Princeton University

Operations Research and Financial Engineering

Nested Models and Nonparametric LSTMs in Vision-Based Autonomous Driving and Developing an R Package for Bayesian-Optimized Deep Learning

Author:
Eddie Zhou

Advisor:
Professor Han Liu

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Operations Research and Financial Engineering
Princeton University

April 2016

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Eddie Zhou

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Eddie Zhou

Acknowledgements

First, I would like to thank my advisor, Professor Han Liu. He has been an amazing teacher and constant guiding hand as an advisor. His dedication to furthering the fields of statistics and machine learning has inspired my academic and professional pursuits. He has also helped me understand the importance of brevity alongside statistical parsimony, and is the reason you are reading 34 and not 84 pages. Additionally, I would like to thank Chenyi Chen, the original DeepDriver, for answering my countless questions about this project and others.

I would also like to thank my family. To Mom and Dad – you have both set inspiring examples for how to live your lives as students, professionals, and people. I am eternally grateful for your work in building the springboard that launched my collegiate and professional life. Steve – it's immeasurably easier to walk a road paved by someone else, and I am forever indebted to you for that.

To all my friends at Princeton – I thank each and every one of you for bringing out a different side of me, and helping me understand myself and grow. Wollack, Katz, Dingus, Ploppy, Daway, Eric, Bee and many others – you've helped define this place for me.

Finally, to Sarah – we did it. Here's to the next chapter.

Contents

Preface

This thesis is divided into three parts, each representing a project completed during the course of my senior year. It is divided as such because each project was assigned successively, conditional on completion of the previous. The first part, *Nested Statistical Models*, was assigned by Professor Liu to be a full senior thesis, but I was able to complete it earlier than expected. I sought to complete more statistics and machine learning research over the year, and Professor Liu generously offered me related work in the form of *Nonparametric LSTMs*, and later, *deepbayes*.

Nested Statistical Models is presented as a standalone scientific paper suitable for publication, written in a succinct 13 pages. *Nonparametric LSTMs* assumes knowledge of the former, and is a logical extension completed in conjunction with Chenyi Chen, ORFE GS. *deepbayes* is less closely tied to the problem of autonomous driving, and may also function as a standalone piece. All three parts are written with brevity in mind, departing from the tradition of the Princeton undergraduate thesis.

All implementation, analysis, and visualization code is available for examination and reproduction in a Github repository located at `https://github.com/edz504/thesis`.

Nested Statistical Models

Nested Statistical Models for Temporal Affordance Learning in Vision-Based Autonomous Driving

Eddie Zhou[*], Chenyi Chen[†], Han Liu[‡], and Alain Kornhauser[§]

Abstract

Most current autonomous vehicle systems largely rely on Light Detection And Ranging (LIDAR), but there is much room for development in purely vision-based systems. Recently, a DeepDriving paradigm of learning affordance indicators from input images was suggested. We extend this model by analyzing the temporal and sequential nature of the input images. To do so, we apply convolutional feature extraction and examine three problems: the usefulness of history, the lag selection problem, and the viability of complex models with history. We train recurrent, or "nested" statistical models of varying complexity, and our results demonstrate that the sequential aspect of this problem is highly important and appropriate for such recurrent statistical models.

Keywords: convolutional neural network; nested models; recurrent; autonomous driving; image recognition, regression, multivariate adaptive regressive splines.

1.1 Introduction

Predictive statistical models are formulated, trained, and used in a variety of applications, from spam filtering and recommender systems to outlier detection and image recognition. One such application that has not been thoroughly explored from a statistical standpoint is autonomous driving. Within this application, vision-based approaches typically adopt one of two paradigms: mediated perception (parsing an entire scene to make a driving decision), or behavior reflex (mapping an input image to a driving action). For a more thorough survey of these paradigms, see Chen et al. (2015). In their work, Chen et al. (2015) also propose an intermediate, "DeepDriving" paradigm that strikes a balance between the two. In their work, image representations are mapped onto a small set of "affordance indicators". These values are effectively a distillation of the driving conditions – they include values such as distances to lane markings and other vehicles, road angles, and others. A simple logic controller can then take the affordance indicator values and translate them into operation of the vehicle accordingly.

[*]Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544, USA; e-mail: edzhou@princeton.edu

[†]Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544, USA; e-mail: chenyi@princeton.edu

[‡]Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544,USA; e-mail: hanliu@princeton.edu

[§]Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544, USA; e-mail: alaink@princeton.edu

DeepDriving still relies on a model that maps an input vector $\mathbf{x} \in \mathbb{R}^n$ and outputs a vector of continuous values $\mathbf{y} \in \mathbb{R}^m$, which points to the task of statistical regression. While the problem of image classification is largely solved with the advancement of deep learning methods (namely, deep convolutional neural networks, or CNNs), our image regression problem is slightly different. The more important aspect of the data at hand is its temporal and sequential nature – in a real setting, images captured by the autonomous vehicle would be processed one after another. Then, at a realistic granularity, previous images should affect the current driving decisions. Chen et al. (2015) relied largely on the widely-used CNNs to map input images to the aforementioned affordance indicators. While powerful, CNNs do not take advantage of any temporal features in the data, as recurrent neural networks do.

Our goal for this paper, however, is not to replicate the work of Chen et al. (2015) and substitute in recurrent neural networks. Instead, we look to apply rigorous statistical analysis to investigate the role of history in predicting affordance indicator values, and subsequently understand the viability of this history in both linear and nonlinear models. We characterize our approach in three problems: the first is to rigorously determine whether past history is predictively useful at all. To do so, we build simple linear models and use p-value analysis for the relevant coefficients. Second, we look to solve the lag order selection problem, which asks the optimal number of past data to include for predictive performance optimization. We approach this second problem through model selection using stacked evaluation metrics. Finally, we examine the predictive viability of nonlinear models operating on historical, lag order selected data, hoping to show that the usefulness of history extends to more complex models.

1.1.1 Related Works

This paper largely builds off of the work of Chen et al. (2015) in validating the DeepDriving, or direct perception, paradigm. As mentioned in Section 1.1, they train a convolutional neural network with data extracted from the TORCS driving video game. The inputs are convoluted image features, while the output is a set of 14 continuous values: the affordance indicators passed to the logic controller. The output data is available via extraction from the game engine, as are the input images (which are passed through a CNN for feature extraction). Note that in our work, we also use the dataset presented in their work. Doing so allows us to investigate the temporal nature within the exact same assumptions.

We also make use of work done by Burnham and Anderson (2004), Kuha (2004), and Liew (2004) in the usage of model selection metrics with respect to the lag order selection problem. The convolutional neural network used in our feature extraction also relies on the standard structure set forth by LeCun et al. (1998). Our nonlinear work is advanced by the seminal paper on MARS, by Friedman (1991), as well as the fast heuristic added in Friedman (1993).

1.1.2 Paper Organization

The rest of this paper is organized as follows. Section 1.2 introduces details of the dataset, explains the feature extraction process, and gives a brief summary of the software used for model fitting. In Sections 1.3, 1.4, 1.5, we refine the three problems presented: deciding whether history is predictively significant, rigorously determining the lag order, and examining the viability of more complex, nonlinear models. Section 1.6 provides numerical and graphical results for the three problems, and section 1.7 contains discussion thereof.

1.2 Data Wrangling and Tools

1.2.1 Data Pipeline

The TORCS dataset exists as a leveldb, a data storage format also developed and used by Google. We use Python's `leveldb` module to connect and read from the images and values stored within it. We also need to use Python's `Caffe` interface to access the `Datum` objects stored within the database. Iterating through the leveldb, we fill `numpy` arrays of dimension 10000 x 3 x 210 x 280, and use `hickle` to save the batched arrays. The dimensions are explained as follows: the data are 3 color channel images, each image is of resolution 210 x 280, and 10000 is the largest even data size that fits in memory. Dividing the data into chunks of 10000 is also suitable for the batched aspect of stochastic gradient descent used later. Finally, we also extract the 14 affordance indicator labels (true values) for each of the 484815 images from the leveldb, and store the resulting 484815 x 14 output matrix separately.

1.2.2 Feature Extraction

We first train a convolutional neural network with LeNet architecture as first proposed by LeCun et al. (1998) for the purpose of feature extraction. We implement the network with `nolearn` by Nouri (2012), a software library that allows for flexible neural network training in the format of the widely-used `scikit-learn` Python library. `nolearn` wraps `Lasagne` by Dieleman et al. (2012), which is, in turn, an abstraction of `Theano`, by Bergstra et al. (2010).

For our feature extraction network, each response variable is scaled to [0, 1]. For a response y_{ij}, representing the ith affordance indicator of the jth data point, we scale using the minimum and maximum values over all j, holding i constant:

$$y_{ij}^{(s)} = \frac{y_{ij} - \min_j y_{ij}}{\max_j y_{ij} - \min_j y_{ij}}$$

Output scaling is done during this process in order to facilitate convergence. Data is loaded and subsequently passed through the model in the aforementioned batches of 10000. We extract the features as the last hidden layer, which has 500 nodes. Therefore, our final training matrix $\mathbf{X}^{(CNN)}$ is 484815 x 500, where one datum, or image, is represented by 500 convoluted image features.

