Kalman Filters and Neural Networks: A Survey

Allen Wang
allenwang@utexas.edu

***Abstract*—Estimation of variables that are difficult to measure is a problem that has been widely studied and applied to different fields such as navigation, guidance and control of vehicles, signal processing, etc. Kalman Filtering has been traditionally used for this type of problem, but requires detailed knowledge of the underlying problem in order to provide accurate estimates. This detailed knowledge becomes increasingly difficult to obtain, especially as the underlying problem becomes more complex. Recently, neural networks and deep learning have been applied to complex problems, including computer vision, speech processing, etc. Recurrent variations such as the Recurrent Neural Network (RNN) and LSTM have been applied to problems traditionally solved by Kalman Filtering. Although RNNs and LSTMs have been shown to have comparable, if not better performance than Kalman Filtering, the complexity of neural network systems makes results uninterpretable.**

***Index Terms*—Kalman Filter, Extended Kalman Filter, Unscented Kalman Filter, Neural Networks, Recurrent Neural Networks, Feedforward Neural Networks, LSTM**

I. Introduction

Accurate estimations of system-states are needed in many engineering tasks, such as in navigation, guidance and control of vehicles, signal processing, etc. However, these system states may not be easily accessible for observation. Instead, only system inputs and outputs are available, which are usually subjected to statistical noise.

Kalman Filtering [1] is a well-known algorithm that provides state estimations as well as covariance estimations of its predictions. It is a recursive technique, where state estimations are obtained in two distinct steps, and where predictions are linear combinations of the filter's internal states and observations. It uses techniques based on Bayesian inference and joint probability distribution estimates. It solves the problem of the state space model, and in order for a Kalman Filter to provide accurate estimates, it must be provided accurate state transitions as well as accurate estimates of the noise covariance. It is a linear predictor, but has nonlinear variations (including the Extended Kalman Filter and Unscented Kalman Filter). Due to its optimality, it has been widely applied to different engineering tasks. However, due to its assumptions and necessary knowledge of the underlying system to predict, it's not always applicable as complexity of the problem increases.

Meanwhile, recent developments in deep learning have led to neural network models becoming state-of-the-art solutions in various fields, including computer vision, speech processing, natural language processing, etc. [2] [3] [4] In the standard tracking problem, neural network based models have been applied as the Recurrent Neural Network, and its variation, the LSTM, have been shown to be effective in processing information over time. [5] [6] However, neural networks are criticized for being too complex, and its results uninterpretable. Furthermore, neural networks are required to go through a training phase to learn the weights, which requires large amounts of data that might not always be accessible. If large amounts of data are provided, the neural network has many parameters which require experience and knowledge of neural network dynamics to tune. Finally, once a neural network is implemented, it may take more computation time to produce an estimate than the Kalman filter would. While neural network based models have been shown to perform comparatively, if not better, than Kalman Filter approaches, [7] [8] it has not yet entirely replaced Kalman filters in practice.

This paper will provide an overview of the formulations of Kalman Filters and its nonlinear variations as well as an overview of the formulations of neural networks and its variations. Further, a survey will be conducted on recent literature comparing the performance of each in various tasks, as well as systems that have used both. Finally, discussion of the appropriateness of each applied to engineering tasks is provided, as well as simulations comparing each.

II. Background: Kalman Filters

In this section, we describe the Kalman Filter, along with the type of problem to be solved. We also discuss variations of the Kalman Filter, as well as advantages and disadvantages of the different variations.

A. Problem Setting

Suppose we have a sequence of unobserved variables $\boldsymbol{x}_1, ..., \boldsymbol{x}_t \in \mathbb{R}^s$. Each unobserved variable $\boldsymbol{x}_t$ has a corresponding observation $\boldsymbol{y}_t \in \mathbb{R}^d$, as well as a corresponding observed action $\boldsymbol{u}_t \in \mathbb{R}^c$. Assume that these variables are related such that

$$\begin{aligned} \boldsymbol{x}_i &= f_i(\boldsymbol{x}_{i-1}, \boldsymbol{u}_{i-1}) + \boldsymbol{w}_k \\ \boldsymbol{y}_i &= h_i(\boldsymbol{x}_i) + \boldsymbol{v}_i \end{aligned} \tag{1}$$

where $\boldsymbol{w}_k, \boldsymbol{v}_k$ are the process and observation noises, with the assumption that $\boldsymbol{w}_k \sim \mathcal{N}(0, \boldsymbol{Q}_k), \boldsymbol{v}_k \sim \mathcal{N}(0, \boldsymbol{R}_k)$ and f_i, h_i are differentiable functions.

In the proceeding sections, we want to predict the value of $\boldsymbol{x}$ at any time step i, given the observations $\boldsymbol{y}_0, ..., \boldsymbol{y}_{i-1}$. The predictor value will be denoted $\hat{\boldsymbol{x}}_i$.

B. Kalman Filter

The Kalman Filter [1] is based on linear dynamical systems that that are discretized in the time domain. Due to this linearity assumption, we modify equation 1 such that f_i, h_i are linear functions. Specifically, the model for linear equations is

$$\begin{aligned} \boldsymbol{x}_i &= \boldsymbol{F}_i \boldsymbol{x}_{i-1} + \boldsymbol{B}_i \boldsymbol{u}_{i-1} + \boldsymbol{w}_k \\ \boldsymbol{y}_i &= \boldsymbol{H}_i \boldsymbol{x}_i + \boldsymbol{v}_i \end{aligned} \tag{2}$$

where $\boldsymbol{F}_i, \boldsymbol{B}_i, \boldsymbol{H}_i$ are matrices. The Kalman Filter is a recursive estimator solving a problem in the state-space model shown in equation 2, with a prediction and update step.

1) Prediction Step:

$$\begin{aligned} \hat{\boldsymbol{x}}_{i|i-1} &= \boldsymbol{F}_i \hat{\boldsymbol{x}}_{i-1|i-1} + \boldsymbol{B}_i \boldsymbol{u}_i \\ \boldsymbol{P}_{i|i-1} &= \boldsymbol{F}_i \boldsymbol{P}_{i-1|i-1} \boldsymbol{F}_i^T + \boldsymbol{Q}_i \end{aligned} \tag{3}$$

2) Update Step:

$$\begin{aligned} \boldsymbol{e}_i &= \boldsymbol{y}_i - \boldsymbol{H} \boldsymbol{x}_{i|i-1} \\ \boldsymbol{S}_i &= \boldsymbol{R}_i + \boldsymbol{H}_i \boldsymbol{P}_{i|i-1} \boldsymbol{H}_i^T \\ \boldsymbol{K}_i &= \boldsymbol{P}_{i|i-1} \boldsymbol{H}_i^T \boldsymbol{S}_i^{-1} \\ \hat{\boldsymbol{x}}_{i|i} &= \hat{\boldsymbol{x}}_{i|i-1} + \boldsymbol{K}_k \boldsymbol{e}_i \\ \boldsymbol{P}_{i|i} &= \boldsymbol{P}_{i|i-1} - \boldsymbol{K}_i \boldsymbol{S}_i \boldsymbol{K}_k^T \\ \boldsymbol{e}_{i|i} &= \boldsymbol{y}_i - \boldsymbol{H}_i \hat{\boldsymbol{x}}_{i|i} \end{aligned} \tag{4}$$

The state of the Kalman filter is typically represented by two variables: $\hat{\boldsymbol{x}}_{i|i}$, the a posteriori state estimate at time i given $\boldsymbol{y}_0, ..., \boldsymbol{y}_i$, and $\boldsymbol{P}_{i|i}$, the a posteriori error covariance matrix.

Note that in this discussion, no assumptions have been made regarding the probability distributions of $\boldsymbol{x}, \boldsymbol{v}$ or $\boldsymbol{w}$. It is actually proven to be the optimal filter [9] in the case that

- The model perfectly matches the real system
- The noise $\boldsymbol{v}, \boldsymbol{w}$ is white (uncorrelated)
- The covariances $\boldsymbol{Q}, \boldsymbol{R}$ are known

However, the main issue with the Kalman Filter is that typically we want to model a system that is not perfectly linear. The following discussion provides variations of the Kalman Filter that deals with nonlinearities.

