dataset in contrast to the AFIT discrete dataset.

Loss Function Exploration

Before exploring results for the bike dataset we must first determine to what extent the different improvements in Section IV had on the overall error for probability model output problems. To explore the performance we trained ANNs with different combinations of GMM changes and analyzed the results to determine which GMM loss function was best for the bike dataset problem.

GMM Loss Changes.

Most of our GMM loss changes were for the GMM custom loss function outlined in Section IV. All of these changes were compared to a base GMM loss function with no modifications. When we used a GMM custom loss function, the specific custom loss function we used was a MSE loss function. MSE loss was preferred since our output was position and MSE relates directly to distance. We include a quick outline of the loss functions changes that were made and how they alter the loss function. We actually combined all possible combinations of changes to see how they interact with each other. The possible changes to the loss functions are as follows:

- Probability Simplification (PS)– Probability simplification reduces numerical errors and can discourage one mixture to dominate as described in Section IV. Applicable to all GMM Loss functions.

- GMM Custom Loss Function Scaling– Allows β scaling parameter between means and covariance. We used values from 10^0 to $10e9$ in increments of powers of 10 for a total of 10 different scalings. Applicable only to GMM Custom Loss functions.

- Gradient Correlation and Scaling (GCS)– Reintroduce covariance information to custom loss function derivative previously found in unmodified GMM loss as described form Section IV. Applicable only to GMM Custom Loss functions.

- Gradient Normalization (GN)– Normalize magnitude of gradient for means and covariances in GMM Custom Loss (still allows scaling) as described in Section IV. Note that our β scaling parameter is still applied to the custom loss function after the normalization. Applicable to any loss function with a summation of other loss functions such as GMM Custom Loss functions.

These four options were combined in all possible ways. The changes PS, GCS, and GN were all booleans and were either True or False. Our β parameter had 10 values. This made a total of 80 different models that used the GMM custom loss function. The unmodified GMM loss had two variants with PS as True and False. Finally there was one regression model with simple MSE loss to serve as a baseline for all the GMM loss function. Since the regression model was very different from the other 82, it was not included in the analysis but served as the alternate performance when covariance was not required. This made a grand total of 83 models (82 in the analysis).

Analysis of GMM Loss Changes.

All 83 loss functions described in Section V were used to train models with the architecture of Section V according to the training described in Section V. The bike dataset with only the 10 training and 9 validation data collects from 2016 was used to train and validate the models. There was no test set for this section and all results were calculated on the validation set. All models were evaluated by calculating the RMS error between the mean of the highest weighted Gaussian and the truth. The NCE metric described in Section IV was also calculated between the truth and the mean and covariance of highest weighted Gaussian distribution.

Since there were 82 models with four factors to evaluate, an ANOVA was created to explain the contribution of all four factors. The ANOVA model was fitted to only 82 models and the regression model was not used. The ANOVA model included all second order interaction terms between the four factors. All the False values from the boolean factors and CLS with a value of 1 were combined into the intercept to make the formula:

$$\begin{aligned} y = &\text{intercept}+ \\ &\text{PS} + \text{CLS} + \text{GCS} + \text{GN}+ \\ &\text{PS} \times \text{CLS} + \text{PS} \times \text{GCS} + \text{PS} \times \text{GN} + \text{CLS} \times \text{GCS} + \text{CLS} \times \text{GN} + \text{GCS} \times \text{GN} \end{aligned} \tag{153}$$

where each variable has an associated β value which denotes the contribution of that parameter. Our parameters PS, GCS, and GN are either 0 or 1 for False and True respectively, CLS takes on the 10 values 1 to $1e10$ in powers of 10, and our intercept value is always 1. The × operator denotes the terms being multiplied together to make a new column for second order interaction terms. This ANOVA was evaluated for three different values of the dependent variable y – RMSE, NCE, and a combined metric. The combined metric normalized RMSE and NCE (minus outliers) then added the result together to reward models that lowered both metrics. The ANOVA results were then calculated as shown in Table 5 (for the raw data see Appendix Section A). The results show the Beta values that are multiplied by each parameter to predict the output. Note that negative Beta values were better since we were attempting to minimize both RMSE and NCE. The table shows some interesting properties of how our factors are changing our metrics. First we examine the first order effects.

Table 5: The ANOVA results of fitting the factors for each GMM loss function to the metrics RMSE, NCE and a combined metric.

Name	RMSE Beta values	NCE Beta values	Combined Metric Beta values
Intercept	77.2059	-18.1787	-9.4786
GCS	3.0715	-8.2863	-3.6033
GN	-0.1007	91.9426	43.2989
PS	7.6556	-10.9132	-4.3932
GCS×GN	-2.7262	-83.8478	-39.5263
GCS×PS	-15.3753	87.5462	39.7368
GN×PS	-2.2656	-82.5163	-39.0897
CLS	-0.3910	-0.1317	-0.1002
CLS×GCS	0.0338	4.9499	2.3379
CLS×GN	1.0381	-4.9863	-2.2474
CLS×PS	-0.1428	5.2056	2.4381

For first order effects it appeared that only GN and CLS were useful in making the RMSE value smaller on average. Smaller RMSE due to CLS made sense because as it grows larger we were adding more emphasis to the custom loss function which only involves RMSE. GN may have been useful because in normalizing the two loss function it may have given less emphasis to variance or more to the means. The other terms GCS and PS both seemed to on average increase the RMSE error when included. Next when observing NCE, the terms GCS, PS and CLS all made the NCE metric better on average. Only GN made the NCE worse on average. This may be because GN attempted to place equal emphasis on the means and covariances which actually produced worse covariance estimates. The trends of NCE were similar to the trends in the combined metric which suggests that NCE had a larger impact on the combined metric compared to RMSE.

Next we look at the second order terms. We first note that the second order terms represented a synergistic effect of the first order terms. For example if factor A had a β value of 10 and factor B had a β value of 20 we would expect both to contribute 30. The second order terms represented the difference from this combined value. For example, if the true contribution of factor A and B from our example was 35, our second order term A×B would be 5 to represent this synergistic effect of using both factors. With this we now move on to explain the second order terms.

First we note that GN combined with any other factor usually decreased both the RMSE and NCE metrics. However, this may have just been compensating to remove the increase GN had as a first order term. Thus, when combined the bad effect of GN was just canceled out. The other notable second order effect was GCS and PS. For the RMSE metric is seems that combining GCS and PS removed the performance drop from the first order terms and then actually made the RMSE better. However in terms of NCE it caused a drop in performance to negate the positive effects of the first order terms. All other second order terms involving CLS were small and only marginally changed the metrics. As with the first order terms, the trends of NCE were mirrored in the combined metric.

For our final model we choose the model that performed best on the combined metric since we were interested in minimizing both metrics. The best model ended up using PS, GN and had a CLS=10. With this chosen model we examined it more closely to compare it to standard techniques. For comparison we show the regression model and the unmodified

GMM Loss function (no additional factors) model compared to our chosen model in Table 6. From the table it become clear that our chosen model was better in RMSE compared to the regression model and better compared to unmodified GMM in terms of NCE. Thus the ANOVA analysis allowed us to chose parameter which improved our RMSE by about 25 m compared to standard regression which provided no covariance. However, when we compared the best GMM loss to unmodified GMM loss, the RMSE improvement is similar but ironically the GMM loss estimate of covariance was very poor compared to our best GMM loss. Also when comparing the combined metric the difference between the two GMM models becomes obvious. Of the 82 models in the ANOVA all but six models had a combined metric under 2 and one of those six worst models was the unmodified GMM loss. In summary, our new GMM loss function was much better compared to both a regression loss model and the standard GMM loss model. With this chosen model we now move on to our analysis of the bike dataset

Table 6: Model comparison between select model loss functions.

Model Loss Function	RMSE value	NCE value	Combined Metric
Regression with MSE Loss	93.5598	N/A	N/A
Classic GMM Loss	69.7261	100.668	10.76
Best GMM Loss (PS, GN, CLS=10)	69.1153	0.23329	0.71

Bike Dataset

This section develops implementations of the ANN and filter integration framework from Chapter III for the bike dataset. First, measurement ANNs were created using VLF signals as input. Next the trained measurement ANNs were integrated with a Kalman filter to show ANN integration with a classic filter. Next, a state prediction ANN was created for the bike dataset which used previous position information to predict the next position. This state prediction ANN was integrated with the previous measurement ANN integrated Kalman filter model by replacing the Kalman filter propagate function with the state predict ANN. Finally complete filter ANNs were created to provide a position estimate from measurements and previous state using only a single ANN.

Bike Dataset Measurement ANN.

This section outlines eight measurement ANNs for the bike dataset. The measurement ANNs came in two groups, without metadata and without metadata. The metadata used for the metadata ANNs included accelerometer, gyroscope, magnetic heading, true heading, heart rate, odometer speed, crankshaft speed, percent through day, and season sine wave according to step tracking from Table 3. The training and validation data was selected from both the 2016 and 2017 bike dataset collects, however the test set was selected exclusively from the 2017 collects. Each model group consisted of measurement ANNs with different loss function outputs. The first measurement ANN in each set was trained to output classification results for binned locations which were converted to 2D position. The next measurement ANN in each set was a regression model trained to output the 2D position directly. The third and fourth measurement ANN was trained to output 2D position probabilities as a GMM. A list of the models is given in Table 7.

For the classification measurement ANNs, the model in Section V was trained according to Section V. The final output layer had 100 outputs for each of the 100 location bins. A softmax activation function was used on the final layer to output a valid probability distribution. One model was trained for each the without metadata and the with metadata groups. The model was trained only on the training data and validation was used to determine the best model and parameters selection. First, the models were trained to output the probability of each binned location. After this training the no metadata model obtained a test accuracy was 64.1% compared to the metadata model which obtained an accuracy of

69.0%. Note the test accuracy is only for runs in the second year of data collects. A heat map of the metadata accuracy is show in Figure 39.

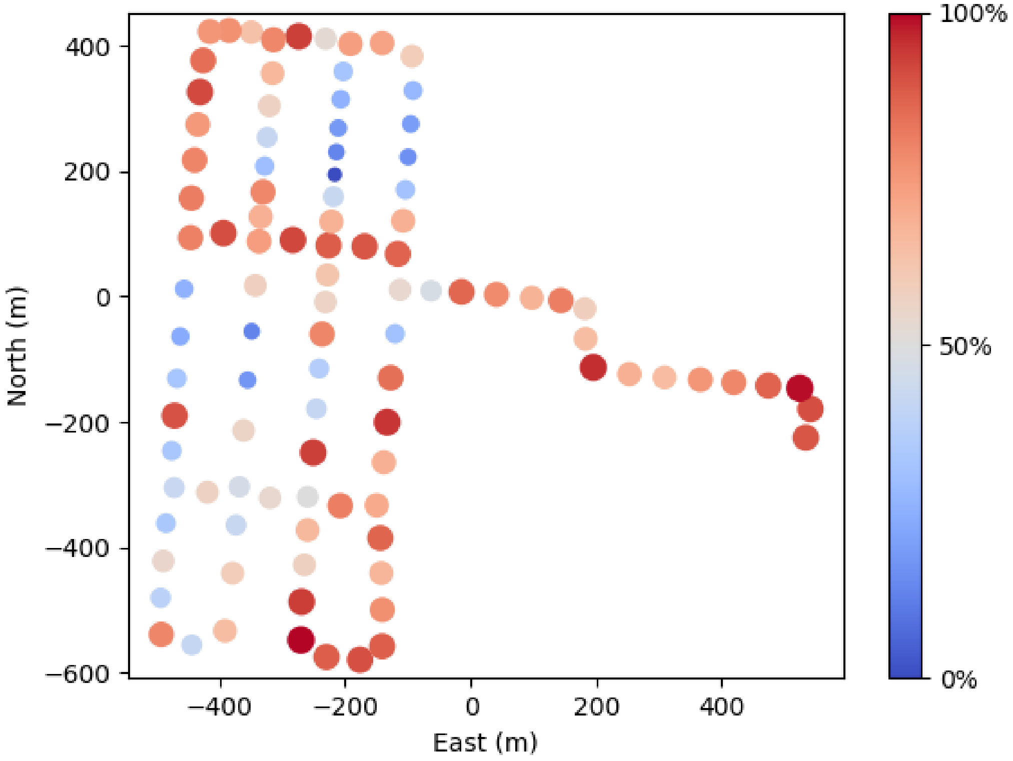

Figure 39: Test results from classification of measurement ANN with metadata. Each circle represents the center of one location bin. Larger and red circles have higher accuracy while smaller and blue circles have lower accuracy.

