UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Nonlinear Modeling of the Early and Mid-Level Visual System

Permalink
https://escholarship.org/uc/item/99p169h0

Author
Oliver, Michael David

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Nonlinear Modeling of the Early and Mid-Level Visual System

by

Michael David Oliver

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Vision Science

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:

Prof. Jack Gallant, Chair
Prof. Frédéric Theunissen
Prof. Bruno Olshausen

Fall 2014

Nonlinear Modeling of the Early and Mid-Level Visual System

© 2014

by Michael David Oliver

Abstract

Nonlinear Modeling of the Early and Mid-Level Visual System

by

Michael David Oliver

Doctor of Philosophy in Vision Science

University of California, Berkeley

Prof. Jack Gallant, Chair

The work presented in this thesis is toward the goal of extracting structure and meaning from neuroscientific data. Data in visual neuroscience is becoming increasingly high dimensional and the stimulus-response relationships can be highly nonlinear. Data in visual neuroscience is also somewhat noisy due to the imprecise separation of signals from multiple neurons on an electrode, nonstationary effects in the brain, and inherent noise in the brain; neurons rarely respond identically to identical stimuli. Finding nonlinear relationships between a high dimensional stimulus and neural responses in the presence of substantial noise is a challenging nonlinear regression problem. This thesis presents effective techniques for solving this problem and creating highly predictive models of neural function. I first introduce linearized regression, a technique for modeling nonlinear responses using linear regression on a nonlinear transformation of the stimulus. Next I demonstrate a method for efficiently finding Volterra series representations of nonlinear neural responses. Finally, I demonstrate that deep neural networks can provide accurate and interpretable models of the neural computations in visual cortex.

Table of Contents

Table of Contents i
List of figures v
Acknowledgements vii
Prologue 1
Chapter 1 5
1.1 Introduction 5
1.2 Background 6
1.2.1 Generalized Linear Models (GLM) 6
1.2.2 The Spike Triggered Average 7
1.2.3 Linearized models 8
1.2.4 Fitting a GLM with regularization 10
1.2.4.1 L2-penalties and the Gaussian prior 10
1.2.4.2 L1-penalties and the Laplacian prior 11
1.2.4.3 Other penalties 12
1.2.4.4 Gradient descent 13
1.2.5 Fitting a GLM: resampling 18
1.2.5.1 Cross-validation (K-fold) 19
1.2.5.2 Bootstrapping 19
1.2.5.3 Jackknifing 19
1.3. STRFlab overview and usage examples 20
1.3.1 Availability and requirements 20
1.3.2 Preprocessing 20
1.3.3 Model specification 23
1.3.4 Resampling 24
1.3.4.1 Cross-validation 24
1.3.4.2 Bootstrapping 25
1.3.4.3 Jackknifing 25
1.3.5 Globalizing data and grouping samples 26
1.3.6 Selecting a Fitting Algorithm 27
1.3.7 Indexing the data and starting the algorithm 28
1.3.8 Finalizing the model 29

1.3.9 Visualizing results 31
1.3.10 Fitting output nonlinearities 31
1.3.11 Prediction and validation 34
1.4 Conclusions 35
Chapter 2 38
2.1 Introduction 38
2.2 The Volterra series 39
2.3 Kernel regression for Volterra estimation 40
2.4 Support vectors and neural networks 41
2.5 Kernel networks as boosted learners 43
2.5.1 Increasing learner diversity 44
2.6 Noise models, robust error functions and output nonlinearities 44
2.7 Extracting Volterra coefficients 51
2.8 Model selection, validation and comparison 52
2.9 Testing on synthetic chemical reaction data 54
2.10 Testing on receptive field data 57
2.10.1 Simulated V1 complex cells 57
2.10.2 Visualizing receptive field models 58
2.10.3 Cat Area 17 complex cells 60
2.10.3.1 Input basis choice 60
2.10.3.2 Model performance 61
2.10.4 Macaque V4 neuron 64
2.10.4.1 Interpreting the model 64
2.11 Relationship to previous work 66
2.12 Conclusions 68
Chapter 3 70
3.1 Introduction 70
3.2 Network architecture 71
3.2.1 Rectified linear hidden units 71
3.2.2 Time-delay neural networks 73
3.3 Fitting deep neural networks 75
3.3.1 Stochastic gradient descent 75
3.3.1.1 Rprop and RMSprop 76

3.3.1.2 Momentum ... 77
3.3.1.3 Nesterov momentum ... 77
3.4 Regularizing deep neural networks ... 78
3.4.1 Bayesian regularization ... 78
3.4.2 Dropout ... 81
3.4.3 DropConnect ... 81
3.4.4 Max-norm regularization ... 83
3.4.5 Multiplicative Gaussian noise ... 83
3.4.6 Poisson noise ... 84
3.4.7 Poisson noise + Dropout ... 86
3.4.8 Output gain ... 88
3.5 Neural Network Inference ... 89
3.5.1 Dropout ... 89
3.5.2 DropConnect ... 89
3.5.3 Poisson Hidden Units ... 90
3.6 Effect of Depth on Model Performance ... 90
3.7 Visualizing feature selectivity of V1 and V2 ... 91
3.7.1 V1 features ... 92
3.7.2 V2 features ... 92
3.8 Comparison to linearized models ... 95
3.9 Future directions ... 97
3.9.1 Smooth-MAX / Smooth-MIN and soft logical operators ... 97
3.10 Conclusions ... 99
Chapter 4 ... 101
4.1 Introduction ... 101
4.2 Foveal V1/V2 data and models ... 102
4.3 Drifting gratings ... 104
4.4 Spike Triggered Average / Spike Triggered Covariance ... 107
4.5 Local Spectral Reverse Correlation ... 110
4.6 Local orientation tuning ... 114
4.7 Angle selectivity ... 118
4.8 Conclusions ... 120
Chapter 5 ... 122

5.1 Summary of contributions 122
5.2 Open questions and future directions 123
5.3 Closing remarks 124
Bibliography 125

List of figures

Chapter 1

Figure 1.1: Example of a Linearized model 9
Figure 1.2: Early stopping of three paths on a two dimensional error surface 14
Figure 1.3: Comparison of paths through a two dimensional weight space 15
Figure 1.4: Regularized Regression in STRFlab 17
Figure 1.5: STRFlab Flow Chart 21
Figure 1.6: STRF visualizations 32
Figure 1.7: Output Nonlinearities 34

Chapter 2

Figure 2.1: Diagram of kernel network 42
Figure 2.2: Illustration of stochastic gradient descent in Volterra space 43
Figure 2.3: Behavior of various error functions 45
Figure 2.4: Comparison of exponential, soft-rectification, and rectification 47
Figure 2.5: Selection of stopping point in boosting 53
Figure 2.6: Stopping point performance on the validation… 53
Figure 2.7: Prediction performance of Volterra models on chemical data 56
Figure 2.8: Filters used for simulated complex cells 58
Figure 2.9: Second order pixel interaction matrix 59
Figure 2.10: Eigenvectors of second order terms of Volterra models… 60
Figure 2.11: Comparison of prediction performance of 2nd and 3rd order… 62
Figure 2.12: Visualizing Volterra series models of cat V1 neurons 63
Figure 2.13: Prediction performance of various model orders on a V4… 65
Figure 2.14: Selectivity of the model of a V4 neuron 66
Figure 2.15: Nonlinear color tuning in the model of a V4 neuron 67

Chapter 3

Figure 3.1: Common activation functions and their derivatives 73
Figure 3.2: Time-delay architectures 74
Figure 3.3: Assessing the improvement in neural network generalization… 80
Figure 3.4: DropOut and DropConnect on time-delay neural networks 82
Figure 3.5: Standard schematic of linear-nonlinear-Poisson (LNP) model 85
Figure 3.6: Effects of regularization on learned filter weights 87
Figure 3.7: Comparing different forms of regularization 88
Figure 3.8: Comparing performance of neural networks… 91
Figure 3.9: Visualization of V1 models 93
Figure 3.10: Visualization of V2 models 94
Figure 3.11: Comparison of prediction performance… 96

Chapter 4

Figure 4.1: Receptive field locations in foveal V1/V2 103
Figure 4.2: Preferred orientation of neural networks models of V1/V2 104
Figure 4.3: Preferred spatial frequency of neural networks models of V1/V2 105
Figure 4.4: Preferred temporal frequency of neural networks models of V1… 105
Figure 4.5: F1/F0 ratio of neural networks models of V1/V2 106
Figure 4.6: Spike Triggered Average (STA) and Spike Triggered Covariance… 107
Figure 4.7: Spike Triggered Average (STA) and Spike Triggered Covariance… 108
Figure 4.8: Distribution of significant excitatory eigenvectors obtained… 109
Figure 4.9: Distribution of significant inhibitory eigenvectors obtained… 109
Figure 4.10: LSRC of V1 neuron models 111
Figure 4.11: LSRC of V2 neuron models 112
Figure 4.12: Maximum orientation selectivity differences within V1 and V2… 113
Figure 4.13: Maximum orientation selectivity differences within V1 and V2… 114
Figure 4.14: Local orientation tuning of V1 models 116
Figure 4.15: Local orientation tuning of V2 models 117
Figure 4.16: Distribution of angle selectivity in models of V2 neurons 118
Figure 4.17: Angle selectivity of V2 models 119

Acknowledgements

This dissertation was made possible by the generous support of the Vision Science Graduate Group, NIH and the NSF Center for the Science of Information.

The support, advice, mentoring and high expectations of my advisor, Jack Gallant, were also crucially important. The Gallant lab has been an amazing environment in which to do science; Jack gives his lab freedom to pursue interesting, challenging and risky projects. I have been surrounded by amazing colleagues from whom I have learned so much. Thomas Naselaris and Shinji Nishimoto taught me almost everything I know about primate electrophysiology. Ryan Prenger inspired me to learn everything I could about machine learning. I've also learned tons (and had a lot of fun) just bouncing ideas off of Dustin Stansbury, Alex Huth, Mark Lescroart, Anwar Nunez, Natalia Bilenko, James Gao and Michael Schachter.

I would like to thank my quals committee: Karen DeValois, Richard Ivry, Frédéric Theunissen and Michael Silver for their time, advice and feedback. I thoroughly enjoyed my conversations with each of them and was continually inspired by their insight and wealth of knowledge. I also thank my dissertation committee: Bruno Olshausen and Frédéric Theunissen for taking the time to read and give feedback on this dissertation. Beyond that, each of them has contributed greatly to my development as a scientist.

I thank the many friends and colleagues who've touched my life and contributed to the person I am today. I have been lucky to know so many wonderful people.

I would also like to thank Kavitha Ratnam for her love, patience and support as I wrote this dissertation. The joy she adds to my life has made the writing process as close to enjoyable as possible.

And finally, I would like to thank my parents for their love and support throughout my life. Since childhood they've fostered my curiosity and encouraged me to pursue my interests. None of this would have happened without them.

(I should also acknowledge my cats Leonid, Cosmia and Ragna for keeping my lap warm during all night writing sessions.)

Prologue

Like many of my peers, I came to study neuroscience from decidedly philosophical bent. I was interested in high-level questions such as consciousness, qualia, and the like, i.e. "the hard problem", or "how is it that a collection of particles can convince itself of having had an experience?" The tools of philosophy, however, seemed hopelessly inadequate and more likely than not to get entangled in rhetorical gimmicks and arguments that hinged on conflicts over the definitions of words. I wanted the ability to not just make a better argument, but to actually prove propositions about the mind right or wrong, to test ideas with empirical evidence.

Of course I quickly discovered how little we can say with any real assurance about the connection between the mind and the brain. While clearly causally related, a precise description of the nature and mechanics of this relationship is elusive. To make progress on a neuroscientific understanding of consciousness, I believed (and still do) we must understand how the objects of conscious awareness are represented in the brain, i.e. we must understand the internal language of consciousness in terms of neural processes. However, I found that our understanding of the neural representations of words and concepts and feelings, i.e. the things with which we consciously interact, is vague at best. In fact our understanding of the neural representations of sensory stimuli beyond the early sensory areas is vague at best.

But scientific understanding requires modeling the world with sufficient precision to make accurate predictions, i.e. quantifying causal relationships in testable ways. The area of neuroscience that seemed best at adhering to this dictum was early sensory physiology. I was in particular attracted to visual neurophysiology, due to both the complexity and phenomenal immediacy of the visual world. In this field, scientists made mathematical models of the behavior of neurons given certain stimuli; attempts were being made to quantify how the external world was represented in the brain. And, to be clear, I don't mean *where* certain things are represented, I mean the algorithmic transforms that lead to a given representation: the *how* and *why* particular features of the external world are represented. This seemed to me a convincing path toward progress.

It is, however, not an easy path. To facilitate the quantification of data and the development of models, many scientists have made use of simplifying assumptions inspired by engineering. For many years the stimuli used in visual neurophysiology experiments was almost exclusively dominated by sinusoidal gratings, white noise and other simple parameterized stimuli. Using complex, unparameterized stimuli such as natural images bordered on heresy. But, while relatively successful in visual

area V1, simple stimuli failed to adequately characterize later visual areas. Furthermore, it was not obvious how well models generated using simple stimuli would generalize to other stimulus domains. The complexity of the visual system could not be forever swept under the rug.

It was the appreciation and articulation of these ideas that attracted me to Jack Gallant's laboratory. What had been vague feelings of discontent were given precise expression. And, more importantly, I was given a workable framework for making progress. This framework can be summarized as applying machine learning algorithms to rich neuroscientific data. Our goal is to collect the most general data we can by sampling the relevant stimulus space as widely as possible, and then use statistical algorithms to extract structure from this data. In essence, we want to induce how the brain works from data, rather than deduce how the brain works by testing a series of hypotheses. Given our meager understanding, the space of possible hypotheses regarding neural computation is too dauntingly large to be assured of real progress.

The work presented in this thesis is toward the goal of extracting structure and meaning from neuroscientific data. Data in visual neuroscience is becoming increasingly high dimensional and the stimulus-response relationships can be highly nonlinear. Data in visual neuroscience is also somewhat noisy due to the imprecise separation of signals from multiple neurons on an electrode, nonstationary effects in the brain, and inherent noise in the brain; neurons rarely respond identically to identical stimuli. Finding nonlinear relationships between a high dimensional stimulus and neural responses in the presence of substantial noise is a challenging nonlinear regression problem.

In Chapter 1 I cover the bread and butter of our lab, linearized regression, and the software package I maintain for performing linearized regression: STRFlab. Linearized regression transforms a nonlinear regression problem into a linear regression problem by means of a nonlinear feature space. Regularizing the linear regression, however, is crucial and nontrivial. I have spent much time investigating various regularization strategies, as well as coming up with various extensions, and have incorporated the ones that proved most useful into STRFlab.

In Chapter 2 I introduce a method fitting the nonlinearities rather than assuming them outright. This work extends the standard spike triggered average and spike triggered covariance analyses to a more general and powerful Volterra series modeling framework by exploiting connections between neural networks and polynomial kernel regression.

In Chapter 3 I make use of recent developments in regularizing neural networks, as well as creating a new method, to fit deep time-delay neural network models of receptive fields in V1 and V2. As deep neural networks are universal approximators, this method makes no strong assumptions about neural response properties and allows the data to determine the model in a relatively unbiased way. This work was greatly facilitated by the unprecedented amount of data that Utah electrode arrays allowed me to collect. The resultant models are highly predictive and can be investigated to yield new insights into V1 and V2.

In Chapter 4 I take the models obtained in Chapter 3 and perform a series of *in silico* experiments on them that are similar to previous *in vivo* studies of V1 and V2. The goal is not only to replicate past results, but to allow past results to be connected to each other and interpreted with respect to each other. Since many previous studies can be replicated on the model of a single neuron we can get a sense for what each previous experiment does and doesn't reveal about neural function. I also present a way to interpret the neural networks themselves that reveals additional interesting insights into neural computation.

Chapter 1

Modeling Receptive Fields with STRFlab

Chapter 1

Modeling Receptive Fields with STRFlab

1.1 Introduction

A central goal of sensory neuroscience is to produce computational models that describe the functional relationship between sensory inputs and neuronal responses. These models generally take the form of a quantitative description of the spatiotemporal / spectrotemporal receptive field (STRF) of a neuron (P. Marmarelis 1978; Theunissen et al. 2001; David, Vinje, and Gallant 2004). Computing the STRF of a neuron provides a formal characterization of the stimulus features for which it is selective (Jones and Palmer 1987). Most STRF analyses can be cast in terms of regression analyses in which the neuronal response is treated as the dependent variable and the stimulus is treated as a set of independent variables (Theunissen et al. 2001). The STRF for a neuron is constructed by finding the appropriate relationship between the independent variables (stimulus) and the dependent variable (response). This can be an exceedingly difficult nonlinear regression problem since the relationship between the stimulus and the neuronal response is often highly nonlinear (Prenger et al. 2004; Rust et al. 2005; Touryan, Felsen, and Dan 2005), neuronal responses are noisy (Stein 1967), and only limited data can be collected due to experimental constraints. Successfully performing such nonlinear regression analyses can require significant experience and expertise.

To allow a wide range of researchers convenient access to state-of-the-art STRF estimation techniques I introduce STRFlab, a MATLAB toolbox that incorporates the most popular and successful approaches developed for estimating STRFs under a linearized regression framework. The STRFlab regression framework transforms a *nonlinear* regression problem into a *linear* regression problem by performing a nonlinear transform on the stimulus. The nonlinear transform decomposes stimuli into a set of nonlinear features the system is hypothesized to represent. So rather than finding a nonlinear relationship between the stimulus and neural response, one can find a linear relationship between nonlinear features of the stimulus and the neural response. The nonlinear transform of the stimulus is thus considered a linearizing transform (David, Vinje, and Gallant 2004). An appropriate choice of linearizing transform is crucial and depends on the system under investigation. STRFlab includes several linearizing stimulus transforms (and more can easily added by the user) to enable tests of a wide range of hypotheses about the underlying computations carried out by neurons. STRFlab allows alternative STRF models to be quickly fit to neural data, validated via prediction on a separate data set, visualized and quantitatively compared. STRFlab thus provides researchers with

an easy to use, powerful and extensible regression toolbox adapted to the demands of STRF modeling.

1.2 Background

1.2.1 Generalized Linear Models (GLM)

The most commonly used STRF analyses are based upon the generalized linear model (GLM) (Nelder and Wedderburn 1972). A GLM based STRF can be written in matrix notation as $f(L(S)\widehat{w}) = \hat{r}$. In this equation, $L(S)$ is a nonlinear transform of the raw stimulus S, and has dimensions *temporal samples* by *stimulus channels*. *Temporal samples* is the number of discrete stimulus samples, e.g. number of movie frames. *Stimulus channels* is the number of stimulus features produced by the transformation L. $\widehat{w}$ is a vector of regression weights on the stimulus channels and has dimensions *stimulus channels* by 1. $\hat{r}$ is the system's predicted response and has dimensions *temporal samples* by 1. f is the function that relates the linearly weighted stimulus channels, $L(S)\widehat{w}$ **,** to the predicted mean response, $\hat{r}$. This function is called the mean function, and for convenience it is usually chosen as the inverse of the canonical link function of hypothesized response distribution (see Table 1.1). (The canonical link function is the function that expresses the canonical parameter θ of the exponential family in terms of the mean μ of the response distribution (Nelder and Wedderburn 1972).) However, model performance is often improved by using non-canonical mean functions, also called output nonlinearities, which can be chosen based on prior knowledge or even fit to the data.

Because there is typically a delay between the presentation of a stimulus and the response of the system, the transformed stimulus matrix will typically contain several temporally shifted copies of each stimulus channel, i.e. it is a Toeplitz matrix. Examining the weight vector $\hat{h}$ reveals which stimulus channels at which temporal delays are important for driving responses in the system.

There are two somewhat conflicting goals when fitting the GLM: 1) finding a weight vector $\widehat{w}$ such that the error between the estimated response $\hat{r}$ and the system's actual response r is minimized and 2) finding a weight vector $\widehat{w}$ that accurately predicts responses to new stimuli. When data is limited and contains noise, there is naturally a tradeoff between how well a model fits a given data set (since $\widehat{w}$ can potentially overfit noise in the data) and how well that model can generalize. The goal then is to find $\widehat{w}$ that fits the data without over-fitting to the noise. Attempting to optimize this tradeoff is known as regularization, which I will discuss after describing the most common GLMs used in STRF estimation.

Table 1.1. Regression Assumptions

Type of Regression	Mean Function	Response Distribution
Linear	$\hat{r} = L(S)\widehat{w}$	Gaussian
Logistic	$\hat{r} = 1/\left(1 + e^{-L(S)\widehat{w}}\right)$	Binomial
Poisson	$\hat{r} = e^{L(S)\widehat{w}}$	Poisson

1.2.2 The Spike Triggered Average

The Spike Triggered Average (STA) is a special case of the GLM that has become the most commonly used STRF analysis technique (de Boer and Kuyper 1968; Jones and Palmer 1987; Rust et al. 2005). The STA is simply the average of the stimulus values that preceded a neuronal spike. In the context of GLMs, the STA is simply a method for solving for the weight vector $\widehat{w}$ of a GLM when the stimulus channels are raw stimulus values (i.e. there is no nonlinear transform) and the mean function is the identity function. Because there is no linearizing transform applied to the stimulus, the STA attempts to find a linear relationship between the stimulus and the response. The concept of the STA can be extended to encompass any neural responses other than spikes, e.g. intracellular voltage (Theunissen et al. 2001). The generalized STA is simply the average of the stimuli S weighted by the system responses r. In matrix notation, the generalized STA can be written:

$$\widehat{w}_{STA} = \frac{S^T r}{N} \qquad [1.1]$$

When the responses are spikes, this average is identical to the traditional STA. But, because I aim for STRFlab to be a general STRF estimation package for both spiking and continuously valued systems, I will discuss only the generalized STA going forward.

The STA assumes each stimulus channel has equal variance and that there are no second-order correlations in the stimulus, i.e. the values of one stimulus channel are not correlated with the values of another stimulus channel. In this case the stimulus covariance matrix is N times the identity matrix. Gaussian white noise stimuli fulfill this requirement and are thus often used for STRF estimation. If the STA is used with non-white stimuli, i.e. natural images, the STA estimate will be biased by the

correlations present in the stimuli. In order to perform an unbiased STA analysis using natural stimuli, a corrected STA (cSTA) has been devised (David, Vinje, and Gallant 2004; Theunissen et al. 2001). The cSTA removes the effect of second-order correlations within the stimuli by multiplying by the inverse of the stimulus covariance matrix.

Mathematically the cSTA can be written:

$$\widehat{w}_{cSTA} = (S^T S)^{-1} S^T r \qquad [1.2]$$

The $S^T r$ term weights the stimuli by the responses and is identical to the STA estimator above. The inverse of the stimulus covariance matrix, $(S^T S)^{-1}$, corrects the estimate for both the magnitude of the stimulus and any correlations between the pixel values. When the cSTA is used with Gaussian white noise stimuli the $(S^T S)^{-1}$ term becomes the inverse of N times the identity matrix, or simply $1/N$, making the cSTA equivalent to the STA. However, when the cSTA is used with high dimensional natural stimuli, the $(S^T S)^{-1}$ term can make this estimator more difficult to calculate because $S^T S$ may not be invertible (it is degenerate) or inversion may not be computationally tractable due to memory and time constraints. For this reason, the fitting routines in STRFlab use fast iterative algorithms that obviate the need for this matrix inversion.

1.2.3 Linearized models

Because the cSTA attempts to find a linear relationship between stimulus and response, its use is limited to systems where a strong linear relationship exists. However, many brain areas exhibit highly nonlinear relationships between stimulus and response, and for these areas cSTA fails as an analysis technique. To model highly nonlinear systems, the stimuli can be transformed into a set of nonlinear features the system is hypothesized to represent. A linear relationship between these nonlinear features and the systems response can then be found. The stimulus has thus been linearized with respect to the responses by the nonlinear transform.

By making reasonable assumptions about the nonlinearities that exist within a system, one can construct powerful nonlinear models that can be fit with a limited amount of data. For example, when modeling the visual system, one can project image pixel values onto a set of rectified Gabor-wavelet functions to create a set of features that may describe the response characteristics of visual neurons. Or when constructing a model of the auditory system, one can project the auditory stimulus into the time-frequency domain (by calculating the square of a short-time Fourier transform) to create a set of frequency-power features that may describe the

response characteristics of auditory neurons. The set of all features used in a model constitutes its 'feature space'. Using this modeling framework, many candidate nonlinear feature spaces can be tested for their ability to model a sensory area. Mathematically, the linearized model can be written as:

$$L(S)\hat{w} = \hat{r} \qquad [1.3]$$

In this equation, L is a nonlinear function of the raw stimulus S that transforms the stimulus into a set of nonlinear features, $\hat{w}$ is a vector of regression weights on the nonlinear features channels, and $\hat{r}$ is the predicted response of the system. The solution for the weight vector of the linearized model has the same form as the cSTA:

$$\hat{w}_L = (L(S)^T L(S))^{-1} L(S)^T r \qquad [1.4]$$

The above solution for the weights in a linearized regression problem is equivalent to finding the weight values that minimize the Least Squares error function.

$$Error = \sum_{i=1}^{N} (r_i - L(S_i)\hat{w}_L)^2 \qquad [1.5]$$

This Least Squares error function, however, only measures how well the weight vector fits the data and gives no way to regularize the solution.

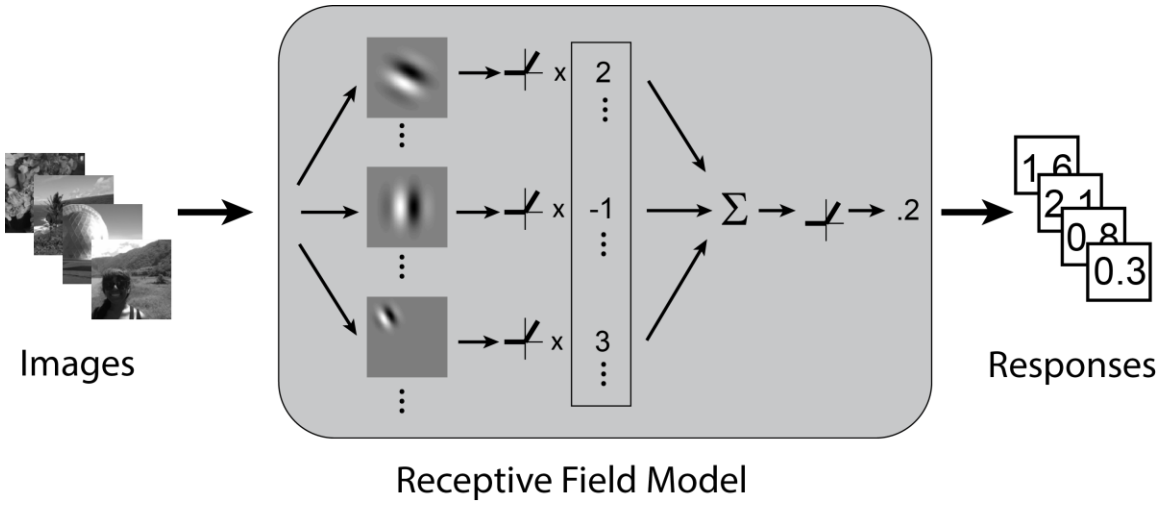


Figure 1.1: Example of a Linearized model.

Images are projected onto a set of rectified Gabor filters, which constitute the feature space of this model. Weights are fit to these projections by regularized linear regression. The weighted sum of all the projections is passed through an output nonlinearity to predict the output of the system.

1.2.4 Fitting a GLM with regularization

Regularization refers to methods for preventing the weight vector $\hat{w}$ from being fit to noise in the data. Regularization results in more robust models that better generalize to new data sets. Regularization can also yield models that are more interpretable. For example, regularization that enforces sparseness in the values of $\hat{w}$, i.e. most values are forced to 0, would yield a model with fewer weights to interpret.

Regularization in regression is typically implemented one of two ways:

1) Adding a weighted penalty on the values of $\hat{w}$ to the Least Squares error function:

$$Error = \sum_{i=1}^{N} (r_i - L(S_i)\hat{w})^2 + \lambda \, Penalty(\hat{w}) \qquad [1.6]$$

2) A heuristic rule, such as a threshold on the values of $\hat{w}$.

From the point of view of Bayesian statistics, different kinds of regularization take the form of different prior distributions over model weights: $P(\hat{w})$. Penalties on the values of $\hat{w}$ are generally easy to describe in terms of a closed form prior distribution, $P(\hat{w})$. But heuristic rules can often achieve similar performance in terms of prediction and interpretability and are often easier to implement (and can be computationally cheaper). STRFlab includes fast and efficient fitting routines implementing the penalties and heuristics described below.

1.2.4.1 L2-penalties and the Gaussian prior

One of the most commonly used penalties added to the Least Squares error function is a penalty on the L2-norm of the weight vector (equation 1.7). The L2-penalty regularization technique is also known as "ridge regression" and is a special case of a more general regularization technique known as Tikhonov regularization (equation 1.8). Ridge regression is equivalent to putting a spherical zero-mean Gaussian prior on the weight vector; i.e. the weight vector parameters are assumed to be independent and come from a zero-mean Gaussian distribution with equal variance in each dimension. This is equivalent to saying the inverse covariance matrix of the Gaussian prior is a scalar multiple of the identity matrix: $\Sigma^{-1} = \lambda I$.

Tikhonov regularization is the more general case because it allows the definition of a matrix Γ which can create arbitrary structure in the inverse covariance matrix of the Gaussian prior: $\Sigma^{-1} = \Gamma^T\Gamma$. Tikhonov regularization thus enables the ability to create dependencies between parameters. For example, if Γ is defined as a Laplacian

operator, the differences between adjacent parameters are penalized, which enforces explicit "smoothness" in the solution. (Note that a Laplacian operator or any other form of Γ can be multiplied by a scalar factor to change the strength of the prior, just as λ changes the strength of the prior in ridge regression.)

However, it should also be noted that when $\Gamma = \lambda I$ and Tikhonov regularization is equivalent to Ridge regularization, the Gaussian prior still leads to "smoother" solutions than unregularized least-squares. This is because λ controls how much S is decorrelated before being projected onto r. The effect of λ is most easily intuited at the limit where λ is very large. In this case the identity matrix I dominates $(S^T S + \lambda I)^{-1}$, which causes it to effectively disappear. The weight vector is then determined by independently projecting each regressor in S onto r: $\hat{w} = S^T r$. This would clearly lead to correlated regressors receiving similar weights. λ thus controls the extent to which correlated regressors are given similar weights. Higher λ values result in "smoother" solutions since adjacent regressors, e.g. adjacent pixel values in a natural movie, are highly correlated.

One potential advantage of these techniques is that they possess a closed form solution to estimate $\hat{w}$ (equations 1.9 and 1.10) which means that if the inversion is computationally tractable, many values of λ or forms of Γ can be tested rapidly to determine what works best to regularize the solution.

$$Error = \sum_{i=1}^{N} (r_i - S_i \hat{w})^2 + \lambda \|\hat{w}\|_2^2 \qquad [1.7]$$

$$Error = \sum_{i=1}^{N} (r_i - S_i \hat{w})^2 + \|\Gamma \hat{w}\|_2^2 \qquad [1.8]$$

$$\hat{w} = (S^T S + \lambda I)^{-1} S^T r \qquad [1.9]$$

$$\hat{w} = (S^T S + \Gamma^T \Gamma)^{-1} S^T r \qquad [1.10]$$

1.2.4.2 L1-penalties and the Laplacian prior

A more recently introduced regularization technique is the use of a penalty on the L1-norm of the weight vector (equation 1.11). This penalty is called the "Least

Absolute Shrinkage and Selection Operator", abbreviated LASSO, because it penalizes the sum of the absolute values of the weight vector and "selects" regressors by assigning non-zero values to only a subset of the weights (Tibshirani 1996). The L1-norm penalty thus induces sparseness in the estimate of $\widehat{w}$. Sparseness is desirable if it is believed that only a subset of the regressors contribute to the response. This penalty is equivalent to putting a zero-mean Laplacian prior on the weight vector; i.e. the weight vector parameters are assumed to be independent and come from a zero-mean Laplace distribution (also known as double exponential distribution) with equal variance in each dimension. The Laplace distribution is highly peaked at its mean, in this case zero, which accounts for this prior's sparsity inducing behavior.