C. Extended Kalman Filter

In the case of nonlinear functions f_i, h_i in equation 1, the linearity assumption of the Kalman filter is violated. The Extended Kalman Filter (EKF) [10] offers a solution to this problem - the Taylor expansion is used to find a linear approximation of the nonlinearities. The EKF then follows a similar prediction and update step:

1) Prediction Step:

$$\begin{aligned} \hat{\boldsymbol{x}}_{i|i-1} &= f_i(\hat{\boldsymbol{x}}_{i-1|i-1}, \boldsymbol{u}_{i-1}) \\ \boldsymbol{P}_{i|i-1} &= \boldsymbol{F}_{i-1} \boldsymbol{P}_{i-1|i-1} \boldsymbol{F}_{i-1}^T + \boldsymbol{Q}_{i-1} \end{aligned} \tag{5}$$

2) Update Step:

$$\begin{aligned} \boldsymbol{e}_i &= \boldsymbol{y}_i - h_i(\hat{\boldsymbol{x}}_{i|i-1}) \\ \boldsymbol{S}_i &= \boldsymbol{H}_i \boldsymbol{P}_{i|i-1} \boldsymbol{H}_i^T + \boldsymbol{R}_i \\ \boldsymbol{K}_i &= \boldsymbol{P}_{i|i-1} \boldsymbol{H}_i^T \boldsymbol{S}_i^{-1} \\ \hat{\boldsymbol{x}}_{i|i} &= \hat{\boldsymbol{x}}_{i|i-1} + \boldsymbol{K}_i \boldsymbol{e}_i \\ \boldsymbol{P}_{i|i} &= (\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i) \boldsymbol{P}_{i|i-1} \end{aligned} \tag{6}$$

where

$$\boldsymbol{F}_{i-1} = \left. \frac{\partial f}{\partial \boldsymbol{x}} \right|_{\hat{\boldsymbol{x}}_{i-1|i-1}, \boldsymbol{u}_{i-1}}$$

$$\boldsymbol{H}_{i-1} = \left. \frac{\partial h}{\partial \boldsymbol{x}} \right|_{\hat{\boldsymbol{x}}_{i|i-1}}$$

Note that the EKF only provides first-order approximations to optimal terms - this can introduce large errors in the true posterior mean and covariance of the transformed random variable, which may lead to suboptimal performance - for instance, when f_i, h_i are highly non-linear.

Ultimately, unlike the Kalman Filter, the Extended Kalman Filter is not an optimal filter (except when f_i, h_i are linear, then the EKF becomes the KF). If the initial

estimate is wrong, or the process modeled incorrectly, then the filter may diverge due to its linearization. Additionally, the estimated covariance matrix $\boldsymbol{P}$ typically underestimates the true covariance matrix. [11]

D. Unscented Kalman Filter

The Unscented Kalman Filter (UKF) [12] is another variation of the Kalman filter applied to nonlinear functions and addresses the issue of errors arising due to approximation in the EKF. The state distribution is modeled by a Gaussian Random variable (GRV), but is specified using a minimal set of sample points that capture the true mean and covariance of the GRV. When propagated through the actual non-linear system (not the approximation as in EKF), the posterior mean and covariance is accurately captured to the 3rd order (Taylor series expansion). Specifically, this process is called the unscented transform, and using this has the benefits of creating a filter that is more accurate than the EKF for some systems, and removes the requirement to calculate Jacobians. [13]

1) Unscented Transform: The Unscented Transform (UT) is a method used to calculate the statistics of random variable that undergoes a nonlinear transformation. Consider a random variable $\boldsymbol{x} \in \mathbb{R}^L$ that is propagated through a nonlinear function, $\boldsymbol{y} = g(\boldsymbol{s})$. Assume $\boldsymbol{x}$ has mean $\bar{\boldsymbol{x}}$, covariance $\boldsymbol{P}_x$. To calculate the statistics of $\boldsymbol{y}$, form a matrix $\boldsymbol{\mathcal{X}}$ of $2L+1$ sigma vectors $\mathcal{X}_i$, such that:

$$\begin{aligned} \mathcal{X}_0 &= \bar{\boldsymbol{x}} \\ \mathcal{X}_i &= \bar{\boldsymbol{x}} + \left(\sqrt{(L+\lambda)\boldsymbol{P}_x}\right)_i \\ \mathcal{X}_i &= \bar{\boldsymbol{x}} - \left(\sqrt{(L+\lambda)\boldsymbol{P}_x}\right)_{i-L} \\ W_0^{(m)} &= \lambda/(L+\lambda) \\ W_0^{(c)} &= \lambda/(L+\lambda) + (1-\alpha^2+\beta) \\ W_i^{(m)} &= W_i^{(c)} = 1/\{2(L+\lambda)\} \\ i &= 1, ..., 2L \end{aligned} \tag{7}$$

where $\lambda = \alpha^2(L+\kappa) - L$ is a scaling parameter. α is used to determine the spread of sigma points around $\bar{\boldsymbol{x}}$ and is usually set to some small value (e.g., 1e-3). κ is a secondary scaling parameter typically set to 0, and β is used to incorporate prior knowledge of the distribution of $\boldsymbol{x}$ (e.g., for Gaussian distributions, $\beta = 2$ is optimal). The term $\left(\sqrt{(L+\lambda)\boldsymbol{P}_x}\right)_i$ is the i-th row of the matrix square root. The sigma vectors are propagated through the nonlinear function, resulting in:

$$\mathcal{Y} = g\left(\mathcal{X}_i\right), \qquad i = 0, ..., 2L \tag{8}$$

where the mean and covariance for $\boldsymbol{y}$ are approximated using a weighted sample mean and covariance of the posterior sigma points,

$$\begin{aligned} \bar{\boldsymbol{y}} &\approx \sum_{i=0}^{L} W_i^{(m)} \mathcal{Y}_i \\ \boldsymbol{P}_y &\approx \sum_{i=0}^{2L} W_i^{(c)} \left\{\mathcal{Y}_i - \bar{\boldsymbol{y}}\right\} \left\{\mathcal{Y}_i - \bar{\boldsymbol{y}}\right\}^T \end{aligned} \tag{9}$$

Note the differences between the UT and other sampling methods (e.g. Monte-Carlo methods like the Particle Filter) - these typically require orders of magnitude of more sample points in order to propagate an accurate distribution of the state. Furthermore, UT results are accurate to the third order for Gaussian inputs for all non-linearities. Meanwhile, for non-Gaussian inputs, approximations are accurate to at least the second order.
The UKF is ultimately using the UT in a filtering setting. This also results in a prediction and update step. Specifically, using our problem formulation in equation 1:

2) Prediction Step: First note that the estimated state and covariance are augmented with the mean and covariance of the process noise:

$$\begin{aligned} \boldsymbol{x}_{i-1|i-1}^{\alpha} &= \begin{bmatrix} \hat{\boldsymbol{x}}_{i-1|i-1}^T & E\left[\boldsymbol{w}_i^T\right] \end{bmatrix}^T \\ \boldsymbol{P}_{i-1|i-1}^{\alpha} &= \begin{bmatrix} \boldsymbol{P}_{i-1|i-1} & 0 \\ 0 & \boldsymbol{Q}_i \end{bmatrix} \end{aligned} \tag{10}$$

Derive a set of $2L+1$ sigma points from the augmented state and covariance from equation 10, where L is the dimension of the augmented state.

$$\begin{aligned} \mathcal{X}_{i-1|i-1}^{0} &= \boldsymbol{x}_{i-1|i-1}^{\alpha} \\ \mathcal{X}_{i-1|i-1}^{k} &= \boldsymbol{x}_{i-1|i-1}^{\alpha} + \left(\sqrt{(L+\lambda)\boldsymbol{P}_{i-1|i-1}^{\alpha}}\right)_k \\ \mathcal{X}_{i-1|i-1}^{k+L} &= \boldsymbol{x}_{i-1|i-1}^{\alpha} - \left(\sqrt{(L+\lambda)\boldsymbol{P}_{i-1|i-1}^{\alpha}}\right)_{k-L} \\ k &= i, ..., L \end{aligned} \tag{11}$$

Propagate the sigma points through f:

$$\mathcal{X}_{i-1|i-1}^{k} = f\left(\mathcal{X}_{i-1|i-1}^{k}\right), \quad k = 0, ..., 2L \tag{12}$$

Our estimates become:

$$\begin{aligned} \hat{\boldsymbol{x}}_{i|i-1} &= \sum_{k=0}^{2L} \boldsymbol{W}_s^k \mathcal{X}_{i|i-1}^k \\ \boldsymbol{P}_{i|i-1} &= \sum_{k=0}^{2L} \boldsymbol{W}_c^k \left[\mathcal{X}_{i|i-1}^k - \hat{\boldsymbol{x}}_{i|i-1}\right] \left[\mathcal{X}_{i|i-1}^k - \hat{\boldsymbol{x}}_{i|i-1}\right]^T \end{aligned} \tag{13}$$

where the weights $\boldsymbol{W}$ are defined as in equation 7.

