Evolutionary and Iterative Training of Recurrent Neural Networks via the Singular Value Decomposition

Politecnico
di Bari

Department of Electrical and Information Engineering
ELECTRICAL AND INFORMATION ENGINEERING
PH.D. PROGRAM
SSD: ING-INF/04 – AUTOMATICA

Final Dissertation

Evolutionary and Iterative Training of Recurrent Neural Networks via the Singular Value Decomposition

---

by

Daniel K. McNeill

Supervisor:

Prof. David Naso

Coordinator of Ph.D. Program:
Prof. Alfredo Grieco

Course n° 32, 01/11/2016–31/10/2019

Politecnico di Bari

LIBERATORIA PER L'ARCHIVIAZIONE DELLA TESI DI DOTTORATO

Al Magnifico Rettore
del Politecnico di Bari

Il sottoscritto Daniel Kyle McNeill nato a Ada (Oklahoma, USA) il 28/01/1982
residente a Barletta in via Nazareth 32 e-mail danmcne@gmail.com
iscritto al 3° anno di Corso di Dottorato di Ricerca in Ingegneria Elettrica e dell'Informazione ciclo XXXII
ed essendo stato ammesso a sostenere l'esame finale con la prevista discussione della tesi dal titolo:
"Evolutionary and Iterative Training of Recurrent Neural Networks via the Singular Value Decomposition"

DICHIARA

1) di essere consapevole che, ai sensi del D.P.R. n. 445 del 28.12.2000, le dichiarazioni mendaci, la falsità negli atti e l'uso di atti falsi sono puniti ai sensi del codice penale e delle Leggi speciali in materia, e che nel caso ricorressero dette ipotesi, decade fin dall'inizio e senza necessità di nessuna formalità dai benefici conseguenti al provvedimento emanato sulla base di tali dichiarazioni;
2) di essere iscritto al Corso di Dottorato di ricerca in Ingegneria Elettrica e dell'Informazione ciclo XXXII, corso attivato ai sensi del "*Regolamento dei Corsi di Dottorato di ricerca del Politecnico di Bari*", emanato con D.R. n.286 del 01.07.2013;
3) di essere pienamente a conoscenza delle disposizioni contenute nel predetto Regolamento in merito alla procedura di deposito, pubblicazione e autoarchiviazione della tesi di dottorato nell'Archivio Istituzionale ad accesso aperto alla letteratura scientifica;
4) di essere consapevole che attraverso l'autoarchiviazione delle tesi nell'Archivio Istituzionale ad accesso aperto alla letteratura scientifica del Politecnico di Bari (IRIS-POLIBA), l'Ateneo archivierà e renderà consultabile in rete (nel rispetto della Policy di Ateneo di cui al D.R. 642 del 13.11.2015) il testo completo della tesi di dottorato, fatta salva la possibilità di sottoscrizione di apposite licenze per le relative condizioni di utilizzo (di cui al sito http://www.creativecommons.it/Licenze), e fatte salve, altresì, le eventuali esigenze di "embargo", legate a strette considerazioni sulla tutelabilità e sfruttamento industriale/commerciale dei contenuti della tesi, da rappresentarsi mediante compilazione e sottoscrizione del modulo in calce (Richiesta di embargo);
5) che la tesi da depositare in IRIS-POLIBA, in formato digitale (PDF/A) sarà del tutto identica a quelle **consegnate**/inviate/da inviarsi ai componenti della commissione per l'esame finale e a qualsiasi altra copia depositata presso gli Uffici del Politecnico di Bari in forma cartacea o digitale, ovvero a quella da discutere in sede di esame finale, a quella da depositare, a cura dell'Ateneo, presso le Biblioteche Nazionali Centrali di Roma e Firenze e presso tutti gli Uffici competenti per legge al momento del deposito stesso, e che di conseguenza va esclusa qualsiasi responsabilità del Politecnico di Bari per quanto riguarda eventuali errori, imprecisioni o omissioni nei contenuti della tesi;
6) che il contenuto e l'organizzazione della tesi è opera originale realizzata dal sottoscritto e non compromette in alcun modo i diritti di terzi, ivi compresi quelli relativi alla sicurezza dei dati personali; che pertanto il Politecnico di Bari ed i suoi funzionari sono in ogni caso esenti da responsabilità di qualsivoglia natura: civile, amministrativa e penale e saranno dal sottoscritto tenuti indenni da qualsiasi richiesta o rivendicazione da parte di terzi;
7) che il contenuto della tesi non infrange in alcun modo il diritto d'Autore né gli obblighi connessi alla salvaguardia di diritti morali od economici di altri autori o di altri aventi diritto, sia per testi, immagini, foto, tabelle, o altre parti di cui la tesi è composta.

Luogo e data Barletta 4/1/2021 Firma Daniel Kyle McNeill

Il/La sottoscritto, con l'autoarchiviazione della propria tesi di dottorato nell'Archivio Istituzionale ad accesso aperto del Politecnico di Bari (POLIBA-IRIS), pur mantenendo su di essa tutti i diritti d'autore, morali ed economici, ai sensi della normativa vigente (Legge 633/1941 e ss.mm.ii.),

CONCEDE

- al Politecnico di Bari il permesso di trasferire l'opera su qualsiasi supporto e di convertirla in qualsiasi formato al fine di una corretta conservazione nel tempo. Il Politecnico di Bari garantisce che non verrà effettuata alcuna modifica al contenuto e alla struttura dell'opera.
- al Politecnico di Bari la possibilità di riprodurre l'opera in più di una copia per fini di sicurezza, back-up e conservazione.

Luogo e data Barletta 4/1/2021 Firma Daniel Kyle McNeill

Politecnico di Bari

RICHIESTA DI EMBARGO

Sottoscrivere solo nel caso in cui si intenda auto-archiviare la tesi di dottorato nell'Archivio Istituzionale ad accesso aperto alla letteratura scientifica POLIBA-IRIS (https://iris.poliba.it)) non in modalità "Accesso Aperto", per motivi di segretezza e/o di proprietà dei risultati e/o informazioni sensibili o sussistano motivi di segretezza e/o di proprietà dei risultati e informazioni di Enti esterni o Aziende private che hanno partecipato alla realizzazione della ricerca.

Il/la sottoscritto/a_____nato/a_____

il_____ residente_____ alla via_____ indirizzo e-mail_____

_____ iscritto/a al corso di dottorato di ricerca_____ ciclo_____

Autore della tesi di dottorato dal titolo_____

e ammesso a sostenere l'esame finale:

NON AUTORIZZA

Il Politecnico di Bari a pubblicare nell'Archivio Istituzionale di Ateneo ad accesso aperto il testo completo della tesi depositata per un periodo comunque non superiore a 12 (dodici) mesi decorrenti dalla data di esame finale.

Specificare la motivazione *(apporre una crocetta sulla motivazione):*

☐ Brevetto
*(indicare nel campo libero la data della domanda di deposito)*_____

☐ Segreto industriale, se è stato firmato un accordo di non divulgazione.
*(indicare nel campo libero gli estremi dell'accordo)*_____

☐ Segreto d'ufficio a tutela di progetti _____

☐ Motivi di priorità nella ricerca (previo accordo con terze parti) _____

☐ Motivi editoriali _____

☐ Altro (*specificare*) _____

Saranno comunque consultabili ad accesso aperto i dati bibliografici e l'abstract.

Il sottoscritto dottorando dichiara in virtù di quanto sopra che si rende opportuno procrastinare la pubblicazione della tesi attraverso l'Archivio Istituzionale ad accesso aperto (POLIBA -IRIS) e di impostare la data di embargo, in fase di deposito della tesi di dottorato in formato elettronico (pdf/A) per un periodo *di embargo* non superiore a 12 (dodici) mesi decorrenti dalla data di esame finale.

Il sottoscritto dottorando dichiara di essere a conoscenza che a scadenza della data di embargo su riportata, la tesi verrà pubblicata attraverso l'Archivio Istituzionale ad accesso aperto alla letteratura scientifica del Politecnico di Bari.

Luogo e data _____ Firma Dottorando _____

Firma Relatore _____

Politecnico di Bari

Department of Electrical and Information Engineering
ELECTRICAL AND INFORMATION ENGINEERING
PH.D. PROGRAM
SSD: ING-INF/04 – AUTOMATICA

Final Dissertation

Evolutionary and Iterative Training of Recurrent Neural Networks via the Singular Value Decomposition

---

by

Daniel K. McNeill

Referees:

Prof. Alberto Cavallo
Prof. Andrea Gasparri
Prof. Antonio Visioli

Supervisor:

Prof. David Naso

Vice Coordinator of Ph.D. Program:
Prof. Alfredo Grieco ENRICO DE TUGLIE

Course n° 32, 01/11/2016–31/10/2019

Abstract

This work examines the use of the singular value decomposition (SVD) from linear algebra as a tool for the analysis of neural networks, as well as its use to speed up or even limit learning (to prevent over-fitting or maintain stability, for example) and as the basis for iterative and evolutionary learning algorithms.

In the process of describing our methods, we wish the reader to keep one main idea in mind: the layers of neural networks, whether recurrent or feedforward, are transformations from one space to another and generally this transformation is an affine transformation composed with a nonlinear transfer function (whose main purpose is to limit the range of the transformation). The affine part of this transformation is simply multiplication by a matrix and addition of a bias vector — and these two elements are the focus of neural network learning – that is, finding the appropriate values for the entries of the matrix and the bias vector.

If one thinks of the simplest possible learning algorithms for adjusting the coefficients of the transformation matrix — perhaps random sampling, retaining the fittest, random sampling again in a smaller neighborhood around the fittest candidate — these methods forget all the structure of the transformation, that is, they treat the matrix as a simple vector.

What we present here are methods of taking the inherent structure of the transformation into account — even while using evolutionary methods — using the singular value decomposition. Of course, preserving some structure of the transforma-

tions is not completely new — whether this means preserving sparseness or some type of invariance, as in the shift invariance of a convolutional layer.

However, we believe this is one of the first works to try to preserve not some extra or imposed structure of neural network layers during the learning process (as in convolutional networks) — but to try to use the structure inherent in the affine transformation at the core of a neural network work to facilitate learning, at least when considering evolutionary algorithms.

We describe several methods for using the SVD in neural network training, parameter reduction and modification — some of these, in particular the evolutionary method, the application to recurrent neural networks, and the pre-training of networks (either on a similar problem or with domain knowledge) and then retraining of the singular values to speed up training are new techniques with this work (and the author's paper [58]). In supervised learning, our methods may begin with linear (or non-linear) least squares models — which we then elaborate and refine with various methods. The first method of training being a simple iterative method based on the decomposition of a matrix into the sum of rank one matrices. Others being a "compact" evolutionary algorithm similar to that described by Mininno, Cupertino & Naso in [59], and an evolutionary algorithm based on "crossing" the singular value decompositions of two matrices developed herein.

Moving to unsupervised learning, we can apply essentially all of the same learning techniques — the main difference being that of discovering a reasonably good initial starting point. However, this can be done by either using the iterative method for the first round of learning (decreasing the dimension of the search spaces), or, if some domain knowledge is available (as in the virtual car control example), some of this knowledge can be "pre-programmed" into the matrix of connections for the controller. Here we even combine these two techniques — beginning an iterative search through the individual singular vectors using the singular vectors of a "pre-

programmed" matrix as initial starting points, just as one would do when beginning with a least squares starting model.

After training for a particular task, we consider how one might use essentially the same neural network for a similar problem — but retrain or fine-tune the network by modifying only the singular values (of which there are n rather than n^2 in our examples). The examples we give are using a price prediction model originally trained for Bitcoin and retuned for Ethereum and fine-tuning a virtual racing car for different types of tracks. We expected both of these models to exhibit similarities between the different cases (with the cryptocurrency models being quite choatic and the driving models being quite continuous), but enough variation that the models could show improvement through retuning of the singular values.

A related application considers how one might use the same decomposition to efficiently store any neural network, throwing away the smaller singular values (and their associated singular vectors), allowing a small amount of performance degradation — but helping to generalize, preventing over-fitting, and then fine tune the network again as needed — this time by modifying not just the singular values, but also the singular vectors associated to the smaller singular values. In fact, one may do this in such a way to maintain the same regions of stability of the original network — so long as the sum of the singular values one changes does not exceed 1 (assuming the transfer function is a contraction).

The final result of the work is a general algorithm for learning models for prediction, system identification and control — where those models are of the form:

$$\mathbf{m}(t+1) = \varphi(A\mathbf{m}(t) + \mathbf{b}), \tag{1}$$

where A is a matrix, $\mathbf{m}$ is a vector of variables under consideration, including input variables, output variables, and hidden variables, $\mathbf{b}$ is a bias vector, and φ is a trans-

fer function (generally a non-linear contraction or a linear function with bounds). We note that such models, while simple, are computationally powerful in general.

The summarized steps of our learning method are:

1. Assuming a supervised learning setting, perform standard analyses of variables for selection and perform scaling/shifting/normalization of variables as necessary.

2. Create a least squares model for the outputs desired using the desired (or required) input variables. This may be a linear least squares solution or a non-linear model. The least squares solution will provide the first singular triple (the first singular vector pair and associated singular value), or, at least, a starting point for an iterative search for the first singular triple.

3. Enlarge the model with hidden variables (hidden neurons). This may be done one (or a few) neurons at a time until one achieves a model of sufficient accuracy, until gains in accuracy are no longer significant or until computing limits are reached. A practical upper limit that we have found, after which model accuracy does not seem to increase is a few more hidden variables than the the sum of the given numbers of input and output variables.

4. Apply the iterative procedure to obtain singular triples according to the dimension of the matrix A — similarly for the bias $\mathbf{b}$. Each iteration has a reduced dimension for the search space, speeding up the search as it goes — and each singular value has a reduced interval to search.

5. After a single pass of the iterative procedure, each of the singular triples has been optimized — *but only with respect to the preceding singular triples*. Were this not a recurrent network, were it simply a mapping from a particular set of input variables to a disjoint set of output variables, this may be sufficient. In the case of a recurrent neural network, it may not be.

6. To remedy Point 5, we have several options: run the iterative algorithm a few more times (preferably not by changing the matrix from the first round, but running the algorithm on a new matrix to be added to the first), apply a random search or apply a singular value decomposition evolutionary search as presented herein.

7. In the case of unsupervised learning (or in the case of improving a model first developed by supervised training), we can directly apply either the iterative method (which has the best early convergence if we have no idea for an initial model) or the singular value evolutionary methods (especially if some initial model can be obtained by expert domain knowledge or previous supervised learning).

8. A major benefit of our technique is this last step — allowing very controlled training in either a supervised or unsupervised situations. Here we maintain most of the information of our model by fixing the singular vectors (what we think of as the "qualitative" aspects of the model — as it encodes the direction of the output for a certain input) and allowing training of the singular values (what we think of as "quantitative" aspects of the model — as they encode the magnitude of an output response for a particular input) — allowing us to fine-tune the network for slightly different purposes by training n rather than n^2 parameters. We may, similarly, elect to allow only the last few singular vectors (those with the smallest singular values and smallest effects) to be fine-tuned as well. This conserves the "gross" behavior of the network and can allow one to limit the network response so that it remains within a particular region, for stability concerns, for example.

   Beyond fine-tuning, we also maintain direct control over the singular values — in the case where we may be concerned about maintaining the stability of

the model (or maintaining a particular region of stability), this direct control allows us to achieve this fairly easily.

These methods allow us to train recurrent neural networks for a variety of problems with changes through time, including price prediction, predictive maintenance and model identification, and automatic control. Our method does not rely on back propogation and can be used in either supervised or unsupervised settings. Further, our models can be easily initialized by using either domain knowledge or (linear) least squares to "pre-program" the model and begin optimization in an area of the solution space likely to yield results. Finally, given a neural network previously trained in one domain, our models and methods allow the reuse and quick retraining for a similar domain, by preserving the inherent structure of the transformation at the heart of the neural network.

Acknowledgements

First, I would like to thank Prof. David Naso and all of the people of the Robotics and PRINCE Labs for their help and guidance during the project. Also, thanks to Dora Tarantino and her parents Michele Tarantino and Domenica Campolongo for the years of support. The greatest thanks must go to my family, including my son, Phillip McNeill, for the motivation to complete this work and to my parents, Phillip and Jo McNeill, who, though far away now, certainly started me on the journey down the path I tread. I would like to thank many other friends, teachers and mentors along the way, including Kenny, Gail, Anita, Jack, and Enzo. My thanks also extends to my friends and colleagues at Exprivia, we've had many interesting discussions and you've been more welcoming than I could have hoped for. And, finally, I would like to thank those who read this work and helped to correct many errors of style and substance, though your contribution is somewhat hidden, you have made my life much better during this process. Saying that, any errors which do remain are my own — and I hope the reader can find something of interest in this work.

Contents

Chapter 1

Introduction

The singular value decomposition has been well known, and well used, in linear algebra for many years — both for the algebraic insights it provides about matrices (or linear operators on finite vector spaces) and for the fact that a proper implementation provides good numerical stability for various problems, finding a linear least squares solution, for example [39].

This work describes various uses of the singular value decomposition (SVD) in the context of neural networks. Some of these uses or applications are a direct application of well-known properties of the SVD in the realm of matrices or linear algebra, with simply an interpretation in the area of neural networks applied. Other properties of the SVD take on additional significance however.

The SVD provides an effective way to tune both the speed of learning as well as, and perhaps more importantly, a way to limit the searched solution space — which can have benefits for both search speed and stability of solutions, depending on our objectives.

We can limit the search space in several different ways:

- We may fix the first few summands of the SVD expansion and learn only the last few to "fine tune" a network — perfectly retaining the lower rank approximation and not allowing the method to drift into a larger (and perhaps unproductive or unstable) solution space.

- We may take the opposite route at first and attempt to find a low-rank approximation, considerably lowering the number of parameters to learn (thereby decreasing the dimension of the solution space under consideration). Depending on implementation, we may also lower computation costs (performing inner products of vectors rather than matrix multiplication).

- We may have a good solution for a similar problem (examples might be a controller for a different car, or a controller for a particular car on a different track). Then, rather than attempting to train an entirely new network (and then store it), we might assume that the fine-tuned controller will perform exactly the same actions, but perhaps with different intensities or weights. We can then attempt to train only the singular values of the network (reducing our parameters from n^2 to n). It may then even be reasonable to store these fine-tuned singular values for different situations.

As an example, in the first training of a network, we wish to train quickly, learning which inputs (and their relations) have the greatest immediate effect on the output. This is to say, we wish to train the first few singular vectors and their corresponding large singular values.

Given a network trained for a similar task, on the other hand, training using the singular value decomposition allows us to maintain the first few singular values/vectors (or change them only slightly), while focusing the majority of the training on the specifics which make this problem different, i.e. learning the less significant singular values/vectors.

This means that we can maintain a sort of basic 'memory' in the network, while still adapting — either learning the specifics of a different job, or adapting to new information as it becomes available — we simply restrict the magnitude of the singular values we are allowed to modify.

In particular, supposing we have obtained the larger singular values which result in a stable solution for a particular problem, we may wish to restrict the fine-tuning of the solution to the smaller singular values which don't force the solution outside the region of stability.

In what follows, we will discuss some preliminaries of the singular value decompostion, some background in time series and neural networks and consider previous work at the intersection of these fields — particularly in the area of our primary application of real-time control. We then examine our various experiments in the chosen application areas of price prediction, system identification and predictive maintenance, and real-time control, using the methods and models of learning developed herein.

We note here that the literature on each of these subjects is vast, but that ours is the first work to consider using the SVD in all the ways we mention as a tool to manage the learning of neural networks.

Chapter 2

Preliminaries

2.1 The Singular Value Decomposition

The singular value decomposition (SVD) has been a powerful tool in linear algebra for many decades (see [30, 39]), used, for example, to find least-squares solutions. For those who may not be familiar with it, we mention that it is in some ways similar to principle components analysis and we provide some basic results and equations we have used.

The SVD allows a matrix $M \in \mathbb{R}^{m\times n}$ to be decomposed in the following way:

$$M = U\Sigma V^T,$$

where $\Sigma \in \mathbb{R}^{m\times n}$ has non-zero entries only on the main diagonal, further, these values are non-negative and non-decreasing, while U and V (and V^T) are square orthogonal matrices (with columns which are orthonormal), $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$.

