A comparison of the Kalman filter and recurrent neural networks for state estimation of dynamical systems

by

Akihiro Takigawa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2023

© Akihiro Takigawa 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The study of dynamical systems is of great interest in many fields, with a wide range of applications. In some cases, these dynamical systems may be affected by noise and the availability of measurements may be limited. State estimations methods which can account for these challenges are valuable tools in analyzing these systems. While for linear systems the standard method is by using an algorithm called the Kalman filter, data-driven methods employing the versatility of artificial neural networks have also been proposed. In this thesis, we first introduce state estimation using the Kalman filter. Next, we provide an overview of a type of artificial neural network called recurrent neural networks (RNNs), which are particularly suited for tasks on time series data. We finally present the results of implementing RNN-based estimators for a number of dynamical systems with comparisons to Kalman filtering.

Table of Contents

Author's Declaration ii

Abstract iii

List of Figures vii

List of Tables ix

List of Algorithms x

1 Introduction 1

2 State Estimation 3

2.1 Weighted Least Squares 5

2.2 Recursive estimation 8

2.3 Process noise 11

2.4 The Kalman filter 14

2.5 Computational cost of the discrete-time Kalman filter 16

2.6 Extended Kalman filter 20

3 Recurrent Neural Networks 23

3.1 Introduction to RNNs 23

3.2 Computational graph of RNNs 25
3.3 The challenge of long-term dependencies 29
3.3.1 Skip Connections 31
3.3.2 Leaky Units 31
3.3.3 LSTMs 32

4 Training RNNs **37**
4.1 Gradient Descent 37
4.1.1 Stochastic Gradient Descent 40
4.1.2 Variants of Stochastic Gradient Descent 42
4.1.3 Adaptive learning rate methods 45
4.2 Backpropagation through time (BPTT) 48
4.3 Quantifying the performance of neural networks 53
4.4 Capacity of neural networks 54
4.5 Regularization 56
4.6 The curse of dimensionality and data selection 57
4.6.1 Persistent excitation 59
4.7 Hyperparameter optimization 61
4.7.1 Grid search and random search 63
4.7.2 Bayesian optimization 65

5 State estimation with RNNs and the Kalman filter **70**
5.1 Previous work 70
5.2 Methodology 71
5.3 Comparing computational cost to classical algorithms 74
5.4 Connected mass-spring-damper systems 77
5.5 Example: 5 DoF mass-spring-damper system 82
5.6 Example: 100 DoF mass-spring-damper system 85

5.7 Example: transfer learning to a clamped-free beam model 88
5.8 Example: randomly generated dynamical systems 97
5.8.1 Random dynamical system of order 20 98
5.8.2 Unstable random dynamical system of order 20 101
5.9 Example: Lorenz system 104

6 Conclusion **107**

List of Figures

3.1 Computational graph of a feedforward neural network, drawn in two different styles. On the left, we draw a node for every unit. On the right, the same neural network is shown more compactly by drawing a node for each vector representing a layer's activations, created by transformations applied by the weights W to x and w to h. Nonlinear activation functions map the connection between nodes. The arrows represent the direction of computation. [1], p. 174 26

3.2 The computational graph of a simple recurrent neural network. [1], p. 376 27

3.3 The unrolled computational graph of the recurrent neural network shown in Figure 3.2, across three time steps. [1], p. 376 28

3.4 A block diagram of a LSTM cell. [1], p. 409 33

4.1 The unrolled computational graph of a RNN. Each step of the input sequence x is transformed into the RNN hidden state h with weight matrix U. A second weight matrix V is used to transform the hidden state into the output o. Taking this as well as the labels y as parameters, the loss L is calculated. The recurrent weights W are applied to the hidden state h to create the recurrence relation across the number of steps contained in the input sequence. The unrolled (unfolded) representation across three time steps is shown on the right. The unrolled computational graph is a directed acyclic graph (DAG), allowing for recursive computation of the gradients of the loss function with respect to the weights of the RNN with BPTT. [1], p.378 49

4.2 A typical relationship between errors and capacity of neural networks. [1], p. 409 56

4.3 Plots of 20 data points, as we increase the dimension of the data space. The volume of the space represented grows quickly and the data becomes sparse. 57

4.4 Evolution of training (green) and validation (red) errors when training with persistently exciting data (left), compared to training with data that is not persistently exciting (right). 61

5.1 Free-body diagram of a two degree-of-freedom mass-spring-damper system. 78

5.2 Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set. . . . 84

5.3 Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set. . . . 87

5.4 A diagram of an clamped-free beam. The beam rotates by torque applied at the hub $x = 0$. [2], p. 180 89

5.5 Training and validation errors over 50 epochs for two RNN-based estimators on the clamped-free beam system. The first RNN was trained with default initial conditions. The second RNN was transferred the weights and biases from a trained RNN-based estimator for a five degree-of-freedom connected mass-spring-damper system prior to training. 95

5.6 Mean squared errors for the RNN-based estimator (green), RNN-based estimator initialized with weights from the RNN trained on the 5 DoF mass-spring-damper system (blue) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set. 96

5.7 Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set. . . . 100

5.8 Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 50 time steps contained in each sequence in the test set. . . . 103

5.9 Mean squared errors for the RNN-based estimator (green) and Extended Kalman filter (red) for the 2000 time steps contained in each sequence in the test set. 105

List of Tables

5.1 Table of values for RNN-based estimators on connected mass-spring-damper systems. 88
5.2 Table of values for RNN-based estimators on a clamped-free beam system. 97
5.3 Table of values for RNN-based estimators on randomly generated LTI systems. 102

List of Algorithms

1 Kalman filter 16
2 Extended Kalman filter 22
3 Batch Gradient Descent 40
4 Stochastic Gradient Descent 42
5 Mini-batch Gradient Descent 43
6 Mini-batch Gradient Descent with momentum 44
7 AdaGrad 46
8 RMSProp 46
9 Adam 48
10 Backpropagation through time 52
11 build_grad 52
12 Basic training loop 53

Chapter 1

Introduction

The study of dynamical systems is of great interest in many fields, with applications ranging from the modeling of orbital mechanics in physics, the structurization of macroeconomic processes in economics and the study of virus spread in epidemiology. The modeling of dynamical systems has enabled us to better understand the key processes which power these phenomena by providing a mathematical basis. It is often the case that our theoretical formulations do not directly coincide with what is observed in the real world due to the presence of noise in the processes or our measurements (or both). Measurements pose other important issues. Not only can the measurements be noisy, we may only be able to take a limited number of them. Furthermore, not every state of the dynamical system may be measured. Hence, techniques that can clean up noise, as well as piece together information from limited measurements to learn the true state of a dynamical system are valuable.

One popular technique to estimate the true state of dynamical systems is Kalman

filtering, pioneered by Swerling, Kalman and Bucy starting in the late 1950's [3]. One of its first major applications was in trajectory estimation, implemented in the navigation computer for NASA's Apollo program to land the first humans on the Moon. Since then, the Kalman filter has been applied to a wide range of problems, such as in the guidance of cruise missiles and more recently in navigation for autonomous vehicles.

However, artificial neural networks have become a popular choice in approximating functions. The first arbitrary-width univeral approximation theorems were proven in the late 1980's, showing that artificial neural networks with as few as one hidden layer can approximate a broad class of real-valued function to an arbitrary degree of accuracy [4]. The versatility of artificial neural networks in their ability to learn almost any kind of process has led to its widespread adoption for a variety of use cases in both academia and industry, including for state estimation.

We introduce these two methods in the context of estimating the states of noisy dynamical systems, focusing on linear systems. We first introduce state estimation with the Kalman filter. Subsequently, we introduce a form of artificial neural network called the recurrent neural network, which operates on sequential data. We also provide an overview of the training process, which enables the neural network to learn to estimate the states of noisy dynamical systems. Finally, we present results in comparing the performance of each method on a number of examples.

Chapter 2

State Estimation

In this chapter, we introduce state estimation for linear dynamical systems. The descriptions of weighted least squares estimation, recursive estimation, and the Kalman filter are based on [5], Chapter 3, and can be found in more detail there.

State estimation aims to determine the internal state of a system, using both a mathematical model in the form of a set of equations describing the physical system and measured data of the system's inputs and outputs. In many real-world applications, knowledge of the internal state of a physical system is essential. For example, in designing a rocket for aeronautical use, an engineer may want to know the internal temperature of its engine core during flight to control the heat degradation of its respective components. A naive solution to this problem would be to place sensors everywhere, and on everything. However, in many applications this is not practical, whether it may be due to cost, or the unavailability of an appropriate sensor. Hence, we seek to determine the internal state of

a physical system from a limited number of imperfect measurements. A framework which allows us to accomplish this is called an estimator.

In mathematical terms, given some system, suppose we have the following mathematical model:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k), \\ y_k &= h(x_k, u_k, k), \end{aligned} \tag{2.1}$$

where x_{k+1}, the state at time $k+1$, depends on a function f applied to x_k, the state at time k, and u_k, the input at time k. Similarly, the measurement y_k depends on a function h applied to x_k and u_k. We seek to determine x_k, the internal state at time k. A state estimator places this mathematical model in parallel with the physical system, and fed the same inputs u_k, produces $\hat{x}_k$, an estimate of the internal state. As the mathematical model is imperfect, its output $\hat{y}_k$ will contain errors when compared to the measured value of the physical system, y_k. The error $\epsilon_y = y_k - \hat{y}_k$ can be fed back into the mathematical model to correct the error. Thus, an estimator can be robust to model inaccuracies and measurement noise, and is an indispensable tool for many real-world problems. In this paper, we focus on estimation for discrete-time linear systems, that is, discrete-time systems where the relationship between the state x_k and the output y_k is a linear mapping. This can be modeled in the form

$$\begin{aligned} x_{k+1} &= A_k x_k + B_k u_k + w_k, \\ y_k &= H_k x_k + D_k u_k + v_k, \end{aligned} \tag{2.2}$$

where at time k, $x_k \in \mathbb{R}^n$ is the state vector, $A_k \in \mathbb{R}^{n\times n}$ is the state (process) matrix, $y_k \in \mathbb{R}^m$ is the output (measurement) vector, $H_k \in \mathbb{R}^{m\times n}$ is the output (measurement) matrix, $u_k \in \mathbb{R}^r$ is the input vector, $B_k \in \mathbb{R}^{n\times r}$ is the input matrix, and $D \in \mathbb{R}^{m\times r}$ is the feedthrough matrix. We also have $w_k \in \mathbb{R}^n$, the process noise at time k and $v_k \in \mathbb{R}^m$, the measurement noise at time k. The model (A_k, B_k, H_k, D_k), the control u_k and measurements y_k are assumed known. The disturbances w_k, v_k and the initial condition x_0 are not known. Given a model (A_k, B_k, H_k, D_k), the state x_k and measurement y evolves over time. In the subsequent subsections, we construct one example of an estimator known as the Kalman filter, which is a widely used estimator for linear systems. Under the assumption that noises in the system are Gaussian, the Kalman filter minimizes the mean squared error (MSE) between the measured state and true state of the system, and is considered to be an optimal estimator for linear systems.

2.1 Weighted Least Squares

Before we delve into the details of the Kalman filter, we first consider the simpler problem of estimating a vector $x \in \mathbb{R}^n$ from a noisy measurement $y \in \mathbb{R}^m$. This is described by the following equation:

$$y = Hx + v, \tag{2.3}$$

where the measurement y is a linear combination of x and measurement noise v. To simplify our calculations, we assume H is invertible. Suppose we would like to construct an estimator to produce an estimate $\hat{x}$. Our goal is to minimize the error between the

measurement y and and the estimated output $H\hat{x}$, given by

$$\epsilon_y = y - H\hat{x}. \tag{2.4}$$

In the case that the covariance of the noise is the same for all elements of y, the most probable value of the vector x is the value of $\hat{x}$ that minimizes the sum of squares between y and $H\hat{x}$. However, the covariance of the noise may be different for each element of y, with each of the m elements of y having standard deviations $\sigma_1, \ldots, \sigma_m$, so y has a covariance matrix

$$R = \begin{bmatrix} \sigma_1^2 & & 0 \\ & \ddots & \\ 0 & & \sigma_m^2 \end{bmatrix}. \tag{2.5}$$

We assume the matrix R to be positive definite. Note that this is the case when R is a diagonal matrix with positive diagonal entries. Then, for example, if σ_1 is relatively large, then y_1 is a relatively noisy measurement, so minimizing ϵ_{y_1} is not as valuable as minimizing the other elements of ϵ_y. This leads us to consider the cost function

$$J = \frac{\epsilon_{y_1}^2}{\sigma_1^2} + \cdots + \frac{\epsilon_{y_m}^2}{\sigma_m^2},$$

known as the weighted sum of squares. Minimizing this cost would place more value in relatively accurate measurements, as opposed to noisy measurements. The method of using the weighted sum of squares for estimation is known as the weighted least squares method.

We can rewrite our cost function as

$$\begin{aligned} J &= \epsilon_y^T R^{-1} \epsilon_y \\ &= (y - H\hat{x})^T R^{-1} (y - H\hat{x}) \\ &= y^T R^{-1} y - \hat{x}^T H^T R^{-1} y - y^T R^{-1} H\hat{x} + \hat{x}^T H^T R^{-1} H\hat{x}. \end{aligned} \tag{2.6}$$

Recalling that R is a positive definite diagonal matrix, for vectors $a \in \mathbb{R}^n$ and $b = Ha$, we can write

$$a^T H^T R^{-1} H a = (Ha)^T R^{-1} H a = b^T R^{-1} b. \tag{2.7}$$

Since R is positive definite, so is R^{-1}. Then, $b \neq 0$ implies $b^T R^{-1} b > 0$ and $b = 0$ implies $b^T R^{-1} b = 0$, so $b^T R^{-1} b \geq 0$ for all $b \in \mathbb{R}^m$. Thus, $H^T R^{-1} H$ is positive semi-definite.

We now take the derivative of our cost function J and set it to zero:

$$\frac{\partial J}{\partial \hat{x}} = -2y^T R^{-1} H + 2\hat{x}^T H^T R^{-1} H = 2(-y + H\hat{x})^T R^{-1} H = 0. \tag{2.8}$$

This yields

$$\hat{x} = (H^T R^{-1} H)^{-1} H^T R^{-1} y. \tag{2.9}$$

Furthermore, taking the second derivative gives us

$$\frac{\partial^2 J}{\partial \hat{x}^2} = 2H^T R^{-1} H \geq 0, \tag{2.10}$$

as we have previously shown $H^T R^{-1} H$ to be positive semi-definite. Hence, (2.9) is the unique minimum of J.

2.2 Recursive estimation

We now consider the more advanced problem of estimation on time-evolving measurements. Suppose the measurement y now changes with time, described by the equation

$$y_k = H_k x_k + v_k, \tag{2.11}$$

where the subscript k denotes a variable at time step k. The signal v_k is Gaussian measurement noise with mean zero and a known covariance matrix R_k at time step k. While it is possible to produce an estimate using the weighted least squares method, this is inefficient due to the need to recalculate our estimate by first producing a new weighted sum of squares cost function, then taking its first derivative at each time step. Instead, we introduce a method known as recursive estimation, where we instead progressively improve the estimate produced in the first time step with new information gleaned from subsequent measurements. Hence, we now seek to minimize the *expected value* of the estimation error $x_k - \hat{x}_k$.

Given a measurement

$$y_k = H_k x_k + v_k, \tag{2.12}$$

a linear recursive estimator has the form

$$\hat{x}_k = \hat{x}_{k-1} + K_k(y_k - H_k \hat{x}_{k-1}). \tag{2.13}$$

At each time step k, the measurement matrix H_k is known but x_k and v_k are unknown. The

vector K_k is called the gain, whose construction differs depending on the kind of recursive estimator. At each time step k, an updated estimate $\hat{x}_k$ is produced by multiplying the gain with the term $y_k - H_k \hat{x}_{k-1}$, called the correction term. We assume that the expected value of the measurement noise v_k, $\mathbb{E}(v_k) = 0$ at each time step. Then, the mean estimation error produced by our linear recursive estimator is

$$\begin{aligned}
\mathbb{E}(x_k - \hat{x}_k) &= \mathbb{E}(x_k - (\hat{x}_{k-1} + K_k(y_k - H_k \hat{x}_{k-1}))) \\
&= \mathbb{E}(x_k - \hat{x}_{k-1} - K_k(y_k - H_k \hat{x}_{k-1})) \\
&= \mathbb{E}(x_k - \hat{x}_{k-1} - K_k((H_k x_k + v_k) - H_k \hat{x}_{k-1})) \\
&= \mathbb{E}(x_k - \hat{x}_{k-1} - K_k H_k (x_k - \hat{x}_{k-1}) - K_k v_k) \\
&= \mathbb{E}(x_k - \hat{x}_{k-1}) - K_k H_k \, \mathbb{E}(x_k - \hat{x}_{k-1}) - K_k \, \mathbb{E}(v_k) \\
&= (I - K_k H_k) \, \mathbb{E}(x_k - \hat{x}_{k-1}) - K_k \, \mathbb{E}(v_k) \qquad (2.14)
\end{aligned}$$

The difference between the expected value of the estimate and the expected true value is known as the estimator's *bias*, denoted $\mathbb{B}(\hat{x}_k) = \mathbb{E}(\hat{x}_k) - \mathbb{E}(x_k)$. In the case of the linear recursive estimator, if $\mathbb{E}(x_k - \hat{x}_{k-1}) = 0$ and $\mathbb{E}(v_k) = 0$, then $\mathbb{E}(x_k - \hat{x}_k) = 0$. In other words, if our initial estimate $\hat{x}_0$ is equal to $\mathbb{E}(x_k)$ and we have mean zero measurement noise v_k, then the mean estimation error $\mathbb{E}(x_k - \hat{x}_k) = 0$, implying $\mathbb{B}(\hat{x}_k) = 0$. Estimators satisfying this are called *unbiased estimators*. Hence, if $\mathbb{E}(x_k) = \hat{x}_0$ and the measurement noise is zero mean and independent as in the previous section, minimizing the square of the expected value of the estimation error is equivalent to minimizing the weighted sum of squares cost because the most probable value of x_k (in other words, $\mathbb{E}(x_k)$) is the $\hat{x_k}$ which minimizes the weighted sum of squares cost.

We now turn our attention to determining the appropriate Kalman gain which minimizes the square of the expected value of the estimation error. Rewriting the cost at time step k gives

$$\begin{aligned} J_k &= \mathbb{E}(||x - \hat{x}_k||^2) \\ &= \mathbb{E}((x - \hat{x}_k)^T (x - \hat{x}_k)) \\ &= \mathbb{E}(\mathrm{Tr}((x - \hat{x}_k)^T (x - \hat{x}_k))) \\ &= \mathrm{Tr}(\mathbb{E}((x - \hat{x}_k)^T (x - \hat{x}_k))) \\ &= \mathrm{Tr}(P_k). \end{aligned} \tag{2.15}$$

The matrix P_k is the *covariance matrix of the estimation error* at time step k. We now formulate a recursive representation of P_k:

$$\begin{aligned} P_k &= \mathbb{E}((x - \hat{x}_k)(x - \hat{x}_k)^T) \\ &= \mathbb{E}(((I - K_k C_k)(x - \hat{x}_{k-1}) - K_k v_k)((I - K_k C_k)(x - \hat{x}_{k-1}) - K_k v_k)^T) \\ &= (I - K_k C_k)\, \mathbb{E}((x - \hat{x}_{k-1})(x - \hat{x}_{k-1})^T)(I - K_k C_k)^T \\ &- K_k\, \mathbb{E}(v_k (x - \hat{x}_{k-1})^T)(I - K_k C_k)^T - (I - K_k C_k)\, \mathbb{E}((x - \hat{x}_{k-1}) v_k^T) K_k^T \\ &+ K_k\, \mathbb{E}(v_k v_k^T) K_k^T. \end{aligned} \tag{2.16}$$

If we assume that the measurement noise v_k has mean zero, then the estimation error at time step $k - 1$ is independent of v_k. This implies

$$\mathbb{E}(v_k (x - \hat{x}_{k-1})^T) = \mathbb{E}(v_k)\, \mathbb{E}((x - \hat{x}_{k-1})^T) = 0. \tag{2.17}$$

Hence we can rewrite our P_k as

$$P_k = (I - K_k H_k) P_{k-1} (I - K_k H_k)^T + K_k R_k K_k^T, \tag{2.18}$$

where $R_k = \mathbb{E}(v_k v_k^T)$ denotes the covariance matrix of the measurement noise at time step k.