3) Update Step: The predicted state and covariance are augmented as in equation 10, but instead with the mean and covariance of the measurement noise.

$$\begin{aligned} \boldsymbol{x}^{\alpha}_{i-1|i-1} &= \begin{bmatrix} \hat{\boldsymbol{x}}^T_{i-1|i-1} & E\left[\boldsymbol{v}^T_i\right] \end{bmatrix}^T \\ \boldsymbol{P}^{\alpha}_{i-1|i-1} &= \begin{bmatrix} \boldsymbol{P}_{i-1|i-1} & 0 \\ 0 & \boldsymbol{R}_i \end{bmatrix} \end{aligned} \tag{14}$$

As in equation 11, $2L+1$ sigma points are derived from the augmented state and covariance. Alternatively, if the UKF prediction was used, the calculated sigma points from before can be augmented themselves such that:

$$\boldsymbol{\mathcal{X}}_{i|i-1} := \begin{bmatrix} \mathcal{X}^T_{i|i-1} & E\left[\boldsymbol{v}^T_i\right] \end{bmatrix}^T \pm \sqrt{(L+\lambda)\boldsymbol{R}^{\alpha}_i} \tag{15}$$

where

$$\boldsymbol{R}^{\alpha}_i = \begin{bmatrix} 0 & 0 \\ 0 & \boldsymbol{R}_i \end{bmatrix} \tag{16}$$

These sigma points are propagated through h:

$$\gamma^k_i = h\left(\mathcal{X}^k_{i|i-1}\right), \quad k = 0, ..., 2L \tag{17}$$

The weighted sigma points are recombined to produce predicted measurement and predicted measurement covariance:

$$\begin{aligned} \hat{\boldsymbol{z}}_i &= \sum_{k=0}^{2L} \boldsymbol{W}^k_s \gamma^k_i \\ \boldsymbol{P}_{z_i z_i} &= \sum_{k=0}^{2L} \boldsymbol{W}^k_c \left[\gamma^k_i - \hat{\boldsymbol{z}}_i\right] \left[\gamma^k_i - \hat{\boldsymbol{z}}_i\right]^T \end{aligned} \tag{18}$$

The state-measurement covariance matrix is denoted as:

$$\boldsymbol{P}_{x_i z_i} = \sum_{k=0}^{2L} \boldsymbol{W}^k_c \left[\mathcal{X}^k_{i|i-1} - \hat{\boldsymbol{x}}_{i|i-1}\right] \left[\gamma^k_i - \hat{\boldsymbol{z}}_i\right]^T \tag{19}$$

This is used to compute the UKF Kalman gain. The final predictions follow:

$$\begin{aligned} K_i &= \boldsymbol{P}_{x_i z_i} \boldsymbol{P}^{-1}_{z_i z_i} \\ \hat{\boldsymbol{x}}_{i|i} &= \hat{\boldsymbol{x}}_{i|i-1} + K_i(\boldsymbol{z}_i - \hat{\boldsymbol{z}}_i) \\ \boldsymbol{P}_{i|i} &= \boldsymbol{P}_{i|i-1} - K_i \boldsymbol{P}_{z_i z_i} K^T_i \end{aligned} \tag{20}$$

Ultimately, the UKF has been shown to perform better than the EKF. However, there are more parameters in the UKF that affect the overall performance and must be tuned. While this is not a big problem in a practical setting, it is a caveat that should be acknowledged when deciding between using a UKF and an EKF.

III. Background: Neural Networks

In this section, we will be providing a brief overview of neural networks, which are typically used in a machine learning setting. Discussion of neural networks can become very complex, so discussion is restricted to feedforward networks, recurrent neural networks, and LSTMs. Much of the information in this section is derived from [14].

A. Problem Setting

Neural networks are commonly used in a machine learning setting. In such a setting, the task is typically formulated in terms of how a machine learning system should process an example. An example is a collection of features quantitatively measured from some event. An example is represented as $\boldsymbol{x} \in \mathbb{R}^n$, with each entry x_i as a feature. There are many tasks that are solved with machine learning, including supervised tasks such as classification and regression, as well as unsupervised tasks such as clustering. In this paper, we will be focusing on supervised tasks such as classification and regression.

In a supervised learning setting, we are typically provided a dataset $\boldsymbol{x} \in \mathbb{R}^{s \times n}$ and $\boldsymbol{y} \in \mathbb{R}^{s \times m}$, where s is the number of samples, n is the number of features, and m is the dimensionality of a single sample of y. The goal of the machine learning system is then to represent some function $f : \mathbb{R}^n \to \mathbb{R}^m$ that minimizes some loss function $\mathcal{L}(f(\boldsymbol{x}), \boldsymbol{y})$ given $\boldsymbol{x}$ and $\boldsymbol{y}$. Some loss functions may include mean squared error (MSE), root mean squared error (RMSE), cross entropy, etc. The total loss of a model typically takes take the form of

$$\mathcal{L}(f(\boldsymbol{x}), \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(f(x_i), y_i) \tag{21}$$

B. Artificial Neural Networks and Artificial Neurons

Neural Networks (NNs) are a family of systems inspired by biological neural networks constituting animal brains. As such, NNs are composed of artificial neurons (nodes). Each node receives an input $\boldsymbol{x} \in \mathbb{R}^d$, has an associated weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times 1}$, and outputs $h \in \mathbb{R}$ based on the weighted sum of its inputs. In other words,

$$h = f(\boldsymbol{W}^T \boldsymbol{x} + b) \tag{22}$$

where $b \in \mathbb{R}$ is some bias term. The choice of f is typically non-linear, popular choices including rectified linear unit (ReLu), sigmoid, softmax, tanh, etc.

C. Feedforward Neural Networks

Feedforward networks are a class of neural networks which consists of an input layer, one or more hidden layers, and one output layer. Any such layer with m neurons will take an input $\boldsymbol{x} \in \mathbb{R}^d$, have an associated weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times m}$, an associated bias vector $\boldsymbol{b} \in \mathbb{R}^m$ and output a vector $\boldsymbol{h} \in \mathbb{R}^m$. Similar to the case of a single neuron, it has an associated nonlinear function f. For some layer i, the behavior can be modeled as

$$\boldsymbol{h}^{(i)} = f_i\left(\boldsymbol{W}^{(i)T}\boldsymbol{x}^{(i)} + \boldsymbol{b}^{(i)}\right) \tag{23}$$

In a feedforward network, the outputs of a layer acts as an input to the next layer. This is illustrated in figure 1. The input $\boldsymbol{x}$ is ultimately propagated throughout the

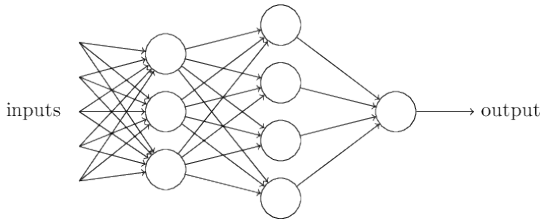

Fig. 1. Illustration of a feedforward network. [15]

layers into the result in the output layer $\boldsymbol{h}$. Therefore, the output $\boldsymbol{h}$ is a function of multiple functions subjected to nonlinearities, which allows for modeling and learning of highly complex functions. In the case of a supervised learning problem, $\boldsymbol{h}$ would represent the prediction of $\boldsymbol{y}$ given $\boldsymbol{x}$.