Unfortunately there are no closed form solutions for L1-penalized regression. There are, however, a number of efficient iterative algorithms that compute a series of solutions for $\widehat{w}$ across a range of values of λ. Generally the algorithms start with a high value for λ, such that only one regressor is included in the solution, and then λ is gradually decreased causing more and more regressors to be included in the solution (Efron et al. 2004; Friedman, Hastie, and Tibshirani 2010a). The full path of solutions is thus obtained as a consequence of solving for any particular value of λ. Consequently, it is easy to test a range of λ values for generalization performance. In the context of STRF fitting, sparse solutions are often desirable because a neurons have localized receptive fields that can usually be described by a small number of features, such as wavelets (Willmore, Prenger, and Gallant 2010; Nishimoto and Gallant 2011).

$$Error = \sum_{i=1}^{N} (r_i - S_i\widehat{w})^2 + \lambda\|\widehat{w}\|_1 \qquad [1.11]$$

1.2.4.3 Other penalties

It has been observed that the L1-norm penalty can sometimes over-sparsify solutions and has a tendency to only select one of a set of highly correlated regressors. This behavior can negatively impact generalization performance. Since I described in 1.2.4.1 how the L2-norm penalty can control how correlated regressors enter the solution, it seems natural to combine both penalties to obtain solutions that are both sparse and smooth. This hybrid penalty (equation 1.12), is known as "Elastic Net" regularization (Zou and Hastie 2005). Elastic net regularization typically finds solutions that generalize better than either LASSO or ridge regression, but at the cost of two hyperparameters, λ_1 and λ_2, that must be optimized rather than just one.

$$Error = \sum_{i=1}^{N} (r_i - S_i\hat{w})^2 + \lambda_1\|\hat{w}\|_1 + \lambda_2\|\hat{w}\|_2^2 \qquad [1.12]$$

Another variant on the L1-norm penalty is the penalty in equation 1.13, which is called the group-LASSO (Friedman, Hastie, and Tibshirani 2010b; Yuan and Lin 2006). This penalty sparsifies with respect to predefined groups or subsets of parameters $\hat{w}_l$. It is easy to see that if each group only contained one parameter, the penalty term would be equivalent to summing over the absolute values of the elements of $\hat{w}$, which is the L1-norm. The group-LASSO is useful if it is known that certain groups of parameters should be used together or not at all. In the context of STRF fitting, weights on the same feature at different time-delays would constitute a reasonable choice of a group since the group-LASSO it would then enforce sparsity in space but not in time.

$$Error = \sum_{i=1}^{N} (r_i - S_i\hat{w})^2 + \lambda \sum_{l=1}^{L} \|\hat{w}_l\|_2 \qquad [1.13]$$

A more exotic penalty is the "Trace-norm" or "Nuclear-norm" penalty. After reformatting the weight vector $\hat{w}$ into a matrix of size features x delays, the penalty term is calculated as $\|\hat{w}\|_* = trace\left(\sqrt{\hat{w}^T\hat{w}}\right)$. The trace-norm is the sum of the singular values of $\hat{w}$, which is used as a proxy for the rank of $\hat{w}$. Penalizing the trace norm thus results in lower rank solutions for $\hat{w}$. Lower rank solutions require less data and have been found to improve the generalization ability of neural receptive field models (David, Vinje, and Gallant 2004; Park and Pillow 2013). At the limit when $\hat{w}$ is rank-1, all the feature weights follow the same temporal pattern across delays resulting in a model that is feature-time separable (David, Vinje, and Gallant 2004). Because the trace-norm penalty does not set an explicit constraint on the rank of the solution, the rank of the solution will depend on both the data and the value of λ.

$$Error = \sum_{i=1}^{N} (r_i - S_i\hat{w})^2 + \lambda\|\hat{w}\|_* \qquad [1.14]$$

1.2.4.4 Gradient descent

Gradient descent is an iterative algorithm for finding values of $\hat{w}$ that minimize the error function. Gradient descent starts with some initial values of $\hat{w}$ (usually all 0's) and finds the gradient (i.e. slope) of the error function at that location in parameter space. The gradient indicates how a change in the value of a given weight in $\hat{w}$ will

increase or decrease the error function. Thus by adjusting each parameter in a direction that decreases the error, i.e. the negative of the gradient, the weight vector $\hat{w}$ moves toward values that will minimize the error function. Equation 1.15 gives the basic equation for gradient descent in which the current weight vector, $\hat{w}^t$, is updated by taking a small step of size, η, in the negative direction of the gradient, $\nabla E(\hat{w}^t)$. Equation 1.16 is the gradient of the least squares error function with respect to $\hat{w}$.

$$\hat{w}^{t+1} = \hat{w}^t - \eta \nabla E(\hat{w}^t) \qquad [1.15]$$

$$\nabla E(\hat{w}) = -S^T(r - S\hat{w}) \qquad [1.16]$$

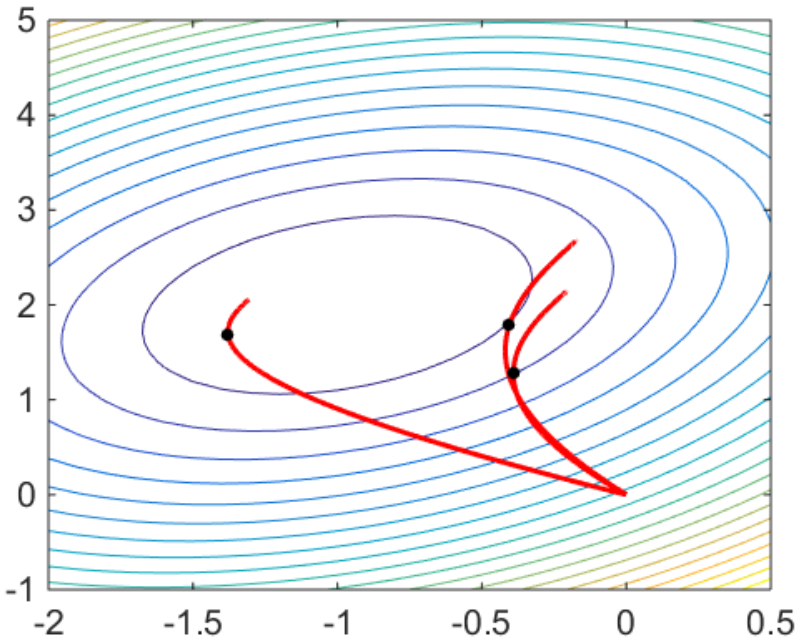


Figure 1.2: Early stopping of three paths on a two dimensional error surface

Three simulated data sets were created by adding noise to the response of a linear model. Gradient descent was performed starting at (0,0) to determine the weights and the paths through weight space are shown in red. The error surface is different for each path due to the addition of noise which in turn causes their trajectories to be different. All paths are plotted on the noiseless error surface whose minimum is at the true weight values. At every iteration the error is evaluated on separate testing data (that also contains noise) to determine the early stopping point. The points marked by black dots are the iterations where early stopping would halt the descent.

Because gradient descent moves toward, and will eventually end up at, the unregularized least squares solution, it is not immediately clear how to add regularization. One simple way to way to regularize gradient descent is with "early stopping" (Zhang and Yu 2005; Bühlmann and Yu 2006). Early stopping works by checking the error of the current model on a separate data set, the "stopping set", at every iteration of gradient descent, and then halting when the error on the stopping set begins to increase consistently. This works because the early stages of gradient descent make large changes in the directions most important for reducing error, i.e. directions for which the data provides a strong signal, while thereafter the descent path is more likely to be affected by noise as it converges to the least squares

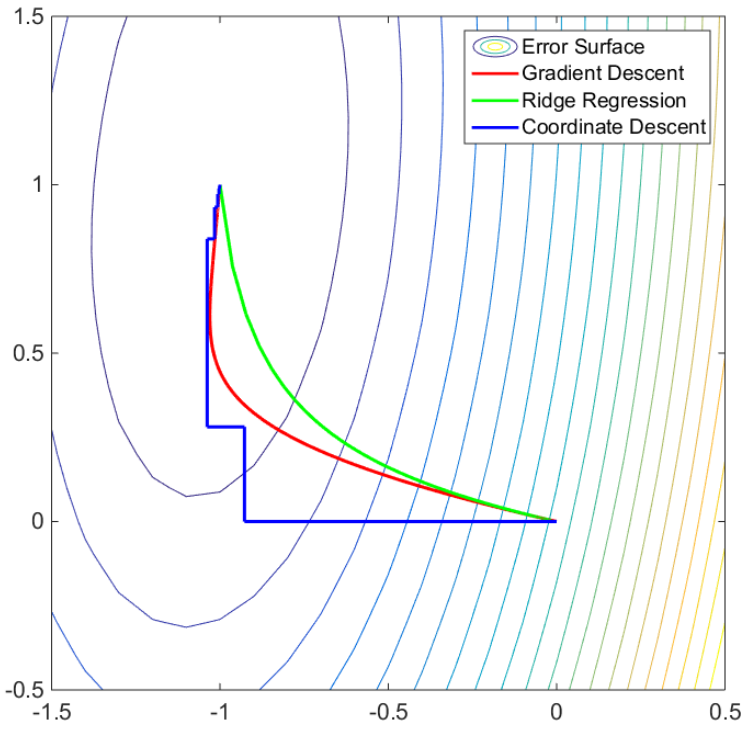


Figure 1.3: Comparison of paths through a two dimensional weight space

Gradient descent, ridge regression and coordinate descent can be seen as tracing paths through weight space starting at (0,0) and moving toward the least squares solution. Rach point on the path thus corresponds to the value of a regularization parameter. The regularization parameter can be explicit, as in the case of ridge regression, or can be implicit in the stopping location determined by early stopping. Ridge regression and gradient descent trace out similar paths, with the gradient descent path more affected by the curvature of the error surface. The coordinate descent path moves one dimension at a time, which will clearly induce sparsity when used with early stopping.

solution (see figure 1.2). Note also that varying the λ in ridge regression from very high values down to 0 creates a path through parameter space similar to the gradient descent path (see figure 1.3). In fact, gradient descent with early stopping has been shown to be equivalent to regularization with a Gaussian prior whose covariance is determined by S (Prenger 2008). Using the intuition that gradient descent initially moves in the most important directions for minimizing error, one can take this idea to its logical extreme by modifying the algorithm to only adjust the weight with the largest gradient at each iteration. By starting this new algorithm with all weights at 0 and using early stopping, the final solution will end up being sparse. This new algorithm is known as L2-Boosting (Friedman 1999) and has similar behavior to L1-norm regularized regression, including the tendency to over-sparsify the solutions. To obtain solutions that are both sparse and smooth, one can modify the L2-Boosting algorithm to adjust at each iteration all weights whose gradient magnitude is equal or greater than some threshold, e.g. a fraction of the maximum gradient magnitude. This algorithm is known as threshold gradient descent (Friedman and Popescu 2003) and has similar behavior to elastic-net regularization, but has the advantage of only having one hyperparameter to optimize rather than two. Threshold gradient descent thus offers a way to smoothly vary between standard gradient descent and L2-Boosting; they correspond to threshold settings of 0 of 1 respectively.

Threshold gradient descent can also be modified to enforce grouping behavior similar to the group-LASSO. This can be done by only updating groups of parameters whose summed gradient magnitude exceeds a percentage of the maximum group gradient magnitude sum. Alternatively, the algorithm could be modified to update all weights within a group as long as one member exceeds threshold.

Thus I have shown how gradient descent can be modified to provide behavior similar to the explicit penalties on $\hat{w}$ discussed above. It is important to note that these algorithmic modifications are really no more or less principled than the use of explicit priors and penalties; they are both means to obtain parameter vectors that have desired properties, such as smoothness and sparseness, and that generalize better to new data sets.

Figure 1.3 demonstrates the effects of STRFlab's regularization techniques on a GLM of a simple cell. The STRF GLM was fit to data recorded from an anaesthetized cat (Touryan, Felsen, and Dan 2005).

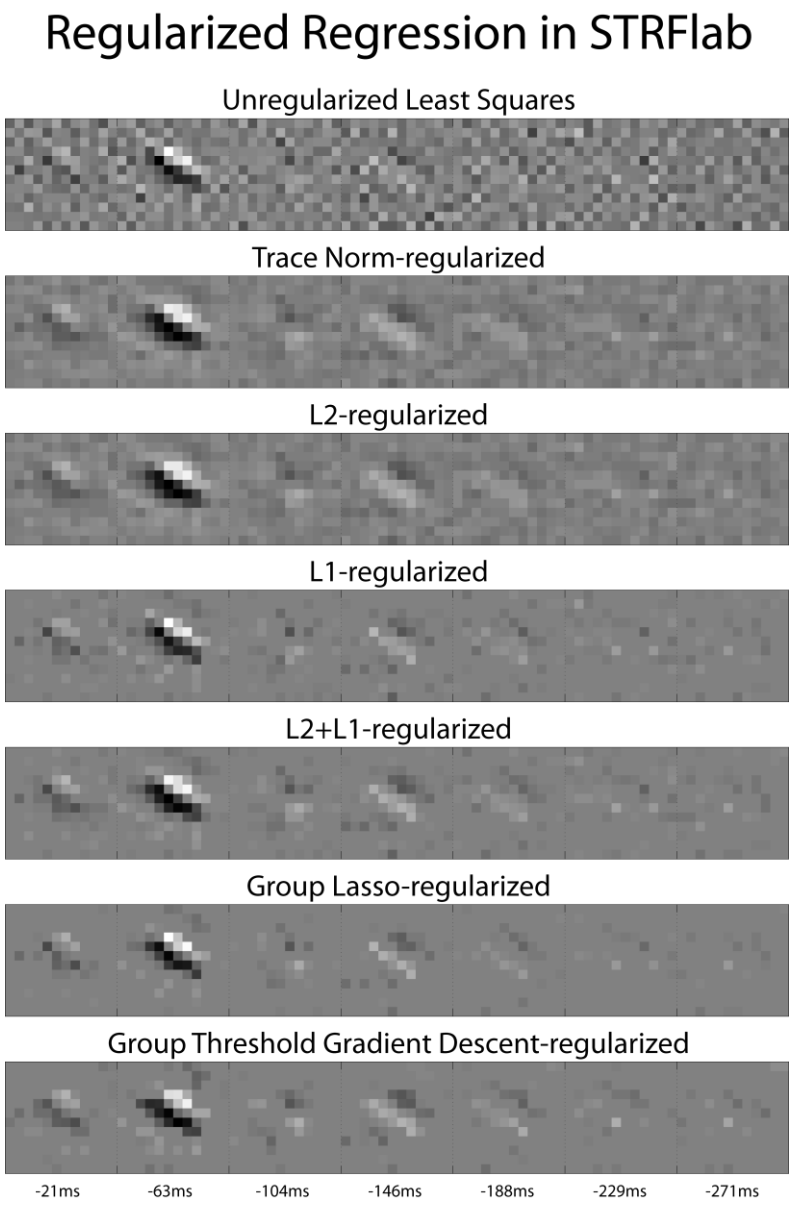


Figure 1.4: Regularized Regression in STRFlab

A linear STRF (similar to the Spike-Triggered Average) fit with various forms of regularization. From left to right in each row are the regression weights on the pixels preceding a spike. The top row shows the corrected Spike Triggered Average solution with no regularization. The following five rows show the effects of different penalties on the solution. The bottom row is the solution obtained by group threshold gradient descent, a heuristic regularizer. The optimal hyperparameters for all forms of regularization were chosen by cross-validation. This cell is clearly sensitive to oriented edges of ~135°. All forms of regularization reveal more structure than is clear in the unregularized solution, i.e. the cell's preferred phase reverses between 146ms and 62ms prior to a spike.

Table 1.2. Table of Regression Penalties in STRFlab

Name of Penalty	Mathematical Form	Effect on Weight estimates
L1-norm, " LASSO"	$\lambda\|\|\widehat{w}\|\|_1$	Forces small weight values to zero inducing sparseness in $\widehat{w}$
L2-norm, "Ridge"	$\lambda\|\|\widehat{w}\|\|_2^2$	Makes weight values more similar (and closer to 0) inducing smoothness in $\widehat{w}$
L1-norm + L2-norm, "Elastic Net"	$\lambda_1\|\|\widehat{w}\|\|_1 + \lambda_2\|\|\widehat{w}\|\|_2^2$	Trades off between L1 and L2 to induce smoothness and sparseness in $\widehat{w}$
"Trace-Norm", "Nuclear-Norm"	$\lambda\|\|\widehat{w}\|\|_* = \lambda\, trace\left(\sqrt{\widehat{w}^T\widehat{w}}\right)$	By reformatting $\widehat{w}$ into a matrix of features x delays, this penalty forces the model toward feature-time separability
"Group LASSO"	$\lambda\sum_{l=1}^{L}\|\|\widehat{w}_l\|\|_2$	Induces sparseness at the level of L groupings within $\widehat{w}$, e.g. the delayed copies of a feature could constitute a group

1.2.5 Fitting a GLM: resampling

Fitting the values of $\widehat{w}$ with regularization involves a trade-off between how well $\widehat{w}$ fits the data and any assumptions about $\widehat{w}$ (formulated either as an explicit prior distribution $P(\widehat{w})$ or heuristic rules). This tradeoff introduces one or more hyperparameters, i.e. λ, that quantify the relative weightings placed on the goodness-of-fit and on the assumptions about $\widehat{w}$. To find a value for the hyperparameters that optimally allows the model to generalize to new data, I will now turn to data resampling.

Resampling data provides a way to estimate hyperparameters, determine the generalization error of a model, the bias of the performance metric, and confidence bounds on weight estimates. Aggregating the models that result from each

resampling is also a simple and powerful method for regularizing the solution and preventing the model from overfitting to noise in the data.

1.2.5.1 Cross-validation (K-fold)

K-fold cross-validation is a resampling technique used to estimate how well a model will generalize to a new set of data (Picard and Cook 1984). The technique involves partitioning the data set into K non-overlapping subsets and fitting the model using K-1 of the subsets. The resultant model is used to predict on the remaining subset and prediction accuracy is then calculated. This process is repeated, with a different subset used as the testing set, until each of the K subsets has been used as a testing set. The prediction accuracy measurements are averaged across the K cross-validation rounds to obtain a measure of the generalization ability of the model. To estimate hyperparameter values, Cross-validation can be performed several times, each time with the hyperparameters set to different values. Finally, the hyperparameters are set to the values that give the best generalization performance and the model is refit using all the data.

1.2.5.2 Bootstrapping

Bootstrapping is a resampling technique used to estimate the properties (i.e. bias, variance) of the weights $\widehat{w}$ of a STRF GLM (Efron 1982). Bootstrapping is the process of taking random samples with replacement from the data set (of an equal size of the data set) many times to create resampled versions of the data. The STRF weights $\widehat{w}$ are estimated on each resampled data set and a distribution over each STRF weight can be obtained. Estimates of the statistical properties of the STRF model (i.e. bias, variance, odds-ratio, etc.) can be obtained by examining the distributions for each of the model weights. For a thorough review of bootstrapping see (Efron 1982).

The STRF estimates from all the bootstraps can be combined in a process known as Bootstrap Aggregation or "Bagging" (Breiman 1996). Bagging involves taking the mean or median of the distribution of each STRF GLM weight to yield a final STRF estimate. Bagging typically makes fitting more resistant to outliers and improves the ability of a model to generalize to new data.

1.2.5.3 Jackknifing

Jackknifing is similar to bootstrapping, but rather than taking random samples with replacement from the fitting data, Jackknifing holds out random portions of the fitting data each time to create resampled versions of the data (Tukey 1958). The STRF model weights $\widehat{w}$ are then fit to each resampled data set. Estimates of the bias

or variance of the model weights can be obtained by examining the distributions of values for each of the model weights. As with Bagging, a final STRF can be obtained by taking the mean or median of the weight distributions. Jackknifing is an older technique than Bootstrapping, and is more narrow in its uses, i.e. only bias and variance estimation, rather than any statistical property. Therefore, in most cases it is advisable to use Bootstrapping over Jackknifing. For a detailed review of Jackknifing see (Efron 1982).

1.3. STRFlab overview and usage examples

STRFlab contains tools for quickly and easily estimating, evaluating and interpreting STRF models of sensory neurons. Included are routines for preprocessing stimuli through projections into various nonlinear feature spaces, fitting the STRF model to neural data, resampling the data to fit hyperparameters, visualizing the resultant model weights, and validating the model by testing its predictive power. These routines are designed to be applied sequentially, forming a powerful analysis stream. STRFlab is object oriented so that model components and data processing choices can be easily swapped in and out to develop customized and flexible analysis streams.

1.3.1 Availability and requirements

STRFlab is available for download from http://strflab.berkeley.edu. STRFlab is released under the BSD license and is provided without any warranty or guaranties. STRFlab was designed to work with MATLAB 7.0 and higher without any additional toolboxes.

For maximum generality, STRFlab is designed to work with any data that can be loaded into MATLAB as a matrix. The only restriction on data is that the temporal sampling rate and number of samples must be identical for the stimulus and response (which may require up or down-sampling depending upon the situation) and that the response is a one-dimensional vector.

1.3.2 Preprocessing

When analyzing a new set of data, the first choice that must be made is what sort of preprocessing will be applied to the stimuli. The preprocessing routine instantiates a hypothesis about the appropriate feature space for modeling and thus the specific choice of preprocessing will vary with the question under investigation. STRFlab comes with several options for preprocessing both visual and auditory stimuli. The preprocessing routines return a matrix with dimension *temporal samples* by *stimulus feature channels*. This new stimulus matrix will be used by STRFlab to

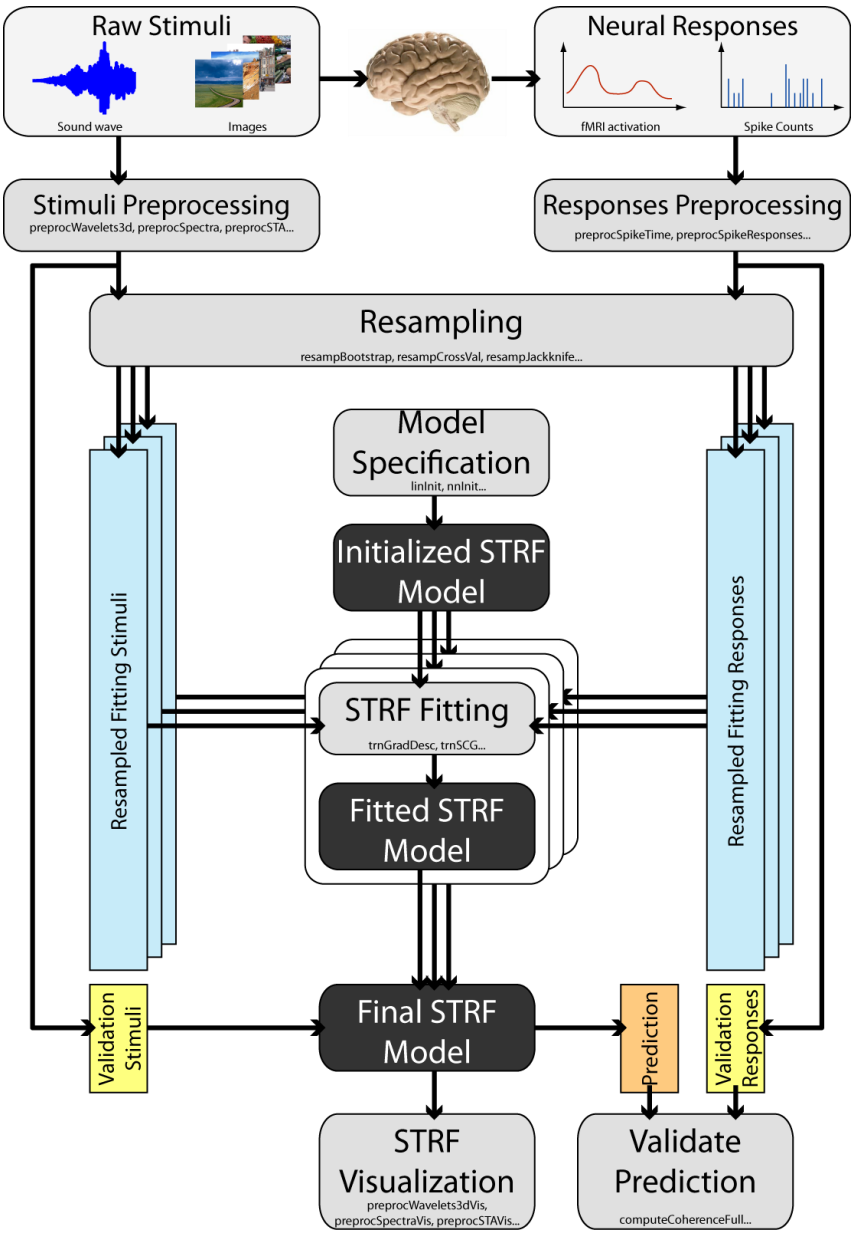


Figure 1.5: STRFlab Flow Chart

Starting from the top left, a visual or auditory stimuli are presented and neural responses are recorded. The raw stimuli are preprocessed by projection into an appropriate feature space and the neural responses are preprocessed to have the same sampling rate as the preprocessed stimuli. A portion of the preprocessed stimuli and responses are reserved as a validation data set, while the rest of the data is resampled. A STRF model is specified and then fit on all the resampled versions of the data. The STRF models are aggregated to form a final STRF model which is used to make predictions on the data reserved for validation. The final model can then be visualized and its predictions evaluated.

estimate the STRF in the preprocessed stimulus space. The object-orientated nature of STRFlab makes it simple to integrate a custom preprocessing routine, should one be desired. Since many neurons encode a nonlinear function of the raw stimulus, the goal of preprocessing is to transform the stimulus into a new feature space to which the neuron has a linear relationship. For example, taking the Fourier power of a set of stimulus images yields a stimulus space that should support an approximately linear relationship to the responses of V1 complex cells (David, Vinje, and Gallant 2004).

STRFlab contains preprocessing routines to enable the spike triggered average (STA), nonlinear transforms that can decompose image and movie stimuli into Fourier power spectra or Gabor wavelets, as well as routines to decompose auditory stimuli into Fourier power spectra. Each preprocessing routine has many options for customizing how the stimuli are transformed; the sampling density of stimulus features (i.e. frequency or orientation), rectifying nonlinearities, normalization, and more can all be tailored to a specific analysis. To clarify this process, I will now demonstrate three typical usage scenarios for STRFlab. The code used for each example is highlighted with a unique color.

Vision Example 1 - STA

Here I use the preprocessing routine for calculating the STA. Because the STA is a linear regression on the pixel values, the preprocSTA routine just formats the stimulus movie to work with STRFlab. I begin by calling the preprocessing routine with no arguments, which returns the default options for the routine:

```
preprocOptSTA = preprocSTA;
```

To preprocess the stimuli I simply pass the movie (an X × Y × T matrix) and the preprocessing options to the preprocessing routine, which outputs the preprocessed stimuli in a form usable by STRFlab:

```
stimSTA = preprocSTA(movie, preprocOptSTA);
```

Vision Example 2 - Gabor Wavelet Model

Here I consider the preprocessing routine for projecting the stimulus onto a 3D Gabor wavelet feature space. The resulting stimulus matrix is a nonlinear transformation of the original stimulus and consists of the responses of model “simple” and “complex” cells of various spatial positions, scales, orientations and movement directions. I begin by calling the preprocessing routine with no arguments, which returns the default options for the routine:

preprocOptWav = preprocWavelets3d;

To preprocess the stimuli I simply pass the movie (an X × Y × T matrix) and the preprocessing options to the preprocessing routine, which outputs the preprocessed stimuli in a form usable by STRFlab:

stimWav = preprocWavelets3d(movie, preprocOptWav);

Auditory Example 1 – Short Time Fourier Transform

For the auditory example I will use the preprocessing routine for projecting the sound waveform into Fourier-power space. The resulting stimulus matrix is a nonlinear transformation of the sound waveform that quantifies how the frequency content of the sound changes over time. I begin by calling the preprocessing routine with no arguments, which returns the default options for the routine:

preprocOptSTFT = preprocSTFT;

To preprocess the stimuli I simply pass the sound file (a 1 × T vector) and the preprocessing options to the preprocessing routine, which outputs the preprocessed stimuli in a form usable by STRFlab:

stimAud = preprocSTFT(audio, preprocOptSTFT);

1.3.3 Model specification

After choosing and applying a preprocessing routine, the details of the STRF GLM must now be specified. The GLM included in STRFlab implements the regression frameworks listed in Table 1: linear regression, logistic regression and Poisson regression. The choice of regression framework is easily switched and the corresponding mean and error functions are automatically chosen. Additionally, because there is often an unknown delay between the stimulus and neural response, the *delays* variable is used to specify the number of stimulus samples preceding a response that the model must take into account.

The object oriented framework enables STRFlab to be easily augmented with new model types. To define a new model that can take advantage of STRFlab's fitting routines, it is necessary to define functions to initialize an instance of the model, compute the output of the model, compute the error function between the model output and neural response, compute the gradient, and do relevant bookkeeping.

A new STRF GLM is instantiated by the function *linInit*.

Vision Example 1 - STA

For the first visual cortex example I create a STRF object with the number of stimulus channels determined by the preprocessed stimuli, delays up to seven frames and a linear mean function:

strfSTA = linInit(size(stimSTA,2), [0:7], 'linear')

Vision Example 2 - Gabor Wavelet Model

For the second visual cortex example I also create a STRF object with the number of stimulus channels determined by the preprocessed stimuli, delays up to seven frames and a linear mean function:

strfWav = linInit(size(stimWav,2), [0:7], 'linear')

Auditory Example 1 – Short Time Fourier Transform

For the auditory cortex example I create a linear STRF object with the number of stimulus channels determined by the preprocessed stimuli, delays up to sixty samples (due to the higher sampling rate of the auditory stimuli), and a linear mean function:

strfAud = linInit(size(stimAud,2), [0:60], 'linear')

1.3.4 Resampling

Before fitting the specified STRF GLM to data, I can choose to use a data-resampling scheme as part of the fitting process. STRFlab includes functions for three popular resampling techniques: bootstrapping, cross-validation, and jackknifing. The choice of resampling technique will depend on the purpose of resampling i.e., hyperparameter fitting, estimating generalization error, computing bias, computing confidence bounds, etc.

1.3.4.1 Cross-validation

The K-fold cross-validation procedure included in STRFlab is *resampCrossVal*. Calling *resampCrossVal* with no arguments returns the default options structure. Using the options structure, STRFlab allows an arbitrary number of subsets to be used. The K data subsets can be created from continuous chunks of data or randomly sampled chunks of data using the randomize option.

Vision Example 1 – STA

For the STA example I will use cross-validation to resample the data and determine the optimal hyperparameter for the fitting routine. For the example I will set the randomize option to 0 in order to use continuous chunks of data.

optionsSTA=resampCrossVal

optionsSTA.kFold = 5;

optionsSTA.randomize = 0;

1.3.4.2 Bootstrapping

The Bootstrapping routine in STRFlab is *resampBootstrap*. Calling *resampBootstrap* with no arguments returns the default options structure. Using the options structure I can specify the number of resamplings to be used.

Vision Example 2 - Gabor Wavelet Model

For the Gabor Wavelet example I will use bootstrapping to resample the data and obtain a distribution of weight values.

optionsWav=resampBootstrap;

optionsWav.nResamp = 10;

1.3.4.3 Jackknifing

The Jackknifing routine in STRFlab is *resampJackknife*. Calling *resampJackknife* with no arguments returns the default options structure. Using the options structure I can specify the number of resamplings to be used and what fraction of the data to leave out on each resampling.

Auditory Example 1 – Short Time Fourier Transform

For the auditory example I set the nResamp option to use 10 resamplings and hold out 10% of the data at each resampling using the jackFrac option.

optionsAud=resampJackknife;

optionsAud.nResamp = 10;

optionsAud.jackFrac = 0.10;

1.3.5 Globalizing data and grouping samples

Because STRFlab contains many functions that act on the same data, and such data is often very large, storing the stimulus and response matrices as global variables results in a performance improvement. STRFlab includes a function for storing the stimulus and response matrices in a global variable named globDat, which the other functions reference internally.

Vision Example 1 - STA

For the STA example, I globalize the *globDat* variable and pass the stimulus matrix *stimSTA* and response vector *respV1* to *strfData*:

```
global globDat;

strfData(stimSTA,respV1);
```

Vision Example 2 - Gabor Wavelet Model

For the 3D wavelet example, I globalize the *globDat* variable and pass the stimulus matrix *stimWav* and response vector *respV1* to *strfData*:

```
global globDat;

strfData(stimWav,respV1);
```

Auditory Example 1 – Short Time Fourier Transform

STRFlab's built in resampling routines will automatically resample the fitting data by treating each sample independently. Some users, however, may desire more control over how their data is resampled. For this purpose I include the option of creating groups within the data. The resampling routines will then resample at the level of whole groups, rather than individual samples. The groups are specified by an indexing vector listing the group number for each sample. For the auditory example, I globalize the *globDat* variable and pass the stimulus matrix *stimAud*, response vector *respAud*, and the group index *groupIdx* to *strfData*:

```
groupIdx = repmat([1:20], [200 1]);

groupIdx = groupIdx(:);

strfData(stimAud,respAud,groupIdx);
```

1.3.6 Selecting a Fitting Algorithm

STRFlab includes two state-of-the-art fitting algorithms designed to fit GLMs with various forms of regularization.

The routine *fitThreshGradDesc* is an iterative threshold gradient descent (Friedman and Popescu 2003) based fitting routine. This routine uses a subset of the fitting data as a "testing set" to regularize the solution via early stopping (Zhang and Yu 2005; Bühlmann and Yu 2006). The algorithm allows fine control over the sparsity of the weight values by means of an adjustable threshold that controls which weights are updated at each iteration. This thresholding operation is a simple but powerful heuristic that allows great flexibility in regularizing STRF GLMs.

The routine *fitFista* is based on the recently developed FISTA algorithm (Beck and Teboulle 2009; Mairal et al. 2010) for penalized regression. *fitFista* allows a wide variety of penalties to be used, i.e. (L1, L2, L1+L2, trace-norm, etc.), each of which corresponds to different assumptions about the distribution of the weight values. The strength of the penalty is controlled by user specified hyperparameter values. *fitFista* can also use subset of the fitting data as a "testing set" to automatically choose from a list of specified hyperparameter values. Figure 1.4 shows the effect of different penalties, each with the optimal hyperparameter determined by a testing set.

Vision Example 1 - STA

For the STA example I will use the *fitFista* algorithm with L1-norm regularization and a large range of lambda values. Calling *fitFista* with no arguments returns the default options structure. I put the options for *fitFista* in the optimization options field, *optimOpt*, of the structure I created earlier using *resampCrossVal*. The selection of a subset of data for a "testing set" will be automatically performed by *resampCrossVal*:

```
optionsSTA.optimOpt = fitFista;

optionsSTA.optimOpt.regul = 'l1';

optionsSTA.optimOpt.lambda = [3e4 2e4 1e4 8e3 6e3 4e3 2e3 1e3 5e2 1e2 5e1 1e1];
```

Vision Example 2 - Gabor Wavelet Model

For the 3D wavelet example, I use the *fitThreshGradDesc* fitting routine. Again I put the options for the fitting routine into the *options.optimOpt* structure created by the resampling routine. A powerful feature of this fitting routine is the ability to treat

the delayed copies of each feature as a group. To obtain high sparsity at the level of feature groups, I will set the overall threshold to 0.9 so only groups including features with gradients greater than 90% of the maximum gradient have their weights updated. To obtain low sparsity within groups I set the within group threshold to 0.2 so only features with gradients greater than 20% of the maximum gradient within a group are updated:

optionsWav.optimOpt= fitThreshGradDesc;

optionsWav.optimOpt.useGroups = 1;

optionsWav.optimOpt.threshold = .9;

optionsWav.optimOpt.thresholdGroup = .2;

Auditory Example 1 – Short Time Fourier Transform

For the 3D wavelet example, I also use the *fitThreshGradDesc* algorithm. I put the default options for the fitting routine into the *optionsAud.optimOpt* structure created by *resampJackknife*:

optionsAud.optimOpt= fitThreshGradDesc;

optionsAud.optimOpt.useGroups = 1;

optionsAud.optimOpt.threshold = .9;

optionsAud.optimOpt.thresholdGroup = .2;

1.3.7 Indexing the data and starting the algorithm

Before fitting the model, the user must create a vector indexing the samples to be used for fitting. After the index of fitting data is created it is a simple matter to run the analysis. Simply pass the STRF GLM, the fitting index, and the options structure to the function *strfOpt*.