We proceed to calculate the Kalman gain which minimizes the cost by taking its derivative with respect to the gain,

$$\frac{\partial J_k}{\partial K_k} = 2(I - K_k H) P_{k-1} (-H^T) + 2K_k R.$$

Setting this to zero and solving for K_k yields a recursive expression for the optimal Kalman gain,

$$K_k = P_{k-1} H^T (H P_{k-1} H^T + R)^{-1}. \tag{2.19}$$

Substituting the optimal K_k into (2.13) enables us to recursively produce state estimates from measurements.

2.3 Process noise

We finalize our discussion of the building blocks of the Kalman filter by addressing the issue of deriving the covariance of the estimation error when the true state is also subject

to noise. Suppose we are given a linear discrete-time system

$$x_k = A_k x_{k-1} + w_k, \tag{2.20}$$

where w_k is a zero-mean Gaussian process noise with covariance matrix Q_k. Then, the evolution of $\hat{x}_k$, the state estimate at time step k is described by

$$\hat{x}_k = A_k \hat{x}_{k-1}, \tag{2.21}$$

since we assume the process noise w_k has mean zero. The covariance matrix P_k of the estimation error at time step k is

$$\begin{aligned}
P_k &= \mathbb{E}((x - \hat{x}_k)(x - \hat{x}_k)^T) \\
&= \mathbb{E}((A_k x_{k-1} + w_k - A_k \hat{x}_{k-1})(A_k x_{k-1} + w_k - A_k \hat{x}_{k-1})^T) \\
&= \mathbb{E}((A_k(x_{k-1} - \hat{x}_{k-1}) + w_k)(A_k(x_{k-1} - \hat{x}_{k-1}) + w_k)^T) \\
&= \mathbb{E}(A_k(x_{k-1} - \hat{x}_{k-1})(x_{k-1} - \hat{x}_{k-1})^T A_k^T + A_k(x_{k-1} - \hat{x}_{k-1})w_k^T + w_k(x_{k-1} - \hat{x}_{k-1})^T A_k^T + w_k w_k^T) \\
&= A_k \, \mathbb{E}((x_{k-1} - \hat{x}_{k-1})(x_{k-1} - \hat{x}_{k-1})^T) A_k^T + \mathbb{E}(A_k(x_{k-1} - \hat{x}_{k-1})w_k^T) + \mathbb{E}(w_k(x_{k-1} - \hat{x}_{k-1})^T A_k^T) \\
&\quad + \mathbb{E}(w_k w_k^T).
\end{aligned} \tag{2.22}$$

Since the estimation error at time step $k-1$ is independent of the process noise w_k at time step k,

$$\begin{aligned}\mathbb{E}(A_k(x_{k-1}-\hat{x}_{k-1})w_k^T) &= A_k\,\mathbb{E}(x_{k-1}-\hat{x}_{k-1}))\,\mathbb{E}(w_k^T) = 0,\\ \mathbb{E}(w_k(x_{k-1}-\hat{x}_{k-1})^T A_k^T) &= \mathbb{E}(w_k)\,\mathbb{E}(x_{k-1}-\hat{x}_{k-1})^T)A_k^T = 0.\end{aligned} \tag{2.23}$$

Substituting this into (2.22) gives us

$$\begin{aligned}P_k &= A_k\,\mathbb{E}((x_{k-1}-\hat{x}_{k-1})(x_{k-1}-\hat{x}_{k-1})^T)A_k^T + \mathbb{E}(w_k w_k^T)\\ &= A_k P_{k-1} A_k^T + Q_k.\end{aligned} \tag{2.24}$$

When the matrices A and Q are constant, this becomes

$$P = APA^T + Q, \tag{2.25}$$

called the discrete Lyapunov equation. It is shown in [5], Section 4.1 that when $\lambda_i\lambda_j \neq 1$ for all $i, j = 1, \ldots, n$ where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A, this has a unique solution for any Q. Specifically, if the system is stable (i.e. states do not explode over time), then the eigenvalues of A will have magnitudes less than one so this condition will always be satisfied. Hence, for stable systems we can always solve the discrete Lyapunov equation to obtain a unique P.

2.4 The Kalman filter

In this section we present the Kalman filter algorithm [6] for linear discrete-time systems. Note that there also exists a Kalman filter which can be applied to linear continuous-time systems. However, we will not delve into this here, and will focus on linear discrete-time systems.

Given a linear discrete-time system:

$$\begin{aligned} x_k &= A_k x_{k-1} + w_k \\ y_k &= H_k x_k + v_k, \end{aligned} \tag{2.26}$$

where $x_k \in \mathbb{R}^n$ is the state at time k, $A_k \in \mathbb{R}^{n \times n}$ is the process matrix at time k, $y_k \in \mathbb{R}^m$ is the measurement at time k, $H_k \in \mathbb{R}^{m \times n}$ is the measurement matrix at time k. We assume w_k, the process noise at time k and v_k, the measurement noise at time k are uncorrelated white noises with zero mean and known covariance matrices Q_k and R_k respectively, that is,

$$\begin{aligned} w_k &\sim N(0, Q_k), \\ v_k &\sim N(0, R_k). \end{aligned} \tag{2.27}$$

The discrete-time Kalman filter applied to the above system will filter out the noise and provide an optimal estimate of the true state. At each time step, the algorithm uses only the updated estimates from the previous step and the given system dynamics. Each

time step consists of two phases, the Predict phase and the Update phase. In the Predict phase, the algorithm produces an initial state estimate x_k^- and estimation error covariance matrix P_k^- at time step k from those of the previous time step $k-1$, filtering the process noise w as described in section 2.3. In the Update phase, the initial state estimate and estimation error covariance matrix are corrected recursively to account for measurement error by calculating the Kalman gain as described in section 2.2, producing the updated state estimate $\hat{x}_k^+$ and estimation error covariance matrix P_k^+ at time step k. These values can then be fed back to the Predict phase at time step $k+1$, allowing the two phases to be repeated indefinitely to produce state estimates for an arbitrary number of time steps. When the process model is constant, the initial prediction of the estimation error covariance P^- simplifies to solving the discrete Lyapunov equation. If the solvability condition for the discrete Lyapunov equation is met, this will have a unique solution. Hence, the Kalman filter will converge to produce the same mean squared error regardless of initial conditions in this case. The Kalman filter algorithm is shown in Algorithm 1.

A key feature of the Kalman filter is its ability to "decide" the degree of confidence it places in the measurement or the model when producing the updated state estimate. If the measurement is to be trusted more than the model, then the norm $||R||$ should be lower than $||Q||$. Hence the greater the trust in the measurement more than the model, the greater the ratio $||Q||/||R||$ will become. This decision is captured in the calculation of the Kalman gain in the update phase.

Algorithm 1 Kalman filter

Require: Initial state estimate $\hat{x}_0^+$
Require: Initial covariance matrix of the estimation error P_0^+
  $k \leftarrow 1$
  while not stopped **do**
    Predict:
    Predict state estimate, $\hat{x}_k^- = A_k \hat{x}_{k-1}^+$
    Predict covariance of the estimation errors, $P_k^- = A_k P_{k-1}^+ A_K^T + Q_k$
    Update:
    Calculate the optimal Kalman gain, $K_k = P_k^- H_{k-1}^T (H_{k-1} P_k^- H_{k-1}^T + R_k)^{-1}$
    Update state estimate, $\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k \hat{x}_k^-)$
    Update covariance of the estimate, $P_k^+ = (I - K_k H_k) P_k^-$
    $k \leftarrow k + 1$
  end while

2.5 Computational cost of the discrete-time Kalman filter

Here, we derive the computational cost of the Kalman filter shown in Algorithm 1 by deriving its asymptotic computational complexity. Note that there are two main notions of computational complexity– time complexity and space complexity. Time complexity describes the amount of computation time an algorithm requires to complete, typically by counting the number of elementary operations that the algorithm needs to perform, under the assumption that each elementary operations requires a fixed amount of time. Space complexity describes the amount of memory space required by the algorithm, similarly found by counting the number of variables the algorithm needs to save, under the assumption that each variable requires a fixed amount of memory space. For both computational complexities, the worst case behavior, that is, the maximum amount of resources that are

needed over all inputs of size n, expressed in asymptotic notation is considered.

In this work, we will focus on the time complexity as the measure of computational cost because an algorithm's space complexity is bounded by its time complexity. Any change to an algorithm which increases its space complexity will require the algorithm to write to the additional spaced used, also increasing its time complexity. We start with calculating the time complexity of the state update,

$$\hat{x}_k^- = A_k \hat{x}_{k-1}^+, \tag{2.28}$$

where $A_k \in \mathbb{R}^{n \times n}$ is the system matrix at time step k, $\hat{x}_k^-, \hat{x}_{k-1}^+ \in \mathbb{R}^n$ are state estimates at time steps k and $k-1$, respectively. Calculation of $A_k \hat{x}_{k-1}^+$ has time complexity $O(n^2)$, due to the properties of matrix-vector multiplication. Next, for the prediction of the covariance matrix of the estimation error,

$$P_k^- = A_k P_{k-1}^+ A_k^T + Q_k, \tag{2.29}$$

where $P_k^-, P_{k-1}^+ \in \mathbb{R}^{n \times n}$ are respectively the predicted covariance matrix of the estimation error at time step k and the corrected covariance matrix of the estimation error at time step $k-1$, and $Q_k \in \mathbb{R}^{n \times n}$ is the covariance matrix of the process noise at time step k. The computation of $A_k P_{k-1}^+ A_k^T$ first requires multiplication of two $n \times n$ matrices $A_k P_{k-1}^+$, and then the multiplication of the resulting $n \times n$ matrix with another $n \times n$ matrix A_k^T. As the multiplication of two $n \times n$ matrices has time complexity $O(n^3)$, the time complexity of the matrix multiplications at this step is $O(2n^3)$. (Note: while Strassen's algorithm [7] is

a more efficient matrix multiplication algorithm which has seen some adoption in publicly available scientific computing software, we use schoolbook matrix multiplication because the efficiency of Strassen's algorithm does not apply to sparse matrix multiplication and it is less numerically stable [8], reducing its generality.) Adding to this the time complexity $O(n^2)$ of the matrix transpose A_k^T, as well as the addition of the n^2 elements of Q_k results in $O(2n^3 + 2n^2)$ time complexity for this step. The "Predict" phase of the Kalman filter algorithm has a total time complexity of $O(2n^3 + 3n^2)$.

We now look at the "Update" phase of the Kalman filter algorithm. We first calculate the optimal Kalman gain,

$$K_k = P_k^- H_{k-1}^T (H_{k-1} P_k^- H_{k-1}^T + R_k)^{-1}, \tag{2.30}$$

recalling that $H_{k-1} \in \mathbb{R}^{m \times n}$ is the measurement matrix at time step $k-1$, and $R_k \in \mathbb{R}^{m \times m}$ is the covariance matrix of the measurement error at time step k. From the properties of matrix multiplication we get time complexity $O(mn^2)$ for $H_{k-1}P_k^-$, so the matrix multiplications in $H_{k-1}P_k^- H_{k-1}^T$ have time complexity $O(mn^2 + m^2 n)$. The matrix transpose H_{k-1}^T, adds another $O(n^2)$ in time complexity, assuming $m \leq n$. Combined with the $O(m^2)$ time complexity of adding R_k, the time complexity of $H_{k-1}P_k^- H_{k-1}^T + R_k$ is $O(mn^2 + m^2 n + n^2)$. We proceed to invert this $n \times n$ matrix, which requires $O(n^3)$ time, resulting in the time complexity of $O(mn^2 + m^2 n + n^2 + n^3)$ to compute $(H_{k-1}P_k^- H_{k-1}^T + R_k)^{-1}$. Now, $P_k^- H_{k-1}^T$ requires $O(mn^2 + n^2)$ time, and since the multiplication of $n \times m$ and $m \times m$ matrices $P_k^- H_{k-1}^T$ and $(H_{k-1}P_k^- H_{k-1}^T + R_k)^{-1}$ requires $O(m^2 n)$ time, the time complexity of this step is $O(n^3 + 2n(m^2 + mn + n))$.

We now apply the Kalman gain to produce the updated state estimate,

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k\hat{x}_k^-), \tag{2.31}$$

where $y_k \in \mathbb{R}^m$ is the measurement at time step k. Since the operations required to compute $y_k - H_k\hat{x}_k^-$ are one matrix-vector multiplication and one addition, we require $O(m + mn)$ time here. The multiplication of a $n \times m$ matrix with a $m \times 1$ vector in $K_k(y_k - H_k\hat{x}_k^-)$ results in the time complexity $O(m+2mn)$. Finally, the addition of $\hat{x}_k^-$ requires $O(n)$ time, so we require $O(m + n + 2mn)$ time to compute this step.

In the final step, we calculate the time complexity of updating the covariance matrix of the estimation error,

$$P_k^+ = (I - K_kH_k)P_k^-. \tag{2.32}$$

First, K_kH_k requires $O(n^2m)$ time from the properties of matrix multiplication. Hence, the time complexity of $I - K_kH_k$ is $O((m+1)n^2)$. As $(I - K_kH_k)P_k^-$ is the multiplication of two $n \times n$ matrices, the total time complexity of this step is $O(n^3 + (m+1)n^2)$.

Summing the time complexities of each step, the time complexity of the "Update" step of the Kalman filter is $O(2n(m^2+n^2+m)+3n^2(m+1)+m+n)$. Adding this to the time complexity of the previous "Predict" step, the total cost of one iteration of the Kalman filter is

$$\begin{aligned} &O(2n^3 + 3n^2) + O(2n(m^2 + n^2 + m) + 3n^2(m+1) + m + n) \\ &\qquad = O(2n(m^2 + 2n^2 + m) + 3n^2(m+2) + m + n). \end{aligned} \tag{2.33}$$

Hence for large n, the time complexity of the Kalman filter scales roughly cubically with respect to the size of the state vector. The computation cost becomes higher for time-varying systems, as static lookup tables cannot efficiently capture behavior of dynamics that change with time. However, an alternative formulation using adaptive lookup tables can be found in [9].

Kalman filters are only applicable for linear dynamical systems. However, there are a number of algorithms for state of estimation of nonlinear systems. One such estimator called the Extended Kalman filter, which extends the Kalman filter for nonlinear systems by linearizing the dynamical system in both the Predict and Update steps.

2.6 Extended Kalman filter

Here we discuss an extension of the Kalman filter called the Extended Kalman filter (EKF) which is applicable to many nonlinear systems.

The Extended Kalman filter is a generalization of the Kalman filter to nonlinear systems. The idea of the EKF is to linearize the system at each time step, treating the system as a time-variant linear system. The EKF requires the Jacobian matrices of the nonlinear process and measurement models with respect to the state and noise to be available. Given

a nonlinear discrete-time system

$$
\begin{aligned}
x_k &= f(k, x_{k-1}, w_k), \\
y_k &= h(k, x_k, v_k), \\
w_k &\sim N(0, Q_k), \\
v_k &\sim N(0, R_k),
\end{aligned}
$$

The Extended Kalman filter algorithm is shown in Algorithm 2.

The EKF uses a first-order approximation to obtain a linear estimate of the system. This may fail to provide a good estimate when the system is highly nonlinear. Kalman filter extensions using higher order approximations exist and may be used in these cases, such as the Iterated EKF and the Second-Order EKF.

The time complexity of the EKF is bounded below by $O(9n^3 + 8n^2 + 2n)$, as it does everything the linear Kalman filter does. The computational cost of linearization depends on the numerical technique used. For example, using a finite difference method [10] such as the forward difference or backward difference methods would lead to an additional $O(n)$ time complexity for each linearization. Due to the popularity of the EKF, many implementations which improve computational cost have been proposed. See [11] for such a implementation for models of radar systems.

Algorithm 2 Extended Kalman filter

Require: Initial state estimate $\hat{x}_0^+$
Require: Initial covariance matrix of the estimation error P_0^+
  $k \leftarrow 1$
  while not stopped **do**
    Predict:
    Predict state estimate, $\hat{x}_k^- = f(k, \hat{x}_{k-1}^+, 0)$
    Linearize the process, $A_k = \frac{\partial f}{\partial x}\Big|_{(k,\hat{x}_{k-1}^+,0)}$,
    $L_k = \frac{\partial f}{\partial w}\Big|_{(k,\hat{x}_{k-1}^+,0)}$
    Predict covariance of the estimation error, $P_k^- = A_k P_{k-1}^+ A_K^T + L_k Q_k L_k^T$
    Update:
    Linearize the measurement, $H_k = \frac{\partial h}{\partial x}\Big|_{(k,\hat{x}_k^-,0)}$,
    $M_k = \frac{\partial h}{\partial v}\Big|_{(k,\hat{x}_k^-,0)}$
    Obtain the optimal Kalman gain, $K_k - P_k^- H_{k-1}^T (H_{k-1} P_k^- H_{k-1}^T + M_k R_k M_k^T)^{-1}$
    Update state estimate, $\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - h(k, \hat{x}_k^-, 0))$
    Update covariance of the estimation error, $P_k^+ = (I - K_k H_k) P_k^-$
    $k \leftarrow k + 1$
  end while

Chapter 3

Recurrent Neural Networks

3.1 Introduction to RNNs

An artificial neural network (NN) is a machine learning tool which can approximate a wide class of functions. A NN is a set of units called artificial neurons, each of which take a weighted sum of a given input and apply a nonlinear transformation. By optimizing the weights of the artificial neurons in a process called training, we can enable the artificial neurons to produce outputs which approximate a target function. In an ordinary neural network (called a feedforward neural network), artificial neurons are arranged in structures called hidden layers. Hence, artificial neurons are also known as hidden units. Artificial neurons in the same hidden layer are independent of one another, thereby enabling each artificial neuron to focus on learning a part of the target mapping. The output of the entire hidden layer is called the hidden state of the NN, which is transformed by an output

layer to return the final output. Multiple hidden layers can be stacked, that is, the output of one hidden layer can be passed to a second hidden layer and so on, to enable NNs to represent increasingly complicated mappings.

A Recurrent Neural Network (RNN) is a form of neural network that is designed to exhibit dynamic behavior. A RNN extends the ordinary NN by having its hidden state incorporate a form of "memory" of events that occurred in the past, enabling the processing of variable length sequential inputs, typically found in tasks such as unsegmented connected handwriting recognition and speech recognition in natural language processing and system identification and state estimation in dynamical systems theory. RNNs are especially suitable for tasks involving dynamical systems because the state of a dynamical system is dependent upon its state at a previous time step, producing sequential time-series data which RNNs excel in handling. While feedforward NNs can also work with sequential data, each item in the sequence must be a separate input, making for large input sizes for long sequences. Furthermore, changing the length of the input sequence would require changing the NN architecture itself, making working with datasets containing variable length sequences inpractical. RNNs can handle sequences efficiently because its architecture remains unchanged even as sequence length is varied.

What makes recurrent networks "recurrent" is parameter sharing. If we were to consider the two sentences "I went to Waterloo in 2023" and "In 2023, I went to Waterloo" and ask which year the subject went to Waterloo, we would recognize the year 2023 as being the relevant text string in both sentences, despite it being in different locations in the two sentences. Now, suppose we task a feedforward neural network with extracting the year, feeding the whole sentences as the input. Then, because the NN would have different

parameters for each input feature, it would need to learn all of the rules of the language separately at each position in the sentence, making this task difficult. However, a recurrent neural network shares the weights across many time steps, with each output being a function of the outputs at previous time steps. The internal recurrence relation produced by the RNN allows for the rules learned in the first sentence to be shared with the second even though the relevant text string is in completely different locations, sidestepping this problem. In general, we refer to RNNs as operating on a sequence of vectors $x^{(t)}$, with t being the time step between 1 and τ.

The description of the structure of a RNN, the forward propagation process, as well as the descriptions of improvements to the RNN in this chapter were adapted from [1], where more details can be found.

3.2 Computational graph of RNNs

A computational graph is a way of visualizing the set of computations involved in mapping inputs and parameters of a neural network to outputs and the loss. Figure 3.1 depicts a simplified version of this for a standard feedforward neural network.

A standard feedforward neural network is a directed acyclic graph. Since the computations done in this network to get an output given an input only move in one direction, it is easy to optimize the weights by backpropagating in the reverse direction (from the output layer to the input layer) to get the gradients with respect to the parameters, and then conducting gradient descent.