1) Back-propagation: Training a Feedforward Neural Network: In order to allow a feedforward neural network to learn the representation for $\boldsymbol{h}$, we observe that the weights and the biases between the layers can be updated to minimize a loss function. A technique called stochastic gradient descent is typically applied [16]. Derivation of gradient descent (GD) and stochastic gradient descent (SGD) is out of the scope of this paper, but the SGD equation applied to a layer of a feedforward neural network is denoted by

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta\nabla\mathcal{L}(\boldsymbol{W}) \tag{24}$$

where that the loss $\mathcal{L}$ is as described in equation 21 and $\boldsymbol{W}$ is the associated weight of the layer. To apply SGD to update the weights of each hidden layer of a feedforward network, we apply the chain rule in calculus. Consider an output layer with activation f_o, weight matrix $\boldsymbol{W}_o$, biases $\boldsymbol{b}_o$, and input $\boldsymbol{x}_o$. To update $\boldsymbol{W}_o$,

$$\nabla\mathcal{L}(\boldsymbol{W}_o) = \frac{\partial\mathcal{L}}{\partial\boldsymbol{W}_o} = \frac{\partial\mathcal{L}}{\partial f_o}\frac{\partial f_o}{\partial\left(\boldsymbol{W}_o\boldsymbol{x}_o + \boldsymbol{b}_o\right)}\frac{\partial(\boldsymbol{W}_o\boldsymbol{x}_o + \boldsymbol{b}_o)}{\partial\boldsymbol{W}_o} \tag{25}$$

The error from the output layer is propagated down through the hidden layers backwards - hence the name, back-propagation. The entire algorithm for forward propagation (used for generating $\boldsymbol{h}$) and back-propagation is below in algorithms 1 and 2.

Algorithm 1 Forward-propagation algorithm

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, ..., l\}$, weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, ..., l\}$, bias parameters of the model
Require: $\boldsymbol{x}$, input
Require: $\boldsymbol{y}$, target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, ..., l$ **do**
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\boldsymbol{h} = \boldsymbol{h}^{(l)}$
  $J = \mathcal{L}(\boldsymbol{h}, \boldsymbol{y})$

Algorithm 2 Back-propagation algorithm

  First perform forward-propagation. Then:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}} J = \nabla_{\boldsymbol{h}}\mathcal{L}(\boldsymbol{h}, y)$
  for $k = l, l-1, ..., l$ **do**
    Convert the gradient on the layer's output into a gradient in the pre-nonlinearity activation.
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on thew eights and biases:
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$
    Propagate the gradients w.r.t the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$
  end for

Feedforward networks with multiple hidden layers are called deep neural networks (DNNs). DNNs have been recently been proven to be successful in different machine learning tasks, including computer vision, speech processing, natural language processing, etc. [2] [3] [4] For tasks involving time series, the recurrent neural network (RNN) and its variant, the Long Short Term Memory Network (LSTM) [17] have been shown to be more effective than feedforward networks. [5] [6]

D. Recurrent Neural Networks

Recurrent Networks (RNNs) are a class of neural networks where connections between neurons form a

directed cycle. Note this distinction from feedforward networks which contain only connections from a layer to the next with no cycles. Directed cycles in RNNs allow the network to learn temporal behavior, and can use internal memory to process inputs. Figure 2 shows an illustration of a RNN, with input $\boldsymbol{x}_t$, layer $\boldsymbol{A}$ and an output $\boldsymbol{h}_t$. In the setting of such a RNN, we might model the behavior such that

$$\boldsymbol{h}_t = f_W(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) \tag{26}$$

where f_W can be understood as some function with parameters W. Note that the same function and same set of parameters W is used at every time step. Note the implication of equation 26: since $\boldsymbol{h}_t$ is a function of $\boldsymbol{h}_{t-1}$, then recursively we can see that ultimately $\boldsymbol{h}_t$ is a function of $\boldsymbol{h}_{t-2}, ..., \boldsymbol{h}_0$ as well. If we "unfold" the RNN, it ultimately seems to be a feedforward network with t hidden layers. This is shown in figure 2 as well.

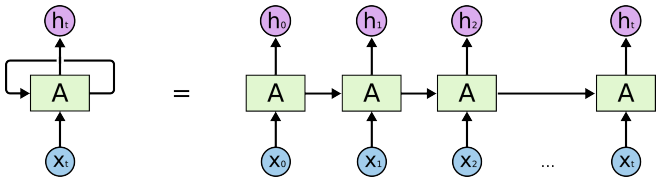

Fig. 2. An unrolled Recurrent Neural Network. [18]

The concept of backpropagation can be applied to train the RNN as well. Consider the vanilla RNN:

$$\boldsymbol{h}_t = f(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{b}) \tag{27}$$

where $\boldsymbol{U}$ and $\boldsymbol{W}$ represent weight matrices, and $\boldsymbol{b}$ represents a bias vector as in a standard feedforward network. Let $\boldsymbol{s}_t = \boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{b}$. If we apply backpropagation to update $\boldsymbol{U}$ for instance, we need to know the derivative of f, e.g.

$$\frac{\partial f}{\partial U} = \frac{\partial f}{\partial \boldsymbol{s}_t}\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{U}} \tag{28}$$

However, note that the second term depends on $\boldsymbol{h}_{t-1}$, which also has a $\boldsymbol{U}$ term. The gradient would actually then take the form:

$$\frac{\partial f}{\partial U} = \sum_{k=1}^{t} \frac{\partial f}{\partial \boldsymbol{s}_t}\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s_k}}\frac{\boldsymbol{s}_k}{\boldsymbol{U}} \tag{29}$$

This method of applying back-propagation over time steps is known as back-propagation through time (BPTT). While theoretically, RNNs should handle long-term dependencies well, they have not been unable to do so in practice. [19] [20] This was observed to be due to what was called the "Vanishing Gradient problem," which emerges from BPTT. An alternative to the RNN was proposed, the Long Short Term Memory (LSTM) that avoids this problem through its construction.

E. Long Short Term Memory Networks (LSTM)

Long Short Term Memory Networks (LSTMs) [17] are a variation of RNNs that are capable of learning long-term dependencies. Where a vanilla RNN may be represented as

$$\boldsymbol{h}_t = \tanh\left(W\begin{pmatrix}\boldsymbol{h}_{t-1}\\ \boldsymbol{x}_t\end{pmatrix}\right), \tag{30}$$

a vanilla LSTM may be represented as

$$\begin{pmatrix}\boldsymbol{i}\\ \boldsymbol{f}\\ \boldsymbol{o}\\ \boldsymbol{g}\end{pmatrix} = \begin{pmatrix}\sigma\\ \sigma\\ \sigma\\ \tanh\end{pmatrix}\boldsymbol{W}\begin{pmatrix}\boldsymbol{h}_{t-1}\\ \boldsymbol{x}_t\end{pmatrix}$$
$$\boldsymbol{c}_t = \boldsymbol{f}\odot\boldsymbol{c}_{t-1} + \boldsymbol{i}\odot\boldsymbol{g} \tag{31}$$
$$\boldsymbol{h}_t = \boldsymbol{o}\odot\tanh(\boldsymbol{c}_t)$$

where, intuitively,

- $\boldsymbol{f}$: forget gate (whether to erase a cell)
- $\boldsymbol{i}$: input gate (whether to write a cell)
- $\boldsymbol{g}$: "gate" gate (how much to write to a cell)
- $\boldsymbol{o}$: output gate (how much to reveal a cell)

A visual representation of the difference between RNNs and LSTMs is shown in figure 3.