Vision Example 1 - STA

To fit the STRF for the STA example, I only need to provide a fitting index, *fittingIdxSTA*, since the resampling routine specified earlier will take care of determining the "testing set". The nSample field in the globDat structure I created with strfData contains the number of samples in the data, so I can use this to help create the indices. I will use the first 80% of the data for fitting the model and reserve the remaining 20% for later validating the model. I then call strfOpt to fit the model:

*fittingIdxSTA=1:round(.8*globDat.nSample);*

*valIdxSTA = round(.8*globDat.nSample)+1:globDat.nSample;*

[strfSTAFit optionsSTAFit]=strfOpt(strfSTA, fittingIdxSTA,optionsSTA);

Vision Example 2 - Gabor Wavelet Model

To fit the STRF for the Gabor Wavelet example, I only need to provide a fitting index, *fittingIdxWav*, since the resampling routine specified earlier will take care of the "testing set". The nSample field in the globDat structure I created with strfData contains the number of samples in the data, so I can use this to help create the indices. I will use the first 80% of the data for fitting the model and reserve the remaining 20% for later validating the model. I then call strfOpt to fit the model:

*fittingIdxWav=1:round(.8*globDat.nSample);*

*valIdxWav = round(.8*globDat.nSample)+1:globDat.nSample;*

strfFitWav=strfOpt(strfWav,fittingIdxWav,optionsWav);

Auditory Example 1 – Short Time Fourier Transform

To fit the STRF for the auditory example, I only need to provide a fitting index, *fittingIdxAud*, since the resampling routine specified earlier will take care of the "testing set". I will use the all but one of the data groups specified earlier for fitting the model and reserve group 12 for validating the model. I then call strfOpt to fit the model:

fittingIdxAud= find(groupIdx ~=12);

valIdxAud = find(groupIdx ==12);

strfFitAud=strfOpt(strfAud,fittingIdxAud,optionsAud);

1.3.8 Finalizing the model

Fitting a model to resampled data results in a separate model for each round of data resampling. To obtain a single final model that can be used for prediction on a validation data set I can either 1) use the resampling techniques to find the hyperparameter value that gives the best generalization performance and then refit the model to all the fitting data using that hyperparameter, or 2) average the model parameters from each resampling.

Vision Example 1 - STA

For the STA example I used cross-validation with the goal of selecting an optimal *lambda*. The optimal lambda from each round of cross validation is stored in the *optionsSTAFit* structure returned by *strfOpt*. To obtain a lambda value for a final fitting without cross-validation, I can take the optimal lambda from each round of cross-validation, concatenate them together and take the median value:

```
best_lambdas =  cat(2, optionsSTAFit.lambda);

optionsSTAFinal = fitFista;

optionsSTAFinal.regul = 'l1';

optionsSTAFinal.lambda = median(best_lambdas);

[strfSTAFinal optionsSTAFinal]=strfOpt(strfSTA, fittingIdxSTA,optionsSTAFinal);
```

Vision Example 2 - Gabor Wavelet Model

In the 3D wavelet example I fit the STRF to 10 resamplings of the fitting data, which resulted in 10 different models. As mentioned earlier, I can take the aggregate the weights across the models to yield a robust final model. To do this I create a new STRF structure by copying the initial STRF structure and then filling in the weights with the median, a robust measure of central tendency (Wilcox and Keselman 2003), of the weight values from all the resamplings:

```
strfWavFinal = strfWav;

strfWavFinal.w1 = median(cat(4, strfFitWav.w1), 4);

strfWavFinal.b1 = median(cat(2, strfFitWav.b1), 2);
```

Auditory Example 1 – Short Time Fourier Transform

In the auditory example I fit the STRF to 10 resamplings of the data, which resulted in 10 different models. As mentioned earlier, I can aggregate the weights across the models to yield a robust final model. To do this I create a new STRF structure by copying the initial STRF structure and then filling in the weights with the mean of the weight values from all the resamplings:

```
strfAudFinal = strfAud;

strfAudFinal.w1 = mean(cat(4, strfFitAud.w1), 4);
```

strfAudFinal.b1 = mean(cat(2, strfFitAud.b1), 2);

1.3.9 Visualizing results

Once a final model has been obtained, it is often useful to visualize the stimulus features represented by the model. Because these stimulus features are dependent upon the stimulus preprocessing used, STRFlab includes a visualization routine for each of preprocessing routines. To visualize a STRF model, all that is necessary is to pass the final STRF structure to the visualization routine.

Vision Example 1 - STA

I will now visualize the finalized STRF GLM in the STA example. Note that because I regularized the weight values using the L1-norm penalty, it is not the same as the traditionally defined Spike Triggered Average or corrected Spike Triggered Average; it is a regularized version of the corrected Spike Triggered Average.

preprocSTA_Vis(strfSTAFinal);

Vision Example 2 - Gabor Wavelet Model

I will also visualize the weights of the Gabor wavelet based STRF model. I see that the final model uses mainly simple-cell-like features, which closely matchs the appearance of the STA model.

preprocWavelets3d_Vis(strfWavFinal);

Auditory Example 1 – Short Time Fourier Transform

I will now visualize the STRF trained in the auditory example. Since each weight in the STRF is a weight on a particular frequency band at a particular time-delay, the weights can be visualized as a weighted spectrogram:

preprocSTFT_Vis(strfAudFinal);

1.3.10 Fitting output nonlinearities

For all of the running examples I have been using a STRF GLM with a linear mean function. Using a linear mean function, however, can lead to biophysically impossible predictions such as negative firing rates. STRFlab can use other mean functions (see table 1) that prohibit negative outputs, but a powerful and more flexible alternative is to fit positive monotonic output nonlinearities after fitting a STRF GLM with a linear mean function. STRFlab contains the function *fitOutputNL* that can fit three families of output nonlinearities: monotonic cubic spline (Park and

Pillow 2011), rectified power function (Talebi and Baker 2012), and the contrast response function (Albrecht, Hamilton, and David 1982). The output nonlinearity is automatically incorporated into the STRF structure and can be visualized using *fitOutputNL_vis.*

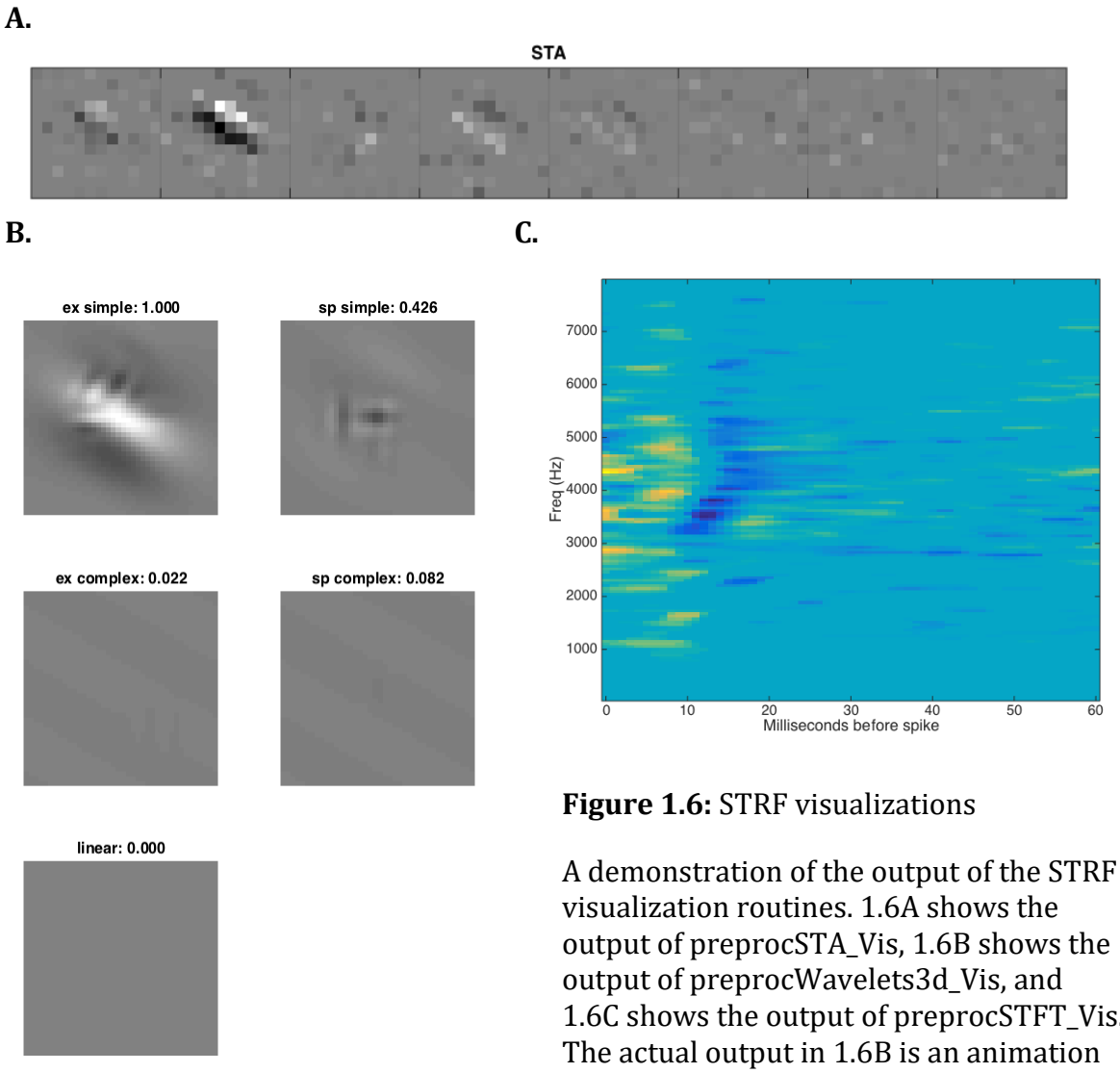


Figure 1.6: STRF visualizations

A demonstration of the output of the STRF visualization routines. 1.6A shows the output of preprocSTA_Vis, 1.6B shows the output of preprocWavelets3d_Vis, and 1.6C shows the output of preprocSTFT_Vis. The actual output in 1.6B is an animation so only the peak frame is shown.

Vision Example 1 – STA

For the STA example I will fit a polynomial spline of order 3 with 5 knots (see figure 1.7A):

optionsNL = fitOutputNL;

optionsNL.type = 'spline';

optionsNL.knots = 5;

optionsNL.order = 3;

strfSTAFinal = fitOutputNL(strfSTAFinal, fittingIdxSTA, optionsNL);

fitOutputNL_Vis(strfSTAFinal)

Vision Example 2 - Gabor Wavelet Model

For the Gabor Wavelet example I will fit a rectified power function of the form: $(|a + b \times \hat{r}|^{+})^{n}$ where the terms a, b, and n are fit (see figure 1.7B):

optionsNL = fitOutputNL;

optionsNL.type = 'rectPower';

strfWavFinal = fitOutputNL(strfWavFinal, fittingIdxWav, optionsNL);

fitOutputNL_Vis(strfWavFinal)

Auditory Example 1 – Short Time Fourier Transform

For the auditory example I will fit a polynomial spline of order 3 with 5 knots (see figure 1.7C):

optionsNL = fitOutputNL;

optionsNL.type = 'spline';

optionsNL.knots = 5;

optionsNL.order = 3;

strfAudFinal = fitOutputNL(strfAudFinal, fittingIdxAud, optionsNL);

fitOutputNL_Vis(strfAudFinal)

1.3.11 Prediction and validation

The final step in STRF modeling is to test a model's predictive ability. The ability of a model to accurately make predictions in novel conditions is the best way to evaluate a model. STRFlab includes the routine *strfValidation* to estimate several metrics of prediction quality: correlation (Hsu, Borst, and Theunissen 2004), percent of explainable variance accounted for (David and Gallant 2005), and coherence (Hsu, Borst, and Theunissen 2004). *strfValidation* will also plot the true neural responses and predictions for visual inspection.

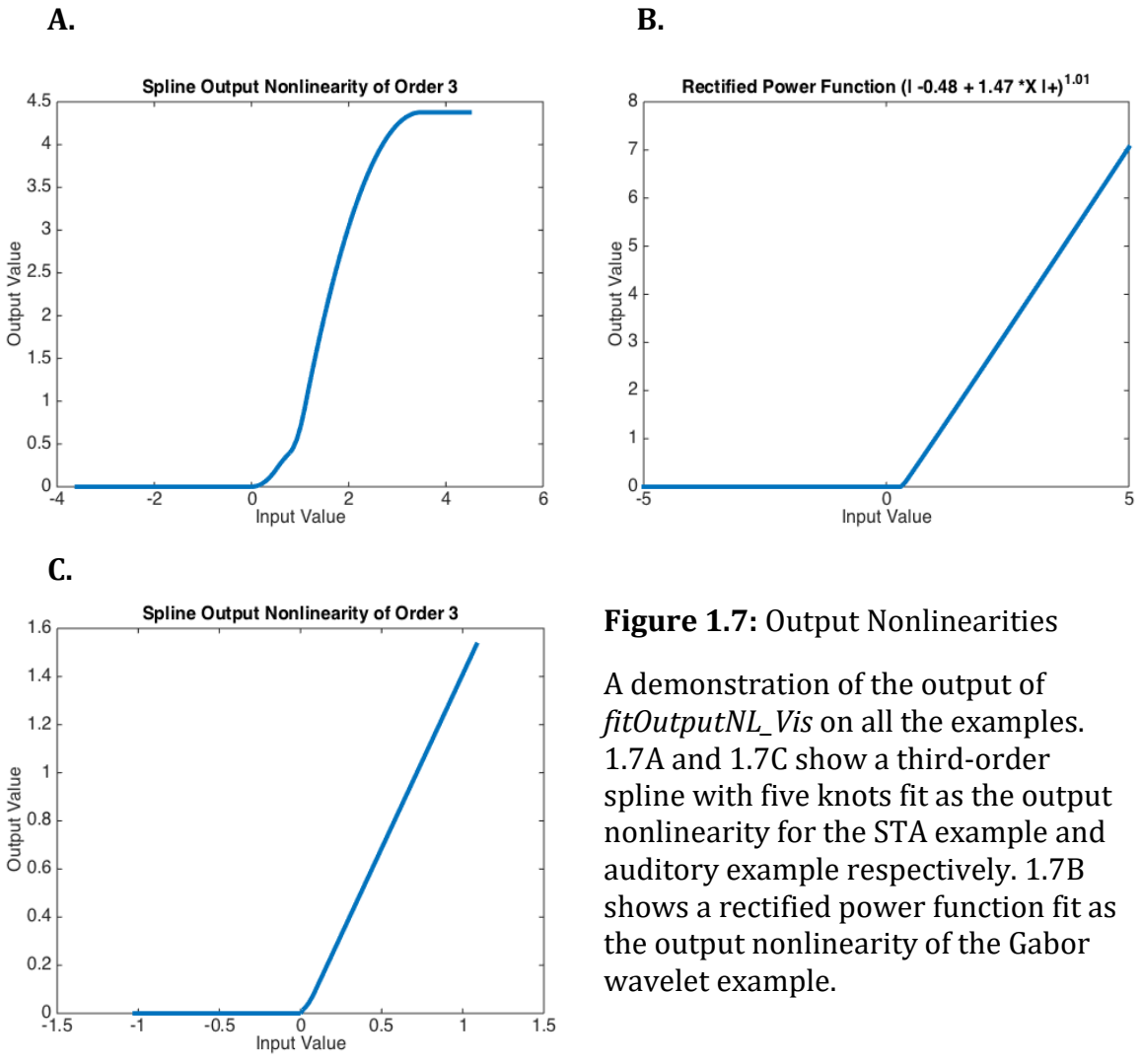


Figure 1.7: Output Nonlinearities

A demonstration of the output of *fitOutputNL_Vis* on all the examples. 1.7A and 1.7C show a third-order spline with five knots fit as the output nonlinearity for the STA example and auditory example respectively. 1.7B shows a rectified power function fit as the output nonlinearity of the Gabor wavelet example.

Vision Example 1 – STA

To conclude the STA example, I will use *strfValidation* to compute the correlation between true neural responses and the prediction of the final STRF on the validation data I reserved earlier and indexed by *valIdxSTA*:

```
optionsSTAVal = strfValidation;

optionsSTAVal.type = 'corr';

[strfSTAFinal, predSTA, corrScore] = strfValidation(strfSTAFinal, valIdxSTA,
optionsSTAVal);
```

Vision Example 2 - Gabor Wavelet Model

To conclude the Gabor Wavelet example, I will use *strfValidation* to compute the percentage of explainable variance accounted for by the prediction of the final STRF on the validation data I reserved earlier and indexed by *valIdxWav*:

```
optionsWavVal = strfValidation;

optionsWavVal.type = 'EV';

[strfWavFinal, predWav, EVscore] = strfValidation(strfWavFinal, valIdxWav,
optionsWavVal);
```

Auditory Example 1 – Short Time Fourier Transform

To conclude the Auditory STRF example, I will use *strfValidation* to compute the coherence between true neural responses and the prediction of the final STRF on the validation data I reserved earlier and indexed by *valIdxAud*:

```
optionsAudVal = strfValidation;

optionsAudVal.type = 'coherence';

[strfAudFinal, predAud, coherenceScore] = strfValidation(strfAudFinal, valIdxAud,
optionsAudVal);
```

1.4 Conclusions

STRFlab adapts many state-of-the-art techniques for regularized regression to the problem of receptive field estimation and makes them accessible to a wide audience. STRFlab also contains stimulus preprocessing routines that implement the most common nonlinear transforms used in receptive field modeling and is easily

extended with user defined preprocessing routines. Figure 1.7 summarizes the workflow enabled by STRFlab. With just a few lines of code it is possible to define complex models, fit them to data, determine their optimal hyperparameters, test their predictive power and visualize the results. The goal of STRFlab is to encourage the widespread use of modern receptive field modeling techniques and advance data driven neuroscience.

Chapter 2

Boosted Inhomogeneous Polynomial Kernel Networks for Estimating High Dimensional High Order Volterra Models of the Visual Cortex

Chapter 2

Boosted Inhomogeneous Polynomial Kernel Networks for Estimating High Dimensional High Order Volterra Models of the Visual Cortex

2.1 Introduction

The Volterra series is a general method for modeling nonlinear systems and is widely used in science and engineering, and particularly in system identification (Korenberg and Hunter 1996). Volterra series methods have a long history in computational neuroscience due to their predictive power and interpretability. The classical techniques known as the Spike Triggered Average (STA) and Spike Triggered Covariance (STC) correspond to the first order and second order Volterra series kernels respectively (see section 2.2). Infinite order Volterra series models can theoretically model any nonlinear system. Yet due to the fact the number of terms in the model grows exponentially as the order increases, only first and second order models are used in practice. While there is some evidence that visual neurons are selective for higher order stimulus statistics (Purpura, Victor, and Katz 1994) it has been impossible to explicitly model this selectivity. Recent work in kernel regression has demonstrated the possibility of fitting Volterra series models implicitly using the Inhomogeneous Polynomial Kernel (Dodd 2002; Franz and Schölkopf 2006), thus bypassing the need to explicitly represent all interaction terms. Unfortunately kernel methods become unwieldy as the size of the data set increases, constraining their usefulness on neurophysiological data sets. To retain an implicit Volterra series representation while enabling the ability to fit large, high dimensional data sets, I propose to exploit the connection between kernel methods and neural networks to construct what I term an Inhomogeneous Polynomial Kernel Network (IPKN).

The IPKN model is similar to a single hidden layer neural network. However, the sigmoidal activation functions of the hidden units are replaced by an arbitrary order Inhomogeneous Polynomial kernel. The IPKN model can be fit using standard neural network fitting methods, i.e. backpropagation and conjugate gradient algorithms. Most importantly, the explicit Volterra series coefficients can be recovered from a fit IPKN model and then interpreted.

To improve the performance of IPKN models, I propose to integrate IPKNs into the Stochastic Gradient Boosting algorithm (Friedman 2002) as the weak learners. In this framework an ensemble of IPKNs are fit one at a time to the residual of the current ensemble prediction. Because the IPKN models are combined additively, the coefficients recovered from each IPKN model can be combined additively to yield a

single set of Volterra series coefficients. The Stochastic Gradient Boosting procedure improves the estimates of the Volterra coefficients over a single IPKN, giving robust estimates of higher order Volterra coefficients. Furthermore, because stochastic gradient boosting is a general regression framework it does not make assumptions about stimulus statistics, as does STC, and it can incorporate appropriate noise models for fitting neural data, unlike STC. Therefore, the proposed algorithms can not only create second order models more flexibly than STC, but have the ability to scale to higher order models as needed. I propose that these higher order coefficients coulc be interpreted analogously to the method used to interpret second order coefficients in STC analysis: High Order Singular Value Decomposition of their tensor representation.

Finally, I demonstrate the applicability of this method using several synthetic data sets as well as real data sets including electrophysiological recordings from visual area V1 of the cat and macaque and area V4 of the macaque. I am able to recover the first three dimensional (X-Y-time) Volterra models of complex cell receptive fields. These models contain over 600,000 implicit parameters and can be fit with as few as 8000 spikes. I then show that allowing Volterra terms above second order generally decreases prediction performance for models of area V1 while higher order models improve prediction performance in area V4.

2.2 The Volterra series

The Volterra series generalizes the Taylor series to capture the "memory" of a nonlinear system. The discrete infinite memory Volterra series model of a system with two variables, x1 and x2, is (with terms up to second order shown):

$$\begin{aligned} y(t) = h_0 + \sum_{l=0}^{\infty} h_1(l)x_1(t-l) + \sum_{l=0}^{\infty} h_2(l)x_2(t-l) \\ + \sum_{l_1=0}^{\infty}\sum_{l_2=0}^{\infty} h_{11}(l_1,l_2)x_1(t-l_1)x_1(t-l_2) + \sum_{l_1=0}^{\infty}\sum_{l_2=0}^{\infty} h_{12}(l_1,l_2)x_1(t-l_1)x_2(t-l_2) \\ + \sum_{l_1=0}^{\infty}\sum_{l_2=0}^{\infty} h_{22}(l_1,l_2)x_2(t-l_1)x_2(t-l_2) + \cdots \end{aligned} \quad [2.1]$$

To model a nonlinear system, the coefficients of the Volterra series terms, here designated by h, are fit to data. In practical applications an infinite memory model is usually unnecessary and impractical and thus a memory length is specified. Additionally, because the number of model coefficients grows exponentially as

higher order terms are included in the model, practical applications of the Volterra series typically use only the low order terms.

2.3 Kernel regression for Volterra estimation

It has recently been shown that Volterra models can be fit implicitly within the kernel regression framework by adopting the Inhomogeneous Polynomial (IP) kernel function in equation 2.2 (Dodd 2002; Franz and Schölkopf 2006). As demonstrated with a second order IP kernel below (equations 2.3-2.6), using a d-th order IP kernel is equivalent to working in a feature space that includes all terms from zeroth order to d-th order. Or in other words, evaluating the kernel function is equivalent to a dot product in the full polynomial feature space. The kernel regression framework obviates the need to explicitly represent and fit weights for the large number of Volterra series terms, making the fitting of higher order Volterra series models tractable. However, there are a couple of significant downsides to the Kernel regression framework. Kernel regression is not easily scalable to data sets where the number of training points N is large because it involves the construction and inversion of an N by N kernel matrix (equations 2.7, 2.8). There has been significant work in the kernel regression literature to overcome this issue by making intelligent approximations (Engel, Mannor, and Meir 2004; Ranganathan, Yang, and Ho 2011; Tsang, Kwok, and Zurada 2006). However, I propose a promising new path that leverages connections between kernel regression, support vector regression and neural networks, and takes advantage of the one of the most effective statistical learning techniques: Stochastic Gradient Boosting (Friedman 2002).

$$k(x, x^i) = (x \cdot x^i + 1)^d \qquad [2.2]$$

$$(x \cdot x^i + 1)^2 = (x_1 x_1^i + x_2 x_2^i + 1)^2 \qquad [2.3]$$

$$(x_1 x_1^i + x_2 x_2^i + 1)(x_1 x_1^i + x_2 x_2^i + 1) \qquad [2.4]$$

$$(x_1 x_1^i)^2 + (x_2 x_2^i)^2 + 2x_1 x_1^i x_2 x_2^i + 2x_1 x_1^i + 2x_2 x_2^i + 1 \qquad [2.5]$$

$$\begin{pmatrix} (x_1)^2 \\ (x_2)^2 \\ \sqrt{2}x_1\ x_2 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} (x_1^i)^2 \\ (x_2^i)^2 \\ \sqrt{2}x_1^i x_2^i \\ \sqrt{2}x_1^i \\ \sqrt{2}x_2^i \\ 1 \end{pmatrix} \qquad [2.6]$$

2.4 Support vectors and neural networks

Training data can be viewed as a set of support vectors, which define, in concert with a kernel function, a space of potential nonlinear functions. Kernel regression and support vector regression attempt to approximate an arbitrary function by linearly weighting kernel functions that utilize this set of support vectors. However, the space defined by the full support of high dimensional input data is likely much larger than the space of the function one wants to approximate; the function can likely be modeled by a kernel function and a small number of support vectors that *aren't* necessarily part of the training data. As I will show, finding this small set of support vectors is actually a more general formulation of single-hidden-layer neural network modeling.

The Kernel regression framework has clear connections with Neural Network modeling, as is evident from equation 2.9. If I choose the hyperbolic tangent function as the kernel function, k, and consider the training points x^i as input weights to the hidden units and the weights α_i as the output weights from the hidden units, it is clear that a kernel regression model is exactly equivalent to a single layer neural network (see figure 2.1). However, there are three important differences in the typical use of kernel regression models and neural network models: (1) because the input weights are clamped to the value of training points in kernel regression, only the output weights α_i need to be estimated, which can be done analytically using equation 2.8., (2) the number of hidden units in a typical neural network model is much smaller than the number of training points, and (3) in neural networks both the input and output weights are initially randomized and fit iteratively by back-propagation. In both frameworks the input weights form a set of support vectors that, along with the kernel function, define the nonlinear regression function. However, in the neural network framework, the input weights/support vectors are fit by back-propagation. This is a straightforward way to find a small set of support vectors that define the desired nonlinear regression function.

$$K = k(x^i, x^j) \qquad [2.7]$$

$$\alpha = K^{-1}y \qquad [2.8]$$

$$y(x) = \sum_i \alpha_i k(x, x^i) \qquad [2.9]$$

Since I can consider neural networks as a specialized case of kernel methods, it is natural to consider the use of activation/kernel functions other than the hyperbolic tangent function. By simply replacing the hyperbolic tangent with the IP kernel function, I create a kernel network that can be used to implicitly estimate a Volterra series model. However, the kernel network must include memory to be a true Volterra series model. Memory is included in a kernel network by concatenating the current input vector, $x(t)$, with the input vectors from previous time points, e.g. $x(t-1)$, $x(t-2)$, to form the input for the kernel network. Thus by inputting

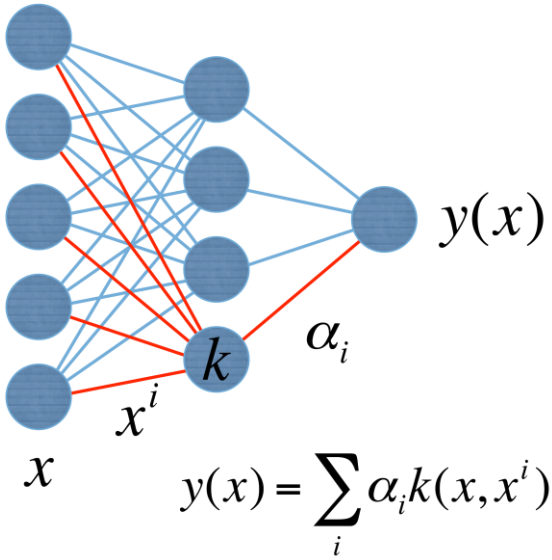


Figure 2.1: Diagram of kernel network

A kernel network is a generalization of a single hidden layer feed-forward neural network. A kernel network differs in that the sigmoid nonlinearity of a neural network is replaced with a kernel function k. The output of the kernel network $y(x)$ given input x is written in kernel regression notation as the weighted sum of the kernel function of x and all the support vectors x^i. In a kernel network the support is fit along with the output weights α^i, whereas in kernel regression, the support is typically composed of training data points and only the output weights are fit.

lagged copies of the input into an IPKN one can create a highly efficient representation of a Volterra series model.

Since the IP kernel function is easily differentiable, it is simple to train the IP network by back-propagation or other gradient based optimization methods, i.e. scaled conjugate gradient (Møller 1993). The number of SCG iterations can be set arbitrarily or determined by "early stopping", i.e. selecting the iteration that produces the minimum error on a separate subset of data not used for training the IPKN.

2.5 Kernel networks as boosted learners

As with neural networks, defining and training kernel networks involves some inherent difficulties. Choosing a sufficient number of hidden units/support vectors is a crucially important step usually solved by the laborious process of testing the performance of different values. In addition, the kernel network solutions found by gradient methods are local minima that can vary greatly depending on starting

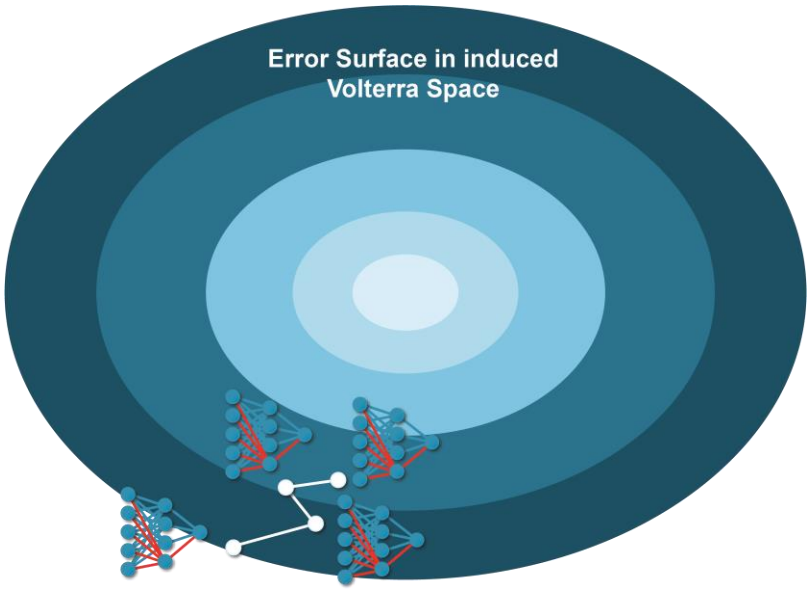


Figure 2.2: Illustration of stochastic gradient descent in Volterra space

A single kernel network corresponds to a set of Volterra coefficients and thus a point in the full Volterra function space. Consequently a sum of the kernel networks corresponds to a single set of Volterra coefficients and thus a single point in Volterra space, since the coefficients from each network combine linearly**.** Iteratively adding kernel networks together to reduce the error function (using the stochastic gradient boosting algorithm) thus corresponds to descending the error surface in the full Volterra function space.

conditions. Ensemble methods offer a simple and elegant way to ameliorate these issues. Simply training many networks with different initial conditions and averaging the output results in stable and accurate models relatively insensitive to the number of hidden units chosen (Perrone and Cooper 1992). Stochastic gradient boosting is a more recent refinement of this simple ensemble approach. Stochastic gradient boosting involves iteratively fitting a weak learner (an IPKN in this case) to the current residual error between a subsample of the training data and the corresponding prediction of the current model. This process can be interpreted as performing stochastic gradient descent in function space (Friedman 2002). Because I have limited the function space to the Volterra series by the choice of the IP kernel, performing SBG with IPKN learners is equivalent to performing stochastic gradient descent in Volterra space (see figure 2.2). And because the data is subsampled during fitting, stochastic gradient boosting scales elegantly to large data sets.

2.5.1 Increasing learner diversity

It is a known property of ensemble models that their performance improves with increasing diversity of their component models (Sollich and Krough 1996). Fitting each model to subsamples of the data and using differing initial weights in the kernel networks introduces some diversity into the population of models, but additional sources of diversity may be beneficial. I have found that clamping the output weights of each network at their initial random is a simple and effective method for creating more diversity among the kernel networks. Clamping the output weights forces training to find the best set of input weights/support vectors given the set of random output weights and consequently encourages a variety of local minima among the learners. I have found this method to significantly improve the generalization performance of boosted ensembles of kernel networks, particularly for higher order models.

2.6 Noise models, robust error functions and output nonlinearities

Stochastic gradient boosting (SGB) is a general nonlinear regression framework. Analogously to the Generalized Linear Model (GLM) framework described in 1.2.1, SGB can incorporate various noise models, along with the corresponding error functions and canonical link functions/mean functions (Friedman 2002). Recall from 1.2.1 that the mean function is usually chosen to be the inverse of the canonical link function for mathematical convenience. However, as I describe below, it is sometimes desirable to use a non-canonical mean function. Algorithm 2.1 presents SGB with a Gaussian noise model; i.e. it uses a linear mean function and the least squares error function. (Algorithms 2.1-2.3 are very similar and so only Algorithm 2.3 will be described in detail below.)

With especially noisy data, robust error functions, such as the Huber error function (Huber 1964; Friedman 1999) or the log-cosh error function, can be used instead. Robust error functions reduce the effect of outliers on regression (see figure 2.3). The Huber error function is quadratic near zero and linear after a specified distance from zero. This prevents outliers from having undue influence on the regression. The log-cosh error function approximates the behavior of the Huber error function without the need to specify a transition point. Algorithm 2.2 modifies algorithm 2.1 to use the median (a robust measure of central tendency) as a starting point and to use the robust log-cosh error function. The regularization effect of the robust log-cosh error function is most obvious in the calculation of its derivative, i.e. the pseudo-residuals. As in the previous algorithm, algorithm 2.2 takes the difference between the desired output and the current function's output, but this difference is then passed through the tanh function. The tanh compresses large and small values to 1 and -1 respectively, while remaining nearly linear near 0. The mean function, however, is still the linear mean function.