Next we turned this classification model into a regression model with the method given in Section V. To obtain a regression result, the output probabilities of the model were weighted by the 2D center of the location and summed along each position dimension. Next the model weights (including the weights for the output probabilities) were fine tuned with a learning rate of 1×10^{-6} for 500 epochs. After fine tune training, the metadata and

Table 7: Test performance for the two groups of regression models. The variances σ_x and σ_y are the variance of the test error for each model. Note the GMM models can give variance per sample unlike the other two models which only have the final variance.

No Metadata		
Loss Function	RMSE (m)	NCE
Classification Fine Tuned	78.68	N/A
Regression	88.86	N/A
Classic GMM ML	62.32	97.16
Best GMM ML	67.72	0.45

Metadata		
Loss Function	RMSE (m)	NCE
Classification Fine Tuned	67.04	N/A
Regression	105.58	N/A
Classic GMM ML	56.32	24.85
Best GMM ML	69.59	0.75

no metadata models achieved a final test RMS error of 67.04 m and 78.63 respectively as shown in Table 7. To provide some perspective on the performance error the entire area of navigation was about 1 square kilometer. In addition, the distance between north-south parallel roads was about 120m and east-west parallel roads was about 300m. Thus, our RMS error is well within the bounds to uniquely identify which road we are traversing. Also note that our truth source is a cell phone GPS receiver which itself is accurate to about 5m [16]. Next we trained a model to estimate position directly.

Regression measurement ANNs were trained to output the 2D position directly as a regression output. These ANNs were based off the architecture in Section V and trained according to Section V. Once again there were two different models trained, one without metadata and one with metadata. The final layer of the regression models had two outputs for the 2D location problem and the final layer activation was linear. As in the last section, the models were trained and validated on data from both years but tested only on the second year of data. This resulted in a test RMS error of 88.86 m for the no metadata measurement ANN and 105.58 m for the metadata model. This is unusual since we expected adding additional metadata to at worst have the same result. It appears that some metadata models did not generalize well to the test set. This same outcome occurred in the next section for GMM loss functions as well. Also the regression model performed worse compared to the classification weighted regression. Estimating the probability of each location created a better solution compared to directly estimating the output. The regression solution was actually the worst as we shall see the MDN in the next section also outperform both regression and classification weighted regression.

Finally MDN measurement ANNs with the architecture from Section V was trained according to Section V. The final layer contained 42 outputs to describe the six parameters of the seven distributions of the Gaussians in the GMM. Two different GMM loss functions were used. First was the unmodified GMM loss with no extra additions. The second was the best performing GMM loss function from Section V. Each of these loss function types were trained without and with metadata to create a total of four models. The GMM activation was used on the final output layer to ensure the covariance parameters created a valid covariance matrix and the weights of each distribution summed to one. The without metadata unmodified GMM loss resulted in a test RMS error of 62.32 m. Similarly the metadata best GMM loss model achieved a test RMS error of 67.72 m. Again we see the unusual result where our metadata loss for the best GMM model was 69.59 m which was worse compared to our without metadata result. As in the last section it appears the metadata model also did not generalize well. Not including the lack of generalization however, the MDN models perform better compared to the regression and classification weighted regression. Next we compare the MDN models.

In a similar fashion to our GMM loss exploratory results in Section V, our unmodified GMM loss function obtained better RMS error compared to the best GMM loss. In our exploratory results the difference was under a meter and thus negligible. In these results, the difference was a few meters for no metadata and worse for metadata since it did not generalize as well. Despite the improvement in RMS error, once again the NCE metric was significantly better for the best GMM loss model compared to the unmodified GMM model. Due to the small but significant difference in RMS error and the very large difference in NCE, we still preferred the best GMM loss model. While the RMS error is better, the entire purpose of using an MDN is to provide covariance. The covariance provided by the unmodified GMM model is quite poor when looking back on the GMM loss exploratory results which had all but 6 models with a NCE under 2. Thus for the following section we used the best GMM loss function model as a baseline to compare to new filter implementations.

Kalman Filter Integration.

This section integrates a Kalman filter with the MDN measurement ANNs from the previous section. A Kalman filter does not know how to relate the VLF signals to position, since the signals are unknown. However the measurement ANNs from Sections V were trained to determine the relationship between VLF signals and position. The measurement ANNs learned position from VLF signals without a priori knowledge of the types of VLF signals and how they relate to position. Thus, the measurement ANNs bridged the gap between VLF signals and a Kalman filter to make position estimates. Using only a Kalman filter, position estimation would not be possible without these ANN enabled measurements.

The Kalman filter leveraged other a priori information about the problem that the measurement ANN did not learn. For example, position is known to be continuous and the integral of velocity. The Kalman filter incorporated this assumption easily in the dynamics equations. The Kalman filter combined the measurement ANN position with the known physical constraints on the position states to improve the final filter estimates.

Kalman Filter Design.

The measurement ANN from Section V and a Kalman filter were integrated by providing the Kalman filter the measurement ANN position estimates. The measurement ANNs estimated the current local level position $\mathbf{z}$ based on inputs $\mathbf{s}$ as shown $\hat{\mathbf{z}}_k = A(\mathbf{s}_k, \mathbf{h}_{k-1}, \boldsymbol{\theta}^A)$. The Kalman filter takes the measurements $\hat{\mathbf{z}}$ into its update equation and outputs a filtered version of the states $\hat{\mathbf{x}}$, since the output position in this case is the states themselves.

The Kalman filter used a First Order Gauss Markov Model (FOGM) model for acceleration and integrated to obtain velocity and then integrated again to obtain position. The Kalman filter also included two states for the measurement bias of the measurement ANNs in the north and east directions. The measurement biases served to remove the time correlation of the assumed independent measurements. Without this bias, the Kalman filter assigned too much confidence to the filter solution since the measurement ANN generates correlated measurements which the Kalman filter believed were independent. The correlation in the measurements was due to the fast update rate (100 ms) and that our measurements ANN had persistent state information. The bias states were also modeled as a FOGM process like the acceleration states. Thus the states of our Kalman filter were:

$$\mathbf{x} = [n, e, \dot{n}, \dot{e}, \ddot{n}, \ddot{e}, n_{mb}, e_{mb}] \tag{154}$$

where $\mathbf{x}$ is the state vector with north and east positions, north and east velocities, north and east accelerations, and north and east measurement biases. Next we define the continuous system dynamics matrix with the FOGM acceleration model with time constants of τ^n_{acc}, τ^e_{acc}

and the measurement bias with time constants $\tau_{mb}^{n}, \tau_{mb}^{e}$ for the north and east directions respectively:

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/\tau_{acc}^{n} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/\tau_{acc}^{e} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1/\tau_{mb}^{n} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1/\tau_{mb}^{e} \end{bmatrix} \tag{155}$$

Next we define the continuous noise correlation matrix:

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\sigma_{n}^{2}/\tau_{acc}^{n} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\sigma_{e}^{2}/\tau_{acc}^{e} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2\sigma_{e}^{2}/\tau_{mb}^{n} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2\sigma_{e}^{2}/\tau_{mb}^{e} \end{bmatrix} \tag{156}$$

where the FOGM acceleration state noise standard deviations are $\sigma_{acc}^{n}, \sigma_{acc}^{e}$, and the measurement bias FOGM process a noise standard deviations are $\sigma_{mb}^{n}, \sigma_{mb}^{e}$. The time constants and noise standard deviations for the acceleration states were determined from examining the autocorrelation of the double difference of position in the validation data. the measurement bias time constants and noise values were determined from examining the autocorrelation of the measurements from the validation set of data. However, the measurement bias noise values from the dataset were multiplied by three because empirically the default value put too much trust in the measurements. The constants for both processes are listed in Table 8. Note that these are the continuous time matrices which were converted into their discrete counterparts with a time step of $t = 0.1$ s before being used in the Kalman filter.

Table 8: Parameters used in the measurement ANN integrated Kalman filter.

Parameter	Value
σ^n_{acc}	0.0577 m/s^2
σ^e_{acc}	0.0388 m/s^2
τ^n_{acc}	0.1 s
τ^e_{acc}	0.1 s
σ^n_{mb}	$14.51 \times 3 = 43.53$ m
σ^e_{mb}	$9.29 \times 3 = 27.87$ m
τ^n_{mb}	44.4 s
τ^e_{mb}	38.0 s

Now we define the measurement matrix where the ANN estimates the north and east position states along with a bias term:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{157}$$

where the **H** matrix adds the measurement bias states and current position to estimate the measured position from the ANN.

The last parameter is the measurement noise matrix **R** which has two forms. The values for the measurement noise matrix $\mathbf{R}_{data}$ were determined by analyzing the covariance of the error between the measurement ANN estimates and the true output values on the validation data set over the entire set. The matrix $\mathbf{R}_{data}$ was static and did not change during filter execution. The second form for the measurement noise matrix used the covariance value from the MDN at each time step denoted as $\mathbf{R}_{ANN}$ on a measurement-by-measurement basis. For this experiment we used the covariance of the Gaussian distribution for which the highest weight occurs. The MDN provided measurement noise matrix changed for each time step based on the measurements. The dynamic property of the measurement noise matrix made our Kalman filter adaptive to the noise in the measurements.

A total of four models were used for the Kalman filter experiment. Two models used the static mean measurement noise matrix and the other two used dynamic measurement noise matrices determined from the measurement MDN. These two types of models were further split into measurements ANNs without metadata and with metadata.

A summary of the results running the four models is given in Table 9. As expected all the test RMS errors improved from their measurement-only counterparts. Thus our filter was able to smooth out some of the incorrect measurements and improve our final position estimate. The metadata results were slightly worse which was due to the metadata measurements being worse as shown in Table 7. In conclusion, the Kalman filter improved performance compared to measurements alone as expected.

Bike Dataset State Predict ANN.

Next we trained a state predict ANN to estimate the next position given the current position. This integration approach was done instead of correcting the Kalman filter state predict because there was little a priori knowledge about the system dynamics. The knowledge that does exist was could only make a simple motion FOGM model as shown in the previous section. Such a simple system dynamics function did not take into account the specifics of the bike problem such as staying on roads or bicycle dynamics. Correcting this simple model may be more difficult compared to directly estimating the next position

Table 9: Comparison of performance between both types of **R** matrix calculations– from data and from ANN. Results persented for both test and validation sets.

No Metadata				
	Test		Validation	
Model	RMSE	NCE	RMSE	NCE
Static $\mathbf{R}_{data}$	63.28	6.33	40.14	8.31
Dynamic $\mathbf{R}_{ANN}$	67.06	5.27	72.21	8.97

Metadata				
	Test		Validation	
Model	RMSE	NCE	RMSE	NCE
Static $\mathbf{R}_{data}$	63.83	2.60	61.81	6.47
Dynamic $\mathbf{R}_{ANN}$	65.98	2.82	64.13	5.70

since we desire the state predict to learn the road position. Thus a state predict ANN was trained from only the collected truth trajectories without any other a priori information.