1.2.3 Training and Testing

In Section 1.3, we use the Ordinary Least Squares implementation of the `statsmodels` package in Python, by Perktold et al. (2009), for our model training and evaluation. Then, due to an increased need for computational efficiency, in Section 1.4 and part of Section 1.5, we use the implementation of stochastic gradient descent within `scikit-learn`, a popular Python library for machine learning developed by Pedregosa et al. (2011). Finally, for one of our nonlinear models in 1.5 we use an implementation of Multivariate Adaptive Regression Splines, within `py-earth`, by Rudy (2013).

1.3 Single History Inclusion

We perform ordinary least squares regression with two model structures: one with the time t convoluted image features as input, and one with both the time t and time $t-1$ convoluted image features as input. For this analysis, we refer to the t structure as "Model 1", indicated by $\widehat{\mathbf{y}}_t$, and the $t, t-1$ structure as "Model 2", indicated by $\widetilde{\mathbf{y}}_{t,t-1}$. For both structures, we train 14 separate models, each predicting a single affordance indicator, or response variable. The two models can be expressed in least squares regression modelling as the following:

$$
\begin{aligned}
\widehat{\mathbf{y}}_t^{(a_1)} &= \boldsymbol{\beta}^{(a_1)}\mathbf{X_t} + \boldsymbol{\epsilon} \\
\widetilde{\mathbf{y}}_{t,t-1}^{(a_1)} &= \boldsymbol{\beta}_{\boldsymbol{t}}^{(a_1)}\mathbf{X_t} + \boldsymbol{\beta}_{\boldsymbol{t-1}}^{(a_1)}\mathbf{X_{t-1}} + \boldsymbol{\epsilon} \\
\widehat{\mathbf{y}}_t^{(a_2)} &= \boldsymbol{\beta}^{(a_2)}\mathbf{X_t} + \boldsymbol{\epsilon} \\
\widetilde{\mathbf{y}}_{t,t-1}^{(a_2)} &= \boldsymbol{\beta}_{\boldsymbol{t}}^{(a_2)}\mathbf{X_t} + \boldsymbol{\beta}_{\boldsymbol{t-1}}^{(a_2)}\mathbf{X_{t-1}} + \boldsymbol{\epsilon} \\
&\vdots \\
\widehat{\mathbf{y}}_t^{(a_{14})} &= \boldsymbol{\beta}^{(a_{14})}\mathbf{X_t} + \boldsymbol{\epsilon} \\
\widetilde{\mathbf{y}}_{t,t-1}^{(a_{14})} &= \boldsymbol{\beta}_{\boldsymbol{t}}^{(a_{14})}\mathbf{X_t} + \boldsymbol{\beta}_{\boldsymbol{t-1}}^{(a_{14})}\mathbf{X_{t-1}} + \boldsymbol{\epsilon}
\end{aligned}
$$

After fitting these 28 models with OLS, we will analyze the p-values for each coefficient, across all affordance indicators (500 x 14 for model 1, and 1000 x 14 for model 2). As a brief reminder on the elementary statistics of least-squares coefficients, each coefficient's p-value is the result of a t-test with the null hypothesis that the coefficient itself is zero. Therefore, finding significantly low values for the p-values corresponding to the coefficients of the past time step will demonstrate that history has predictive value within this model.

1.4 Lag Order Selection

Assuming predictive significance of past history, the next step is to decide exactly how much history is relevant – in other words, to examine the lag order selection problem. To do this, we again fit a series of simple linear models – now, however, each model corresponds to including varying amounts of history, from only the most recent image to the previous 12 images. We wrangle the 484815 x 500 $\mathbf{X}^{(CNN)}$ matrix into 12 more matrices, each corresponding to using one more previous time step. These are of dimension 484814 x 1000, 484813 x 1500, etc. Notice that we lose one more data point from the beginning of the temporal sequence with each time inclusion. We see this by noting that our first data point has no $t-1$ image to concatenate for one time step inclusion, our second data point has no $t-2$ image to concatenate for two time step inclusions, and so on. We only lose 12 data from the beginning of the sequence, however, which is negligible with respect to the magnitude of our entire dataset.

Naive OLS matrix inversion is no longer computationally viable for these larger matrices, so we use the classical method of stochastic gradient descent (SGD), with squared loss and an L2, or ridge, penalty. Due to the high dimensionality, we also use a large penalty

multiplier to push coefficients closer to 0, as well as a higher number of training epochs.

We can apply model selection between the 12 structures using three different evaluation metrics. The first two, Akaike and Bayesian information criterion (AIC and BIC, respectively), are classical relative measures for statistical models based on empirical likelihood. As likelihood-based methods, they can be computed entirely from the result of model training, eliminating the need to hold out a test set. For both criteria, lower values point to better models. Because we have 14 affordance indicators, however, we must compute "stacked" versions of the AIC and BIC. These are versions of the standard AIC and BIC that aggregate across each affordance indicator. The time inclusion can be seen as a model structure, so we want to compare these metrics for each time inclusion, not across the response variables. Standard and stacked IC formulas, as a function of the linear coefficients $\boldsymbol{\beta}$, are given as follows:

$$\begin{aligned}
&\text{AIC}(\boldsymbol{\beta}) = 2k - 2\log(\widehat{L}) \\
&\text{BIC}(\boldsymbol{\beta}) = k\log N - 2\log(\widehat{L}) \\
&\text{AIC}^{(t-s)}(\boldsymbol{\beta}^{(t-0)}, \boldsymbol{\beta}^{(t-1)}, ..., \boldsymbol{\beta}^{(t-s)}) = 2\sum_{a=1}^{14} k_a - 2\sum_{a=1}^{14} \log(\widehat{L}_a) \\
&\text{BIC}^{(t-s)}(\boldsymbol{\beta}^{(t-0)}, \boldsymbol{\beta}^{(t-1)}, ..., \boldsymbol{\beta}^{(t-s)}) = \sum_{a=1}^{14} k_a \log N_a - 2\sum_{a=1}^{14} \log(\widehat{L}_a)
\end{aligned}$$

where k is the model degree of freedom (number of non-zero predictor coefficients), and the log-likelihood is calculated as follows:

$$\begin{aligned}
\log(\widehat{L}) &= -\frac{N}{2}\log(2\pi\widehat{\sigma}^2) - \frac{SSR}{2\widehat{\sigma}^2} \\
SSR &= \sum_{i=1}^{N}(y_i - \widehat{y}_i)^2 \\
\widehat{\sigma}^2 &= \frac{SSR}{N}
\end{aligned}$$

Our third model selection metric is evaluation-based, and requires withholding a test set. The RMSE, or root-mean-square-error, is frequently used in statistics and machine learning for evaluating predictive performance. We apply an 80/20 split to our dataset, using 80% of the data to train 14 models for each time inclusion structure. Then, we can evaluate the models on the remaining 20% of the data by aggregating RMSE across all 14 affordance indicators for each time step:

$$\begin{aligned}
&RMSE(\boldsymbol{\beta}) = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2}{N}} \\
&RMSE^{(t-s)}(\boldsymbol{\beta}^{(t-0)}, \boldsymbol{\beta}^{(t-1)}, ..., \boldsymbol{\beta}^{(t-s)}) = \sqrt{\frac{\sum_{a=1}^{14}\sum_{i=1}^{N}(y_{i,a} - \widehat{y}_{i,a})^2}{\sum_{a=1}^{14} N_a}}
\end{aligned}$$

1.5 Nonlinear Models

If we solve the lag order selection problem, we would like to examine if including history provides better predictive power in models beyond simple linear models. We therefore choose to examine two nonlinear models with our selected lag order, and evaluate them on prediction metrics given above to analyze their viability. These two models are chosen for their simplicity and because they exist as simple conceptual extensions of linear models.

1.5.1 Basis Expansion

The first nonlinear model we explore is a simple basis expansion model. There is an abundance of basis expansions that are viable, with infinite polynomial operations. Because we are not concerned with feature-feature interaction and we wish to retain computational feasibility, however, we simply add a polynomial basis expansion for each response variable up to the third power.

$$\begin{aligned} \mathbf{x} &= [x_1, x_2, ..., x_k] \\ \mathbf{x}^{(expand)} &= [x_1, x_1^2, x_1^3, x_2, x_2^2, x_2^3, ..., x_k, x_k^2, x_k^3] \end{aligned}$$

Depending on the selected lag order s, the data will then have a new dimension of $1500s$, as each additional lag order provides another 500 dimensions, and the basis expansion provides another factor of 3. After expanding the basis, it becomes a matter of fitting another simple linear model with stochastic gradient descent to the new, expanded data and evaluating it.

1.5.2 Multivariate Adaptive Regressive Splines

The next nonlinear framework to explore is multivariate adaptive regressive splines (MARS), introduced by Friedman (1991). This process is a more complex nonlinear method than simply extending the basis and fitting a linear model. As a non-parametric regression, MARS models non-linearities through basis functions composed of a constant, a hinge function, or a product of multiple hinge functions. MARS training is done with a greedy forward pass that adds basis functions that give maximal reduction in SSR, followed by a backwards pass that prunes the model using generalized cross validation, which can be seen as a form of regularization.

Due to the expensive nature of the continual forward and backward passes, there are various heuristics to speed up the model. The first heuristic, proposed by the original author of MARS in Friedman (1993), limits the maximum number of parent terms considered at each step of the forward pass. It does so by cutting off the priority queue ranking of parent functions at some parameter K.

This fast heuristic is usually available in implementations of MARS, and is controlled by varying the `fast_K` parameter. In general, lower values of the parameter decrease the training time (decreasing the number of parent terms considered), but at the expense of the quality of the model. Higher values generally result in longer training times and stronger models, but random variation makes this rule general, rather than rigid, as per Milborrow (2016).