Non-canonical mean functions can easily be incorporated into the algorithm if needed. To model the responses of neurons, I will adopt the Poisson noise model as is standard practice (Pillow 2005). The log-likelihood under a Poisson noise model, i.e. the error function, is shown in equation 2.10 and its derivative in equation 2.11. $R_{obs}(t)$ is the number of spikes observed at time t, and $r(t)$ is the predicted response at time t. Unfortunately, the canonical mean function for a Poisson noise model, the exponential function, is far more expansive than the nonlinear functions typically found in neurons (see figure 2.4). The output nonlinearity of neurons is

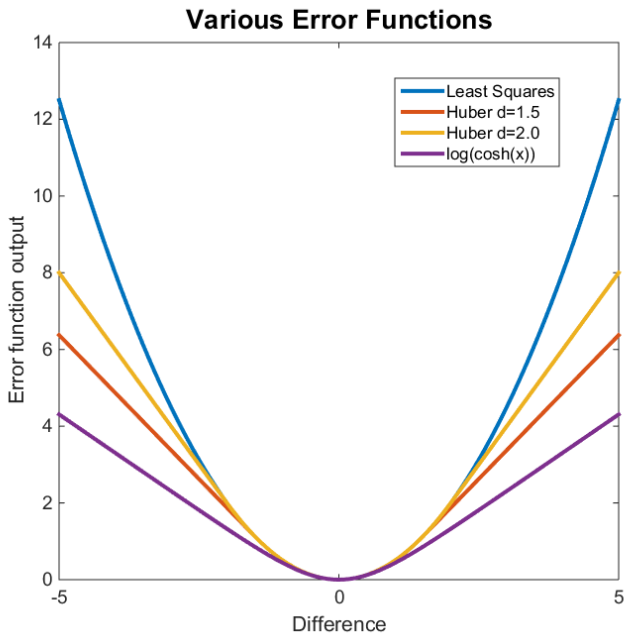


Figure 2.3: Behavior of various error functions

Each line shows the output of an error function plotted against the difference between a data point and the model prediction. The least squares error function returns exponentially increasing values as the difference increases. If there are outliers, this behavior will pull the function estimate towards the spurious data points. The use of robust error functions can mitigate this behavior.

much better approximated by a rectified-linear function (Park and Pillow 2011). A commonly used (and differentiable) alternative to the rectified-linear function is the soft-rectification function (equation 2.12). However, I adopt a simple modification of the soft-rectification function (equation 2.13) that allows a single parameter k to control the "softness" of the rectification while maintaining differentiability (see figure 2.4). This parameterization has the desirable property that at the limit of $k \to \infty$ it is identical to the rectified-linear function. In practice, only a moderate value of k, on the order of 10, is needed to closely approximate the rectified-linear function. The derivative of the Poisson log-likelihood with the modified soft-rectification mean function is shown in equation 2.14. The Poisson noise model and modified soft-rectification function are incorporated into the stochastic gradient boosting framework in Algorithm 2.3.

$$LL = \sum_t \big(R_{obs}(t) \log\big(r(t)\big) - r(t)\big) \qquad [2.10]$$

$$\frac{\partial LL}{\partial r} = \sum_t \left(\frac{R_{obs}(t)}{r(t)} - 1\right) r'(t) \qquad [2.11]$$

$$r(t) = \log\big(1 + e^{F(x(t))}\big) \qquad [2.12]$$

$$r(t) = \frac{1}{k}\log\big(1 + e^{k \cdot F(x(t))}\big) \qquad [2.13]$$

$$\frac{\partial LL}{\partial r} = \sum_t \left(\frac{R_{obs}(t)}{\frac{1}{k}\log\big(1 + e^{k \cdot F(x(t))}\big)} - 1\right)\left(\frac{e^{k \cdot F(x(t))}}{1 + e^{k \cdot F(x(t))}}\right) \qquad [2.14]$$

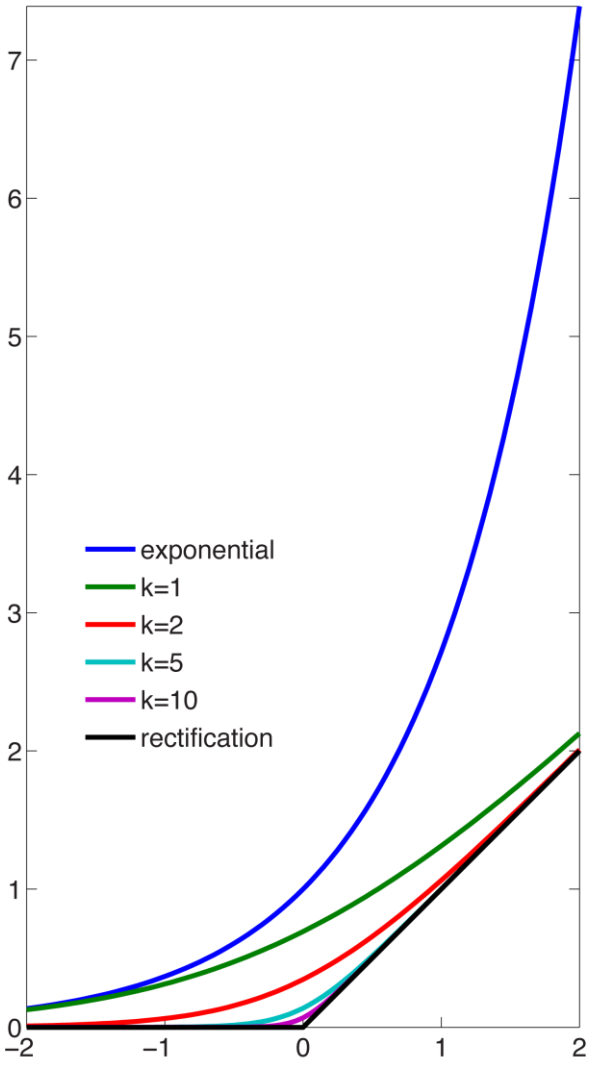


Figure 2.4: Comparison of exponential, soft-rectification, and rectification

The exponential output nonlinearity is more expansive, i.e. it grows more quickly, than the nonlinearities found in actual neurons, which are well approximated by a simple rectification function. The modified soft-rectification is a differentiable alternative to rectification whose "softness" can be controlled with the parameter k.

Algorithm 2.1

$F_0(x) = mean\big(R_{obs}(t)\big)$

$for\ m = 1{:}\ M$

$\{t\}_1^N = rand_perm\{i\}_1^N$

$\tilde{y}(t) = \dfrac{\partial LL}{\partial r} = R_{obs}(t) - F_{m-1}\big(x(t)\big)$

$a_m = argmin_a \sum_t \big(\tilde{y}(t) - h(x(t); a)\big)^2$

$\rho_m = argmin_\rho LL\left(R_{obs}, F_{m-1}\big(x(t)\big) + \rho \cdot h(x(t); a_m)\right) =$
$\sum_t \left(R_{obs} - \left(F_{m-1}\big(x(t)\big) + \rho \cdot h(x(t); a_m)\right)\right)^2$

$F_m(x) = F_{m-1}(x) + \nu \cdot \rho_m \cdot h(x; a_m)$

end

$r(t) = F_M(x)$

Algorithm 2.2

$F_0(x) = median\big(R_{obs}(t)\big)$

$for\ m = 1{:}\ M$

$$\{t\}_1^N = rand_perm\{i\}_1^N$$

$$\tilde{y}(t) = \frac{\partial LL}{\partial r} = tanh\left(R_{obs}(t) - F_{m-1}\big(x(t)\big)\right)$$

$$a_m = argmin_a \sum_t \big(\tilde{y}(t) - h(x(t); a)\big)^2$$

$$\rho_m = argmin_\rho LL\left(R_{obs}, F_{m-1}\big(x(t)\big) + \rho \cdot h(x(t); a_m)\right) = \sum_t log\left(cosh\left(R_{obs} - \left(F_{m-1}\big(x(t)\big) + \rho \cdot h(x(t); a_m)\right)\right)\right)$$

$$F_m(x) = F_{m-1}(x) + \nu \cdot \rho_m \cdot h(x; a_m)$$

end

$r(t) = F_M(x)$

Algorithm 2.3

$F_0(x) = median\left(R_{obs}(t)\right)$

$for\ m = 1{:}\ M$

$\{t\}_1^N = rand_perm\{i\}_1^N$

$$\tilde{y}(t) = \frac{\partial LL}{\partial r} = \left(\frac{R_{obs}(t)}{\frac{1}{k}\log\left(1 + e^{k \cdot F_{m-1}(x(t))}\right)} - 1\right)\left(\frac{e^{k \cdot F_{m-1}(x(t))}}{1 + e^{k \cdot F_{m-1}(x(t))}}\right)$$

$$a_m = argmin_a \sum_t \left(\tilde{y}(t) - h(x(t); a)\right)^2$$

$$\rho_m = argmin_\rho LL\left(R_{obs}, F_{m-1}(x(t)) + \rho \cdot h(x(t); a_m)\right) =$$
$$\sum_t \left(R_{obs}(t) \log\left(\frac{1}{k}\log\left(1 + e^{k \cdot \left(F_{m-1}(x(t)) + \rho \cdot h(x(t); a_m)\right)}\right)\right)\right.$$
$$\left. - \frac{1}{k}\log\left(1 + e^{k \cdot \left(F_{m-1}(x(t)) + \rho \cdot h(x(t); a_m)\right)}\right)\right)$$

$$F_m(x) = F_{m-1}(x) + \nu \cdot \rho_m \cdot h(x; a_m)$$

end

$$r(t) = \frac{1}{k}\log\left(1 + e^{k \cdot F_M(x(t))}\right)$$

Algorithm 2.3 begins by initializing the function to the median observed spike count. The algorithm then runs for M steps. At each step, N samples of data are randomly selected to create the index t and the current "pseudo"-residuals $\tilde{y}(t)$ are calculated using equation 2.14. The pseudo-residual vector is the derivative of the error function, i.e. the log-likelihood function, evaluated at the current location in function space. The parameters $a_m$of a nonlinear weak learner h are then fit to the pseudo-residuals under a least-squares cost function to find a function that moves along the gradient in function space. Note that the least-squares error function can be used to fit the weak learner regardless of the overall error function that is being minimized. This is because the algorithm only needs to find a function that approximates the current derivative. A line search is used to find a step size ρ_m for the weak learner that minimizes the overall error function. This moves the current function estimate $F_{m-1}(x)$ along the gradient toward the minimum of the error function. Finally the weak learner is added to the current function $F_{m-1}(x)$, scaled by both the step size ρ_m and a learning rate ν. Typically smaller values of ν, i.e. around 0.01 or less, lead to better generalization performance (Friedman 1999), but at the cost of slower learning and more computation. Generally N, i.e. the size of the data used for fitting each weak learner, should be set by experimentation so it is just large enough so that the weak learners have a high enough signal-to-noise ratio to perform some amount of learning that generalizes to a separate test data set.

As mentioned earlier, the nonlinear weak learner I have adopted is the IPKN fit by SCG. This choice of IPKNs as weak learners involves the selection of two hyperparameters that control the "strength" of the learner: the number of hidden units in the IPKN and the number of SCG iterations. These parameters can be considered analogous to the maximum depth of decision trees that were used as the weak learners in the original Stochastic Gradient Boosting formulation (Friedman 2002). In practice I have found the algorithm to be relatively insensitive to these specific parameter values within reasonable ranges of about 5-15 hidden units and 10-30 SCG iterations. For the analyses presented I used 8 hidden units and 25 SCG iterations.

2.7 Extracting Volterra coefficients

Given the formal equivalence between an IP kernel regression model and an IPKN, it is a simple matter to analytically recover the Volterra coefficients from a fit IP network in the same manner as from IP kernel regression model using equations 2.15 and 2.16 (Franz and Schölkopf 2006). The coefficient vector of the nth order Volterra series terms $\eta_n = (h^{(n)}_{1,1,\ldots,1}, h^{(n)}_{1,2,\ldots,1}, h^{(n)}_{1,3,\ldots,1}, \ldots)$ can be obtained by constructing a design matrix of the nth order Volterra feature space expansions of

the input weights, x^i, for each hidden unit, i.e. $\Phi_n = (\phi_n(x^1), \phi_n(x^2), \ldots, \phi_n(x^i))$, and the multiplying this matrix by the output weights α. A scalar correction factor a_n is used to adjust the coefficients to correct for the implicit feature weights in the IP kernel-induced space.

$$\eta_n = a_n \Phi_n^T K^{-1} y \qquad [2.15]$$

$$\eta_n = a_n \Phi_n^T \alpha \qquad [2.16]$$

In general, the downside to ensemble methods is the difficulty of interpreting the final model. If all I desire is good generalization performance, "black box" ensemble models are a good choice. However, model interpretation is a vital component of science and therefore more interpretable models are generally favored even if their performance is suboptimal. Ensembles of IPKNs offer a distinct advantage in this regard. Because each IP network can be converted into a set of Volterra series coefficients, averaging over IP networks is equivalent to averaging the Volterra series coefficients extracted from each model. Similarly for Stochastic gradient boosted IPKNs, I take a weighted sum (with the weights determined during the SGB algorithm) of the coefficients derived from each IPKN rather than a simple average. Thus a single set of Volterra series coefficients can be obtained that is exactly equivalent to the ensemble model.

2.8 Model selection, validation and comparison

The final remaining issue regarding the algorithm is how to select M, the number of iterations to perform. If I view the algorithm as performing gradient descent in Volterra space, I want select M to terminate the descent just before the algorithm begins overfitting to noise in the data. This can be accomplished by early stopping, i.e. evaluating the prediction of the model at each iteration on a separate test data set and then selecting the iteration where the error on the test data set is smallest (see figure 2.5).

After the final model has been selected, its performance can be evaluated in terms of its predictions on a separate validation data set (see figure 2.6). Note that this data set should be separate from training data and test data set. The use of a validation set is an objective way to compare models that use inhomogeneous polynomial kernels of different orders. Because a d-th order kernel induces a space that includes all terms from zeroth to d-th order, comparing the prediction performance

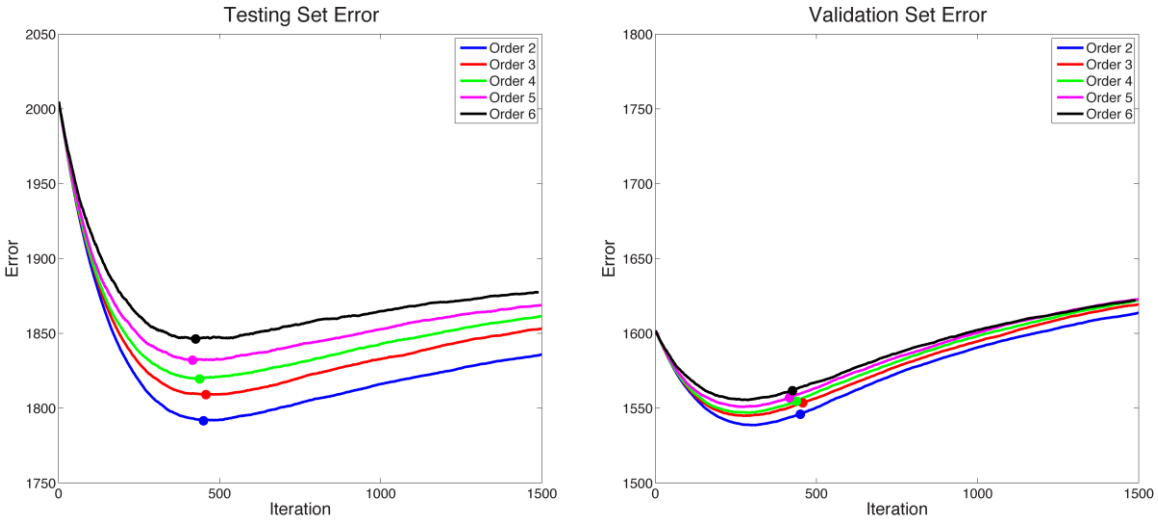


Figure 2.5: Selection of stopping point in boosting

The iteration (marked by a circle) where the error is at a minimum on the testing data set (left) is selected as the final iteration to include in the final model. The same iteration is marked on the validation data (right). Due to sampling effects on limited data, the minimum of the Testing Set Error and Validation Set Error are not exactly at the same iteration, but the differences in error between the minimum and the selected stopping iteration on the Validation set are relatively small. In all cases increasing the order of the model hurts prediction performance.

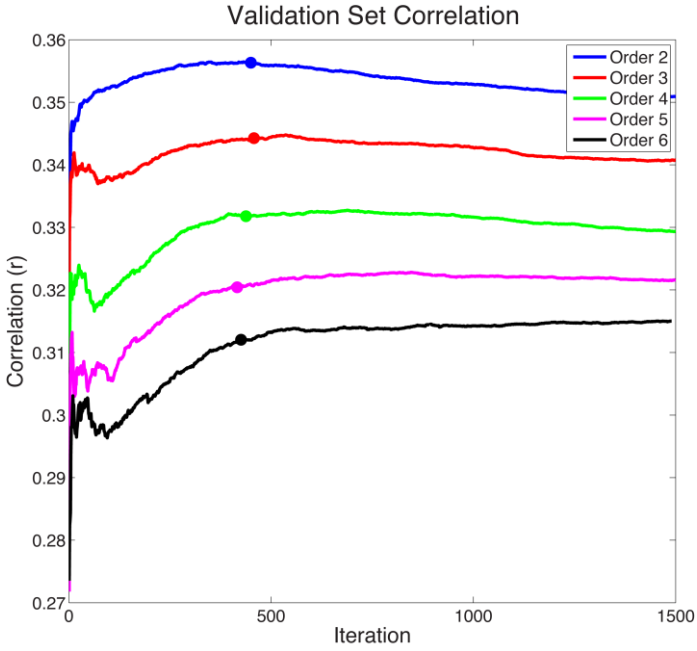


Figure 2.6: Stopping point performance on the validation set in terms of correlation

The final iteration selected by early stopping to be included in the final model is marked by a circle. The prediction performance in terms of correlation is very near the maximum on the validation data set for all model orders.

of different order models allows the importance of higher order nonlinearities to be objectively quantified.

2.9 Testing on synthetic chemical reaction data

To demonstrate the generality and power of this method for fitting high order Volterra series models, I will first apply the method to a simulated data set that has been used to test previous methods for fitting Volterra models. I generated data from a simulated polymerization reaction whose state-space representation is described by a set of coupled differential equations (Richards and Congalidis 1987; Favier, Kibangou, and Bouilloc 2012):

$$\dot{x}_1 = 60 - 10x_1 - 2.45684x_1\sqrt{x_2} \quad [2.17]$$

$$\dot{x}_2 = 80u - 10.1022x_2 \quad [2.18]$$

$$\dot{x}_3 = 0.0024121x_1\sqrt{x_2} + 0.112184x_2 - 10x_3 \quad [2.19]$$

$$\dot{x}_4 = 245.979x_1\sqrt{x_2} - 10x_4 \quad [2.20]$$

$$y = \frac{x_4}{x_3} \quad [2.21]$$

The reaction system these equations describe employs azo-bis-isobutyronitrile as an initiator and toluene as a solvent in the isothermal free radical polymerization of methyl-methacrylate. The output of the system, y, is the average molecular weight of the final product in kg/kmol. The input to the system u, is the inlet initiator flow rate. The initial conditions of the system are described in Table 2.1:

Table 2.1.

Variable	Nominal Value	Units
x_{10}	5.12796	kmol/m^3
x_{20}	0.4791035	kmol/m^3
x_{30}	0.00623093	kmol/m^3
x_{40}	87.308673	kg/m^3
u_0	0.0605	m^3/h
y_0	14012	kg/kmol

Similar to past efforts to model this system with the Volterra series (Favier, Kibangou, and Bouilloc 2012), I will generate the simulated data by inputting a signal that varies with a random magnitude between 0 and 0.1 m^3/h, but is constant for 10 time steps before changing. The length of the time step in this simulation will be 0.06h. I will use an input sequence of 20,000 time steps, the first 15,000 of which will be used to fit the model with the remaining 5,000 samples to be used for validation. However, unlike past work using the Volterra series to model this system (Favier, Kibangou, and Bouilloc 2012; Soni 2006), I will not use a linearized version of these state-space equations that is truncated at the third order, but rather attempt a more complete simulation using the numerical differential equation solver, ode45, present in MATLAB (Shampine 1994). As a consequence the, output of the modeled system is more nonlinear than in past simulations. Following the work in (Favier, Kibangou, and Bouilloc 2012), all Volterra models will use a memory length of 20 time steps, but I will test all model orders between 2nd and 10th order. Figure 2.7A shows how the prediction performance of the Volterra model increases nearly monotonically as the model order increases from 2nd to 10th order. Figure 2.7B shows the prediction a 10th order Volterra series model overlaid on the output of the system. Compared with previous efforts to apply the Volterra series to model this system, higher order Volterra series models were needed for best prediction performance. The need for higher order models is a natural consequence of using the original set of state-space equations rather than a truncated linearized set of equations. However, this result nicely demonstrates the ability of the algorithm to fit and make predictions using very high order Volterra series models.

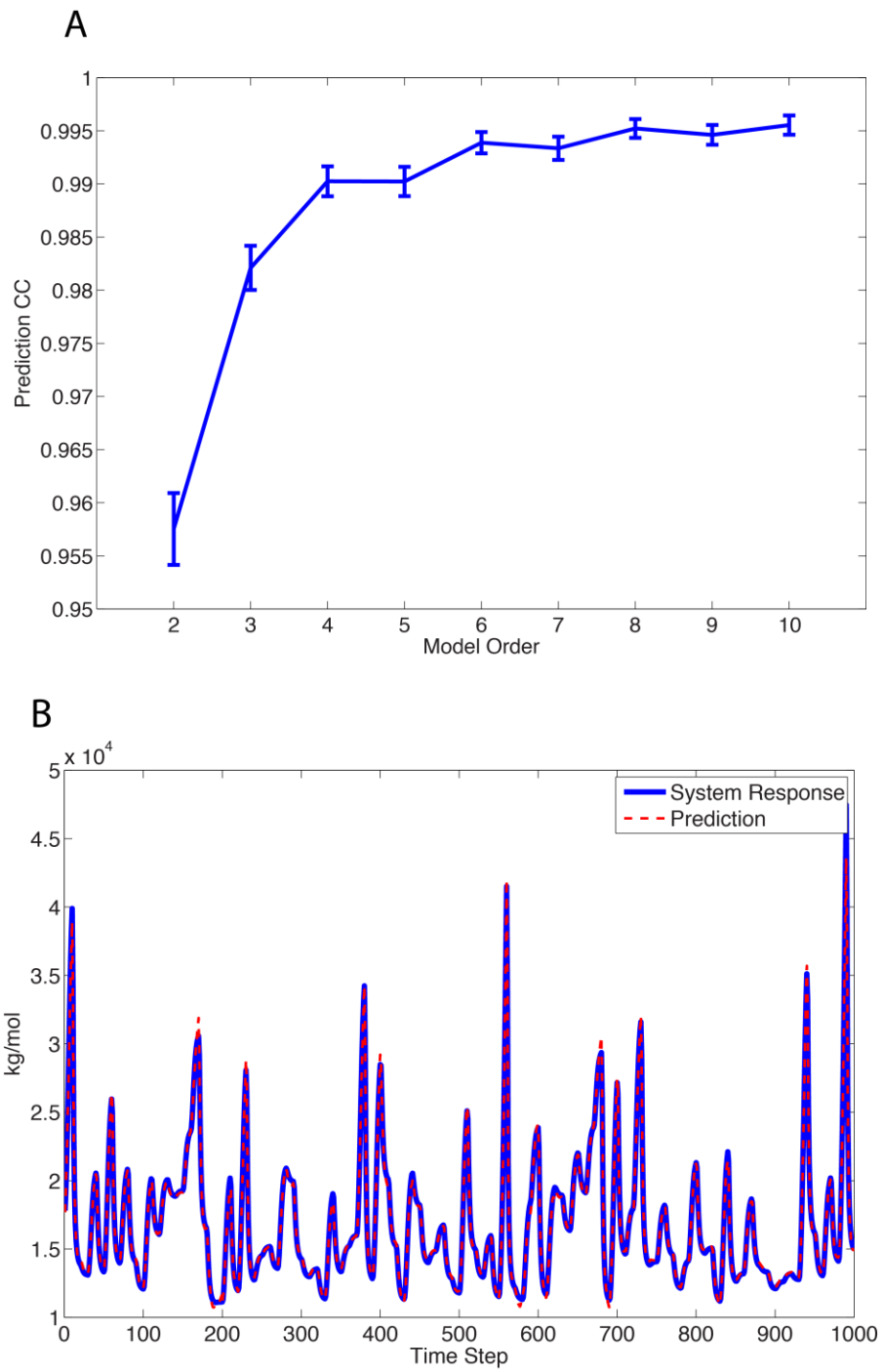


Figure 2.7: Prediction performance of Volterra models on chemical data

As shown in 2.7A, the prediction performance increases with model order. The average prediction at each order was calculated from 30 instantiations of the system with different inputs each fit with models from 2nd to 10th order. As shown in 2.7B, the predictions of a 10th order model are nearly identical to the true output of the system on the validation data.

2.10 Testing on receptive field data

To demonstrate the scalability of the algorithm to high input dimensionality and large sample size I will apply it to three sets of visual neuroscience data. I will use the algorithm to construct receptive field models of simulated V1 complex cell responses, actual V1 complex cell responses recorded from a cat and actual V4 cell responses recorded from a macaque.

2.10.1 Simulated V1 complex cells

As a proof of concept, the first set of receptive field data I applied the algorithm to are simulated V1 complex cell responses. Simulated responses were generated according to the standard "energy model" for complex cells with a pair of Gabor filters 90° out of phase, and using two alternative implementations of inhibition that have been hypothesized to exist in visual cortex: divisive inhibition (Heeger 1992) and subtractive inhibition (Holt and Koch 1997). The simulated responses are generated using the excitatory filters k^E and inhibitory filters k^I (shown in figure 2.8) and the equations:

$$R_1 = \frac{\sqrt{\sum_E (k^E \cdot s)^2}}{1 + \sqrt{\sum_I (k^I \cdot s)^2}} \qquad [2.22]$$

$$R_2 = \left| \sqrt{\sum_E (k^E \cdot s)^2} - \sqrt{\sum_I (k^I \cdot s)^2} \right|^+ \qquad [2.23]$$

The signal R_1 is the square root of the sum of the squared output of the excitatory filters, k^E, with the stimulus s, divided by the square root of sum of the squared output of the inhibitory filters, k^I, with the stimulus s, with a constant added to prevent divisions by zero. The signal R_2 is the square root of the sum of the squared output of the excitatory filters, k^E, with the stimulus s, minus the square root of the sum of the squared output of the inhibitory filters, k^I, with the stimulus s. Signal R_2 is then positively rectified to prevent negative output. To better simulate neural responses, the fitting data is generated by a Poisson process with the distribution parameter λ driven by the signals R_1 and R_2.

The stimulus s consists of 200,000 frames of white noise. The first 180,000 frames are used for fitting the Volterra models, the next 10,000 frames are used to test the

model and determine the stopping point, and the last 10,000 frames are used to validate the models. The testing and validation data will be the noiseless signals R_1 and R_2, that is the signals before Poisson sampling, so that the model fit can be assessed independently of the stochastic spiking. I fit 2nd order Volterra models to both signals with the soft rectification output nonlinearity and Poisson error function using algorithm 2.3. The correlations of the model predictions with the noiseless validation signals R_1 and R_2 were 0.80 and 0.91 respectively. (I also fit 3rd and 4th order Volterra models to these signals but they did not improve prediction performance.) I extracted all first and second order Volterra coefficients from the model as described above. As expected, the first order terms of both models exhibited no real structure (since there is no linear component of the equations) while the second order terms were highly structured.

2.10.2 Visualizing receptive field models

After extracting the Volterra coefficients from the ensemble model of a receptive field, they can then be interpreted in standard ways. In visual neuroscience the first order Volterra coefficients correspond to the Spike Triggered Average and the second order coefficients correspond to the Spike Triggered Covariance matrix. The second order coefficients extracted from the model can be used to form a symmetric

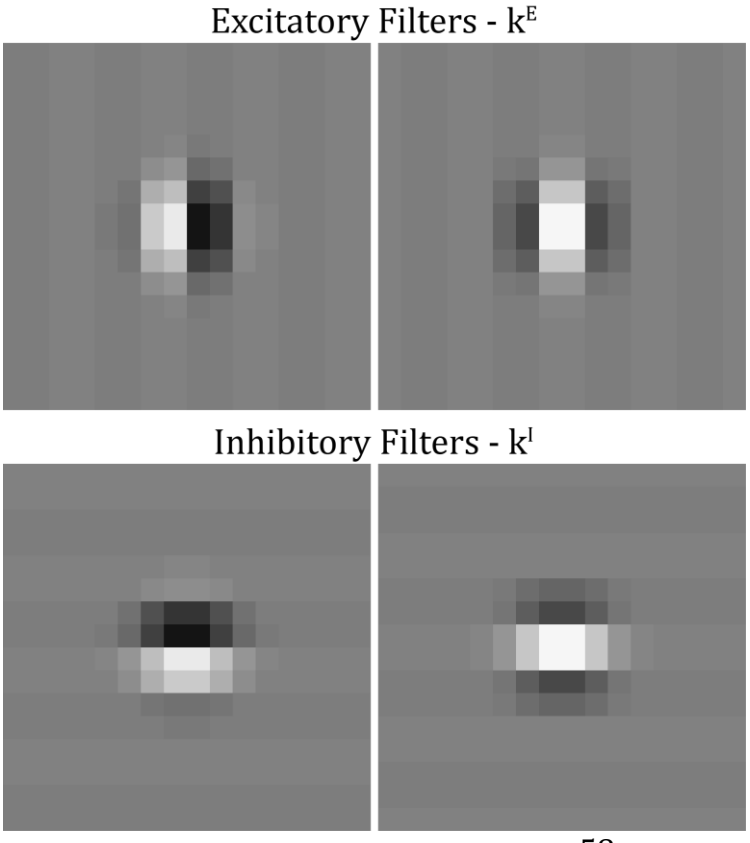


Figure 2.8: Filters used for simulated complex cells

These pairs of 16x16 filters were used to generate simulated complex cells that exhibited both phase invariant excitation and inhibition. For this simulation no temporal kernel was used and the response was just determined by the instantaneous dot product between these filters and the stimuli in equations 2.15 and 2.16.

matrix (see figure 2.9), akin to the STC matrix, which can then be subjected to eigenvector decomposition.

As with traditional STC analysis, the eigenvectors can be reshaped to the original stimulus size and visualized. Figure 2.10 shows the four significant eigenvectors of the second order terms extracted from the Volterra models of signals R_1 and R_2 reshaped to the original filter dimension (as in figure 2.8). The eigenvectors clearly resemble the original filters but with slightly different phases. Since the equations that define R_1 and R_2 create phase invariant responses, the exact phases of the eigenvectors are arbitrary and only the relative phases are significant. Clearly boosted IPKNs can extract the relevant features from phase invariant systems that possess either divisive or subtractive inhibition.

Since only second order terms were used to define the signals R_1 and R_2 it is not surprising that higher order models failed to improve prediction performance. However, higher order models may be relevant to modeling actual neural responses. The Volterra coefficients of higher order Volterra models can be reshaped into tensors, just as the second order terms were reshaped into a matrix. For example third order Volterra terms could be reshaped into a 3 dimensional tensor (i.e. a cube) which corresponds to weights on every 3-way pixel interaction. Interpreting the tensors of higher order Volterra coefficients is a heretofore unexplored area in neuroscience, but extensions to traditional methods, such as high order singular value decomposition (HO-SVD), may prove useful. The HO-SVD decompositions of third and fourth order tensors would yield the higher order analogs of eigenvectors: skew-vectors and kurtosis-vectors respectively.

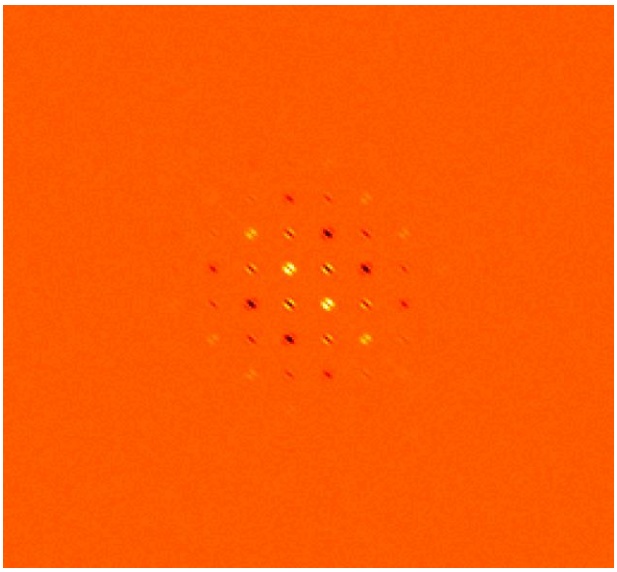


Figure 2.9: Second order pixel interaction matrix

Second order coefficients extracted from the Volterra model of the simulated signal R_2 reshaped into a matrix and reflected across the diagonal to mimic a covariance matrix, such as the STC matrix. The terms are equivalent to weights on every possible pixel interaction. The structure within the matrix can be interpreted via eigenvector decomposition.

2.10.3 Cat Area 17 complex cells

The second set of receptive field data used to test the boosted IPKN algorithm comes from recordings of cat complex cells stimulated with natural image patches (Touryan, Felsen, and Dan 2005). The stimulus set consisted of 24,000 frames of natural image patches shown at 24Hz. To enable higher temporal resolution for the receptive field model, I temporally upsampled to 48,000 frames by doubling each frame and rebinning the spikes at 48Hz. The first 43,200 frames were used for fitting the boosted IPKN models and the last 4,800 frames were used to test and validate the models.

2.10.3.1 Input basis choice

The algorithms thus far described are agnostic with respect to the choice of input basis. For receptive field estimation, the stimulus pixel values are the most obvious choice of basis (as was used in section 2.10.1). However, various linear transforms of the pixels, such as Principal Components Analysis, the Discrete Cosine Transform,

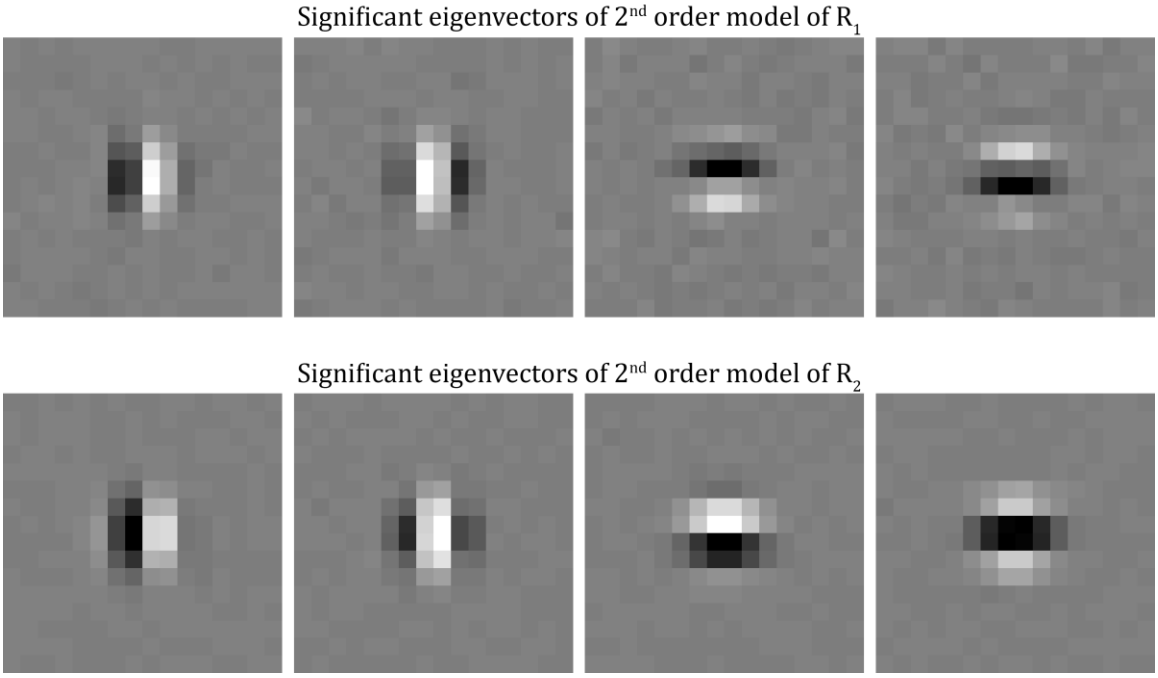


Figure 2.10: Eigenvectors of second order terms of Volterra models of R_1 and R_2

The first two eigenvectors extracted from each model clearly correspond to the excitatory filters in figure 2.5. Similarly, the second two eigenvectors closely match the inhibitory filters in figure 2.5. Though they differ in absolute phase, their relative phases are 90° apart in each pair. Therefore if these eigenvectors were used as filters in equations 2.15 and 2.16 they would behave equivalently to the true filters used.

or various wavelet transforms can be used to reduce the input dimensionality by discarding high spatial frequency information in the stimuli that is deemed irrelevant to the receptive field. Also, because these linear transforms aggregate pixel values globally (or locally) such a basis can be more sensitive to patterns relevant to the receptive field and thus improve the performance of the algorithm. I have found PCA to work well in practice and have adopted it for modeling the real receptive field data. In my analyses I selected the number of principal components such that they could account for greater than 95% of the variance of the original pixel data.