This decomposition allows for a simple view of a linear transformation as well as several algebraic and numerical insights. The first is that when we consider the map:

$$M : \mathbb{R}^n \to \mathbb{R}^m \text{ by } \mathbf{x} \mapsto M\mathbf{x},$$

we can see this as the composition of three simpler maps:

$$\mathbf{x} \mapsto U(\Sigma(V^T\mathbf{x})).$$

First, V^T, being an orthonormal matrix, simply performs a collection of rotations and/or reflections on $\mathbb{R}^n$. Next, Σ maps $\mathbb{R}^n$ to $\mathbb{R}^m$, but as it is a diagonal matrix, it does so in a very simple way, simply stretching or shrinking the unit basis vectors (some of which may go to zero). Finally, U performs another action similar to V^T; it is simply rotation and/or reflection in $\mathbb{R}^m$.

Let us consider $M \in \mathbb{R}^{n\times n}$, i.e. a square matrix, for the remainder of the work. In this case, we have that each of U, Σ and V^T are square matrices in $\mathbb{R}^{n\times n}$. Therefore, since U and V^T are orthogonal matrices (and full-rank), the rank of M corresponds to the rank of Σ.

The construction of Σ, however, with non-zero (and non-negative) entries only on the diagonal, means that the rank of Σ corresponds exactly to the number of non-zero entries on the diagonal (or the dimension of the kernel/nullspace corresponds exactly to the number of zeros on the diagonal).

While this explanation of the decomposition is extremely useful and leads to many insights, further decomposition may be even more so. Given always a matrix $M \in \mathbb{R}^{n\times n}$, we can write

$$M = U\Sigma V^T = \sum_{i=1}^{n} s_i \mathbf{u}_i \mathbf{v}_i^T,$$

where s_i is the "i^{th} singular value," that is, the i^{th} entry on the diagonal of Σ, while $\mathbf{u}_i$ and $\mathbf{v}_i$ are the i^{th} left and right singular vectors respectively, i.e. the i^{th} columns of U and V.

This explicitly expresses M as the sum of outer products, or the sum of rank one matrices. Further, as the values s_i are decreasing, and the vectors $\mathbf{u}_i$ and $\mathbf{v}_i$

have norm one, each additional summand contributes less and less to the total. This suggests a method to approximate the matrix M, in the "best" possible way, using a matrix of lower rank — iteratively searching for each of the "singular triples" (right and left singular vectors and the associated singular value) successively.

To speak of finding a "best" approximation, we must define what we mean by "best" — of course we mean the nearest approximation with respect to a particular matrix norm. Two of the most used, and those which we will use throughout the paper are the L_2 norm and the Frobenius norm.

The L_2 norm is the induced matrix norm from the standard Euclidian norm of the vector $\mathbf{x}$:

$$||\mathbf{x}||_2 = \sqrt{\mathbf{x}^T\mathbf{x}}.$$

The L_2 norm of the matrix (or linear operator) M is then the maximum L_2 norm of the image of the unit ball:

$$||M||_2 = \max\{||Mx||_2 : ||x||_2 = 1\}.$$

A well-known result is that the L_2 norm of the matrix M corresponds to the first (largest) singular value in the singular value decomposition of M. That is, if

$$M = U\Sigma V^T \text{ and } \sigma_1 = \Sigma_{1,1} = \max\Sigma$$

then

$$||M||_2 = \sigma_1.$$

Minimizing the L_2 norm of a residual matrix then is precisely the same as minimzing the largest singular value of said matrix.

The Frobenius norm of a matrix A is defined to be the square root of the sum of the squares of the entries of the matrix:

$$||A||_{\mathrm{F}} = \sqrt{\sum_{i,j=1}^{n} a_{ij}^2}.$$

This matrix norm is often used as it represents, in some sense, the total error in a matrix approximation. And frequently the total error (or average error) can still be reduced even if the maximum error (which is given by the L_2 norm) cannot.

This becomes even more apparent when we consider the formulation of this norm using the singular value decomposition. Letting

$$A = U\Sigma V^T \text{ and } \sigma_i = \Sigma_{i,i},$$

we have that:

$$||A||_{\mathrm{F}} = \sqrt{\sum_{i=1}^{n} \sigma_i^2}.$$

That is, the Frobenius norm of A is simply the square root of the sum of the squares of the singular values.

Clearly then, reduction of *any* of the singular values will reduce the Frobenius norm of A — while the L_2 norm will remain the same so long as the largest singular value remains the same. However, the most efficient method to reduce the Frobenius norm of A is to reduce the largest term in the sum

$$\sqrt{\sum_{i=1}^{n} \sigma_i^2},$$

namely, the largest singular value, σ_1.

Given two matrices A, B, if we consider the L_2 norms of images:

$$||A\mathbf{x} - B\mathbf{x}||_2,$$

what we notice is that reduction of the induced L_2 norm between A and B will reduce the *maximum* error observed. While reduction of the Frobenius norm will reduce the *total* or *average* error.

Given what we have seen, relating these two norms and the singular value decomposition, we will always use the Frobenius norm in combination with the L_2 norm when possible — always attempting to reduce the Frobenius norm, by reduction of the L_2 norm when possible.

We note how the SVD decomposes a matrix — a linear transformation — into elements which show us the "most important" actions of that linear transformation — both in "quantitative" terms (changing the singular values changes the norm of the matrix/transformation) and in "qualitative" terms (changing the singular vectors changes the directions of the actions of the transformation). We hope to use what the SVD gives us in the way of control of a transformation and apply it to learning for neural networks in what follows.

Chapter 3

Background

It is well-known that recurrent neural networks are useful for learning sequences or analyzing time-series data [23, 88, 25, 78, 38, 54]. Since price prediction over time and control of simulated vehicles are our intended applications, we have therefore chosen to use this type of network.

There have been many recent advances in recurrent neural networks (see [73]), but many, if not most, of these developments have be based on structural or model changes. It is also well-known that recurrent neural networks are difficult to train, especially via gradient methods, i.e. backpropagation through time, due to the exploding/vanishing gradient problem [67, 87]. There are various solutions to this, [67] suggests a method of simply limiting the magnitude of the gradient, and therefore the speed of learning, for example. We mitigate this problem with the use of iterative and evolutionary training algorithms based on the singular value decomposition (SVD) of a matrix as we will see in the rest of this work, however, we must realize that the exploding/vanishing gradient problem is inherent in the learning of recurrent neural networks, and that any mitigation is exactly that and not a total avoidance of the problem.

To accelerate the learning using these methods, we use "pre-programming" or "pre-training" of our controllers using domain knowledge or least squares models derived from example data as a starting point. This general idea has been explored

before, as in [54], but not in the same manner as we discuss here, as TimeNet is a deep reccurent network meant to be a general time series classifier which can be used off-the-shelf. Our method simply adjusts the "quantitative" response of a network for a domain similar to that on which it was previously trained — or, when using a single layer, allows a domain expert to directly program the network, providing a base on which to begin learning.

3.1 Time Series

In the most general sense, we will be considering vector-valued nonlinear autoregressive exogenous models of time series. That is, models of the form

$$\mathbf{y}_t = F(\mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots, \mathbf{u}_t, \mathbf{u}_{t-1}, \ldots) + \varepsilon_t, \tag{3.1}$$

where $\mathbf{y}$ is the variable of interest, F is a nonlinear function, $\mathbf{u}$ is the externally defined variable and ε is the error of the model.

Typical models for time series prediction and forecasting are the SARIMAX models — or Seasonal, AutoRegressive, Integrated, Moving Average models with eXogeneous variables.

As our own models are recurrent non-linear modifications of the general SARIMAX model (with the addition of hidden variables as well), we will explain how each part of these models (and the whole) function in some detail.

Generally the first step in time series modelling is to attemp to remove any seasonality or trends in the series — making the result more "stationary." Removing seasonality and trends can be a sophisitcated topic — particularly for multi-year time series with monthly seasonality (especially since months are not the same length) or concerning movable holidays like Easter or Thanksgiving. This can lead to sophisticated Fourier analysis to remove seasonality and the use of carefully cho-

sen nonlinear functions to capture trends — and the combination of the two, particularly in data where the variation tends to be a percentage of the value.

We will however, stick to relatively simple models to reduce trend and seasonality in this work. For data which tends to have variation as a percentage of the value, we will use the logarithmic transformation of the data. For data which has a clear trend, we will use differencing methods to reduce this. In fact, this is the "integration" component of the SARIMAX model — the use of first, second or higher order differences (i.e. numerical derivatives) of the time series to obtain a more stationary series.

An autoregressive (AR) model for a time series Y_t is a model which uses a linear combination of the previous time series values to predict the next value. For this to function, the time series needs to show correlation from one value to the next (even if that is negative correlation). Assuming a positive correlation, the simplest form of AR model is simply to take the current value as an estimate for the next value(s). That is, given a time series Y_t, we model the series as:

$$Y_{t+1} \approx Y_t.$$

If Y_t has some trend, we may have the slightly more complicated model:

$$Y_{t+1} \approx \alpha Y_t + \mu,$$

where μ captures either an upward or downward trend depending on the sign. As we have said, we will generally assume that any seasonality or trend has been removed from our series, so we can also write this as:

$$Y_{t+1} - \mu \approx \alpha Y_t.$$

The last component to introduce is a "white noise" (or error) term, ε, so that we have:

$$Y_{t+1} = \alpha Y_t + \mu + \varepsilon_{t+1}.$$

Finally, when Y_t has high (partial) correlations with terms further removed from just the previous term, we have:

$$Y_{t+1} = \mu + \sum_{i=0}^{n} \alpha_i Y_{t-i} + \varepsilon_{t+1}. \tag{3.2}$$

The coefficients, α_i may be solved for in many ways, including least squares — where, essentially, the goal is to reduce the influence of the noise term, i.e. to have it come from a distribution with as little variance as possible.

We note that the model:

$$Y_{t+1} \approx \mu + \sum_{i=0}^{n} \alpha_i Y_{t-i},$$

without the white noise term, is a deterministic model, providing a precise prediction for the next value of Y. Meanwhile,

$$Y_{t+1} = \mu + \sum_{i=0}^{n} \alpha_i Y_{t-i} + \varepsilon_{t+1},$$

where ε_{t+1} is a random value chosen from a probability distribution (generally a distribution determined by the errors of the deterministic model) provides a stochastic model. Applying this model several times, with different values of ε_{t+1} chosen from our probability distribution provides a range of possible values for the next step in our time series — allowing us to provide estimates of confidence intervals and other statistics.

A moving average (MA) model, on the other hand, uses previous "white noise" terms to model the current value. In the simplest form we have:

$$Y_{t+1} = \mu + \varepsilon_{t+1},$$

where, again, μ is a trend term and ε_{t+1} is a "white noise" term — a random value chosen from a distribution, generally determined by the errors when modelling simply with μ.

A more sophisticated MA model uses a linear combination of several previous noise terms:

$$Y_{t+1} = \mu + \sum_{i=0}^{n} \alpha_i \varepsilon_{t-i} + \varepsilon_{t+1}. \tag{3.3}$$

This seems to have a very similar form to the autoregressive model of before, however, whereas in the training stage of developing the autoregressive model the time series values are known and one can perform least-squares optimization for the coefficients over a range of values for t, this model requires non-linear optimization methods to find appropriate values for the coefficients.

The other great difference between the two is that while the "white noise" term in the moving average model explictly has effects on a finite number of the successive terms, the "white noise" term of the autoregressive model is incorporated completely into the next term, and its effects may even be compounded in successive terms. This is important to note as we are not always concerned with finding exact forecasts or predictions, but ranges — or even investigations of stability, whether this is interpreted to mean "stays relatively near some mean value," "does not suffer oscillations," or, quite simply "does not explode."

An exogeneous variable for time series modelling and forecasting is simply any related variable which is not the variable being modelled — but the values of which the modelled variable may depend upon. If one were modelling the price of a particular stock, for example, prices of related stocks, market activity as a whole or

knowledge about happenings in that sector could be useful. That is, we think of the variable in question as being dependent on the values of some other variable or variables.

Given a time series Y_t and exogeneous variables (related, but different time series) $\mathbf{X}_t$, and a function μ_t capturing trend and seasonality SARMAX models (without the "integration") are simply linear models of the form:

$$Y_{t+1} = \mu_t + \sum_{i=0}^{n} \alpha_i Y_{t-i} + \sum_{j=0}^{m} \beta_j \varepsilon_{t-j} + \sum_{k=0}^{p} \boldsymbol{\gamma}_k^T \mathbf{X}_{t-k} + \varepsilon_{t+1}. \tag{3.4}$$

When one uses an "integrated" model, rather than applying the ARMA model to the original time series Y_t, one instead applies it to the "differences" (i.e. derivatives) of first, second, or some higher order. This is generally done until the difference term seems to be a stationary series — allowing its distribution to be used as the "white noise" term.

Our modelling, prediction, and control models, are in some ways a slight generalization of what we have seen before in the SARIMAX model and in some ways a simplification. In general, for example, we concern ourselves not with a real-valued time series Y_t, but with a vector-valued time series $\mathbf{y}_t$. Further, we will include hidden variables (hidden neurons) which serve to capture some of the dynamics of the system which may not be apparent or possible to capture with a simple least squares model.

In particular, we will generally use a model of the following form:

$$\begin{pmatrix} \mathbf{y}_{t+1} \\ \mathbf{h}_{t+1} \end{pmatrix} = \varphi(A(\mathbf{y}_t^T, \mathbf{x}_t^T, \boldsymbol{\varepsilon}_t^T, \mathbf{h}_t^T)^T + \mathbf{b}) + \boldsymbol{\varepsilon}_{t+1}. \tag{3.5}$$

Note $(\mathbf{y}_t^T, \mathbf{x}_t^T, \boldsymbol{\varepsilon}_t^T, \mathbf{h}_t^T)^T$ is simply another way to write the column vector:

$$\begin{pmatrix} \mathbf{y}_t \\ \mathbf{x}_t \\ \boldsymbol{\varepsilon}_t \\ \mathbf{h}_t \end{pmatrix}.$$

Here φ is a nonlinear "transfer" function (separating our model from the linear examples above), A is a matrix (i.e. a linear transformation), $\mathbf{x}$ represents external inputs (exogeneous variables), $\mathbf{h}$ represents hidden values (again, separating our model from the models above), $\mathbf{b}$ is a vector providing a shift (or "bias") and $\boldsymbol{\varepsilon}$ is a vector of "white noise" (which we may or may not use depending on whether we desire a range of predicted values or deterministic control).

In other words, we use a single layer recurrent network as our vector-valued nonlinear autoregressive moving average exogenous time series model. Generally, we will use only one previous step in our model — relying either on explicit computation of derivatives and the like given in $\mathbf{x}$ (i.e. as an "exogenous variable") or for effects through time to be absorbed in the hidden neurons $\mathbf{h}$.

3.2 Neural Networks: Evolution and SVD methods

While we will be exploring prediction of time series and real-time control in this work, the method by which we will mostly do so will be recurrent neural networks. McCulloch and Pitts [57] developed a computational model for neural networks based on mathematical algorithms in 1943. After Rosenblatt's [72] creation of the perceptron, neural networks enjoyed a short summer and a great deal of research.

Research in neural networks slowed considerably after Minsky and Papert [60] discovered two significant difficulties. One problem was that the exclusive-or circuit is impossible to process on single-layer (feedforward) neural networks. The second

was that computers simply weren't powerful enough to run, or worse, train, large neural networks.

Research in artificial neural networks slowed until great advances in computer processing power were achieved. In fact, in recent years, neural networks, especially in the guise of "deep-learning" with very deep networks, has had a massive resurgence [34]. However, other than a few simple architectural modifications (such as very deep networks), many of the advances can be attributed more to advances in processing power than to theoretical advances in neural networks.

Neural networks are used in many areas of machine learning, including prediction and control, image recognition and classification, image (re)creation, game play (e.g. chess, go and Atari), and many others [42, 29, 56, 76, 61]. However, while there have certainly been advances in neural network design and learning (particularly adversarial learning), many of the greatest leaps seem to come from also using the latest and most powerful hardware, GPUs and TPUs, for example [66, 82].

The basic learning algorithm for most neural networks is some kind of backpropagation, where gradients of the loss function are calculated and used to reduce errors for each layer of the neural network [37]. Even adversarial networks, which use two "adversarial" networks in competition in order to improve both, generally use backpropagation as their basic individual learning mechanism. Recurrent neural networks also often use a type of backpropagation, called "backpropagation through time" in which the network is "unrolled in time" (for some defined interval of time) and errors are backpropagated much the same as the usual way (but where each "layer" in the "unrolling" is a copy of the network in question) [83].

There are, however, limitations to the backpropagation learning mechanism. When speaking of feed-forward networks (those with only forward connections from inputs to outputs and no loops), backpropagation can optimize a particular chosen neural network architecture for a particular objective function — however, it cannot optimize the architecture itself. That is, it can optimize the weights of

the connections between the neurons between each layer, but it cannot optimize the number of layers, the size of the layers, the type of neurons (e.g. transfer function) in each layer, etc.

Each of these parameters is generally chosen heuristically by the machine learning practitioner — frequently with the practitioner trying several different combinations and honing in on the best one for their purposes. Some learning mechanisms, such as genetic or evolutionary algorithms, can be used to optimize these parameters automatically [52, 6, 41, 20].

Backpropagation also has difficulty training recurrent neural networks (RNNs) — that is, artificial neural networks with closed circuits in their connection network. This is due to the "exploding/vanishing gradient" problem and can also create difficulties with very deep networks [67, 33].

In fact, the general method of training RNNs is "backpropagation through time," where the RNN is "unrolled" into a deep feedforward network over several timesteps, with a certain structure to map the weights back to the smaller RNN. This is often effective, especially if the particular training problem is of the "vanishing gradient" kind. If the gradient "explodes" instead, this works less well. Further, even when it works, it presents yet another parameter of the machine learning method to optimize — that is, how far back in time should the network be "unrolled" [83].

Since recurrent networks can be difficult to train via backpropagation, and since they can have relatively few neurons and/or explicit layers and still be infinitely deep for all practical purposes, evolutionary or genetic training algorithms are sometimes used and have often been explored [21, 31, 9, 8, 44, 3, 74].

Many different kinds of genetic and evolutionary algorithms have been used in this context. Even stochastic gradient descent can be thought of as a kind of evolutionary algorithm in some of its simplest implementations. In fact, as shown in [55], a simple random search can sometimes be a competitive approach to reinforment learning.

Our method also leverages the singular value decomposition of a matrix in a new way, however, the SVD, a powerful tool, has been used in the training of neural networks by Abid, Fnaiech & Najim [1] and by Psichogios & Ungar [70] — specifically, in relation to pruning extraneous hidden neurons in feed-forward networks.

Similarly, Kanjilal, Dey & Banerjee [45] have developed reduced size feed-forward neural networks using the singular value decomposition and subset selection.

More recently, the work of Xue, Li & Gong [85] is similar in that it first trains a deep (feed forward) neural network, then uses the SVD to "compress" the network. Extracting the most important pathways and weights and allowing the elimation of some connections, thereby reducing the size (and cost) of the network. The resulting network, its architecture somewhat changed, can then be retrained (or have its weights refined) by another round of the usual backpropagation training.

Huynh & Won [40] have used the singular value decomposition for training single hidden layer feed-forward networks, and a recent paper of Fontenla-Romero, Pérez-Sánchez & Guijarro-Berdiñas [28] develops a non-iterative method for training single hidden layer feed-forward neural networks based on the singular value decomposition. We note that these works, while focusing on training single hidden layer neural networks using the singular value decomposition, are somewhat different from our work as they restrict themselves to feed-forward networks.

Jia [43] has considered training feed-forward neural networks with the constraint that the singular values of the connection matrix be bounded near 1, i.e. the connection matrix is nearly orthogonal, and achieved state-of-the-art results on various benchmarks. Essentially, this is a way of conditioning the matrix and forcing the exploding/vanishing gradient problem away. A more recent work, [89], uses the SVD as a method to stabilize gradients for deep neural networks. While not directly related to our work, the concept, similar to the idea of throwing away small singular

values, can be useful. Perhaps this is especially so in the case of recurrent neural networks, though this is not an idea which we have explored to date.

Cox [24] has considered the degredation of short-term memory from parameter compression in recurrent neural networks using the singular value decomposition. His work indicating that considerable compression can be achieved, without too much degration, and that this kind of compression could allow more use of such neural networks on low-power and resource-constrained devices.

Meanwhile, Teoh, Tan & Xiang [79] and Santos, Barreto & Medeiros [75] have applied the singular value decomposition to estimating the number of hidden neurons in a feed-forward network. Finally, Cai, *et al.*, in [16] consider using the SVD for fast learning in deep (feed-forward) neural networks.

A work quite close to our own, and apparently contemporaneous, is [7], which gives a good overview of the singular value decomposition and some ideas of how it might be used in multi-layered neural networks. Our networks will be single-layer recurrent networks, but the recurrence clearly has some of the same effects as multiple layers.