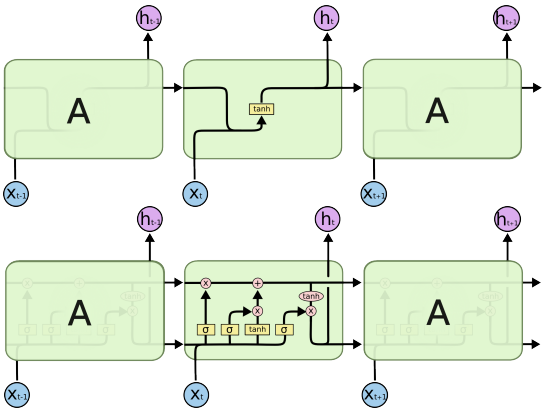

Fig. 3. Visual Comparison between RNN and LSTM. [18]

The LSTM also can "unfold" through time, but the key difference is that the LSTM has regulating structures called gates that optionally let information through. LSTMs also are typically trained using BPTT as in RNNs, but in an LSTM block, when error values are back-propagated from the output, the error is retained in the block's memory. This is continuously fed back

between the gates, until they learn to cut off the value. This key innovation avoids the vanishing gradient problem, and as such, LSTMs are typically preferred over the regular RNN in practice.

IV. Comparing Kalman Filters and Neural Networks

This section will compare Kalman Filters (and their variations) and Neural Networks (and their variations) by exploring contributions in various literature. Specifically, comparisons between how each can be used to solve the same problems will be explored, as well as how each have been beneficial in the same system. For simplicity, in this section, mention of Kalman Filters (KFs) will refer to Kalman Filters and its variations, while Neural Networks (NNs) will refer to Neural Networks and its variations.

As established in section 2, the family of Kalman filters are used to solve problems in the state-space form, i.e., in the form of equation 1. However, the main issue with the family of Kalman Filters is that explicit knowledge of the state space model must be known, including the transition matrices and covariance matrices of the assumed noise. For real life applications, such as tracking an object given noisy sensor measurements with occlusions, formulating the problem in a state-space model becomes more difficult. Furthermore, some strong assumptions made in the Kalman Filter may become unknowingly violated. An example noted in [21], an engineer may measure the variance of a GPS unit by placing it in a fixed, known position, and directly taking the empirical variation as the noise covariance for the KF. This would actually lead to the KF providing poor estimates of the state, because the GPS errors are correlated over time, whereas the straightforward KF assumes that the error is independent. Neural Networks, however, make almost no assumption on the formulation of the problem.

A. Deep Tracking

In [22], a RNN was used to predict an unoccluded scene given noisy, occluded sensor measurements from the dynamics of complex environments. This methodology, called Deep Tracking, was motivated by Bayesian filtering with the objective of modeling

$$P(y_t|x_{1:t}) \tag{32}$$

where $y_t \in \mathbb{R}^n$ is a discrete-time stochastic process modeling the surrounding scene, $x_t \subseteq y_t$ and are the sensor measurements of directly visible scene parts. x_t is encoded such that $x_t = \{v_t, r_t\}$ where $\{v_t^i, r_t^i\} = \{1, y_t^i\}$ if the i-th element is observed, and $\{0, 0\}$ otherwise. Another underlying Markov process, h_t, is assumed to exist that captures the state of the world with the joint pdf:

$$P(y_{1:N}, x_{1:N}, h_{1:N}) = \prod_{t=1}^{N} P(x_t|y_t)P(y_t|h_t)P(h_t|h_{t-1}) \tag{33}$$

where

- $P(h_t|h_{t-1})$ denotes the hidden state transition probability capturing the dynamics of the world;
- $P(y_t|h_t)$ models the instantaneous unoccluded sensor space,
- $P(x_t|y_t)$ describes the sensing process.

Estimating $P(y_t|x_{1:t})$ is then framed with recursive Bayesian estimation of the belief $\mathcal{B}_t$ which corresponds to a distribution $Bel(h_t) = P(h_t|x_{1:t})$ at any time t. This can be computed recursively for the model as

$$\begin{aligned} Bel^-(h_t) &= \int_{h_{t-1}} P(h_t|h_{t-1})Bel(h_{t-1}) \\ Bel(h_t) &\propto \int_{y_t} P(x_t|y_t)P(y_t|h_t)Bel^-(h_t) \end{aligned} \tag{34}$$

where $Bel^-(h_t), Bel(h_t)$ act as a prediction step and measurement updated step as in Kalman Filters. Then,

$$P(y_t|x_{1:t}) = \int_{h_t} P(y_t|h_t)Bel(h_t) \tag{35}$$

Computation of $P(y_t|x_{1:t})$ is then calculated by updating the belief $\mathcal{B}_t$ and providing prediction $P(y_t|x_{1:t}) = P(y_t|\mathcal{B}_t)$:

$$\begin{aligned} \mathcal{B}_t &= F(\mathcal{B}_{t-1}, x_t) \\ P(y_t|x_{1:t}) &= P(y_t|\mathcal{B}_t) \end{aligned} \tag{36}$$

Furthermore, y_{t+n} can be predicted with $P(y_{t+n}|x_{1:t})$ instead of just y_t by providing empty observations $x_{(t+1):(t+n)} = \emptyset$. However, realistically, this approach is limited since it requires a suitable belief representation as well as explicit knowledge of distributions in equation 33. RNNs are then used in order to avoid the need to specify this knowledge - one for modeling $F(\mathcal{B}_{t-1}, x_t)$ and one for modeling $P(y_t|\mathcal{B}_t)$. As discussed in previous sections, these models can learn these distributions given only training data. However, a key innovation that was contributed in this paper was that instead of modeling $P(y_t|\mathcal{B}_t)$, the network predicts $P(y_{t+n}|x_{1:t})$, i.e. a state in the future. Forcing this constraint upon the network forces it to learn the dynamics of the system, and must

learn to represent and simulate object dynamics when the object is not seen. This must then be converted to predict $P(y_{t+n}|\mathcal{B}_{t+n})$. The paper concludes with experimental results, showing that it was able to achieve highly faithful reconstructions of the synthetic simulated data. Ultimately, Deep Tracking provided an end-to-end approach using RNNs for a tracking problem based on raw sensor data.

B. Target Tracking with Kalman Filtering, KNN and LSTMs

In [7], Kalman Filtering and LSTMs were directly compared in the setting of tracking an unknown number of targets given measurements from multiple noisy sensors. The multi-target tracking problem was separated into two subproblems of:

- creating new tracks when a target is born and removing old tracks when a target dies, and associating measurements with either the target they originated from, or clutter (in the case of false positives)
- tracking a single target given only valid measurements that were created by the target

A Rao-Blackwellized particle filter was used in order to handle target birth, death, and measurement-target association, reducing this into a single target tracking problem. For experimentation, Iter et al. performed experiments on the KITTI object tracking benchmark, based on an autonomous driving setting, where human annotated bounding boxes of objects are provided. The Kalman Filter used was a fairly standard Kalman Filter used for predicting the x, y position as well as their respective velocities. In the LSTM approach, a problem found was that because LSTMs are discriminative models, they provide only predicted values, and not a distribution - which a generative model like a Kalman Filter can provide. A distribution is useful in the measurement target association, so the key innovation here was that two LSTMs were trained - one to predict the next position, and another to predict the variance in the prediction. Ultimately, they found that the LSTMs had a more complex set up due to tuning model parameters, although the LSTM performed about twice as well for single tracking (in terms of mean squared error on a validation set). However, when using the predictions as input into the Rao-Blackwellized particle filter for the multi-target tracking problem, the LSTM and Kalman Filter both performed similarly.

C. An Algorithmic Approach to Adaptive State Filtering Using Recurrent Neural Networks

In [8], Parlos, Menon and Atiya use a feedforward network and RNNs to create a recursive filtering system for state space models similar to the EKF, i.e. with a predict and update step. Two models are created: the "Nonadaptive Neural State Filter" and the "Adaptive Neural State Filter."