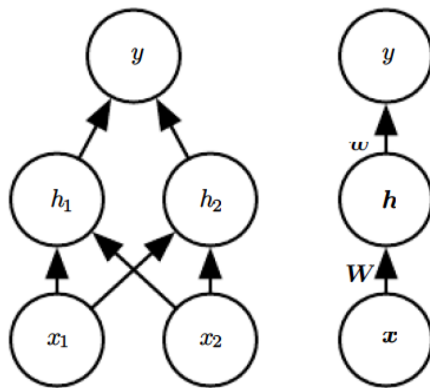

Figure 3.1: Computational graph of a feedforward neural network, drawn in two different styles. On the left, we draw a node for every unit. On the right, the same neural network is shown more compactly by drawing a node for each vector representing a layer's activations, created by transformations applied by the weights W to x and w to h. Nonlinear activation functions map the connection between nodes. The arrows represent the direction of computation. [1], p. 174

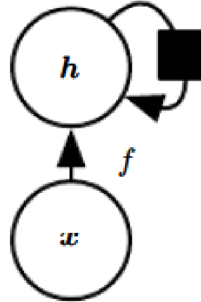

Figure 3.2: The computational graph of a simple recurrent neural network. [1], p. 376

However, many recurrent neural networks define the values of their hidden neurons by

$$h^{(t)} = \sigma(h^{(t-1)}, x^{(t)}; \theta),$$

where $h^{(t)}$ is the state of the hidden neuron at time step t, $x^{(t)}$ is the input at time step t, θ is the set of parameters of the neural network and σ is a nonlinear activation function. Because the state of the hidden neurons depend on its states at previous time steps, the computational graph of a recurrent neural network typically looks like Figure 3.2, with weights U and V connecting the input layer to the hidden layer, then to the output layer respectively, while an additional weight W relates the previous states of the hidden layer to its current state.

The computational graph of a recurrent neural network is therefore a directed *cyclic* graph, making backpropagation not as straightforward as the previously described feed-forward case. Fortunately, for an input sequence of finite length τ, we know that the cycle at the hidden layer will only loop around $\tau - 1$ times. Hence, we can redraw our compu-

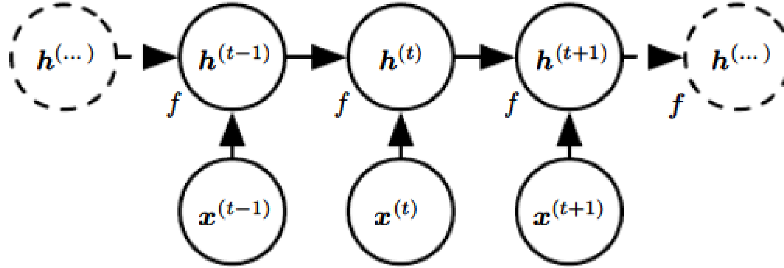

Figure 3.3: The unrolled computational graph of the recurrent neural network shown in Figure 3.2, across three time steps. [1], p. 376

tational graph in the unrolled form as seen in Figure 3.3, by applying the definition of h $\tau - 1$ times. More precisely, from time steps $t = 1$ to $t = \tau$, we apply the following update equations:

$$
\begin{aligned}
a^{(t)} &= Wh^{(t-1)} + Ux^{(t)}, \\
h^{(t)} &= \sigma_h(a^{(t)}), \\
o^{(t)} &= Vh^{(t)} + c, \\
y^{(t)} &= \sigma_o(o^{(t)}),
\end{aligned}
$$

where at time step t, $a^{(t)}$ is the activation for the hidden layer, $h^{(t)}$ is the state of the hidden layer, $o^{(t)}$ is the activation of the output layer and $y^{(t)}$ is the output. The parameters are the weight matrices W, U, V as well as the biases b and c, and finally we have two activation functions σ_h and σ_o for the hidden and output layers respectively.

Unrolling the RNN's computational graph transforms it into a directed acyclic graph, allowing us to apply backpropagation to get the gradients with respect to the parameters of the neural network in a similar manner to the feedforward neural network case. This

variant of backpropagation is called backpropagation through time (BPTT), which will be discussed further in the next chapter.

3.3 The challenge of long-term dependencies

A difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep. Feedforward networks with many layers have such deep computational graphs. This is also the case for recurrent networks working on long sequences, which construct very deep computational graphs by repeatedly applying the same operations at each time step.

As an example, suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix W. After t steps, this is equivalent to multiplying by W^t. Let the decomposition of W in terms of its eigenvalues λ be $W = V\,\mathrm{diag}(\lambda)V^{-1}$. Then,

$$W^t = (V\,\mathrm{diag}(\lambda)v^{-1})^t = V\,\mathrm{diag}(\lambda)^t V^{-1}. \tag{3.1}$$

Any eigenvalues λ_i that are not near a magnitude of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. The *vanishing* and *exploding gradient problems* refers to the fact that gradients through a computational graph are also scaled according to $\mathrm{diag}(\lambda)t$. In the next chapter, we will see how gradient-based optimization methods are used to optimize the parameters of a RNN to complete a given task. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

Recurrent networks use the same matrix W at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem [12]. See [13] for more details on these challenges. For RNNs, this means that when working with temporal sequences with long term dependencies, the gradient of a long term interaction has exponentially smaller magnitude than the gradient of a short term interaction. While it is not impossible to learn, it may take a very long time to learn these long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, [13] showed that as the span of the dependencies that need to be captured increases, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a RNN via stochastic gradient descent rapidly reaching 0 for sequences of only length 10 or 20. We will discuss stochastic gradient descent further in the next chapter.

Various solutions have been proposed to combat the challenge posed by long-term dependencies in RNNs. One idea is to design a neural network architecture that operates at multiple time scales so that so that some parts of the network operate at smaller intervals between time steps and can handle short-term dependencies, while other parts operate at longer intervals and transfer dependencies from the distant past to the present more efficiently.

3.3.1 Skip Connections

One way to operate at longer time scales is to add direct connections from variables in the distant past to variables in the present. The idea of using such connections was inspired by experiments in incorporating delays in feedforward neural networks, and first attributed to [14]. In [14], since gradients may vanish or explode exponentially with respect to the number of time steps τ, recurrent connections with a time-delay of d, called *skip connections*, were introduced to mitigate this problem. With skip connections, gradients now diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ. However, since not all connections are delayed, gradients may still explode exponentially in τ. Hence, skip connections allow the learning algorithm to capture longer dependencies although not all long-term dependencies may be represented well in this way.

3.3.2 Leaky Units

Another way to obtain paths on which the product of the gradients is close to one is to have units with linear self-connections and a weight near one on these connections. Linear self-connections behave similarly to a running average. Suppose we accumulate a running average $\mu^{(t)}$ of some value $v^{(t)}$ by updating

$$\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t)}. \tag{3.2}$$

Then the parameter α is an example of a linear self-connection from $\mu^{(t-1)}$ to $\mu^{(t)}$. When α is near one, the running average remembers information about the past for a longer time,

and when α is near zero, information about the past is quickly discarded. Hidden units with linear self-connections are called leaky units, and were first proposed by [15] and [16].

Skip connections through d time steps are a way of ensuring that the RNN always learns to be influenced by a value from d time steps earlier. The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past, allowing this effect to be adapted more flexibly by adjusting the real-valued α rather than by adjusting the integer-valued skip length.

3.3.3 LSTMs

The most effective recurrent neural networks belong to a class called *gated RNNs*. Like leaky units, gated RNNs seek to create paths in the computational graph through time where the gradients with respect to the parameters do not vanish nor explode. For leaky units, this was done by choosing a parameter α which serves as a connection weight governing the importance placed on long-term dependencies. In gated RNNs, we generalize this idea further by allowing the connection weights to change at each time step. The reason for this is while leaky units enable the RNN to accumulate information over a long timeframe, it has no mechanism to explicitly forget information, instead relying on the values of the connection weights to let it decay passively. As the utility of explicitly forgetting information may be difficult to see, we provide an example from natural language processing. Suppose we would like to predict the next word in the statement,

$$\text{“Ann went to her room and slept. Bob was working in ____.”} \tag{3.3}$$

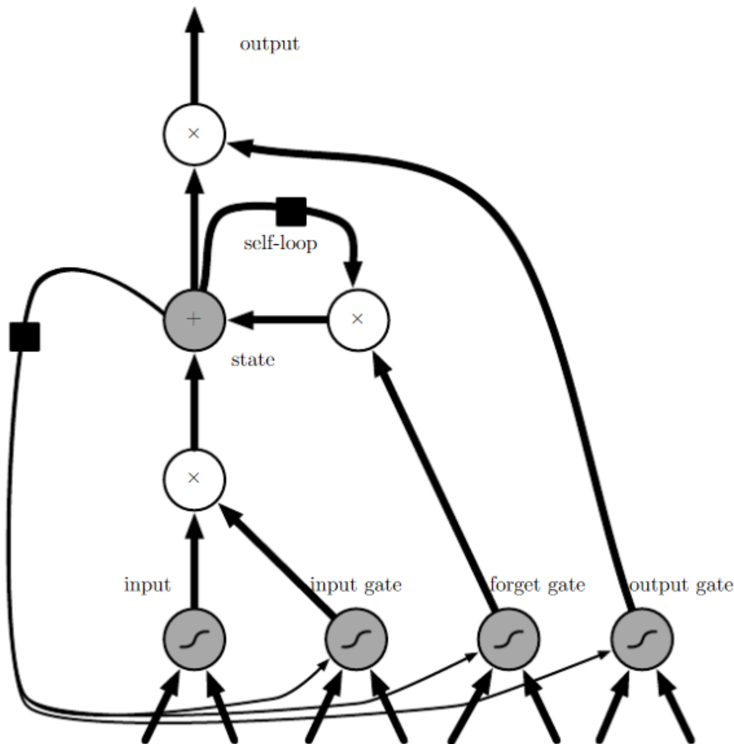

Figure 3.4: A block diagram of a LSTM cell. [1], p. 409

As the subject of the first sentence (Ann) is different from that of the second (Bob), the information from the first sentence is redundant in predicting the next word in this statement. However, a leaky unit would not be able to discern this, only forgetting information as dictated by the static connection weights. The idea is a gated RNN would be able to learn when to forget information, discarding unnecessary information and keeping the paths through which the gradients are calculated succinct.

Long short-term memory (LSTM) [17] is the most widely used gated RNN implementation. A block diagram of the LSTM is shown in Figure 3.4. While in a simple RNN such as that described earlier in this chapter, we simply apply an element-wise nonlinearity to

the affine transformation of inputs and recurrent units, LSTMs have an internal recurrence in a structure known as a *LSTM cell*, in addition to the outer recurrence present in all RNNs. Taking the same inputs and outputs as an ordinary RNN, each LSTM cell contains a system of sigmoidal gates which control the flow of information.

The basic component of a LSTM cell is the state unit $s_i^{(t)}$ for time step t and LSTM cell i. This stores the LSTM cell state, which has a self-connection like the leaky unit introduced in the previous section. A LSTM cell behaves much like a conveyer belt, passing the LSTM cell state though each gate. The final output gate transforms the LSTM cell state into the hidden state we are familiar with from our analysis of a simple RNN. We now proceed to describe the gates in a LSTM cell.

The forget gate

The first step in an LSTM is to decide what information to throw away from the cell state. This is done by the forget gate unit $f_i^{(t)}$, which takes the input vector $x^{(t)}$ at time step t and hidden state $h^{(t-1)}$ at the previous time step $t-1$ to give an output between 0 and 1 for each element of the cell state $s_i^{(t)}$ using the sigmoid function. A value near one means that the variable is important for the given task and should not be forgotten, while a value near zero says that the information should be forgotten. This is represented as

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right), \tag{3.4}$$

where b_i^f, $U_{i,j}^f$ and $W_{i,j}^f$ are the biases, input weights and recurrent weights for the forget gates respectively. This is used as the self-connection weight for the LSTM state unit,

acting similarly to the leaky unit self-connection weight, but having the flexibility of being learnable at each time step.

The external input gate

Next, we must decide what new information to store in the LSTM cell state. We do this in two steps, the first step being deciding what values of the LSTM cell state to update. This is done through the external input gate,

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right), \tag{3.5}$$

where b_i^g, $U_{i,j}^g$ and $W_{i,j}^g$ are the biases, input weights and recurrent weights for the forget gates respectively. The external input gate is structurally similar to the forget gate, producing values between zero and one for each element in the LSTM cell state. A value near one means that the corresponding element of the LSTM cell state should be updated, while a value near zero signifies that it should not.

After deciding which values of the LSTM cell state to update, we update our state unit, giving the update of the LSTM cell state as

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right), \tag{3.6}$$

where b, U and W are the biases, input weights and recurrent weights of the LSTM cell, analogous to those of recurrent units in a simple RNN.

The output gate

Finally, we produce the output $h_i^{(t)}$, which is the hidden state at time step t. This is done through the output gate, which uses a sigmoid function to filter what elements of the LSTM cell state should be reflected in the hidden state. The output gate is represented as

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right), \tag{3.7}$$

where b_i^o, $U_{i,j}^o$ and $W_{i,j}^o$ are the biases, input weights and recurrent weights for the output gates respectively. Once again, a value near one means that the corresponding element of the LSTM cell state should be reflected in the hidden state, while a value near zero signifies that it should not. The final output is produced by passing the filtered LSTM cell state to a hyperbolic tangent activation function,

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}. \tag{3.8}$$

LSTM networks have been shown to learn long-term dependencies more easily than simple RNNs [18]. LSTMs has been successful in many applications, such as speech recognition, machine translation and state estimation of dynamical systems [19].

Chapter 4

Training RNNs

We now discuss the procedure of determining good values for the weights of a RNN in order to get a desired behavior, called training.

We first introduce some basic concepts of training RNNs. Many of these are applicable to training other machine learning algorithms as well. We then present the Backpropagation-through-time (BPTT) algorithm briefly introduced in the previous chapter. Once again, much of the content of this chapter can be found in more detail in [1].

4.1 Gradient Descent

Optimization is at the heart of machine learning, with most algorithms attempting to minimize (or maximize) some objective function. In particular, during training, we seek to minimize a *cost function* which gives the error between the output of the neural network

and the target values (called labels). Note that cost function is often used interchangeably with *loss function* and *error function* in the literature. However, to avoid ambiguity, here we will use "cost" as a measure of error over multiple examples, such as the training set, and "loss" to refer to the measure of error on single examples.

Given a function $f : \mathbb{R}^m \to \mathbb{R}$ that we seek to minimize, the most commonly used technique to find the minimum is called gradient descent, first introduced by Cauchy in 1847. At a point $x \in \mathbb{R}^m$, the directional derivative of f in the direction of the unit vector u represents the instantaneous rate of change of f when moving through x in the direction u. This is given by the derivative of $f(x + \alpha u)$ evaluated at $\alpha = 0$, giving $\frac{\partial}{\partial \alpha} f(x + \alpha u) = u^T \nabla_x f(x)$. To minimize f, we first find the direction in which f decreases most quickly by minimizing the directional derivative. We get

$$\min u^T \nabla_x f(x) = \min ||u||_2 ||\nabla_x f(x)||_2 \cos\theta, \tag{4.1}$$

where θ is the angle between u and $\nabla_x f(x)$. Since u is a unit vector, we substitute $||u||_2 = 1$ to get

$$\min u^T \nabla_x f(x) = \min ||\nabla_x f(x)||_2 \cos\theta. \tag{4.2}$$

It is clear that the directional derivative of f at point x is minimized when $\cos\theta$ is at its minimum value of -1, that is, when the direction vector u points in the opposite direction as the gradient $\nabla_x f(x)$. Hence, the gradient represents the direction of steepest ascent, and we can decrease f by moving in the opposite direction, the direction of steepest descent.

The gradient descent algorithm iterates to a new point

$$x' = x - \epsilon \nabla_x f(x), \tag{4.3}$$

where the *learning rate* ϵ is a scalar parameter specifying the step size of the descent. Steepest descent will repeatedly propose new points until the gradient $\nabla_x f(x)$ is zero. (Recall that when $\nabla_x f(x) = 0$, x is a critical point of f.)

While the above description applies to gradient descent from a single point, the same idea is used when working with multiple samples to train a neural network, in a dataset called the *training set*. Instead of iterating to the opposite direction of a single gradient, we can take the average of the gradients with respect to all points in the training set, the opposite direction of this corresponding to a direction moving closer to a optimal solution in general. This algorithm is known as *batch gradient descent*, or simply gradient descent, and is the most basic gradient-based optimization algorithm used in the training of neural networks.

In more formal terms, suppose our training set consists of N sample inputs $x_i, \ldots, x_N$. Then, our cost function can be written as an average loss over the training set,

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i; \theta)), \tag{4.4}$$

where L is the per-example loss function, $f(x_i; \theta)$ is the output of a neural network with input x_i, θ are the parameters of the neural network, and y_i is the target output corresponding to input x_i. The batch gradient descent algorithm is described in Algorithm

3.

Algorithm 3 Batch Gradient Descent

Require: $N > 0, \epsilon > 0$
  while stopping threshold is not met **do**
    for $i = 1, \ldots, N$ **do**
      $\nabla_\theta J(\theta) \leftarrow \nabla_\theta J(\theta) + \nabla_\theta L(y_i, f(x_i; \theta))$
    end for
    $\theta \leftarrow \theta - \epsilon \frac{1}{N} \nabla_\theta J(\theta)$
  end while

For one iteration of the batch gradient descent algorithm, we must calculate all m gradients for the samples in the training set. While this may be fine when working with moderately sized training sets, when the training set is large, the amount of time required for the algorithm to converge may become impractical. This issue has led to the development of other variants of gradient descent which reduce the number of gradient calculations required before iterating.

4.1.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is the most common optimization algorithm used in machine learning to minimize the cost function associated with the algorithm. While it is well-known mostly among the machine learning community today, its origin can be traced much eariler to the work of Robbins and Monro, in which they presented a rootfinding algorithm for functions that are difficult to compute analytically. SGD is particularly useful for large training sets, as it updates the neural network's parameters based on a random data point, rather than the entire dataset. This makes it computationally efficient

and scalable to large datasets.

Given a cost function

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i; \theta)), \tag{4.5}$$

where N is the number of training examples, y_i is the target output for the ith example, x_i is the input for the ith example, $f(x_i; \theta)$ is the predicted output for the ith example, and θ are the algorithm parameters, we first initialize the algorithm parameters θ to random values. For each iteration, we randomly select a single sample from the training data and compute the gradient of the loss function with respect to the algorithm parameters, which is $\nabla_\theta L(y_i, f(x_i, \theta))$. The new gradient is used to give the updated cost function parameters

$$\theta(t+1) = \theta(t) - \epsilon \nabla_\theta L(y_i, f(x_i; \theta)), \tag{4.6}$$

where ϵ is the learning rate. These steps are repeated until a stopping criterion is met, such as a maximum number of iterations or a desired level of performance.

We assume the training set contains independent, identically distributed samples. Since SGD randomly samples from this dataset in each iteration, we get from linearity of expectation

$$\begin{aligned}
\mathbb{E}(\nabla_\theta L(y_i, f(x_i; \theta))) &= \nabla_\theta \, \mathbb{E}(L(y_i, f(x_i; \theta))) && (4.7) \\
&= \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i; \theta)) && (4.8) \\
&= \nabla_\theta J(\theta). && (4.9)
\end{aligned}$$

Hence, the expected gradient with respect to the training set is the gradient of the overall cost. Therefore, SGD will iterate closer to a local minima on average.

The key difference between stochastic gradient descent and batch gradient descent is that stochastic gradient descent uses only a single sample from the data to compute the gradient at each iteration. This makes it faster and more scalable to large datasets. Another benefit which arises from the stochastic nature of SGD is that the noisy gradient estimates can sometimes dislodge the algorithm from a local minima of the cost function [20], allowing it to converge to a global minima. However, this randomness also makes SGD more prone to convergence issues such as the algorithm oscillating between higher cost and lower cost solutions instead of smoothly iterating to lower cost solutions. The algorithm is shown in Algorithm 4.

Algorithm 4 Stochastic Gradient Descent

Require: $N > 0, \alpha > 0$
  while stopping criterion is not met **do**
    Randomly shuffle the samples in the training set
    for $i = 1, \ldots, N$ **do**
      $\theta \leftarrow \theta - \epsilon \nabla_\theta L(y_i, f(x_i; \theta))$
    end for
  end while

4.1.2 Variants of Stochastic Gradient Descent

There are several variants of stochastic gradient descent that address some of the issues of the basic algorithm, such as oscillating training and slow convergence.

Mini-batch gradient descent

Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent, where the algorithm parameters are updated based on a small fixed-size batch of training examples B called a mini-batch, rather than a single sample in the case of stochastic gradient descent, or the entire dataset in the case of batch gradient descent. This reduces the noise and oscillations of stochastic gradient descent, while being more scalable than batch gradient descent. This algorithm is shown in Algorithm 5, and can be derived from a small modification to the SGD algorithm shown in Algorithm 4.