The state predict ANN was trained using the training set and the validation set was used for parameter and best model selection. The input to the state predict ANN was the true normalized 2D position. The ANN used the best GMM loss function and thus outputted the parameters for a GMM. Results were reported for the Gaussian distribution with the maximum likelihood estimate.

Unlike the AFIT dataset state predict ANN in Section V, the bike state predict did *not* become an very good identity transform. Interestingly the model favored output which was biased to the inside of the roads as shown in Figure 40. A more desirable result would have been better if the output an identity transform because that may have at least stayed on the roads more often. However, it appears that training state predict ANNs is difficult as previously noted in the AFIT dataset. Despite this result, we proceeded to integrate this state predict function with a Kalman filter.

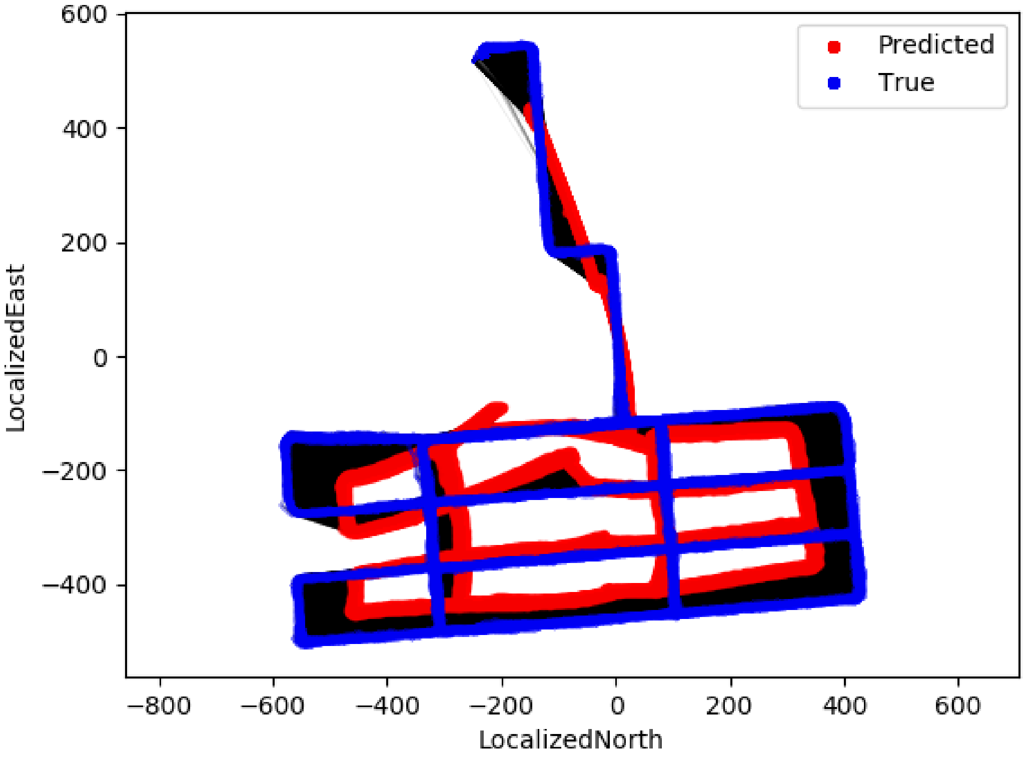

Figure 40: Bike state predict function output. Position estimates are red and connected to their true blue positions by a black line. Note the ANN input was the true position and the ANN still favored outputting position that were not on the actual roads.

Bike Dataset Filter with State Predict ANN.

Next we replaced the state predict function of the Kalman filter with our trained ANN. We used the Kalman filter from Section V with a few modifications. The first modification was that we only estimated position and measurement bias thus removing velocity and acceleration. This reduced our states to:

$$\mathbf{x} = [n, e, n_{mb}, e_{mb}] \tag{158}$$

where $\mathbf{x}$ is the state vector with north and east positions, and north and east measurement bias. Next we define the system dynamics but instead of using a matrix to multiply by the previous state, we use the state predict from the ANN:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{f}^{NN}(\hat{\mathbf{x}}_{k-1}|\boldsymbol{\theta}^F)$$

where our propagated state estimate (without measurement update) is $\hat{\mathbf{x}}_{k|k-1}$, and $\mathbf{f}^{NN}$ is our trained state predict ANN with weights $\boldsymbol{\theta}^F$ which takes the previous position $\hat{\mathbf{x}}_{k-1}$ as input. Since we are now using a function to propagate and not a matrix, we must swap our Kalman filter for an EKF. To do this we require the derivative of the state predict ANN with respect to the input positions. We can have calculated this derivative since the ANN is differentiable; however, to save on computation time we assumed the derivative was an identity matrix $\boldsymbol{I}_2$. This was usually very close to the true derivative because our update rate (10 times a second) was so high, and our bicycle dynamics so slow, that the position rarely changed much between time steps. Next we define the continuous noise correlation matrix:

$$\mathbf{Q} = \begin{bmatrix} \sigma^n_{pos} & 0 & 0 & 0 \\ 0 & \sigma^e_{pos} & 0 & 0 \\ 0 & 0 & 2\sigma_e^2/\tau^n_{mb} & 0 \\ 0 & 0 & 0 & 2\sigma_e^2/\tau^e_{mb} \end{bmatrix} \tag{159}$$

where we used the same measurement bias constants as from the previous section. However, our noise values for the position states were determined by the covariance of the prediction error on the validation set. The off diagonal values were small and thus assumed to be zero making our noise sources in the state predict independent. A list of the parameters used in the EKF are given in Table 10.

Table 10: Parameters used in the measurement ANN integrated EKF.

Parameter	Value
σ^n_{pos}	250 m
σ^e_{pos}	170 m
σ^n_{mb}	$14.51 \times 3 = 43.53$ m
σ^e_{mb}	$9.29 \times 3 = 27.87$ m
τ^n_{mb}	44.4 s
τ^e_{mb}	38.0 s

Our results for integrating were actually very poor as expected since we used a poor state predict function that was biased toward the center of the map. The biased state predict function caused all the filter estimates to be biased to the center. This in turn lead to a bad filter solution as shown in Table 11.

Incorporating the state predict ANNs into the filter did not improve the performance of the filter. In fact these are the worst results yet. Using only the measurement ANN would have produced better results. However, this was due to the poor state predict ANN training. It appears that training a stand alone state predict ANN was difficult for both discrete and continuous problems. However in the discrete case a complete filter was able to learn dynamics and become our best performing model. Thus, in the next section we create a complete filter for the continuous output of the bike dataset.

ANN as Filter for Bike Dataset.

In this section ANNs were trained as a complete filter to determine position in the bike dataset. Unlike previous ANNs, these filter ANNs used the last position estimate as well as VLF features and metadata. This allowed the complete filter ANNs to make accurate filtered predictions based on the learned system dynamics as well as the measurements. To

Table 11: Performance of EKF integrated with a measurement ANN and the state predict ANN for both the no metadata and metadata datasets.

No Metadata		
Set	RMSE (m)	NCE
train	113.89	2.352
valid	128.09	3.74
test	132.75	3.78

Metadata		
Set	RMSE (m)	NCE
train	114.77	2.524
valid	126.32	3.58
test	133.36	3.83

improve the performance compared to a measurement ANN, this filter ANN must learn system dynamics. Without system dynamics, the filter ANN would become a slightly different version of a measurement ANN.

Two filter ANNs were trained. One ANN did not use metadata as input while the other was given the metadata input for the bike dataset. The previous position data consisted of the mean and covariance of the Gaussian distribution with the maximum likelihood point. The covariance information was in the upper triangular of the inverse covariance matrix form as defined in Section II. Thus, the ANN was given a 2D point estimate of location and the information to derive the covariance matrix. The filter ANN calculated the full GMM distribution to characterize the output. During training the filter ANNs started with the previous position information zeroed out. The filter ANNs were strictly given the last predicted positions, and thus teacher forcing was never used.

The results of this training for both the no metadata and the metadata models are given in Table 12. From the table, the test RMS error was actually worse compared to our previous estimates besides the regression model and state predict ANN integrated model. To explain this performance, the table includes the training data as well as the validation and test to show an interesting trend. Both filter ANN models over fit to the training data, which was similar to the measurement ANNs; however, the filter ANNs have a problem in generalizing that the other models did not.

While we did provide covariance information about the previous position to the filter ANN, there was no guarantee the ANNs would utilized this information properly. For example, during training the ANN might have learned some internal idea of how much to trust the previous position, ignoring the covariance information. However, when evaluating the validation set, the internal covariance built into the ANN was incorrect, since it was based on training accuracies. This mismatch in covariance may have caused the over fitting in all the models to be compounded in the filter ANN. The over fitting due to measurements created further over fitting in the previous position estimate.

The discrete filter ANN in Section V avoided this problem because the previous estimate was represented with probabilities making it impossible to separate location from confidence in the location. Thus in order to improve the continuous filter ANN performance, some method to prevent this over fitting is required. If this over fitting problem is not dealt with then the MDN measurement ANNs may achieve better performance as shown in our experiment.

Bike Dataset Conclusion.

After compiling all results as shown in Table 13 we now examine the trends across all the trained models. The exact trends were more difficult to pin down compared to the AFIT dataset, since the bike dataset contained some outliers and unexpected results. However we still uncovered some general trends.

Table 12: Model comparison between chosen GMM loss, regression and unmodified GMM Loss.

No Metadata		
Set	RMSE	NCE
train	26.21	0.6079
valid	86.83	3.4667
test	88.31	2.1651

Metadata		
Set	RMSE	NCE
train	10.69	0.7694
valid	71.67	9.1634
test	83.22	9.4645

Our first trend is that unlike the AFIT dataset, which generally showed improved performance when using metadata, the bike dataset actually experienced worse results when using metadata. There are two possible reasons for this. The first is that more data may be required to use the metadata well. The AFIT dataset contained much more data compared to the bike dataset. Also, the continuous nature of the bike dataset along with this lack of data, may have made it even more difficult to use the metadata. However, to provide evidence for this conjecture we would need to collect more bike data. Another explanation is that the metadata was just not as useful in the bike dataset compared to the AFIT dataset. For example step data in the AFIT dataset may have been more useful compared to odometry bike dataset. This would be strange since both datasets contained the same metadata besides a few fields. The fields that were not in both also should have

Table 13: Performance for the models of the Bike dataset.

No Metadata						
	Test		Valid		Train	
Loss Function	RMSE	NCE	RMSE	NCE	RMSE	NCE
Classification Fine Tuned	78.68	N/A	78.35	N/A	13.62	N/A
Regression	88.86	N/A	95.81	N/A	51.86	N/A
Classic GMM ML	62.32	97.16	64.12	88.02	16.45	0.724
Best GMM ML	67.72	0.45	73.20	0.22	19.74	1.310
Static $\mathbf{R}_{data}$	63.28	6.33	40.14	8.31	22.77	0.758
Dynamic $\mathbf{R}_{ANN}$	67.06	5.27	72.21	8.97	31.27	0.365
ANN Prop	132.75	3.78	128.09	3.74	113.89	2.352
Filter ANN	88.31	2.16	86.83	3.46	26.21	0.607

Metadata						
	Test		Valid		Train	
Loss Function	RMSE	NCE	RMSE	NCE	RMSE	NCE
Classification Fine Tuned	67.04	N/A	66.08	N/A	13.03	N/A
Regression	105.58	N/A	93.34	N/A	53.83	N/A
Classic GMM ML	56.32	24.85	48.52	23.06	15.81	0.278
Best GMM ML	69.59	0.75	65.29	0.45	16.34	1.348
Static $\mathbf{R}_{data}$	63.83	2.60	61.81	6.47	19.14	0.913
Dynamic $\mathbf{R}_{ANN}$	65.98	2.82	64.13	5.70	29.36	0.493
ANN Prop	133.36	3.83	126.32	3.58	114.77	2.524
Filter ANN	83.22	9.46	71.67	9.16	10.69	0.769

been providing similar information. For example odometry in the bike data gives a speed in a similar manner to the speed provided by the step counter in the AFIT data.