1.6 Results

1.6.1 History Inclusion

The first model simply predicts the response values as a linear function of the current time step's image features. As expected, we see low p-values for a significant number of the 500 features. For almost every affordance indicator, the median p-value (across all 500 coefficients) is less than 0.05, demonstrating that over half of the convoluted image features in each of the 14 individual models are significant at a 5% confidence level. Table 1.1 shows the distributions of these 500 p-values in the form of summary statistics for each affordance indicator.

Table 1.1: p-value summary for Model 1 $\boldsymbol{\beta}$ coefficients.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
$a1$	0	1.541e-31	1.142e-05	0.2229	0.6058	0.9716
$a2$	0	1.697e-05	0.1004	0.2926	0.6946	0.9889
$a3$	7.275e-218	1.569e-12	0.02273	0.294	0.7537	0.9964
$a4$	0	4.727e-10	0.04213	0.2813	0.6944	0.9874
$a5$	0	1.655e-14	0.009673	0.1359	0.1436	0.9978
$a6$	0	2.698e-13	0.007431	0.2	0.3722	0.9859
$a7$	0	3.849e-14	0.003059	0.1093	0.05035	0.9935
$a8$	0	3.372e-09	0.001924	0.1226	0.07164	0.9831
$a9$	0	1.47e-06	0.06613	0.2064	0.3358	0.9971
$a10$	0	9.744e-09	0.01287	0.1966	0.415	0.9866
$a11$	0	4.802e-09	0.007885	0.1157	0.07668	0.9934
$a12$	0	1.234e-10	0.005826	0.1	0.03029	0.9981
$a13$	6.051e-291	2.937e-09	0.0194	0.2836	0.7434	0.9874
$a14$	0	2.422e-07	0.00165	0.1384	0.1424	0.9953

In the second model, we input both the 500 convoluted image features of the current time step, as well as the 500 convoluted image features of the previous time step, for a total of 1000 features. In Tables 1.2 and 1.3, we see that the p-value distribution for the 500 $\boldsymbol{\beta_t}$ coefficients in Model 2 is larger than the corresponding distribution of 500 p-values in Model 1. This is intuitive, though, given the addition of 500 new features not included in Model 1 and increased model size. More features inherently means that a sufficiently parsimonious model will regard fewer features as important, and the p-value distribution will be shifted to the right compared to a smaller model. Moreover, the p-values for the $\boldsymbol{\beta_{t-1}}$ coefficients are overwhelmingly significant, which answers our question of interest. It is clear that the features from the previous time step are significant in predicting affordance indicators for the current time step – we include a graphical representation of the distribution of the $\boldsymbol{\beta_{t-1}}$ p-values for further confirmation.

Table 1.2: p-value summary for Model 2 $\boldsymbol{\beta_t}$ coefficients.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
$a1$	1.868e-287	2.627e-09	0.02677	0.2161	0.4138	0.9925
$a2$	3.211e-106	0.009934	0.3882	0.4276	0.862	0.9988
$a3$	1.271e-74	0.0003873	0.226	0.396	0.9093	0.9999
$a4$	8.369e-128	0.0004557	0.2257	0.3932	0.8869	1
$a5$	1.361e-81	0.0003228	0.04829	0.1531	0.1317	0.9982
$a6$	1.681e-85	5.213e-05	0.1444	0.1871	0.2097	0.9908
$a7$	4.051e-105	8.328e-05	0.01527	0.1447	0.1641	1
$a8$	1.943e-135	0.0001521	0.0088	0.1519	0.1899	0.9959
$a9$	4.69e-78	0.009723	0.1822	0.2368	0.3035	0.9915
$a10$	8.137e-100	0.001327	0.1711	0.2912	0.5374	0.9975
$a11$	3.133e-100	0.0004654	0.009569	0.1409	0.1303	0.9973
$a12$	1.215e-151	0.0001589	0.008946	0.1281	0.1532	0.9921
$a13$	6.864e-72	0.00198	0.2783	0.3594	0.7223	0.9886
$a14$	1.091e-116	0.0005832	0.00637	0.1561	0.2204	0.9986

Table 1.3: p-value summary for Model 2 $\boldsymbol{\beta_{t-1}}$ coefficients.

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
$a1$	0	4.004e-14	0.001513	0.1808	0.401	0.9868
$a2$	0	3.374e-05	0.0695	0.2896	0.6123	0.9954
$a3$	1.875e-123	6.232e-08	0.04029	0.292	0.6005	0.9992
$a4$	2.921e-150	4.874e-07	0.05511	0.2675	0.5239	0.9988
$a5$	2.609e-193	3.417e-08	0.03431	0.1649	0.2446	0.9952
$a6$	3.009e-183	1.626e-07	0.03167	0.1852	0.2335	0.9921
$a7$	4.245e-292	1.435e-08	0.006297	0.1501	0.1567	0.9934
$a8$	0	1.514e-06	0.005839	0.1487	0.1798	0.9977
$a9$	0	1.412e-05	0.05789	0.2005	0.2799	0.9996
$a10$	5.842e-206	4.892e-05	0.06788	0.2259	0.4497	0.9921
$a11$	2.047e-131	1.548e-05	0.01567	0.1896	0.2996	0.9976
$a12$	6.702e-140	9.075e-06	0.009431	0.1585	0.1948	0.9923
$a13$	3.324e-69	0.0001888	0.07977	0.2762	0.6316	0.9882
$a14$	3.431e-107	0.0001382	0.02132	0.1873	0.2827	0.9939

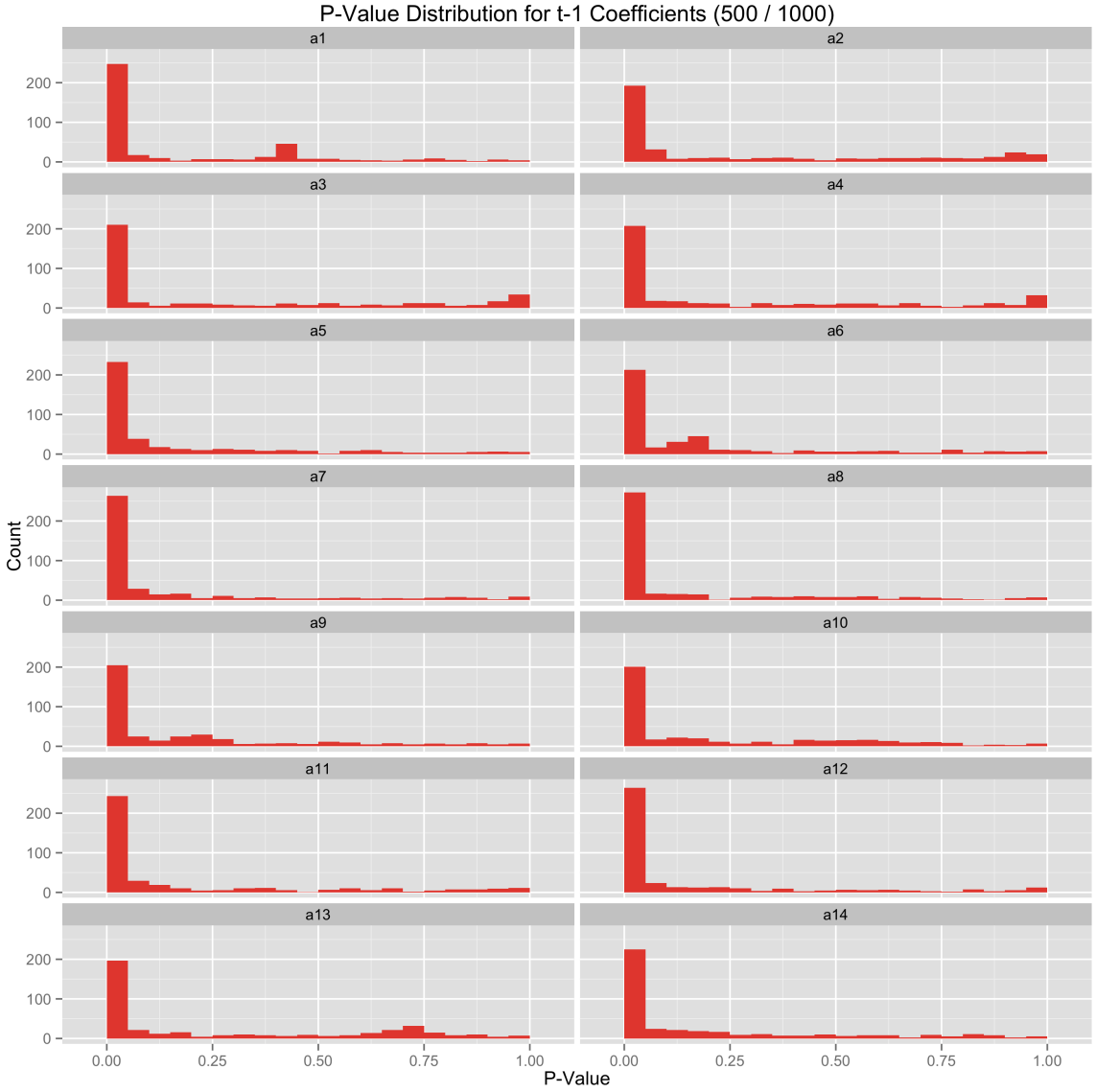

Figure 1.1: The distributions of the 500 $t-1$ step p-values in our Model 2 structure. For each of the 14 models, each corresponding to a response variable, these histograms show that nearly half of the p-values fall in the 0.05 bucket. This indicates that past history, in the form of the previous time step's convolute image features, is statistically significant for prediction within the model.