Algorithm 5 Mini-batch Gradient Descent

Require: $b > 0, \alpha > 0$
  while stopping criterion is not met **do**
    Randomly select a subset B of the training set
    for $i = 1, \ldots, b$ **do**
      $\nabla_\theta J(\theta) \leftarrow \frac{1}{|B|} \sum_{i=1}^{b} \nabla_\theta L(y_i, f(x_i; \theta))$
      $\theta \leftarrow \theta - \epsilon \nabla_\theta J(\theta)$
    end for
  end while

Note that $b = \lceil \frac{N}{|B|} \rceil$ is the number of subsets of the training set. The gradient is now an average of the the gradients of the losses for samples in each mini-batch, reducing the algorithm's susceptibility to oscillations from noisy gradients, and often improving convergence speed.

Momentum

Momentum is another technique used to smooth out weight updates and prevent oscillations during training. Just as a weighted ball rolling downhill would not stop in small holes

along the hill, and would only stop at the bottom of the valley, momentum can prevent the training process from converging to local minima of the cost function. At time step t, the momentum term $v(t)$ is defined as a moving average of the past gradients,

$$v(t) = \beta v(t-1) + (1-\beta)\nabla_\theta J(\theta). \tag{4.10}$$

The parameter β controls the degree to which gradients from the distant past are considered. The updated parameters are

$$\theta(t+1) = \theta(t) - \alpha v(t). \tag{4.11}$$

Another benefit of momentum is that it can help to accelerate the convergence of the optimization process, by allowing the gradient to accumulate in the direction of the dominant features of the loss surface [21]. This can be particularly useful in deep learning, where the loss surface can be very complex and high-dimensional. Mini-batch gradient descent with added momentum is shown in Algorithm 6.

Algorithm 6 Mini-batch Gradient Descent with momentum

Require: $N > 0, \alpha > 0$
  while stopping criterion is not met **do**
    Randomly select a subset $B(t)$ of the training set
    for $i = 1, \ldots, N$ **do**
      $\nabla_\theta J(\theta) \leftarrow \frac{1}{|B(t)|} \sum_{i=1}^{N} \nabla_\theta L(y_i, f(x_i; \theta))$
      $v(t) \leftarrow \beta v(t-1) + (1-\beta)\nabla_\theta J(\theta)$
      $\theta \leftarrow \theta - \alpha v(t)$
    end for
  end while

4.1.3 Adaptive learning rate methods

These methods adapt the learning rate based on the gradient history, in order to improve the convergence rate and avoid oscillations. Some popular examples are AdaGrad, RMSProp, and Adam, which we introduce below.

AdaGrad

The AdaGrad algorithm [22], presented in Algorithm 7, customizes the learning rates of each algorithm parameter by adjusting them in proportion to the inverse square root of their historical squared values. This approach causes parameters with large gradients with respect to the cost function to experience a more significant reduction in their learning rate, while those with smaller gradients experience a more modest reduction. Hence, AdaGrad is claimed to improve upon SGD in cases where the gradient vectors are sparse. However, a disadvantage of AdaGrad is the accumulation of squared gradients from the start of training may result in progressively greater reductions in learning rate, especially when the gradients are large, stalling training.

RMSProp

The RMSProp algorithm [23], is a modification to AdaGrad that aims to enhance its performance in non-convex settings. While AdaGrad converges quickly when applied to a convex function, it may struggle with non-convex functions, particularly if the learning rate becomes very small before reaching a locally convex region. To address this issue, RMSProp introduces momentum to the squared gradients, using an exponential moving

Algorithm 7 AdaGrad

Require: Learning rate $\epsilon > 0$
Require: Decay rate $\rho > 0$
Require: A small constant $\delta > 0$ to ensure numerical stability of division.
  $r = 0$
  while stopping criterion is not met **do**
    Sample a subset $B(t)$ of the training set
    Compute the gradient $g \leftarrow \frac{1}{|B(t)|} \sum_{i=1}^{N} \nabla_\theta L(y_i, f(x_i; \theta))$
    Accumulate the squared gradient $r \leftarrow r + g \odot g$
    Compute parameter update $\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{r}+\delta} \odot g$
    Update parameters $\theta \leftarrow \theta + \Delta\theta$
  end while

average to discard distant historical information and converge quickly once a locally convex region is found, behaving much like an instance of AdaGrad that was initialized within that region. The updated algorithm is presented in Algorithm 8, where a new hyperparameter ρ controls the length scale of the moving average.

Algorithm 8 RMSProp

Require: Learning rate $\epsilon > 0$
Require: A small constant $\delta > 0$ to ensure numerical stability of division.
  $r = 0$
  while stopping criterion is not met **do**
    Sample a subset $B(t)$ of the training set
    Compute the gradient $g \leftarrow \frac{1}{|B(t)|} \sum_{i=1}^{N} \nabla_\theta L(y_i, f(x_i; \theta))$
    Accumulate the squared gradient $r \leftarrow \rho r + (1 - \rho) g \odot g$
    Compute parameter update $\Delta\theta \leftarrow -\frac{\epsilon}{\sqrt{r+\delta}} \odot g$
    Update parameters $\theta \leftarrow \theta + \Delta\theta$
  end while

Adam

Adam [24] is another widely used adaptive learning rate optimization algorithm. It is named for "adaptive moments," and is inspired by RMSProp. Like in RMSProp, momentum is introduced by keeping exponential moving averages. In Adam, these are kept for the sum of gradients (called the first moment) and the sum of squared gradients (called the second moment). An issue that both RMSProp and Adam faces is that the exponential moving averages may be biased upon initialization because they are always initialized to zero regardless of the true moments. This leads to suboptimal parameter updates which may slow down training before the exponential moving average can correct itself. Adam improves upon RMSProp by correcting for the bias.

Adam is considered to be fairly robust to the choice of hyperparameters [1], making it a popular optimizer. The algorithm is presented in Algorithm 9. At the time of writing, Adam is a recommended optimizer in several machine learning software libraries, including PyTorch and Keras.

Algorithm 9 Adam

Require: Learning rate $\epsilon > 0$
Require: Exponential decay rates $\rho_1, \rho_2 \in [0, 1)$ for each momentum.
Require: A small constant $\delta > 0$ to ensure numerical stability of division.
  $r = 0, s = 0$
  $t = 0$
  while stopping criterion is not met **do**
    Sample a subset $B(t)$ of the training set
    Compute the gradient $g \leftarrow \frac{1}{|B(t)|} \sum_{i=1}^{N} \nabla_\theta L(y_i, f(x_i; \theta))$
    $t \leftarrow t + 1$
    Update biased first moment estimate $r \leftarrow \rho_1 r + (1 - \rho_1) g$
    Update biased second moment estimate $s \leftarrow \rho_2 s + (1 - \rho_2) g \odot g$
    Correct the bias in the first moment estimate $\hat{r} \leftarrow \frac{r}{1-\rho_1^t}$
    Correct the bias in the second moment estimate $\hat{s} \leftarrow \frac{s}{1-\rho_2^t}$
    Compute parameter update $\Delta\theta \leftarrow -\frac{\epsilon \hat{r}}{\sqrt{\hat{s}}+\delta} \odot g$
    Update parameters $\theta \leftarrow \theta + \Delta\theta$
  end while

4.2 Backpropagation through time (BPTT)

In the previous section, we discussed how we can optimize the parameters of a machine learning alogirthm by finding the minima of a cost function which gives a measure of the error between the target values and algorithm outputs on the training set, using gradient-based optimization algorithms such as stochastic gradient descent. In this section, we discuss how to calculate the gradients required in these algorithms, specifically focusing on the case of RNNs. The algorithm which does this is called backpropagation through time (BPTT).

Computing the gradient of a recurrent neural network means recursively computing the gradient of a given loss function $L(Y, f(X; \theta))$ with respect to the parameters of the

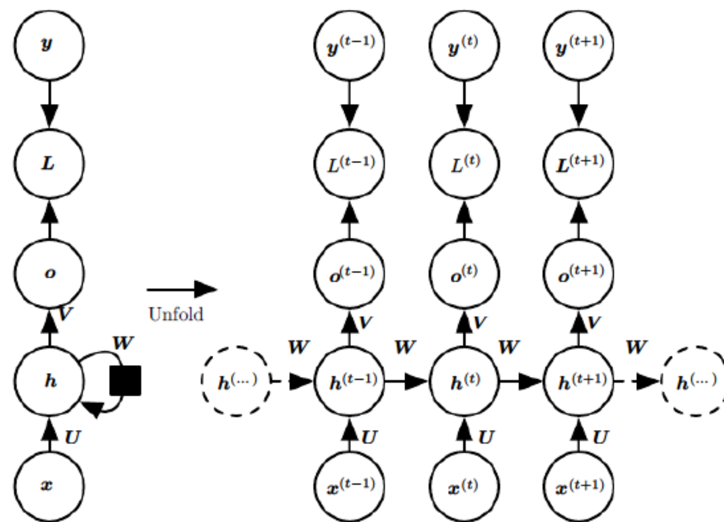

Figure 4.1: The unrolled computational graph of a RNN. Each step of the input sequence x is transformed into the RNN hidden state h with weight matrix U. A second weight matrix V is used to transform the hidden state into the output o. Taking this as well as the labels y as parameters, the loss L is calculated. The recurrent weights W are applied to the hidden state h to create the recurrence relation across the number of steps contained in the input sequence. The unrolled (unfolded) representation across three time steps is shown on the right. The unrolled computational graph is a directed acyclic graph (DAG), allowing for recursive computation of the gradients of the loss function with respect to the weights of the RNN with BPTT. [1], p.378

network. In recurrent neural networks, if we define a loss $L^{(t)}$ at each time step t, then the total loss across the network is simply the sum of all losses across all time steps $\sum_{\forall t} L^{(t)}$.

We now illustrate an example of calculating the gradients of the recurrent neural network described in the previous chapter. In the interest of simplicity, we construct a concrete example with commonly used activation and loss functions, letting the output activation σ_o be the softmax function, defined as

$$\sigma_o(x_i) = \frac{\exp x_i}{\sum_{i=1}^{n} x_i} \tag{4.12}$$

for each element x_i of an input vector $x \in \mathbb{R}^n$. For the activations at the hidden layers σ_h, the hyperbolic tangent function is used. We use a negative log-likelihood of the true target as our loss function. We start the recursion at the nodes immediately before the final loss in the unrolled computational graph in Figure 4.1. Then,

$$\frac{\partial L}{\partial L^{(t)}} = 1. \tag{4.13}$$

Next, we calculate the gradient on the outputs $o_i^{(t)}$ at time step t. For each element i of $o^{(t)}$:

$$(\nabla_{o^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = y^{(t)} - 1. \tag{4.14}$$

We now work our way backwards, starting from the end of the sequence at time step τ. Let V be the output layer weight. Since $h^{(\tau)}$ only has $o^{(\tau)}$ as a descendent, the gradient

with respect to the recurrence weight is

$$\nabla_{h^{(\tau)}} L = V^T \nabla_{o^{(\tau)}} L. \tag{4.15}$$

We continue to iterate backwards in time from time steps $\tau - 1$ to 1. $\text{diag}(1 - (h^{(t+1)})^2)$ is the Jacobian of the hyperbolic tangent activation σ_h function associated with the hidden layer at time $t+1$. Since for $t < \tau$, $h^{(t)}$ has descendants in both $o^{(t)}$ and $h^{(t+1)}$, its gradient is

$$\nabla_{h^{(\tau)}} L = W^T(\nabla_{h^{(t+1)}} L) + \text{diag}(1 - (h^{(t+1)})^2) + V^T(\nabla_{o^{(t)}} L). \tag{4.16}$$

Once we compute the gradients with respect to the internal nodes $h^{(t)}, o^{(t)}$, we can proceed to calculate the gradients with respect to the hidden layer weight U, the output layer weight V, the recurrent weight W, hidden layer bias b and output layer bias c in much the same way. In fact, the gradient of the parameters are

$$\begin{aligned}
\nabla_U L &= \sum_t \text{diag}(1 - (h^{(t)})^2)(\nabla_{h^{(t)}} L) {x^{(t)}}^T, \\
\nabla_V L &= \sum_t (\nabla_{o^{(t)}} L) {h^{(t)}}^T, \\
\nabla_W L &= \sum_t \text{diag}(1 - (h^{(t)})^2)(\nabla_{h^{(t)}} L) {h^{(t-1)}}^T, \\
\nabla_b L &= \sum_t \text{diag}(1 - (h^{(t)})^2) \nabla_{h^{(t)}} L, \\
\nabla_c L &= \sum_t \nabla_{o^{(t)}} L.
\end{aligned} \tag{4.17}$$

The algorithm is typically implemented as shown in Algorithm 10.

Algorithm 10 Backpropagation through time

Require: $\mathbb{T} \neq \emptyset$, the set of variables whose gradients we seek to compute.
Require: $G(V, E)$, the computational graph of the RNN. V and E are the set of vertices and edges of the graph, respectively.
Require: z, the variable to be differentiated.
  $G \leftarrow G'$, the subgraph of G containing only nodes that are parents of z and children of nodes in $\mathbb{T}$.
  $gradients[z] \leftarrow 1$, where *gradients* is a data structure associating RNN parameters to their gradients.
  for $v \in \mathbb{T}$ **do** build_grad$(V, G, G', gradients)$
  end for

Much of the computation is done in the recursive `build_grad` method, as shown in Algorithm 11. The BPTT algorithm requires $O(n)$ operations, where n is the number

Algorithm 11 build_grad

Require: v, the variable whose gradient should be added to *gradients*.
Require: G, the computational graph of the RNN.
Require: G', the pruned computational graph of the RNN.
Require: *gradients*, the data structure mapping nodes to their gradients.
  if $v \in gradients$ **then return** $gradients[v]$
  end if
  $i \leftarrow 1$
  for all children c of G' **do**
    $D \leftarrow$ build_grad$(c, G, G', gradients)$
    $g^{(i)} \leftarrow$ gradients calculated using the chain rule, from parents of c in G'
    $i \leftarrow i + 1$
  end for
  $g \leftarrow \sum_i g^{(i)}$
  $gradients[v] = g$

of nodes in the RNN's computational graph. However, each operation may itself have higher order time complexity with respect to its input– for example, the time complexity of schoolbook matrix multiplication is cubic with respect to the largest matrix dimension.

Armed with a way to measure the error of a recurrent neural network, a gradient descent-based method of reducing it, and finally a way to calculate the gradients with respect to the weights of the network, we are now ready to present a basic procedure for training RNNs, shown in Figure 12.

Algorithm 12 Basic training loop

Require: $X \neq \emptyset$, the input features of the training set.
Require: $Y \neq \emptyset$, the target labels of the training set.
  while maximum number of epochs is not met **do**
    $Y' \leftarrow f(X)$
    Get the cost of Y' with respect to the target labels Y.
    Backpropagate on the cost function with respect to the neural network parameters.
    Using a gradient-based optimizer, update the parameters of the neural network.
  end while

4.3 Quantifying the performance of neural networks

The main challenge in training a neural network is that it must perform well on unseen inputs. The ability of a trained neural network to perform on previously unobserved inputs is called *generalization*. Typically, when we train a neural network, we are able to calculate a measure of error on the training set (called the training error). However, what we would actually like to have is the expected error on all unobserved inputs, called the *generalization error*, which we cannot measure. We can estimate this by separately collecting a representative sample (called a test set) in addition to the training set, and evaluating the error of the neural network on this, called the test error. Since the two datasets were separately produced from the same generating process, we assume the samples in each dataset are

independent from each other, and the two datasets are identically distributed. Then, the expected training error is equal to the expected test error because the two datasets share the same data generating distribution. Hence, by optimizing the parameters of the neural network to reduce the training error, we can expect to reduce the test error as well, which is equivalent to reducing the estimated generalization error.

When a neural network fails to generalize, the neural network can be broadly considered considered to be either *underfitting* or *overfitting*. Underfitting occurs when the neural network is unable to obtain a sufficiently low training error and hence fails to generalize to the test data, while overfitting occurs when the neural network fails to generalize to the test data because it has learned too much specific information of the training data.

4.4 Capacity of neural networks

One of the main methods of controlling how likely a neural network is to overfit or underfit is by modifying its capacity. A neural network's capacity is its ability to approximate a variety of functions. When the capacity is too low, it would underfit because it would be unable to fit a function to the training set. On the hand, if the capacity is too high, the neural network can overfit by memorizing properties of the training set that are not reflected in the true data generating distribution. In order to modify the capacity of the neural network, we can first vary the number of inputs it accepts. The capacity generally increases when the number of allowable inputs increases. Next, we can also alter the number of parameters that are tied to each input, such as the size of a hidden layer as well as the number of hidden layers in the case of a neural network. Naturally, a neural

network with a large number of parameters would generally have the capacity to fit more complicated functions.

It is difficult to give an exact quantification of a neural network's capacity. A neural network's capacity is not only affected by its number of inputs and parameters, but also by the optimization algorithm chosen for the training process. Once the optimizer's initial conditions are known it becomes a deterministic algorithm, so the optimizer would only be able to procedurally generate a subset of the parameters configurations in the entire parameter space. Hence, the optimal parameter configuration may never be chosen, so the effective capacity of a neural network may differ from its representational capacity. However, typically the relationship between the capacity and the training and test errors can be categorized into the *underfitting regime* and *overfitting regime*, as shown in Figure 4.2. When the capacity is low, training and test errors are both high and the neural network is in the underfitting regime. As capacity is increased the training error decreases, but the difference between the training and test errors increases. The capacity eventually becomes too high and this difference outweighs the decrease in the training error. As a result, the neural network enters the overfitting regime. The sweet spot between the two regimes is where the neural network is at optimal representational capacity.

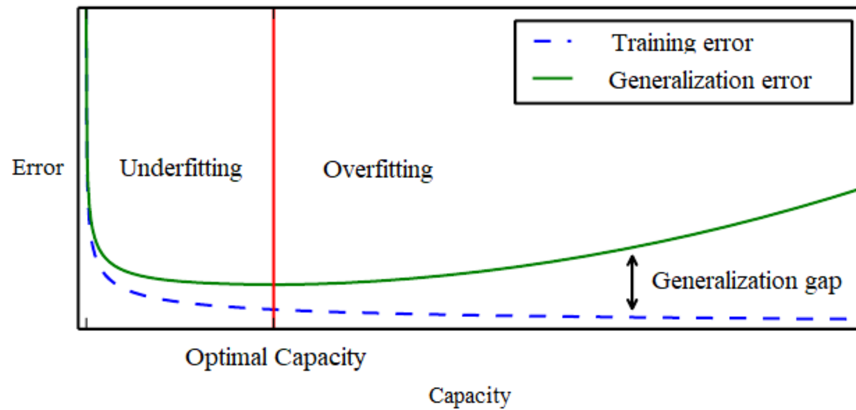

Figure 4.2: A typical relationship between errors and capacity of neural networks. [1], p. 409

4.5 Regularization

As we mentioned earlier, our choice of optimizer in the training process has an effect on the effective capacity of a neural network. Since neural networks use probabilistic rules to infer rules that are probably correct for most members of a data distribution, we can optimize the effective capacity of a neural network by preferring solutions which closely match the data distribution, only choosing another solution if it significantly fits the training data better than the equivalent preferred solution. The methods of both implicitly and explicitly expressing preferences for certain solutions are known as *regularization*. A common way of explicitly expressing preferences for certain solutions is by adding a penalizing regularizer term to the cost function $J(\theta)$ we minimize in training. For example, L^2 regularization, commonly known as weight decay, imposes the penalty $\lambda w^T w$, where w are the weights of a neural network and the decay rate λ is chosen beforehand. The new cost function

becomes

$$J(\theta)_{reg} = J(\theta) + \lambda w^T w. \tag{4.18}$$

The larger λ is, the more the regularizer term penalizes large weights by amplifying the cost associated with them. Hence, increasing λ makes our training prefer progressively smaller weights.

4.6 The curse of dimensionality and data selection

The quality of data is just as important as the configuration of a neural network. As its strong name suggests, the curse of dimensionality is an observation on the increasing amount of data required to train a neural network to generalize sufficiently well as the number of dimensions of the input becomes larger. The gist of the curse can be seen by looking at how representative a number of given data points is of the data space as we increase its number of dimensions.