1) Nonadaptive Neural State Filter: In the nonadaptive approach, the system solves the problem of the form of equation 1. Neural networks are used to learn the functions f, h, as well as a filter relation $\mathcal{K}$ (which is similar to the Kalman gain). To obtain a state estimate, $\hat{\boldsymbol{x}}(i+1|i+1)$, the predict step is

$$\begin{cases} \hat{\boldsymbol{x}}(t+1|t) = f_{mod}(\hat{\boldsymbol{x}}_{NN}(t|t), \boldsymbol{u}(t)) \\ \hat{\boldsymbol{y}}(t+1|t) = h_{mod}(\hat{\boldsymbol{x}}(t+1|t)) \end{cases} \tag{37}$$

where $\hat{\boldsymbol{x}}(t+1|t), \hat{\boldsymbol{y}}(t+1|t)$ are the system state and output predictions, $\hat{\boldsymbol{x}}_{NN}(t|t)$ is neural state estimate, and f_{mod}, h_{mod} are the assumed functions of the model in equation 1. The update step is

$$\hat{\boldsymbol{x}}_{NN}(t+1|t+1) = \mathcal{K}_{NN}(\hat{\boldsymbol{x}}(t+1|t), \boldsymbol{\mathcal{Y}}(t+1), \boldsymbol{\mathcal{E}}(t+1)) \tag{38}$$

where $\boldsymbol{\mathcal{Y}}(t+1)$ is a vector that contains present and past system output measurements and $\boldsymbol{\mathcal{E}}(t+1)$ is a vector containing present and past innovation terms, where an innovation term is defined

$$\epsilon(t+1) \equiv \boldsymbol{y}(t+1) - \hat{\boldsymbol{y}}(t+1|t) \tag{39}$$

Note some of the differences between the Nonadaptive Neural State Filter and the EKF: f, h do not need to be explicitly known and therefore the Jacobians are not necessary. Also, where in the EKF, the update step is just a linear combination of the latest innovations, the Nonadaptive Neural State Filter can take a nonlinear form and can include past innovations.

2) Adaptive Neural State Filter: The Adaptive Neural State Filter does not assume any system model - rather, only that a finite set of input, state, and output measurements are available. A key difference utilized in the adaptive approach is that the state space representation in 1 can be rewritten as

$$\begin{cases} \hat{\boldsymbol{x}}(t+1|t) = f_{innov}(\hat{\boldsymbol{x}}(t|t-1), \boldsymbol{u}(t), \boldsymbol{\mathcal{E}}(t)) \\ \hat{\boldsymbol{y}}(t+1|t) = h_{inov}(\hat{\boldsymbol{x}}(t|t-1), \boldsymbol{u}(t), \boldsymbol{\mathcal{E}}(t)) \end{cases} \tag{40}$$

where f_{innov}, h_{innov} are nonlinear functions related to f, h of the noise representations.
Note that the innovations term $\boldsymbol{\mathcal{E}}$ accounts for stochastic

effects and modeling uncertainties that are not predicted by the filter predictor. Therefore, error in the state prediction $\hat{\boldsymbol{x}}(t|t-1)$ for the innovations form of the state-space representation (eq. 40) is compensated by $\boldsymbol{\mathcal{E}}$. This implies that for predictions, only one of $\boldsymbol{\mathcal{Y}}(t)$ and $\hat{\boldsymbol{\mathcal{Y}}}(t|t-1)$ are required, since the error terms are encapsulated by $\boldsymbol{\mathcal{E}}$. Finally, f_{innov}, h_{innov} are both learned using neural networks. For the Adaptive Neural State Filter, the prediction step is

$$\begin{cases} \hat{\boldsymbol{x}}_{NN}(t+1|t) = f_{NN}\left(\hat{\boldsymbol{x}}_{NN}(t|t), \boldsymbol{u}(t), \hat{\boldsymbol{\mathcal{Y}}}_{NN}(t|t-1)\right) \\ \hat{\boldsymbol{y}}_{NN}(t+1|t) = h_{NN}\left(\hat{\boldsymbol{x}}_{NN}(t|t), \boldsymbol{u}(t), \hat{\boldsymbol{\mathcal{Y}}}_{NN}(t|t-1)\right) \end{cases} \tag{41}$$

where $\hat{\boldsymbol{x}}_{NN}(t+1|t), \hat{\boldsymbol{y}}_{NN}(t+1|t)$ are the neural state and output predictions, and where $\hat{\boldsymbol{\mathcal{Y}}}_{NN}(t|t-1)$ is the vector containing current and past output predictor responses. The update step is

$$\hat{\boldsymbol{x}}_{NN}(t+1|t+1) = \mathcal{K}_{NN}(\hat{\boldsymbol{x}}_{NN}(t+1|t), \boldsymbol{\mathcal{Y}}(t+1), \boldsymbol{\mathcal{E}}_{NN}(t+1)) \tag{42}$$

Experiments were run comparing the Neural State Filters and EKFs with different complexities. One experiment simulated a known state-space model, and another experiment simulated a more complex problem of a UTSG process simulator requiring two state filters. In the state-space experiment, the EKFs and Neural State Filters performed comparatively, with no demonstrated significant benefit of the neural filtering algorithms with the added cost that the neural filtering algorithms does not track state estimation error. In the more complex UTSG system, the EKFs did not converge to a steady-state value in the simulation time-frame, while both neural filters converged rapidly. Estimations of the neural filters were acceptable, as compared to EKF estimations, affirming that the neural filter algorithms can be applied to complex filtering problems. Ultimately, the authors noted that neural filters are only appropriate in more complex filtering problems.

D. Kalman filters improve LSTM network performance in problems unsolved by traditional recurrent nets

In [23], decoupled EKFs (DEKFs) are used as a replacement to the gradient descent training algorithm typically used in training neural networks. Note that this paper will not provide a complete description of DEKFs. The motivation behind this approach is that gradient descent algorithms depend on instantaneous estimations of gradients, but does not take into consideration the history information for weight updates. The DEKF overcomes this limitation by considering training as an optimal filtering problem and uses all information supplied to the network at any given time step. It does so by assuming that the optimum setting of weights is stationary. However, since there are so many weights in the network, the resulting matrices become so large that a node-decoupled version of the algorithm is instead used. If the network uses a quadratic error function,

$$E = \frac{1}{2}\sum_{i=1}^{n_Y}(d_i(t) - y_i(t))^2 \tag{43}$$

where $d_i(t)$ is the target for the ith output at time t, then this can be formulated as

$$\boldsymbol{G}_i(t) = \boldsymbol{K}_i(t-1)\cdot\boldsymbol{C}_i^T(t)\cdot \left[\sum_{i=1}^{g}\boldsymbol{C}_i(t)\boldsymbol{K}_i(t-1)\boldsymbol{C}_i^T(t) + \boldsymbol{R}(t)\right]^{-1} \tag{44}$$

$$\boldsymbol{w}_i(t) = \boldsymbol{w}_i(t-1) + \boldsymbol{G}_i(t)[\boldsymbol{d}(t) - \boldsymbol{y}(t)] \tag{45}$$

$$\boldsymbol{K}_i(t) = \boldsymbol{K}_i(t-1) - \boldsymbol{G}_i(t)\boldsymbol{C}_i(t)\boldsymbol{K}_i(t-1) + \boldsymbol{Q}_i(t) \tag{46}$$

where g is the number of neurons, i is a particular neuron ($1 \leq i \leq g$) and $\boldsymbol{w}_i$ is a vector with all the weights leading to neuron i. Framing this as an EKF problem, $\boldsymbol{G}_i$ would denote the Kalman gain, $\boldsymbol{K}_i$ would denote the error covariance matrix, $\boldsymbol{Q}_i$ would represent the covariance matrix of the process noise and $\boldsymbol{R}$ is the covariance matrix of the measurement noise. Finally, the Jacobian $\boldsymbol{C}_i$ would represent the partial derivatives (which can be computed using backpropagation or backpropagation through time) of the function defining the output $\boldsymbol{y}$ of the network such that

$$\boldsymbol{C}_i = \begin{pmatrix} \frac{\partial y_1}{\partial w_{i1}}(t) & \frac{\partial y_1}{\partial w_{i2}}(t) & \dots & \frac{\partial y_1}{\partial w_{in_i}}(t) \\ \frac{\partial y_2}{\partial w_{i1}}(t) & \frac{\partial y_2}{\partial w_{i2}}(t) & \dots & \frac{\partial y_2}{\partial w_{in_i}}(t) \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial y_{n_y}}{\partial w_{i1}}(t) & \frac{\partial y_{n_y}}{\partial w_{i2}}(t) & \dots & \frac{\partial y_{n_y}}{\partial w_{in_i}}(t) \end{pmatrix} \tag{47}$$

The authors applied this update rule (replacing gradient descent based algorithms) on LSTMs. Specifically, they observed that training based on DEKFs resulted in faster convergence and better performance. However, they did note that the time complexity of DEKF weight updates increases per example. For typical interesting neural networks in practice, however, thousands of data samples are required in order for the network to learn good predictions. While DEKFs as a weight update rule may converge faster, it might end up taking even longer to converge in terms of real computation seconds in practical cases.