1.6.2 Lag Order Selection

As described in Section 1.4, we calculate stacked information criterion and aggregate RMSE values for each additional inclusion of history. The model selection criteria are plotted in Figures 1.2 and 1.3, along with dashed lines representing LOESS-smoothing.

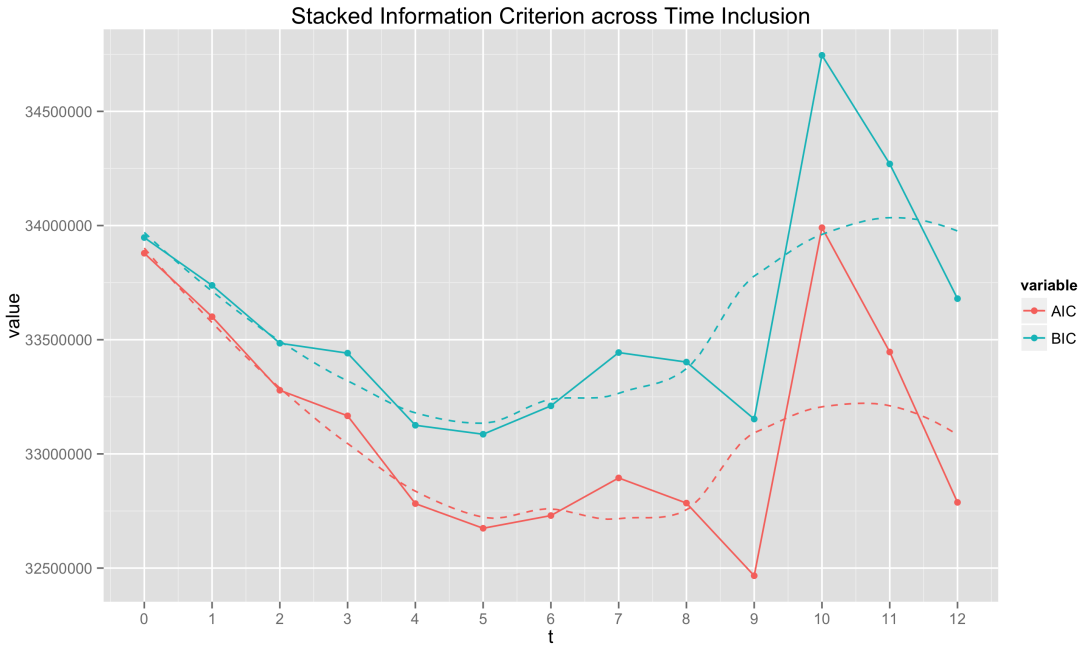

Figure 1.2: Stacked AIC and BIC for each additional time step inclusion.

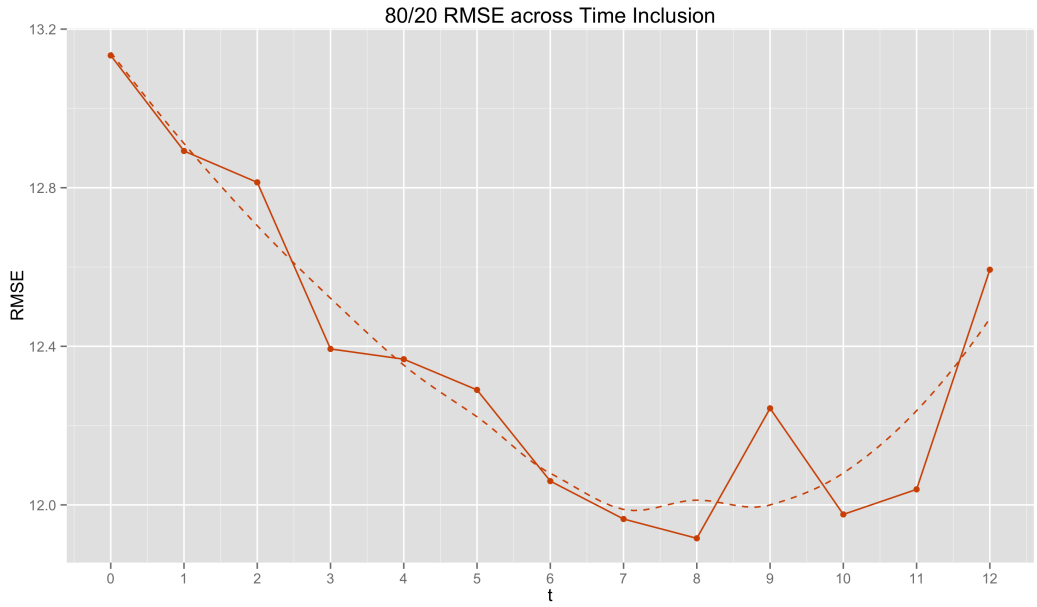

Figure 1.3: Aggregated RMSE for each additional time step inclusion.

In the stacked information criterion plot, we see a steady decrease of both AIC and BIC as we include more history, until we get to $t-5$. When we include the past 6 convoluted features, though, both of the information criterion begin to rise again, and we see much more noise. In the aggregated RMSE plot, we see a similarly convex curve that decreases until around $t-7$, and then begins rising (again, with more noise on the ascent). It is clear that there is a decrease, followed by a noisy increase, and that the smallest value lies somewhere in the aforementioned range.

The implementation of the stacked IC and the nature of RMSE dictates that smaller values point to better models. In using these metrics, we note that BIC seeks the "true" model, whereas AIC tries to select the model that most adequately describes an unknown, high dimensional reality, as described by Burnham and Anderson (2004), Kuha (2004). Moreover, in the lag order selection literature, Liew (2004) showed that AIC exists as the best lag length selection criteria. Low RMSE values also point directly to predictive power, so we move forward with the $t-7$ model structure pointed to by the LOESS-smoothed AIC and RMSE results.

1.6.3 Nonlinear Models

In both of our nonlinear models, we use RMSE for our evaluation criteria. The calculation of AIC and BIC relies on likelihood functions, which are more complex and ill-suited for nonlinear models.

Basis Expansion

For our basis-expanded linear models, we train a variety of models, varying the hyperparameter α, which controls regularization within stochastic gradient descent. Figure 1.4 shows, however, that this basis expansion does not result in a consistent decrease in RMSE over the $t-7$ linear model selected by the methodology of Section 1.4. Different values of the regularization parameter α did not succeed in achieving lower RMSEs than the best simple linear models, which were below 12.0 (see Section 1.6.2).

MARS

As described in Section 1.5, we train a variety of MARS models, varying the `fast_K` parameter. We record training time and RMSE for each of the models, to demonstrate the tradeoff between time and performance as discussed by Milborrow (2016).

As is clear from Figure 1.5, the MARS models outperform both the simple linear SGD models and the non-linear basis-expanded SGD models significantly. Even at the lowest values of `fast_K`, where the training time is just under 5 hours, the RMSE is around 11. At the highest value used, the RMSE drops all the way to just 9.91 (while training time is at a long, but not impossible 32 hours). We see a convex shape in the RMSE over `fast_K` plot, which indicates the likelihood that increasing `fast_K` gives increasingly marginal decreases in RMSE, while the training time increases linearly. This suggests that while we might achieve an even lower RMSE by increasing `fast_K`, it may not be worth the cost of the increased training time.

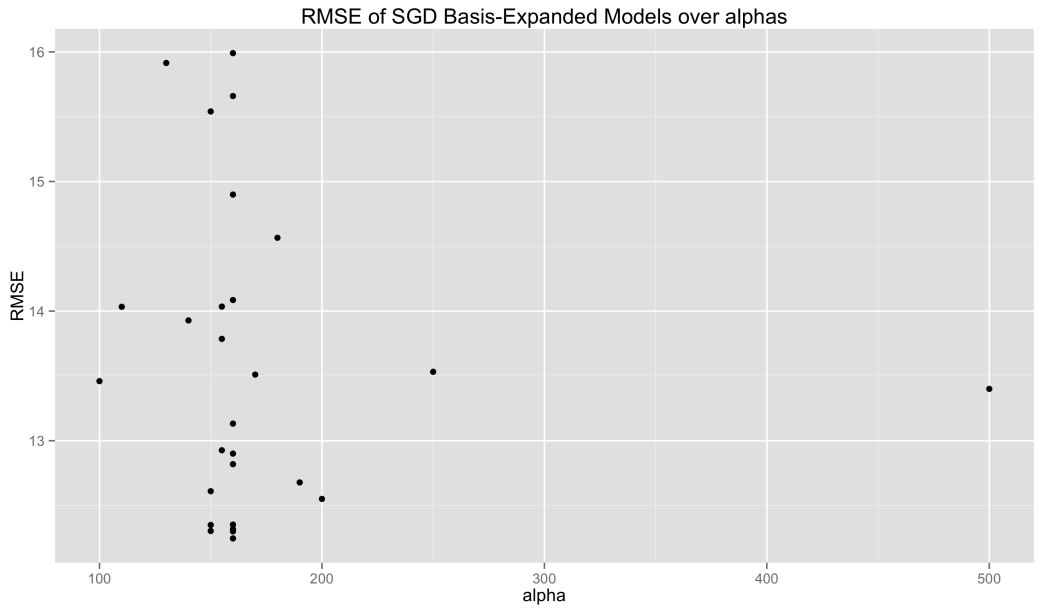

Figure 1.4: Many different values of α are used for fitting the basis-expanded model during SGD, but none of the resulting models achieve an RMSE lower than the highest-performing linear models. Moreover, different iterations of a model trained using a given α (for example, $\alpha = 160$) give largely variable RMSE scores.