3.3 Application Areas

Now that we have given an overview of the methods we will use, and pointed to how our methods and models differ in some way from each of them, combining previous ideas in a novel way, we turn to the various applications we have tested our methods and model against.

Each of these is a very well-researched field, with a literature frankly too large to survey completely. We will mention some of the literature with the greatest overlap with our work — indicating that the models should be effective in the chosen domains as well as highlighting the difference with our work.

3.3.1 Price Prediction

Our first example comes from price prediction. Prediction of prices or other chaotic time series is a well-researched field, and neural networks, even recurrent neural networks, have been applied and have been shown to perform well [36, 53, 47, 46].

The work of Han, *et al.* ([36]) is of particular interest for us as it demonstrates the effectiveness of recurrent neural networks for these kinds of time series.

For our experiments, we use public data for the price of Bitcoin to develop various models, particularly of the ARIMA type and the type we have described — but without exogenous variables. After developing a model for Bitcoin price prediction which performs better than using the first lag (i.e. the previous day's price), we apply the same model to the Ethereum cryptocurrency. We then re-optimize the model, adjusting only the singular values, to obtain a model again better than using the previous day's price as a prediction.

3.3.2 System Modelling and Predictive Maintenance

Another area of interest is that of System Identification and Modelling and Predictive Maintenance.

Here we point to the study of Ho, *et al.* ([38]) which compare the effectiveness of ARIMA, feed-forward neural networks and recurrent neural network for time series prediction in the area of predicting mechanical failures. In their experiments, ARIMA and recurrent neural network models outperformed feed-forward models. We take our lead from them developing first ARIMAX models and then our single-layer recurrent neural network model.

In our example, we have no previous information on actual breakage of the machine (or even simulations of such breakage), but we develop a predictor for the process variables using our model based on the process variables of a previous time step as well as the setpoint and PID outputs. Our model allows us to estimate

the probability that the machine will go out of the normal operating range (for the particular current state) in the near future.

Here, again, we develop the model in a more or less agnogstic manner — without detailed domain knowledge of the physical processes or control systems utilized. Again, we obtain a model with errors of the same order of magnitude as the first lag model — but with significantly more time allowance, something necessary in the event that the machine needs to be shut down to avoid going outside its limits.

3.3.3 Self-driving cars and TORCS

Developing self-driving control systems with realistic actions can be a difficult task — and one which has received much study both in simulation and in real autonomous vehicles (see [81] and [32]). First, we focus on those works which have used the TORCS/SCRC [84, 50] system to have the most direct comparison to our work.

The Simulated Car Racing Championship (SCRC) has been held in 2007, 2009, 2010, 2011, 2012, 2013, and 2015 [51, 50].

Butz & Lönneker [15] have developed an effective TORCS/SCRC controller dubbed COBOSTAR. Butz, Linhardt & Lönneker [14] have subsequently developed this controller further.

Cardamone, Loiacono & Lanzi [17, 18] have applied on-line neuroevolution to the problem of developing a controller for TORCS/SCRC.

Muñoz, Gutierrez & Sanchis [63] have developed a "human-like" TORCS controller for the SCRC, as well as a controller created by imitation of hand-coded controllers and human play [62] and considered a multi-objective optimization of their controller via evolutionary algorithms [64].

Yee & Teo [86] have developed evolutionary spiking neural networks as controllers for the SCRC/TORCS.

Preuss, Quadflieg & Rudolph [69] have considered TORCS/SCRC sensor noise removal and multi-objective track selection for adapting the driving style of a controller.

Botta, Gautieri, Loiacono & Lanzi [10] have considered the problem of evolving the optimal racing line for a track in the TORCS simulator.

Athanasiadis, Galanopoulos & Tefas [4] have developed a progressive neural network and a training methodology for it for TORCS.

Quadflieg, Rudolph & Preuss [71] have examined the difficulty of optimizing parameters for a controller to perform well on many tracks — and the trade-offs inherent to that attempt.

Turning to autonomous vehicle control in general, the list of papers is too numerous to list in its entirety in this work. As our work is mostly concerned with methods of learning for recurrent neural networks, with control of the simulated car being simply an example for both supervised and unsupervised learning, we note a few which have considered neural network or similar controllers for some aspect of the vehicle.

Pérez, Milanés & Onieva [68] have considered cascade architectures applied to lateral control of real autonomous vehicles. Deep neural networks have been used by Li, Mei, Prokhorov & Tao [49] for structural prediction and lane detection in traffic scenes.

3.3.4 TORCS, SCRC and SnakeOil

Our example in the realm of automatic control focuses on using a Python implementation of a recurrent neural network as a controller for a car as in the Simulated Car Racing Championship (SCRC) [50] — which is built atop The Open Racing Car Simulator (TORCS) [84]. We use SnakeOil [26] as a client for SCRC to maintain focus on developing a controller for the car and to allow the use of Python.

TORCS/SCRC has a number of features that lend it to researching self-driving vehicles and video game AIs. First, TORCS is a quite realistic simulator with a sophisticated physics engine taking into account many aspects of driving such as collisions and traction under various conditions of wheel spin and acceleration. TORCS also provides many different tracks with varying lengths, road surfaces, changing altitudes (i.e. roads with a high grade), and various types and degrees or lengths of straight-aways and curves. We, however, will restrict ourselves to eight tracks here.

Two alpine tracks with road grade changes: Alpine 1 and Alpine 2

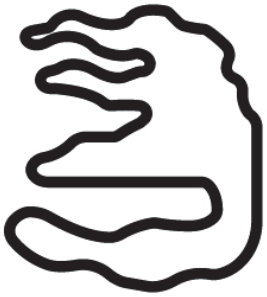

Figure 3.1: Alpine 1

Figure 3.2: Alpine 2

Two relatively fast tracks: Forza and Ruudskogen

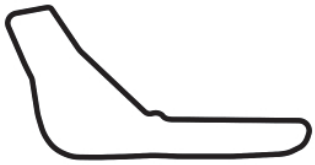

Figure 3.3: Forza

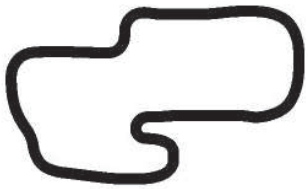

Figure 3.4: Ruudskogen

Two curvy tracks: Aalborg and Brondehach

Two relatively fast tracks with hard corners tracks: Wheel 1 and Wheel 2

These tracks represent a selection of most of the track types, except those with dirt surface, while not being extremely long. They are also some of the most com-

monly used for both the TORCS and SCRC competitions allowing some comparison to other controllers.

The SCRC modification of TORCS provides a collection of simple sensors for observing the local environment at small time steps (with the option that the sensors are noisy), a standardized racing car and a real-time client/server modularity — allowing one to develop the car controller without learning a great deal about the functionality of TORCS as well as forcing one to use only the given local sensor information and to develop a controller capable of operating in real-time.

SnakeOil simply implements the client in Python — allowing us to focus exclusively on developing the controllers in our programming language of choice.

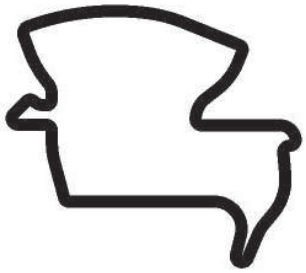

Figure 3.5: Aalborg

Figure 3.6: Brondehach

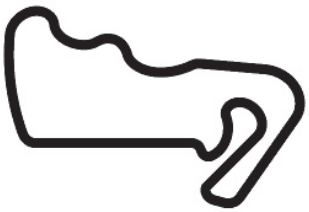

Figure 3.7: Wheel 1

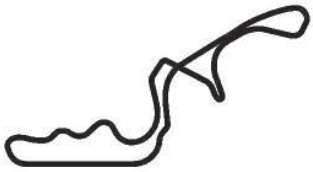

Figure 3.8: Wheel 2

Figure 3.9: Screenshot of TORCS

Chapter 4

Iterative SVD Approach

The simplest way to think of the iterative singular value approach is in comparison to coordinate descent method. The coordinate descent method is a simple way of minimizing a function by following along one coordinate until further movement does not decrease the function value, then switching to the next coordinate and doing the same. This very simple approach does not converge as quickly as gradient descent, but it has the advantage of not needing to compute derivatives or gradients — something which may be time-consuming or impossible in some cases.

However, the method is not always guaranteed to converge and can theoretically become "stuck" if, for example, the gradient is at a 45° angle to all coordinates and the curvature of the function is sufficiently tight. Small random perturbations are sufficient to prevent this from happening on well-behaved functions, however.

Were one to apply this method to minimize the norm of:

$$||f(\mathbf{x}) - \varphi(A\mathbf{x} + \mathbf{b})|| \tag{4.1}$$

where $A \in \mathbb{R}^{n \times n}$, this would entail incrementing each of the n^2 values of A in turn (as well as the n terms of $\mathbf{b}$), then looping back through this procedure until the minimum is found. For the singular value decomposition iterative method, rather than iterating through each of the parameters and optimizing them regardless of their

importance, the method attempts to optimize iteratively according to the importance of each coordinate — in fact, a major part of the method is finding the directions (or coordinate transformations) which are the most important.

Recalling that given a matrix M with singular value decomposition $U\Sigma V^T$, we note that this can be written also as the sum of the outer products of the singular vectors along with their associated singular value:

$$M = U\Sigma V^T = \sum_i s_i \mathbf{u}_i \mathbf{v}_i^T,$$

where s_i is the i^{th} singular value, while $\mathbf{u}_i$ and $\mathbf{v}_i$ are the i^{th} left and right singular vectors respectively.

As the singular values are ordered by their magnitude, each successive partial sum provides the best possible approximation to the matrix (using various matrix norms) of that rank.

This suggests a particular way to approach learning using the SVD — we wish to learn the "most important" parts of our connection matrix first.

Whether we work with an unsupervised or supervised training method, we will attempt to learn first a low rank (or even rank one) matrix which performs as well as possible before moving on to a higher rank matrix. We note here that we may actually use full rank matrices, but the learning will be focused on one particular singular vector pair and the associated singular value — what we may call a "singular triple."

However, we must also note that modifying a singular vector must necessarily change some of the following singular vectors. This means when working with full rank matrices, though we may intend to apply an iterative procedure, to each singular vector/triple in turn, this is not strictly the case. In fact, the impossibility of maintaining the same singular vectors when one changes is one reason to use the evolutionary precedures in Chapter 6.

If, on the other hand, one allows U and V to stray from the orthogonal, one can train each column iteratively — but one is no longer training singular vectors directly.

Depending on what information we have available, we can proceed as follows. First, we assume that the process $\mathbf{y}_t = P(\mathbf{x}_t)$, that we wish to model can be modelled as a vector-valued nonlinear autoregressive time series with exogenous variables as we saw before:

$$\mathbf{y}_t = \varphi(A\mathbf{x}_t + \mathbf{b}) + \varepsilon_t,$$

or, more completely, expanding out $\mathbf{x}_t$ into previous values of $\mathbf{y}$, exogenous variables ($\mathbf{u}$), previous errors/white noise ($\boldsymbol{\varepsilon}$), and hidden variables ($\mathbf{h}$), and concatenating the next values of the hidden variables with our intended output:

$$\begin{pmatrix} \mathbf{y}_t \\ \mathbf{h}_t \end{pmatrix} = \varphi\left(A \begin{pmatrix} \mathbf{y}_{t-1} \\ \mathbf{u}_t \\ \boldsymbol{\varepsilon}_{t-1} \\ \mathbf{h}_{t-1} \end{pmatrix} + \mathbf{b} \right) + \boldsymbol{\varepsilon}_t, \tag{4.2}$$

Let us first assume, here and throughout the work, that we have scaled both the values of the coordinates of $\mathbf{x}$ and the coordinates of $\mathbf{y}$. This is always possible, using the "tanh" function, for example, though we should remain cognizant of the range of feasible values before simply scaling in this manner — clearly better knowledge of the statistics of these values can lead to better scaling and better learning. We will generally try to apply scaling (and perhaps centering) which maintains separation of values within the possible (or normal) range, if possible.

If we are allowed to input chosen values into the function P, and to observe the resulting outputs, this is clearly the best possible position we can be in. Our first action in this case is to determine the image of the zero vector under P, i.e. $P(\mathbf{0})$, this allows us to have a better initial approximation for $\mathbf{b}$, which otherwise we

would initialize to near $\mathbf{0}$, under the assumption that the mean of the input values should produce an output near the mean of the output values (though centering our variables is necessary for this assumption to apply well).

Next we wish to find that unit vector $\mathbf{x}_1$ for which the output vector, $\mathbf{y} = P(\mathbf{x})$, has the greatest magnitude, suppose this is $\mathbf{y}_1$. Then we can construct a rank one approximation for A as, very simply, the outer product of $\mathbf{x}_1$ and $\mathbf{y}_1$, namely

$$A = \mathbf{y}_1 \mathbf{x}_1^T,$$

or, identifying $\mathbf{x}_1 = \mathbf{v}_1$ as $\mathbf{x}_1$ is a unit vector and letting $\mathbf{y}_1 = \sigma_1 \mathbf{u}_1$ where $\mathbf{u}_1$ is a unit vector and σ_1 is the magnitude of $\mathbf{y}_1$, we have:

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T.$$

So far, assuming we have a good sample of data, this process (except evaluation at $\mathbf{x} = \mathbf{0}$ to obtain a better initial value for $\mathbf{b}$) could be done with any supervised learning method we will consider. That is, for a supervised learning problem, we can always very quickly construct a rank one approximation — disregarding, however, the effects of hidden neurons and errors/white noise.

To obtain the best quality rank two approximation to our matrix A, we would like to sample the input space which is perpendicular (orthogonal) to the vector $\mathbf{x}_1$ which we used to produce the rank one approximation. Clearly, this is not possible in all situations — though we can consider finding an input vector "nearly" orthogonal to the first and which has the image vector with the greatest magnitude of those.

In general, one can continue this iterative process, and though it is impossible to guarantee that each iteration will produce the singular vectors associated to the next largest singular value, with some cleverness, one can make that more likely.

Assuming we are working with a linear (or bounded linear) transfer function, we may set our first approximate singular value somewhat above the limit of what it reasonably could be. Writing our approximation of s_1 as $\tilde{s}_1$, we start with $\tilde{s}_1 > s_1$. Using $\tilde{\mathbf{u}}_1$ and $\tilde{\mathbf{v}}_1$ as our approximations for the first singular vectors, then the only way to minimize

$$||A - \tilde{s}_1 \tilde{\mathbf{u}}_1 \tilde{\mathbf{v}}_1^T||_F$$

is to let $\tilde{\mathbf{u}}_1$ and $\tilde{\mathbf{v}}_1$ approach $\mathbf{u}_1$ and $\mathbf{v}_1$. On the other hand, if $\tilde{s}_1 = 1$, then the Frobenius norm above is minimized for any $\tilde{\mathbf{u}}_1$, $\tilde{\mathbf{v}}_1$ which form a singular vector pair with a corresponding singular value greater than or equal to 1. That is, we may not converge to a unique pair of singular vectors, in particular to the singular vectors corresponding to the largest singular value, which we desire — but may instead converge to any singular vector pair which will allow reduction of the Frobenius norm.

It is important to note that we first allow the singular vectors to converge as much as possible or necessary, after which we may decrease the approximated singular value $\tilde{s}_1$ — while doing so improves performance.

We next approximate s_2, $\mathbf{u}_2$ and $\mathbf{v}_2$. This time we may use $\tilde{s}_2 = \tilde{s}_1$ as the first approximation of s_2, as we can be sure that s_2 is less than or equal to s_1. We can also restrict our choices for $\tilde{\mathbf{u}}_2$ and $\tilde{\mathbf{v}}_2$ to be (nearly) orthogonal to $\tilde{\mathbf{u}}_1$ and $\tilde{\mathbf{v}}_1$ respectively.

Three things work in our favor here: most importantly, by selecting candidates $\tilde{\mathbf{u}}_2$ and $\tilde{\mathbf{v}}_2$ orthogonal to the first singular vectors, we have reduced the dimension of our problem. Second, we have a firm starting value for $\tilde{s}_2$, one which is often not far from the true value. Third, though we are doing less work for each iteration, each iteration also becomes less important for the outcome, we may choose to stop when we think the problem has been modelled well enough.

Since each iteration only gives us an approximation, it is important to allow our approximations for the next iteration to wander slightly out of the space orthogonal

to the space spanned by the previous singular vector approximations, clearly the amount we allow is closely related to the accuracy of the previous approximations. However, if we don't allow this, our approximations of each succesive singular vector pair will get progressively worse.

If desired, we can, of course, obtain a perfectly orthonormal basis by simply applying the SVD to the final result.

We present below several figures which illustrate the convergence of the iterative SVD method in various situations. We also note that these could be optimized more than we have done, and that we restricted ourselves to situations where a single run lasted less than a minute on four 2.7 GHz processors with 7 GB of RAM in Scilab on Ubuntu 18.04 LTS. It is interesting to note some differences in the convergence when using different transfer functions — however, this is directly related to the range of possible norms of the output.

4.1 Comparison to Stochastic Descent

One of the key elements of a learning algorithm is the comparison to a simple type of random search. We therefore decided to test our iterated learning method against a very simple implementation of stochastic descent in learning a test function of the form:

$$\mathbf{y}_t = \varphi\left(M\left(\mathbf{y}_{t-1}\right) + \mathbf{b}\right). \tag{4.3}$$

Where φ is the transfer function and M is the matrix we are trying to find or approximate. That is, we are in the best case scenario of trying to approximate functions of the exact same type as our models — trying to highlight the differences between the simple stochastic descent method and the iterated SVD method in learning not general functions, but learning within the class of functions of our models.

Our "random" method to test against here is a very simple implementation of stochastic descent which consists simply of testing two potential matrices, retaining the better matrix and then obtaining a new candidate by perturbing the retained candidate by an amount proportional to the difference in the norms of the previous candidate solutions, approximately in the direction of their difference — clearly this is much more efficient than simple random guessing and places some bounds on the next candidate solution.

Clearly, there is an inverse relationship between the speed of convergence and the amount to which the solution space is explored. We determined the learning rate for each of our experiments experimentally, settling on a value for each function which provided very good convergence for almost all examples.

The simple stochastic descent search sometimes performed better, but had a greater variability in results, as can be seen in tables and graphs from Table 4.1 (page 33) to Figure 4.24 (page 38). (In the figures and tables, "SD" refers to the "stochastic descent" method, while "ISVD" refers to our "iterative SVD" method.)

Here, both the function we wished to approximate and the models we used were of the form:

$$\mathbf{y}_t = \varphi\left(M\mathbf{y}_{t-1}\right), \tag{4.4}$$

where M was a random matrix in $[-1,1]^{25\times 25}$. Given an approximating matrix A, to measure nearness to our desired function, we used the Frobenius norm:

$$|\varphi(AY) - \varphi(MY)|_{\mathrm{F}}, \tag{4.5}$$

where $Y \in [-1,1]^{25\times 25}$ was a random matrix, i.e. a random sample of the domain space.

For each approximation attempt we allowed up to 10,000 iterations of the technique — stopping if the Frobenius norm dropped below 1. For each transfer func-

tion φ we ran 1000 trials to produce the statistics in Tables 4.1 to 4.8, meanwhile the figures show the decrease in the Frobenius norms for a selection of these trials.

Naturally, one method to attempt to take advantage of both types is to use a hybrid approach. As the rate of convergence tends to be greatest for the random search in the beginning, while the iterative SVD method accelerates toward the end (due to the reduced dimension of the search space) — this would seem to indicate that using the random method in the beginning and then switching to the iterative method as the random method levels off may be the best combination of the two.

By using a hybrid method, we can take a sample of the entire search space, and, by focusing on a region suggested by the sample, we can considerably reduce the search space for the first singular vectors — if not the dimension of the space.

Supposing our random search has given us a matrix A, with SVD

$$A = U\Sigma V^T,$$

we can focus our search for singular vectors on relatively narrow cones about the singular vectors of A (conditioned further by orthogonality constraints for subsequent singular vectors).