Figure 4.3 plots 20 data points in a (a) 1D (b) 2D and (c) 3D setting, divided into

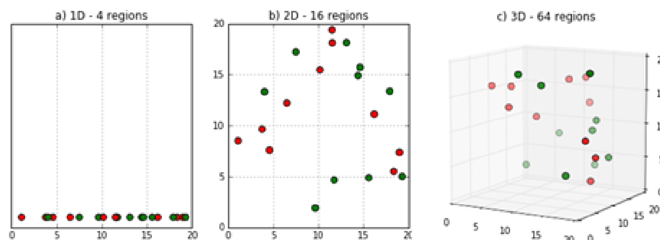

Figure 4.3: Plots of 20 data points, as we increase the dimension of the data space. The volume of the space represented grows quickly and the data becomes sparse.

equally sized regions. In the one-dimensional setting, the data is contained in all four line segments of length 5 constituting the data space, and hence is representative of the data space. However, as we increase its number of dimensions, the data becomes sparse and less representative of the data space, with the data only being captured in 11/16 squares in the two dimensional setting, and 13/64 cubes in the three dimensional setting.

Trunk [25] provides an example which demonstrates how the number of features and the size of the training set can affect the performance of a neural network. Consider a neural network tasked with classifying l-dimensional data points (with l features) into one of two classes ω_1 and ω_2, each with equal a priori probabilities $P(\omega_1) = P(\omega_2) = 1/2$. We assume that each class is normally distributed with mean μ for class ω_1 and $-\mu$ for class ω_2, with

$$\mu = \left[1, \quad \tfrac{1}{\sqrt{2}}, \quad \tfrac{1}{\sqrt{3}}, \quad \ldots, \quad \tfrac{1}{\sqrt{l}}\right]^T. \tag{4.19}$$

Thus, each additional feature of points in ω_1 and ω_2 has mean nearer to zero, reducing its discriminatory power. We also assume that each feature is independent, with each class having the same covariance matrix $\Sigma = I$. Then, the optimal classifier rule is equivalent to the minimum Euclidean distance classifier, which classifies a feature vector x to ω_1 if

$$||x - \mu||^2 < ||x + \mu||^2, \tag{4.20}$$

or equivalently $x^T \mu > 0$, and classifies to ω_2 otherwise.

Trunk found that for any l, if both the mean and variance of each class are known, then we can discriminate between the two classes with perfect accuracy by arbitrarily

increasing the number of features. However, if the variance is known but the mean of each class is estimated based on a finite training set, then increasing the number of features leads to lower accuracy, which eventually degrades to the accuracy of random guessing as the number of features tends to infinity. Trunk's results suggested that with finite training data, the number of features must be kept low for good classification performance and the optimal number of features increases with the number of samples in the training set.

While Trunk's example illustrates an extreme case of the curse of dimensionality in machine learning, it is generally the case that as the number of input dimensions is increased, the same number of samples will become much less representative of the overall data space, increasing the likelihood of overfitting. While in theory we can always rectify this issue simply by producing larger and larger datasets, this is often infeasible due to resource constraints, such as the limited availability of time and computing power to generate new samples. For this reason, we must take careful consideration of what information we feed to our neural network, taking care that the data collected is a representative sample of the actual data generating distribution. For example, if we seek to estimate the state of a dynamical system, we must ensure that the sampled measurements are rich enough to fully capture the dynamics of the system. This condition is known as *persistent excitation*.

4.6.1 Persistent excitation

System identification is an experimental alternative to mathematical modeling. Instead of analytically describing the behavior of a process or phenomenon starting from basic physical laws, in system identification experiments are performed on the system; the recorded data is

then fitted to an appropriate model structure by finding the optimal parameters. Persistent excitation [26] is a fundamental concept in system identification that refers to the property of a system input that ensures the identification of the system parameters. In particular, a system input is said to be persistently exciting if it contains enough energy across a range of frequencies, such that it can uniquely identify the system parameters. If the system input is not persistently exciting, it may not contain enough information to accurately identify the system parameters, leading to poor estimator performance. In particular, if the input energy is concentrated in a narrow frequency band, it may not be possible to accurately estimate the parameters that govern the dynamics of the system at other frequencies.

To ensure persistent excitation, it is important to design the system input in such a way that it contains enough energy across a range of frequencies. This can be achieved by using signals with a wide frequency content, such as white noise or pseudo-random binary sequences, or by designing specific input signals that are tailored to the system under consideration. It is shown in [26] that for controllable (i.e. for any initial state x_0 and final state x_f, we can determine a finite sequence of inputs to change the state of the system from x_0 to x_f) linear discrete-time systems, such input will always be persistently exciting.

Figure 4.4 illustrates the importance of selecting quality data, in the context of training a RNN to estimate the states of a 50 DoF connected mass-spring-damper system. Connected mass-spring-damper systems and the experiment setup are described in detail in the subsequent chapter. The two plots show the training and validation (discussed in the next section) losses when training over 500 iterations on the training data. The plot on the left was produced when the training data was generated with persistently exciting white noise

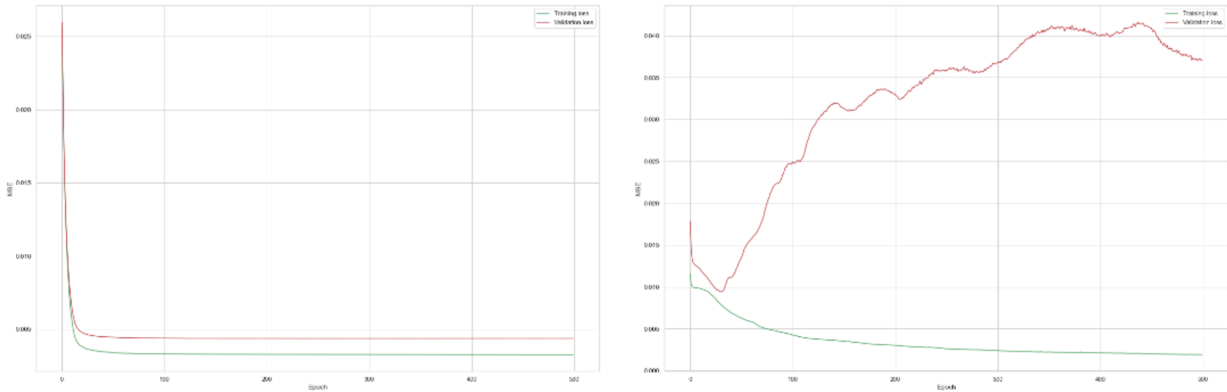

Figure 4.4: Evolution of training (green) and validation (red) errors when training with persistently exciting data (left), compared to training with data that is not persistently exciting (right).

input to the system. The plot on the right was produced when the input was Gaussian noise with mean value 1, which is not persistently exciting. The training converges rapidly to low error values when the data is persistently exciting, while training fails to converge when they are not. All other variables were selected to be the same.

4.7 Hyperparameter optimization

The training process of a neural network enables it to learn various parameters in order to best approximate a target function. We saw earlier that gradient descent-based optimizers are an effective manner of doing this. However, there are also settings that are not learned by the algorithm during training but are rather set manually by the user beforehand. These settings have to do with the behavior of the training process itself, and

are called hyperparameters. Examples of hyperparameters include the learning rate and regularization strength of gradient-based optimizers, the number of hidden layers and the size of the hidden layers in neural networks, and the number of epochs (iterations on the training set) for which training should be conducted.

There are several reasons why hyperparameters are distinct from regular parameters. In many cases, a setting is a hyperparameter because it would be inappropriate to learn it during the training process. For example, if we were to learn the size of a hidden layer during training, the optimizer would always choose the largest possible value because this results in the highest neural network capacity, leading to overfitting. Hence we produce a separate dataset called the *validation set* to evaluate the effects of the hyperparameters. As the test set must be unobserved by the neural network until its final evaluation, the validation set must be independent of it. Thus, it is typical to create the validation set as a subset of the training data, with 20% of the training data held out as the validation set, and the remaining 80% being used for training. This is fitting, as in a sense we use the validation set to "train" the hyperparameters of the neural network. Just as the neural network's test error is an estimate of the generalization error, so is the error over the validation set (called the validation error) because it is a dataset that is not used for learning the algorithm parameters. For this reason, the validation error is often used as a proxy for the generalization error during the training process. A common method of diagnosing overfitting due to overtraining is to plot the training error and validation error after each epoch on the same graph. The epoch where the training error begins diverging from the validation error is a good estimate for the optimal number of epochs. However, it must be noted that if the validation set is used for tuning the hyperparameters of the

training, the validation error tends to underestimate the generalization error.

The process of tuning the hyperparameters in order to find the combination of hyperparameter values that result in the best training performance is called hyperparameter optimization. Specifically, given an objective function (cost function) $f(x)$ with $x \in X$, the set of hyperparameter choices, we would like to find

$$x^* = \arg\min_{x \in X} f(x), \tag{4.21}$$

the choice of hyperparameters that yields the lowest value of the objective function. Hyperparameter optimization is an important step in building effective neural networks. It involves searching through a space of possible hyperparameter values to find the combination that yields the best performance on a validation dataset. There are several different techniques used for hyperparameter optimization. The most basic of these are grid search and random search. However, there also exists more advanced techniques such as Bayesian optimization. These methods differ in terms of their computational cost and search strategy, but they all aim to find the best hyperparameters to optimize the training performance.

4.7.1 Grid search and random search

Grid search and random search are two simple methods for hyperparameter optimization. In grid search, we create a grid of all possible combinations of hyperparameter values and evaluate each combination using cross-validation. This involves dividing the dataset into a training set and a validation set, then training the neural network on the training set

with each combination of hyperparameters and finally evaluating the performance of the neural network on the validation set. The combination of hyperparameters that results in the best performance is then selected.

While grid search is simple and easy to implement, it can be computationally expensive when there are many hyperparameters to tune, as it requires training a neural network for every combination of hyperparameters in the grid. Given k hyperparameters to be tuned, with each hyperparameter having n_k possible choices, let $a = \max(n_1, \ldots, n_k)$. Then, the number of hyperparameter combinations we must check is bounded above by a^k, giving grid search an exponential time complexity of $O(a^k)$. The effectiveness of grid search can also be limited by the granularity of the grid, which when set too coarse can miss important regions of the hyperparameter space. On the other hand, random search involves randomly sampling hyperparameter values from a specified distribution and evaluating the neural network trained with each set of sampled hyperparameters. Unlike grid search, random search does not rely on a pre-defined set of hyperparameter values, which can help to avoid missing important regions of the hyperparameter space.

While random search can be less computationally expensive than grid search, it can be less effective at finding the optimal hyperparameters, especially when the hyperparameter space is large and complex. However, it can be a good starting point for hyperparameter optimization and can be useful for exploring the hyperparameter space.

4.7.2 Bayesian optimization

An inherent issue with random search is that is it inefficient. As hyperparameters are selected with no assumptions made on the distribution of well-performing hyperparameters, many hyperparameters returned by random search will not improve training performance. Bayesian optimization is a more advanced method for hyperparameter optimization that uses a probabilistic model to select the next set of hyperparameters to evaluate, based on the results of previous evaluations. It is particularly effective for high-dimensional and noisy hyperparameter spaces, where grid search and random search can prove to be computationally expensive.

In Bayesian optimization, we search for the optimal hyperparameters x^* by first constructing a model of $p(y|x)$, the probability of error y given hyperparameters x, called the *surrogate model*. Next, we select promising hyperparameters by using a criterion of how desirable a given hyperparameter combination is in improving training performance, called the *acquisition function*. A popular acquisition function is Expected Improvement.

Expected Improvement

Expected Improvement (EI) is a popular acquisition function first proposed in [27], and further developed in [28]. Let $y = f(x)$. Then, the Expected Improvement is defined as

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy, \tag{4.22}$$

where the threshold value y^* is a control variable set by the user, x is a choice of hyperparameters to evaluate and y is the actual value of the objective function on hyperparameters x. The threshold value y^* is typically chosen to be the current best value of the objective function on a previous evaluation. We see that if $p(y|x)$ is zero everywhere when $y < y^*$, then $EI_{y^*}(x) = 0$. Otherwise, this integral will be positive, meaning that the choice of hyperparameters x is expected to yield a better result than the current best evaluation on the objective function.

In other words, the Expected Improvement is the expected increase in performance over the current best performance if the surrogate model is evaluated on a choice of hyperparameters x. If the expected performance of x is worse than the current best performance, the Expected Improvement is 0.

Furthermore, it was shown in [28] that the Expected Improvement at x can be expressed as

$$\begin{aligned} EI_{y^*}(x) &= \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy \\ &= \sigma(x)(s\Phi(s) + \phi(s)), \end{aligned} \tag{4.23}$$

where $s = (\mu(x) - y^*)/\sigma(x)$ is the Expected Improvement at x normalized by its standard deviation, and ϕ and Φ are the standard normal probability density and cumulative density functions respectively. Hence, EI can be thought of as the improvement averaged with respect to the posterior probability of obtaining it. Therefore, EI balances the exploitation of solutions which are very likely to be a little better than y^* with the exploration of other solutions which may turn out to be significantly better, albeit with lower probability.

In order to evaluate the acquisition function, we must model $p(y|x)$. One way of doing this is by using a framework called the Tree-structured Parzen estimator.

Tree-structured Parzen Estimator

The Tree-structured Parzen Estimator (TPE) [29] is a framework which implements the surrogate model in a way which streamlines the process of choosing hyperparameters with Expected Improvement.

Instead of directly modeling $p(y|x)$, TPE models $p(x|y)$, the probability of observing hyperparameters x given training error y, by creating a "good" distribution $l(x)$ and "bad" distribution $g(x)$:

$$p(x|y) = \begin{cases} l(x), & \text{if } y < y^* \\ g(x), & \text{if } y \geq y^*, \end{cases} \tag{4.24}$$

where y^* is a threshold value of the objective function chosen so that $p(y < y^*) = \gamma$ for some quantile γ of the observed training errors. For example, $\gamma = 0.5$ means hyperparameters which produce the top 50% of the smallest errors will be used to create $l(x)$, while the bottom 50% will be used to create $g(x)$. Hence, when the training error is lower than y^*, $p(x|y)$ is the probability distribution $l(x)$, otherwise it is $g(x)$.

Intuitively, it should be the case that promising hyperparameters would display high probability under the "good" distribution $l(x)$ while simultaneously have low probability under the "bad" distribution $g(x)$. Then, the ratio $l(x)/g(x)$ is a measure of the "goodness" of hyperparameters x.

In fact, since (4.24) implies

$$p(x) = \int_{-\infty}^{\infty} p(x|y)p(y)dy = \gamma l(x) + (1-\gamma)g(x),$$

substituting (4.24) into (4.22) through Bayes theorem, the Expected Improvement becomes

$$\begin{aligned} EI_{y^*}(x) &= \int_{-\infty}^{y^*} (y^* - y)p(y|x)dy \\ &= \int_{-\infty}^{y^*} (y^* - y)\frac{p(x|y)p(y)}{p(x)}dy \\ &= \int_{-\infty}^{y^*} (y^* - y)\frac{p(x|y)p(y)}{\gamma l(x) + (1-\gamma)g(x)}dy \\ &= \frac{l(x)}{\gamma l(x) + (1-\gamma)g(x)} \int_{-\infty}^{y^*} (y^* - y)p(y)dy \\ &= \frac{\gamma y^* l(x) - l(x)}{\gamma l(x) + (1-\gamma)g(x)} \int_{-\infty}^{y^*} p(y)dy. \end{aligned} \tag{4.25}$$

Hence, $EI_{y^*}(x)$ is proportional to $(\gamma + \frac{g(x)}{l(x)}(1-\gamma))^{-1}$, that is, the EI is proportional to the ratio $l(x)/g(x)$. Thus, by sampling hyperparameters from the "good" distribution $l(x)$ and ranking candidates according to $l(x)/g(x)$, TPE efficiently returns hyperparameter candidates which are guaranteed to increase EI without having to evaluate the integral itself. The time complexity of TPE is linear with respect to the number of hyperparameters being optimized and the number of training errors observed per evaluation.

Once a set of candidate hyperparameters is proposed, this is evaluated on the objective function. The result is used to update the surrogate model, completing one iteration of Bayesian optimization with TPE. For the full description of the framework see [29]. The Bayesian optimization process is repeated until a stopping criterion is met, such as a

maximum number of evaluations or a desired level of performance.

Bayesian optimization can be computationally expensive, as it requires evaluating the performance of the surrogate model on each set of hyperparameters. However, it is often still much cheaper to compute than the objective function itself, especially for high dimensional neural networks (see [30] for an example of this in a computer vision application), making this method of hyperparameter optimization indispensable in the training of certain complex neural networks. It must be said that while Bayesian optimization has yielded fantastic results in certain applications, it is not a catch-all method for optimizing hyperparameters [1].

Chapter 5

State estimation with RNNs and the Kalman filter

In this section we compare the performance of the Kalman filter to that of recurrent neural networks for state estimation of noisy dynamical systems. We first present some previous work done on this subject.

5.1 Previous work

An early comparison of the performance of the Kalman filter to RNNs was conducted in by DeCruyenaere et al. [31]. They compared the performance of the two methods on 24 different examples of linear dynamical systems, not all of which satisfy the Kalman filter requirements of Gaussian noise. They found that if the Kalman filter requirements

are met, then a Kalman filter and recurrent neural network performs similarly in terms of accurately predicting the true state of the noisy systems. However, when the Kalman filter assumptions are not met, we can find a recurrent neural network which produces significantly more accurate estimates than the Kalman filter. More recently, Chenna et al. [32] conducts a similar comparison for a state estimation problem and a tracking problem, coming to similar conclusions.

For nonlinear systems, Feldkamp et al. [19] compared the performance of the Extended Kalman filter to a recurrent neural network. Their conclusions were in line with those from the studies done on linear systems. For noisy nonlinear systems such that the EKF provided high accuracy, a recurrent neural network was able to perform to a similar degree of accuracy. However, for highly nonlinear systems where the EKF performed poorly, a recurrent neural network was able to provide much more accurate estimates of the true state.

5.2 Methodology

We implement RNN-based state estimators for a number of example dynamical systems and compare them to a Kalman filter. For each system considered, we simulate the measurements y_k and the true states x_k across time steps k, which is used as data for training, validation and testing of the RNNs. Each sample in our data is a sequence of tuples $\{(y_k, x_k)\}_{k=1}^{N}$. The number of time steps N in the sequence is varied as needed for training. Each sequence is started at random non-zero initial conditions and in order to excite all of the modes of our system, we add Gaussian white noise inputs at each time step. The

simulated data is split into a training set and validation set, using a standard 80%-20% split. The size of the dataset is varied as required for each example.

We train the RNNs using the measurements y_k as the features and the true states x_k as the labels. The goal is for the RNN to produce accurate state estimates $\hat{x}_k$ given measurements y_k.

The RNN architecture consists of one or more LSTM hidden layers, combined with a feedforward output layer. The RNN is constructed using the PyTorch 1.13 [33] machine learning software library. We train the weights of this network by minimizing the Mean Squared Error cost function, using the Adam optimizer. The training is conducted using mini-batches from our training set, the sizes of which are varied as needed.

The hyperparameters of the RNN are the number of LSTM layers, the number of LSTM cells in each layer, the learning rate of the Adam optimizer and the number of epochs to train for. The values for these hyperparameters were first tuned manually. Then, using these values as a starting point, random search then Bayesian optimization using TPE was conducted to seek improved hyperparameter combinations.

The RNN-based estimator's performance is compared against a benchmark Kalman filter on a previously unseen test set. The test set is generated in the same manner as the training and validation sets, and its size is chosen to be equal to that of the validation set. For each sequence in the test set, the mean squared errors of the two estimators with respect to the true states is used as a measure of accuracy. Furthermore, we also measure the execution times of each estimator. The average of these over the test data is compared.

The benchmark Kalman filter's performance is verified by comparing the mean squared

errors of the state estimate produced on the validation set against the average trace of the covariance matrix P of the estimation error produced by the Kalman filter at the end of each batch. Since the Kalman filter minimizes the mean squared error of its estimates which is equivalent to $\text{tr}(P)$ and we can find a unique minimum for this by solving the discrete Lyapunov equation, we first establish that the Kalman filter's mean square error on the validation set is close to this value.

For each evaluation of the Kalman filter, the initial estimate is set to zero, while the estimation error covariance matrix is set to an identity matrix of appropriate dimensions. Similarly, the RNN hidden state and LSTM cell state are zero-initialized as well.