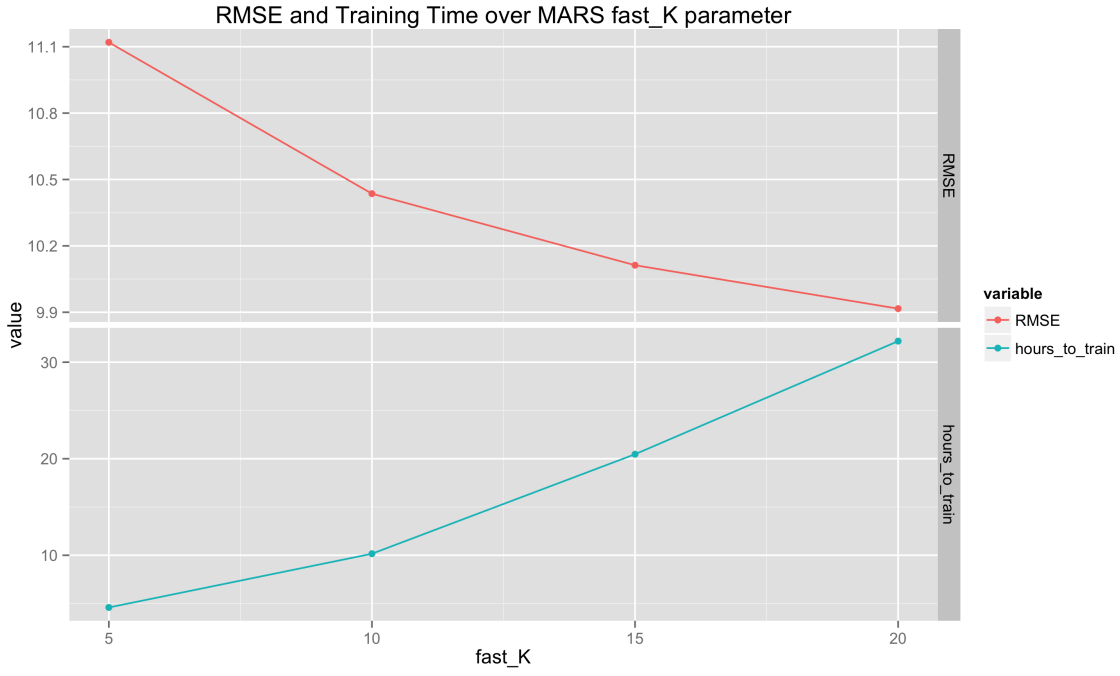

Figure 1.5: RMSE and training time of MARS models for various values of "fast K". Even the lowest values of "fast K" result in RMSE lower than the linear and basis-expanded models, while increasing "fast K" brings an even more drastic decrease (albeit at the expense of longer training times).

1.7 Discussion

Our original goal was to answer three questions, which largely relied on each other in succession. First, for the problem of applying the direct perception paradigm for autonomous driving, we asked if immediately historical data is predictively useful. We provided an unequivocally positive answer to this question by using ordinary least squares to fit simple and interpretable models for two models: Model 1, which used 500 convoluted image features for the current time step t, and Model 2, which used the 500 convoluted image features from the $t-1$ step alongside the current 500 features. We saw that the distribution of the p-values for the coefficients corresponding to the $t-1$ features in Model 2 pointed to a very high significance level, one that completely confirms the importance of history.

Our second question, conditional upon a positive answer to the first, delved deeper – we sought to rigorously determine not just if, but how much history was predictively relevant. We answered this question by using stochastic gradient descent to fit a set of models that included incrementally more steps of history, and comparing the subsequent stacked information criterion metrics. This process showed that for our TORCS data, including the most recent 7 data points gave the best performance with respect to prediction for the validation set.

Finally, we simply wanted to validate that applying more complex, nonlinear models on data that included history would improve upon the predictive power of our linear recurrent models. We fit both basis-expanded and MARS models, and found that the MARS models, which employ a similar iterative forward-backward training process to neural networks, achieved a performance significantly better than our linear recurrent models.

We were able to thoroughly answer all three of our questions with rigorous statistical analysis. Note that we do not seek to compare our highest-performing nonlinear model to Chen et al. (2015) on a performance basis. The goal for this paper was to establish confirmation of the importance and feasibility of using history and recurrency in the TORCS driving problem. Therefore, a next extension of both this work and the work of Chen et al. (2015) would make use of more advanced and powerful models as Chen et al. (2015) did, but also factor a recurrence in. This could be done either through training a state-of-the-art non-recurrent model on recurrent data, similar to our methodology (for example, applying a convolutional neural network to recurrent data), or through using an actual recurrent model (a recurrent neural network, or more specifically, a Long Short-Term Memory neural network, applied to the non-recurrent data).

Nonparametric Long Short-Term Memory Neural Networks

2.1 Introduction

2.1.1 TORCS, Direct Perception, and Recurrency

As discussed in Part 1, Chen et al. (2015) showed the feasibility of the direct perception paradigm as applied to a TORCS driving dataset. We then showed that implementing a recurrency within the data model offered predictive benefits, through a series of rigorous statistical analyses. The next step, then, is to build a recurrent model using state-of-the-art methods, and evaluate it on the TORCS dataset. To do this, we look to deep artificial neural networks, which have recently gained traction as the industry standard in machine leraning models.

While LeCun et al. (1998) and many others have shown that convolutional neural networks (CNNs or convnets) have provided massive performance gains in image recognition, recurrent neural networks (RNNs) have proved very useful for temporal and sequential data. The idea behind RNNs simply relies on activation and excitation of neuron layers without external inputs – in other words, connections between units can form directed cycles, allowing for the network to exhibit dynamic and temporal behavioral qualities. Most practical applications of RNNs use a special network structure called Long Short-Term memory, or LSTM networks.

LSTM networks have recently experienced a resurgence within machine learning applications – Google, for example, has LSTM implementations powering parts of Google Brain, applied to voice and facial recognition. LSTMs were first developed and analyzed in 1997, by Hochreiter and Schmidhuber (1997). Previous work by Hochreiter (1991) showed that in training previous recurrent models, error signals flowing backwards in time either blew up or vanished. In the first case, weights would oscillate without convergence, and in the second, the network would be unable to learn to bridge the long time lags effectively. Hochreiter and Schmidhuber (1997) proposed LSTM as a recurrent network architecture designed to overcome error-flow problems, and bridge long time intervals despite noise issues. After introducing the architecture, they conducted a series of experiments that showed the consistent high performance of LSTM models in long minimal time lag data. For the experiments, they selected arbitrary LSTM architecture (single memory cell, two memory cells, etc.) and compared them to baseline RNNs, such as RTRL (Real-Time Recurrent Learning) and BPTT (Back-Propagation Through Time). Empirical evaluation in these experiments demonstrated the potential of LSTM models in machine learning efforts.

2.1.2 LSTM in Caffe

Donahue et al. (2015) proposed a Long-Term Recurrent Convolutional Network (LRCN) that combines the strength of CNNs in visual recognition problems with stacked LSTM modules for image description tasks. They demonstrated the viability of this approach through empirical evaluation, and were generous enough to provide a pull request to the official Caffe repository adding support for RNNs and LSTMs. Chen (2016) then adapted their LSTM and RNN layer implementations to the TORCS data from Chen et al. (2015) and Part 1 of this thesis, creating multiple model structures implementing a recurrency. For our nonparametric extension, we choose one of said pipelines. The structure is included in Figure 2.6, courtesy of Chen (2016).