The algorithms used have been somewhat optimized for smooth transfer functions. However, an iterative SVD-like method is particularly well-suited to the step-

Final Norm Statistics
Identity Transfer Function

	SD	ISVD	Hybrid
min	0.872	1.00	0.925
Q1	0.967	1.80	0.978
Q2	0.988	2.16	0.992
Q3	0.995	2.65	0.998
max	1.00	4.01	1.15

Table 4.1: Quartiles

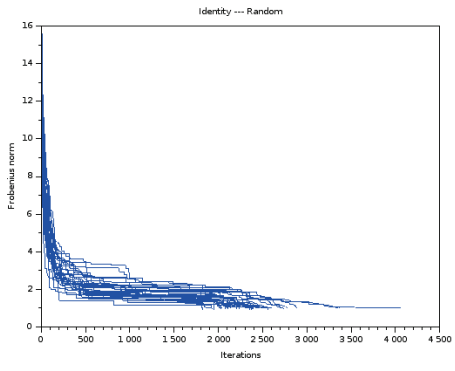

Figure 4.1: Convergence of SD

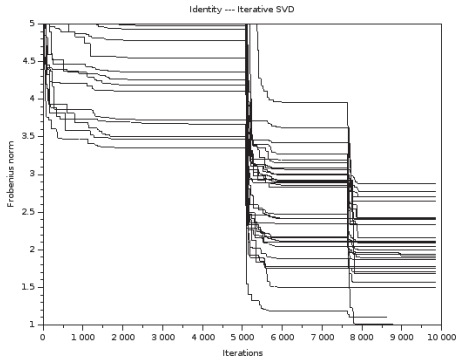

Figure 4.2: Convergence of ISVD

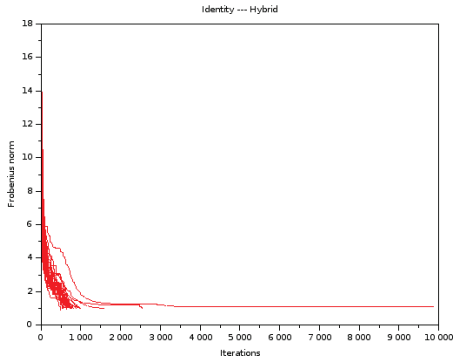

Figure 4.3: Convergence of Hybrid

Final Norm Statistics
Tanh Transfer Function

	SD	ISVD	Hybrid
min	0.834	1.00	0.760
Q1	0.953	1.01	0.947
Q2	0.976	1.02	0.969
Q3	0.992	1.05	0.990
max	2.43	1.42	1.24

Table 4.2: Quartiles

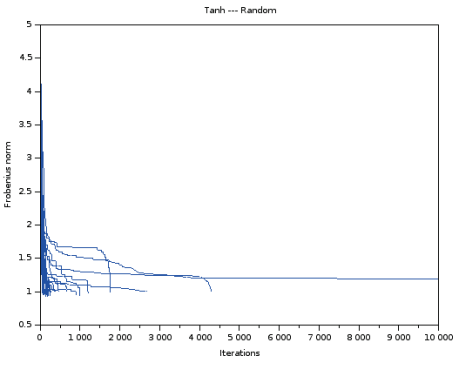

Figure 4.4: Convergence of SD

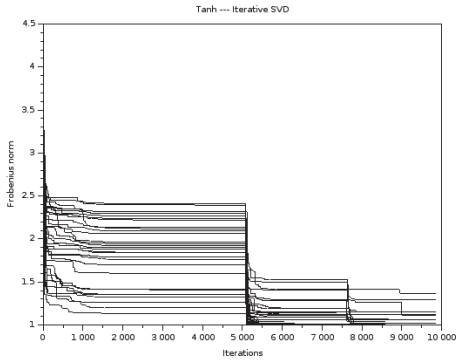

Figure 4.5: Convergence of ISVD

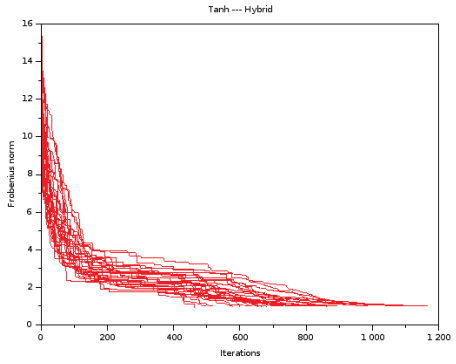

Figure 4.6: Convergence of Hybrid

Final Norm Statistics
Sigmoid Transfer Function

	SD	ISVD	Hybrid
min	0.869	0.538	0.860
Q1	0.957	0.670	0.952
Q2	0.971	0.736	0.974
Q3	0.986	0.780	0.989
max	1.06	0.868	1.00

Table 4.3: Quartiles

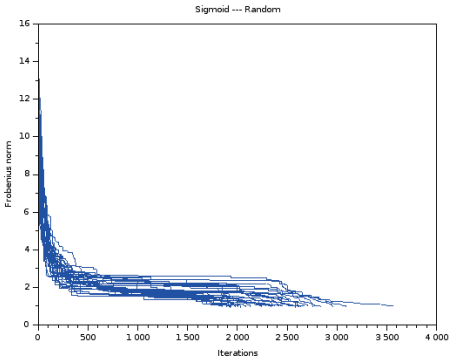

Figure 4.7: Convergence of SD

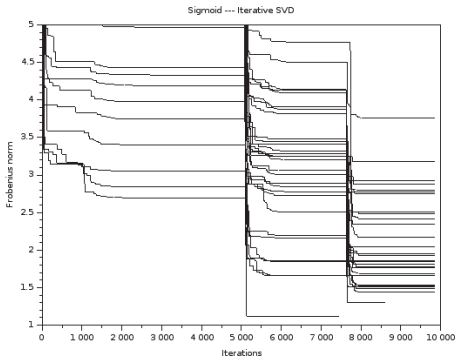

Figure 4.8: Convergence of ISVD

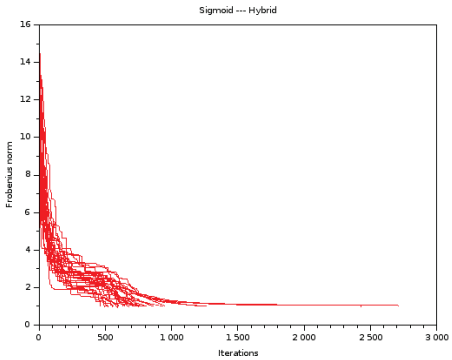

Figure 4.9: Convergence of Hybrid

Final Norm Statistics
ReLu Transfer Function

	SD	ISVD	Hybrid
min	0.828	0.847	0.736
Q1	0.966	1.03	0.954
Q2	0.999	1.26	0.988
Q3	2.05	1.73	1.36
max	4.82	3.29	3.77

Table 4.4: Quartiles

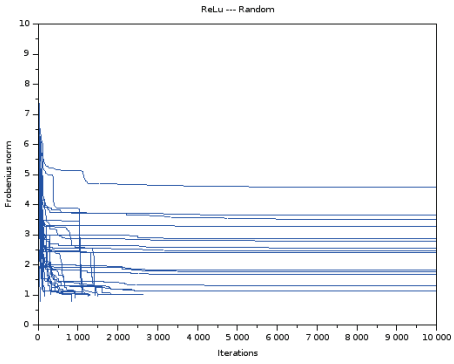

Figure 4.10: Convergence of SD

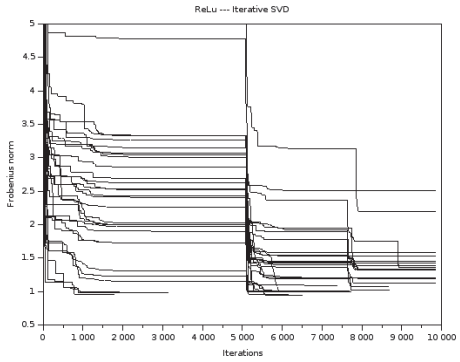

Figure 4.11: Convergence of ISVD

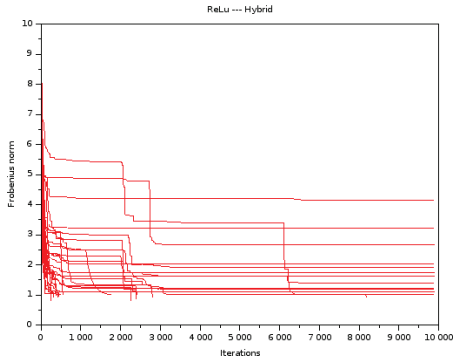

Figure 4.12: Convergence of Hybrid

Final Norm Statistics
Bounded Id Transfer Function

	SD	ISVD	Hybrid
min	0.825	1.00	0.650
Q1	0.964	1.01	0.960
Q2	0.994	1.05	0.988
Q3	1.38	1.17	1.03
max	3.73	1.79	2.97

Table 4.5: Quartiles

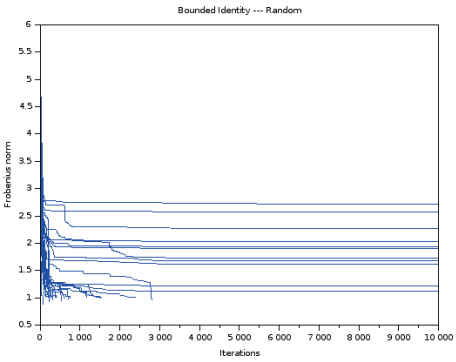

Figure 4.13: Convergence of SD

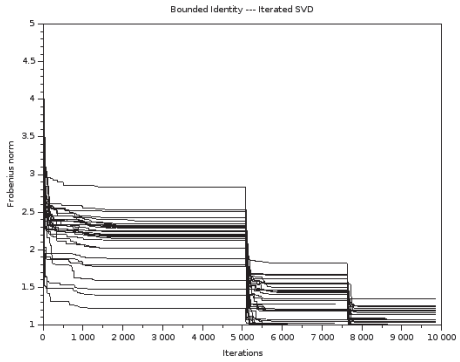

Figure 4.14: Convergence of ISVD

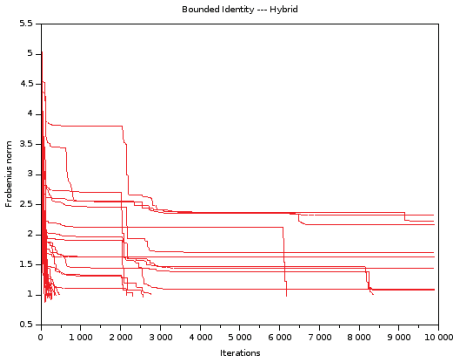

Figure 4.15: Convergence of Hybrid

Final Norm Statistics
Bounded ReLu Transfer Function

	SD	ISVD	Hybrid
min	0.811	0.522	0.439
Q1	0.970	0.885	0.906
Q2	0.997	0.982	0.956
Q3	1.29	1.01	0.994
max	2.24	1.45	1.71

Table 4.6: Quartiles

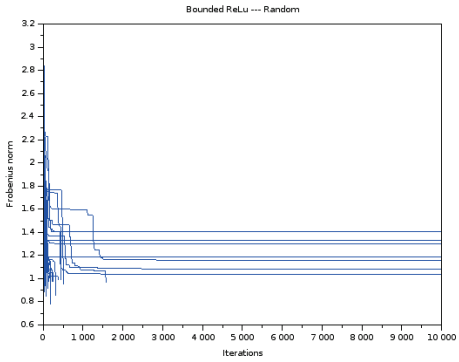

Figure 4.16: Convergence of SD

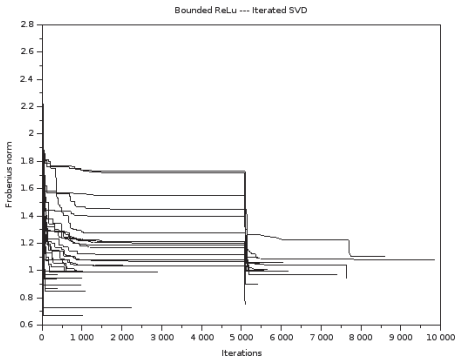

Figure 4.17: Convergence of ISVD

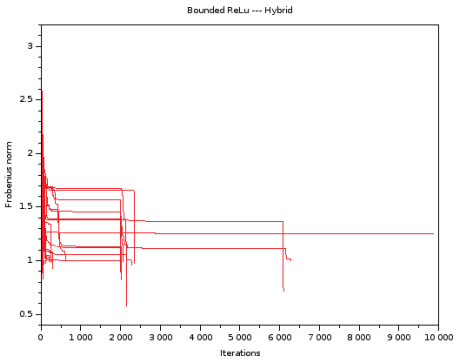

Figure 4.18: Convergence of Hybrid

Final Norm Statistics
Sign Transfer Function

	SD	ISVD	Hybrid
min	2.00	2.00	0.00
Q1	3.46	2.00	2.00
Q2	4.47	2.83	2.83
Q3	5.29	3.46	3.46
max	6.93	5.00	5.66

Table 4.7: Quartiles

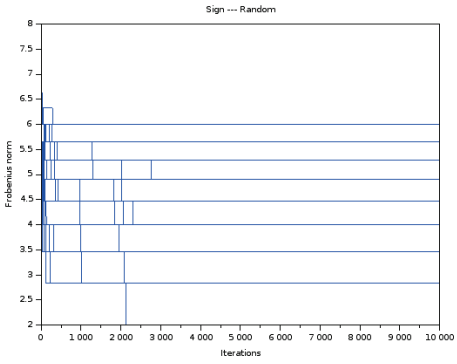

Figure 4.19: Convergence of SD

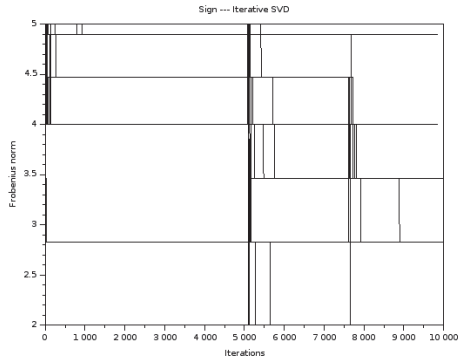

Figure 4.20: Convergence of ISVD

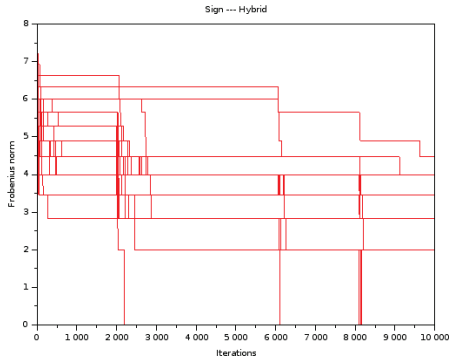

Figure 4.21: Convergence of Hybrid

Final Norm Statistics
0/1 Transfer Function

	SD	ISVD	Hybrid
min	1.00	1.41	1.00
Q1	1.73	1.73	1.41
Q2	2.24	2.00	1.73
Q3	2.65	2.24	2.00
max	3.32	2.65	2.45

Table 4.8: Quartiles

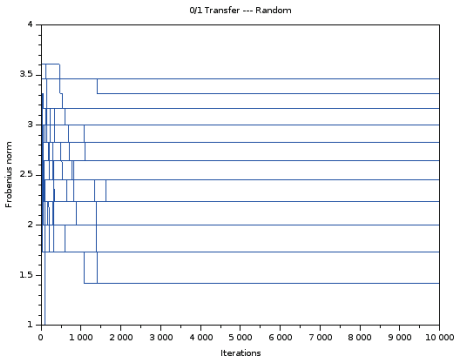

Figure 4.22: Convergence of SD

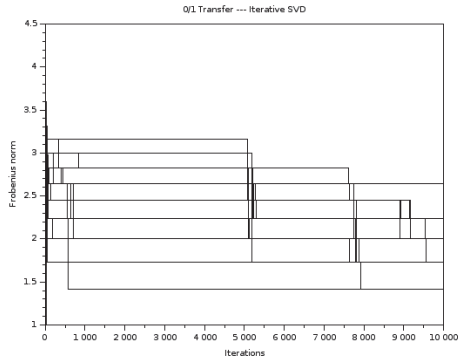

Figure 4.23: Convergence of ISVD

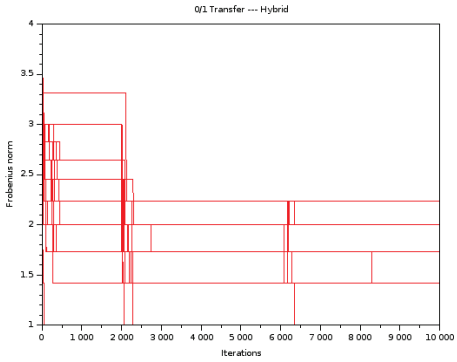

Figure 4.24: Convergence of Hybrid

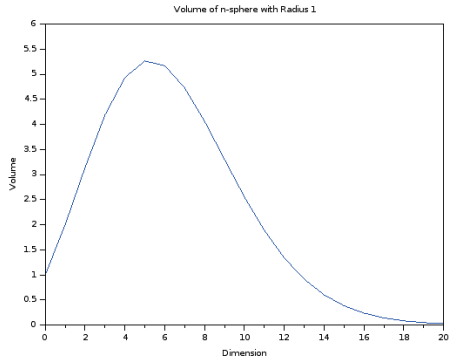

Figure 4.25: Volume of n-sphere

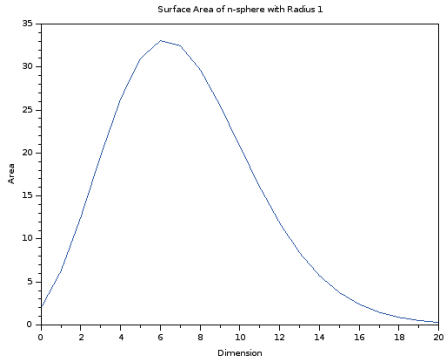

Figure 4.26: Surface Area of n-sphere

wise transfer functions — the main difference being that one can (or must) relax the orthogonality constraint. On the step-wise transfer functions, the SVD algorithm can be improved upon by switching to seaching for a new "singular vector" pair whenever the error falls. However, one cannot expect subsequent vector pairs to be anywhere near orthogonal to the previous pairs.

Considering the volume (or surface area) of the n-sphere, we can see that the method becomes more efficient in comparison with the simple stochastic gradient descent search as the dimension increases beyond approximately 5. That is, the surface area and volume of the n-sphere decreases as n grows larger, as can be seen in Figure 4.25 — whereas the volume of the n-cube increases exponentially.

In general, given that we want to find a matrix $A \in [-1,1]^{n\times n}$, a random search will potentially search a volume of

$$\text{Vol} = 2^{n^2},$$

meanwhile, each iteration of our method searchs for the i^{th} singular vectors, $\mathbf{u}_i, \mathbf{v}_i \in \mathbb{S}^{n-i}$, and the i^{th} singular value, $\sigma_i \in [0,1]$. The total volume potentially searched is therefore:

$$\text{Vol} = \sum_{i=0}^{n-1} (2S_i + 1),$$

where S_i is the surface area of the i-sphere (a ball in $i+1$-dimensional space), and

$$S_i = \frac{2\pi^{\frac{i+1}{2}}}{\Gamma\left(\frac{i+1}{2}\right)}.$$

We can see in Figure 4.27 how S_i, and hence the volume to search, increases only very slowly after $n = 15$. This is, naturally, somewhat offset by the computation required at each step to (nearly) orthogonalize the new prospective singular vectors to the previous singular vectors. However, with careful attention to how these vectors are updated during their convergence, this step need only be done at the very beginning. Further, while this orthogonalization step becomes more expensive with each iteration (that is, with each new singular vector pair), the space to be searched becomes smaller.

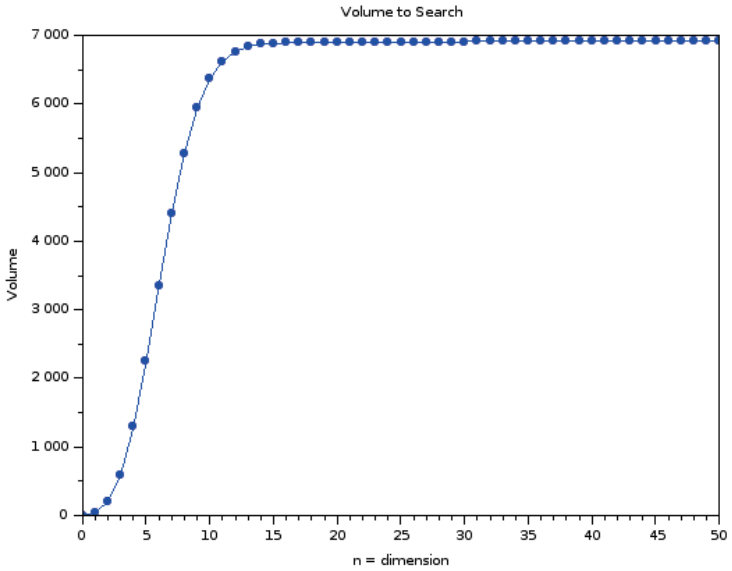

Figure 4.27: Volume searched by Iterative Method

Of course, TANSTAAFL, "there ain't no such thing as a free lunch." When we consider the real parameter space to be searched, whether by the the random search, or by the iterative SVD method, they are of equal dimension.