2.10.3.2 Model performance

The boosted IPKN models were able to recover a significant amount of structure from the responses of the cat complex cells. Generally second order models were sufficient to model the neural responses and higher order models did not improve prediction performance (see figure 2.11). However, higher order models performed similarly (and made very similar predictions) to the second order models. Because the higher order Volterra models implicitly define extremely large parameter spaces that include all lower order parameters, this similarity indicates that boosted IPKNs with early stopping are quite resistant to overfitting and are able to fit lower order subspaces within the higher order spaces defined by their kernel functions. As a point of comparison, it should be noted that the highest prediction correlation obtained in the original analysis of this data (Touryan, Felsen, and Dan 2005) was about r=0.38 (based on figure 6 in their paper). As shown in figure 2.9, there are four cells where the boosted IPKN model prediction is higher than r=0.38, and the best performing model has a prediction performance of r=0.71. Furthermore, the IPKN models are predicting at twice the temporal resolution used in the original study, which makes the improvement even more striking. The improvement is likely due to a combination of factors: the fact that boosted IPKN models can fit temporal response properties along with spatial selectivity (unlike the STC analysis used in the original study), the simultaneous estimation and use of first and second order terms, better regularization, and a more accurate noise model and output nonlinearity.

Figure 2.12 shows two examples of the linear coefficients and second order eigenvectors of extracted from the Volterra models of neurons from area 17 of a cat's visual cortex.

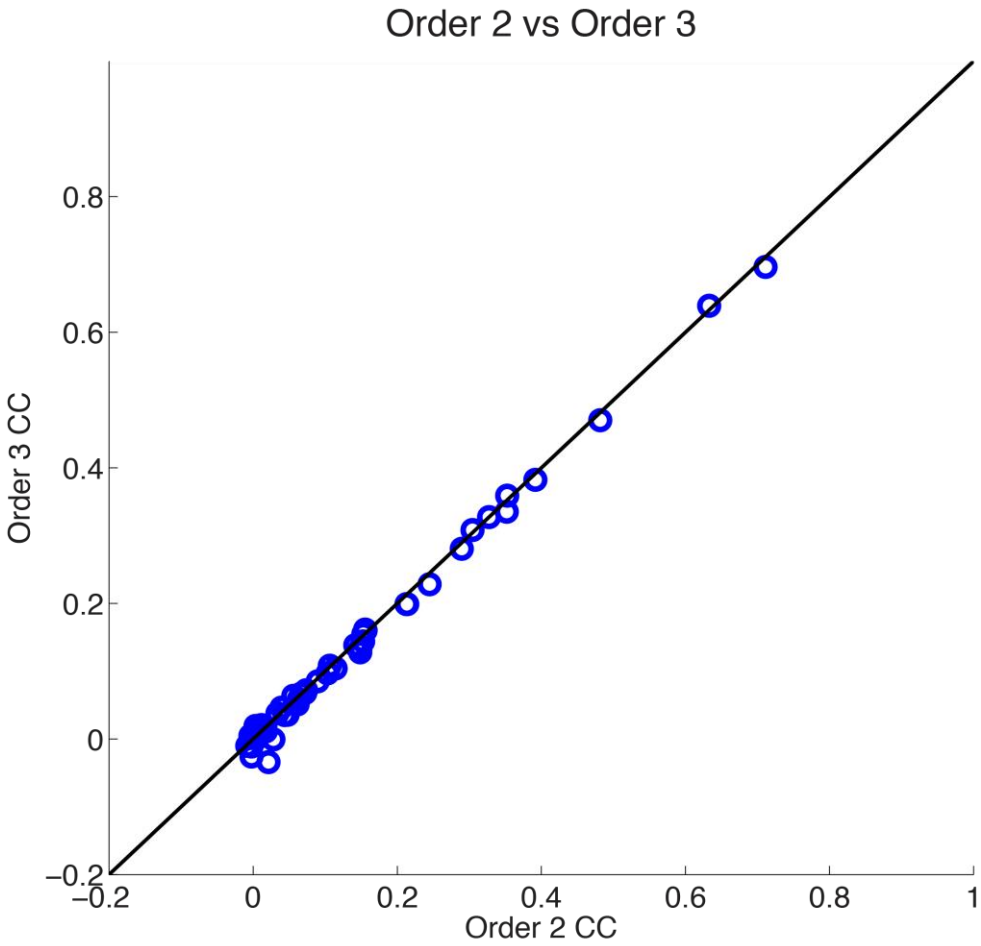


Figure 2.11: Comparison of prediction performance of 2nd and 3rd order models

The correlation of the prediction with the actual neural responses on the validation data is about the same or slightly higher for the 2nd order models of each cell relative to 3rd order models. This pattern continues with higher order models; i.e. there is a slight advantage for 2nd order models. This indicates that there is no good evidence for higher order stimulus selectivity in this data set.

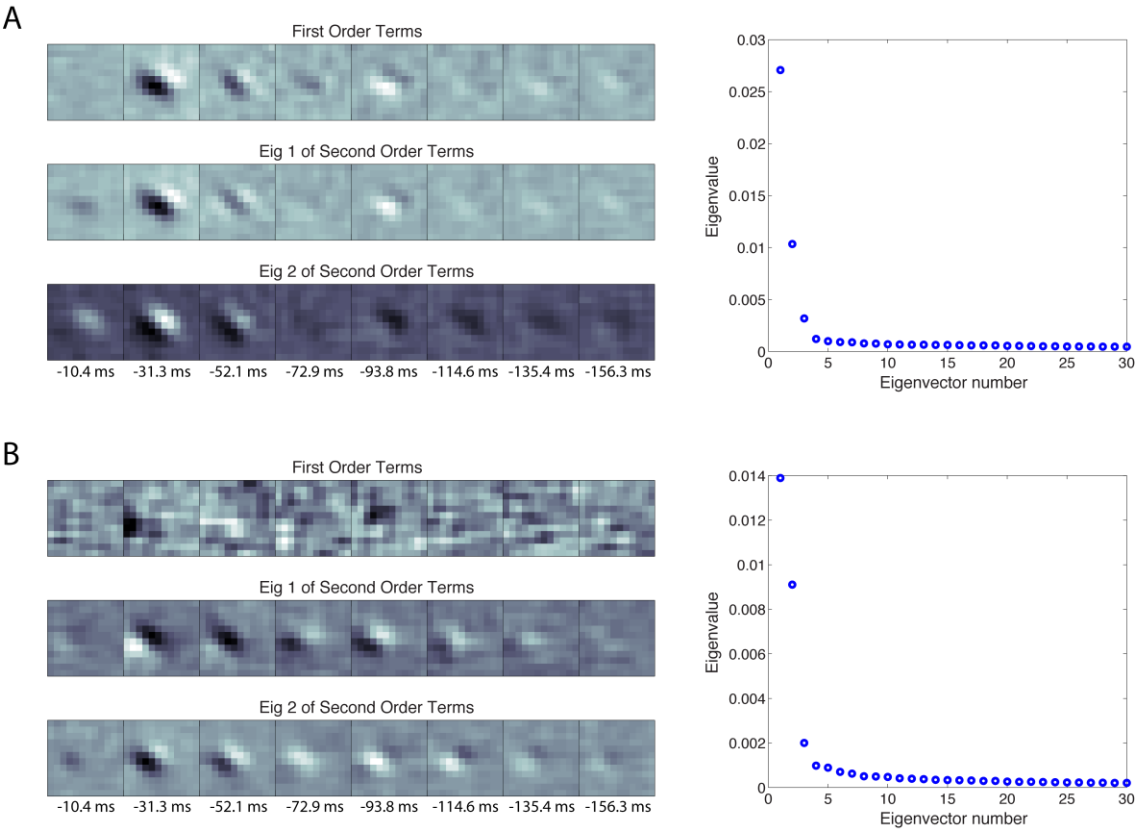


Figure 2.12: Visualizing Volterra series models of cat V1 neurons

The first order Volterra coefficients, i.e. the linear terms, are plotted such that each square corresponds to a stimulus frame, and the time before a spike increases to the right. The first two eigenvectors of the second order terms are plotted equivalently. The scree plots to the right clearly show that the first two eigenvectors of the second order terms in each model account for most of their variance. A) This cell shows both significant first order and second order components. This cell clearly has a biphasic response profile with a reversal of preferred phase happening around -72.9 ms. B) This cell shows little structure in the first order terms, but the second order eigenvectors show a graduate change in preferred phase over time, indicating this cell shows temporal frequency and direction selectivity.

2.10.4 Macaque V4 neuron

The final set of receptive field data I will apply the boosted IPKN algorithm to comes from recordings I made from macaque V4 neurons stimulated with full screen color natural movies. The stimulus set consisted of over 1 million natural movies frames taken from the documentaries "Planet Earth" and "Life" shown at 60Hz. The scene changed every ~300ms to simulate average fixation duration. As a proof of concept I will be showing the results from one of the neurons which was stable across several days of recording. The neuron was stimulated by 500,000 frames of fitting stimuli and 10,000 frames of validation stimuli.

V4 neurons have large receptive fields and a 256x256 pixel window on the stimulus was necessary to fully encompass the receptive field. Including all the RGB pixels as well as temporal delays would result in a model with over 1 million input dimensions. To reduce dimensionality, I rotated the RGB values into the YCbCr colorspace, a space with 1 luminance dimension and 2 color dimensions and then performed principle components analysis (PCA) on each dimension. I selected the first 500 PCs from the luminance dimension and the first 250 PCs from each color dimension to use as input. More luminance PCs are used to better represent the higher spatial frequency content of luminance in natural images. Figure 2.13 shows the prediction performance of 2^{nd}-4^{th} order IPKNs at each boosting iteration on data from a V4 neuron. Clearly there is a significant gain in moving beyond second order models in V4.

2.10.4.1 Interpreting the model

My attempts to visualize the model as I had done with the models in V1 yielded little in the way of interpretable results. It is of course possible that V4 tuning may not admit simple visualization. But because the model has reasonable predictive performance I wanted to compare it to previous attempts to characterize V4 neural tuning. To do so, I stimulated the model with two sets of artificial stimuli that have previously been used to demonstrate the selectivity of V4 neurons to curvature: non-Cartesian gratings (Gallant et al. 1996) and parameterized shapes (Pasupathy and Connor 2001). Several hundred stimuli were used and the stimuli that evoked the 40 highest responses are shown in figure 2.14 ordered and colored by response size. Clearly the model responded most strongly to polar and hyperbolic gratings, similar to real V4 cells (Gallant et al. 1996). Also the basic ordering of the responses to the parameterized shapes is highly reminiscent of the tuning shown in the original study (Pasupathy and Connor 2001). The model is clearly able to capture some of the complex selectivity of the V4 neuron, enough so that is predicts

responses fairly well on natural movies and exhibits tuning to simple curvature patterns and shapes similar to that seen in previous studies of V4.

Another commonly reported property of at least some V4 neurons is selectivity for color (Zeki 1973; Conway, Moeller, and Tsao 2007). However, this tuning has been shown to be influenced by context, i.e. the color tuning in the center of the receptive field can shift based on the surrounding color (Kusunoki, Moutoussis, and Zeki 2006). This shift in color tuning is thought to play a role in the computation of color constancy. To investigate whether such effects were present in the model, I showed the center and surround of the model's receptive field various pairs of isoluminant colors. The results of this investigation are shown in figure 2.15. The color of the center is plotted along the bottom of the figure and the color of the surround is indicated by the color of each line. The model shows a peak response when the center is teal and the surround pink. The V4 cell model is clearly color opponent since the response of the center to teal is almost completely attenuated when the surround is teal as well. Interestingly, the peak tuning for the center color shifts when the color of the surround changes. As the color of the surround shifts toward orange, the peak response for the center shifts rightward towards blue. And as the

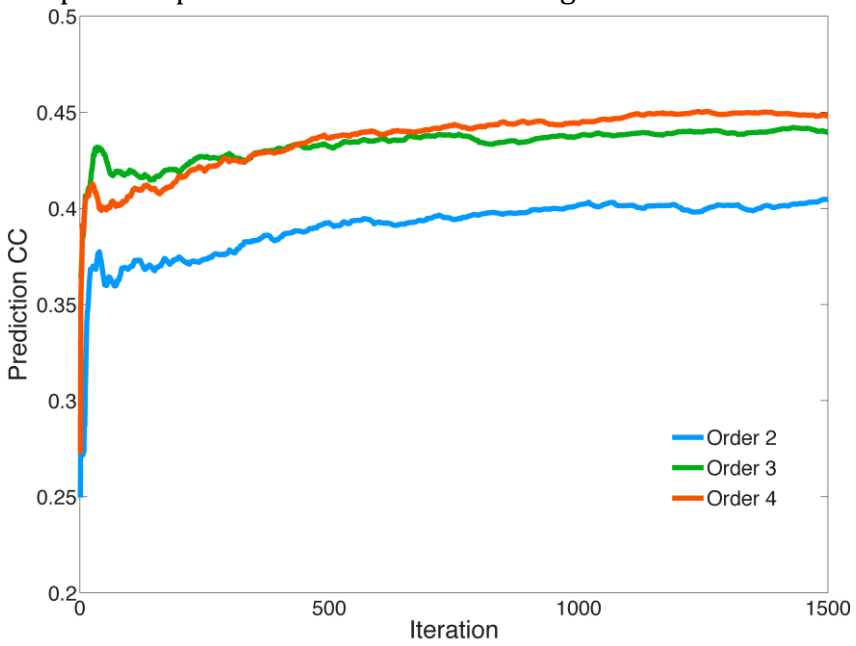


Figure 2.13: Prediction performance of various model orders on a V4 Neuron

The prediction performance at each boosting iteration of 2nd, 3rd and 4th order IPKNs. Third order models show a significant increase in performance over second order models, and fourth order models show a minor increase over third order models. This is quite unlike V1 neurons where high order models usually fare about the same or worse than second order models.

color of the surround shifts toward violet, the peak response for the center shifts leftward towards green. Were the color tuning linear, the peak response for the center would only be attenuated but not shifted. Clearly the model is able to capture the nonlinear contextual color tuning seen in V4 neurons. What is most interesting, however, is that both curvature tuning and contextual color tuning were present in the model of a single V4 neuron, indicating that neurons that exhibit these properties are not necessarily distinct populations.

2.11 Relationship to previous work

While Volterra estimation is a well-studied problem, there have been to my knowledge no approaches demonstrated capable of efficiently scaling to the size of modern visual electrophysiological data sets, i.e. data containing both a large number of input variables (~10,000) and a large number of samples (~500,000).

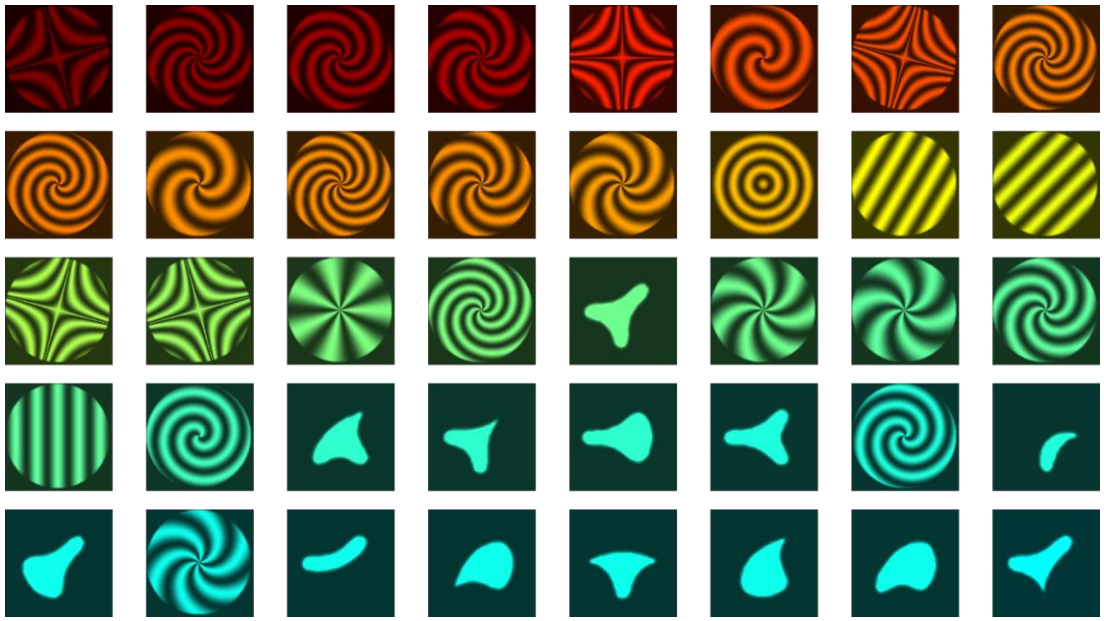


Figure 2.14: Selectivity of the model of a V4 neuron

Artificial stimuli like those used in previous studies of V4 (Gallant et al. 1996; Pasupathy and Connor 2001) were shown to the model of a V4 neuron. Only the stimuli with the 40 highest responses are shown here. The model response level is indicated by color as in (Gallant et al. 1996) with the upper-left red being the highest response. The model strongly preferred non-Cartesian gratings to Cartesian gratings and exhibited selectivity among the shape stimuli similar to that seen the original study.

However, the use of inhomogeneous polynomial kernel networks for Volterra estimation has strong similarities with some previous approaches. It has been demonstrated that a Volterra series representation could be obtained from a standard single hidden layer neural network model by using the Taylor expansion of the hidden unit activation function, typically tanh (Wray and Green 1994). Neural networks with tanh activation functions could be used in place of kernel networks as weak learners in stochastic gradient boosting to yield a very similar algorithm. However, because the Taylor expansion of tanh is an infinite series, it is impossible to constrain the model to a given Volterra series order. There is thus no way to find the optimal order for a Volterra model of a given system, and this unlimited flexibility typically results in models that overfit the training data and have worse generalization performance.

The approach most similar to the IPKN is known as the separable Volterra network (SVN) (V. Z. Marmarelis and Zhao 1997). The SVN proposed replacing the activation

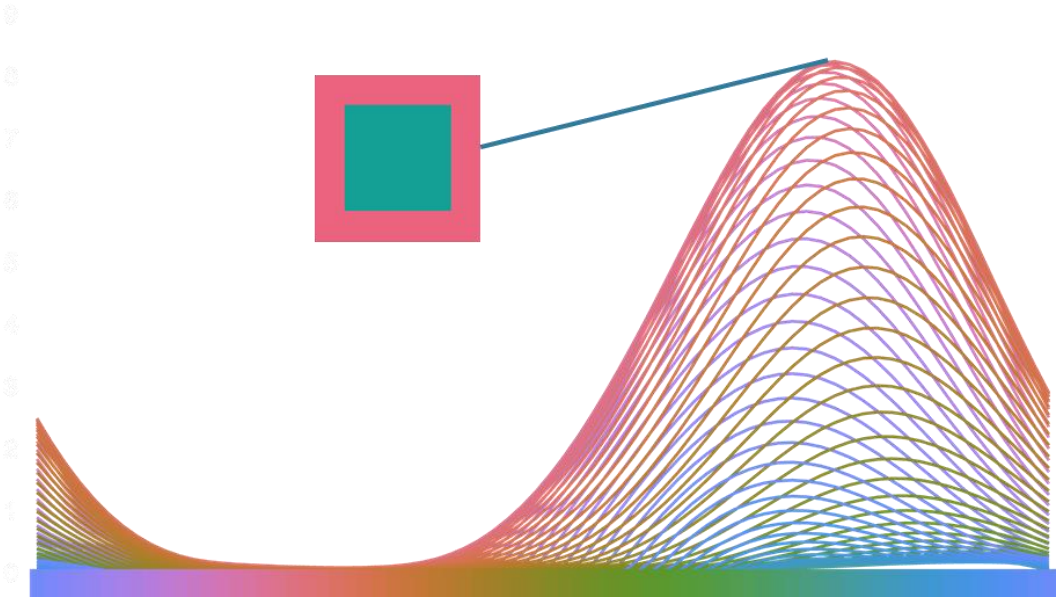


Figure 2.15: Nonlinear color tuning in the model of a V4 neuron

A wide range of center-surround pairs of isoluminant colors were shown to the model of a V4 cell. The color shown in the center is indexed by the color bar on the bottom of the figure while the surround color is indexed by the color of each line. Each line thus shows the tuning of the center when the surround is the color of the line. Note below the peak response that as the color of the surround shifts toward orange, the peak response for the center shifts rightward towards blue. And also as the color of the surround shifts toward violet, the peak response for the center shifts leftward towards green.

function in a single hidden layer neural network with polynomial functions. However, unlike the inhomogeneous polynomial activation function used by the IPKN, the polynomial functions they proposed

$$g_j(v_j) = \alpha_{0,j} + \alpha_{1,j} v_j + \cdots + \alpha_{i,j} v_j^i + \cdots \quad [2.24]$$

contained all lower order terms, as well as weights on each term which are also fit during back propagation. By using the connection to polynomial kernel regression and support vectors in 2.3 I showed that the lower order terms are implicit in the inhomogeneous polynomial kernel function and that only the input and output weights need to be fit to obtain a full Volterra model. Using the IP kernel reduces both the computational and algorithmic complexity, but may in some cases require more hidden units to match the flexibility of a SVN and the flexible parameterized activation function 2.24. It should also be noted that both the SVN and IPKN are also structurally similar to parallel cascades of linear-nonlinear (LN) paths with polynomial nonlinearities, which have also been used for Volterra estimation (Korenberg 1991).

2.12 Conclusions

Visual neuroscience data sets are becoming increasingly complex and high dimensional. New technologies must be developed to analyze the nonlinear structure contained within this myriad data. Volterra based methods, i.e. STA and STC have been some of the most successful computational modeling techniques in visual neuroscience, but face difficulties in scaling to larger numbers of dimensions, larger number of data points and higher order nonlinearities. This work was motivated primarily by the desire to scale up standard Volterra series models to higher dimensional, naturalistic and large data sets and to allow modeling of the higher order nonlinearities they may contain. On simulated and real data I have demonstrated that boosted IPKNs make this goal quite viable. Interpreting the models is still an active area of research, however. Tensor decomposition techniques, such as HO-SVD, are a straightforward way forward, but at the moment many of the tensor decomposition libraries are somewhat limited. Furthermore, even constructing higher-order tensors for moderate sized problems can exceed the memory limits of most computers.

Chapter 3

Modeling nonlinear feature selectivity in visual cortex using deep time-delay neural networks

Chapter 3

Modeling nonlinear feature selectivity in visual cortex using deep time-delay neural networks

3.1 Introduction

It is widely accepted that neurons in visual cortex encode a distributed representation of the visual world, i.e. each neuron is responsive to a limited set of features within a limited region of the visual field known as the receptive field (Hubel and Wiesel 1959; Olshausen and Field 1997). This view, which I will call the feature-coding paradigm, is intuitively appealing because it offers the possibility of understanding computation in visual cortex in terms of features that can be visualized. The feature-coding paradigm is most evident in V1 where neurons are maximally responsive to Gabor-wavelet like features of specific orientations and spatial frequencies within small receptive fields. However, it has proven extraordinarily difficult to find equivalently general quantitative descriptions of the features to which neurons respond in areas anterior to V1 (Hegdé and Van Essen 2000; Kobatake and Tanaka 1994; Willmore, Prenger, and Gallant 2010). A major source of difficulty is due to the fact that the areal size of receptive fields of cells in areas anterior to V1 increases greatly: V2 receptive fields are larger by about a factor of four relative to V1, and V4 receptive fields are larger by about a factor of four relative to V2. This increase in receptive field size vastly increases the number of pixel patterns that could fall within the receptive field, making it increasingly difficult to fully sample the space of stimuli that effectively drive a neuron during a recording session. Furthermore, neurons in areas V2 and V4 appear to nonlinearly combine features of various orientations to create selectivity to more complex patterns.

I have found deep time-delay neural networks to be extremely effective for modeling neural selectivity and extracting relevant features despite high input dimensionality and functional complexity. Time-delay neural networks are attractive for modeling neurons in poorly understood visual areas like V2 and V4 since they make no strong assumptions about the computations performed by each neuron, aside from setting a maximum window for temporal integration. It is also desirable that the neural networks be deep, i.e. have more than one hidden layer, since deeper architectures can approximate complicated nonlinear functions more efficiently than shallow neural networks (Bengio and Delalleau 2011). It is also useful that the first layer of a deep neural network consists of weights on pixels, i.e. it is a set of filters, since these can be directly visualized and used to interpret the selectivity of being modeled.

Neural networks, however, are extremely flexible models and are prone to overfitting noise in data rather than extracting meaningful structure. To deal with this issue, I have experimented with different network architectures, several training algorithms and many forms of regularization. For the sake of completeness and comparison, this chapter details some techniques I found useful, but ultimately abandoned in favor of more promising methods. The architecture, training algorithm and regularization methods I have settled on require relatively little hyperparameter tuning yet result in models that generalize very well to new data.

I will demonstrate the efficacy of these various neural network regularization techniques on the cat data described in the previous chapter as well as data I recorded from the foveal V1/V2 border in a macaque. The V1/V2 macaque data is significantly larger than the cat data, in terms of the number of input pixels, the number of stimulus frames and the number of neurons recorded. The V1/V2 macaque data is also significantly more naturalistic than the cat data since it was acquired from an awake animal watching large color natural movies, rather than from an anesthetized animal stimulated with rapidly flashed grayscale natural image patches. This dataset thus provides a strenuous test of the ability of deep neural networks to model receptive fields. The performance of the neural networks is assessed both in terms of their ability to predict new data and their ability to extract meaningful features. Deep time-delay neural networks are successful on both accounts. Not only is their prediction performance state-of-the-art, but the features extracted from V1 neurons are in excellent agreement with past studies; the features generally resemble Gabor wavelets or Gaussian blobs. This result makes the features extracted from the V2 cells extremely compelling; this is the first time V2 selectivity has been accurately visualized in the native pixel space of images. The features found for V2 neurons are more diverse than those in V1; many include combinations of Gabor wavelet-like features of multiple orientations.

3.2 Network architecture

3.2.1 Rectified linear hidden units

Deep neural networks have enjoyed a renaissance in recent years due to improvements in training algorithms and the availability of ever-faster computer hardware. One notable development has been a shift from using *sigmoid* shaped activation functions to the rectified linear, or *relu*, activation function (equation 3.1). Networks employing the *relu* activation function have proved faster to train (Dahl, Sainath, and Hinton 2013) and exhibit state-of-the-art generalization performance on many datasets (Maas, Hannun, and Ng 2013). In addition, it has been argued that the *relu* activation function is more biologically plausible than *sigmoid* activation

functions since cortical neurons rarely reach saturation (Glorot, Bordes, and Bengio 2011). Interestingly, the sigmoid activation functions were inspired by the binary nature of neuronal spiking. The *relu* function, however, approximates neuronal rate coding, wherein the code involves spikes per unit time, rather than binary spiking. The *relu* function is not smooth and can thus must be differentiated piecewise for backpropagation training (equation 3.2).

An alternative to the *relu* activation function is the *soft-plus* activation function (equation 3.3), also known as *soft-rectification*. Previous studies have shown neural networks using the *soft-plus* activation function are outperformed by networks using the *relu* function (Glorot, Bordes, and Bengio 2011). Thus I adopt a modification to the soft-plus function (equation 3.4). This is the same function used in the previous chapter as the non-canonical mean function in the stochastic gradient boosting algorithm. As previously described in section 2.6, the parameter k controls the "softness" of the rectification (see figure 3.1).

I have found networks employing the modified *soft-plus* function to be less susceptible to getting stuck during training (which allows for larger step sizes) and to generalize as well or slightly better than networks using the *relu* activation function. It is interesting to note that the derivative of the *soft-plus* function (equation 3.5) is in fact the sigmoid function.

$$a(x) = \max(0, x) \tag{3.1}$$

$$\frac{\partial a}{\partial x} = \begin{cases} 1 \; if \; x > 0 \\ 0 \; if \; x \leq 0 \end{cases} \tag{3.2}$$

$$a(x) = \log(1 + e^{x}) \tag{3.3}$$

$$a(x) = \frac{1}{k}\log(1 + e^{k \cdot x}) \tag{3.4}$$

$$\frac{\partial a}{\partial x} = \frac{e^{k \cdot x}}{1 + e^{k \cdot x}} = \frac{1}{1 + e^{-k \cdot x}} \tag{3.5}$$

3.2.2 Time-delay neural networks

Fitting spatio-temporal receptive field models to the responses of neurons stimulated with natural movies is a difficult nonlinear time series-modeling problem. Time-delay neural networks (TDNN), a simple extension of neural networks, have long been applied to other difficult time-series modeling problems, such as speech recognition (Waibel et al. 1989). In TDNNs the hidden units receive input from the previous layer's current and past outputs up to a set number of time-delays. Time-delays can in principal be added at any layer, but to date only neural networks with time-delays in the first hidden layer (see figure 3.2a) have been used to model receptive fields (Prenger et al. 2004). Using time-delays in the first hidden layer forces the network to learn a set of spatiotemporal filters. However, when input dimensionality is large, as is the case in receptive field modeling, using time-delays in the first hidden layer can substantially increase the number of weights in the network. This is because a weight is required for each input dimension at each

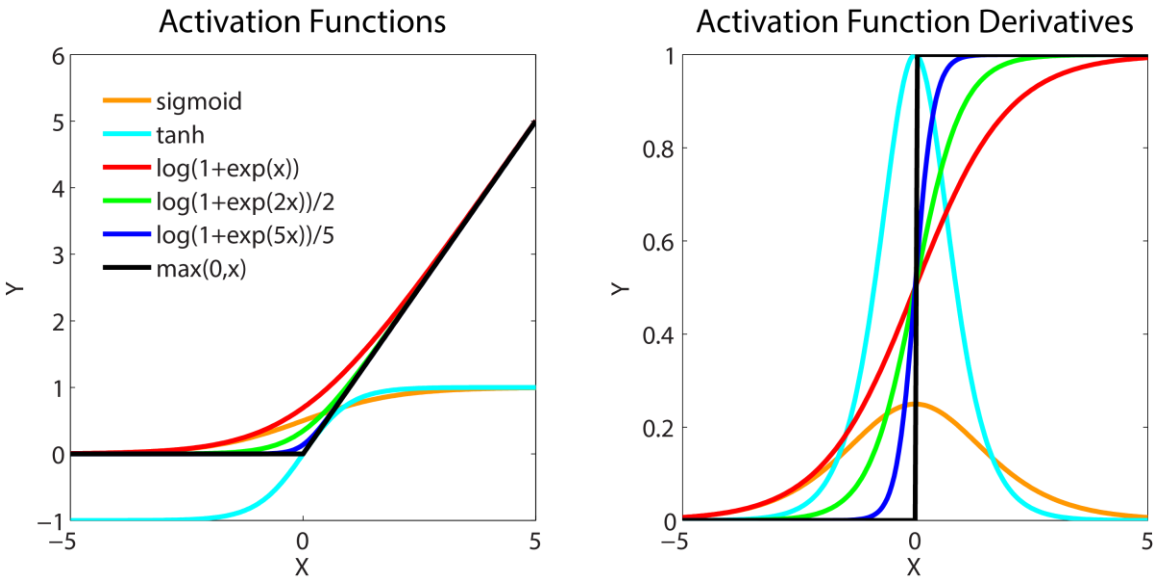


Figure 3.1: Common activation functions and their derivatives

The *sigmoid* and *hyperbolic tangent* (tanh) activation functions have long been the standard activation functions for hidden units in neural networks. In recent years the *relu* activation function, $\max(0, x)$, has gained popularity since it seems to make networks easier to train and typically improves generalization performance. The *soft-plus* function (red line) has been proposed as a differentiable alternative to rectification but has not been widely adopted as it was not seen to offer tangible benefits. Its usefulness may be improved, however, by changing a single parameter that alters the sharpness of the transition (green and blue lines). The derivatives for each activation function are plotted in the right panel.