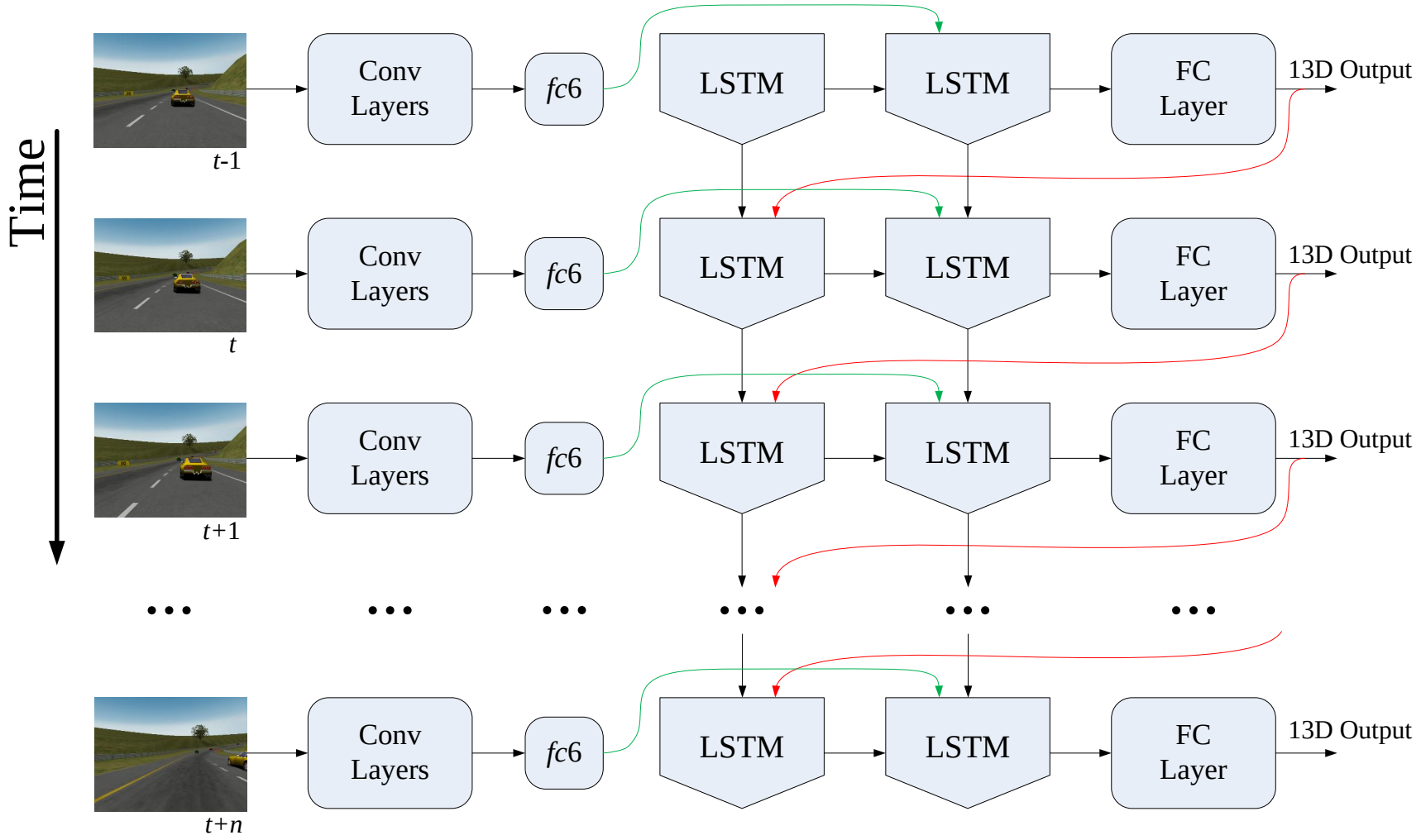

Figure 2.6: Chen's TORCS 2-Layer Skipped LSTM Structure. Images are fed into convolutional layers, followed by a fully connected activation. These, in turn, feed into the second of two LSTM modules (LSTM 2), as does the first of the LSTM modules (LSTM 1) and the LSTM 2 of the previous time step. LSTM 1 also receives input from the LSTM 1 and the affordance indicator output at the previous time step. LSTM 2 is then followed by another fully connected layer, which produces the final output response, the 13 affordance indicators. Note that in Part 1, a 14th response variable pointing to TORCS video clip endings was used, but it is omitted in the output of this model.

2.2 Methods

2.2.1 Nonparametric ReLU

As is standard within deep neural networks, there are activation layers surrounding fully connected and other layer types in the LSTM network structure of Chen (2016). The function of these layers can be abstractly thought of as the firing of neurons within the brain. Popular activation functions include the sigmoid, softmax, and rectified linear unit (ReLU) functions. The LSTM network structure we use largely contains ReLU layers for activation.

We look to implement nonparametric basis extensions for these ReLU layers, and examine the performance of the modified network. As proposed by Prof. Liu, the addition of nonparametric bases into deep neural networks increases the model exploration space. Past work has shown that this enables the network to better capture the deep structures within the data, resulting in a more powerful model. For the TORCS data and our selected LSTM structure, we create nonparametric versions of the ReLU layer by implementing additive factors for sigmoid and Fourier bases. By adding these basis curves to the activation layers, we move closer to true nonparametric modeling, which is far more flexible, while still maintaining an activation function suitable for backpropagation. The forward passes for ReLU, nonparametric sigmoid ReLU, and nonparametric Fourier ReLU are as follows:

$$
\begin{aligned}
f(\mathbf{x}) &= \max(0, \mathbf{x}) \\
f_s(\mathbf{x}) &= \max(0, \mathbf{x}) + \beta_s \frac{1}{1 + e^{-\mathbf{x}}} \\
f_F(\mathbf{x}) &= \max(0, \mathbf{x}) + \beta_F^{(1)} \sin(\mathbf{x}) + \beta_F^{(2)} \cos(\mathbf{x})
\end{aligned}
$$

To implement our new layers in Caffe, we also need to specify the backward pass computation. This pass propagates error differentials backwards from the loss at the end, allowing for each layer to update its parameters through the use of partial derivatives and the chain rule. For our new layers, we have the partial derivatives with respect to the input and the weight parameters β:

$$
\begin{aligned}
\frac{\partial f_s}{\partial \mathbf{x}} &= \mathbb{1}_{\mathbf{x}>0} + \beta_s \frac{1}{1 + e^{-\mathbf{x}}} \left(1 - \frac{1}{1 + e^{-\mathbf{x}}}\right) \\
\frac{\partial f_s}{\partial \beta_s} &= \frac{1}{1 + e^{-\mathbf{x}}} \\
\frac{\partial f_F}{\partial \mathbf{x}} &= \mathbb{1}_{\mathbf{x}>0} + \beta_F^{(1)} \cos(\mathbf{x}) - \beta_F^{(2)} \sin(\mathbf{x}) \\
\frac{\partial f_s}{\partial \beta_F^{(1)}} &= \sin(\mathbf{x}), \frac{\partial f_s}{\partial \beta_F^{(2)}} = \cos(\mathbf{x})
\end{aligned}
$$

2.2.2 Parametric Initialization

In line with past work from Prof. Liu, the training of our nonparametric deep neural network relies on a parametric initialization. In other words, we first train the normal LSTM network structure for some number of iterations, until validation loss has converged. Then, we initialize the nonparametric version with the parametric network's converged weights, and with all β basis factors initialized to 0. We then train the nonparametrically modified network for some number of iterations, until validation loss has converged.

2.3 Results

Our parametric initialization is completed after 50,000 iterations, and we re-train both sigmoid and Fourier nonparametric networks for another 50,000 iterations. The networks are trained on 50% of 10,000 of our TORCS images, with the other 50% set aside for validation, using a Eucliean loss across all 13 output variables. Figures 2.7 and 2.8 show the validation loss of training iteration of our two networks, and Figure 2.9 plots them together for comparison.

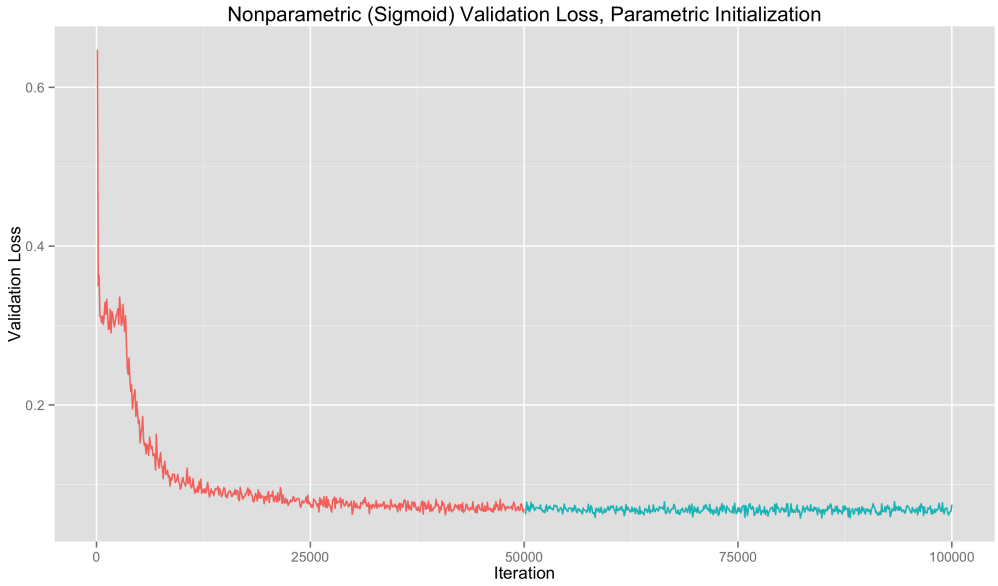

Figure 2.7: Validation loss of the parametric and sigmoid nonparametric training.

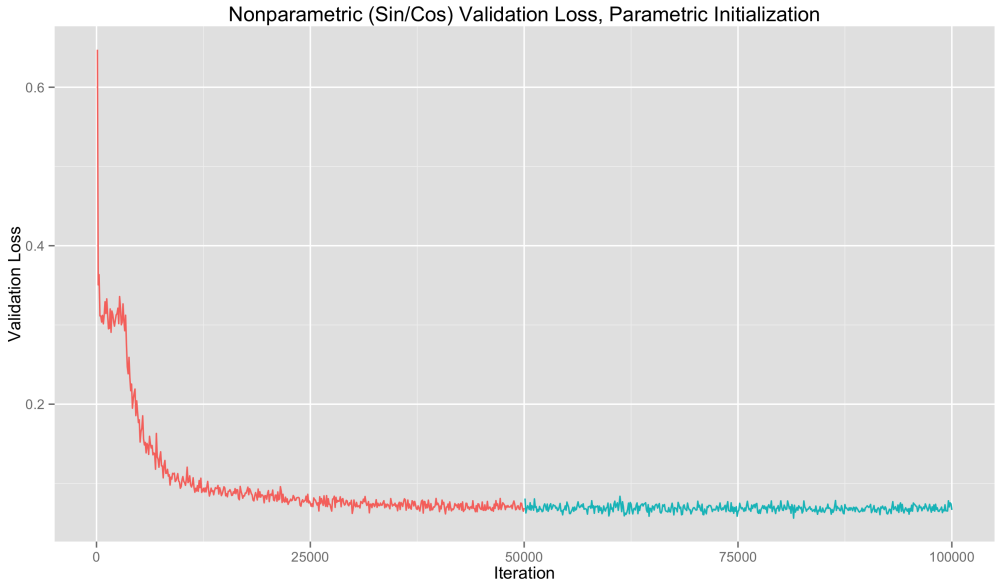

Figure 2.8: Validation loss of the parametric and Fourier nonparametric training.