That is, given a matrix $A \in [-1,1]^{n\times n}$, it is clear that with the random search, the real parameter space has dimension n^2. With the iterative SVD search, $\mathbf{u}_1$ and $\mathbf{v}_1$, being on the surface of the ball in $\mathbb{R}^n$, require $n-1$ real parameters each and the singular value σ_1 another. The singular vectors $\mathbf{u}_2$ and $\mathbf{v}_2$ require $n-2$ and σ_2 another. This continues until $\mathbf{u}_n$ and $\mathbf{v}_n$ require a choice between two directions (i.e. they are *not* real parameters) while σ_n requires another real parameter.

The sum of all of these parameters is then, again, n^2. However, as we have stated many times, each successive singular triple becomes less and less important — allowing us the option to halt the search without searching those parameter spaces if desired.

To tie this iterative search to our later novel evolutionary SVD search, we wish to make note of a few things. The main thing to note is that one round of the iterative SVD search, even if one goes until convergence for each of the singular triples in order, may not find the optimal model (even restricting our view to the chosen space of models).

The iterative SVD method first finds the best rank one matrix approximation for our model. Then it finds, not the best rank two matrix approximation for our model, but the best rank two correction of the previous rank one model. This means the iterative SVD method should be run again if we wish to find the best model within the space.

But changing our found matrix, in particular changing the first singular vectors, means altering all subsequent singular vectors as well. Suppose then, that we have reached an approximation A_1 with a single round of the iterative search:

$$\mathbf{y} = \varphi\left(A_1\mathbf{x} + \mathbf{b}\right). \tag{4.6}$$

Rather than changing A_1, we can insert another matrix A_2, intended as a correction for A_1,

$$\mathbf{y} = \varphi\left((A_1 + A_2)\mathbf{x} + \mathbf{b}\right), \tag{4.7}$$

and run the iterative SVD algorithm on A_2. Finding a rank one, then rank two, etc. approximation for A_2, until the sequence converges again. We may continue if we like, finding A_3, A_4, etc., so that we have:

$$\mathbf{y} = \varphi\left(\left(\sum_{i=1}^{n} A_i\right)\mathbf{x} + \mathbf{b}\right), \tag{4.8}$$

where n is the number of times we are willing to run the iterative SVD algorithm.

Chapter 5

Supervised SVD Learning

The supervised learning method involves training from examples. Examples of supervised learning are learning to assign labels to images given a set of similar labelled images as in [19], or, as we have mentioned earlier, learning models for time series or learning continuous control behaviors (as for a car) from examples.

Generally supervised learning considers the problem of learning as either a classification problem or a regression problem. While SVD methods may also work for classification problems (as these are often carefully formed into a type of regression), they are particularly useful in regression problems.

Let's first consider the simplest possible case: the linear regression. Given a collection of observations $\{(\mathbf{x},y)_n\}$, we wish to find a linear (or affine) map $f(\mathbf{x})$ of the form:

$$f(\mathbf{x}) = M\mathbf{x} + b$$

so that $f(\mathbf{x}) \approx y$. This is generally taken to mean that we should minimize:

$$\sum_n (y - f(\mathbf{x}_n))^2,$$

the familiar "least squares" minimization.

The SVD, by way of the pseudoinverse, gives a straightforward (and, if implemented correctly, numerically stable) way to solve the linear regression. However,

even when one intends to obtain a non-linear regression through iterative methods, often the linear regression provides a good starting point for the iteration [77].

With regard to neural networks, we often use transfer functions which are nearly linear near the origin, and intentionally scale, center and normalize our variables in such a way as to concentrate their values near the origin. This being the case, we maximize the usefulness of the linear regression as a starting point, and the SVD as a method of performing regression.

Our experiments with supervised learning using the SVD consist of two parts. In one we consider time series approximation, starting with ARIMAX models (or truncated ARIMAX models) and then elaborating them with the addition of non-linear transfer functions and hidden variables/neurons. The second part involves learning a control mechanism for the TORCS/SCRC simulator. To this end, we first developed a traditional physics-based controller, which was able to complete several tracks in times comparable to the qualifiers of SCRC 2009 [51] and others [86, 10, 15, 14] without tuning parameters for each track or maintaining a map of the track, to act as input for the supervised method. We did not use human inputs as a sufficiently skilled driver was not available.

One of the most important elements of the supervised method (or any case in which we have sample inputs and outputs) is that we may be able to determine the true dimension of the output space and the necessary rank of the connection matrix — assuming a very thorough sample of the output space has been obtained with the given inputs. Of course, it should always be the case that we have some upper bound on the output space — namely the number of output variables. However, frequently these variables have dependencies among them, which we may use to our advantage — either by selecting candidate singular vectors only from the image space (reducing the dimension of the search space) and/or simply not searching the space of higher rank matrices, for example.

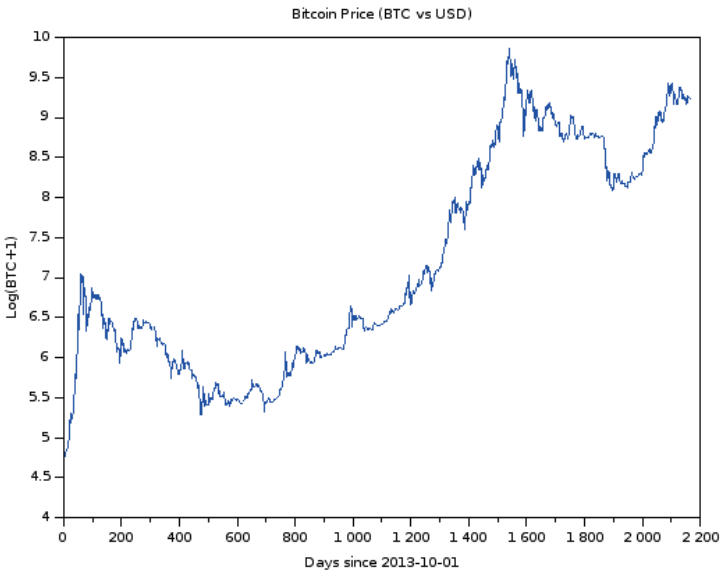

Figure 5.1: BTC price history (logarithmic scale)

5.1 Price Modelling

Our example of supervised learning is the modelling of Bitcoin price data. We obtained daily Bitcoin closing prices in euros from CoinDesk from 2013/04/28 to 2018/04/16 — a total of 1813 daily prices.

As usual with financial data where percentage change is the norm, we use the logarithm to obatin a more managable graph and condition the problem better for linear models — this means that any error we report is a relative error [27, 48, 65, 80]. While Figure 5.1, shows simply the logarithm of the price, to train and test our models, we will also normalize and center our data for training and testing.

5.1.1 Experiment Design

We will consider several different ARIMA type models, by way of comparison, before testing our fully recurrent neural network model.

	Mean	Standard Deviation
Training	0.001203	0.0458867
Validation	0.002866	0.0318813
Test	0.004742	0.0555125

Table 5.1: First Differences Statistics

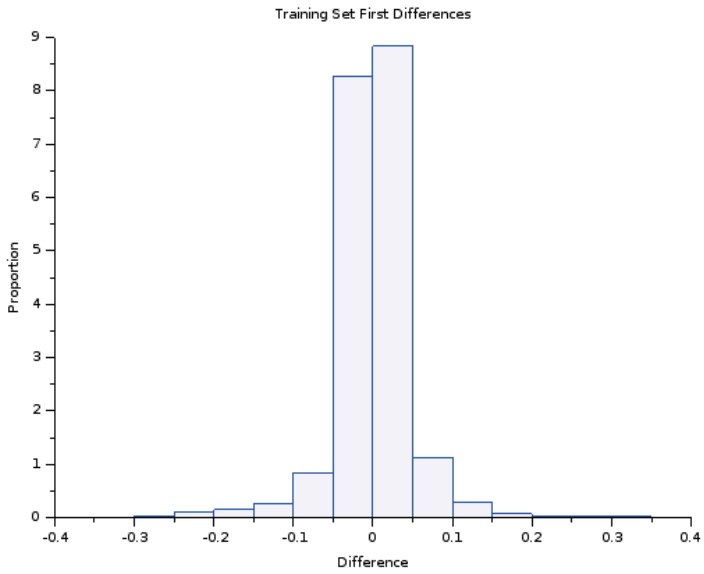

Figure 5.2: Histogram of First Differences of Training Set

We divided the data set into training, validation and test sets chronologically, using the first 1087 prices for training, the next 363 prices for validation, and the final 363 prices for testing.

Considering the first differences, Table 5.1 and the histograms in Figures 5.2 and 5.3 indicate we have fairly similar distributions of first differences between the different sets. However, the KolmogorovSmirnov test, indicates that the distributions may not be exactly the same — they are not, however, normal distributions.

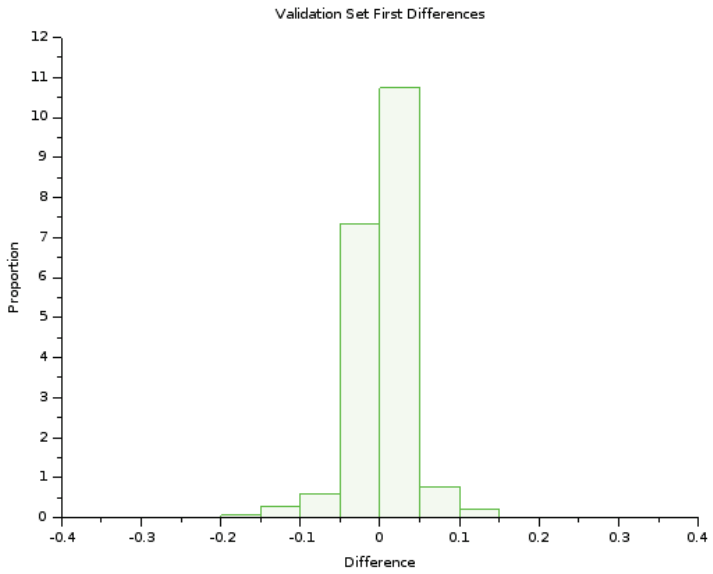

Figure 5.3: Histogram of First Differences of Validation Set

5.1.2 ARIMA Models

Our baseline model is the autoregressive model with a single term, namely if Y_t is the logarithm of the price (plus 1) at t, we model it by:

$$Y_{t+1} = \alpha Y_t + \varepsilon_{t+1},$$

where ε comes from the distribution defined by $Y_t - Y_{t-1}$. This should be a good baseline model as the correlation between Y_t and Y_{t-1} is more than 0.999. Using least squares we find $\alpha = 0.9993314$ — which we will round to 1, with the sum of least squares error of 2.4229876 and mean squared error of 1.3365×10^{-3}.

We note here that $\alpha = 0.9993314$ would provide an asymptotically stable model (converging to 0) if the approximation were left to run rather than being corrected by true values for each new timestep. Using $\alpha = 1$ instead, we obtain a simply stable model, rather than asymptotically stable, but, this model, predicting the next value with simply the previous value, is one of the simplest baseline models — the persistence forecast.

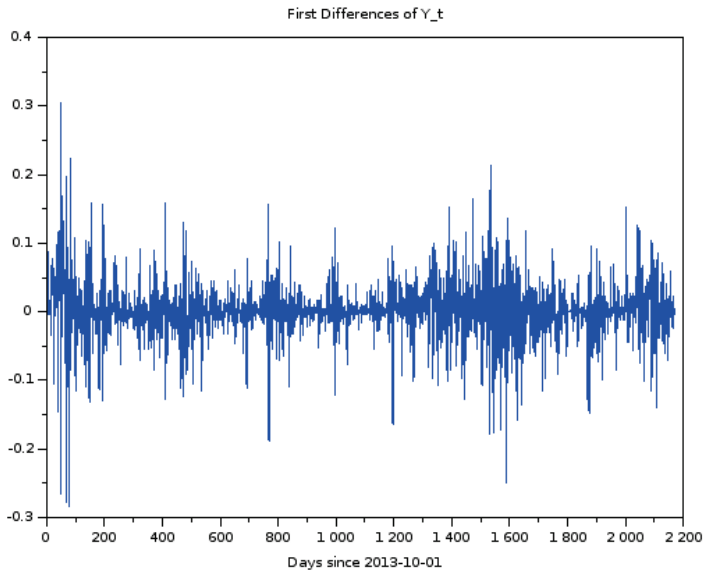

Figure 5.4: Differences in logarithm of prices

We can see in Figure 5.4 the first differences of Y_t, namely $Y_t - Y_{t-1}$ — which is centered near zero and has no apparent patterns. Further the correlation of one value to the next is approximately -0.003 — a very low correlation. Other tests for stationarity similarly indicate that the first difference is a stationary series. Figure 5.5 clearly shows a well centered histogram for the first differences of Y_t. The KolmogorovSmirnov test indicates that the distribution is not normal.

Using the model below with the constant μ:

$$Y_{t+1} = \mu + \alpha Y_t + \varepsilon_{t+1},$$

reduces the error to 2.4221776 and mean squared error to 1.336×10^{-3}.

Our next example uses the error from the in the first autoregressive term for the estimate of the previous value to correct the prediction for the next value. That is, our model is of the form:

$$Y_{t+1} = \alpha Y_t + \beta e_t + \varepsilon_{t+1}, \tag{5.1}$$

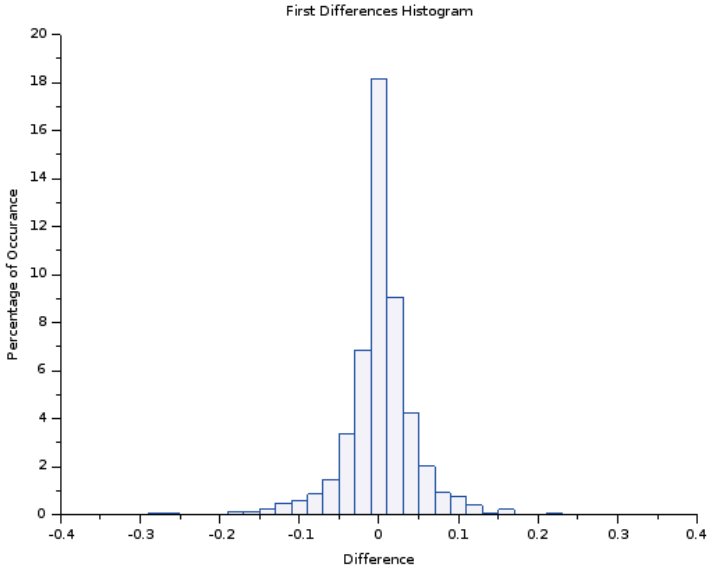

Figure 5.5: Histogram of First Differences

where e_t is the error in the prediction of Y_t. However, since the first differences have very little correlation, we shave a little off the sum of the squared error to 2.4221552 with a mean squared error of 1.336×10^{-3}.

Combining the two methods, we get:

$$Y_{t+1} = \mu + \alpha Y_t + \beta e_t + \varepsilon_{t+1}, \tag{5.2}$$

where we obtain again a sum of squared error 2.4221552 and the same MSE of 1.336×10^{-3}.

Using an integral of the original function with exponentially decaying weights as input as well, we have a model:

$$Y_{t+1} = \mu + \alpha Y_t + \beta e_t + \gamma \int_0^t e^{\kappa(t-\tau)} Y_\tau d\tau + \varepsilon_{t+1}, \tag{5.3}$$

for which we get the total error of 2.4211501 with MSE 1.3354×10^{-3}.

Using a two-sample Kolmogorov-Smirnov test on our test set, we have $D =$ 0.0589319 which is larger than the cut-off of 0.0547389 for significance at level

$\alpha = 0.05$ — allowing us to reject the null hypothesis and say that the distributions of errors for the lag model and this model are significantly different.

However, on the validation set, we get $D = 0.0248619$ while the cut-off is 0.0999392 — we cannot therefore reject the null hypothesis that the distribution of the errors in prediction are the same in this case.

Clearly, using slightly more complicated models, we can slowly shave the error down — up to some limit. But our examples so far have been linear models, and the error term they use (not the stochastic white noise term) is derived only from the error of the first autoregressive term.

5.1.3 Recurrent Neural Network Model

Let us move now to our more complete model, with hidden units and an error term based on the errors of it's own prediction. Our model is of the form:

$$\begin{pmatrix} Y_{t+1} \\ \mathbf{h}_{t+1} \end{pmatrix} = \varphi(A(Y_t, \mathbf{x}_t^T, e_t, \mathbf{h}_t^T)^T + \mathbf{b}) + \boldsymbol{\varepsilon}_{t+1}, \tag{5.4}$$

where φ is a transfer function (here simply a limit on the minimum and maximum values), Y_t is as before, $\mathbf{x}_t$ are chosen exogenous variables (the integral in Eqn. 5.3, for example), e_t is the error of the previous approximation (not just that of the first autoregressive term), $\mathbf{h}_t$ is a collection of hidden variables, and $\boldsymbol{\varepsilon}_t$ is a white noise or stochastic term if ranges of predictions are desired.

In fact, we use an example very similar to that of Eqn. 5.3, with simply the addition of hidden terms — and begin our training of the model with the least squares solution from that example.

Using a single hidden variable, we obtain a sum of squared error of 2.421098, MSE of 1.3354×10^{-3}.

Coin	Method	Error	MSE
Bitcoin	$Y_{t+1} = \alpha Y_t + \varepsilon_{t+1}$	2.4230	1.3365×10^{-3}
Bitcoin	$Y_{t+1} = \mu + \alpha Y_t + \varepsilon_{t+1}$	2.4222	1.3360×10^{-3}
Bitcoin	$Y_{t+1} = \alpha Y_t + \beta e_t + \varepsilon_{t+1}$	2.4222	1.3360×10^{-3}
Bitcoin	$Y_{t+1} = \mu + \alpha Y_t + \beta e_t + \varepsilon_{t+1}$	2.4222	1.3360×10^{-3}
Bitcoin	$Y_{t+1} = \mu + \alpha Y_t + \beta e_t + \gamma \int_0^t e^{\kappa(t-\tau)} Y_\tau d\tau + \varepsilon_{t+1}$	2.4212	1.3354×10^{-3}
Bitcoin	Equation 5.4 (single hidden variable)	2.4211	1.3354×10^{-3}
Ethereum	$Y_{t+1} = \alpha Y_t + \varepsilon_{t+1}$	1.4347	9.603×10^{-4}
Ethereum	Equation 5.4 (Bitcoin model)	1.4396	9.636×10^{-4}
Ethereum	Equation 5.4 (singular values retrained)	1.4328	9.59×10^{-4}

Table 5.2: Summary of Price Prediction Results

5.1.4 Retraining Singular Values

To further demonstrate the usefulness of the method, we trained a network first on time series data for the Bitcoin price. Then we trained the same model again to model the Ethereum price, but modified only the singular values and the shift or bias vector making the transformation affine — holding the singular vector pairs fixed from the training for Bitcoin.

Using only the lagged values for the Ethereum price resulted in a sum of squares error of 1.4346877 with an MSE of 9.603×10^{-4} . When using the model trained directly on Bitcoin, meanwhile, the error was 1.4396236 with an MSE of 9.636×10^{-4}. After training only the singular values and the shift vector (training $2n$ parameters rather than $n^2 + n$ parameters), we obtain a sum of squared errors of 1.4328113 with an MSE of 9.59×10^{-4}.

The results of our experiments with Bitcoin and Etherium price prediction are summarized in Table 5.2.

5.1.5 Remarks

The previous section concerns one of the main benefits of our method — which should be highlighted. We developed and trained our original model on one set of data (Bitcoin prices), obtaining results better than the standard autoregressive

models, but then we were able to turn to a similar problem and reuse most of our model — not just the general structure of the model, but also the "structure" of the connections between the the variables given by the singular vectors.

In fact, besides the novel method of learning, using the SVD in an evolutionary method (which we will see later), this is one of the main contributions of this work — the idea of reusing the structure of recurrent neural networks (as defined by the singular vectors) and retraining only the relative strengths of those structural components for a new problem. Clearly, in general, the reuse of neural network "structures" is widespread — however, the "structure" intended in these contexts has almost always meant the neural network architecture. However, reusing just the architecture means retraining all the connections as well — possibly n^2 connections between two layers of n neurons. Meanwhile, reusing the "structure" given by the SVD means retraining just n singular values between those two layers of n neurons.

5.2 Predictive Maintenance

Another example we consider is the development of a model for the prediction of the operation of an industrial machine, used to test resistance of materials and electronic components to temperature fluctuation, and, particularly, stochastic models which allow us to develop intervals of normal operation.