V. Experiments

This section will directly compare the performance between Kalman Filters and Neural Networks in a tracking task. There will be two tasks - the Univariate non-stationary growth model (UNGM) and a simulation of tracking the position of a differential 2-wheel drive robot. In each of these tasks, neural network models will be implemented in Keras with a Tensorflow backend.

A. UNGM

The Univariate non-stationary growth model (UNGM) [24] is noted for its high degree of nonlinearity and bimodality of measurements, which causes difficulties for filters. The model is specified as

$$x_k = 0.5x_{k-1} + 25\frac{x_{k-1}}{1+x_{k-1}^2} + 8\cos(1.2(k-1)) + w_k$$
$$z_k = \frac{x_k^2}{20} + v_k \tag{48}$$

where $w_k \sim \mathcal{N}(0, Q), v_k \sim \mathcal{N}(0, R)$ (traditionally $Q = R = 1$). In our simulations, $T = 50$. Note that due to the nonlinearities of the UNGM, we do not expect a vanilla Kalman Filter to work well. Therefore, we will be comparing the performance of the EKF, UKF, LSTM, and RNN. The LSTM and RNN models will each consist of a LSTM (or RNN) input layer with Dropout, and a Dense output layer. We will compare performance for varying numbers of units, specifically comparing 10 and 100 units. The LSTM (RNN) model with 10 LSTM (RNN) units will be denoted lstm10 (rnn10), while the LSTM (RNN) model with 100 LSTM (RNN) units will be denoted lstm100 (rnn100). To train the LSTM and RNN units, 1000 training samples are generated, where $x_0 \sim \mathcal{N}(0, \mathcal{X})$. Each are trained with 1000 epochs, with a validation set of size 330, meaning that for each epoch, the network is trained on 670 samples. MSE will be used as the metric for evaluation, with 1000 simulations for evaluation.

B. Position of differential drive 2-wheel robot

For this task, we will be comparing the performance of an Extended Kalman Filter and a LSTM-based network in the context of estimating the position of a simulated differential drive 2-wheel robot. In this simulation, a 2-wheeled robot randomly moves around in an (x, y) plane. There are 5 landmarks which the robot can detect with its sensors. It uses an encoder to create a sense of the velocity and direction of its wheels, providing $(v_l^t, v_r^t, x_{odom}^t, y_{odom}^t, \theta_{odom}^t)$ where, at time t,

- v_l^t, v_r^t refer to the detected velocities by the encoder of the left and right wheels (which are subjected to Gaussian noise),
- $x_{odom}^t = x_{odom}^{t-1} + \left(\frac{v_l^t + v_r^t}{2}\sin\left(\theta_{odom}^{t-1}\right)\right)$
- $y_{odom}^t = y_{odom}^{t-1} + \left(\frac{v_l^t + v_r^t}{2}\cos\left(\theta_{odom}^{t-1}\right)\right)$
- $\theta_{odom}^t = \theta_{odom}^{t-1} + \frac{v_l + v_r}{L}$, L is the distance between the wheels.

It uses the ranger to get a range and bearing to each landmark, providing $(\boldsymbol{r}, \boldsymbol{\phi})$, where $\boldsymbol{r} = \{r_i\}_{i=1}^5$ refers to the range and $\boldsymbol{\phi_i} = \{\phi_i\}_{i=1}^5$ refers to the bearing, and i refers to the landmark.
The neural network will consist of a layer of 500 LSTM units and dropout, and a dense layer output. It will be provided only the sensor information, i.e., $(v_l, v_r, x_{odom}, y_{odom}, \theta_{odom}, \boldsymbol{r}, \boldsymbol{\phi})$, as well as the true locations of the landmarks. 100,000 simulations of 15 seconds and a timestep size of 0.1s (150 total time steps) will be created, each starting at position $(0, 0)$.

VI. Results

A. UNGM

A table below is shown of the MSE of the different models applied to the UNGM task, where $Q = R = \mathcal{X} = 1$.

Model	MSE
EKF	49.622 ± 319.2
UKF	1.2784 ± 0.3633
LSTM10	1.5947 ± 0.2207
LSTM100	1.5814 ± 0.2515
RNN10	1.4081 ± 0.1657
RNN100	1.5549 ± 0.2717

Note the error in the EKF - from simulations, we noticed that the EKF would tend to "spike" at different time steps. This contributed greatly to the large MSE values, but otherwise, the EKF seemed to do a good job at following the function's trajectory. The results in a single trial is shown in figure 4. During the training phase, the networks were trained on 1000 epochs. The networks all converged (the training and validation MSE were both not decreasing significantly), but the networks with 100 units converged much faster. Overall, for the UNGM task, all models performed comparably except for the EKF.
In order to test different scenarios with varying signal-to-noise ratios, another simulation was run, except where $Q = 5, R = 20, \mathcal{X} = 10$. The UKF resulted in an MSE of 4.934 ± 1.414 while LSTM100 resulted in an MSE of 3.524 ± 0.444 (the LSTM/RNNs had very similar results

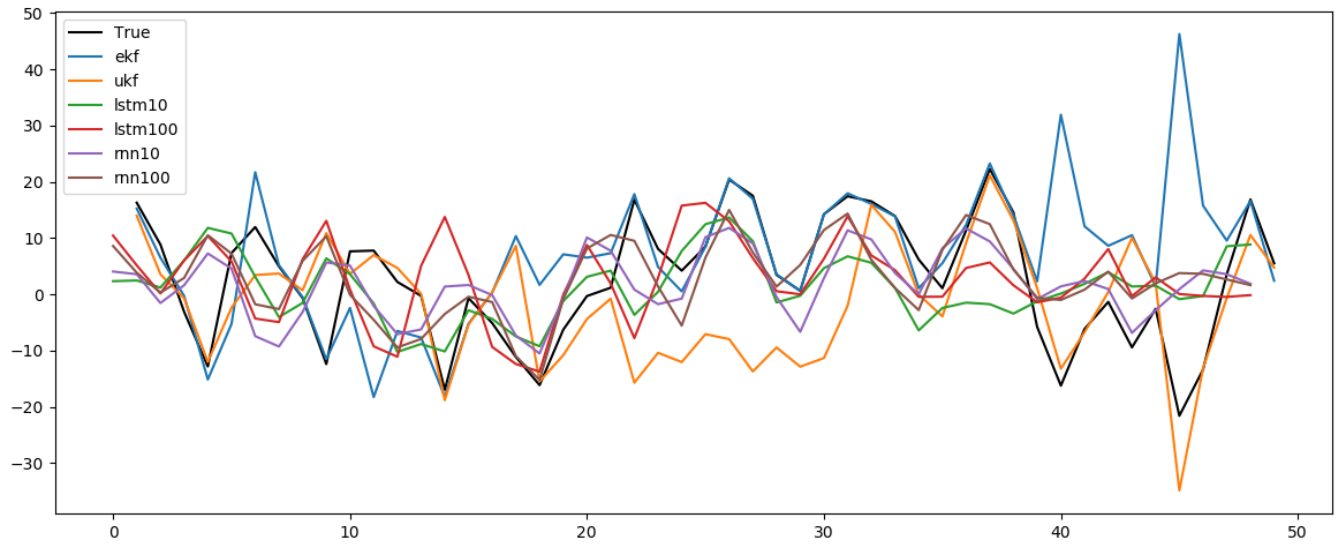

Fig. 4. UNGM results for one trial

while the EKF performed poorly again). While from these numbers, it may seem that the LSTM outperformed the UKF, Figure 5 shows a plot of the two compared in a single trial. We note that the LSTM basically estimated the mean, which turned out to minimize the MSE while the UKF was "overshooting" constantly. In this specific case, where the noise is greater, one might actually prefer the UKF predictions over the LSTM predictions despite the LSTM resulting in a lower MSE.