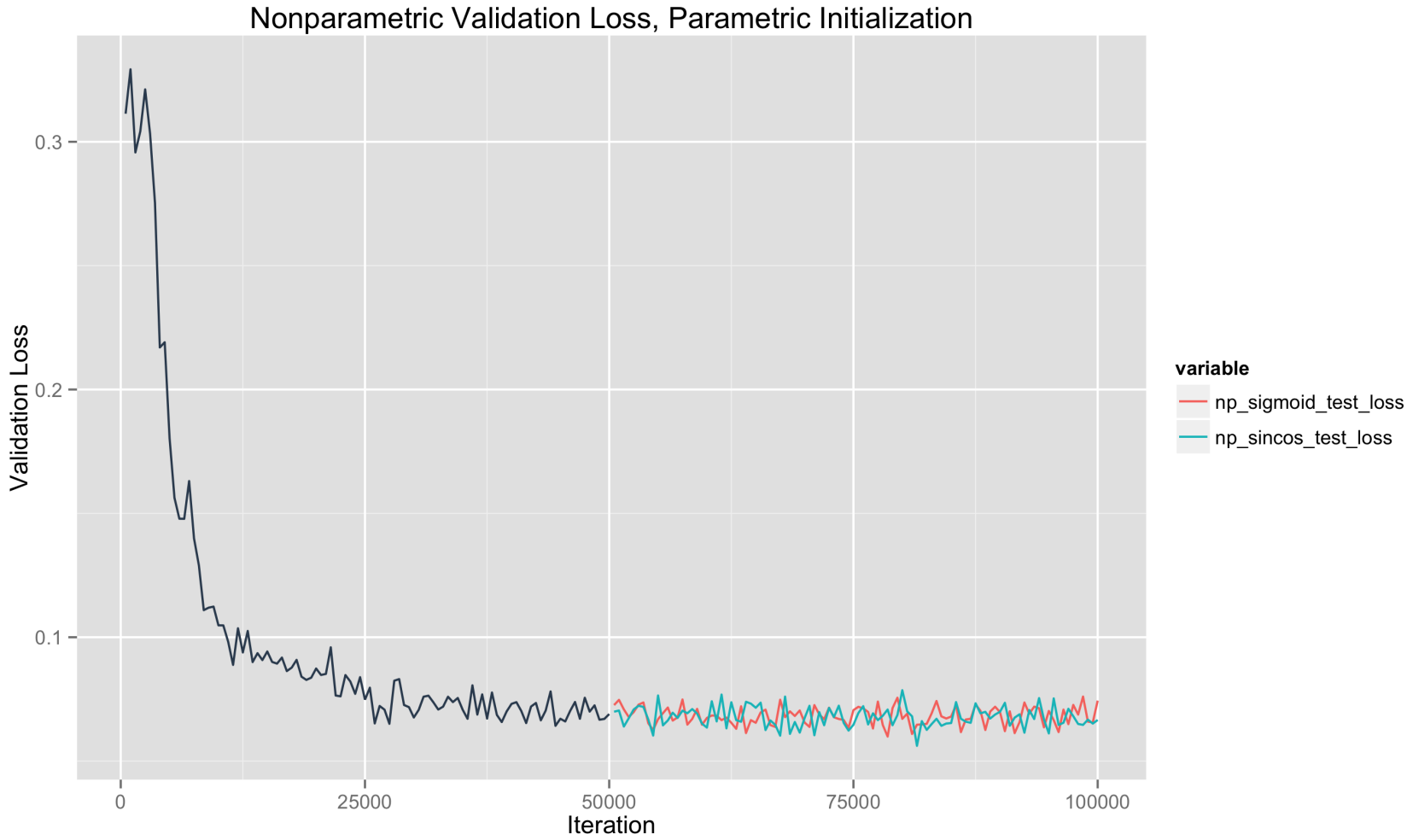

Figure 2.9: Validation loss of the parametric initialization, and both sigmoid and Fourier nonparametric training. The majority of the loss decrease is obtained during the parametric initialization, with the nonparametric networks' loss remaining around the same values as those of the parametric initialization. Neither the sigmoid nor the Fourier-based models show stronger performance than the other, both achieving validation loss convergence at a similar level.

2.4 Discussion

In past work by Professor Liu's group, this same methodology applied to the MNIST and other benchmark dataset showed an initial increase in loss when the nonparametric model was switched in, followed by a sharp decrease that dropped past the converged loss of the parametric initialization. From the individual plots above, though, we see that neither our sigmoid nor Fourier basis expansions provide significant drops in validation loss – the majority of the decrease in loss is achieved during the parametric initialization phase. The additive nonparametric factors appear to be unable to break through the loss floor established during the initialization phase. Moreover, the overlay plot shows that neither the sigmoid nor Fourier basis expansion achieves much lower loss than the other, so we cannot make strong conclusions comparing the validity of the two as applied to this data. We also note that the scale of loss here is not directly comparable to that in Part 1, *Nested Statistical Models*, because different magnitudes of training and validation set sizes are used.

As Prof. Liu notes, however, the nonparametric modified networks do not result in an overfitting and subsequent increase in validation loss, despite the largely increased model complexity. The lack of additional loss decreases simply indicates that these nonparametric expansions are not structurally beneficial for the dataset at hand. It is possible that other additive factors to the ReLU layers could be more suitable for the TORCS data, and enable a clear drop in validation loss. This points to an area of further investigation: one could explore using polynomial bases – for example, the Lagrange or Chebyshev polynomials, which both have useful properties in interpolation and representing function spaces. Any of these smooth functions could provide a nonparametric approximation for true structure within the data, allowing for a chance to better model the TORCS data, and subsequently, the vision-based autonomous driving problem.

deepbayes: An R Package for Deep Learning and Bayesian Optimization

3.1 Introduction

As the hype of artificial intelligence and deep learning has spiked within academia and the tech industry, computer scientists have made great strides in implementing widely used software libraries for deep learning. Statisticians, however, have lagged behind, with only 1 widely used package on CRAN for deep learning – `h2o`, developed by Aiello et al. (2015). While some statisticians are now using Python for scientific computing, many statisticians are still attached to R, prized for its familiarity, open-source nature, and dedicated use for statistical computing. `h2o`, the current standard for deep learning in R, is relatively easy to use, but heavily constrained to generic artificial neural networks, lacking functionality for state-of-the-art structures (convolutional, recurrent, etc.).

We look to develop a software package for R that provides an intuitive interface for statisticians, while allowing for deep flexibility with respect to network construction. If we can rely on a C-based engine that makes use of GPU computation, we will maintain ease-of-use through the scripting functionality of R, while providing computational feasibility. We also look to tie in Bayesian Optimization for hyperparameter tuning (see 3.1.1).

3.1.1 Bayesian Optimization

The goal of nonlinear optimization can be summarized as finding a global maximizer or minimizer for an objective function, where $f : \mathcal{X} \mapsto \mathbb{R}$ is some black box function taking input $\mathbf{x}$ from the domain $\mathcal{X}$ and returning a real number. Moreover, the Bayesian optimization framework allows f to take probabilistic characteristics, so there is no deterministic constraint.

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x})$$

In most practical applications, the function is unknown or does not have a closed form solution to its gradient. Any optimization method must therefore consist of evaluating the function in some capacity. One of the special extremes of this problem is the case where each function evaluation is very expensive. Of course, this applies generally – in attempting to discover the global optimum, we would like to solve the problem with fewer, rather than more, function evaluations. In the extremely expensive evaluation case, however, we must prioritize the efficiency of an optimization algorithm over other aspects (stability, etc.), as even a slightly smaller number of function evaluations could save hours or days in model development and release. These situations are where Bayesian optimization is especially valuable, with its ability to discover a reasonable estimate of the optimizer within a constrained budget.

The general idea of Bayesian optimization is to use prior and posterior updating to iteratively optimize acquisition functions and augment our decision-making. Acquisition functions map potential candidate points for evaluation to their utility, usually by striking a balance between exploitation (choosing points we are certain to be near-optimal) and exploration (choosing points we know nothing about). The framework begins with a probabilistic surrogate model, which consists of a prior belief distribution on the behavior of our unknown objective function, and an observation model that embodies the data generation. Using a regret-based version of our acquisition functions, new points are chosen and then used to update the posterior distribution. For an extensive and formal explanation of the Bayesian optimization framework, as well as several applications and evaluations, please see the thorough work by Shahriari et al. (2015).