5.2.1 Model Development

We examine the various variables of the machine and first develop a deterministic model to predict the main variables. Our first step was to consider the correlations, specifically the Spearman correlation, between the various variables and transformations of the variables. We used 0.5 as the cut-off for the strength of correlation to consider.

Using the 35 variables themselves, we obtained 232 pairs with an absolute correlation above 0.5. Considering correlations of first differences (i.e. numerical derivatives) of the variables we obtained only 12 pairs with a correlation above the 0.5 level. Next we considered the original variables against the derivatives, from which we obtained again 12 pairs of correlates above the 0.5 level.

Then the original variables against the cumulative sum of each of the others — leading to 19 pairs of correlations, always considering those above the level of 0.5.

Finally, we considered the original variables against a rolling exponentially weighted integral of the form:

$$I_X(t+1) = X(t+1) + 0.999 I_X(t). \tag{5.5}$$

From these comparisons, we obtained 329 correlations above the level of 0.5.

Taking the intersection of all the variables appearing as a correlate, we obtained only the four variables: T1, T2, T3, T4 — four temperature readings within the machine.

As the machine is used to test resistance to changes in temperature, these are indeed the most important variables. After this simple analysis our work became to provide a model for the temperature, to predict future values (or a probable range of values), and inform the operator if these values fall outside the range of normal operation.

Our goal was a prediction which has an error of approximately the same as the simple first lag prediction, but with as much warning time as possible.

The variables with the highest correlation (> 0.8) to the variable T1 (temperature reading one) are as in Table 5.3 as an example of the main variables used and their inter-correlations. The other temperature readings had similar correlations. Of course, to make predictions into the future, we will need to consider lagged versions of most of these variables — which also have high correlations.

Description	Name	Correlation
T1 lagged by 1	$T_1(t-1)$	0.9999
T2	T_2	0.9909
T3	T_3	0.9942
T4	T_4	0.9927
Setpoint	SP	0.8696
Rolling integral of PID1	$IPID_1$	0.9386
Rolling integral of PID2	$IPID_2$	0.9375
Rolling integral of PID5	$IPID_5$	0.9393

Table 5.3: Correlations to T1

Of particular importance, however, is the Setpoint of the machine, which, as the name suggests is the setpoint for the temperature given as a control by the human operator. The differences between the Setpoint variable and the various temperature measurements are used as the input for the PID controllers — which control various heating and cooling systems in an effort to reach and maintain the temperature given by the Setpoint.

We note that the parameters of the various PIDs and how they were obtained, were not available to us.

Besides the fact that the Setpoint variable provides us with the intended value of the temperature, it is interesting as, under normal operation, it is a discontinuous function (see Figure 5.6) and, most importantly, it is known in advance — meaning we can use not only current values, but future values to predict future states of the machine.

As mentioned earlier, we wish to develop a model which has error in predictions at the same order of magnitude as using just the previous value for an estimate of the current temperature. That is, if we consider the model:

$$T1(t+1) = T1(t) + \varepsilon(t+1), \tag{5.6}$$

we wish to develop a model with errors (ε) of the same magnitude, but with as much forewarning to the operator as possible.

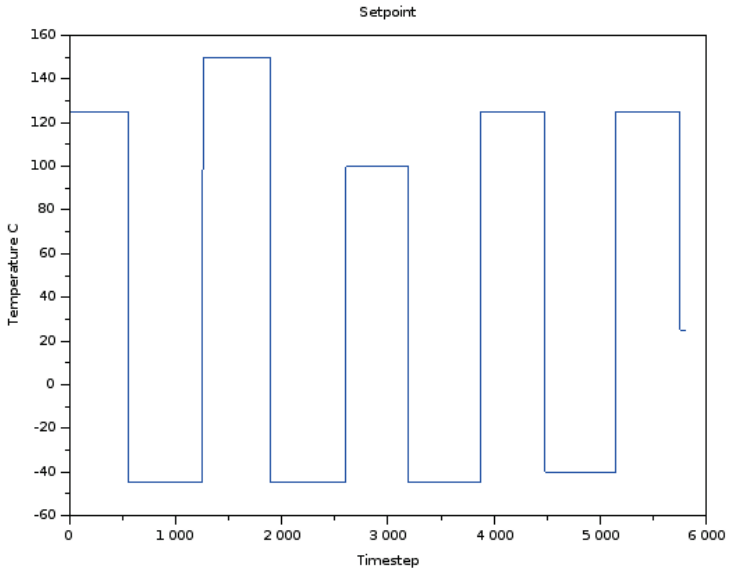

Figure 5.6: Setpoint

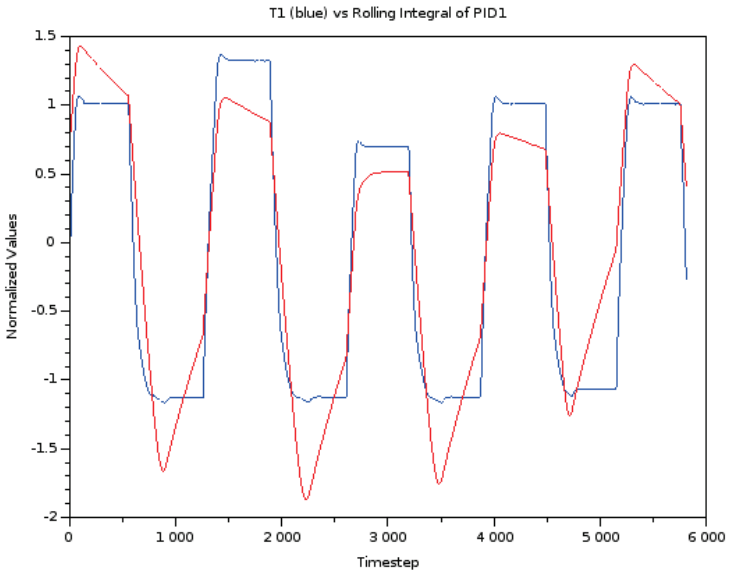

Figure 5.7: T1 vs Rolling Integral of PID1

After centering and normalization, the sum of squared error between the first lagged value and T1 is 0.3828, while the MSE is 6.59×10^{-5}. Therefore, our goal is to develop a model which predicts as far into the future as possible, with errors of the same order of magnitude. See Figures 5.8 and 5.10 for the error and a histogram of the errors from this model. Note, of course, that the greatest errors come around the time that the Setpoint changes and the machine alters the temperature very quickly.

As a first attempt, we considered the autoregressive model with the exogenous variables above, with a lag of 5 timesteps, except for the Setpoint variable, for which we used the current value. That is, our model is of the form:

$$T1(t+5) = \sum_{i=1}^{4} \alpha_i T_i(t) + \beta SP(t+5) + \sum_{i=1,2,5} \gamma_i IPID_i(t) + \delta e(t) + \mu + \varepsilon(t+5), \tag{5.7}$$

where $e(t) = T1(t) - T1(t-5)$ is the actual difference in temperature from time $t-5$ to time t — one can think of this as either a derivative or the error one would get by simply using $T(t-5)$ as a model.

This model provided a sum of squared errors of 0.6631 and a MSE of 1.141×10^{-4} — just above, but very near the limits of error which was our goal. See Figures 5.9 and 5.11 for the errors and an error histogram for this model.

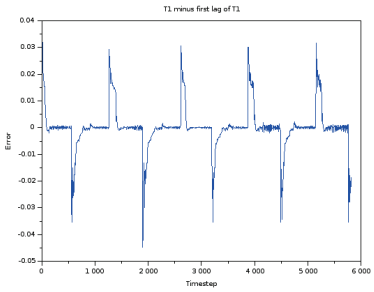

Figure 5.8: Error in approximating T1 with the first lag

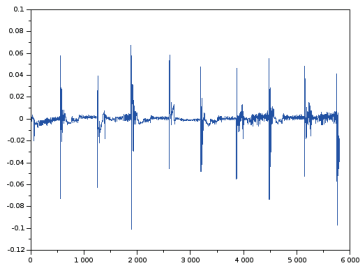

Figure 5.9: Error in approximating T1 as in Eqn. 5.7

Using the distribution of the error from the deterministic part of the model, we can provide a range of expected values for T1. That is, adding and stochastic term, with values taken from the distribution of errors we found and running our stochastic

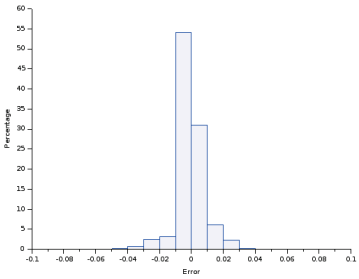

Figure 5.10: Histogram of error in approximating T1 with the first lag

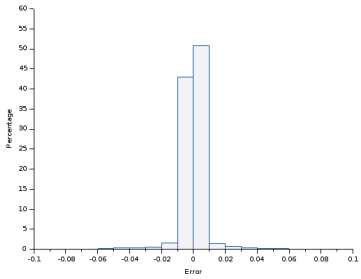

Figure 5.11: Histogram of error in approximating T1 as in Eqn. 5.7

model several (in this case 1000) times, we obtain a range of values for the $t+5$ value of $T1$. Maintaining just the middle 95% of the values allows us to obtain a confidence interval for the value of $T1$ as in 5.2.1. In fact, counting the number of times the value of $T1$ is outside of this interval, we get 242 instances of 5811 timesteps, or a percentage of about 4.2.

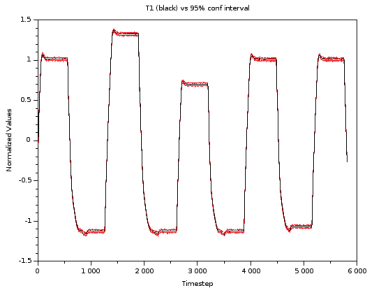

Figure 5.12: T1 and Confidence Interval

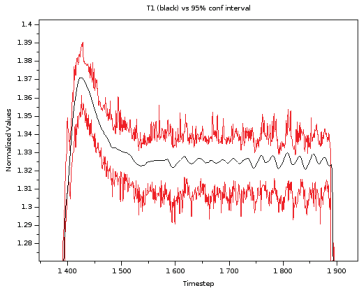

Figure 5.13: Zoom In of T1 and Confidence Interval in HOLD UP State

5.2.2 Remarks on the Neural Network Model

Our next step was an attempt to improve the model, as with the Bitcoin example, with the addition of a hidden variables and a non-linear transfer function. However, though we achieved results comparable to the previous models, the results using the fully recurrent neural network model were not an improvement.

So here we saw some of the limits of the fully recurrent model — in that, given the same variables and information, the extra model complexity provided no ben-

efit, at least not for the prediction of the value of a single temperature sensor. We are left with the possibility of training the neural network model for one temperature sensor and then retraining only the singular values for the others, or, perhaps, producing a holistic vector valued predictor for all temperature sensors. However, these possibilities must be left to future work due to the limitations of time.

5.3 Driving in TORCS/SCRC

Our main experiment was to design, implement, train and test a neural network controller for the Simulated Car Racing Championship (SCRC) real-time driving simulator based on The Open Racing Car Simulator (TORCS). The TORCS/SCRC simulator provides several variables about the car's position and movement as well as controls for steering, acceleration, braking and gear shifting — all of these variables and controls are updated on very short time intervals (20ms, essentially real-time), requiring potential controllers to maintain a high response time for optimal performance [50].

5.3.1 Experiment Design

The first step of our experiment design was to select a number of tracks available in TORCS, according to the criteria below. We then split the tracks into three sets: training, validation and test. As the tracks are few in number and in the interest of optimizing learning time, different tracks were carefully chosen for these purposes. First, the training and validation tracks are generally shorter than the test tracks. Second, the tracks have been divided into essentially four types: fast, curvy, alpine (changes in elevation), and general (fast with hard turn). Three of the classes, fast, curvy and alpine, are represented in the training set, a "general" type for the validation, and all four types are represented in the test set.

The training set consisted of tracks: Ruudskogen (fast), Aalborg (curvy) and Alpine 2 (alpine).

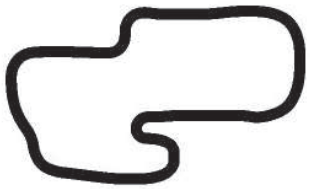

Figure 5.14: Ruudskogen — fast training

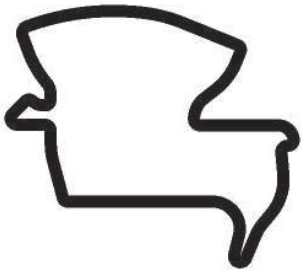

Figure 5.15: Aalborg — curvy training

Figure 5.16: Alpine 2 — alpine training

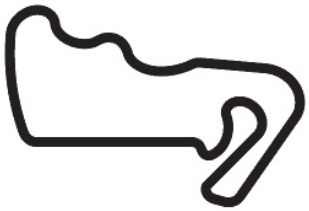

Figure 5.17: Wheel 1 — general validation

The validation set consisted of the track Wheel 1 (general).

Finally the test tracks consisted of: Forza (fast), Brondehach (curvy), Alpine 1 (alpine), and Wheel 2 (general).

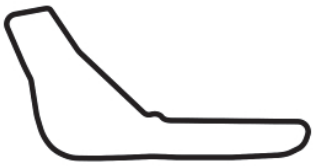

Figure 5.18: Forza — fast test

Figure 5.19: Brondehach — curvy test

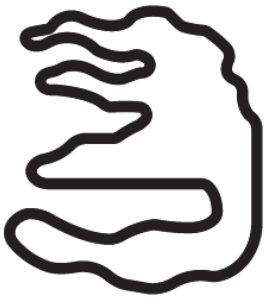

Figure 5.20: Alpine 1 — alpine test

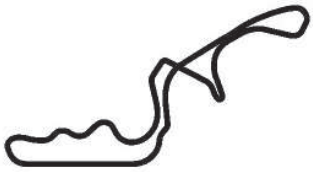

Figure 5.21: Wheel 2 — general test

5.3.2 Programmed Controller

Before attempting supervised learning for our neural networks, we first needed a controller to model. SnakeOil provides a very simple controller, but it is extremely slow and maintains a position directly in the center of the track. Though in future experiments this type of controller may be an interesting starting point — modelling a very conservative controller and then attempt to improve upon it using unsupervised learning — we instead elected to produce a physics-based controller which could complete the chosen tracks in a fairly competitive time.

To this end, we developed a relatively simple controller based on "desired" setpoints for speed (actually for kinetic energy) and steering angle as well as PIDs for smooth control to reach these setpoints and deal with traction control — these PIDs were not highly optimized but sufficient for our purposes. For throttle and brake control, in particular, an anti-skid/traction control (ABS) system was implemented — limiting either throttle or brake when overspin or skidding was detected.

We developed a controller much faster than the simple SnakeOil controller by using a setpoint for kinetic energy rather than speed. The setpoint was based on how much track the car needed to come to a complete stop (assuming braking distance being proportional to kinetic energy) and the amount of track available. That is, the setpoint for kinetic energy was the maximum value for which the car would have enough track to stop given the visible track ahead. As we did not consider the

internal workings of TORCS, this value was approximated experimentally — using a value which allowed the controller to complete all chosen tracks in a competitive time. Certainly, on different road surfaces, such as the dirt tracks which were not a part of our experiment, this value would change.

For steering, rather than simply sticking to the middle of the track, we based the steering setpoint on the direction in which the longest length of track was available. That is, whichever angle provided the most visible track became the setpoint for steering. This, together with a small offset to prevent the wheels going off the track (especially on sharp turns), allowed the controller to cut turns quite effectively. (We also modified the forward track sensors of the SCRC car to be more heavily focused to the front of the car, rather than equally spaced in the forward semi-circle as is the default for the SCRC/SnakeOil software.)

We note that the controller developed did not have maps for planning future actions — to optimize speed around corners by following the best line or to drift to the outside before a corner, for example. Instead, the maximum speed allowed (or maximum allowed kinetic energy) was, as previously mentioned, simply based on the required stopping distance at that speed and the amount of track locally visible in front of the car. Put another way, the programmed controller didn't "know" that a turn wasn't simply a dead-end as the only information available to it was the local track information described more fully in Table 5.5 on page 64.

It is of note that the range-finding variables used have an i.i.d. normal noise of 10% of the sensor's current real value when the "noisy sensors" option is enabled — which we have enabled for our work. For our hand-coded example we did nothing to counteract or filter this. Also, interactions with other cars, overtaking, etc., were not considered, both for simplicity and as this would also best be tackled with maps or models of the track and perhaps the opponents as well.

Having the set-points for speed (or actually kinetic energy) and direction, we then developed very simple PID controllers for acceleration, braking and steering

	SnakeOil	Hand-Coded	Median SCRC 2009	COBOSTAR
Ruudskogen	118	82		≈ 72
Aalborg	151	84		
Alpine 2	156	110	118	≈ 136
Wheel 1	165	109		
Forza	208	115	111	
Brondehach	149	100		
Alpine 1	262	169		
Wheel 2	233	146		

Table 5.4: Track Times in Seconds

to acheive responsive and relatively smooth control. As the set-points were almost continually changing, the proportional element of the controllers was the most important element. These parameters had to be arrived at experimentally (though they were not perfectly optimized) by examining the controller's acceleration and braking (particularly whether large amounts of over- or under-spin or whether high frequency switching between acceleration and braking were present) and examination of whether the controller was able to make turns appropriately. (We also maintained a very simple traction control system, reducing braking or accelleration whenever under- or over-spin of the tires was detected.)

The various parameters of this controller were then tuned to run the chosen tracks as quickly as it could, and after a few iterations, we were able to develop a controller with best track times as in Table 5.4 — acheiving times comparable with those of SCRC competitors and others appearing in the literature [51].

In addition to our hand-coded controller, we list the best one-lap times (in seconds) of the provided SnakeOil controller, median times for the 2009 SCRC ranking competitors, and times reported for the COBOSTAR controller of the 2009 SCRC in Table 5.4 [13, 51]. (Other SCRC competitions used entirely new tracks generated for the competition and unseen previously by the competitors.)

Though the data is sparse, we note that the controller seems to be somewhat slower on fast tracks and somewhat faster on curvier tracks than the SCRC 2009

competitors — we speculate that the speed on fast tracks could be improved with maps and planning, but that is outside the scope of this work.

5.3.3 Supervised Learning

Given our construction of the physics and PID based controller, we considered two different methods of control for the neural network. First, we could train the neural network to control the steering and acceleration directly (though we would keep the traction control), or we could have the neural network simply set the setpoints and allow the steering and acceleration PIDs to function as before.

We chose to control the steering and acceleration/braking directly, as a more robust test of the real-time control properties of the neural network method. However, we may consider the setpoint method in the future, given the smoothness, robustness and transparency of the PID control.

As in the previous examples, we first developed an VARX ("V" for vector) type model to model or predict the next state (outputs and inputs) of the physics based controller. This example was clearly more complex than the previous models, being vector valued — though this could have been explored in the previous example.

Most of the variables used were a subset of those in the TORCS/SCRC implementation (see [84]). In particular, we used the variables in Table 5.5 directly from TORCS/SCRC.

In addition, we used some calculated variables, which were also used by the physics based controller, as inputs for the neural network. These were acceleration and kinetic energy in the X (forward) and Y (side) directions and also appear in Table 5.6.

Meanwhile, the outputs of the physics based controller from the previous timestep, as in Table 5.7, were available as inputs to the recurrent neural network during train-

ing. These variables were also, of course, available to the neural network during operation.

We did apply smoothing to the track distance sensors during training and operation of the neural network — optimizing the smoothing parameter against a clean run. Using smoothing of the form:

$$\text{smoothTrack}(t) = (1-\alpha)\text{track}(t) + \alpha\text{smoothTrack}(t-1),$$

with the smoothing factor of $\alpha = 0.9$, we had a relative error against the noisy sensors of about 3.03% and a relative error against the non-noisy sensors of about 9.82%. The graphs of two of the track sensors and the associated smoothed track variables for one of the chosen tracks can be seen in Figure 5.22.

We first simplified our model by not considering acceleration (or gas) and braking separately — but by combining the two, with positive numbers indicating acceleration and negative numbers indicating braking.