Our second observation is that in both the bike and AFIT dataset, training a state predict ANN was difficult. Once again our standard filter dynamics obtained superior results compared to our ANN provided dynamics. This is more evidence that some other method is required to train a state predict ANN. Despite the failings of our state predict ANN, the standard filter integration did in general provide better performance compared to a measurement ANN only.

Conclusion

In conclusion we have shown two problems that used VLF signals to navigate using ANNs. Our problems included a discrete indoor problem and a continuous outdoor problem. Both problems used different examples of ANN integration to navigate with varying degrees of success. Both problems created measurement ANNs to determine position from VLF features. The measurement ANNs were successful in determining position of the mobile antenna. The indoor case could discriminate between adjacent rooms while the outdoor case could discriminate between adjacent parallel roads. Next filtering with dynamics was introduced in different methods that included standard methods and ANN enabled methods. The standard methods improved performance while the ANN methods sometimes improved performance. The complete filter ANNs in the AFIT dataset obtained the best performance on the AFIT dataset, while the Kalman filter provided dynamics obtained the best performance on the bike dataset. Overall these results would not be possible without ANNs given the lack of a priori information about the VLF signals.

VI. Summary and Future Work

This dissertation outlined a framework for integrating ANNs and navigation filters. The frameworks consists of two distinct sections that show how ANNs can replace filter components and how ANNs can augment existing filters. These two types are not mutually exclusive, and the third section detailed how the two types can complement each other. Examples of previous work were placed into the proposed framework. This showed how the proposed framework can be used to describe many different types of work, not just the ones included in this dissertation.

Next, methods to improve training and performance of GMM MDNs were outlined. The performance improvements focused on a new type of GMM loss function that separates the mean estimation from the covariance. Further alterations to the GMM custom loss function were also outlined. An ANOVA analysis was performed to determine the best set of improvements to use for our bike dataset. The training methods remove some of the guess work for users when training GMM MDNs.

Finally to showcase both the framework and MDN improvements, position estimates were generated for a VLF navigation problem using different architectures outlined in the framework with ANNs trained with the MDN improvements. The input and output features for both problems, as well as the base ANN architecture and training, were defined before starting the problem. The first problem was discrete indoor navigation on the AFIT dataset and the second problem was continuous outdoor navigation on a bicycle.

First the AFIT dataset was analyzed. Measurement ANNs were trained to determine discrete position directly from VLF features. The measurement ANNs achieved an accuracy of 81.5% on the test dataset which was interspersed with the training set and 58.6% on the test set seven months later. Next, the measurement ANNs were integrated with a Discrete Particle Filter (DPF) to introduce simple transition dynamics to the problem. The DPF integration increased the accuracy on all sets by about 2%. In order to introduce more advanced dynamics, a state predict ANN was trained to predict state transitions based on the training data. Unfortunately this state predict ANN only learned a poor identity function and did not improve the accuracy when inserted as the dynamics for the DPF integration. Finally a complete filter ANN was trained to use both measurements and previous position probabilities to predict the next position. The complete filter ANN performed the best of all the integrations with an accuracy of 86.7% on test data collected during the training data and an accuracy of 63.8% seven months later.

Next, the bike dataset continuous position problem was analyzed. Measurement ANNs which could output 2D position probabilities were trained to predict 2D position using VLF measurements. The measurement ANNs used the best custom loss function determined from the previous section. The test set RMSE of the measurement ANNs was 67.72 m. Next these measurement ANNs were integrated with a Kalman filter in order to introduce simple bike dynamics to the problem. The Kalman filter integration reduced the RMSE to 63.28 m. Next a state predict ANN was trained to learn the bike dynamics and road traveled. However, the state predict ANN became biased to the center of the area of navigation and did not improve our solution when integrated into the Kalman filter (now an EKF). Finally, a complete filter ANN was trained to predict 2D continuous position from measurements, previous position, and previous position covariance. However, the complete filter ANN did not generalize well to the test data, because the complete filter ANN ignored the given position covariance and instead developed an internal covariance based on the training set which was not representative of the test set.

Thus, different types of integrations were explored in order to provide different solutions for the VLF signals navigation problem. Determining measurements would not be possible without measurement ANNs. Introducing simple dynamics improved solutions, but ANN provided dynamics did not improve the solution. Finally, complete filter ANNs replaced entire filters, producing estimates with all available measurements and previous state estimates.

Future Work

First for future work we propose implementing more examples of the ANN framework. This dissertation describes implementations of a few examples of the framework, but there were a few key examples that did not get completed. The first incomplete example would be to explore state correction, since state correction was not explicitly found in the literature but some examples were very close. Setting up this framework implementation would also make the other correction methods easier to implement. The next recommendation would be to explore the ANN as filter parts and eventually make a gain ANN as outlined in Section III. A dataset which lends itself well to this kind of problem would need to be found, since it does not seem like the large measurement space of VLF signals would be an appropriate test example.

This dissertation outlined the idea of Gradient Normalization (GN) for GMM custom loss functions. The concept was to normalize the relative contribution of the means and covariances. While experimenting GN, it became apparent that GN also fixed the absolute magnitude of each contribution. The fixed magnitude created problems in some experiments where large learning rates prevented training. Without choosing an appropriate learning rate for these examples, GN may have not worked at all. Thus some scheme where the relative difference becomes fixed but the absolute magnitude is still related to the original contributions should be developed and tested.

In both datasets the state predict ANNs did not perform well. The state predicts ANNs either learned a poor identity transformation or were heavily biased. The continuous state predict ANN estimated position but it could have estimated other navigation states or a combination of states. For example velocity or acceleration states could have been estimated which may have removed the position bias. Another possibility for both dataset would be to use very large number of time steps per training sequence. Longer training sequences may force the ANNs to learn the actual dynamics instead of the identity. Another possibility is using an entirely new architecture called attention networks. Recently these networks have improved performance for machine translation [10] applications and could show improvements for this application as well.

All the ANNs in this dissertation were trained using supervised learning with labeled training data. Collecting the training, validation and test data was a menial task spanned years and the indoor dataset required manual labeling. Also, as shown in the AFIT dataset, the training data can become outdated and require new collects to learn the changes in the environment. If some method to extract VLF features without supervised data and be robust to changes in the future were developed, it may be possible to use VLF signals in new locations without previous data collects. Such a method would work well in Simultaneous Localization And Mapping (SLAM) applications where new areas are explored and mapped at the same time.

Bibliography

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

[2] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from Data: a short course*. AMLBook, 2012. ISBN 1600490069 9781600490064.

[3] Ahmed, N. U. *Linear and nonlinear filtering for scientists and engineers*. World Scientific, 1998. ISBN 981-02-3609-3.

[4] Almonacid, F., P. J. Pérez-Higueras, Eduardo F. Fernández, and L. Hontoria. "A methodology based on dynamic artificial neural network for short-term forecasting of the power output of a PV generator". *Energy Conversion and Management*, 85:389–398, 2014. ISSN 01968904. URL http://dx.doi.org/10.1016/j.enconman.2014.05.090.

[5] Bensoussan, Alain. *Stochastic Control of Partially Observable Systems*. Cambridge Univ Press, 1992. ISBN 0521611970. URL http://books.google.com/books?id=g9SIhOCD3ooC{&}pgis=1.

[6] Bhattacharya, Shrabani, Sindhu Verma, and Siddhartha Mukhopadhyay. "Comparative Study of Variants of Kalman Filter aided by Artificial Neural Networks for Improved Kinematic State Estimation of Air-borne Vehicles". *Proceedings of the IEEE INDICON 2004. First India Annual Conference*, 310–313, 2004. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=1497761{&}isnumber=32114.

[7] Bishop, Christopher M. *Mixture Density Networks*. Technical report, Aston University, Birmingham UK, 1994. URL http://publications.aston.ac.uk/373/1/NCRG{_}94{_}004.pdf.

[8] Bousquet, Olivier and André Andre Elisseeff. "Stability and Generalization". *Journal of Machine Learning Research*, 2:499–526, 2002. ISSN 15324435. URL http://dl.acm.org/citation.cfm?id=944801{%}5Cnhttp://www.crossref.org/jmlr{_}DOI.html.

[9] Bridle, John S. J.S. “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition”. *Neurocomputing*, (C):227–236, 1990. URL http://www.springerlink.com/index/10.1007/978-3-642-76153-9{_}28.

[10] Chen, Mia Xu, Orhan Firat, Ankur Banpna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, NikiParmar, Mike Schuster, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. “The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation”. 1–12, 2018.

[11] Chen, S, S A Billings, and P M Grant. “Non-linear system identification using neural networks”. *Int. J. Control*, 51(6):1191–1214, 1990. ISSN 0020-7179.

[12] Chin, L. “Application of neural networks in target tracking data fusion”. *IEEE Transactions on Aerospace and Electronic Systems*, 30(1):281–287, 1994. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=250437{&}isnumber=6411.

[13] Cho, Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734, 2014. ISSN 09205691. URL http://arxiv.org/abs/1406.1078.

[14] Choi, Minyong, R. Sakthivel, and Wan Kyun Chung. “Neural network-aided extended Kalman filter for SLAM problem”. *Proceedings - IEEE International Conference on Robotics and Automation*, 1(April):1686–1690, 2007. ISSN 10504729.

[15] Dauphin, Yann N, Harm De Vries, Junyoung Chung, and Yoshua Bengio. “RMSProp and equilibrated adaptive learning rates for non-convex optimization”. *Cornell University Library*, 2014. URL https://arxiv.org/abs/1502.04390.

[16] van Diggelen, Frank and Per Enge. “The World’s first GPS MOOC and Worldwide Laboratory using Smartphones”. *Proceedings of the 28th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2015)*, 361–369. 2015. URL https://www.ion.org/publications/abstract.cfm?articleID=13079.

[17] Doucet, Arnaud, Nando de Freitas, and Neil Gordon. “An Introduction to Sequential Monte Carlo Methods”. *Sequential; Monte Carlo Methods in Practice*, 3–14, 2001. ISSN 0040-1706. URL http://link.springer.com/chapter/10.1007/978-1-4757-3437-9{_}1.

[18] Draganov, Alexandr. “FUNNEL : Filtering Using Neural Networks for Exactness and Leanness”. *Expedition Technology, Inc.*, 2017.

[19] Dreyfus, G, O Macchi, S Marcos, O Nerrand, L Personnaz, P Roussel-Ragot, D Urbani, and C Vignat. “Adaptive Training of Feedback Neural Networks for Non-Linear Filtering”. *Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop*, 550–9, 1992. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=253657.

[20] Elanayar, Vt and Yc Shin. “Radial basis function neural network for approximation and estimation of nonlinear stochastic dynamic systems”. *Neural Networks, IEEE Transactions*, 5(4):594–603, 1994. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=298229.

[21] Ferguson, Thomas S. “A Bayesian Analysis of Some Nonparametric Problems”. *The Annals of Statistics*, 1(2):209–230, 1973. ISSN 0090-5364. URL http://projecteuclid.org/euclid.aos/1176342360.

[22] Gers, F.A. and J. Schmidhuber. “Recurrent nets that time and count”. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 1:189–194 vol.3, 2000. ISSN 1098-6596. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=861302.

[23] Glorot, Xavier, Yoshua Bengio, and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. *Journal of Machine Learning Research*, 9:249–256, 2010. ISSN 1935-5130. URL http://machinelearning.wustl.edu/mlpapers/paper{_}files/AISTATS2010{_}GlorotB10.pdfhttp://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf.