3.1.2 Technical Choices

`Rcpp`

We look to funnel the power and speed of C/C++ into an interface for R, noting that we must interface with external C/C++ libraries (rather than simply write C functions from scratch). We therefore cannot simply use the trivial `.C` calls from R, due to our need to link with complicated library structures and header files. We use `Rcpp`, an R package developed by Eddelbuettel and François (2011) for this exact purpose. It has been used to create thousands of R packages, and is well-documented with knowledgable authors.

`Caffe`

Jia et al. (2014) released `Caffe` a few years ago, enabling researchers to make use of powerful and flexible deep learning framework. Under the umbrella of the Berkeley Vision and Learning Center, it is especially suited for image recognition and has strong support for convolutional neural networks. Due to the design choices, however, it is very easy to add other layers (as shown in Part 2, *Nonparametric LSTMs*), making it a strong candidate for our use. Its best selling point, however, is that it is self-contained and written in C++ , allowing us to link it using `Rcpp` (as opposed to embedding with Theano in Python, etc.).

Training in `Caffe` requires specifying two files: a network and a solver. These both come in the widely-used protocol buffer format, a data structure proposed and championed by Google. A network file is constructed by specifing data, transformation, neuron, loss, and other types of layers, forming a directed acylic graph. The solver file specifies other hyperparameters for training and testing, such as learning rate policies and training iterations. To create a new layer in C++ , one merely needs to specify a forward and backward computation, and the modularity of `Caffe` takes care of the rest. There is already, however, a multitude of layers implemented within `Caffe`, representing state-of-the-art advances in convolutional and recurrent neural networks. Therefore, using a `Caffe`-based system allows for researchers to experiment with the latest network types or even develop their own layers.

BayesOpt

Bayesian optimization has recently garnered lots of attention, but there are few software libraries with proper documentation. We choose `BayesOpt`, a C++ library developed by Martinez-Cantin (2015) with straightforward C and C++ APIs. To optimize a function in `BayesOpt`, one simply needs to provide a function signature fitting into the provided template, and specify parameters for a continuous or discrete domain.

3.2 Package Usage

As of the writing of this thesis, `deepbayes` is not yet a production-ready CRAN package. There are a variety of small improvements and touch-ups left, as we have laid the groundwork for finishing off the software. In this section, we will outline how the package is implemented, and how to use it. Please see the Github repository for source code.

3.2.1 Data

An easy example to follow along with is given in the file `max_iter_example.R`, which contains `max_iter_example()`, an R function that demonstrates how one would implement a deep network tuning procedure. First, we generate data and split it into folds with functions from `data_util.R`. For most cases, users will likely have their own dataset – they can follow the simulation methods and wrangle their data into HDF5 format (the easiest data format for interchange between R and `Caffe`). The `.h5` file can be located anywhere, but should be accompanied by a `.txt` file in the same directory and with the same file prefix; this text file is necessary due to the way `Caffe` interfaces with HDF5 data. The simulation and splitting methods in `data_util.R` write to `datasets/` within the package directory.

3.2.2 Creating Network and Solver Protocol Buffers

The next part requires the most user implementation. Depending on the hyperparameter tuning, net and solver `.prototxt` files should be generated programmatically. In our example, this is done in `create_protos_max_iter()`. It first pulls in a network template within the `nets/` folder in the package directory, and creates a new proto for each individual network. In this example, the hyperparameter to be tuned is specified in the solver file, so we only need to create k network protos. We then need to create kn solver files: for each of the n values of `max_iter` in consideration, we need to create k protos that point to each of the k folds of the dataset. Note that the paths in the network protocol buffers must point correctly to the data at hand. These protocol buffers themselves must be located within the installed package directory, as those created by `create_protos_max_iter()` do.

3.2.3 Optimizing

Finally, we simply call the `Rcpp`-exported `optimize()` function, providing the hyperparameter matrix, number of cross-validation folds, an evaluation budget, a vector of parameter prefixes to retrieve the solver files, and some system paths. This function has been generalized to optimize cross-validation error over all hyperparameters provided by feeding `cv_error_function()` to the BayesOpt API.

3.2.4 Other Examples

In `max_iter_example()`, we tune a hyperparameter that is specified and contained within the solver file. Users may also wish to tune other meta-parameters such as the structure of the network itself. We outline such an example in `num_node_example()`, where all solver-based hyperparameters are held constant; we take a network with one hidden layer and tune the number of nodes within that layer. Note that because the number of training iterations is held constant, the optimum is somewhat artificially induced, as larger networks should require more training iterations before convergence.

We also provide a multi-dimensional tuning example, in `iter_layer_example()`, to demonstrate the capabilities of Bayesian optimization with respect to multiple variables, as well as to deal with the relation between model complexity and training iterations. This example simultaneously tunes the number of fully connected layers and the number of training iterations. Note that the protocol buffer file creation becomes complex when dealing with network structure rather than pure hyperparameters, so abstraction is ideal for future development (see Section 3.3.2).

3.3 Discussion and Future Work

In this section, we discuss improvements necessary for `deepbayes` to become published on CRAN, so that a researcher in the future has a documented start to finishing and refining the package.

3.3.1 External vs. Internal Libraries

Currently, the R package as developed functions assuming that the external C++ libraries `Caffe` and `BayesOpt` are installed on the user's machine. The general practice in R package development is to only make this assumption for very popular or widely-used external libraries, such as `boost`. While `Caffe` is gaining in popularity, it is not reasonable to assume that a given user has it installed – moreso for the lesser known `BayesOpt` library. It is therefore preferable – and likely necessary for CRAN-publication – that `Caffe` and `BayesOpt` are included internally with the source code of `deepbayes`. The `Makevars` file is configured with respect to the Princeton SMILE machine and its installations; requiring users to modify this source to enable installation of `deepbayes` is unwieldy and detrimental to ease of use.

3.3.2 Abstraction of Protocol Buffer Creation

While the optimization has been elegantly abstracted and generalized for the user, the creation of protocol buffer files for both the networks and solvers remains somewhat ungainly. Ideally, the user would like to avoid having to directly modify files within R to change network structure and hyperparameter values. If a `deepbayes` API could provide a generalized process for taking in user input for hyperparameter variation, and properly hiding the file creation, then the implementation required from the user drops drastically. In other words, the statistician could then spend more time experimenting and tuning their networks to their data, rather than implementing file manipulation methods.

3.3.3 Bayesian Optimization Parameters

Because it is highly desirable to hide the core C++ away from the R user, we also wish to have an R-side process for specifying parameters for the `BayesOpt` API call. Currently, the only mutable parameter is the budget, even though `BayesOpt` provides a wealth of options for priors, kernels, and much more. This is simply a matter of passing a dictionary-like object from R and parsing it into the `bayesopt::Parameters` object.

3.4 Conclusion

We have laid the groundwork for `deepbayes`, an R package that interfaces with the modular deep learning library `Caffe`, as well as `BayesOpt`, a Bayesian optimization library. `deepbayes` allows R users flexibility in developing and experimenting with their own network, while providing an efficient framework for hyperparameter tuning. All source code for `deepbayes` is available on Github (`https://github.com/edz504/thesis/tree/master/project3`) as mentioned in the Preface, allowing for further development. Several additions and refinements are necessary to make `deepbayes` publishable on CRAN, which would allow for widespread adoption by the statistics community and allow for more statistical approaches to deep learning problems in a field currently dominated by computer scientists.

Bibliography

Aiello, S., Kraljevic, T. and Maj, P. (2015). *Package h2o*.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Burnham, K. P. and Anderson, D. R. (2004). Multimodel inference: Understanding aic and bic in model selection. *Sociologal Methods and Research* **33** 261–304.

Chen, C. (2016). *Algorithmic Approaches to Extracting Cognition out of Images for the Purpose of Autonomous Driving*. Ph.D. thesis, Princeton University.

Chen, C., Seff, A., Kornhauser, A. and Xiao., J. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of 15th IEEE International Conference on Computer Vision*.

Dieleman, S., Raffel, C., Olson, E., Schlter, J. and Snderby, S. K. (2012). lasagne. `https://github.com/Lasagne/Lasagne`.

Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K. and Darrell, T. (2015). Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*.

Eddelbuettel, D. and François, R. (2011). Rcpp: Seamless R and C++ integration. *Journal of Statistical Software* **40** 1–18.
URL `http://www.jstatsoft.org/v40/i08/`

Friedman, J. H. (1991). Multivariate adaptive regression splines. *The Annals of Statistics* **19** 1–67.

Friedman, J. H. (1993). Fast mars. Tech. rep., Department of Statistics, Stanford University.

Hochreiter, J. (1991). *Untersuchungen zu dynamischen neuronalen Netzen*. Ph.D. thesis, Institut fur Informatik Technische Universitat Munchen.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation* **9** 1735–1780.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* .

Kuha, J. (2004). Aic and bic: Comparisons of assumptions and performance. *Sociologal Methods and Research* **33** 188–229.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*.

Liew, V. K. (2004). Which lag length selection criteria should we employ. *Economics Bulletin* **3** 1–9.

Martinez-Cantin, R. (2015). Bayesopt. https://github.com/rmcantin/bayesopt.

Milborrow, S. (2016). *Package earth*.

Nouri, D. (2012). nolearn. https://github.com/dnouri/nolearn.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* **12** 2825–2830.

Perktold, J., Seabold, S. and Taylor, J. (2009). statsmodels. http://www.statsmodels.org/stable/index.html.

Rudy, J. (2013). py-earth. https://github.com/scikit-learn-contrib/py-earth.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P. and de Freitas, N. (2015). Taking the human out of the loop: A review of bayesian optimization. IEEE.