We then developed least squares models to fit the "acceleration minus braking" (amb) and steering output variables of our hand-coded model. The model for the

Name	**Description**
distRaced	Distance covered by car from the beginning of the race
speedX	Speed of car along logitudinal axis (i.e. the usual speed)
speedY	Speed of the car along the transverse axis
track	Vector of 19 range sensors returning distance to the edge of the track at specific angles in front of the car
trackPos	Normalized distance between the car and track axis

Table 5.5: Input Variables from TORCS/SCRC

Name	Description
accX	Acceleration in X direction
accY	Acceleration in Y direction
keX	Kinetic energy in X direction
keY	Kinetic energy in Y direction

Table 5.6: Calculated Input Variables

Name	Description
accel	Virtual gas pedal
brake	Virtual brake pedal
steer	Steering value

Table 5.7: Output Variables Used as Input Variables for Neural Network

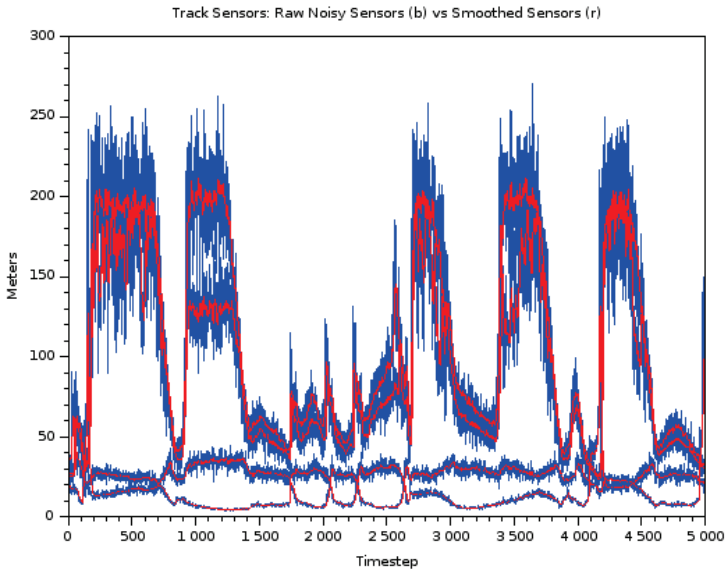

Figure 5.22: Raw versus Smoothed Track Range Sensors

"amb" variable was:

$$\begin{aligned}\text{amb}(t+1) &= \varphi\,(c_1\text{angleN}^2(t)+c_2\text{speedXN}(t)+c_3\text{keXN}(t)\\ &\quad +c_4\text{steer}^2(t)+(c_5,\cdots,c_{23})\mathbf{smoothTrackN}(t)\\ &\quad +c_{24}\text{trackPos}^2(t)+b),\end{aligned}$$

where the variables are as above and in [50], with the addition of "N" indicating scaling the indicated variable to have unit standard deviation. Note that the "angleN," "steer," and "trackPos" variables are squared — making all variables used here non-negative. The chosen transfer function, either a bounded identity function or the tanh function, is represented by φ.

When we considered modelling the "acceleration minus brake" variable (range $[-1,1]$) using the bounded linear (identity) transfer function, we obtained a MSE of 4.167×10^{-2}. Whereas, using non-linear least squares, we obtained a MSE of 4.274×10^{-2} using the tanh transfer function. Graphs comparing the hand-coded "amb" variable with the models developed here, using the two different transfer functions, can be seen in Figures 5.23 and 5.24.

Meanwhile the model for the "steer" variable was:

$$\begin{aligned}\text{steer}(t+1) &= T(c_1\text{angleN}(t)+c_2\text{speedXN}(t)+c_3\text{keXN}(t)\\ &\quad +c_4\text{amb}(t)+(c_5,\cdots,c_{23})(\sin\boldsymbol{\theta}*\mathbf{smoothTrackN}(t))\\ &\quad +c_{24}\text{trackPos}(t)+b).\end{aligned}$$

Notice here that the variables "angleN" and "trackPos" are not squared, but retain their sign. Meanwhile $\sin\boldsymbol{\theta}*\mathbf{smoothTrackN}(t)$ is the element-wise product of the track sensors with the sine of their angle from forward — encoding the direction of the sensor as well as the magnitude for steering.

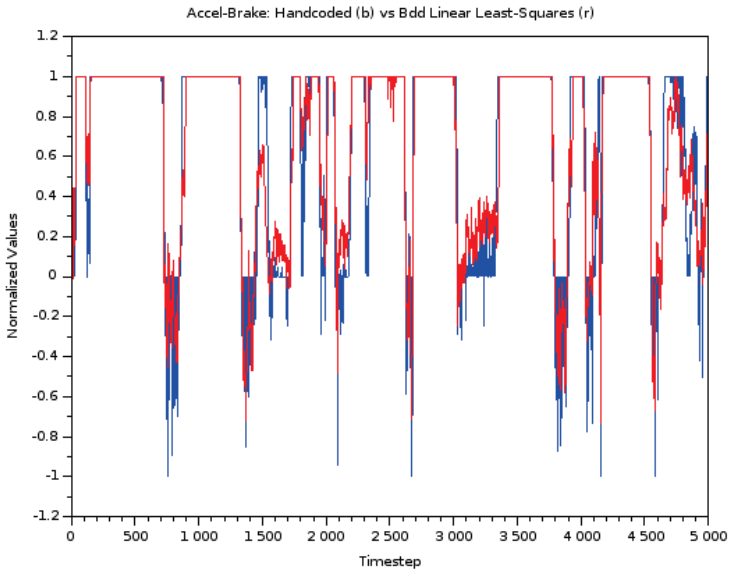

Figure 5.23: Hand coded versus Bounded Linear Least Squares Model for Accel-Brake

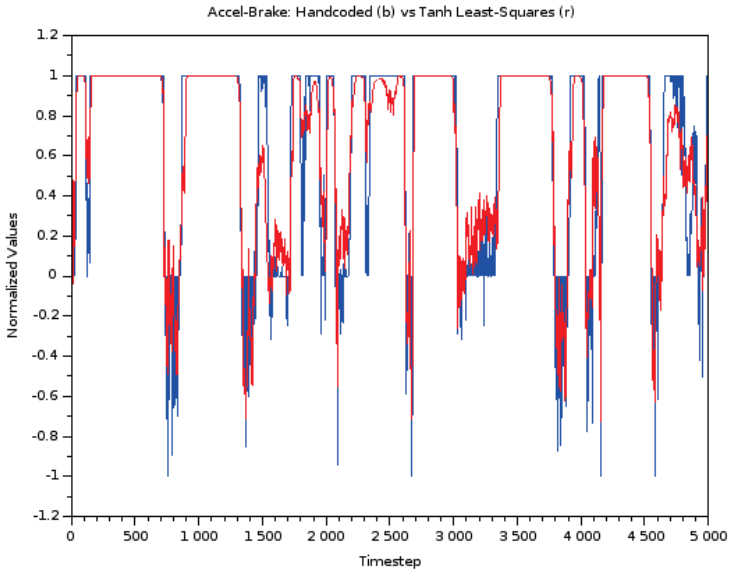

Figure 5.24: Hand coded versus Tanh Least Squares Model for Accel-Brake

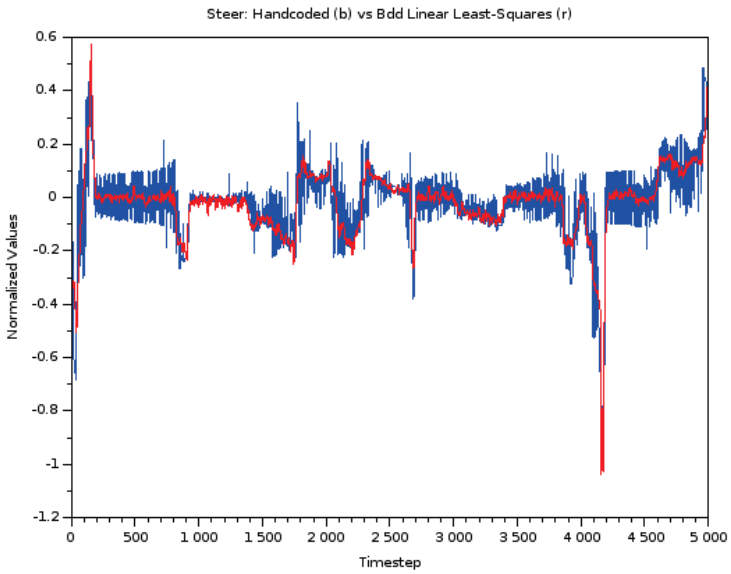

Figure 5.25: Hand coded versus Bounded Linear Least Squares Model for Steer

Error for steering (again range of $[-1,1]$) using the bounded linear (identity) transfer function gave a MSE of 5.797×10^{-3}. In fact, the result was much smoother than the hand-coded steering, as can be seen in Figure 5.25. Though the magnitude of the coefficient vector for this model is greater than 1, indicating that the model may be unstable, not a pleasant thought for a steering model, this is a result of the very low weighting of the central track sensors. If these weightings are instead considered to be a part of the coefficients, we obtain a norm less than 1 — and a stable model, meaning our steering will naturally track toward the center after perturbation.

Using the tanh transfer function allowed for a MSE of 5.730×10^{-3} — with the result being again smoother than the original (see Figure 5.26). And, again, after adjustment by the sine of the track sensor angle, the norm of our coefficient vector is less than 1 — lending stability to the model in the face of perturbation.

The next step was to test our models obtained by least-squares fitting to the Acceleration-Brake and Steer variables against the tracks we had chosen. In Table

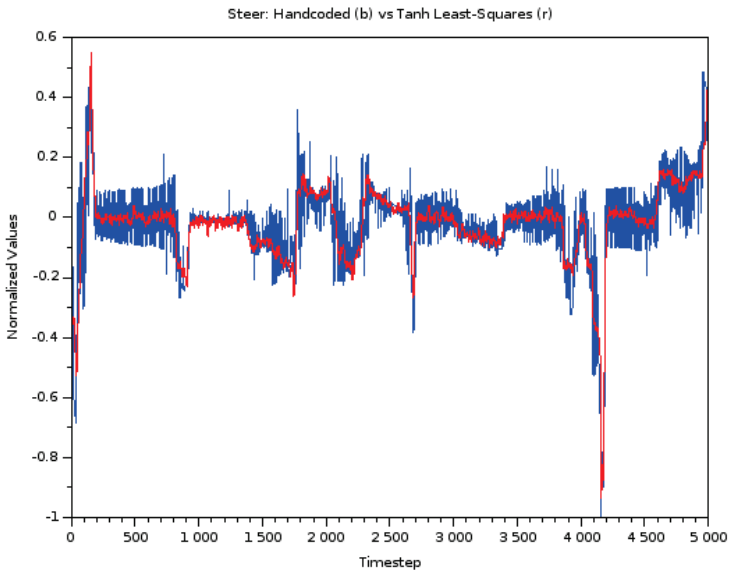

Figure 5.26: Hand coded versus Tanh Least Squares Model for Steer

	Hand Coded Controller	BLLS	Tanh LS
Ruudskogen	82	83	85
Aalborg	84	88	86
Alpine 2	110	113	114
Wheel 1	109	110	113
Forza	115	118	117
Brondehach	100	105	102
Alpine 1	169	176	175
Wheel 2	146	150	150

Table 5.8: Track Times

5.8 we list the resulting times, “BL” stands for the “bounded linear” transfer function — all times are the second best of 5 runs rounded to the nearest second.

5.3.4 Neural Network Model

Our next step was to apply the procedure we have outlined, using the iterative SVD search with the addition of 5 hidden variables. We considered only the model with the tanh transfer function as it had performed better on the test tracks.

After this we attempted to “fine tune” the neural network for each type of track by changing only the singular values. Doing this, we acheived times on the test track

	Hand Coded Controller	Tanh LS	Supervised SVD	Fine-tuned
Ruudskogen	82	85	83	82
Aalborg	84	86	83	82
Alpine 2	110	114	108	106
Wheel 1	109	113	107	105
Forza	115	117	113	112
Brondehach	100	102	98	95
Alpine 1	169	175	173	171
Wheel 2	146	150	145	145

Table 5.9: Track Times

approximately 1% better than the "general" controller. But this will be discussed futher in Chapter 6 on the unsupervised evolutionary SVD algorithm.

Chapter 6

Unsupervised Evolutionary SVD

The unsupervised evolutionary SVD algorithm is, in some sense, the simplest — but, if starting from a random initialization, requires the longest time to converge as it must learn directly from interaction with the environment and adaptation to it, rather than copying a model.

Because of the possible time needed to converge upon an usable solution using only unsupervised methods, it is clearly beneficial to use the supervised methods whenever possible, however, we have also tested the unsupervised method on problems for which the supervised methods are also applicable — as a method of comparison for its efficiency. Also, we note that there are occasions when we would like to attempt to improve upon the model developed by supervised learning — something we will discuss at some length.

In this method we construct single layer recurrent neural networks as in the previous methods. Naturally, we require some fitness function for our learning problem, which we seek to minimize.

6.1 TORCS/SCRC Experiment

Our example experiment is again to develop a controller for the TORCS/SCRC simulated car. Given a particular track, our fitness function was simply the distance

travelled along the track in a fixed length of simulation time. We note that the 2009 SCRC competition used distance raced in 10,000 game ticks (about 3 minutes and 20 seconds of real time) on each track as the qualifying stage [51].

The unsupervised method for training our fully recurrent neural networks consists of seeding several initial controllers with random connections. We note, again, however, that one benefit of using the single- layer fully connected neural network (allowing connections directly from input to output), is that we can "pre-program" the network with weights which are, at least, probably of the correct sign — this can speed up convergence to a usable controller.

Another method, providing even faster convergence in the unsupervised stage, is to use networks previously trained using supervised methods. Of course, further refining models trained under supervised methods may not always be possible in the context of a particular problem, but in the case of the TORCS/SCRC controller, we were able to shorten the time needed for the learning to converge by a considerable amount.

6.2 SVD Evolutionary Method

Our SVD evolutionary method is based on "mixing" or "crossing" the singular value decompositions of two candidate solutions. Given two matrices M_1 and M_2, we "cross" or combine the two by crossing the "left side" of SVD of one matrix with the "right side" of the other. Meanwhile, we create a new set of singular values — this may be done with the geometric mean of the parent singular values, or some other way, perhaps even the drastic measure of setting them all near one and retraining.

Given two matrices M_1 and M_2,

$$M_1 = U_1 \Sigma_1 V_1^T \text{ and } M_2 = U_2 \Sigma_2 V_2^T,$$

and letting "$\widehat{\cdot}$" denote a perturbation to be defined below, we obtain a "child" matrix, M, in the following way:

$$M = \widehat{U_1}\Sigma\widehat{V_2}^T, \tag{6.1}$$

where $\widehat{U_1}$ and $\widehat{V_2}$ are slight rotations of U_1 and V_2 respectively.

(We will come back to this point in the concluding remarks, but it may be that the control of this perturbation is the most important part of the method. We note here that these are rotations because we are generally assuming that the singular vectors of U_1 and V_2 are already fairly close to their appropriate values — reflections being, therefore, unnecessary.)

The singular values for M may be obtained in various ways. One is to use a perturbation of the geometric mean of Σ_1 and Σ_2, that is,

$$\Sigma = (\widehat{\Sigma_1\Sigma_2})^{0.5}, \tag{6.2}$$

using the geometric mean as the values are all non-negative. The perturbation in this case being multiplication by $\widehat{I}$, a diagonal perturbation of the identity matrix. Note that the matrix Σ, in this way, remains a diagonal matrix with non-negative values on the diagonal. However, depending on the magnitude of the perturbation, and the relative differences between the singular values, the values on the diagonal may no longer be in decreasing order.

There are a few methods to combine two matrices using the singular value decomposition. One, and the simplest, is to do exactly as we have described above. When one simply applies this method, the order of the singular vectors is affected by the order of the associated singular values. Hence, even if two matrices have very similar singular vector pairs, but singular values which happen to be in a different

order, one can break the pairing of the left and right singular vectors. Therefore the order of the singular values is vitally important.

This method can have difficulties converging, in fact, without an objective function to minimize, the norm may increase through time. That is, generating two random matrices, "crossing" them in this manner (with a small random perturbation), and continuing to cross the resulting matrices, may not converge to any matrix. And it's fairly clear why this is the case. Though the singular values may quickly converge, the right and left singular vectors simply get swapped back and forth (first U_1 is matched with V_1, then with V_2, then with V_1 again, and so on) with small random perturbations.

Without any "pressure" from an objective function or without some kind of interchange of information or "averaging" between the pairs of singular vectors, there is no impetus to converge, just the switching of the singular vector pairs *ad infinitum* (with random perturbations). However, we should keep in mind that there is no expection of convergent behavior in the abscence of an objective function.

A more sophisticated method involves ranking the pairs of singular vectors by their similarity. This has the advantage of allowing singular values to change order while still maintaining the same or similar singular vector pairs. This allows the "magnitude" of response, encoded in the singular values, to be learned almost independently from the "method" or type of response, encoded by the pairs of singular vectors — and this, frankly, is one of the very reasons to use an SVD method of learning.

6.2.1 Entirely Unsupervised Method

Combining these methods, and the iterative approach which we have explained above we obtain the following:

1. Initialize the evolutionary process with random networks (preferably using rank one connection matrices), or programmed networks (as we have done for the TORCS/SCRC problem) if some information about the problem is known or networks obtained from previous supervised learning.

2. Test the population of initial networks.

3. Retain the best few networks and apply crossing and mutation to their singular value decompositions to obtain the next generation.

4. Repeat testing, selection and crossing/mutation for each generation.

One major area where this basic algorithm can be optimized the most is the method of crossing and mutation. The method we have found to be the best, whether beginning with random rank one networks or with pre-programmed networks, is to apply an approach similar to the iterative approach.

When starting with the rank one networks, it is fairly clear how to do this from the examples we have already given:

1. Intitialize random first singular vector pairs and choose a first singular value greater than necessesary.

2. Combine (average) and mutate the singular vector pairs of the best performers until performance no longer improves, the singular vectors have sufficiently converged, or an iteration limit is reached.

3. Reduce the magnitude of the approximate singular value as long as doing so improves performance.

4. Now that the first singular vector pair and singular value are set, choose a second singular value approximation equal to the first, and random singular vector pairs approximately orthogonal to the first pair.

5. Repeat the process of combination/mutation for the new pairs, the process of magnitude reduction for the singular value, and continue for the rest of the singular vector pairs and values.

6. This will, as we have mentioned before, optimize the later singular triples with respect to the previous values, but further iterations or the holistic evolutionary approach is necessary to re-optimize the first singular triples with respect to the later ones.

6.2.2 Unsupervised Refinement of Previous Model

Finally rather than starting with either random or "hand-initialized" matrices, we tried initializing the unsupervised training with some of the results of the supervised training.

This followed the same method as above, and as with the supervised method, we started with a set of training tracks: Ruudskogen (fast), Aalborg (curvy) and Alpine 2 (alpine); a validation track: Wheel 1; and a set of test tracks: Forza (fast), Brondehach (curvy), Alpine 1 (alpine) and Wheel 2 (fast with hairpin).

Recall from Chapter 5 that we wished to see if we could "fine tune" the singular values of our controllers for each track. Clearly this type of fine tuning cannot be done using the same split of training, validation and test tracks that we have used. The type of fine tuning we wish to accomplish is based on running each track as quickly as possible, so that the training and testing should simply be on the same track. Rather than try to fine tune for every track and note the differences, however, we decided to attempt fine tuning for three classes of tracks: fast, curvy and alpine.

Since we knew the controllers would run on every track, our only goal was to show that fine tuning using the singular values was possible. And since we wanted only further proof that singular values could be tuned in this way, reducing the

	Hand-Coded	Fine Supervised	Unsupervised	Fine Unsup
Ruudskogen	82	82	88	87
Aalborg	84	82	88	86
Alpine 2	110	106	117	117
Wheel 1	109	105	117	116
Forza	115	112	122	123
Brondehach	100	95	105	104
Alpine 1	169	171	179	174
Wheel 2	146	145	154	151

Table 6.1: Track Times

number of parameters to tune, we used only the best performing controller for this experiment.

The results, in the Table 6.1, show an improvement in performance for the tracks within a class, with the improvement most marked in the "curvy" track class. However, we note that the controllers specially fine-tuned for the "fast" tracks obtained abyssimal performance in the "curvy" and "alpine" tracks. This was due to crashing or going off track.

It's important to note that any type of neural network may be "fine-tuned" in the manner above, when one expects similar types of responses from a certain input (similar "qualitative" responses), but with varying magnitudes. We emphasize that this is because we are maintaining an awareness of the properties of the transformation at the heart of a neural network layer by using the SVD. The singular vectors encode many of the "qualitative" aspects of a neural network response, while the singular values encode "quantitative" aspects — especially, it might be noted, when the transfer function is restricting the range of the transformation and the possibility of explosive divergence.