[24] Graves, Alex. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012. ISBN 2000201075. URL https://www.cs.toronto.edu/{~}graves/preprint.pdf.

[25] Graves, Alex. “Generating Sequences with Recurrent Neural Networks”. *Technical Reports*, 1–43, 2013. ISSN 18792782. URL http://arxiv.org/abs/1308.0850.

[26] Graves, Alex, S Fernández, and J Schmidhuber. “Multi-dimensional recurrent neural networks”. *Artificial Neural Networks–ICANN*, 1(1):549—-558, 2007. ISSN 03029743. URL http://link.springer.com/chapter/10.1007/978-3-540-74690-4{_}56.

[27] Graybill, Franklin A. “Mixtures of Dirichlet processes with applications to Bayesian nonparametric problems”. *Annals of Statistics*, 2(6):1152–1174, 1974.

[28] Haarnoja, Tuomas, Anurag Ajay, Sergey Levine, and Pieter Abbeel. “Backprop KF: Learning Discriminative Deterministic State Estimators”. *NIPS 2016*. 2016. URL http://arxiv.org/abs/1605.07148.

[29] Hartigan, J. a. and M. a. Wong. “Algorithm AS 136: A K-Means Clustering Algorithm”. *Journal of the Royal Statistical Society C*, 28(1):100–108, 1979.

ISSN 00359254. URL http://www.jstor.org/stable/2346830{%}5Cnpapers2://publication/uuid/6437C437-7330-4249-BD7B-2642C9B2C6BE{%}5Cnhttp://www.jstor.org/stable/10.2307/2346830?origin=crossref.

[30] Hartigan, Jhon A. "Clustering Algorithms". *a Wiley Publication in Applied Statistics*, 1–351, 1975. ISSN 00401706. URL citeulike-article-id:9527640{%}5Cnhttp://dx.doi.org/10.1002/0471725382.scard.

[31] Hatsopoulos, N G and W H Warren. "Visual Navigation With a Neural Network". *Neural Networks*, 4(3):303–317, 1991. URL papers3://publication/uuid/B793BA34-547B-4E77-9D59-FCBDF92E0CA5.

[32] Haykin, Simon. *Kalman Filtering and Neural Networks*, volume 5. John Wiley & Sons, Inc., 2001. ISBN 0471369985. URL http://onlinelibrary.wiley.com/doi/10.1002/0471221546.fmatter{_}indsub/pdf.

[33] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity mappings in deep residual networks". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9908 LNCS:630–645, 2016. ISSN 16113349.

[34] He, Kamiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". *arXiv*, 1–12, 2015. ISSN 15737721.

[35] Hochreiter, Sepp, S Hochreiter, Jürgen Schmidhuber, and J Schmidhuber. "Long short-term memory." *Neural computation*, 9(8):1735–80, 1997. ISSN 0899-7667. URL http://www.ncbi.nlm.nih.gov/pubmed/9377276.

[36] Hornik, Kur, Maxwell Stinchcombe, and Halber White. "Multilayer Feedforward Networks are Universial Approximators". *Neural Networks*, 2:359–366, 1989.

[37] Horton, M. P. "Real-time identification of missile aerodynamics using a linearised Kalman filter aided by an artificial neural network". *IEEE Proceedings - Control Theory and Applications*, 144(4):299–308, 1997. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=610222{&}isnumber=13312.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org/.

[39] Iiguni, Y., H. Sakai, and H. Tokumaru. "A real-time learning algorithm for a multilayered neural network based on the extended Kalman filter". *IEEE Transactions on Signal Processing*, 40(4):959–966, 1992. ISSN 1053587X. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=127966.

[40] International Telecommunications Union. *Radio Regulation Articles*. Technical report, International Telecommunications Union, 2012.

[41] ITU-R. “CCIR - Characteristics and Applications of Atmospheric Radio Noise Data.PDF”, 1983.

[42] ITU-R. *RECOMMENDATION ITU-R P.372-8 Radio noise **. International Telecommunication Union, 2003. ISBN 1951195319561.

[43] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN 1461471370, 9781461471370.

[44] Japkowicz, Nathalie and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011. ISBN 9781139494144. URL https://play.google.com/store/books/details?id=VoWIIOKVzR4C.

[45] Jazwinski, Andrew H. *Stochastic processes and filtering theory*. Dover Publications, 2007. ISBN 9780486462745.

[46] Jordan, M.I. *The Learning of Representations for Sequential Performance*. University of California, San Diego, 1985. URL https://books.google.com/books?id=E-JdHQAACAAJ.

[47] Jwo, Dah-Jing, Chi-Shui Chang, and Chia-Hsin Lin. “Neural Network Aided Adaptive Kalman Filtering for GPS Applications”. *2004 IEEE International Conference on Systems, Man and Cybernetics Neural*, 3686–3691, 2004.

[48] Jwo, Dah-jing, Jyh-jeng Chen, and Guidance Engineering. “Neural network aided adaptive Kalman filter for GPS / INS navigation system design”. *9th IFAC Workshop ”ADAPTATION AND LEARNING IN CONTROL AND SIGNAL PROCESSING” (ALCOSP’07)*, 1–7, 2005.

[49] Jwo, Dah-Jing and H C Huang. “Neural network aided adaptive extended Kalman filtering approach for DGPS positioning”. *Navigation: Journal of The Institute of Navigation*, 57(3):449–463, 2004. ISSN 0373-4633.

[50] Kallianpur, Gopinath. *Stochastic Filtering Theory*. Springer, New York, 1980. ISBN 978-1-4419-2810-8.

[51] Kang, Jeong Gwan, Su Yong An, and Se Young Oh. “Modified neural network aided EKF based SLAM for improving an accuracy of the feature map”. *Proceedings of the International Joint Conference on Neural Networks*, 2010. ISSN 1098-7576.

[52] Kang, Jeong-Gwan, Sunhyo Kim, Su-Yong An, and Se-Young Oh. “A new approach to simultaneous localization and map building with implicit model learning using neuro evolutionary optimization”. *Applied Intelligence*, 36(1):242–269, 2012. ISSN 0924-669X.

[53] Katayama, Seiji. "Kalman Filtering Compensated by Radial Basis Function Neural Network for Seam Tracking of Laser Welding". *IEEE Transactions on Control Systems Technology*, 21(5):1916–1923, 2013. ISSN 1063-6536. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6353188.

[54] Kay, Steven M. *Fundamentals of Statistical Signal Processing Volume I, Estimation theory*, volume Volume I. Prentice Hall, 1993. ISBN 0071444742.

[55] Kay, Steven M. *Fundamentals of Statistical Signal Processing Volume II, Detection theory*, volume Volume II. 1993. ISBN 0071444742.

[56] Kingma, Diederik and Jimmy Ba. "Adam: A Method for Stochastic Optimization". *International Conference on Learning Representations*, 1–13, 2014. URL http://arxiv.org/abs/1412.6980.

[57] Krishnan, Rahul G., Uri Shalit, and David Sontag. "Deep Kalman Filters". *Cornell University Library*, 1(2):1–7, 2015. URL http://arxiv.org/abs/1511.05121.

[58] LeCun, Yann A., Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. "Efficient backprop". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:9–48, 2012. ISSN 03029743.

[59] Lee, Suk Jin, Yuichi Motai, and Martin Murphy. "Respiratory motion estimation with hybrid implementation of extended kalman filter". *IEEE Transactions on Industrial Electronics*, 59(11):4421–4432, 2012. ISSN 02780046.

[60] Li, Jinyu, Abdelrahman Mohamed, Geoffrey Zweig, and Yifan Gong. "LSTM time and frequency recurrence for automatic speech recognition". *2015 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2015 - Proceedings*, 187–191, 2016.

[61] Li, Peijuan, Xiaofei Zhang, and Xiaosu Xu. "Novel Terrain Integrated Navigation System using Neural Network aided Kalman Filter". *2010 Sixth International Conference on Natural Computation (ICNC 2010) Novel*, 1(6):445–448, 2010.

[62] Lipton, Zachary C, John Berkowitz, and Charles Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning". *Cornell University Library*, 2015. URL https://arxiv.org/abs/1506.00019.

[63] Ljung, Lennart and Jonas Sjoberg. "a System Identification Perspecitve on Neural Nets". *Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop*, 1992. URL http://ieeexplore.ieee.org/abstract/document/253670/.

[64] Mandic, Danilo P. "Data-Reusing Recurrent Neural Adaptive Filters". *Neural Computation*, 14:2693–2707, 2002. ISSN 0899-7667.

[65] Maybeck, Peter S. *Stochastic Models, Estimation and Control: Volume 1*. Navtech Book & Software Store, Arlington, VA, 1994.

[66] Maybeck, Peter S. *Stochastic Models, Estimation and Control: Volume 2*. Navtech Book & Software Store, Arlington, VA, 1994.

[67] Misra, Pratap. *Global positioning system : signals, measurements, and performance*. Ganga-Jamuna Press, 2012. ISBN 978-0970954428.

[68] Moon, Seung Ill, Ali Keyhani, and Srinivas Pillutla. “Nonlinear neural-network modeling of an induction machine”. *IEEE Transactions on Control Systems Technology*, 7(2):203–211, 1999. ISSN 10636536.

[69] Morris, Guy V. *Airborne pulsed Doppler radar*. 1988. ISBN 0890068674.

[70] Narendra, K S and K Parthasarathy. “Identification and control of dynamical systems using neural networks.” *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 1(1):4–27, 1990. ISSN 1045-9227.

[71] Nerrand, O and P Roussel-Ragot. “Neural networks and nonlinear adaptive filtering: unifying concepts and new algorithms”. *Neural Computation*, 5:1–18, 1993. ISSN 0899-7667. URL http://www.mitpressjournals.org/doi/abs/10.1162/neco.1993.5.2.165.

[72] Ngia, L S H and J. Sjoberg. “Efficient training of neural nets for nonlinear adaptive filtering using a recursive Levenberg-Marquardt algorithm”. *IEEE Transactions on Signal Processing*, 48(7):1915–1927, 2000. ISSN 1053587X. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=847778.

[73] Oja, Erkki. “Principal Compnents, Minor Components, and Linear Neural Networks”. *Neural Networks*, 5:927–935, 1992.

[74] Oksendal, Bernt, B Oksendal, and B K Ksendal. *Stochastic differential equations: an introduction with applications*, volume 5. 1992. ISBN 9783642143946.

[75] Oppenheim, Alan V and George C Verghese. *SIGNALS, SYSTEMS, and INFERENCE*. MIT Press, 2010. URL http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Signals,+systems+and+inference{#}8.

[76] Parlos, Alexander G. “Adaptive Filtering Using Recurrent Neural Networks Precise system models are not needed for state filtering .” *Information Sciences*, 1(May):2005, 2005.

[77] Parlos, Alexander G., Sunil K Menon, and Amir F Atiya. “An Algorithmic Approach to Adaptive State Filtering using Recurrent Neural Networks”. *Neural Networks, IEEE Transactions on*, 12(6):1411–32, 2001. ISSN 1045-9227. URL http://www.ncbi.nlm.nih.gov/pubmed/18249970.

[78] Plesa, James. *Loop Antenna Specifications*. Private Communications, 2015.

[79] Puskorius, Gintaras V. and Lee A. Feldkamp. "Neurocontrol of Nonlinear Dynamical Systems with Kalman Filter Trained Recurrent Networks". *IEEE Transactions on Neural Networks*, 5(2):279–297, 1994. ISSN 19410093.

[80] Rajagopal, K. and N. Pappa. "State Estimation of a Nonlinear System by Neural Extended Kalman Filter". *IEEE*, 1–6, 2006. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=4086223{%}5Cnhttp://ieeexplore.ieee.org/ielx5/4086217/4086218/04086223.pdf?tp={&}arnumber=4086223{&}isnumber=4086218.