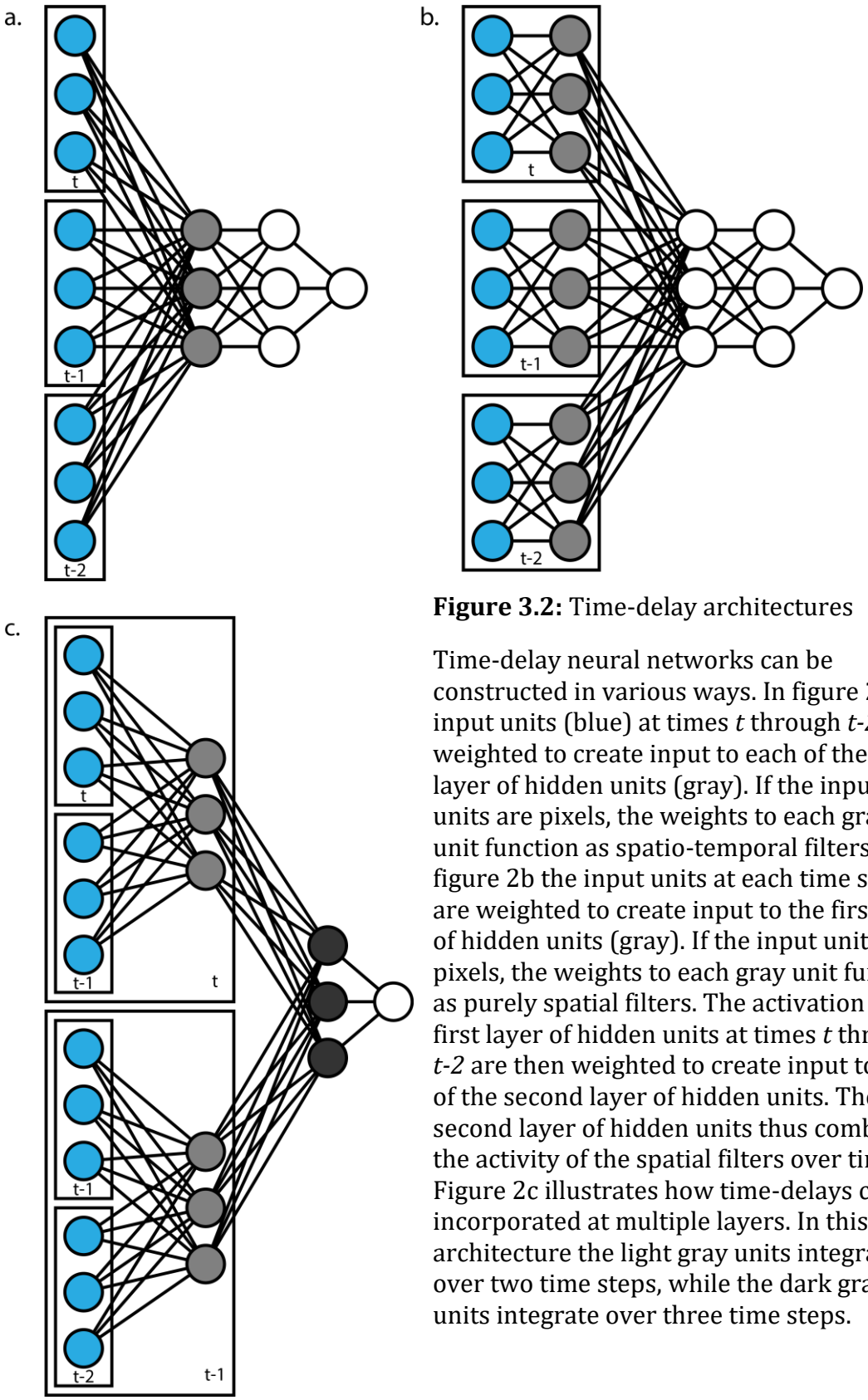


Figure 3.2: Time-delay architectures

Time-delay neural networks can be constructed in various ways. In figure 2a the input units (blue) at times *t* through *t-2* are weighted to create input to each of the first layer of hidden units (gray). If the input units are pixels, the weights to each gray unit function as spatio-temporal filters. In figure 2b the input units at each time step are weighted to create input to the first layer of hidden units (gray). If the input units are pixels, the weights to each gray unit function as purely spatial filters. The activation of the first layer of hidden units at times *t* through *t-2* are then weighted to create input to each of the second layer of hidden units. The second layer of hidden units thus combines the activity of the spatial filters over time. Figure 2c illustrates how time-delays can be incorporated at multiple layers. In this architecture the light gray units integrate over two time steps, while the dark gray units integrate over three time steps.

time-delay. If instead time-delays are used only in the second hidden layer, the first hidden layer will learn static spatial filters while the second hidden layer will learn useful ways to combine the outputs of these spatial filters across time (see figure 3.2b). Despite the separation of the spatial and temporal representations, such a network is not limited to space-time separable representations. For example, a unit in the second hidden layer could represent the direction of motion by temporally switching between first layer filters that resemble phase-shifted Gabor filters.

3.3 Fitting deep neural networks

3.3.1 Stochastic gradient descent

The first highly successful algorithm for training neural networks is known as backpropagation (Rumelhart, Hinton, and Williams 1986a). Backpropagation is essentially the same as the gradient descent algorithm described in section 1.2.4.4, but adapted to work with neural networks. The gradient calculated from the output of the neural network is propagated backwards (using the chain rule for differentiating compositions of functions from basic calculus) toward the input layer in order to calculate the gradient with respect to each weight in the model. Stochastic gradient descent (SGD) modifies the backpropagation algorithm by computing the gradient using small samples of data rather than the whole data set (Bottou 1991). SGD can operate "online", by using data samples as they arrive, or use mini-batches, typically 10-100 data points randomly sampled from the data set. Relative to standard back-propagation, SGD can scale more efficiently to large data sets and is much less likely to end up in bad local minima due to its stochastic nature. For these reasons, SGD has become the de facto standard optimization algorithm for fitting neural networks (Bottou 1991; Bottou 2012).

At each iteration, SGD uses the sampled data to compute the gradient of the error function with respect to the model weights, $\nabla E(w^{t-1})$, and then multiplies the gradient by a learning rate (also called step size), ε^t, to calculate the update for the model weights (equation 3.6). The update is then applied (equation 3.7) and a new gradient is calculated using a different sampling of the data.

$$\Delta w^t = -\varepsilon^t \nabla E(w^{t-1}) \qquad [3.6]$$

$$w^t = w^{t-1} + \Delta w^t \qquad [3.7]$$

These equations constitute the basic definition of SGD. However, in practice most people use one of the many variants of SGD. There are many simple modifications to this basic algorithm that speed convergence and help prevent overfitting to noise in the data.

3.3.1.1 Rprop and RMSprop

A major problem in applying gradient methods such as SGD to deep neural networks is that the magnitude of the gradient signal tends to shrink as it is backpropagated through the network. Weights near the input layer may see only very small gradients and will consequently change very little. This can slow or even prevent convergence. There is considerable literature on methods to ameliorate this problem by adaptively changing the effective step size for each weight (Duchi, Hazan, and Singer 2011; Zeiler 2012). One of the simplest and most effective techniques is resilient back-propagation, also known as Rprop (Riedmiller and Braun 1992). Rprop disregards the magnitude of the gradient for each parameter and instead uses only the sign of the gradient for each parameter multiplied by a small learning rate. Note that using the sign of the gradient is equivalent to dividing the gradient by the magnitude (absolute value) of each element.

$$\Delta w^t = -\varepsilon^t sign(\nabla E(w^{t-1})) = -\varepsilon^t \frac{\nabla E(w^{t-1})}{|\nabla E(w^{t-1})|} \qquad [3.8]$$

To speed convergence, the learning rate is heuristically adapted for each weight. The standard Rprop heuristic is to multiply the learning rate by 1.2 if the gradient has the same sign as the last iteration, and multiply by 0.5 if the sign of the gradient changes across iterations. To prevent the learning rate from growing pathologically large or small a maximum and minimum step size are defined as well. Rprop has most of the advantages of second-order optimizers (i.e. Hessian based methods) without the added complexity. In second-order methods the gradient is scaled using a quadratic approximation of the error surface such that gradient directions with high curvature are scaled down, and gradient directions with low curvature are scaled up. This prevents second order optimizers from getting stuck in wide shallow (low curvature) parts of the error surface and speeds convergence. The adaptive step size in Rprop provides similar behavior because steps are scaled up in directions where the gradient consistently has the same sign (which is indicative of a region with low curvature) and scaled down when the sign of the gradient oscillates (which is indicative of a region with high curvature). Rprop works well with batch learning, i.e. using the whole data set for gradient calculations, but can have problems with mini-batch learning due to the noisiness of the gradient estimates and its aggressive heuristic adaptation.

However, a recently developed technique known as RMSprop (Tieleman and Hinton 2012) , maintains the advantages of Rprop but works well with mini-batch learning. RMSprop works by keeping a running estimate of the expected square of the

gradient (equation 3.9) and then dividing the current gradient by the square root of this estimate (equation 3.10). RMSprop thus uses a root-mean-square (RMS) estimate of the expected gradient magnitude. The α parameter controls how quickly this running estimate can change.

$$r^t = \alpha\, r^{t-1} + (1-\alpha)(\nabla E(w^{t-1}))^2 \qquad [3.9]$$

$$\Delta w^t = -\varepsilon^t \frac{\nabla E(w^{t-1})}{\sqrt{r^t}} \qquad [3.10]$$

Like Rprop, RMSprop can use adaptive step sizes to provide pseudo curvature information. However, due to the noisiness of mini-batch learning, the adaptation is usually far more conservative. The RMSprop step size is adapted by multiplying the learning rate for each parameter by $(1+\gamma)$ when the gradient has the same sign as the last step and multiplying the learning rate by $(1-\gamma)$ when the gradient changes sign from the last step. I use $\gamma = 0.001$ so that the step sizes adapt slowly and are thus not sensitive to mini-batch noise.

3.3.1.2 Momentum

A very common modification to SGD is the addition of a momentum term to the weight update rule (Rumelhart, Hinton, and Williams 1986b). The momentum term is simply a fraction the weight update at the previous iteration:

$$\Delta w^t = -(1-p^t)\varepsilon^t \nabla E(w^{t-1}) + p^t \Delta w^{t-1} \qquad [3.11]$$

Adding a momentum term distributes gradient information over several weight update iterations, thus smoothing the trajectory through weight space. This tends to damp oscillations and speed up convergence. While the momentum term can be kept at a fixed value, it has become commonplace to gradually increase the amount of momentum (by increasing p^t) during fitting to ensure convergence.

3.3.1.3 Nesterov momentum

Nesterov momentum is a simple modification to standard momentum which is even more effective in stabilizing gradient descent and preventing oscillations (Sutskever et al. 2013). To compute Nesterov momentum, the gradient is not calculated with respect to the current parameter values, but with respect to where the current momentum term will move the parameters (equation 3.12). This allows the gradient to correct for errors that momentum introduces (such as overshooting) and helps stabilize the descent path.

$$\Delta w^t = -(1-p^t)\varepsilon^t \nabla E(w^{t-1} + p^t \Delta w^{t-1}) + p^t \Delta w^{t-1} \quad [3.12]$$

Nesterov momentum can be combined with RMSprop by modifying the gradient calculation to use Nesterov momentum as shown in equations 3.13 and 3.14.

$$r^t = \alpha\, r^{t-1} + (1-\alpha)(\nabla E(w^{t-1} + p^t \Delta w^{t-1}))^2 \quad [3.13]$$

$$\Delta w^t = -(1-p^t)\varepsilon^t \frac{\nabla E(w^{t-1} + p^t \Delta w^{t-1})}{\sqrt{r^t}} + p^t \Delta w^{t-1} \quad [3.14]$$

I found that tuning the rate of momentum increase was both necessary and difficult and that increasing the momentum during training usually led to worse solutions than using a fixed value for the momentum term. So for the experiments presented below, I used RMSprop with a fixed Nesterov momentum term of $p = 0.5$. Also, I set the parameter α to be 0.9 as suggested in (Tieleman and Hinton 2012).

3.4 Regularizing deep neural networks

3.4.1 Bayesian regularization

Bayesian regularization methods can be used to impose prior distributions on the weights that reflect knowledge about the system. Bayesian methods have a long history of use with neural networks (Mackay 1995). When fitting such flexible models to noisy data, the use of priors can be extremely beneficial to constrain the solution space. Using priors is equivalent to adding penalty terms to the error function. As described in 1.2.4, the addition of an L1-norm penalty on the weights corresponds to a Laplacian prior which enforces sparseness, while the addition of an L2-norm penalty on the weights corresponds to a Gaussian prior which enforces smoothness. For additional flexibility, penalties can be linearly combined and different penalties can be applied to different layers of the network. For example, if I believe that the features learned in the first layer should be both sparse and smooth, I can apply a linear combination of L1 and L2 penalties to the weights only in the first layer. An L2 penalty could then be applied to the weights in the rest of the network.

To incorporate the Bayesian priors into the training process one needs only to calculate the gradient of corresponding penalties along with the error function. The gradient of an L1-norm penalty is simply the sign of each term, i.e. +/- 1, as in equation 3.15. The gradient of an L2-norm penalty is added as the second term of equation 3.17. A linear combination of the gradients of the L1-norm and L2-norm

penalties is shown in equation 3.16. Equations 3.16 and 3.17 implement the configuration described in the previous paragraph: a linear combination of L1 and L2 penalties applied to the weights in the first layer, w_1, and an L2 penalty applied to the weights in the rest of the network, $w_{\sim 1}$.

$$\nabla_{w_1} Ep = \nabla_{w_1} E + \alpha * sign(w_1) \quad [3.15]$$

$$\nabla_{w_1} Ep = \nabla_{w_1} E + \alpha * ((1-\beta) * sign(w_1) + \beta * w_1) \quad [3.16]$$

$$\nabla_{w_{\sim 1}} Ep = \nabla_{w_{\sim 1}} E + \gamma * w_{\sim 1} \quad [3.17]$$

When using RGB color image as inputs, however, it is necessary to modify the L1 penalty; the input filters should be sparse spatially, not sparse across RGB channels. The group-LASSO (Yuan and Lin 2006), is therefore a natural choice, as it enforces sparseness at the level of groups of channels rather than individual channels. Defining the RGB values at each pixel to constitute a group and using a group-LASSO penalty on the first layer weights enforces spatial sparseness in the RGB filters. The gradient of the group LASSO penalty is shown in the second term of equation 3.18. Group identity is indexed by k. The connection to the LASSO/L1 penalty can be seen by considering that if a group has only single member, this term is equivalent to $sign(w_1)$. Equation 3.19 combines the group LASSO penalty with an L2 penalty.

$$\nabla_{w_1} Ep = \nabla_{w_1} E + \alpha * \frac{w_1^k}{\left\| w_1^k \right\|_2} \quad [3.18]$$

$$\nabla_{w_1} Ep = \nabla_{w_1} E + \alpha * ((1-\beta) * \frac{w_1^k}{\left\| w_1^k \right\|_2} + \beta * w_1) \quad [3.19]$$

The combination of equations 3.19 and 3.17 are thus well suited for regularizing neural network models of the macaque V1/V2 data; equation 3.19 will make the filters in the first layer both spatially sparse and smooth and equation 3.17 will enforce smoothness on the rest of the network. A downside of using Bayesian regularization, however, is the proliferation of hyperparameters; each of the penalty terms has a weighting hyperparameter that should be optimized. For instance, using a combination of group-LASSO and L2-norm penalties on the first layer (equation 3.19) and a L2-norm penalty on the weights of the rest of the network (equation

3.17) introduces three hyperparameters: α, β, γ. Due to the computational expense of training the networks, optimizing each hyperparameter for the model of each neuron is intractable. However, as demonstrated in figure 3.3, it is possible to determine conservative values for each hyperparameter that improve prediction performance across many cells in the macaque V1/V2 data and don't significantly hurt the performance of any of the models. (Note that DropConnect discussed in section 3.4.3 was also used in this analysis.) However, further gains are likely possible by optimizing the hyperparameters for each model. Thus for the rest of the analyses to be presented, I used forms of regularization that are adaptive (Wager, Wang, and Liang 2013) and thus require substantially less hyperparamter fiddling.

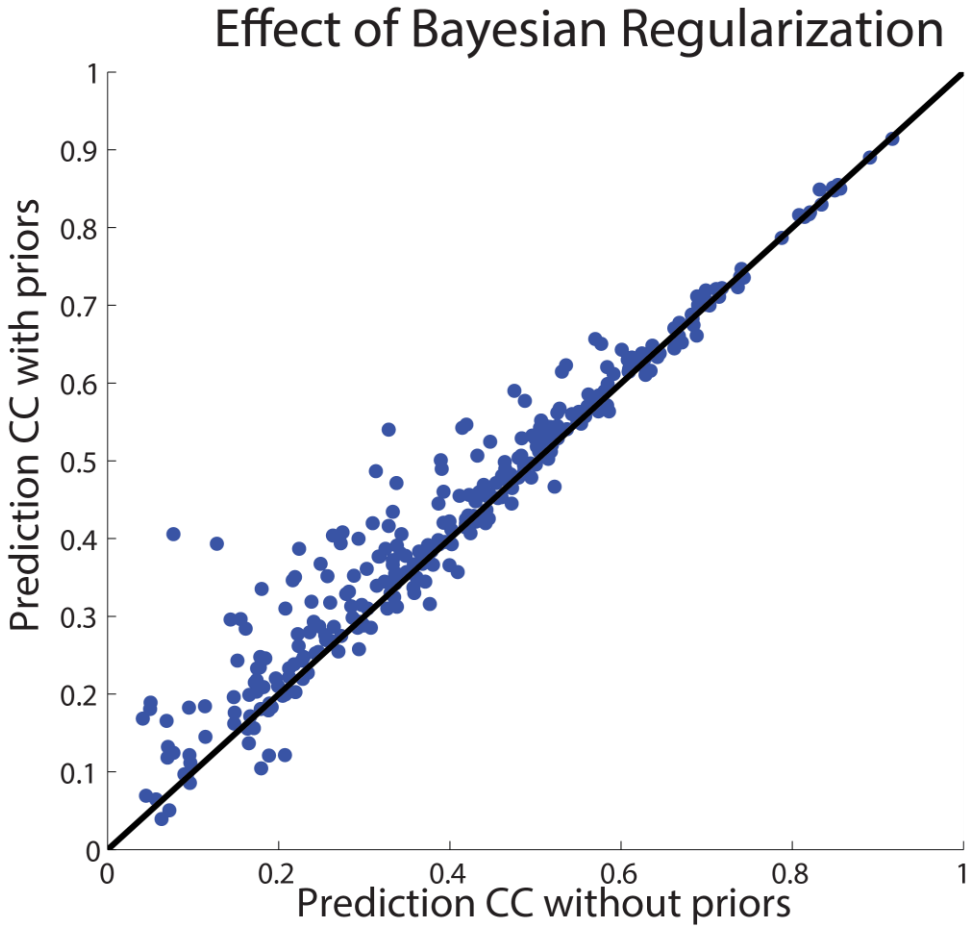


Figure 3.3: Assessing the improvement in neural network generalization performance due to Bayesian regularization in macaque V1/V2 data

Two models were fit to each V1/V2 neuron using DropConnect regularization. One of the models used Bayesian priors in the form of the group LASSO penalty on the filter weights and an L2 penalty on the weights in the rest of the network. In nearly all cases the penalized networks predicted neural responses to new data as well or better than the unpenalized networks.

3.4.2 Dropout

Dropout was recently developed as a simple and effective technique for regularizing neural networks (Hinton et al. 2012). Dropout is accomplished by randomly removing a set percentage of the hidden units at each iteration of stochastic gradient descent (e.g. hidden units in a certain layer are retained with probability p, typically ~0.5). In a given iteration, the removed hidden units do not contribute to the gradient calculation and thus their input and output weights are not updated. Because each hidden unit cannot depend on any of the other hidden units being present, they are prevented from co-adapting. Each hidden unit must therefore learn a useful feature that does not depend on the presence of other features, i.e. each feature independently contributes to the model. In a deep neural network, Dropout can be applied at any or all layers of hidden units. However, as more units are dropped out it takes longer and longer for the training of the network to converge. Thus it is typical to perform Dropout on either the first or last layer, or on both (Dahl, Sainath, and Hinton 2013; Srivastava et al. 2014). For my analyses, I found dropping out the first layer led to the best performance and most interpretable features. Because the first layer acts directly on pixels, the first layer weights form a bank of rectified filters which can be directly visualized. Dropout seems to help insure that all the filters are independently useful for predicting responses and this in turn seems to improve their visual appearance and interpretability.

3.4.3 DropConnect

DropConnect is a generalization of Dropout in which a random fraction of the individual connections between hidden units are randomly dropped rather than whole hidden units (see figure 3.4) (Wan et al. 2013). DropConnect can thus explore many more patterns of connectivity during training than Dropout. DropConnect fundamentally requires two (or more) hidden layers, i.e. a deep architecture, whereas Dropout was originally formulated using only one hidden layer (Hinton et al. 2012) and then later extended to deep architectures (Dahl, Sainath, and Hinton 2013).

However, through my experience applying both Dropout and DropConnect to my data, and through conversations with other researchers, I have come to the conclusion that Dropout seems to work (slightly) better in most situations. This may be because the removal of whole hidden units is a larger perturbation of the network which better prevents coadaptation. Thus for the analyses presented below, I have used Dropout in the first hidden layer rather than DropConnect between the first and second layers.

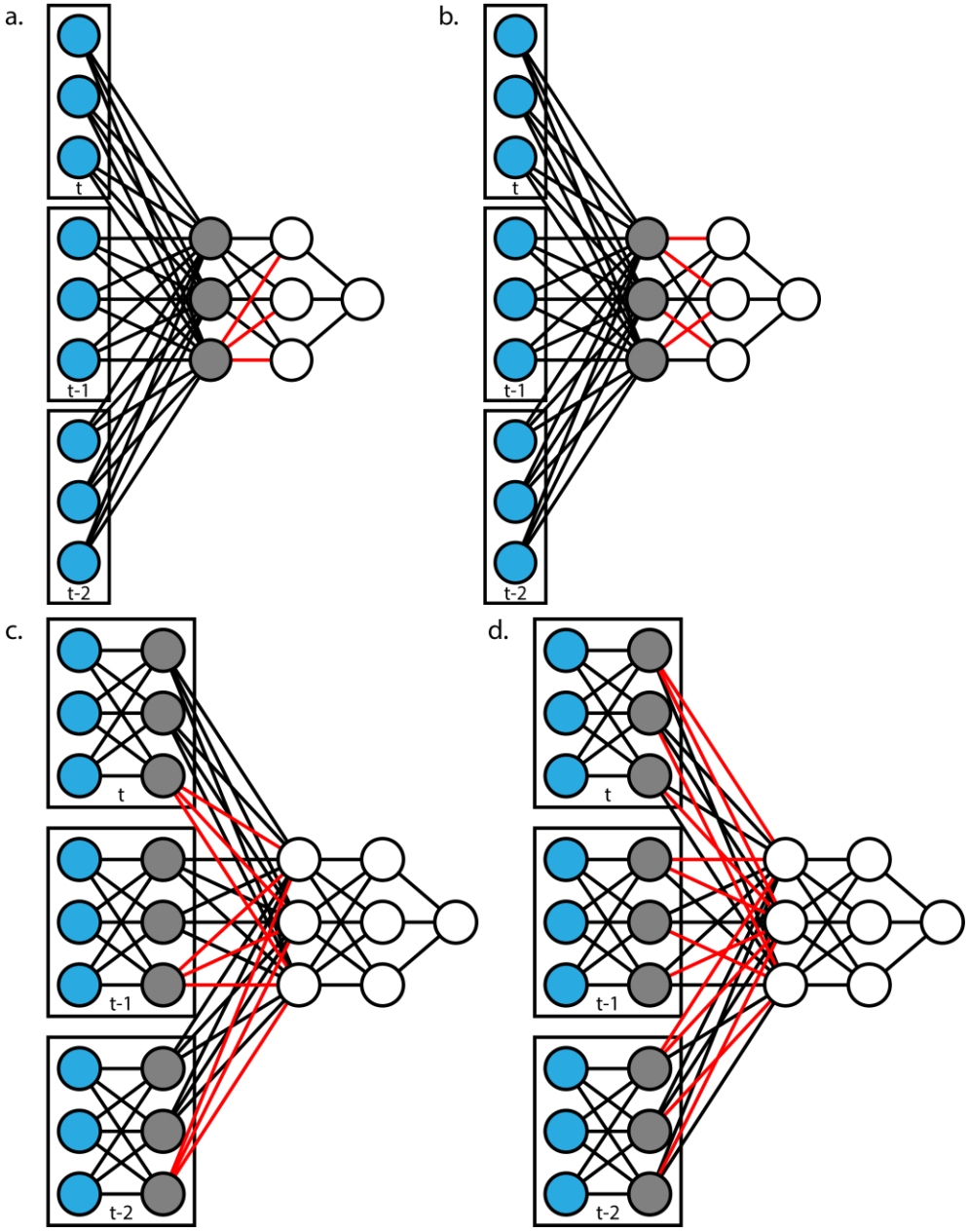


Figure 3.4: DropOut and DropConnect on time-delay neural networks

DropOut is illustrated in figure 3.4a and 3.4c where the bottom gray hidden unit is dropped out, which is equivalent to zeroing out the red connections. DropConnect is demonstrated in figure 3.4b and 3.4d where a subset of the connections, shown in red, between the gray and white hidden units are zeroed out. Figures 3.4a and 3.4b represent the same pattern of dropped connections as 3.4c and 3.4d respectively, but with different time-delay architectures. Figures 3.4c and 3.4d have more dropped connections than 3.4a and 3.4b, but only because the same connections at all time-delays are dropped.

3.4.4 Max-norm regularization

Max-norm regularization constrains the norm of the weight vector to each hidden unit to be less than or equal to a fixed constant c. This constraint controls the size of the weights within the network and prevents them from exploding even with large step sizes. Max-norm regularization has been found to be particularly beneficial when combined with Dropout and large initial step sizes during training (Srivastava et al. 2014). This combination lets the network explore many potential configurations during training without getting stuck in degenerate locations. Max-norm regularization is implemented by a simple rule: after each weight update, simply renormalize by division any weight vector whose norm exceeds the chosen value (Hinton et al. 2012). For the results presented below, I used have used max-norm regularization to constrain the weights for each filter in the first layer to a norm of 1. While using max-norm regularization improved performance, other values for c that I tested did not seem to appreciably affect the results.

3.4.5 Multiplicative Gaussian noise

Dropout and DropConnect are essentially ways of adding noise during training to a neural network's representation of the input. Adding noise forces the neural network to learn a set of representations and transformations that are robust to the addition of the added noise. Dropout and DropConnect both apply binary multiplicative noise to the weights between hidden units, i.e. a masking matrix, and differ only in the structure of the masking matrix. Dropout masks all connections coming from a random subset of hidden units, while DropConnect masks the weights between hidden layers randomly. But note that Dropout can also be viewed as adding binary multiplicative noise to hidden unit activations; zeroing out the activation of a hidden unit has the same effect as zeroing out all of its output weights.

Viewing Dropout as creating noisy hidden unit activations begs the question of whether other ways of creating noisy activations may be even more beneficial than Dropout in improving generalization performance. A recent paper (Srivastava et al. 2014) proposed using Gaussian multiplicative noise rather than binary multiplicative noise. The noisy activations would then be generated according to equations 3.20-3.22. The hidden unit activation is simply multiplied by a Gaussian with mean and standard deviation equal to 1. Consequently the noiseless activation determines the mean, μ, and standard deviation, σ, of a Gaussian, $\mathcal{N}(\mu, \sigma^2)$, from which samples are drawn to determine the noisy activation.

$$r(t) = r(t) + r(t)\mathcal{N}(0,1) \quad [3.20]$$

$$r(t) = r(t)\mathcal{N}(1,1) \quad [3.21]$$

$$r(t) = \mathcal{N}(r(t), r(t)^2) \quad [3.22]$$

The paper also proposes that the standard deviation of the Gaussian can be adjusted by the parameter p as in equations 3.23-3.24. This is analogous to the parameter p in Dropout which controls the probably of a unit being retained. The paper demonstrates that for the same settings of p multiplicative Gaussian noise performs slightly better than Dropout.

$$\sigma = \sqrt{\frac{1-p}{p}} \quad [3.23]$$

$$r(t) = r(t)\mathcal{N}\left(1, \frac{1-p}{p}\right) \quad [3.24]$$

With multiplicative Gaussian noise, the network is thus forced to learn a code that is robust when the output of each hidden unit can be much larger or smaller than its deterministic activation. In fact, there is evidence that multiplicative noise within a network favors an independent component-like internal representation when the input to the network has non-Gaussian statistics, e.g. natural images (Gottschalk, Sexton, and Roschke 2004).

3.4.6 Poisson noise

As an alternative to Gaussian noise, one could take inspiration from neuroscience and let the noiseless hidden unit activation determine the rate of an inhomogeneous Poisson process as in equation 3.25. The output of the Poisson process, i.e. the noisy activation, is passed to the rest of the network. Each hidden unit thus operates according to the standard Linear-Nonlinear-Poisson (LNP) model of a neuron (Chichilnisky 2001) (See figure 3.5).

$$r(t) = Pois(r(t)) \quad [3.25]$$

A practical problem with using Poisson hidden unit activations occurs when the signal being fit has a small average value, as commonly occurs with binned neural spike trains. In this situation the hidden unit activations are typically driven towards small values, i.e. <<1. When these small values drive a Poisson process to generate the noisy hidden unit activation, it becomes very likely that a majority of the hidden units have an activation of 0. Once this occurs, the fitting of the network often becomes irrecoverably stuck because gradient information can no longer propagate through the network. A simple solution is to use a Gaussian approximation of Poisson noise: let the noiseless activation determine the mean, μ, and variance, σ^2, of a Gaussian, $\mathcal{N}(\mu, \sigma^2)$, from which samples are drawn to determine the noisy activation (equations 3.26-3.27). Note that the variance of the noise is equal to the activation rather than the square of activation as in section 3.4.5. From here on out "Poisson hidden unit activations" will refer to this Gaussian approximation.

$$r(t) = r(t) + \sqrt{r(t)}\,\mathcal{N}(0,1) \qquad [3.26]$$

$$r(t) = \mathcal{N}(r(t), r(t)) \qquad [3.27]$$

Of course the coupling of the mean and variance in Poisson hidden unit activations can also be modified by adding a hyperparameter to force the variance to be either smaller or larger than the mean (see equation 3.28). However, I have not yet

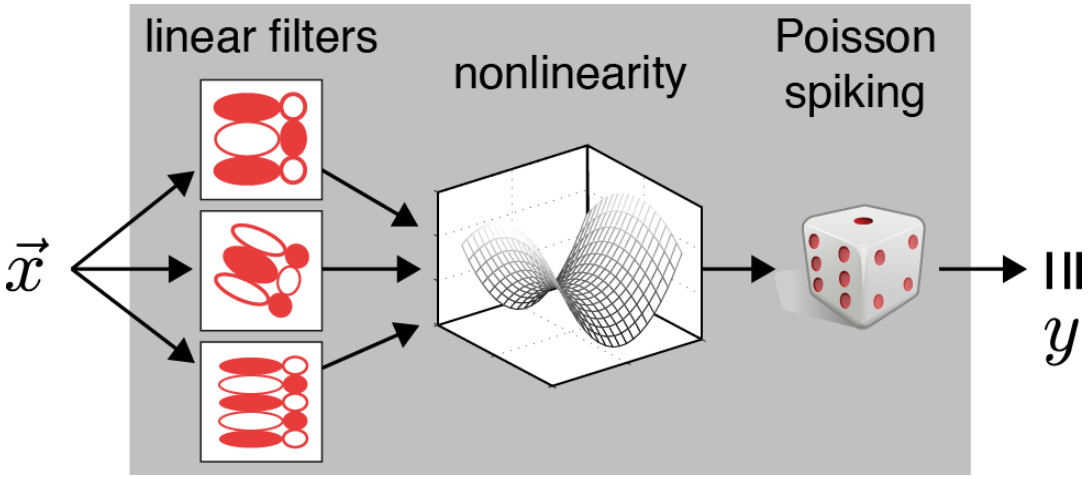


Figure 3.5: Standard schematic of linear-nonlinear-Poisson (LNP) model

An input vector is projected onto a set of linear filters, whose output is summed and passed through a nonlinear function. The output of the nonlinear function drives a Poisson spiking process. Adapted from (Park and Pillow 2011)

explored this possibility since any improvements are likely to be small and I did not want to introduce another hyperparameter.

$$r(t) = r(t) + p\sqrt{r(t)}\,\mathcal{N}(0,1) \qquad [3.28]$$

To compare the regularization effects of Dropout with Poisson hidden units I fit deep time-delay neural networks to the cat complex cell data described in the previous chapter. For each neuron, I fit ten networks with different initial condition for every regularization condition. As seen in figure 3.6B, the representation learned by a network with Poisson hidden units tends to be well distributed among the hidden units in order to average out the noisy activations. This distributed representation largely prevents unused or "dead" hidden units better than Dropout alone. As shown in figure 3.6A, the network trained using Dropout contains many more filters basically unchanged from their random initializations, which indicates a dead unit. The Poisson hidden units seem to adaptively balance between having enough variety to faithfully represent the function and enough similarity to average out noisy activations.

3.4.7 Poisson noise + Dropout

Both Dropout and Poisson hidden unit activations help prevent overfitting, and give similar increases in prediction performance as demonstrated in figure 3.7. However, the markedly different filters seen in 3.6A and 3.6B indicate that these techniques prevent overfitting in different ways. Interestingly, I found that using Dropout in the first hidden layer combined with Poisson activations for all hidden units yielded a significant increase in prediction performance for many cells (figure 3.7, red dots). Also, this combination of regularization techniques resulted in cleaner filters learned in the first hidden layer (figure 3.6C). What is perhaps most interesting, however, is that training networks with both Dropout and Poisson noise, results in the very consistent solutions from the different initial conditions. This is indicated by the tight clustering of the red dots in figure 3.7 relative to the other conditions. Exactly why using two kinds of noise during training results in both better and more consistent solutions than either kind of noise alone is a ripe area for future research. However, given that I can be assured of a near optimal solution from an arbitrary initial condition, I have adopted this as my default regularization technique for fitting neural networks. This consistency is an extremely valuable property because fitting deep time-delay neural networks on the macaque V1/V2 data can be very time consuming. The macaque V1/V2 results reported on in section 3.6 and 3.7 are from single networks fit using Dropout and Poisson hidden units, i.e. they were not selected among multiple fits with different initial conditions.

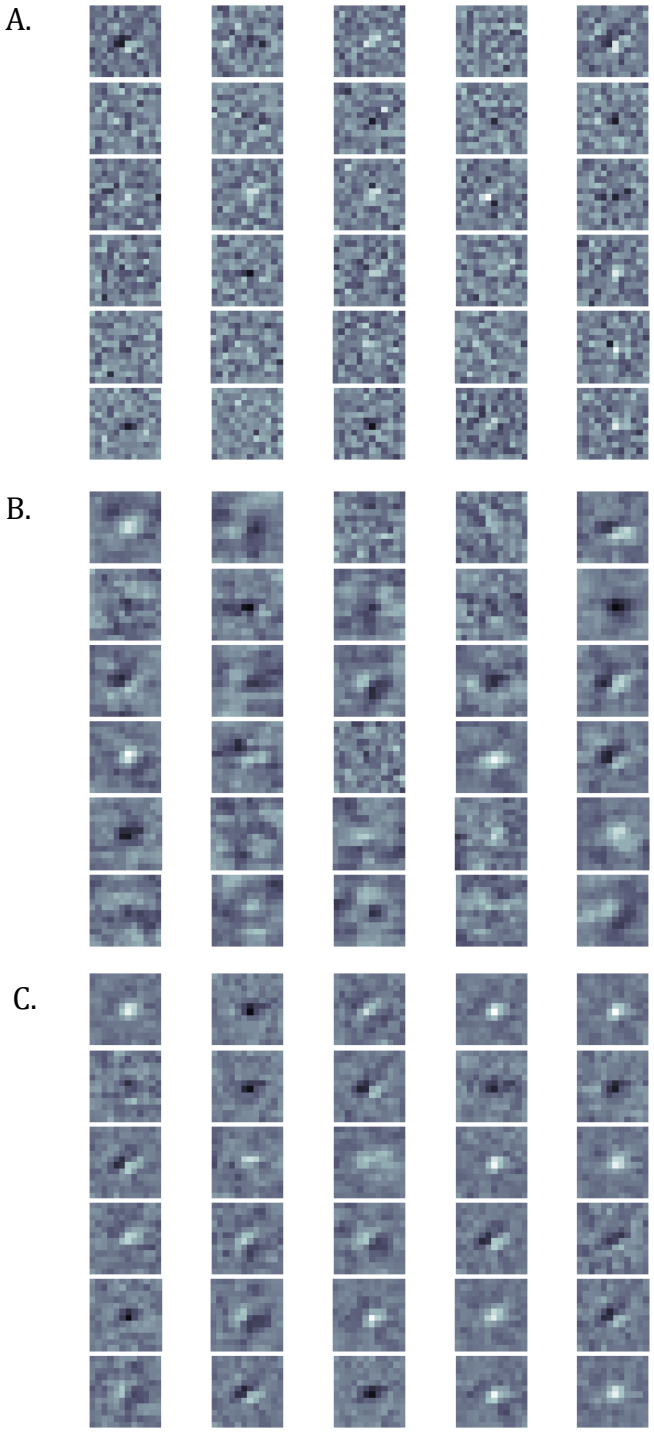


Figure 3.6: Effects of regularization on learned filter weights

Figures A-C show the first layer filters in a deep time-delay neural network learned on data from a cat complex cell. A is regularized with Dropout, B is regularized with Poisson hidden unit activation and C is regularized with both Dropout and Poisson hidden unit activation. C has the cleanest and most consistent filters and the model predicts significantly better than the others.