6.3 Compact Evolutionary SVD

This method defines a distribution for each of the singular vectors and singular values — sampling from the distributions, testing and then refining the distributions on

each iteration. Note that a selection of previous singular vectors restricts the choice of subsequent singular vectors to be orthogonal. However, we did not explicitly implement this dependance of the distributions in the experiments we conducted — instead sampling according to the normal distribution around the parameters under consideration similar to [59] and then orthogonalizing afterward. The distributions do have a tendency to orthogonalize themselves after a few iterations however.

In fact, this method can be used in either an iterative method, where one defines and allows to converge distributions for each of the singular triples in turn. Or, it may be used in a more holistic search — always keeping a distribution for each of the singular vectors and values, but searching for all of them at once.

This method performed very similarly to the iterative method when based on it, but quite poorly when used as a holistic method — unless begun with a very good initial population and very small initial standard deviations for the distributions. We presume that this is because the distributions cannot easily encode the orthogonality constraint and the synergistic effects of the singular vectors with each other in the context of recurrent neural networks. This, or a way to efficiently encode the orthogonality constraint in the distribution may be an area of future research.

Chapter 7

Concluding Remarks

In an effort to better respect the nature of the affine transformation at the heart of a neural network layer during the learning phase and to allow learning of "qualitative" structure and then reuse of the structure and fine-tuning of "quantitative" responses in similar domains, we have developed a collection of machine learning methods based on the singular value decomposition of a matrix — a way of viewing a matrix or linear transformation as a mapping from certain "important" inputs to "important" outputs and an inherent quality of the neural network layer as a transformation.

Whether one requires a method for supervised learning or unsupervised learning, for the single layer recurrent neural networks we have considered, our method is capable of producing results on par with other methods in domains as diverse as price prediction, predictive maintenance, and automatic control (as for a self-driving car).

Several methods and benefits of using the singular value decomposition have been presented. Of the methods, these include:

1. Supervised Methods

    (a) Iterative method from zero: Using the further decomposition of the SVD into a sum of rank one matrices, or the sum of outer products, we start

with a zero matrix and attempt to find the most important singular triples (two vectors and the associated singular value) in order.

(b) Iterative method from linear model: Assuming we are given a linear model for the process or function we wish to model, using the variables we have available, we can decide how many hidden variables may be useful — or simply increase the number of hidden variables until we reach the level of accuracy desired, or until further increases do not improve the model.

When we begin this method, we note that the first singular triple (the first singular value and the first pair of singular vectors) should correspond very closely to the linear model — if the transfer function is nearly linear in the range of the linear model. With a non-linear transfer function (assuming it is chosen with the appropriate range of values), more work is likely necessary — choosing a transfer function similar to the identity reduces this work.

Going forward with the method, supposing our matrix is $n \times n$, this means $3n$ parameters to train (assuming the original shift from the linear model is left unchanged) for each new singular triple. Further, the dimension of the search space is reduced for each iteration (including the "first" as the linear model provides the primary singular triple). Further, we can bound the other singular values as we like, perhaps well below the first, ensuring that our modifications are only small improvements to the linear model — not changing its regions of stability, for example.

2. Unsupervised Methods

(a) Iterative method from zero: As before, even without a particular function to model — as in our unsupervised self-driving experiment — we may think of the SVD as further decomposed into a sum of rank one

matrices, or outer products of vectors. We take a sample of possible singular vectors and values and, after constructing our connection matrix using the outer product, test our solutions. Whichever solutions perform best, we retain and perturb to obtain the next iteration of possible solutions (still of the same rank).

After these solutions converge, whether by equivalent performance on the test or convergence of the matrices, we then move on to the same type of search for the next singular triplet — always with the note that each new search for a singular vector pair must be orthogonal to all the previous singular vector pairs, therefore reducing the dimension of the search space.

(b) Compact evolutionary method: This method defines a distribution for each of the singular vectors and singular values — sampling from the distributions, testing and then refining the distributions on each iteration. Note that a selection of previous singular vectors restricts the choice of subsequent singular vectors to be orthogonal. However, we did not explicitly implement this dependance of the distributions in the experiments we conducted — instead sampling similar to [59] and then orthogonalizing afterward. The distributions do have a tendency to orthogonalize themselves after a few iterations however.

In fact, this method can be used in either an iterative method, where one defines and allows to converge the distributions for each of the singular triples in turn. Or, it may be used in a more holistic search — always keeping a distribution for each of the singular vectors and values, but searching for all of them at once.

(c) Evolutionary SVD method: As described before, this method attempts to find solutions by using the *left* singular vectors from one candidate

solution and the *right* singular vectors from another. The singular values may be averaged (using the geometric mean) and/or retrained for this network.

This "mixing" of two different candidate solutions using the singular value decomposition in this manner leads to the "strongest" inputs of one candidate solution being "connected to" the "strongest" outputs of the other in the new resulting network. That is, the first right singular vector of one matrix is associated to first left singular vector of the other, the second right singular vector of one to the second left singular vector of the other and down the line.

For completely random initializations, this kind of "mixing" or "cross-over" may not make much sense — and an iterative or even random search may be more useful. However, if the controllers have been "pre-programmed" in the manner we suggested in Chapter 6 — or have gone through an initial stage of random search — then the evolutionary SVD algorithm is effective at refining these networks.

We note again here that perhaps one of the most important elements of this method is not the evolutionary mixing, but the method of perturbation — perhaps particularly of the right and left singular vectors. That is, that we perturb the matrices U and V using rotations.

In future work we would like to examine this on its own, particularly since one could apply different degrees of rotation (different magnitudes of perturbation) to the different singular vectors — applying a relatively small perturbation to the first singular vectors and a relatively large perturbation to the last singular vectors, for example.

Clearly, the magnitude of a perturbation is extremely important in stochastic methods — to our knowledge, this is the first work to consider the

possibility of tuning the levels of perturbation to the inherent properties of the neural network transformation which are separated out by the SVD.

(d) Of particular importance for our methods is the ability to convert the method from supervised to unsupervised learning with very little change — meanwhile, the magnitude and kind of learning effected can be kept within certain bounds.

This can be done by only reoptimizing new singular values — allowing the type of network response to remain the same and allowing one to train only n (or $2n$ if the shift/bias is also trained) values rather than n^2. Or, this could be retraining only the singular triples with singular values below a certain value — e.g. limiting the learning to singular values which will not allow the solution to blow up (when the transfer function is unbounded, for example).

These combination of methods allow us to train recurrent neural networks for a variety of problems with changes through time, including price prediction, predictive maintenance and model identification, and automatic control. Our method does not rely on back propogation and can be used in either supervised or unsupervised settings. Further, our models can be "seeded" (and convergence sped up) by using either domain knowledge or (linear) least squares to "pre-program" the model to an area of the solution space likely to be most useful. And, given a neural network previously trained in one domain, they allow the reuse and quick retraining for a similar domain, by preserving the inherent structure of the transformation at the heart of the neural network.

Bibliography

[1] Sabeur Abid, Farhat Fnaiech, and Mohamed Najim. A new neural network pruning method based on the singular value decomposition and the weight initialisation. In *Signal Processing Conference, 2002 11th European*, pages 1–4. IEEE, 2002. Cited on 18

[2] Florent Altché and Arnaud de La Fortelle. Partitioning of the free space-time for on-road navigation of autonomous ground vehicles. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2126–2133. IEEE, 2017. Cited on

[3] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65, 1994. Cited on 17

[4] Christos Athanasiadis, Damianos Galanopoulos, and Anastasios Tefas. Progressive neural network training for the open racing car simulator. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 116–123. IEEE, 2012. Cited on 22

[5] Laura Balzano and Stephen J Wright. On GROUSE and incremental SVD. In *2013 5th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 1–4. IEEE, 2013. Cited on

[6] PG Benardos and G-C Vosniakos. Optimizing feedforward artificial neural network architecture. *Engineering Applications of Artificial Intelligence*, 20(3):365–382, 2007. Cited on 17

[7] Bernhard Bermeitinger, Tomas Hrycej, and Siegfried Handschuh. Singular value decomposition and neural networks. In *International Conference on Artificial Neural Networks*, pages 153–164. Springer, 2019. Cited on 19

[8] Armando Blanco, Miguel Delgado, and Maria C Pegalajar. A real-coded genetic algorithm for training recurrent neural networks. *Neural networks*, 14(1):93–105, 2001. Cited on 17

[9] Armando Blanco, Miguel Delgado, and MC Pegalajar. A genetic algorithm to obtain the optimal recurrent neural network. *International Journal of Approximate Reasoning*, 23(1):67–83, 2000. Cited on 17

[10] Matteo Botta, Vincenzo Gautieri, Daniele Loiacono, and Pier Luca Lanzi. Evolving the optimal racing line in a high-end racing game. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 108–115. IEEE, 2012. Cited on 22, 44

[11] Matthew Brand. Incremental singular value decomposition of uncertain data with missing values. In *European Conference on Computer Vision*, pages 707–720. Springer, 2002. Cited on

[12] Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear algebra and its applications*, 415(1):20–30, 2006. Cited on

[13] Martin V Butz, Matthias J Linhardt, and Thies D Lonneker. Effective racing on partially observable tracks: Indirectly coupling anticipatory egocentric sensors with motor commands. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):31–42, 2010. Cited on 62

[14] Martin V Butz, Matthias J Linhardt, and Thies D Lonneker. Effective racing on partially observable tracks: Indirectly coupling anticipatory egocentric sensors with motor commands. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):31–42, 2011. Cited on 21, 44

[15] Martin V Butz and Thies D Lonneker. Optimized sensory-motor couplings plus strategy extensions for the TORCS car racing challenge. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 317–324. IEEE, 2009. Cited on 21, 44

[16] Chenghao Cai, Dengfeng Ke, Yanyan Xu, and Kaile Su. Fast learning of deep neural networks via singular value decomposition. In *Pacific Rim International Conference on Artificial Intelligence*, pages 820–826. Springer, 2014. Cited on 19

[17] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. On-line neuroevolution applied to the open racing car simulator. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 2622–2629. IEEE, 2009. Cited on 21

[18] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Learning to drive in the open racing car simulator using online neuroevolution. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(3):176–190, 2010. Cited on 21

[19] Gustavo Carneiro, Antoni B Chan, Pedro J Moreno, and Nuno Vasconcelos. Supervised learning of semantic classes for image annotation and retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 29(3):394–410, 2007. Cited on 43

[20] Adenilson R Carvalho, Fernando M Ramos, and Antonio A Chaves. Metaheuristics for the feedforward artificial neural network (ANN) architecture optimization problem. *Neural Computing and Applications*, 20(8):1273–1284, 2011. Cited on 17

[21] Rohitash Chandra and Mengjie Zhang. Cooperative coevolution of Elman recurrent neural networks for chaotic time series prediction. *Neurocomputing*, 86:116–123, 2012. Cited on 17

[22] KyungHyun Cho and Nima Reyhani. An iterative algorithm for singular value decomposition on noisy incomplete matrices. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, 2012. Cited on

[23] Jerome T Connor, R Douglas Martin, and Les E Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994. Cited on 9

[24] Jonathan A Cox. Parameter compression of recurrent neural networks and degradation of short-term memory. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 867–872. IEEE, 2017. Cited on 19

[25] Georg Dorffner. Neural networks for time series processing. *Neural Network World*, 6:447–468, 1996. Cited on 9

[26] Chris X Edwards. SnakeOil. http://xed.ch/project/snakeoil/index.html, 2016. Cited on 22

[27] Carl JG Evertsz. Fractal geometry of financial time series. *Fractals*, 3(03):609–616, 1995. Cited on 45

[28] Oscar Fontenla-Romero, Beatriz Pérez-Sánchez, and Bertha Guijarro-Berdiñas. LANN-SVD: a non-iterative SVD-based learning algorithm for one-layer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2017. Cited on 18

[29] Giorgio Giacinto and Fabio Roli. Design of effective neural network ensembles for image classification purposes. *Image and Vision Computing*, 19(9-10):699–707, 2001. Cited on 16

[30] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear Algebra*, pages 134–151. Springer, 1971. Cited on 4

[31] Garrison W Greenwood. Training partially recurrent neural networks using evolutionary strategies. *IEEE transactions on speech and audio processing*, 5(2):192–194, 1997. Cited on 17

[32] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020. Cited on 21

[33] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, 2017. Cited on 17

[34] Wilfried Haensch, Tayfun Gokmen, and Ruchir Puri. The next generation of deep learning hardware: Analog computing. *Proceedings of the IEEE*, 107(1):108–122, 2018. Cited on 16

[35] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011. Cited on

[36] Min Han, Jianhui Xi, Shiguo Xu, and Fu-Liang Yin. Prediction of chaotic time series based on the recurrent predictor neural network. *IEEE transactions on signal processing*, 52(12):3409–3416, 2004. Cited on 20

[37] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992. Cited on 16

[38] Sui-Lau Ho, Min Xie, and Thong Ngee Goh. A comparative study of neural network and Box-Jenkins ARIMA modeling in time series prediction. *Computers & Industrial Engineering*, 42(2-4):371–375, 2002. Cited on 9, 20

[39] Roger A Horn and Charles R Johnson. *Matrix Analysis*. Cambridge University Press, 2012. Cited on 1, 4

[40] Hieu Trung Huynh and Yonggwan Won. Training single hidden layer feedforward neural networks by singular value decomposition. In *Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on*, pages 1300–1304. IEEE, 2009. Cited on 18

[41] Mohammed Amine Janati Idrissi, Hassan Ramchoun, Youssef Ghanou, and Mohamed Ettaouil. Genetic algorithm for neural network architecture optimization. In *2016 3rd International Conference on Logistics Operations Management (GOL)*, pages 1–4. IEEE, 2016. Cited on 17

[42] George William Irwin, George William Irwin, K Warwick, and Kenneth J Hunt. *Neural network applications in control*. Number 53. Iet, 1995. Cited on 16

[43] Kui Jia. Improving training of deep neural networks via singular value bounding. *CoRR*, abs/1611.06013, 2016. Cited on 18

[44] Chia-Feng Juang. A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2):997–1006, 2004. Cited on 17

[45] PP Kanjilal, PK Dey, and DN Banerjee. Reduced-size neural networks through singular value decomposition and subset selection. *Electronics Letters*, 29(17):1516–1518, 1993. Cited on 18

[46] Dulakshi SK Karunasinghe and Shie-Yui Liong. Chaotic time series prediction with a global model: Artificial neural network. *Journal of Hydrology*, 323(1-4):92–105, 2006. Cited on 20

[47] Henry Leung, Titus Lo, and Sichun Wang. Prediction of noisy chaotic time series using an optimal radial basis function neural network. *IEEE Transactions on Neural Networks*, 12(5):1163–1172, 2001. Cited on 20

[48] Moshe Levy and Sorin Solomon. Power laws are logarithmic Boltzmann laws. *International Journal of Modern Physics C*, 7(04):595–601, 1996. Cited on 45

[49] Jun Li, Xue Mei, Danil Prokhorov, and Dacheng Tao. Deep neural network for structural prediction and lane detection in traffic scene. *IEEE transactions on neural networks and learning systems*, 28(3):690–703, 2017. Cited on 22

[50] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated Car Racing Championship: Competition Software Manual. *CoRR*, abs/1304.1672, 2013. Cited on 21, 22, 58, 66

[51] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A Pelta, Martin V Butz, Thies D Lonneker, Luigi Cardamone, Diego Perez, Yago Sáez, et al. The 2009 Simulated Car Racing Championship. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):131–147, 2010. Cited on 21, 44, 62, 72

[52] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018. Cited on 17

[53] Liam P. Maguire, B Roche, T. Martin McGinnity, and LJ McDaid. Predicting a chaotic time series using a fuzzy neural network. *Information Sciences*, 112(1-4):125–136, 1998. Cited on 20

[54] Pankaj Malhotra, Vishnu TV, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. TimeNet: Pre-trained deep recurrent neural network for time series classification. *arXiv preprint arXiv:1706.08838*, 2017. Cited on 9, 10

[55] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018. Cited on 17

[56] Marco Marchesi. Megapixel size image creation using generative adversarial networks. *arXiv preprint arXiv:1706.00082*, 2017. Cited on 16

[57] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. Cited on 15

[58] Daniel K McNeill. Training RNN simulated vehicle controllers using the SVD and evolutionary algorithms. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1949–1953. IEEE, 2018. Cited on iv

[59] Ernesto Mininno, Francesco Cupertino, and David Naso. Real-valued compact genetic algorithms for embedded microcontroller optimization. *Evolutionary Computation, IEEE Transactions on*, 12:203 – 219, 05 2008. Cited on iv, 78, 81

[60] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 1969. Cited on 15

[61] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. Cited on 16

[62] Jorge Muñoz, German Gutierrez, and Araceli Sanchis. Controller for TORCS created by imitation. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 271–278. IEEE, 2009. Cited on 21

[63] Jorge Muñoz, German Gutierrez, and Araceli Sanchis. A human-like TORCS controller for the Simulated Car Racing Championship. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 473–480. IEEE, 2010. Cited on 21

[64] Jorge Muñoz, German Gutierrez, and Araceli Sanchis. Multi-objective evolution for car setup optimization. In *Computational Intelligence (UKCI), 2010 UK Workshop on*, pages 1–5. IEEE, 2010. Cited on 21

[65] Jean-François Muzy, Jean Delour, and Emmanuel Bacry. Modelling fluctuations of financial time series: from cascade process to stochastic volatility model. *The European Physical Journal B-Condensed Matter and Complex Systems*, 17(3):537–548, 2000. Cited on 45

[66] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004. Cited on 16

[67] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013. Cited on 9, 17

[68] Joshué Pérez, Vicente Milanés, and Enrique Onieva. Cascade architecture for lateral control in autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 12(1):73–82, 2011. Cited on 22

[69] Mike Preuss, Jan Quadflieg, and Günter Rudolph. TORCS sensor noise removal and multi-objective track selection for driving style adaptation. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 337–344. IEEE, 2011. Cited on 22

[70] Dimitris C Psichogios and Lyle H Ungar. SVD-NET: An algorithm that automatically selects network structure. *IEEE Transactions on Neural Networks*, 5(3):513–515, 1994. Cited on 18

[71] Jan Quadflieg, Günter Rudolph, and Mike Preuss. How costly is a good compromise: Multi-objective TORCS controller parameter optimization. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 454–460. IEEE, 2015. Cited on 22

[72] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. Cited on 15

[73] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017. Cited on 9

[74] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. Cited on 17

[75] Jose Daniel A Santos, Guilherme A Barreto, and Claudio MS Medeiros. Estimating the number of hidden neurons of the MLP using singular value decomposition and principal components analysis: a novel approach. In *Neural Networks (SBRN), 2010 Eleventh Brazilian Symposium on*, pages 19–24. IEEE, 2010. Cited on 19

[76] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. Cited on 16

[77] Jonas Sjoberg. On estimation of nonlinear black-box models: How to obtain a good initialization. In *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, pages 72–81. IEEE, 1997. Cited on 44

[78] Christopher Smith and Yaochu Jin. Evolutionary multi-objective generation of recurrent neural network ensembles for time series prediction. *Neurocomputing*, 143:302–311, 2014. Cited on 9

[79] Eu Jin Teoh, Kay Chen Tan, and Cheng Xiang. Estimating the number of hidden neurons in a feedforward network using the singular value decomposition. *IEEE Transactions on Neural Networks*, 17(6):1623–1629, 2006. Cited on 19

[80] Ruey S Tsay. *Analysis of financial time series*, volume 543. John Wiley & Sons, 2005. Cited on 45

[81] Miguel I Valls, Hubertus FC Hendrikx, Victor JF Reijgwart, Fabio V Meier, Inkyu Sa, Renaud Dubé, Abel Gawel, Mathias Bürki, and Roland Siegwart. Design of an autonomous racecar: Perception, state estimation and system integration. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 2048–2055. IEEE, 2018. Cited on 21

[82] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019. Cited on 16

[83] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. Cited on 16, 17

[84] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014. Cited on 21, 22, 63

[85] Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pages 2365–2369, 2013. Cited on 18

[86] Elias Yee and Jason Teo. Evolutionary spiking neural networks as racing car controllers. In *Hybrid Intelligent Systems (HIS), 2011 11th International Conference on*, pages 411–416. IEEE, 2011. Cited on 21, 44

[87] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. Cited on 9

[88] Jia-Shu Zhang and Xian-Ci Xiao. Predicting chaotic time series using recurrent neural network. *Chinese Physics Letters*, 17(2):88, 2000. Cited on 9

[89] Jiong Zhang, Qi Lei, and Inderjit S Dhillon. Stabilizing gradients for deep neural networks via efficient SVD parameterization. *arXiv preprint arXiv:1803.09327*, 2018. Cited on 18