The PyTorch library makes extensive use of parallel programming. On modern computers, the speedup caused by parallelism is known to be more significant for algorithms evaluated on high dimensional data [34], as increased CPU and GPU core counts and random-access memory allow for many simultaneous operations and advanced algorithms for linear algebra, sometimes built into the CPU itself, enable efficient computation. To allow a fair comparison, our Kalman filter must be parallelized where possible as well. Hence, our Kalman filter is implemented using the CuPy [35] software package, which provides a GPU-accelerated version of a subset of the functionality of the popular NumPy [36] linear algebra software package.

All code was compiled and executed on a workstation utilizing: AMD Threadripper 1950X 16-core CPU; NVIDIA GTX 1070 GPU; 128GB DDR4-2666 RAM.

5.3 Comparing computational cost to classical algorithms

As we derived in Chapter 2, the time complexity of the Kalman filter is $O(2n(m^2 + 2n^2 + m) + 3n^2(m+2) + m + n)$ per iteration, where n is the size of the state vector and m is the size of the measurement vector. We would like to set some expectations about the runtime of a RNN-based estimator by comparing its time complexity to that of a Kalman filter. For our examples we use LSTMs, so here we derive the time complexity of a one-layer LSTM operating for one time step.

In the case of a neural network with LSTM layers, we also need to take into account the recurrent connections in the LSTM cells. In the case of a standard RNN, its hidden layers compute

$$h_t = \tanh(W_{ih}y_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}), \tag{5.1}$$

where $W_{ih} \in \mathbb{R}^{h\times m}$ is the input weight, $y_t \in \mathbb{R}^m$ is the input vector, $b_{ii} \in \mathbb{R}^h$ is the input bias, $W_{hh} \in \mathbb{R}^{h\times h}$ is the matrix of recurrent connections, $h_{t-1} \in \mathbb{R}^h$ is the hidden state at time step $t-1$ and finally $b_{hh} \in \mathbb{R}^h$ is the bias for the recurrent connections. It is easy to see that this is analogous to the computation of one of the gates in a LSTM cell. Hence, we can derive the time complexity of a standard RNN by deriving that of a LSTM, so we will not do this separately.

In order to stay true to the calculations done in our examples, we use the notation used in the PyTorch [33] implementation. For the evaluation of one hidden layer of LSTM cells

across one time step, the required computations are represented by

$$\begin{aligned}
i_t &= \sigma\left(W_{ii}y_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}\right)\\
f_t &= \sigma\left(W_{if}y_t + b_{if} + W_{hf}h_{t-1} + b_{hf}\right)\\
g_t &= \tanh\left(W_{ig}y_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}\right)\\
o_t &= \sigma\left(W_{io}y_t + b_{io} + W_{ho}h_{t-1} + b_{ho}\right)\\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t\\
h_t &= o_t \odot \tanh\left(c_t\right),
\end{aligned} \tag{5.2}$$

where $\odot$ is element-wise multiplication. Note that this deviates from the LSTM described in Chapter 3, pp. 28 - 32 by using the hyperbolic tangent function in the weights g_t for the external input gate, while the previously described LSTM uses the sigmoid function.

We start with calculating the time complexity of i_t (the input gate). Recall that f_t (forget gate), g_t (cell) and o_t (output gate) work in the same way, so they will have identical time complexity. The equation for the input gate is

$$i_t = \sigma\left(W_{ii}y_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}\right), \tag{5.3}$$

where $W_{ii} \in \mathbb{R}^{h \times m}$ is the input gate weight, $x_t \in \mathbb{R}^m$ is the input vector, $b_{ii} \in \mathbb{R}^h$ is the input gate bias, $W_{hi} \in \mathbb{R}^{h \times h}$ is the matrix of recurrent connections, $h_{t-1} \in \mathbb{R}^h$ is the hidden state at time step $t-1$ and finally $b_{hi} \in \mathbb{R}^h$ is the bias for the recurrent connections. Now, looking at each operation inside the equation, we first see $W_{ii}y_t$ has time complexity $O(hm)$, due to the properties of matrix-vector multiplication. Hence, $W_{ii}x_t + b_{ii}$ has time

complexity $O(hm + h)$ because it is the sum of two vectors, both with size h. Similarly, $W_{hi}h_{t-1}$ has time complexity $O(h^2)$, so $W_{hi}h_{t-1} + b_{hi}$ has time complexity $O(h^2 + h)$. Thus, the time complexity of the external input gate is

$$O(hm + h + h^2 + h) = O(hm + 2h + h^2) = O(h(m + 2 + h)). \tag{5.4}$$

As the time complexity of the sigmoid function is $O(h)$, the time complexity of the input gate is $O(h(m + 2 + h))$. As the two remaining gates have the same time complexity, we get $O(3h(m + 2 + h))$ for the time complexity after computing the three gates in the LSTM cells.

Similarly, the time complexity of the hyperbolic tangent function is $O(h)$, the time complexity before the LSTM cell update is $O(4h(m + 2 + h))$.

For the cell update

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t, \tag{5.5}$$

where $c_t \in \mathbb{R}^h$ is the LSTM cell state, as element-wise multiplication has time complexity $O(h)$, we get a time complexity of $O(2h)$. Anologously, the computation of the hidden state at time step t

$$h_t = o_t \odot \tanh\left(c_t\right), \tag{5.6}$$

also has time complexity $O(2h)$.

Therefore, the overall time complexity of a LSTM is the summed time complexity of its parts,

$$O(4h(m + 2 + h) + 4h) = O(4h(m + 3 + h)). \tag{5.7}$$

The time complexity of one iteration of the LSTM is overall quadratic with respect to the hidden layer size, but will also scale linearly with respect to the input size. If we use more than one hidden layer, than the time complexity would scale linearly with this as well.

In comparison, the Kalman filter has a time complexity that is cubic with respect to the system order. Hence, in cases where a LSTM has a hidden layer size not much larger than the system order, it may have better runtime than a Kalman filter. However, this will ultimately depend on the exact architecture used.

5.4 Connected mass-spring-damper systems

The mass-spring-damper system is a standard dynamical system which is taught as an example in many classes. It is a useful system to study because it is representative of many kinds of oscillating physical processes.

Like many mechanical systems, Newton's laws are used in the analysis of this dynamical system. Newton's second law,

$$\sum F = ma = m\frac{\partial^2 x}{\partial t^2}, \tag{5.8}$$

states that the sum of the forces acting on a body equals the product of its mass and acceleration. Newton's third law states that whenever a body exerts a force on another object, then the second object exerts a contact force of the same magnitude, acting in the opposite direction.

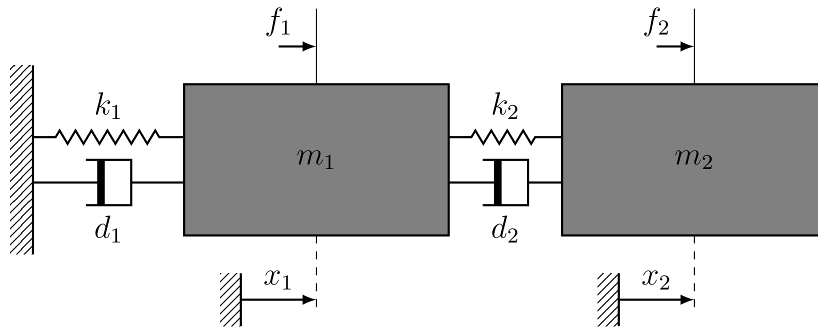

Figure 5.1: Free-body diagram of a two degree-of-freedom mass-spring-damper system.

As shown in the free-body diagram in Figure 5.1, the spring forces are proportional to the displacement of the masses acting on it, x_1 and x_2. We consider the displacement of the masses to the right to be the positive direction. While only one mass is acting on the second spring, we must be careful to note that both masses are acting on the first spring. Similarly, the viscous damping force is proportional to the velocities of the masses, $\dot{x_1}$ and $\dot{x_2}$. Since both forces are opposite the motion of the masses, they are in the negative direction.

Now, summing the forces acting on each mass and applying Newton's second law, we can describe a system of n connected mass-spring-dampers as a system of differential

equations

$$
\begin{aligned}
m_1\ddot{x}_1(t) &= -k_1x_1(t) + k_2(x_2(t) - x_1) - d_1\dot{x}_1(t) + d_2(\dot{x}_2(t) - \dot{x}_1(t)) + f_1(t) \\
m_2\ddot{x}_1(t) &= -k_2(x_2(t) - x_1(t)) + k_3(x_3(t) - x_2(t)) - d_2(\dot{x}_2(t) - \dot{x}_1(t)) + d_3(\dot{x}_3(t) - \dot{x}_2(t)) + f_2(t) \\
m_3\ddot{x}_3(t) &= -k_3(x_3(t) - x_2(t)) + k_4(x_4(t) - x_3(t)) - d_3(\dot{x}_3(t) - \dot{x}_2(t)) + d_4(\dot{x}_4(t) - \dot{x}_3(t)) + f_3(t) \\
&\vdots \\
m_n\ddot{x}_n(t) &= -k_3(x_n(t) - x_{n-1}(t)) - d_n(\dot{x}_n(t) - \dot{x}_{n-1}(t)) + f_n(t), \qquad (5.9)
\end{aligned}
$$

where $m_1, \ldots, m_n$ are the weights of each mass, $k_1, \ldots, k_n$ are the spring coefficients of the springs above each mass, $d_1, \ldots, d_n$ are the damping coefficients, and $f_1, \ldots, f_n$ are the inputs applied to each mass. Written in state space form, this becomes

$$
\begin{aligned}
\dot{x}(t) &= Ax(t) + Bu(t) \\
y(t) &= Cx(t) + Du(t), \qquad (5.10)
\end{aligned}
$$

with matrices

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ -\frac{k_1+k_2}{m_1} & -\frac{d_1+d_2}{m_1} & \frac{k_2}{m_1} & \frac{d_2}{m_1} & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots \\ \frac{k_2}{m_2} & \frac{d_2}{m_2} & -\frac{k_2+k_3}{m_2} & -\frac{d_2+d_3}{m_2} & \frac{k_3}{m_2} & \frac{d_3}{m_2} & 0 & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \dots & \dots & \dots \\ 0 & 0 & \frac{k_3}{m_3} & \frac{d_3}{m_3} & -\frac{k_3+k_4}{m_3} & -\frac{d_3+d_4}{m_3} & \frac{k_4}{m_3} & \frac{d_4}{m_3} & 0 & \dots \\ & & & & \ddots & & & & & \\ 0 & \dots & \dots & \dots & \dots & \dots & \frac{k_n}{m_n} & \frac{d_n}{m_n} & -\frac{k_n}{m_n} & -\frac{d_n}{m_n} \end{bmatrix}, \tag{5.11}$$

$$B = \begin{bmatrix} 0 & \dots & \dots & \dots \\ \frac{1}{m_1} & 0 & \dots & \dots \\ 0 & \dots & \dots & \dots \\ 0 & \frac{1}{m_2} & 0 & \dots \\ & & \ddots & \\ 0 & \dots & \dots & \frac{1}{m_n} \end{bmatrix}. \tag{5.12}$$

The state and input vectors are

$$x(t) = \begin{bmatrix} x_1(t) \\ \dot{x}_1(t) \\ \vdots \\ x_n(t) \\ \dot{x}_n(t) \end{bmatrix} \text{ and } u(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_{n-1}(t) \\ f_n(t) \end{bmatrix}. \tag{5.13}$$

The input vector is given at each time step, representing input force on each of the n masses. The measurement matrix C is a $n \times 2n$ matrix such that the $(2i-1)$-th element of the ith row is 1 and all other elements are 0 while D is a zero matrix, so our measurement vector contains only the positions of each mass. The state vector contains the displacement and velocity of each mass in the system, and is to be estimated.

In the next three sections, we present some results utilizing an RNN to estimate the state of the spring-mass-damper system shown above. For each example, we construct a dataset by simulating the system. We first discretize the continuous-time system using the zero-order hold method, where we assume each sample value is fixed until the next sample time. The sampling rate is chosen to be higher than the Nyquist rate, which is twice the frequency of the highest frequency component of the system. The effects of discretization are not considered in this work. The system is then made noisy by adding process noise w_k and measurement noise v_k to the process and measurement equations respectively. The noises are Gaussian, zero mean and uncorrelated.

5.5 Example: 5 DoF mass-spring-damper system

We begin with a small mass-spring-damper system with five connected masses. The masses, the spring constants and the damping constants were chosen uniformly at random. The masses were drawn from the interval $[50, 200]$, the spring constants from $[500, 3000]$ and the damping constants from $[2, 20]$. At each time step, a Gaussian white noise input with zero mean and covariance matrix $\text{diag}(10 \dots 10) \in \mathbb{R}^{5\times5}$ is applied. There is no correlation between the input force applied to one mass and the input force applied to another. Furthermore, we inject Gaussian process noise with zero mean and the covariance matrix $Q = \text{diag}(0.01, \dots, 0.01) \in \mathbb{R}^{10\times10}$, and Gaussian measurement noise with zero mean and the covariance matrix $R = \text{diag}(0.0625, \dots, 0.0625) \in \mathbb{R}^{5\times5}$.

The highest frequency component was found at 15.28Hz. The system was discretized using the zero-order hold method, with a sampling rate of 0.01s.

We generate a dataset consisting of 640 independent sequences of 100 time steps, with each initial state variable drawn uniformly from the interval $[-50, 50]$. This was further separated into twenty batches of 32 sequences each. Of these, four batches were selected at random to form a validation set. The remaining batches were used to form the training set.

The RNN estimator consists of one hidden layer of 25 LSTM cells, giving a total of 4750 trainable parameters. The output layer is linear, applying a linear transformation to the final hidden states of the LSTM, and producing a state estimate of the connected mass-spring-damper system. Training was conducted for 450 epochs, with the learning rate of the Adam optimizer set to 0.001.

Separately, we generated a test set consisting of four batches of sequences of length 100, randomly initialized to non-zero values. Applying the Kalman filter to the test data produced a mean squared error of 8.532, averaged across twenty batches. Evaluation of our trained RNN on the test data produced a mean squared error of 8.097, a 5.09% improvement compared to the Kalman filter. The theoretical minimum mean squared error, found by solving the discrete Lyapunov equation was 8.730. The mean squared errors across one sample sequence in the test set is shown in Figure 5.2. We were able to train the RNN-based estimator to exceed the performance of the Kalman filter. However, much of the error of the Kalman filter is concentrated in the first ten time steps. The Kalman filter and the RNN-based estimator performed similarly in subsequent time steps.

The training time for the RNN was 21.4 seconds. The execution time of the RNN-based estimator on the test set was 14 milliseconds. In comparison, the execution time of the Kalman filter on the same data was 19 milliseconds. Not only were we able to achieve a better estimate with the RNN-based estimator, it also does so faster than the Kalman filter.

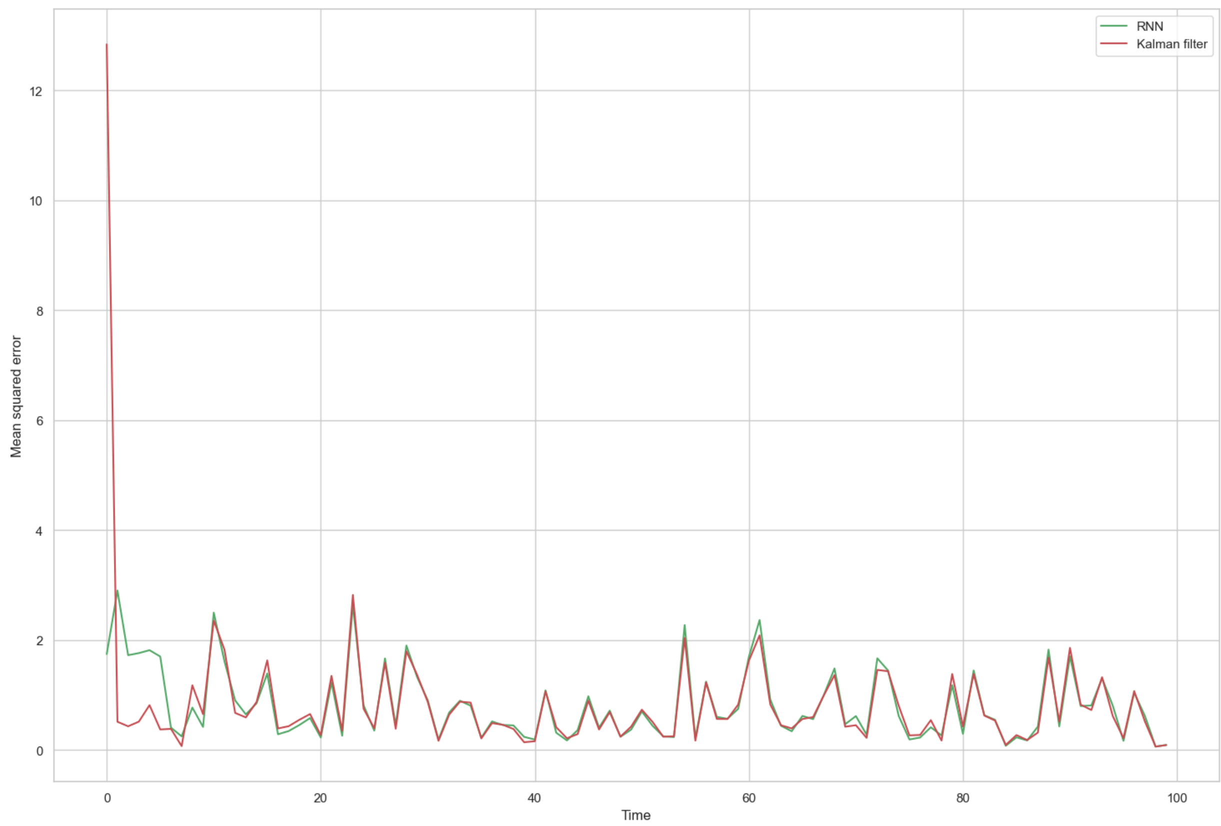

Figure 5.2: Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set.

5.6 Example: 100 DoF mass-spring-damper system

We seek now to evaluate the efficacy of a RNN-based estimator on a larger dynamical system. Here, we construct a RNN-based estimator for a mass-spring-damper system with 100 connected masses. The state vector of this system contains the displacement and velocity for each of the 100 masses, for a total of 200 state variables to estimate.

The masses, spring constants and damping constants were chosen uniformly at random. The masses were drawn from the interval $[50, 200]$, the spring constants from $[500, 3000]$ and the damping constants from $[2, 20]$. At each time step, a Gaussian white noise input with zero mean and covariance matrix $\text{diag}(10 \dots 10) \in \mathbb{R}^{100\times100}$ is applied. There is no correlation between the input force applied to one mass and the input force applied to another. Furthermore, we inject Gaussian process noise with zero mean and the covariance matrix $Q = \text{diag}(0.01, \dots, 0.01) \in \mathbb{R}^{200\times200}$, and Gaussian measurement noise with zero mean and the covariance matrix $R = \text{diag}(0.0625, \dots, 0.0625) \in \mathbb{R}^{100\times100}$.

The highest frequency component was found at 8.24Hz. The system was discretized using the zero-order hold method, with a sampling rate of 0.01s.

We generate a dataset consisting of 3200 independent sequences of 100 time steps, each with random initial state variables drawn uniformly from the interval $[-50, 50]$. This was further separated into 100 batches of 32 sequences each. Of these, twenty batches were selected at random to form a validation set. The remaining batches were used to form the training set.

With 100 connected masses, we now have a dynamical system of order 200. Here, we start to see the curse of dimensionality come into play. While we were able to train a RNN-

based estimator which exceeded the performance of the Kalman filter with a dataset of 640 sequences, for our larger system we were only able to obtain satisfactory performance when training with 3200 sequences. Furthermore, as we have twenty times the number of state variables, our dataset is 100 times the size of the dataset used to train the RNN to estimate the 5 DoF mass-spring-damper model presented in the previous section. This is easily seen from the file size of the datasets– 2.23 gigabytes for the 100 DoF system, compared to the 23.4 megabyte file size for the 5 DoF system. As a result, we see a signficant increase in the amount of time required for data generation– 2.3 minutes for the 5 DoF system, and 27 minutes For the 100 DoF system. Note that the time required should not be expected to scale linearly, especially as the systems being simulated become high dimensional, as the matrix multiplications in the simulation requires $O(n^3)$ time.

We constructed our RNN with one hidden layer of 200 LSTM cells, giving $281,800$ trainable parameters. While other configurations were considered, we found that the hidden layer size of 25 used in the 5 DoF system case did not provide enough representational capacity to fit the high-dimensional data. On the other hand, we also found no significant benefit in using hidden layer sizes larger than 200 or using additional layers.