3.4.8 Output gain

A simple modification that I have found to be useful with Poisson hidden units is the addition of a fractional gain term on the output of the network. If the output of the network is multiplied by a small gain value, e.g. 0.1, the hidden unit activations will be forced toward larger values to compensate. Due to the mean-variance coupling of the hidden unit activations, such a gain term will increase the variance of the noise for each hidden unit and thus change the regularization effects of the noise. In the case where a small target signal causes hidden unit activations to be uniformly small, the gain term allows the regularizing effects of noise to operate more effectively than simply scaling the variance of the noise. This technique is especially applicable in regression where the target signal has a set, and possibly small, scale. Thankfully, this parameter does not need much tuning in practice since I have found a wide range of values 0.5-0.01 to provide virtually identical results. For all the results shown, the gain term has been set to 0.1.

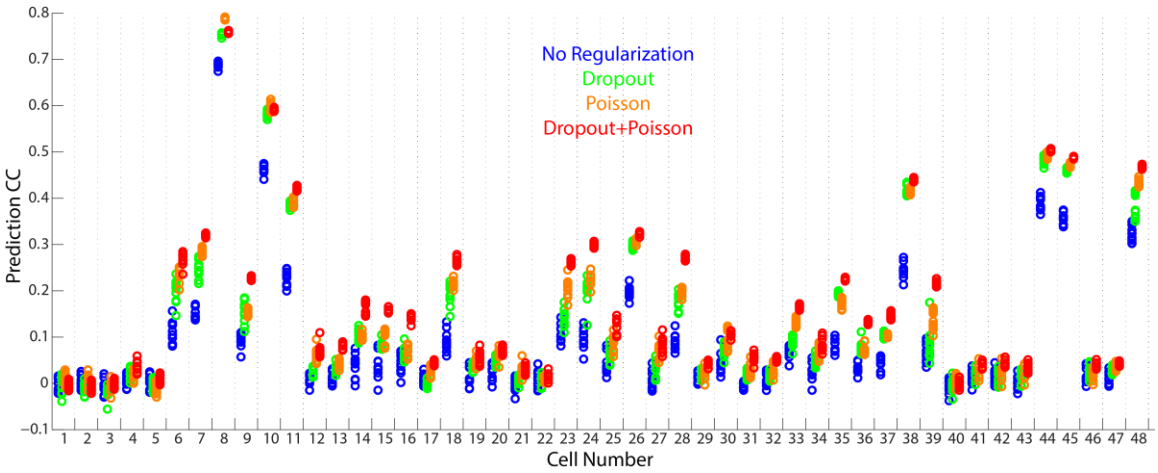


Figure 3.7: Comparing different forms of regularization

For each type of regularization ten deep time-delay neural networks, each with different initial conditions, were fit to 48 neurons from the cat complex cell data. Each dot corresponds to the prediction performance of a single network, thus there are 40 dots for each cell. Dropout and Poisson activations improve prediction performance relative to no regularization, but a combination of the two leads to the highest performance. Combining Dropout and Poisson activations also leads to the smallest variance in prediction performance across initial conditions.

3.5 Neural Network Inference

3.5.1 Dropout

When networks are trained with Dropout, inference/prediction is not as simple as a forward pass using the final set of learned weights. Because a random fraction of the hidden units are always missing, the units following the Dropout layer will receive more input when Dropout is turned off. Consequently, the weights learned between a Dropout layer and the subsequent layer are larger than they should be for inference. In the original Dropout paper (Hinton et al. 2012), the authors proposed a simple weight scaling procedure in which the trained weights are simply multiplied by p at inference time, where p is the probability of keeping a hidden unit during a minibatch. Thus if half the hidden units are removed during DropOut, i.e. $p = 0.5$, the learned weights will be too large by a factor of 2 and should be scaled by 0.5.

Alternatively, the output of the non-dropped units in the Dropout layer can be scaled up during training by a factor of $\frac{1}{p}$. This keeps the average output of the DropOut layer the same as if DropOut was not used. Consequently the weights learned during training will be of the correct scale for inference when Dropout is turned off and the output of the Dropout layer is no longer scaled up. This method is equivalent to the weight scaling procedure, but generally makes for cleaner implementations.

3.5.2 DropConnect

As with DropOut, when networks are trained with DropConnect the final weights are not adapted to the situation where all the weights are active. In their paper (Wan et al. 2013), the authors point out that weight scaling procedure used by DropOut (Hinton et al. 2012; Srivastava et al. 2014) is not theoretically justified and that the correct formulation for the output of the DropConnect layer is $r = 1/|M| \sum_M a((M \star W)v)$, where v are the outputs of the previous layer, W is the weight matrix, M is a binary masking matrix, a is the *relu* activation function, and $\star$ indicates element-wise multiplication. This equation averages over all the masking matrices that were used during training. The authors point out that the weight scaling procedure is making the approximation $\sum_M a((M \star W)v) \approx a(\sum_M (M \star W)v)$, i.e. they are averaging across masks before the activation function rather than after. However, because the correct equation is infeasible to evaluate, the paper proposes the following Gaussian approximation. Consider a single unit u_i before the activation function, $u_i = \sum_j (W_{ij} v_j) M_{ij}$ which is a weighted sum of Bernoulli variables M_{ij}. This weighted sum can be approximated by a Gaussian

where the mean and variance of the units are: $E_M[u] = pWv$ and $V_M[u] = p(1-p)(W \star W)(v \star v)$. The authors propose to draw samples from this Gaussian and pass them through the activation function before averaging and passing the results to the next layer.

However, in the case of the *relu* activation function such sampling is unnecessary because a closed form solution is possible. If $G(u)$ is the Gaussian with mean $E_M[u]$ and variance $V_M[u]$, averaging the sampled Gaussian after applying the *relu* activation function is equivalent to integrating $uG(u)$ from 0 to ∞. This integration has a closed form solution in terms of the Gauss error function:

$$r = \frac{V_M}{\sqrt{2\pi V_M}} e^{\frac{-E_M{}^2}{2V_M}} + \frac{1}{2} E_M \left(1 + \text{erf}\left(\frac{E_M}{\sqrt{2V_M}}\right)\right) \qquad [3.29]$$

3.5.3 Poisson Hidden Units

To make predictions using a neural network with Poisson hidden unit activations, one simply removes the Poisson sampling stage and allows the hidden unit activations to be passed to the rest of the network. Each hidden unit is thus activated at its expected value under the Poisson distribution.

3.6 Effect of Depth on Model Performance

As neural networks increase in depth their expressive power is greatly increased relative to a shallow network with the same number of hidden units (Bengio and Delalleau 2011). Functions involving complex dependencies can be represented far more efficiently with deep architectures than with shallow architectures. Thus the depth of the network required to accurately model an unknown function gives some insight into the complexity of the function.

I fit neural networks of varying depths (2 - 4 hidden layers) to stimulus-response data from visual areas V1 and V2 in order to determine the best network architecture for modeling each area. Because neurons in V2 have more complex selectivity than is found in V1, their models might require deeper architectures. But, the added flexibility of deeper networks naturally makes it easier to over-fit noise in the training data. As shown in figure 3.8, however, I found that deeper networks rarely performed substantially worse than shallower networks, and often performed somewhat better. But improvements with deeper networks were evenly distributed between V1 and V2 cells and had to do more with the signal-to-noise ratio (SNR) than functional complexity. As shown in figure 3.8, three layer networks are consistently better than two layer networks only for neurons where the

correlation coefficient of the prediction is greater than about r=0.5. This indicates that above a certain level of SNR the increase in model flexibility is beneficial. Moving to four hidden layers, however, made very little difference. Three hidden layers are thus adequate to model V1 and V2 cells.

3.7 Visualizing feature selectivity of V1 and V2

One of the appealing advantages of neural networks is that since the features learned in the first layer are weights on pixels, they can be directly visualized. However, interpreting these features can be tricky because the network's response to the activation of any given feature can be highly nonlinear. The output of each feature is rectified and then passed to the rest of the network where is can form nonlinear relationships with other features. Nonetheless, all network computations are in terms of this feature space and so visualizing the features gives some insight into the functional properties of the network.

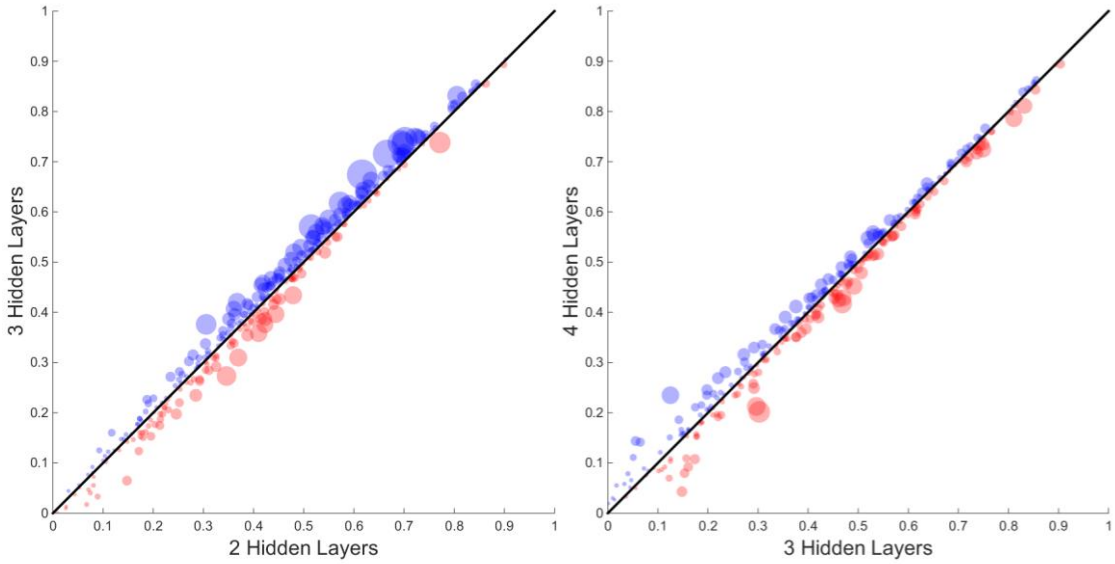


Figure 3.8: Comparing performance of neural networks with different depths

Each dot represents the prediction performance of a model in terms of correlation coefficient. Blue indicates an improvement and red a reduction in performance. The size of each dot reflects the change in variance explained. This is done to emphasize the fact that changes at the upper end of correlation reflect larger changes in variance explained than changes at the lower end. Three layer networks are consistently better that two layer networks past a threshold of about r=0.5. Four layer networks perform about the same as three layer networks with no clear trend in the differences.

It is important to note that the filters are learned directly on RGB pixel values and the only preprocessing performed was Z-scoring each color channel across all pixels (not each individual pixel). No whitening, histogram normalization, or other form of contrast enhancement was used. This forces the neural network to learn all required transformations of the pixel values needed to model the neural response.

The filters in figures 3.9 and 3.10 are ordered, and their alpha values adjusted, based on their individual contributions to the output of the model. This was assessed by simulations in which each filter was activated one at a time while all others were held constant at 0. These outputs were compared to the model's baseline output when all filters have 0 activation to determine the degree of excitation or inhibition contributed independently by each filter.

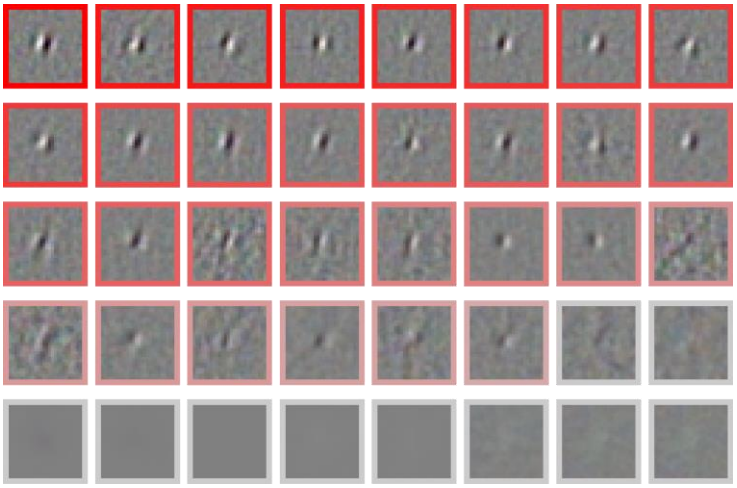
3.7.1 V1 features

The features learned in the first layer of neural networks trained on V1 data agree with previous work on V1 selectivity. As shown in figure 3.9, most of the filters resemble Gabor filters of a particular spatial frequency and orientation or are unorientated blobs (Ringach 2002). While most of the filters were selective for luminance patterns, i.e. they were black and white, some of the filters exhibited color selectivity as well (figure 3.9C). This also agrees with previous reports of V1 neurons exhibiting both color and orientation tuning (Conway 2001; Conway and Livingstone 2006). This is the first time, however, that such tuning has been observed with natural stimuli. Given the fact these filters are learned from RGB values in natural images and the fact that the filters learned on most cells were black and white, the combined tuning to both color and orientation tuning is likely to be non-artifactual and carry functional significance.

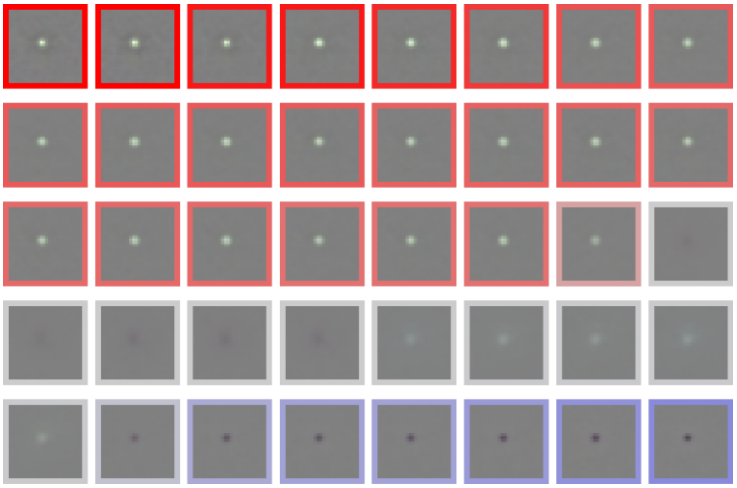
3.7.2 V2 features

The features learned in the first layer of neural networks trained on V2 data are substantially different than the V1 filters. As shown in figure 3.10, the filters of a single neuron are more diverse and have components selective for more than one orientation. The presence of tuning for more than one orientation agrees with past studies of V2 selectivity (Anzai, Peng, and Van Essen 2007; Ito and Komatsu 2004). However, this is the first visualization of V2 feature selectivity in pixel-space; previous visualizations have always been in terms of the parameterized stimuli used. It is clear, just from the filter visualization, that V2 neurons have more complicated selectivity than V1 neurons and are not as easily parameterized. However, the V2 filters do resemble superpositions of V1/Gabor-like filters. For example, many of the filters in figure 3.10A appear to consist of spatially offset

A.



B.



C.

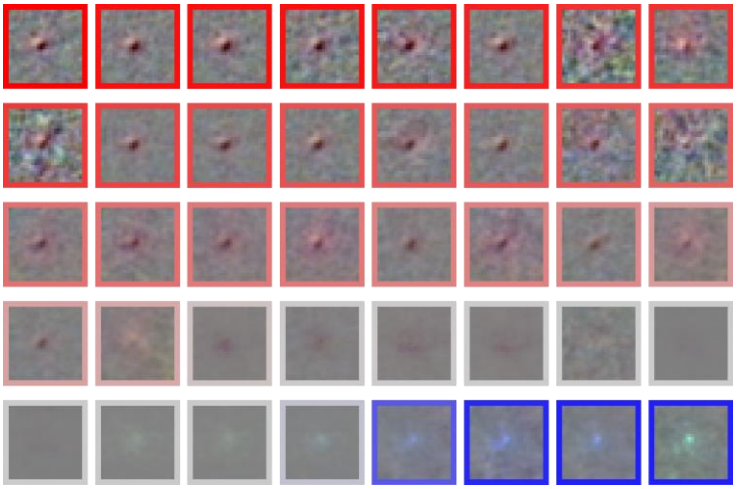
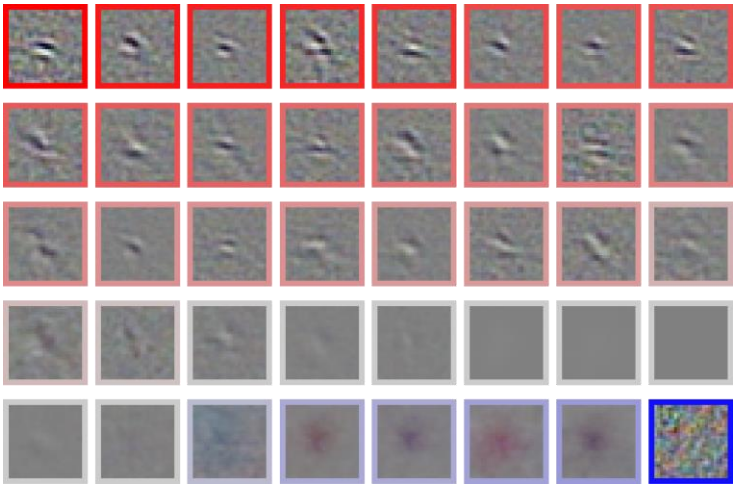


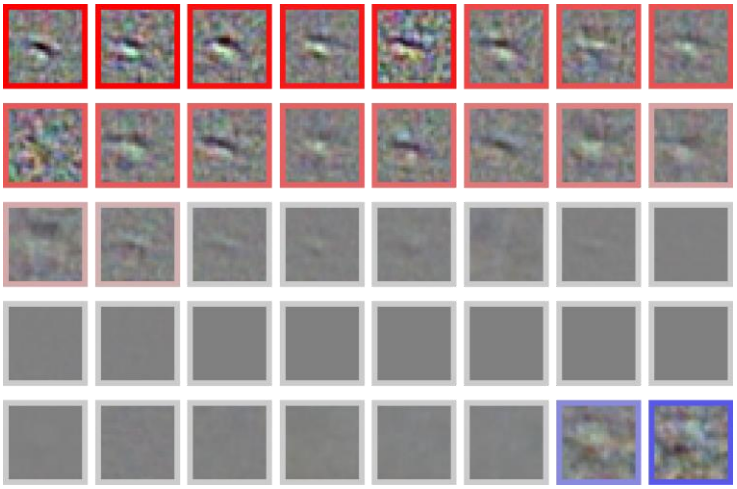
Figure 3.9: Visualization of V1 models

Each plot shows the forty filters learned in the first layer of a neural network model of a V1 neuron. The filters are ordered, with colored borders, from most excitatory (red) to most inhibitory (blue). The alpha value of each filter is also adjusted to reflect its individual contribution. The filters in A are of various phases and each can individually excite the model, indicating that the corresponding neuron is a complex cell. The filters in B are virtually identical blobs, indicating the neuron has no orientation selectivity and is only tuned to luminance spots. The filters in C are all of the same phase, indicating a simple cell. In addition, the excitatory filters in C all have a red tint and a the inhibitory filters have a blue tint, indicating the neuron exhibits color opponency in addition to orientation tuning.

A.



B.



C.

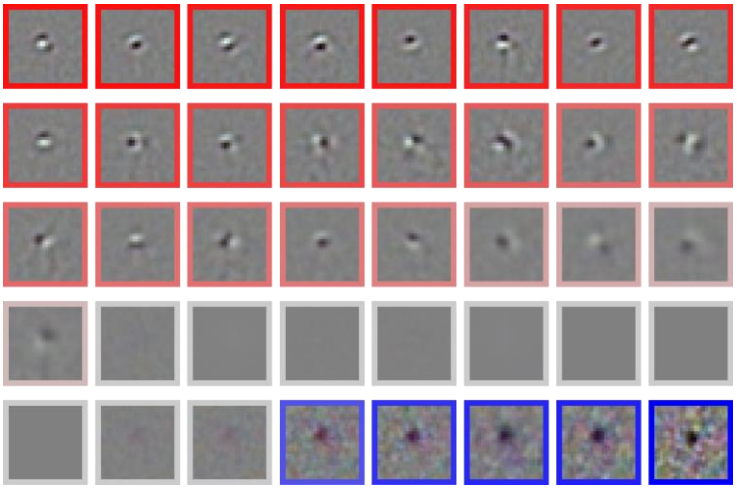


Figure 3.10: Visualization of V2 models

Each plot shows the forty filters learned in the first layer of a neural network model of a V2 neuron. The filters are ordered, with colored borders, from most excitatory (red) to most inhibitory (blue). The alpha value of each filter is also adjusted to reflect its individual contribution. The filters of the V2 neuron models show much more diversity relative to the V1 neuron models. The filters in A appear to combine various phases of spatially offset filters with horizontal and oblique orientation selectivity. The filters in B also appear to combine horizontal and oblique selectivity, but the component filters are phase aligned in order to create curves. The filters in C mostly resemble V1/Gabor-like filters of various orientations, while a few appear to combine orientations.

Gabor-like filters with horizontal and oblique orientation selectivity at various relative phases. Such selectivity could result from an AND-like operation on the output of V1 complex cells. The filters in 3.10B, however, also appear to combine horizontal and oblique selectivity, but the phases are aligned within a filter such that each is a continuous curve. Many of the filters in 3.10C resemble V1/Gabor-like filters of various orientations, while a few appear to combine orientations. While the neuron corresponding to figure 3.10B seems like it could be interpreted in terms of phase-invariant contour selectivity, the interpretation of the other two neurons is less straightforward. It seems likely that V2 may best be understood, analogously to V1 (Olshausen and Field 1997), as computing a higher-order basis for representing natural stimuli.

3.8 Comparison to linearized models

While deep time-delay neural networks are powerful and flexible models, this power comes at the cost of needing more and/or cleaner data to estimate predictive models. Because neuroscience data is typically limited and noisy, linearized regression has been the dominant, and quite successful, paradigm for estimating receptive field models (David, Vinje, and Gallant 2004; David and Gallant 2005; David, Hayden, and Gallant 2006; Nishimoto and Gallant 2011; Willmore, Prenger, and Gallant 2010). A previous attempt to model macaque V1 neurons using neural networks performed somewhat worse than a simple linearized (e.g. fourier power linearizing transform) model (Prenger et al. 2004). Due to advancements in recording technology, the macaque V1/V2 data I collected contains significantly more samples than past work, but it is also much richer since the stimuli are color natural movies, rather than a sequence of grayscale natural images (Prenger et al. 2004). Thus, it is unclear how linearized regression will compare to neural networks on this data.

The linearizing transform I will compare to is the current state-of-the-art of our lab: a bank of model simple and complex cells of various positions, orientations, and spatial and temporal frequencies constructed using 3D Gabor wavelets (Nishimoto and Gallant 2011). The parameters determining the size of the wavelets were chosen based on the filters observed in the neural network models. Because, a large majority of the V1/V2 neural network models contained filters that were luminance selective, I transformed the color stimuli to grayscale before applying the linearizing transform. Furthermore, it is unclear what an appropriate linearizing color wavelet transform would be. The RGB to grayscale transform was based on the luminance selective filters found by the neural network models. This actually gives an advantage to the linearized models since fewer weights need to be learned relative

to a full color model and the grayscale transform appropriate for most cells is empirically determined. Even still, the number of wavelets was about five times larger than the number of grayscale pixels.

The linearized model was fit with threshold gradient descent with grouping over delays (see section 1.2.4.4). To get the best estimate of the early stopping point, it was determined by the first half of the validation data. Similarly, the optimal depth of neural network model for each neuron was determined by prediction performance on the first half of the validation data. The relative performance of the models is compared using the second half of the validation data for each neuron. The relative performance of linearized regression models vs. deep neural network

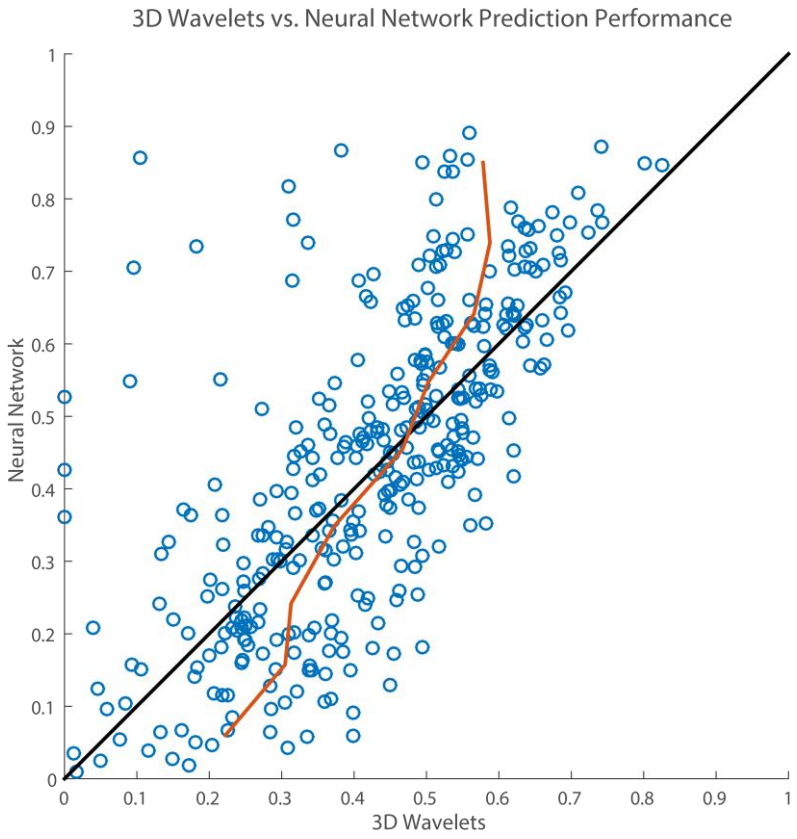


Figure 3.11: Comparison of prediction performance of linearized regression using 3D Gabor wavelets and deep neural networks on macaque V1/V2 data

Each dot represents the prediction performance of both models for a single neuron in terms of correlation coefficient. Dots above the black line indicate the neural network model performed better for that neuron, and dots below the black line indicate the linearized regression model performed better. The orange line plots the average performance of a linearized model at a given performance level of neural network models. The neural network model clearly performs better above a certain performance threshold.

models is plotted in figure 3.11. Linearized regression models generally perform better on neurons where the neural network models give poor prediction performance, i.e. the correlation coefficient is between 0 and 0.4. The two techniques have roughly equal performance in the middle range of prediction performance, i.e. a correlation coefficient between 0.4 and 0.6, with approximately equal numbers of cases where one model is superior to the other. However, above a correlation coefficient of about 0.6, the neural network model is clearly superior. Similarly to the comparison of neural network depths in section 3.6, this indicates that more flexible models perform better when the data contains enough signal to properly constrain them.

3.9 Future directions

3.9.1 Smooth-MAX / Smooth-MIN and soft logical operators

Hidden units in neural networks typically pool their input with a weighted sum, which is then passed to a nonlinear function. A weighted sum, however, is only one possible way of pooling information. For, example MAX pooling has become a standard pooling function for the feature maps (i.e. the output of convolutional filtering) in convolutional neural nets (Riesenhuber and Poggio 1999; Ranzato et al. 2007). More recently Lp-norm pooling has emerged as a more flexible alternative to MAX pooling that approaches MAX as p goes to infinity (Gulcehre et al. 2013; Sermanet, Chintala, and LeCun 2012).

$$a(x) = \left(\sum_n |x_n|^p \right)^{1/p} \qquad [3.30]$$

Lp-norm pooling can smoothly vary between a sum like operation, $p = 1$, and a MAX like operation, $p = \infty$, but there is a major discontinuity at $p = 0$. As p approaches 0 the formula approaches infinity. So while the Lp-norm function can represent the MIN function at $p = -\infty$, the transition to negative values of p is basically impossible for gradient based methods. (A related deficiency of Lp-norm pooling is that the scale of the function output can vary substantially for different settings of p. This creates ambiguity between effects due to scale and effects due to the shape of the Lp-norm nonlinearity and consequently makes optimization difficult.)

The MAX and MIN functions have interesting connections to the concepts of invariance and selectivity widely used in visual neuroscience. Spatial invariance means that a neuron responds to a certain visual feature regardless of its specific location within the neurons' receptive field. A simple way to compute this property

would be to tile the receptive field with filters selective for the visual feature and then take the MAX of the filter outputs. Selectivity, on the other hand, means that a neuron only responds to a specific visual feature; neurons with "high" selectivity only respond to very specific complex visual features. V2 neurons, for example, are more selective than V1 neurons. A simple way to compute selectivity to complex features would be to take the MIN of a few filters that have different individual selectivities. The MIN would output high values only when all the component filters have high activations and is thus selective for the combination. Giving neural networks the ability to adaptively choose MIN or MAX pooling would thus endow them with substantial additional power and flexibility. The following function can compute the min, mean or max of a vector for different settings of α.

$$a(x) = \frac{\sum x e^{\alpha \cdot x}}{\sum e^{\alpha \cdot x}} \qquad [3.31]$$

When $\alpha = -\infty$, the function computes the MIN, when $\alpha = 0$ the function computes the mean, and when $\alpha = \infty$ the function computes the MAX. Incorporating weighting of the vector components, however, is not as simple as just weighting all instances of the vector in the equation. If I wish the weights to reflect how much each of the vector components contributes to the MIN or MAX-like operation, the following equation will not work as desired.

$$a(x, w) = \frac{\sum w \cdot x e^{\alpha \cdot w \cdot x}}{\sum e^{\alpha \cdot w \cdot x}} \qquad [3.32]$$

To see why, consider the case of computing a MIN-like operation where α is negative, all components of x are positive, and all weights are positive but one, which is 0. In this case the function will tend toward 0 because it is computing the MIN of the weighted values. Rather than remove a component from consideration, a 0 weight will have a large impact on the function output. However, by modifying this function I obtain a function that works as desired.

$$a(x, w) = \frac{\sum w \cdot x e^{\alpha \cdot x}}{\sum w \cdot e^{\alpha \cdot x}} \qquad [3.33]$$

A hidden unit using this new function in place of a weighted sum can become selective for, or invariant to, selected components of its input. A neural network employing this activation function in its hidden layers could efficiently learn functions that require both nonlinear selectivity and invariance, and thus may be ideally suited to modelling the behavior of neurons in visual cortex.

3.10 Conclusions

I have demonstrated that deep time-delay neural networks provide a powerful framework for receptive field modeling and provided methods for surmounting the inherent difficulties in fitting such flexible models. Despite making no strong assumptions about neural function, it is remarkable that the prediction performance of these neural network models can equal or surpass linearized regression models that use a feature space based on the known properties of V1 neurons. However, as was expected, linearized regression performed better in the lower signal-to-noise regime. Additionally I found that V1 and V2 neurons can be adequately modeled using neural networks with three hidden layers; the addition of more layers provided no consistent benefit. Then based on the first layer of the models of V2 neurons, I created the first pixel-space visualizations of the relevant features for a V2 neuron.

These results indicate that neural networks are a viable and promising tool for analyzing the large datasets created by modern electrophysiology techniques. However, these networks must be further interpreted in order to gain new insights into the neural computations being modeled. Interpreting neural networks is an active area of research and there is no universally accepted method. In the next chapter I turn to investigating the properties of macaque V1 and V2 neurons through *in silico* experiments on their neural network models.

Chapter 4

Simulating Neurophysiology Experiments in V1 and V2

Chapter 4

Simulating Neurophysiology Experiments in V1 and V2

4.1 Introduction

Deep neural network models of visual neurons are very different than the models typically used in visual neurophysiology. Most of the models used and experiments performed in neurophysiology are based (either implicitly or explicitly) on strong hypotheses about neural function. This contrasts greatly with approach taken here in which a highly flexible model is fit to very general and naturalistic data. Thus it is important to connect the neural network models with existing modeling frameworks and with the known properties of V1 and V2 neurons. This can be done by performing *in silico* experiments on the neural models that mimic experiments typically performed on neurons recorded from animals. The *in silico* experiments are performed by formatting the chosen stimulus set to the appropriate size and number of frames to mimic in the presentation during an *in vivo* recording session, and then passing the stimuli to the deep neural networks to obtain predicted responses. The model responses can then be analyzed and compared to previous results using a similar stimulus set.

The ability to replicate several past experiments on models of neurons also makes it possible to connect past results whose relationship was previously unclear. Most visual neurophysiology experiments set out to test a specific hypothesis about neural selectivity by means of a carefully constructed stimulus set. Because tuning is more complex in visual areas beyond V1, a wide array of stimulus sets have been tested. However, this diversity of stimuli makes it difficult to compare results across studies and draw firm conclusions about the coding properties of extra striate cortex. For example, it is impossible to know how the neurons featured in the example figures from such studies would have responded to different stimuli. So in addition to connecting neural network models to previous experiments, *in silico* experiments allow past experiments to be compared with each other.

Because the deep time-delay neural network model was fit to data from a wide variety of natural movies it is not biased by any strong hypotheses about the neural selectivity to be modeled; the strongest assumption built into the model is a finite temporal integration window. I can thus make comparisons of the selectivity measured with artificial stimuli to the selectivity exhibited with natural stimuli. I can then assess whether a particular set of artificial stimuli effectively probe selectivity and truly give insight into neuronal responses. In addition, I plan to make these neural network models available so that other researchers can more

effectively generate new stimuli that better probe the selectivity of visual cortex. Due to the ease with which *in silico* experiments can be performed, many more (and truly unlimited) experiments can be performed relative to in vivo experiments. I hope that this will lead to new ideas, experiments, and discoveries. I also hope that such experiments may highlight deficiencies in the neural network modeling framework and lead to more concise and interpretable models.