B. Position of differential drive 2-wheel robot

The code used for the robot simulation and the Kalman Filter was adapted from [25]. For 500 simulations, we achieved a RMSE of 0.876 ± 0.550 for the Kalman Filter and a RMSE of 5.11 ± 1.777 for the LSTM model. The LSTM took a long time to train, and the RMSE reported was on a model that had only been trained for 30 epochs and a training size of 75,000 simulations. Due to time constraints, we weren't able to continue training the model until it converged to a good solution - we do note that the RMSE continually decreased over epochs - for instance, at around 10 epochs, the LSTM was reporting an RMSE of around 11, and around 8 at 20 epochs. We are confident that with more training time, the LSTM would eventually converge to a solution with comparable performance to the Kalman filter.

However, we note that there were many problems with the LSTM approach. At first, we were training on 5,000 samples, and noticed that the model began to overfit - the training MSE would decrease while the validation loss did not. We continued to increase the sample size until we decided on 100,000 samples. This is a problem in a practical setting - getting 100,000 samples of something is not easily attainable. If an engineer only had access to 1,000 samples of a robot in this setting, then the Kalman Filter would out-perform the LSTM. Further, even if ample data is provided, the network would still take a long time to converge to a good solution. Finally, with better tuned hyperparameters and even architectural changes, the network could have converged to a better solution faster. However, due to the complexity of neural networks, these changes aren't obvious as deep learning is still a relatively new field.

On the other hand, the neural network did not need to know the structure of the underlying problem. It did not need to know where the sensor data came from, but instead would learn the relations over training samples. Constructing the neural network then became a problem of knowing how to construct the appropriate network given the sensor information, whereas constructing a Kalman Filter required knowledge of the sensor behavior to create an accurate model. While we were unable to create a more complex problem due to time

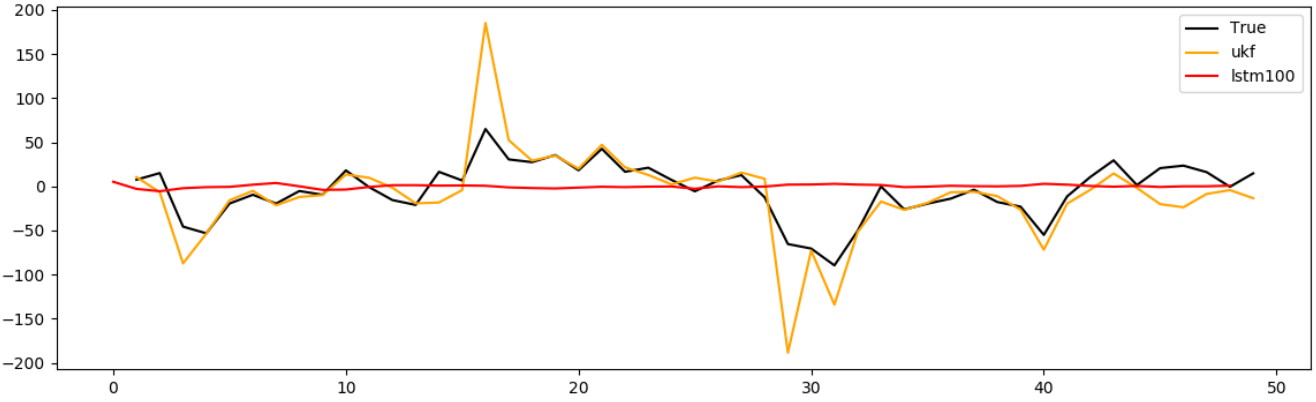

Fig. 5. Filter results for a noisy UNGM model

constraints, we would expect that a neural network would outperform the Kalman Filter given more complex tasks.

VII. Conclusions

Kalman Filters and Neural Networks were both originally designed to solve different problems. However, we have seen that the RNN and LSTM are both plausible alternatives to Kalman Filters. Each have their own requirements for providing accurate estimates. The Kalman Filter requires in-depth knowledge of the underlying system, including state transitions and noise covariances, while the RNN/LSTM require large amounts of data, a lot of computing power or training time, and knowledge of neural network architectures in order to create the optimal network for the problem. Further, the Kalman Filter, once its requirements are met, has theory-backed optimality conditions, while theory for neural networks is still underdeveloped. Ultimately, Kalman Filters should be used when possible, while neural networks are only more appropriate as the underlying problem becomes more complex.

References

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[2] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, November 2012.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[4] R. Socher, Y. Bengio, and C. D. Manning, "Deep learning for nlp (without magic)," in *Tutorial Abstracts of ACL 2012*, ACL '12, (Stroudsburg, PA, USA), pp. 5–5, Association for Computational Linguistics, 2012.

[5] F. A. Gers, D. Eck, and J. Schmidhuber, *Applying LSTM to Time Series Predictable through Time-Window Approaches*, pp. 669–676. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

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

[7] D. Iter, J. Kuck, and P. Zhuang, "Target tracking with kalman filtering, kjnn and lstms," 2016.

[8] A. G. Parlos, S. K. Menon, and A. Atiya, "An algorithmic approach to adaptive state filtering using recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 12, pp. 1411–1432, Nov 2001.

[9] P. Matisko and V. Havlena, "Optimality tests and adaptive kalman filter," *IFAC Proceedings Volumes*, vol. 45, no. 16, pp. 1523 – 1528, 2012. 16th IFAC Symposium on System Identification.

[10] B. Anderson and J. Moore, *Optimal Filtering*. Englewood, NJ: Prentice Hall, 1979.

[11] G. P. Huang, A. I. Mourikis, and S. I. Roumeliotis, "Analysis and improvement of the consistency of extended Kalman filter-based SLAM," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (Pasadena, CA), pp. 473–479, 2008.

[12] S. J. Julier, "A new extension of the kalman filter to nonlinear systems," *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*, vol. 3, 1997.
[13] E. A. Wan and R. V. D. Merwe, "The unscented kalman filter for nonlinear estimation," pp. 153–158, 2000.
[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[15] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.
[16] L. Bottou, *Large-Scale Machine Learning with Stochastic Gradient Descent*, pp. 177–186. Heidelberg: Physica-Verlag HD, 2010.
[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.
[18] C. Olah, "Understanding lstm networks," 2015.
[19] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, pp. 157–166, Mar. 1994.
[20] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. III–1310–III–1318, JMLR.org, 2013.
[21] P. Abbeel, A. Coates, M. Montemerlo, A. Y. Ng, and S. Thrun, "Discriminative training of kalman filters," in *in Proceedings of Robotics: Science and Systems*, 2005.
[22] P. Ondruska and I. Posner, "Deep tracking: Seeing beyond seeing using recurrent neural networks," *CoRR*, vol. abs/1602.00991, 2016.
[23] J. A. Prez-Ortiz, F. A. Gers, D. Eck, and J. Schmidhuber, "Kalman filters improve lstm network performance in problems unsolvable by traditional recurrent nets," *Neural Networks*, vol. 16, no. 2, pp. 241 – 250, 2003.
[24] "Performance analysis of improved iterated cubature kalman filter and its application to gnss/ins," *ISA Transactions*, 2016.
[25] S. Idelson, "Kalman-filter-simulator." https://github.com/Sanderi44/Kalman-Filter-Simulator.