The Kalman filter applied to the test set produced a mean squared error of 195.647. Training for 580 epochs with a learning rate of 0.0005 over the course of 32 minutes, the RNN-based estimator produced a mean squared error of 176.511, a 9.53% improvement compared to the Kalman filter as shown in Figure 5.3. The theoretical minimum mean squared error was 215.813. However, much of the error of the Kalman filter is concentrated in the first ten time steps. The Kalman filter and the RNN-based estimator performed similarly in the subsequent time steps.

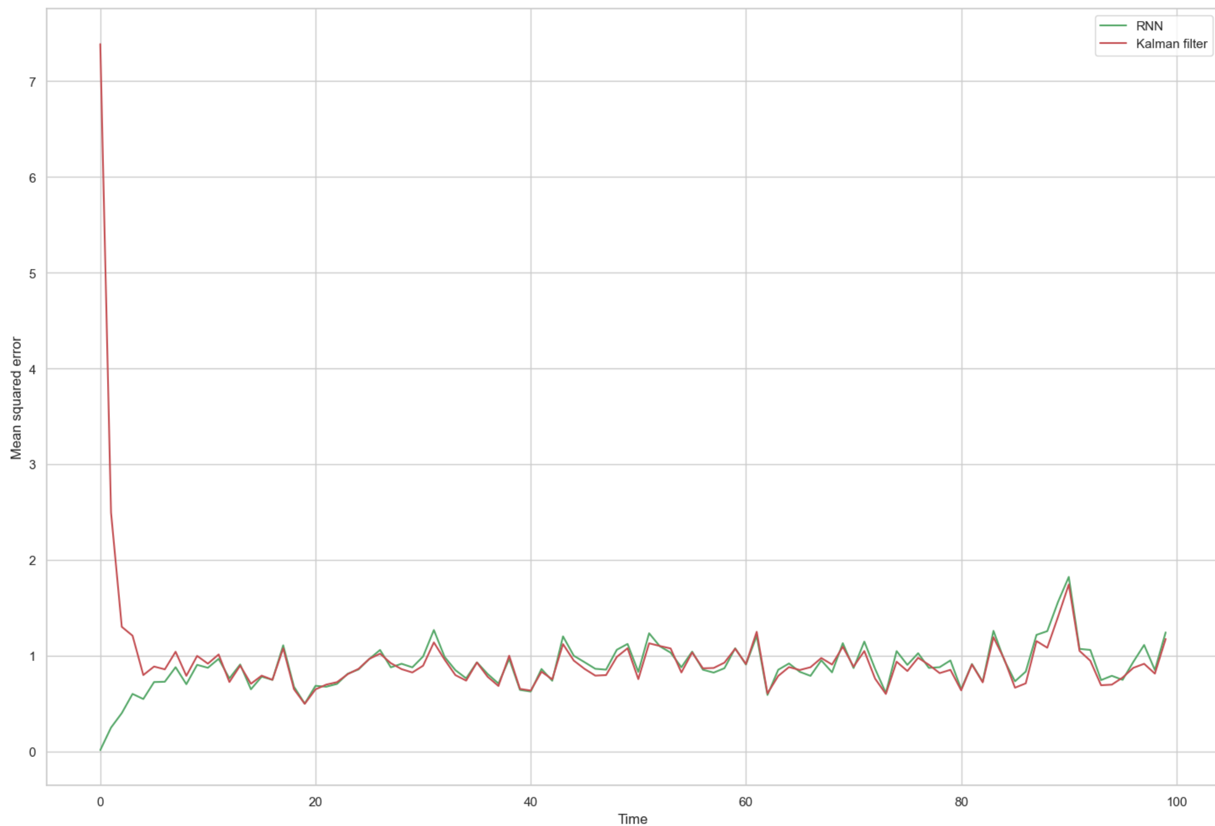

Figure 5.3: Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set.

We were once again able to produce a RNN-based estimator with better performance than the Kalman filter. The advantage that the RNN's parallelism poses is even more noticeable with this larger system– the execution time of the RNN-based estimator is a 230 milliseconds on the test set, compared to 325 milliseconds for the Kalman filter.

The results for the two mass-spring-damper examples are summarized in Table 5.1.

System	RNN parameters				MSE	Kalman filter MSE	Runtime (ms)	KF runtime (ms)
	Number of layers	LSTM cells per layer	Learning rate	Epochs				
5 DoF	1	25	0.001	450	8.097	8.532	14	19
100 DoF	1	200	0.0005	580	176.511	195.647	230	325

Table 5.1: Table of values for RNN-based estimators on connected mass-spring-damper systems.

5.7 Example: transfer learning to a clamped-free beam model

Our numerical examples on estimating the states of connected mass-spring-damper systems of varying orders has shown that a RNN-based estimator could be a viable and even a faster alternative to a Kalman filter. However, one significant disadvantage of utilizing RNNs is the required training process, which we would like to shorten.

To this end, we would like to see if a RNN-based estimator trained on data from one dynamical system would perform well when applied to a different system with similar dynamics. We also seek to find a way to utilize the results of training a RNN-based estimator of one dynamical system to facilitate better training on a different system with similar dynamics.

As we mentioned earlier, the mass-spring-damper system bears similarity to many kinds of oscillating processes. One similar oscillating process is that of an clamped-free beam.

The clamped-free beam is shown in Figure 5.4. The system is a single beam connected to a hub on one end. The beam is moved by applying torque at the hub. This system is modeled by the Euler-Bernoulli equation, which is a partial differential equation relating the static deflection deflection of a beam $w(x)$ to its bending stiffness EI and applied load

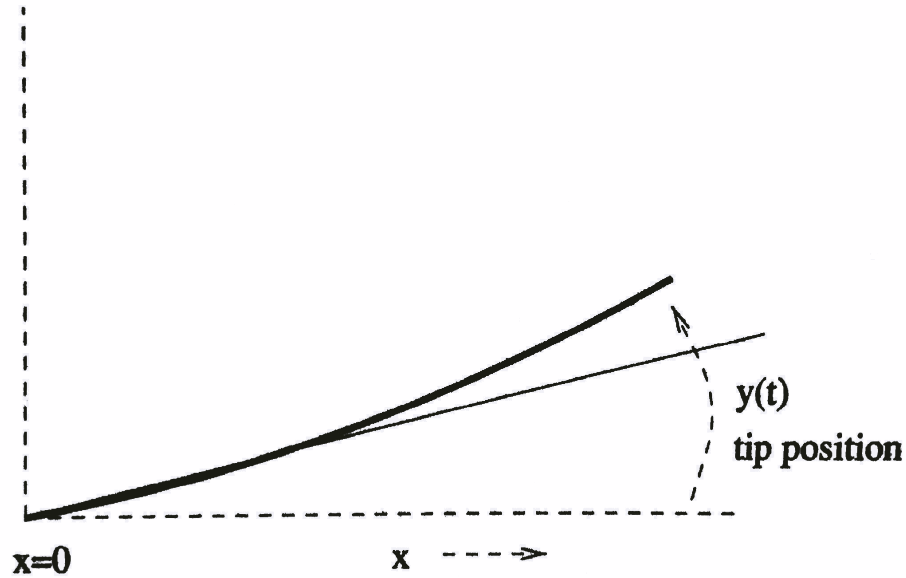

Figure 5.4: A diagram of an clamped-free beam. The beam rotates by torque applied at the hub $x = 0$. [2], p. 180

q, written as

$$EI\frac{\partial^4 w}{\partial x^4} = q. \tag{5.14}$$

Since this is a fourth-order partial differential equation, we have four initial conditions to consider. In the case of the clamped-free beam, these are

$$\begin{aligned} w(0) &= w'(0) = 0, \\ w'''(L) &= w''(L) = 0. \end{aligned} \tag{5.15}$$

By taking a truncation of the Fourier transform of this partial differential equation, we can obtain a system of linear ordinary differential equations which approximates the above dynamical system. For our example, we take an example from [2], obtaining a tenth-order

dynamical system,

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du, \tag{5.16}$$

with the process matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & 1 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & -\omega_1^2 & -2\zeta\omega_1 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & 1 & 0 & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & -\omega_2^2 & -2\zeta\omega_2 & 0 & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 & 1 & 0 & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & -\omega_3^2 & -2\zeta\omega_3 & 0 & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 1 \\ 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & -\omega_4^2 & -2\zeta\omega_4 \end{bmatrix} \tag{5.17}$$

and

$$B = \frac{1}{0.0829}\begin{bmatrix} 0 \\ 1 \\ 0 \\ 2.886 \\ 0 \\ -2.345 \\ 0 \\ -0.910 \\ 0 \\ -0.454 \end{bmatrix}. \tag{5.18}$$

The measurement matrix C is the 10×10 identity matrix and D is a matrix with all elements zero. The first four natural frequencies are $\omega_1 = 55.89$, $\omega_2 = 131.74$, $\omega_3 = 313.81$ and $\omega_4 = 603.67$ radians per second. The damping ratio for each frequency is set to $\zeta = 0.002$.

At each time step, a Gaussian white noise input torque with zero mean and variance one is applied. Furthermore, we inject Gaussian process noise with zero mean and the covariance matrix $Q = \text{diag}(0.02, \ldots, 0.02) \in \mathbb{R}^{10\times10}$, and Gaussian measurement noise with zero mean and the covariance matrix $R = \text{diag}(0.05, \ldots, 0.05) \in \mathbb{R}^{10\times10}$.

For our simulation, we discretize this system with the zero-order hold method, choosing a sampling time of 0.002 seconds. We then generate 640 sequences of 100 time steps to produce our training data.

We first train an RNN-based estimator for this system. As the 5 DoF mass-spring-damper and the clamped-free beam are both systems of order ten, we use the same configuration for the RNN, constructing it with one hidden layer of 25 LSTM cells. Splitting our training set into twenty batches of 32 sequences each, we trained for 300 epochs with a learning rate of 0.001. The training required 170.3 seconds on our machine.

The RNN-based estimator produced a mean squared error of 495.590 over the test set. The Kalman filter produced a mean squared error of 514.615. Hence, the RNN-based estimator's performance was comparable to Kalman filter, with less than 5% difference in MSE. The theoretical minimum mean squared error was 496.194.

We next attempt to improve the training of the RNN-based estimator by conducting what is known as *transfer learning* [37]. Transfer learning is a training technique that allows the knowledge gained from training one neural network on a specific task to be transferred to a different but related task. It is particularly useful when the target task has limited training data, but there is a large amount of training data available for a related task.

The basic idea behind transfer learning is to use a pre-trained model as a starting point, and then fine-tune it on the target task. The pre-trained model is typically trained on a large dataset and has learned a set of features that are useful for many different tasks. These features can then be used as a starting point for the target task, rather than starting from scratch.

There are several ways to perform transfer learning, depending on the similarity between the source and target tasks. The two most common approaches are:

- *Feature extraction.* In this approach, the pre-trained model is used as a fixed feature extractor, and only the final layer(s) of the model are replaced and trained on the target task. This is particularly useful when the source and target tasks are similar in terms of the input features, but different in terms of the output labels. See [38] for an example of this approach in the context of classifying brains tumors using a convolutional neural network (CNN) trained on brain MRI images.

- *Fine-tuning.* In this approach, the pre-trained model is refined on the target task by updating the weights of some or all of the layers in the model. This is particularly useful when the source and target tasks are similar in terms of both the input features and the output labels. See [39] for an application of this approach in the context of training an LSTM model to predict the health of industrial manufacturing machines.

Transfer learning has several advantages over training a model from scratch. First, it can significantly reduce the amount of training data required for the target task, as the pre-trained model has already learned a set of useful features. This can be particularly useful in applications where labeled data is expensive or difficult to obtain. It can also reduce the amount of training required, lowering the computational cost of training.

Transfer learning can also help to improve the generalization error of the model, by providing a better starting point for the optimization process. This is useful when the target task has a limited number of labeled examples, as it can help to avoid overfitting.

As both our clamped-free beam system and 5 DoF mass-spring-damper system are dynamical systems of order ten and only five of ten state variables are measured, both RNN-based estimators can have the same architecture. Hence, we take the fine-tuning

approach, reusing the parameters of the trained RNN-based estimator for the 5 DoF mass-spring-damper system as the initial conditions of the RNN-based estimator to be trained for the clamped-free beam system.

As can be seen in Figure 5.5, we immediately found that the training of this new RNN-based estimator started with training and validation errors much lower than that of the previous RNN-based estimator. This led to a significant reduction in the number of epochs required for training to converge, with the new RNN converging at approximately epoch 13, while the RNN trained from scratch converged at epoch 30. Furthermore, we found that the training losses and validation losses achieved by the weight transferred RNN were on average lower than that of the RNN initialized with default settings. Our new RNN-based estimator achieved a mean square error of 471.718, a 8.34% improvement in accuracy over the Kalman filter.

As they share the same architecture, the execution times of the two RNN-based estimators were virtually indistinguishable, clocking in at 14 milliseconds and 15 milliseconds on the test set. In comparison, the Kalman filter required 18 milliseconds.

The mean squared errors produced by each estimator on the test set are shown in Figure 5.6. We find that transfer learning can reduce the training time and improve training outcome when creating RNN-based estimators for systems with similar dynamics.

The results for the damp-free beam examples are summarized in Table 5.2.

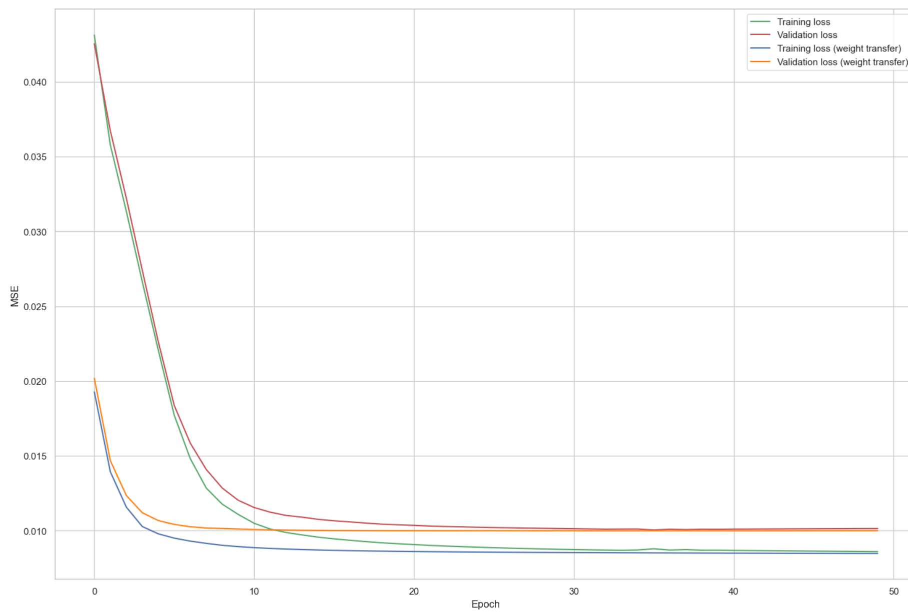

Figure 5.5: Training and validation errors over 50 epochs for two RNN-based estimators on the clamped-free beam system. The first RNN was trained with default initial conditions. The second RNN was transferred the weights and biases from a trained RNN-based estimator for a five degree-of-freedom connected mass-spring-damper system prior to training.

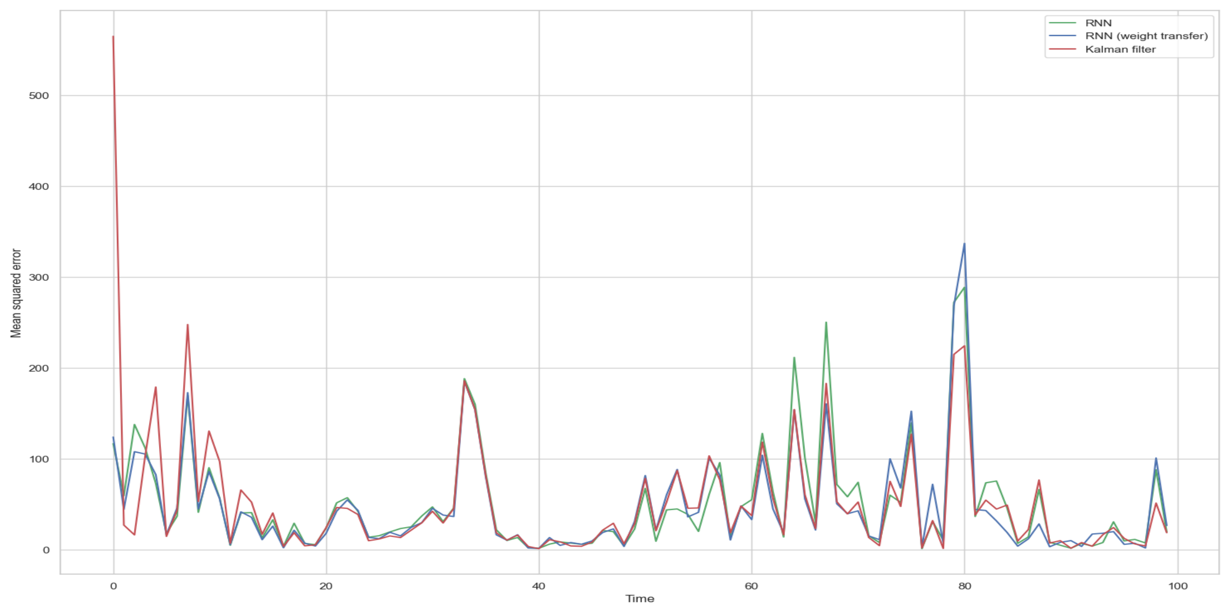

Figure 5.6: Mean squared errors for the RNN-based estimator (green), RNN-based estimator initialized with weights from the RNN trained on the 5 DoF mass-spring-damper system (blue) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set.

Reused weights	RNN parameters			MSE	Kalman filter MSE	Runtime (ms)	KF runtime (ms)
	LSTM cells	Learning rate	Epochs				
No	25	0.001	300	495.590	514.615	14	18
Yes	25	0.0005	100	471.718	514.615	15	18

Table 5.2: Table of values for RNN-based estimators on a clamped-free beam system.

5.8 Example: randomly generated dynamical systems

We now would like to see if we are able to create performative RNN-based estimators for arbitrary dynamical systems. To this end, we randomly generate a system of order 20 with ten inputs and ten outputs. For our second example, we modify this system to induce instability. The dynamical systems were generated with the `drss` function in MATLAB r2022b, which generates random discrete-time LTI (linear, time-invariant) test models with unspecified sampling rates. Furthermore, we verified that for both systems the current state can be estimated using only the information from the measurements, a condition called *observability* (see [5], Chapter 1, Section 1.7.2). For LTI systems of the form

$$
\begin{aligned}
x_k &= Ax_{k-1} + Bu_k \\
y_k &= Hx_k + Du_k, \qquad (5.19)
\end{aligned}
$$

one way to check for observability is to see if the observability matrix

$$\begin{bmatrix} H \\ HA \\ \vdots \\ HA^{n-1} \end{bmatrix} \tag{5.20}$$

has column rank n. Then, each state variable can be represented as a linear combination of the measurement variables y_k. As these systems are random, there is no physical interpretation attached to the state.

5.8.1 Random dynamical system of order 20

At each time step, we inject Gaussian process and measurement noises with zero mean. For this system, they are covariances $Q = \text{diag}(1, \ldots, 1) \in \mathbb{R}^{20\times 20}$ and $R = \text{diag}(2.5, \ldots, 2.5) \in \mathbb{R}^{10\times 10}$, respectively. Furthermore, to ensure persistent excitation of the system, we apply a Gaussian white noise input with zero mean and covariance $\text{diag}(1.5, \ldots, 1.5) \in \mathbb{R}^{20\times 20}$ as well.

For our RNN, we choose an architecture of one hidden layer containing thirty LSTM cells, yielding 5660 trainable parameters. The training set contains 160 batches of 32 sequences, each of which are length 100 with state variables initialized randomly from a uniform distribution on the interval $[-10, 10]$. We train the RNN to estimate the states of this system for 1350 epochs using the Adam optimizer, with a learning rate of 0.0015. The values of the hyperparameters were chosen through trial-and-error, as well as Bayesian

optimization.

We found training our RNN-based estimator for this system to be more challenging than the previous systems. We attribute this to the random nature of this system. In the case of the connected mass-spring-dampers, we know that every second state variable are velocities and hence intrinsically tied to the state variables preceding it (the displacement of the corresponding masses). As randomly generated dynamical systems have no physical interpretation, it is natural that a RNN-based estimator for this system would require a combination of greater model capacity and larger datasets to train.