[81] Rajesh, M.V., R. Archana, a. Unnikrishnan, and R. Gopikakaumari. "Particle filter based neural network modeling of nonlinear systems for state space estimation". *2009 Chinese Control and Decision Conference*, 1477–1482, 2009.

[82] Raquet, J and R K Martin. "Non-GNSS radio frequency navigation". *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, 5308–5311. mar 2008. ISSN 1520-6149.

[83] Rasmussen, C.˜E. "The infinite Gaussian mixture model". *Advances in Neural Information Processing Systems 12*, (1):554–560, 2000.

[84] Sak, HaÅim, Andrew Senior, and Françoise Beaufays. "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling". *Interspeech 2014*, 1(September):338–342, 2014. URL http://193.6.4.39/{~}czap/letoltes/IS14/IS2014/PDF/AUTHOR/IS141304.PDF.

[85] Sastry, P. S., G. Santharam, and K. P. Unnikrishnan. "Memory Neuron Networks for Identification and Control of Dynamical Systems". *IEEE Transactions on Neural Networks*, 5(2):306–319, 1994. ISSN 19410093.

[86] Schuster, Mike. "Better Generative Models for Sequential Data Problems: Bidirectional Recurrent Mixture Density Networks". *Advances in Neural Information Processing Systems 12*, 589–595, 2000. ISSN 10495258. URL http://papers.nips.cc/paper/1778-better-generative-models-for-sequential-data-problems-bidirectional-recurrent-mixture-density-networks.pdf{%}5Cnfiles/5768/Schuster-2000-BetterGenerativeModelsforSequentialDataProbl.pdf{%}5Cnfiles/5769/1778-better-ge.

[87] Seeger, Matthias. *Gaussian processes for machine learning.*, volume 14. MIT Press, 2004. ISBN 026218253X. URL http://www.gaussianprocess.org/gpml/chapters/RW.pdf.

[88] Shamos, Michael Ian and Dan Hoey. "Closest-point problems". *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, 151–162, 1975. ISSN 02725428. URL http://ieeexplore.ieee.org/document/4567872/.

[89] Singhal, Shared and Lance Wu. “Training Multilayer Perceptrons with the Extended Kalman Algorithm”. *NIPS*, 133–140, 1988.

[90] Spaulding, AD and JS Washburn. “Atmospheric radio noise: Worldwide levels and other characteristics”. (April), 1985. URL http://www.its.bldrdoc.gov/publications/download/85-173.pdf.

[91] Srinivasan, K. and J. Prakash. “Non-linear State Estimation for Continuous Stirred Tank Reactor using Neural Network State Filter”. *2006 Annual IEEE India Conference*, 1–4, 2006. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4086231.

[92] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *Journal of Machine Learning Research*, 15:1929–1958, 2014. ISSN 15337928.

[93] Stubberud, Stephen C, Robert N Lobbia, and Mark Owen. “An Adaptive Extended Kalman Filter using Artificial Neural Networks”. *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, (December):1852–6, 1995.

[94] Stubberud, Stephen C, Senior Member, and Kathleen A Kramer. “System Identification Using the Neural-Extended Kalman Filter For Control Modification”. 8882–8888, 2006. ISSN 10987576.

[95] Sutskever, Ilya, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. *NIPS*, 3104–3112, 2014. ISSN 09205691.

[96] Szilas, N. and C. Cadoz. “Adaptive networks for physical modeling”. *Neurocomputing*, 20(1-3):209–225, 1998. ISSN 09252312. URL http://www-acroe.imag.fr/.

[97] Talebi, H. A., K. Khorasani, and S. Tafazoli. “A recurrent neural-network-based sensor and actuator fault detection and isolation for nonlinear systems with application to the satellite’s attitude control subsystem”. *IEEE Transactions on Neural Networks*, 20(1):45–60, 2009. ISSN 10459227.

[98] Teissier, P., B. Perret, E. Latrille, J. M. Barillere, and G. Corrieu. “A hybrid recurrent neural network model for yeast production monitoring and control in a wine base medium”. *Journal of Biotechnology*, 55(3):157–169, 1997. ISSN 01681656.

[99] Ting-Ho Lo, J. “Synthetic approach to optimal filtering.” *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 5(5):803–811, 1994.

[100] Vaidehi, F., N. Chitra, C.N. Krishnan, and M. Chokkalingam. “Tracking Applications”. *Proceedings of the 1999 IEEE Radar Conference. Radar into the Next Millennium (Cat. No.99CH36249)*, 160–165, 1999.

[101] Wang, Zhou and Alan C. Bovik. “Mean squared error: Love it or leave it? A new look at signal fidelity measures”. *IEEE Signal Processing Magazine*, 26(1):98–117, 2009. ISSN 10535888.

[102] Widrow, Bernard. *Adaptive Signal Processing*. Prentice Hall, 1985. ISBN 0130040290.

[103] Williams, Peter M. “Using Neural Networks to Model Conditional Multivariate Densities”. *Neural Computation*, 8(4):843–854, 1996. ISSN 08997667.

[104] Wilson, D. Randall and Tony R. Martinez. “The general inefficiency of batch training for gradient descent learning”. *Neural Networks*, 16(10):1429–1451, 2003. ISSN 08936080.

[105] Wilson, Robert C. and Lh Finkel. “A neural implementation of the Kalman filter”. *Advances in neural information* ..., 22:2062–2070, 2009. URL http://books.nips.cc/papers/files/nips22/NIPS2009{_}0920.pdf{%}5Cnhttp://machinelearning.wustl.edu/mlpapers/paper{_}files/NIPS2009{_}0920.pdf.

[106] Xu, Fan and Jiancheng Fang. “Velocity and position error compensation using strapdown inertial navigation system/celestial navigation system integration based on ensemble neural network”. *Aerospace Science and Technology*, 12(4):302–307, 2008. ISSN 12709638.

[107] Yao, Kaisheng, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. “Depth-Gated LSTM”. 1–6, 2015. URL http://arxiv.org/abs/1508.03790.

[108] Yu, Zhi-Jun, Shao-Long Dong, Jian-Ming Wei, Tao Xing, and Hai-Tao Liu. “Neural Network Aided Unscented Kalman Filter for Maneuvering Target Tracking in Distributed Acoustic Sensor Networks”. *Proc. IEEE International Conference on Computing: Theory and Applications*, 245–249, 2007.

[109] Zekić-Sušac, Marijana, Nataša Šarlija, and Sanja Pfeifer. “Combining Pca Analysis and Artificial Neural Networks in Modelling Entrepreneurial Intentions of Students”. *Croatian Operational Research Review (CRORR)*, 4(4):306–317, 2013.

[110] Zhan, Ronghui and Jianwei Wan. “Neural network-aided adaptive unscented Kalman filter for nonlinear state estimation”. *IEEE Signal Processing Letters*, 13(7):445–448, 2006. ISSN 10709908.

Appendix: Appendix

The Appendix contains extra sections that expand on derivation or definitions in the body of the dissertation. Also included are more statistics on the datasets and results.

Probability NaN Derivation

This section contains a longer derivation of how a NaN shows up in a GMM loss function. When the algorithm calculates the derivate of $-\ln(0)$ it will make a calculation similar to:

$$\frac{\partial}{\partial x} - \ln(p(x)) = -\frac{\frac{\partial}{\partial x} p(x)}{p(x)} \tag{160}$$

Next we substitute the probability for the sum of exponentials raised to some function which is the form of a Gaussian mixtures distribution probability from Equation 58:

$$\begin{aligned} p(x) &= \sum_{j=1}^{M} \exp(f^j(x^j)) \\ \frac{\partial}{\partial x} - \ln(p(x)) = & -\frac{\frac{\partial}{\partial x} \sum_{j=1}^{M} \exp(f^j(x^j))}{\sum_{j=1}^{M} \exp(f^j(x^j))} \\ \frac{\partial}{\partial x} - \ln(p(x)) = & -\frac{\sum_{j=1}^{M} \exp(f^j(x^j)) \frac{\partial}{\partial x} f^j(x^j)}{\sum_{j=1}^{M} \exp(f^j(x^j))} \end{aligned}$$

Next we note that if the total probability is zero and all the probabilities are greater than or equal to zero, thus all the probabilities must be zero:

$$\begin{aligned}
& & \exp(f^j(x^j)) = 0 \forall j \in \{1 \ldots M\} \\
\frac{\partial}{\partial x} - \ln(p(x)) &= & -\frac{\sum_{j=1}^{M} 0 \frac{\partial}{\partial x} f^j(x^j)}{\sum_{j=1}^{M} 0} \\
\frac{\partial}{\partial x} - \ln(p(x)) &= & \frac{0}{0} \\
\frac{\partial}{\partial x} - \ln(p(x)) &= & NaN
\end{aligned}$$

which results in the partial derivative being Not a Number or NaN.

Probability Simplification Steps

This section contains the derivation to calculate the derivatives of GMM loss functions for the probability simplification. First, the exponent of a Gaussian distribution is denoted as $\boldsymbol{e}$:

$$\boldsymbol{e}(\mathbf{y}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \left(-\frac{1}{2}(\mathbf{y} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu}) - \ln\left((2\pi)^{D/2} |\boldsymbol{\Sigma}|^{1/2}\right)\right) \tag{161}$$

The derivative of a single Gaussian distribution is then:

$$\begin{aligned}
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= & -\frac{\partial \ln\left(w_0 \exp\left(\boldsymbol{e}_0\right)\right)}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= & -\frac{\partial \ln\left(\exp\left(\boldsymbol{e}_0 + \ln(w_0)\right)\right)}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= & -\frac{\partial\left(\boldsymbol{e}_0 + \ln(w_0)\right)}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= & -\frac{\partial \boldsymbol{e}_0}{\partial \boldsymbol{e}}
\end{aligned}$$

Now expanding to the a mixture of Gaussian distributions the derivative is:

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\frac{\partial \ln\left(\sum_{j=0}^{M-1} w_j \exp\left(\boldsymbol{e}_j\right)\right)}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\frac{\partial \ln\left(\sum_{j=0}^{M-1} \exp\left(\boldsymbol{e}_j + \ln(w_j)\right)\right)}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\frac{\frac{\partial\left(\sum_{j=0}^{M-1} \exp\left(\boldsymbol{e}_j + \ln(w_j)\right)\right)}{\partial \boldsymbol{e}}}{\sum_{k=0}^{M-1} \exp\left(\boldsymbol{e}_k + \ln(w_k)\right)} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\frac{\sum_{j=0}^{M-1} \frac{\partial \exp\left(\boldsymbol{e}_j + \ln(w_j)\right)}{\partial \boldsymbol{e}}}{\sum_{k=0}^{M-1} \exp\left(\boldsymbol{e}_k + \ln(w_k)\right)} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\frac{\sum_{j=0}^{M-1} \exp\left(\boldsymbol{e}_j + \ln(w_j)\right) \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}}}{\sum_{k=0}^{M-1} \exp\left(\boldsymbol{e}_k + \ln(w_k)\right)} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\sum_{j=0}^{M-1} \frac{\exp\left(\boldsymbol{e}_j + \ln(w_j)\right)}{\sum_{j=0}^{M-1} \exp\left(\boldsymbol{e}_k + \ln(w_k)\right)} \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &= -\sum_{j=0}^{M-1} \frac{\exp\left(\boldsymbol{e}_j + \ln(w_j)\right)}{\sum_{k=0}^{M-1} \exp\left(\boldsymbol{e}_k + \ln(w_k)\right)} \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}}
\end{aligned}
$$

Now realize that this can conveniently be written as a softmax function across all exponents:

$$
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} = -\sum_{j=0}^{M-1} \operatorname{softmax}\left(\boldsymbol{e}_j + \ln(w_j)\right) \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}} \tag{162}
$$

Next we assume that logarithms distribute over summations and take the derivative of the resulting loss function:

$$
\begin{aligned}
\mathcal{L}_{GMM} &= -\ln\left(\sum_{j=0}^{M-1} w_j \exp\left(\boldsymbol{e}_j\right)\right) \\
\mathcal{L}_{GMM} &\approx -\sum_{j=0}^{M-1} \ln\left(\exp\left(\boldsymbol{e}_j + \ln(w_j)\right)\right) \\
\mathcal{L}_{GMM} &\approx -\sum_{j=0}^{M-1} \boldsymbol{e}_j + \ln(w_j) \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &\approx -\sum_{j=0}^{M-1} \frac{\partial}{\partial \boldsymbol{e}}\left(\boldsymbol{e}_j + \ln(w_j)\right) \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{e}} &\approx -\sum_{j=0}^{M-1} \frac{\partial \boldsymbol{e}_j}{\partial \boldsymbol{e}}
\end{aligned}
$$

Thus we have calculated the derivative for various GMM loss functions.