The RNN-based estimator gave a mean squared error of 23.527 on the test set. In comparison, the mean squared error of the Kalman filter was 22.422, a 4.93% percent better accuracy. The theoretical minimum mean squared error was found to be 18.737.

The training of the RNN required 192 seconds. The execution time of the RNN-based estimator was 12 milliseconds, while that of the Kalman filter was 26 milliseconds.

We were able to train an RNN-based estimator to perform comparably to a Kalman filter in estimating the states of a randomly generated LTI system of order twenty. However, we found this to be more challenging than the previous examples, requiring extensive optimization of the RNN's hyperparameters.

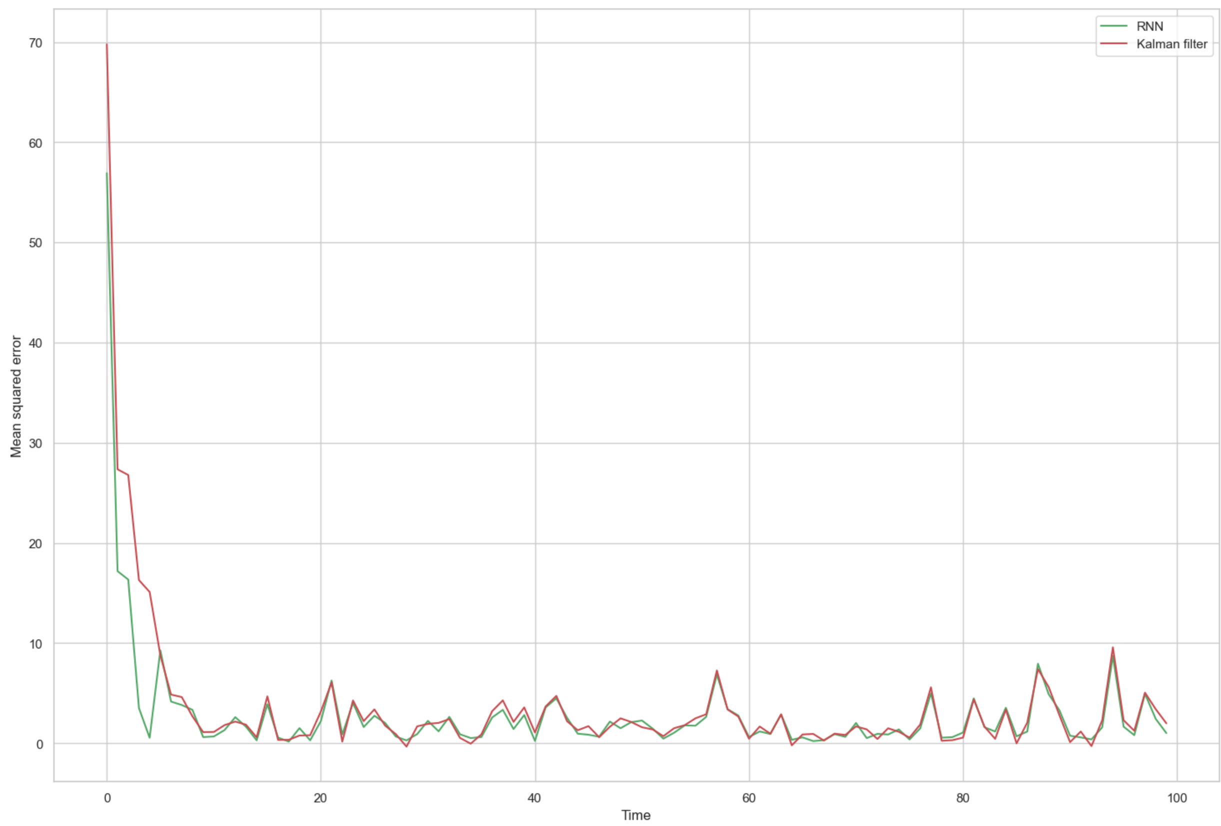

Figure 5.7: Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 100 time steps contained in each sequence in the test set.

5.8.2 Unstable random dynamical system of order 20

We are now interested in whether or not we can train a RNN-based estimator for an unstable LTI system. A system is *stable* when its output signal is bounded and does not increase to infinity as time goes on. LTI systems of the form

$$x_k = Ax_{k-1} + Bu_k$$
$$y_k = Hx_k + Du_k \tag{5.21}$$

are stable when the real parts of the eigenvalues of the system matrix A have absolute values less than or equal to 1. As the sum of the real parts of the eigenvalues of A is equal to $\mathrm{tr}(A)$, we can induce instability by adding a diagonal matrix with diagonal elements greater than or equal to one. Hence, we create an unstable system by adding the identity matrix to the system matrix A of the random system of order 20 from the previous example. This new system was verified to be observable by looking at the rank of its observability matrix.

For our RNN, we choose an architecture of one hidden layers each containing 32 LSTM cells for a total of 6292 trainable parameters. The training set contains 120 batches of 32 sequences, each of which are length 50. The state variables were initialized randomly from a uniform distribution on the interval $[-10, 10]$. The short sequence lengths were necessitated by the unstable system, as longer sequences would blow up. We also found compensating for the shorter sequences with a large number of batches to be detrimental to training performance, requiring us to use less data. We train the RNN to estimate the

states of this system for 500 epochs using the Adam optimizer, with a learning rate of 0.01. The values of the hyperparameters were chosen through trial-and-error, as well as Bayesian optimization.

The mean squared errors produced on the test set are shown in Figure 5.8. Our RNN-based estimator gave a mean squared error of 65.102 on the test set. In comparison, the mean squared error of the Kalman filter was 49.534, a 41.43% better accuracy. The theoretical mean squared error was 59.83.

The training of the RNN required 251 seconds. The execution times of the RNN-based estimator and Kalman filters were 8 milliseconds and 10 milliseconds, respectively.

Despite intensive effort, we found it challenging to train a RNN-based estimator to obtain performance comparable to a Kalman filter in this case. The results for the two mass-spring-damper examples are summarized in Table 5.3.

System	**RNN parameters**			**MSE**	**Kalman filter MSE**	**Runtime (ms)**	**KF runtime (ms)**
	LSTM cells per layer	**Learning rate**	**Epochs**				
Order 20	30	0.0015	1350	23.527	22.422	12	26
Unstable	32	0.01	500	65.102	49.534	8	10

Table 5.3: Table of values for RNN-based estimators on randomly generated LTI systems.

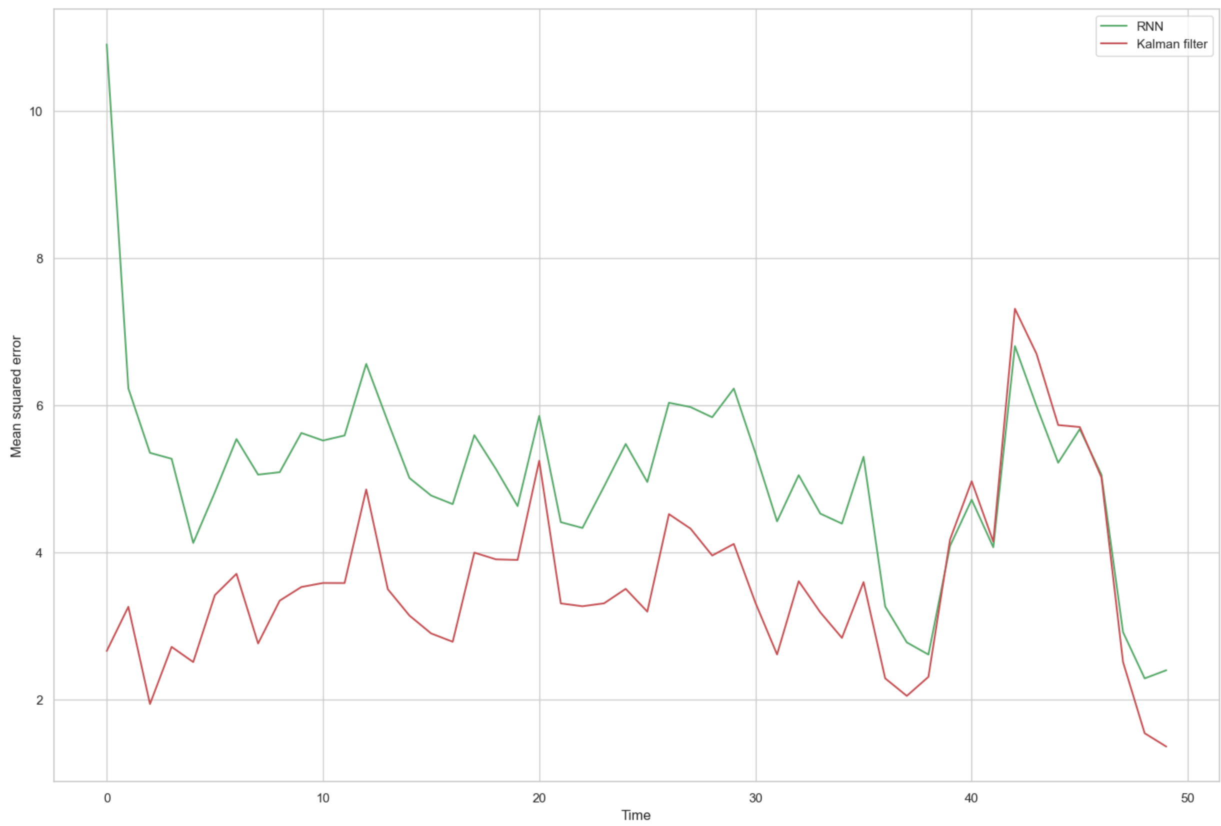

Figure 5.8: Mean squared errors for the RNN-based estimator (green) and Kalman filter (red) for the 50 time steps contained in each sequence in the test set.

5.9 Example: Lorenz system

As our final example, we construct a RNN-based estimator for the Lorenz attractor, a nonlinear system which is known for the butterfly effect [40], describing the phenomenon where a small change in one state of a deterministic nonlinear system can result in large differences in a later state. We compare the performance of this against the Extended Kalman filter. The Lorenz system is described by

$$\begin{aligned}\dot{x} &= \sigma(y - x)\\ \dot{y} &= x(\rho - z) - y\\ \dot{z} &= xy - \beta z,\end{aligned} \tag{5.22}$$

where σ, ρ and β are system parameters. For our example, we choose $(\sigma, \rho, \beta) = (10, 28, \frac{8}{3})$.

For our RNN, we choose an architecture of one hidden layer of twenty LSTM cells, for a total of 1903 trainable parameters. The training set contains eight batches of 32 sequences, each of which are length 2000, initialized randomly. We train the RNN to estimate the states of this system for 150 epochs using the Adam optimizer, with a learning rate of 0.001. The values of the hyperparameters were chosen through trial-and-error.

Our RNN-based estimator gave a mean squared error of 3.492 on the test set, while the EKF gave a mean squared error of 24.480. For the Lorenz system, our RNN-based estimator performed 85.73% better than the EKF. The execution time of our RNN-based estimator was also faster, taking 48 milliseconds compared to 221 milliseconds for the EKF. The training time for the RNN was 50 seconds.

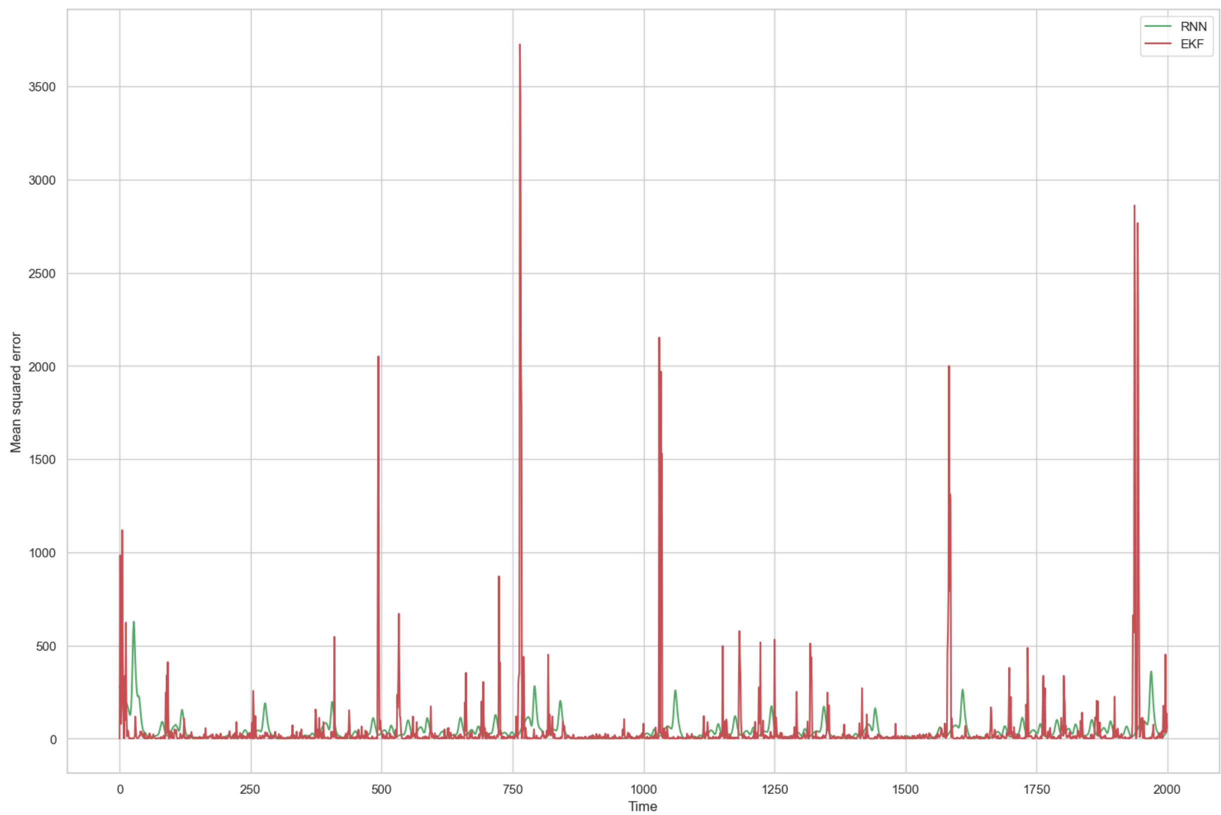

Figure 5.9: Mean squared errors for the RNN-based estimator (green) and Extended Kalman filter (red) for the 2000 time steps contained in each sequence in the test set.

Using the same methodology as was used for the preceding linear systems, we were able to train a RNN to estimate the states of this non-linear system as well, with an accuracy significantly better than an Extended Kalman filter.

Chapter 6

Conclusion

Through our examples, we have demonstrated that RNNs are able to estimate the states of noisy linear dynamical systems. We have found several benefits to the RNN-based estimator. Primarily, we were able to train a RNN to achieve equivalent performance in comparison to a Kalman filter. Another advantage which the RNN-based estimator possesses is it does not require prior knowledge of the governing equations of the dynamical system. Unlike the Kalman filter, the purely data-driven nature of our RNN-based estimator allow its use in cases where the dynamical system has not been fully identified.

A major disadvantage of the RNN-based estimator is the amount of work required before it can be used. While a Kalman filter can be used immediately after implementation, a RNN-based estimator also has to be trained, requiring us to 1) gather a large amount of high-quality data, 2) run the training procedure for a number of iterations, 3) optimize the hyperparameters of the recurrent neural network, and 4) verify the generalization

of the trained model. Not only are these steps time and labor intensive, the curse of dimensionality increases the difficulty of these tasks considerably for high-dimensional data, though techniques such as transfer learning can also make them easier. Combined with the stochastic nature of gradient-based optimization, it is not guaranteed that an optimal solution can be found. One the other hand, the Kalman filter is a minimum mean squared error estimator where its expected degree of performance is known. A significant amount of labor and computational resources must be dedicated in order to produce a RNN-based estimator, so this must also be considered prior to deciding which method to use.

In the end, there is no clear answer with regard to which of the two estimation methods is superior. Instead, this comes down to situation and user preference. We can only conclude that RNN-based estimators are viable for state estimation of noisy linear dynamical systems. They have several benefits over a Kalman filter, and may warrant consideration as a alternative to Kalman filtering in a variety of applications.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Kirsten A. Morris. *Introduction to Feedback Control*. Harcourt-Brace, 2000.

[3] O. A. Stepanove. Kalman filtering: Past and present. an outlook from Russia. (on the occasion of the 80th birthday of Rudolf Emil Kalman). *Gyroscopy and Navigation*, 2(2):105, 2011.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[5] Dan Simon. *Optimal state estimation: Kalman, H Infinity, and nonlinear approaches.* John Wiley and Sons, Inc., 2006.

[6] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[7] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[8] Webb Miller. Computational complexity and numerical stability. *SIAM Journal on Computing*, 4(2):97–107, 1975.

[9] C Guardiola, B Pla, D Blanco-Rodriguez, and L Eriksson. A computationally efficient kalman filter based estimator for updating look-up tables applied to nox estimation in diesel engines. *Control engineering practice*, 21(11):1455–1468, 2013.

[10] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J van der Walt, Matthew Brett, Joshua Wilson, K Jarrod Millman, Nikolay Mayorov, Andrew R J Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E A Quintero, Charles R Harris, Anne M Archibald, Antônio H Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[11] Piotr Kaniewski. Extended kalman filter with reduced computational demands for systems with non-linear measurement models. *Sensors*, 20(6):1584–, 2020.

[12] David Sussillo. Neural circuits as computational dynamical systems. *Current opinion in neurobiology*, 25:156–163, 2014.

[13] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[14] Tsungnan Lin, B.G. Horne, P. Tino, and C.L. Giles. Learning long-term dependencies in narx recurrent neural networks. *IEEE transactions on neural networks*, 7(6):1329–1338, 1996.

[15] Michael C. Mozer. Induction of multiscale temporal structure. *Advances in neural information processing systems*, 4, 1991.

[16] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. *Advances in neural information processing systems*, 8, 1995.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] Alex Graves. Supervised sequence labelling with recurrent neural networks. *Studies in Computational Intelligence*, 2012.

[19] Lee A. Feldkamp, Danil V. Prokhorov, and Timothy M. Feldkamp. Simple and conditioned adaptive behavior from Kalman filter trained recurrent networks. *Neural Networks*, 16(5-6):683–689, 2003.

[20] Bobby Kleinberg, Yuanzhi Li, and Yang Yuan. An alternative view: When does SGD escape local minima? In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2698–2707. PMLR, 2018.

[21] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David

McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[22] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[23] T. Tieleman and G. Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. In *Neural Networks for Machine Learning*, volume 6.5, pages 26–31. Coursera, 2012.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[25] G. V. Trunk. A problem of dimensionality: A simple example. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):306–307, 1979.

[26] Michael Green and John B. Moore. Persistence of excitation in linear systems. In *1985 American Control Conference*, pages 412–417, 1985.

[27] Jonas Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference: Novosibirsk, July 1–7, 1974*, pages 400–404. Springer, 1975.

[28] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455, 1998.

[29] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[30] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[31] J.P. DeCruyenaere and H.M. Hafez. A comparison between Kalman filters and recurrent neural networks. In *IJCNN International Joint Conference on Neural Networks*, volume 4, pages 247–251, 1992.

[32] S. Kumar Chenna, Yogesh Kr. Jain, Himanshu Kapoor, Raju S. Bapi, N. Yadaiah, Atul Negi, V. Seshagiri Rao, and B. L. Deekshatulu. State estimation and tracking problems: A comparison between Kalman filter and recurrent neural networks. In Nikhil Ranjan Pal, Nik Kasabov, Rajani K. Mudi, Srimanta Pal, and Swapan Kumar Parui, editors, *Neural Information Processing*, pages 275–281. Springer, 2004.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Py-

Torch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[34] Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. In *Programming Massively Parallel Processors (Fourth Edition)*, pages 123–147. Morgan Kaufmann, fourth edition edition, 2023.

[35] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[36] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[37] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[38] S. Deepak and P.M. Ameer. Brain tumor classification using deep CNN features via transfer learning. *Computers in Biology and Medicine*, 111:103345, 2019.

[39] Lixiong Wang, Hanjie Liu, Zhen Pan, Dian Fan, Ciming Zhou, and Zhigang Wang. Long short-term memory neural network with transfer learning and ensemble learning for remaining useful life prediction. *Sensors (Basel, Switzerland)*, 22(15):5744–, 2022.

[40] Catherine Rouvas-Nicolis and Gregoire Nicolis. Butterfly effect. *Scholarpedia*, 4(5):1720, 2009.