GMM Loss Derivations

This section contains the derivative of a GMM loss function with respect to the means. From [103] we note the summation form of the derivative of one of the Gaussian mixtures as:

$$\begin{aligned}
\eta_d &= \mu_d - y_d \\
\xi_j &= \sum_{d=j}^{n} A_{jd} \left(\mu_d - y_d\right) \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \mu_i} &= \sum_{j=1}^{i} \xi_j A_{ji}
\end{aligned}$$

we rewrite the derivative of one of the Gaussian mixtures using matrix multiplication instead of using summations:

$$\begin{aligned}
\boldsymbol{\xi} &= \mathbf{A}\boldsymbol{\eta} \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} &= \left(\boldsymbol{\xi}^T \mathbf{A}\right)^T \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} &= \left((\mathbf{A}\boldsymbol{\eta})^T \mathbf{A}\right)^T \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} &= \left(\boldsymbol{\eta}^T \mathbf{A}^T \mathbf{A}\right)^T \\
\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} &= \mathbf{A}^T \mathbf{A}\boldsymbol{\eta}
\end{aligned}$$

and substituting using equation 64

$$\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1}\boldsymbol{\eta} \tag{163}$$

and then substitute for $\boldsymbol{\eta}$

$$\frac{\partial \mathcal{L}_{GMM}}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{\mu} - \mathbf{y}\right) \tag{164}$$

Mahalanobis Distance

This section outlines Mahalanobis distance as a metric to evaluate the accuracy of a covariance estimate. This metric was not used in the dissertation but may be useful for other applications involving GMM MDN where accuracy of covariance must be described beyond one single number.

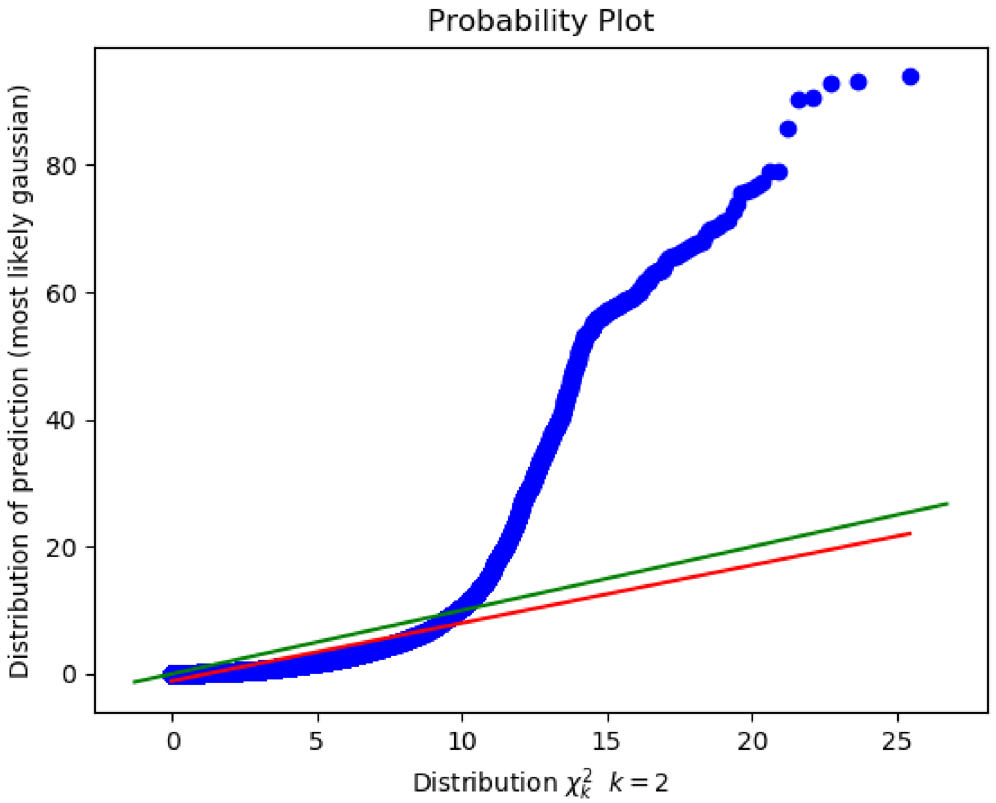

Figure 41: Q-Q plot of Mahalanobis distance squared vs Chi squared k=2 distribution. Green line is reference 45 degree line. Red line is best fit line to quantile data. Data comes from the bike dataset best GMM loss measurement ANN with no metadata

In order to better quantify where the MDN makes accurate/poor covariance estimates, we introduce a visual method to evaluate covariance. For a single Gaussian distribution that attempts to estimate a true value y with mean μ and variance σ^2 we can calculate the probability of the target point y evaluated with the parameters μ, σ^2. However, the probability includes the error or the distance the mean was from the target as well as the variance. If we wanted to single out the variance we would require more samples of y, which we could then compute a sample variance in order to determine if the estimated variance is accurate.

In our GMM examples however, we only have one sample per estimate of mean and variance. To compare across different Gaussian distributions, we scale the error by the variance to bring the estimates back to a assumed standard normal distribution with $\sigma^2 = 1$. Next the Mahalanobis distance is calculated for each estimate in the distribution. The Mahalanobis distance is a multi-dimensional generalization of how many standard deviations away our target y is from the mean μ. Mahalanobis distance can be thought of like a hill climbing distance where it measures how much effort it takes to get to the top of the hill (the mean) from a given point on the hill (our estimate). The steeper the climb the larger the Mahalanobis distance and the more standard deviations away from the mean the point was.

We will use the fact that Mahalanobis distance squared is distributed like a χ^2_k distribution with k degrees of freedom if the observations were normally distributed. We can plot the Mahalanobis distance squared against the χ^2_k on a Q-Q plot. The Q-Q plot is useful to visualize the differences between two distributions. It can specifically show when distributions have more or less variance compared to each other.

For example we use the bike dataset test results of the best GMM loss measurement ANN with no metadata to show the Mahalanobis distance squared Q-Q plot in Figure 41. In this plot the estimated distribution overall has a similar variance to a Chi squared distribution since the best red fit line is close to the desired green 45 degree line. However there are a few points with larger distance which contain much more variance compared to a Chi squared distribution. Thus this plot could be interpreted as the estimates of variance are accurate when the Mahalanobis distance is under 10. This corresponds to a sigma value of about 3, so we could say our estimates are accurate up to the 3 sigma point of the distribution. When points are more than 3 sigma away, our variance estimates become too large compared to the true variance.

Thus the Mahalanobis distance Q-Q plot informed the user to the how accurate the variance estimates were and where the variance estimates were incorrect. This is much more than a single number could provide.

AFIT Dataset information

This section lists supplementary information for the AFIT dataset including number of runs for each set and class distributions.

Table 14: Number of samples of each location bin in the AFIT Jan-Apr training set. Each sample is for one 100ms window.

Location Bin Name	Samples Per Bin
646 Rm 212 (0)	8285
646 Rm 214 (1)	6805
641-642 Hallway (2)	9404
646 Rm 216 (3)	8886
641 Lobby North Hallway (4)	9119
646 Rm 218 (5)	11566
640 Rm 240 (6)	8157
640 Rm 242 (7)	14581
640 Rm 244 (8)	10963
641 Lobby (9)	8848
641 Lobby North Hallway Diagonal (10)	6209
641 North Hallway (11)	15193
646 Hallway (12)	28137
640 Stairs East Hallway (13)	11515
640 Stairs North Hallway (14)	10064
640 Middle Hallway (15)	17412
640 South Hallway (16)	29247
640 West Hallway (17)	12748
641 Lobby East Hallway (18)	10468
640 Bane Lobby (19)	12923

642 Kenney Lobby (20)	21646
640 Stairs South Hallway (21)	7323
640 Stairs West Hallway (22)	10840
640-646 North Skywalk (23)	6828
640 Rm 232 (24)	8046
640 Rm 234 (25)	11339
640 Rm 236 (26)	4945
640 North Stairs (27)	6109
640 Bane Hall (28)	9200
640 East Hallway (29)	19546
640 North Hallway (30)	24413
640-646 South Skywalk (31)	6940
642 South Hallway (32)	14791
646 Rm 220 (33)	5759
642 Locker Hallway (34)	12685
642 East Lobby (35)	6363
642 North Hallway (36)	17719
646 South Hallway (37)	2826
Einstein's Bagel Shop (38)	21652
642 Kenney Auditorium (39)	15663
640 ANT Lab (40)	26272
641 East Hallway (41)	4925
640 South Stairs (42)	5048
642 East Hallway (43)	12051
642 Library Hallway (44)	8390
642 Kenney Exit (45)	6228
646 North Hallway (46)	4949
640 Doolittles (47)	19509
640 Rm 222 (48)	15761
640 Rm 220 (49)	13524

AFIT Dataset Result Tables

This section lists the detailed results of the AFIT experiment. There were a total of four experiments which included a measurement ANN, measurement ANN DPF integration, measurement ANN and state predict ANN integration, and finally a complete filter ANN. Results are included for all sets for both no metadata and metadata. Note the

Table 15: Number of runs in each of the AFIT datasets. Each run was about 45 minutes long. Note that hardware difficulties at the beginning of the data collection process prevented many early runs from containing metadata.

Dataset	No Metadata	Metadata
Jan-Apr Training	24	19
Jan-Apr Validation	22	19
Jan-Apr Test	21	17
May	6	6
July	2	2
August	9	9
September	9	9
October	9	9
November	9	9
December	7	7

measurement ANN table is a repeat of Table 4 included for comparison against the other tables.

Table 16: Performance comparison of the AFIT Map measurement ANN without metadata and with metadata.

No Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	93.6%	0.934	2.47%
Jan-Apr Validation	76.6%	0.760	2.06%
Jan-Apr Test	79.6%	0.790	2.59%
May	75.2%	0.746	4.43%
July	64.3%	0.633	9.41%
August	70.2%	0.694	4.41%
September	66.1%	0.651	4.07%
October	68.6%	0.678	4.49%
November	61.5%	0.605	4.47%
December	58.6%	0.575	4.96%

Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	95.0%	0.948	2.72%
Jan-Apr Validation	78.7%	0.782	2.77%
Jan-Apr Test	81.5%	0.810	2.86%
May	75.1%	0.744	5.51%
July	63.3%	0.674	9.41%
August	72.5%	0.718	4.41%
September	71.2%	0.703	4.07%
October	69.9%	0.692	4.49%
November	59.1%	0.580	4.47%
December	57.5%	0.564	4.96%

Table 17: Performance comparison of the AFIT measurement ANN integrated with a Discrete Particle Filter (DPF). The DPF uses a state transition matrix to propagate. Results listed for integration with the measurement ANN without metadata and with metadata.

No Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	93.6%	0.946	2.47%
Jan-Apr Validation	78.5%	0.779	2.06%
Jan-Apr Test	81.3%	0.807	2.59%
May	77.6%	0.770	4.43%
July	66.2%	0.653	9.41%
August	72.3%	0.715	4.41%
September	67.6%	0.664	4.07%
October	70.4%	0.696	4.49%
November	63.7%	0.627	4.47%
December	61.1%	0.601	4.96%

Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	96.1%	0.960	2.72%
Jan-Apr Validation	80.9%	0.804	2.77%
Jan-Apr Test	83.3%	0.828	2.86%
May	77.4%	0.768	5.51%
July	70.4%	0.696	9.41%
August	74.5%	0.738	4.41%
September	72.9%	0.720	4.07%
October	72.3%	0.717	4.49%
November	62.0%	0.609	4.47%
December	60.1%	0.591	4.96%

Table 18: Performance comparison of the measurement ANN integrated with a discrete particle filter that used a state predict ANN to propagate the states. Results listed for integration with the measurement ANN without metadata and with metadata.

No Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	87.8%	0.875	2.47%
Jan-Apr Validation	74.8%	0.741	2.61%
Jan-Apr Test	76.8%	0.761	2.59%
May	72.3%	0.715	4.43%
July	59.1%	0.579	9.41%
August	67.6%	0.667	4.41%
September	60.4%	0.593	4.07%
October	65.3%	0.644	4.49%
November	57.9%	0.567	4.47%
December	56.4%	0.552	4.96%

Metadata			
Dataset	Accuracy	Cohen's Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	89.5%	0.892	2.72%
Jan-Apr Validation	76.2%	0.755	2.77%
Jan-Apr Test	78.7%	0.780	2.86%
May	72.5%	0.717	5.51%
July	62.7%	0.617	9.41%
August	69.8%	0.690	4.41%
September	63.0%	0.620	4.07%
October	65.8%	0.649	4.49%
November	58.0%	0.568	4.47%
December	56.4%	0.552	4.96%

Table 19: Performance of the AFIT Map Filter ANN without and with metadata.

No Metadata			
Dataset	Accuracy	Cohen’s Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	96.0%	0.959	2.72%
Jan-Apr Validation	80.5%	0.800	2.77%
Jan-Apr Test	83.8%	0.834	2.86%
May	76.8%	0.763	5.15%
July	66.1%	0.652	9.41%
August	73.1%	0.724	4.41%
September	71.5%	0.706	4.07%
October	70.2%	0.695	4.49%
November	63.2%	0.623	4.47%
December	63.8%	0.629	4.96%

Metadata			
Dataset	Accuracy	Cohen’s Kappa	Acc. 95 % Confidence Interval
Jan-Apr Training	97.6%	0.975	2.72%
Jan-Apr Validation	83.4%	0.830	2.77%
Jan-Apr Test	86.7%	0.863	2.86%
May	81.8%	0.814	5.51%
July	67.7%	0.669	9.41%
August	76.8%	0.762	4.41%
September	74.8%	0.740	4.07%
October	75.7%	0.751	4.49%
November	65.3%	0.644	4.47%
December	62.7%	0.617	4.96%

ANOVA Raw Data

This section contains the raw data from the ANOVA analysis. Note the columns with N/A in GCS and GN were the classic GMM loss functions without and with PS respectively.

GCS	GN	PS	CLS	Combined	RMSE (m)	NCE
N/A	N/A	FALSE	1	45.772995	69.726103	0.228782
N/A	N/A	TRUE	1	0.188199	85.662322	0.233288
TRUE	FALSE	FALSE	1	-0.265811	80.619274	0.238526
TRUE	FALSE	FALSE	10	-1.309523	70.695612	0.321132
TRUE	FALSE	FALSE	100	-1.320229	70.33945	0.360994
TRUE	FALSE	FALSE	1000	-1.26566	70.685172	0.372704
TRUE	FALSE	FALSE	10000	-1.138036	72.202244	0.405604
TRUE	FALSE	FALSE	100000	-0.673512	77.215511	0.426425
TRUE	FALSE	FALSE	1000000	-1.034788	73.110566	0.44665
TRUE	FALSE	FALSE	10000000	-0.74423	75.963637	0.449248
TRUE	FALSE	FALSE	100000000	-0.79609	75.019656	0.455714
TRUE	FALSE	FALSE	1000000000	-1.480262	67.935236	0.467144
TRUE	TRUE	FALSE	1	-1.492374	70.147922	0.472635
TRUE	TRUE	FALSE	10	1.636828	101.553251	0.484346
TRUE	TRUE	FALSE	100	-1.451049	69.567034	0.512125
TRUE	TRUE	FALSE	1000	-0.935367	74.836686	0.512675
TRUE	TRUE	FALSE	10000	0.922616	93.781834	0.513197
TRUE	TRUE	FALSE	100000	0.574057	88.456586	0.532787
TRUE	TRUE	FALSE	1000000	0.518785	89.562357	0.546051
TRUE	TRUE	FALSE	10000000	0.559771	89.845495	0.564287
TRUE	TRUE	FALSE	100000000	1.322013	96.050865	0.564997
TRUE	TRUE	FALSE	1000000000	1.430566	97.844571	0.573388
FALSE	FALSE	FALSE	1	1.105425	95.141747	0.583356
FALSE	FALSE	FALSE	10	-0.548162	77.631919	0.583541
FALSE	FALSE	FALSE	100	-0.894666	75.46416	0.590837
FALSE	FALSE	FALSE	1000	-0.877185	74.624161	0.595601
FALSE	FALSE	FALSE	10000	-0.814929	75.3642	0.595931
FALSE	FALSE	FALSE	100000	-0.938939	73.97075	0.60848
FALSE	FALSE	FALSE	1000000	-1.026453	73.038954	0.616812
FALSE	FALSE	FALSE	10000000	-0.946043	73.797783	0.621135
FALSE	FALSE	FALSE	100000000	-1.215495	71.320243	0.621453
FALSE	FALSE	FALSE	1000000000	-1.519742	67.77992	0.628468
FALSE	TRUE	FALSE	1	-1.211159	70.38245	0.641903
FALSE	TRUE	FALSE	10	-1.595653	69.115295	0.671211
FALSE	TRUE	FALSE	100	-1.214236	72.406012	0.672203
FALSE	TRUE	FALSE	1000	0.305777	86.461644	0.674786
FALSE	TRUE	FALSE	10000	0.542004	88.795275	0.675812
FALSE	TRUE	FALSE	100000	0.191324	86.487612	0.683521
FALSE	TRUE	FALSE	1000000	0.39636	87.347349	0.691934

FALSE	TRUE	FALSE	10000000	0.114492	85.123423	0.692622
FALSE	TRUE	FALSE	100000000	0.053337	84.561007	0.70347
FALSE	TRUE	FALSE	1000000000	-0.120242	80.835561	0.707005
TRUE	FALSE	TRUE	1	2.775221	69.14568	0.713375
TRUE	FALSE	TRUE	10	4.578785	67.188347	0.722834
TRUE	FALSE	TRUE	100	-0.282658	74.503952	0.748598
TRUE	FALSE	TRUE	1000	-0.852359	70.037958	0.786926
TRUE	FALSE	TRUE	10000	0.791763	75.313927	0.830866
TRUE	FALSE	TRUE	100000	-0.519984	71.511894	0.844774
TRUE	FALSE	TRUE	1000000	-1.155973	70.157876	0.936413
TRUE	FALSE	TRUE	10000000	762.323852	71.588184	0.950627
TRUE	FALSE	TRUE	100000000	10.95005	68.617037	1.050585
TRUE	FALSE	TRUE	1000000000	2.169566	65.126116	1.065419
TRUE	TRUE	TRUE	1	7.800259	68.244478	1.537873
TRUE	TRUE	TRUE	10	1.196377	66.452154	1.620042
TRUE	TRUE	TRUE	100	0.532502	72.307573	1.659819
TRUE	TRUE	TRUE	1000	0.529078	84.584611	1.903883
TRUE	TRUE	TRUE	10000	0.963217	73.529916	2.020177
TRUE	TRUE	TRUE	100000	0.133006	69.617398	2.382587
TRUE	TRUE	TRUE	1000000	0.148753	68.469452	3.12183
TRUE	TRUE	TRUE	10000000	7.384288	72.310681	3.145951
TRUE	TRUE	TRUE	100000000	3.845222	63.023289	3.395685
TRUE	TRUE	TRUE	1000000000	0.73483	65.28342	3.799097
FALSE	FALSE	TRUE	1	1.599356	97.898703	4.016963
FALSE	FALSE	TRUE	10	-0.146745	77.073715	4.070446
FALSE	FALSE	TRUE	100	1.527519	75.162006	4.089652
FALSE	FALSE	TRUE	1000	1.384543	73.30684	4.123112
FALSE	FALSE	TRUE	10000	0.054194	71.961493	4.750703
FALSE	FALSE	TRUE	100000	0.447322	76.104728	5.562708
FALSE	FALSE	TRUE	1000000	-0.454622	76.788024	5.61043
FALSE	FALSE	TRUE	10000000	-0.14643	73.589647	5.691395
FALSE	FALSE	TRUE	100000000	0.521506	75.543265	5.974995
FALSE	FALSE	TRUE	1000000000	1.248092	69.290493	6.234133
FALSE	TRUE	TRUE	1	0.442774	71.227033	6.712612
FALSE	TRUE	TRUE	10	0.767009	67.602282	9.053484
FALSE	TRUE	TRUE	100	1.487359	98.378512	9.50618
FALSE	TRUE	TRUE	1000	0.771749	93.386564	13.046662
FALSE	TRUE	TRUE	10000	1.615113	99.868352	13.740747
FALSE	TRUE	TRUE	100000	0.900425	92.992929	18.635074
FALSE	TRUE	TRUE	1000000	0.974199	95.014701	20.360916
FALSE	TRUE	TRUE	10000000	1.018826	94.740834	26.970581
FALSE	TRUE	TRUE	100000000	1.393434	97.246486	100.668437
FALSE	TRUE	TRUE	1000000000	0.874345	93.342547	1621.491315

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From — To)*
13–09–2018	Doctoral Dissertation	Oct 2015–Sep 2018

4. TITLE AND SUBTITLE	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
Navigation with Artificial Neural Networks	
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
Curro II, Joseph A., Captain, USAF	
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	**8. PERFORMING ORGANIZATION REPORT NUMBER**
Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB, OH 45433-7765	AFIT-ENG-DS-18-S-007
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
Intentionally Left Blank	**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

The objective of this dissertation is to explore the applications for Artificial Neural Networks (ANNs) in the field of Navigation. The state of the art for ANNs has improved significantly so now they can rival and even surpass humans in problems once thought impossible. We present different methods to augment, combine, or replace existing Navigation filters with ANN. The main focus of these methods is to use as much existing knowledge as possible then use ANNs to extend the current knowledge base. Next, improvements are made for a class of ANNs which provide covariance called MDNs. MDNs are necessary since covariance is required for navigation problems. Finally the improvements and framework are demonstrated in a VLF signals navigation problem. Without ANNs, our VLF signals navigation problem would be very difficult.

We conduct two VLF navigation experiments with an indoor and outdoor environment. The ANNs used for these problems provide confidence with probabilistic estimates of position either through class probabilities or probability distributions parameterized by the output of MDNs. ANNs need a measure of confidence in their estimates to work with the filters since navigation filters require a confidence of their estimates. In our problems we achieve an indoor localization accuracy of 86.7% for 50 discrete locations, and a 2D RMS error of 63m for a 1km^2 area of navigation.

15. SUBJECT TERMS
Artificial Neural Network, Alternative Navigation, Very Low Frequency (VLF), Mixture Density Network (MDN)

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** John F Raquet (ENG)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	233	(937) 785-3636 x4580 John.Raquet@afit.edu