4.2 Foveal V1/V2 data and models

All the data used in this chapter comes from chronic recordings I made using a Utah microelectrode array (Nordhausen, Rousche, and Normann 1994) implanted on the V1/V2 border of an awake macaque. The animal fixated during the presentation of large field, color natural movies. All receptive fields were between 0.5° and 1.0° below the fixation point, i.e. they were in the fovea. Windows of 30x30 pixels were cut out of the stimuli around the receptive field of each neuron. Each of the RGB pixels within these windows constituted the input to a neural network model, thus the input dimensionality was 2700, i.e. 30x30x3. The neural network models were fit as described in chapter 3. To ensure high quality results for the *in silico* experiments I only used models that could account for greater than 20% of the variance in their validation data, i.e. the correlation coefficient of the prediction was greater than 0.447. While lower thresholds did not significantly change the results, this rather stringent criterion ensures that the results are good reflections of the true neural properties. There were 157 out of 313 V1 neurons and 34 out of 86 V2 neurons whose model performance exceeded this threshold.

As an initial sanity check of the models before performing the *in silico* experiments, I determined whether they conform to what is known about the large scale organization of visual cortex. Visual cortex is organized retinotopically; i.e. nearby locations in cortex correspond to nearby locations in space. However, few studies have been able create high resolution maps in the fovea (Dow, Vautin, and Bauer 1985), and none have sampled cortex as densely as the Utah array used in this recording. It is thus interesting to see whether the population of V1/V2 models exhibits strong retinotopy. To obtain a receptive field location and size from the neural network model of a neuron, I simply averaged the absolute values of the filters from the first layer and fit a 2D Gaussian to the results. I plotted the center and full width at half maximum of each Gaussian fit using dots and circles in figure 4.1. The plot makes it clear that the foveal receptive field locations found by the neural network models are highly retinotopic with very little scatter in receptive field centers. Also aparent in the plot is the reversal at the lower vertical meridian corresponding to the the V1/V2 border: progressing from the lower left of the array

to the upper right, receptive field centers move towards the vertical meridian and then away. At the very least the neural network models can clearly estimate receptive fields accurately enough to capture high resolution retinotopic organization near the fovea. Thus, I will now turn to *in silico* experiments based upon the dominant paradigms used to study V1 and V2. I will evaluate each of the experiments based on what they reveal about V1 and V2 selectivity and how well they can distinguish V1 and V2 cells. The following *in silico* results comprise the most detailed study to date of foveal V1 and V2 neurons in an awake animal.

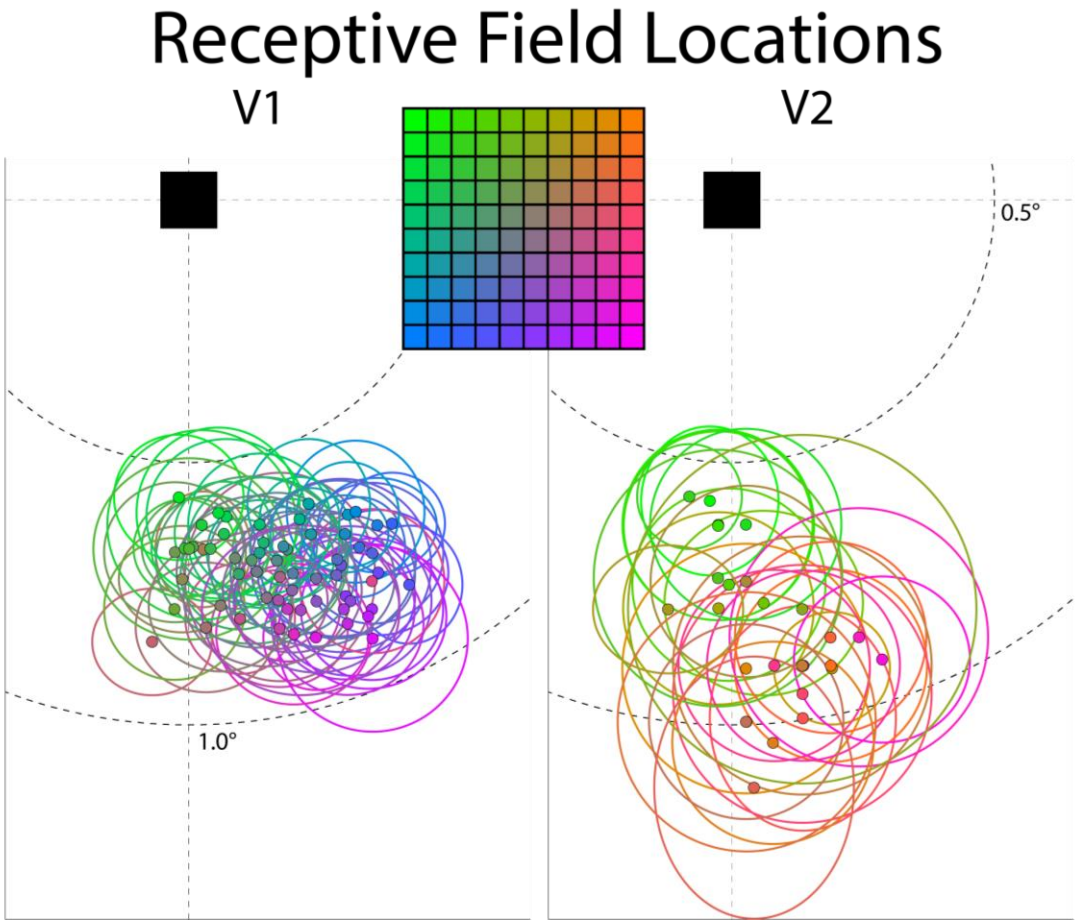


Figure 4.1: Receptive field locations in foveal V1/V2

The black square in each plot represents the fixation point and the concentric dotted lines indicate half degrees of eccentricity. The multicolored square represents the Utah array with color serving to index electrode location. Each receptive field is plotted with a dot and circle to indicate its center and size. The dots and circles are colored based on their corresponding electrode.

4.3 Drifting gratings

Sinusoidal gratings have long been used to quantify the response properties of neurons in visual cortex in terms of orientation tuning, spatial and temporal frequency tuning, direction selectivity (Foster et al. 1985) and contrast sensitivity (Sceniak et al. 1999). To verify that the neural network models behave similarly to real neurons I tested drifting gratings with 12 orientations, 7 spatial frequencies, and 14 temporal frequencies in each direction of motion to find the optimal grating for each model. As shown in figure 4.2 the population of models exhibited tuning to all orientations. Additionally there seems to be a slight bias towards horizontal and vertical orientations (Appelle 1972; Li, Peterson, and Freeman 2003) but this did not reach statistical significance. The distributions of spatial frequency selectivity in V1 and V2 shown in figure 4.3 are in line with past results near the fovea (Foster et al. 1985), but biased toward higher spatial frequencies in V2 than previous results. This is not surprising since the receptive fields of all neurons were closer to fixation than previous studies; all were from between 0.5 and 1 degree from fixation. Unfortunately, the Nyquist limit for the stimulus apparatus used was about 4.5 cycles/degree so higher spatial frequencies could not be present in the model or tested. The distributions of temporal frequency selectivity in V1 and V2 shown in figure 4.4 are similar to past results (Foster et al. 1985) that showed V2 neurons between the V1 and V2 populations was statistically significant, p=0.0036.

Simple and complex cells in V1 are usually distinguished by the F1/F0 ratio: the ratio of response modulation to the average response during stimulation with an optimal drifting grating (Skottun et al. 1991). This ratio quantifies the degree to

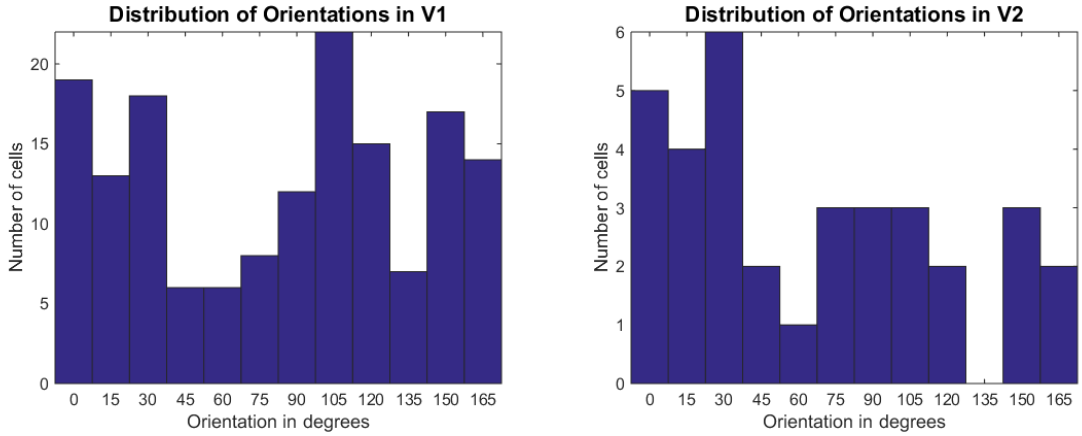


Figure 4.2: Preferred orientation of neural networks models of V1/V2

The models had well distributed preferred orientations. The slight apparent bias for the cardinal directions in both V1 and V2 did not reach statistical significance, p=0.075.

which a cell's response is invariant to the phase of the grating. The distribution of this ratio in V1 neurons is reported to be bimodal with a minima at 1; simple cells are defined as having F1/F0 ratios greater than 1, while complex cells have F1/F0 ratios less than 1 (Skottun et al. 1991). I computed the F1/F0 ratio for each of the V1 and V2 models using the harmonics of their responses to optimal drifting gratings. Figure 4.5 shows the distribution of the F1/F0 ratio for V1 and V2. Most of the neurons have a ratio less than 1, and there are too few with ratios above 1 to determine if the distribution is bimodal. This is actually similar to past work in

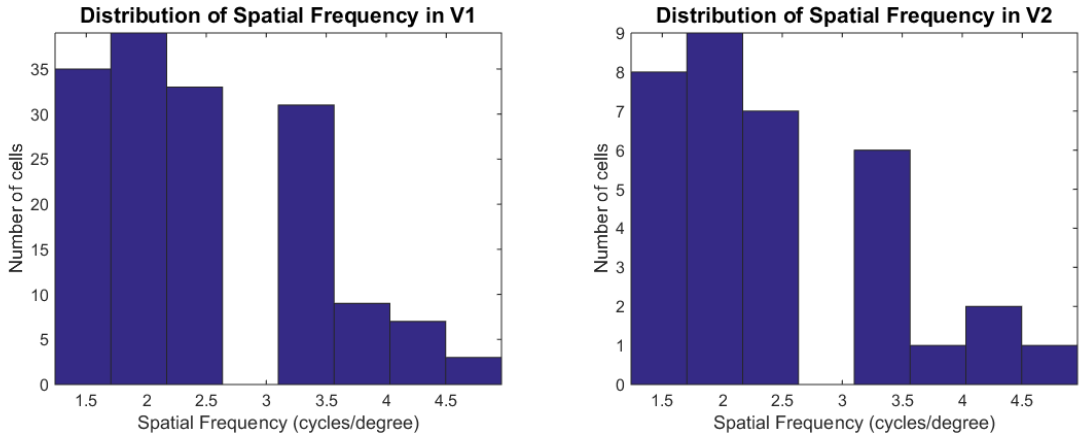


Figure 4.3: Preferred spatial frequency of neural networks models of V1/V2

The distribution of preferred spatial frequency was similar in V1 and V2, p=0.91. While roughly consistent with previous reports, V2 neurons are typically reported to prefer lower spatial frequency gratings. However, the sample size is relatively small and the recording is nearer the fovea than past reports.

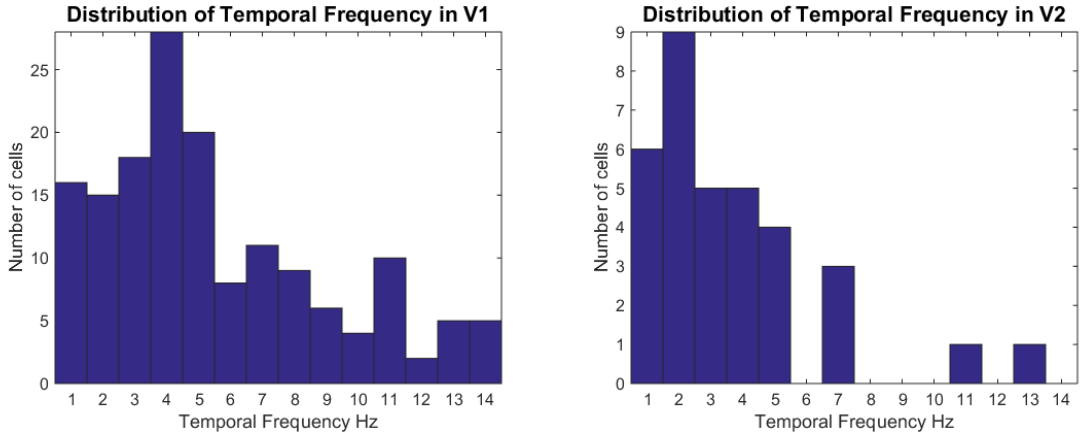


Figure 4.4: Preferred temporal frequency of neural networks models of V1/V2

The distributions of preferred temporal frequency of the models of V1 and V2 neurons are similar, but indicate that V1 neurons are more likely prefer higher temporal frequencies than V2 neurons. This difference is significant at p=0.0036.

awake animals that found relatively few simple cells and a non-bimodal distribution of the F1/F0 ratio (Kagan, Gur, and Snodderly 2002). This result is more evidence of the relative importance of complex cells in natural visual and further indication that some effects obtained under anesthesia may not generalize to awake animals.

The neural network models behaved similarly to real neurons stimulated with drifting gratings and the population statics align well with past reports. V1 and V2 neurons are notoriously difficult to distinguish based on drifting gratings (Mahon and De Valois 2001; Foster et al. 1985). While there was a statistically significant difference found between the temporal frequency distributions in V1 and V2, the distributions are largely overlapping. Consequently it would be impossible to distinguish a given cell based on this measure. Furthermore, standard drifting gratings cannot give insight into anything fundamentally different that V2 neurons compute relative to V1.

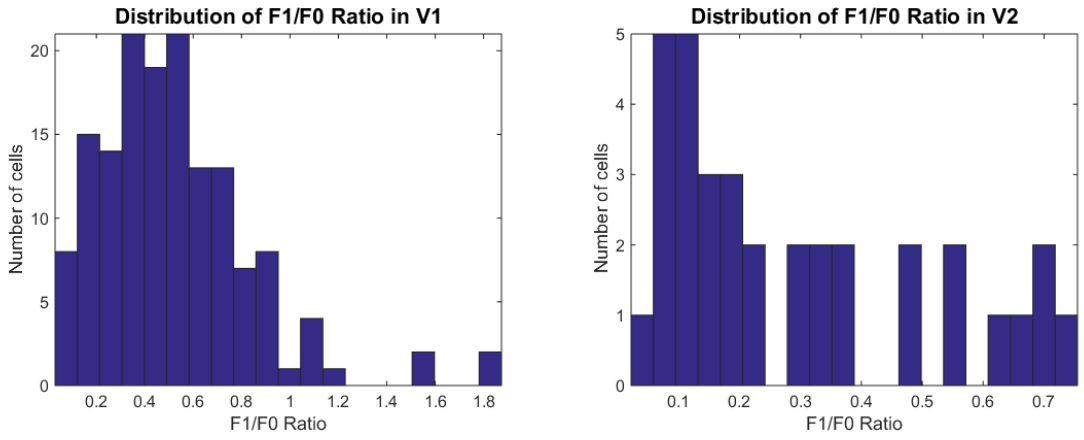


Figure 4.5: F1/F0 ratio of neural networks models of V1/V2

Only a few V1 neuron models exhibited a F1/F0 ratio greater than 1. The bimodal distribution of F1/F0 ratios seen in anesthetized V1 recordings was not present. In addition of the V2 neurons had ratios less than 1 indicating a preponderance of phase insensitive cells in awake V1 and V2. There was no significant difference between these distributions, p=0.52.

4.4 Spike Triggered Average / Spike Triggered Covariance

The spike triggered average (STA) and spike triggered covariance (STC) methods are among the most commonly used data analysis techniques in visual neurophysiology (see section 1.2.2). The STA corresponds to the average of all stimuli preceding a spike, while STC corresponds to the covariance of the stimuli preceding a spike. Together they form a Gaussian approximation of the distribution of stimuli that precede spikes (Schwartz et al. 2006). These methods are used to determine directions in stimulus space, e.g. pixel space, that are relevant for a neuron's responses. Typically only the first few and last few eigenvectors of the STC

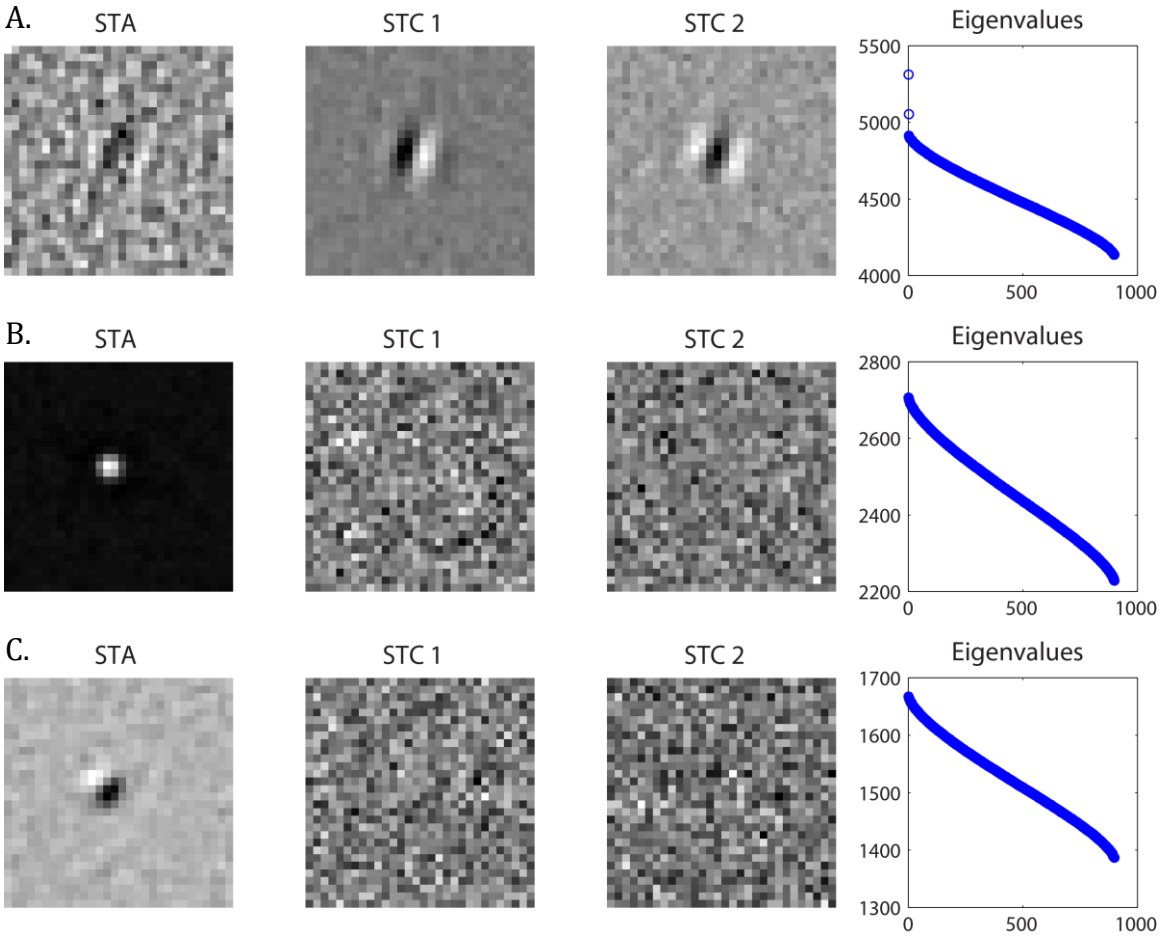


Figure 4.6: Spike Triggered Average (STA) and Spike Triggered Covariance (STC) of V1 models

The results of STA and STC on the outputs of neural network models of V1 neurons. 4.6A clearly shows a complex cell with two significant excitatory eigenvectors, while 4.6B and 4.6C show simple cells with no significant eigenvectors of the STC matrix. The stimulus directions found by STA and STC closely match the actual filters used by the model, which are shown in figure 3.9

matrix are considered (Schwartz et al. 2006). When the first few eigenvalues are significantly larger than the rest of the eigenvalues, they correspond to excitatory eigenvectors, i.e. stimulus directions that excite the neuron. And, correspondingly, when the last few eigenvalues are significantly smaller, they correspond to inhibitory eigenvectors, i.e. stimulus directions that inhibit the neuron. To confirm the neural network is capturing these basic statistics and compare the directions found by the STA and STC to the model filters, I computed the STA and STC on the outputs from each model stimulated with white noise. Because the output of the

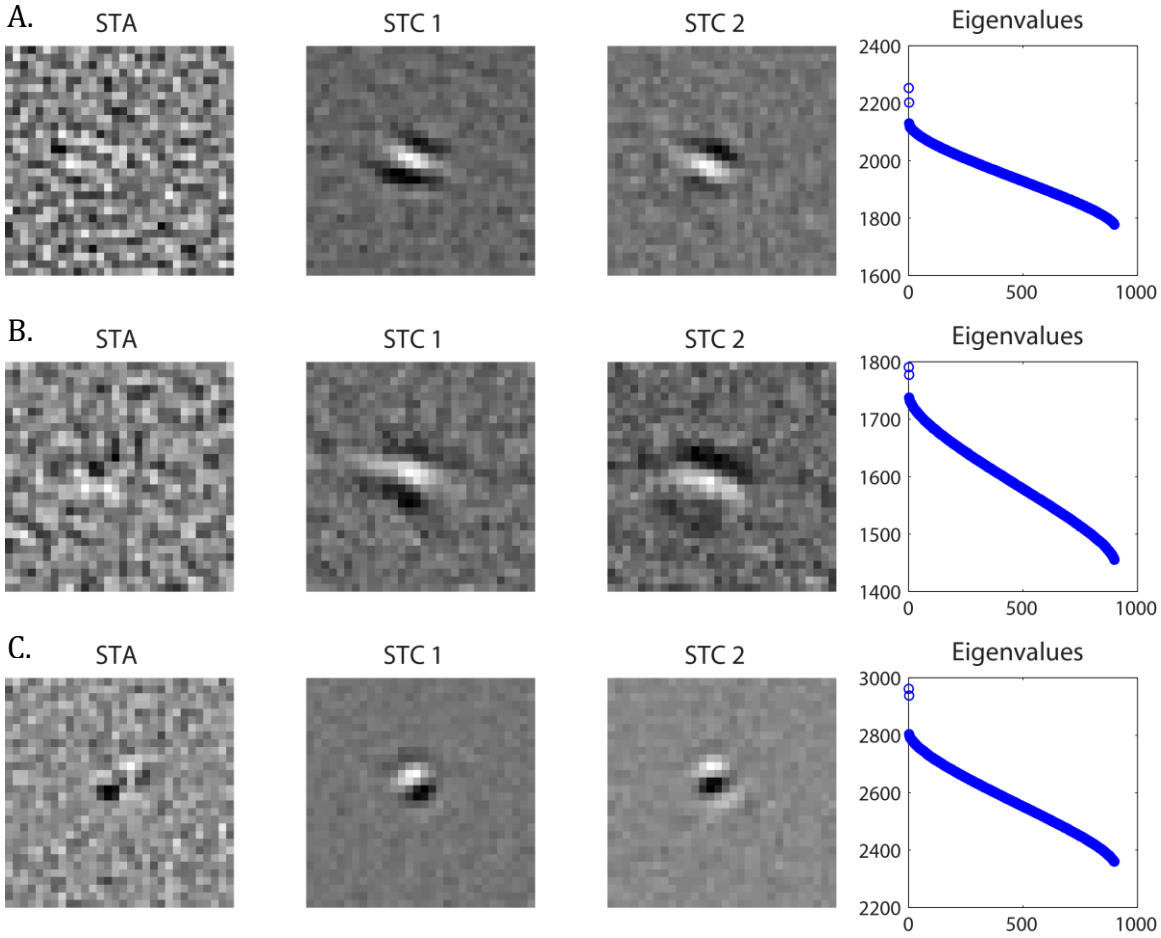


Figure 4.7: Spike Triggered Average (STA) and Spike Triggered Covariance (STC) of V2 models

The results of STA and STC on the outputs of neural network models of V2 neurons. There are two significant eigenvectors of the STC matrix in each case, while all the STAs show little structure. However, the stimulus directions found by STC resemble those from a canonical complex cell and give a limited picture of the models' true selectivity. The corresponding filters shown in 3.10, include these stimulus directions, as well as several others, indicating a much richer selectivity that would be inferred by STC.

model is not a series of spikes but an estimated rate, I used the continuous version of the STA and STC described in section 1.2.2.

To make comparisons with the neural network filters easier, the STA and STC were calculated only at the peak time lag before a response. Figure 4.6 shows the STA and two eigenvectors of the STC matrix for the same V1 models whose filters were

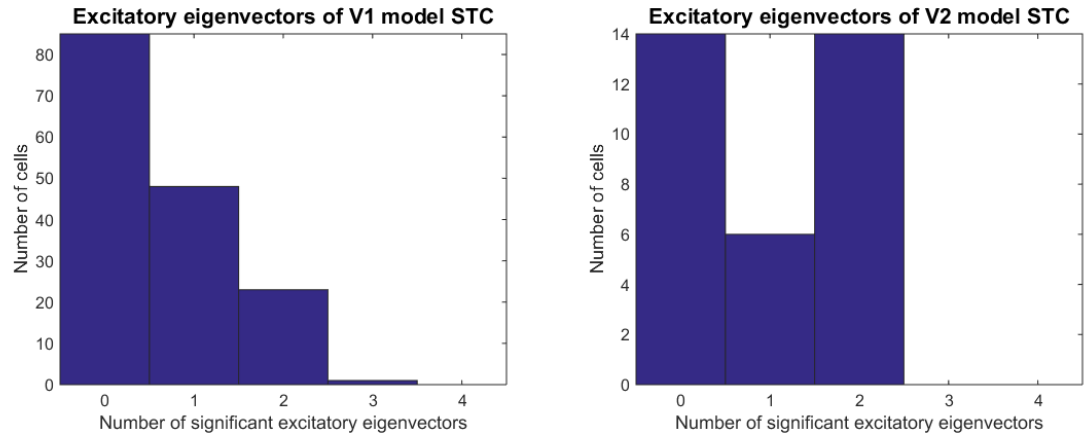


Figure 4.8: Distribution of significant excitatory eigenvectors obtained by STC on V1/V2 models

These histograms show the number of significant excitatory eigenvectors obtained by STC on the population of V1 and V2 models. These distributions are significantly different, p=0.01, with a greater proportion of the V2 models creating two significant excitatory eigenvectors in STC analysis.

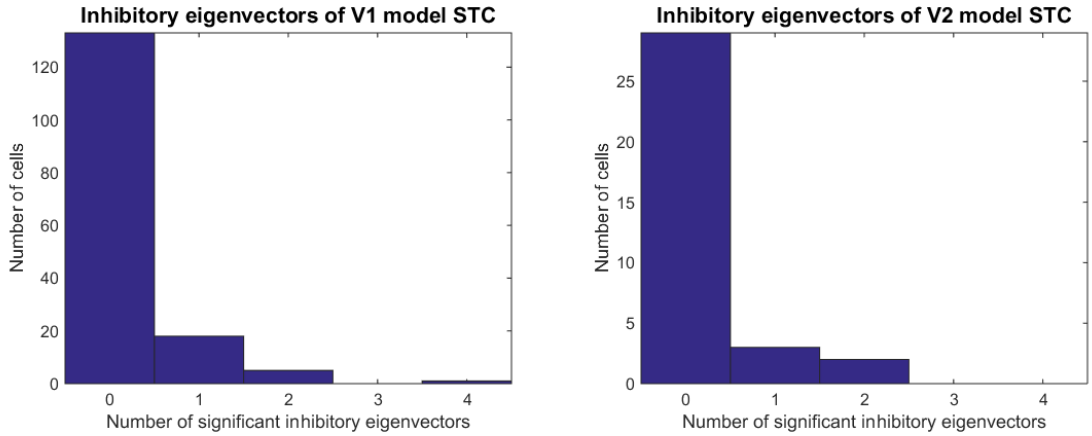


Figure 4.9: Distribution of significant inhibitory eigenvectors obtained by STC on V1/V2 models

These histograms show the number of significant inhibitory eigenvectors obtained by STC on the population of V1 and V2 models. These distributions are not significantly different, p=0.98.

shown in figure 3.9 in chapter 3. For the first neuron in 4.6A the STA shows little structure, while the first two eigenvectors capture phase invariant spatial frequency and orientation tuning. The scree plot on the right clearly shows two eigenvalues distinct from the rest. These two stimulus directions clearly resemble the filters in 3.9A. The latter two neuron models show no significant eigenvectors, but have clearly defined STAs that resemble the filters in figure 3.9B-3.9C. Based on the model filters shown in figure 3.9, the stimulus directions found by STA and STC come out exactly as would be expected.

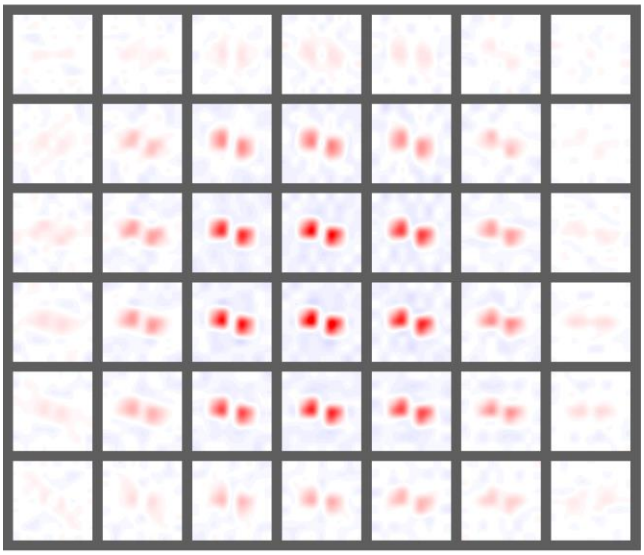
Figure 4.7 shows the STA and two eigenvectors of the STC matrix for the same V2 neuron models whose filters were shown in figure 3.10 of chapter 3. In all cases there are two significant excitatory eigenvectors of the STC matrix and little structure in the STA. However, it is obvious that the stimulus directions found by STC do not fully capture the diversity of filters seen in figure 3.10. The directions found by STC resemble those obtained from a V1 complex cell, as in figure 4.6A, and obscure the more complicated tuning inherent in the neural network. This result shows definitively that white noise STC analysis is insufficient to fully characterize V2 neurons. (Note that the three V1 and three V2 cells seen in figures 4.6 and 4.7 will also be used as examples in the simulations below.)

Interestingly, however, there was a significant difference between the distributions of significant excitatory eigenvectors obtained from the V1 and V2 model responses. Figure 4.8 shows that V2 models generally produce more significant excitatory eigenvectors from their STC matrices. (There were, however, no significant differences with the inhibitory eigenvectors, as seen in figure 4.9.) This confirms the general consensus that V2 neurons are more nonlinear than V1 neurons (Willmore, Prenger, and Gallant 2010; Anzai, Peng, and Van Essen 2007; Schmid et al. 2009; Freeman et al. 2013).

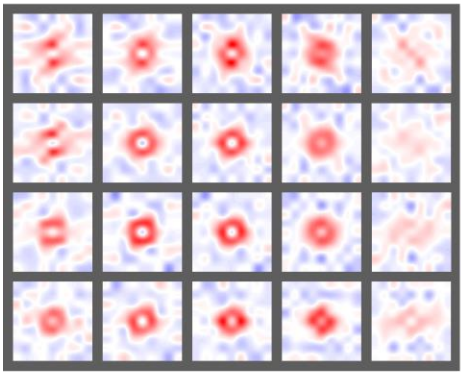
4.5 Local Spectral Reverse Correlation

An alternative way analyze neural responses to white noise is with Local Spectral Reverse Correlation (LSRC) (Nishimoto, Ishida, and Ohzawa 2006). Similarly to the STA / STC methodology, LSRC analyzes the stimuli that precede a spike. However, instead of computing the average or covariance of the pixels, LSRC computes the average 2D Fourier power of the spike triggered stimuli in several small overlapping windows throughout the receptive field. Because Fourier power is phase insensitive, LSRC can quantify orientation selectivity of both simple and complex cells.

A.



B.



C.

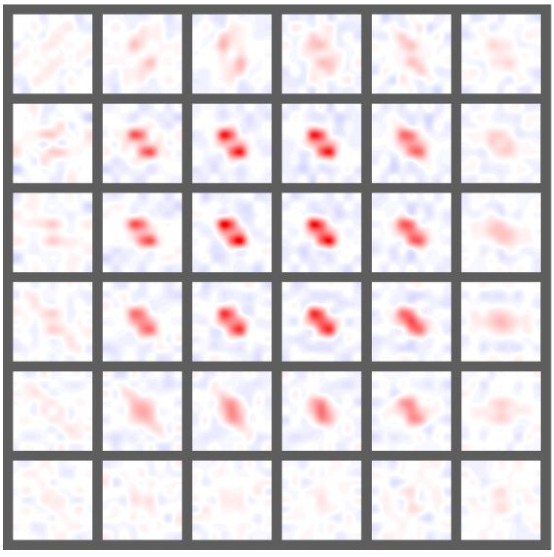
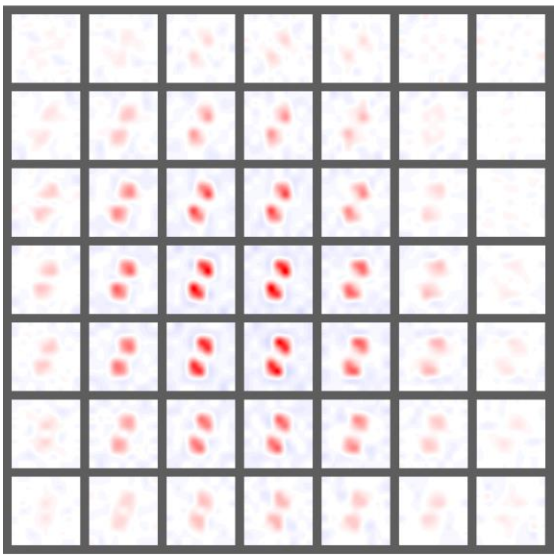


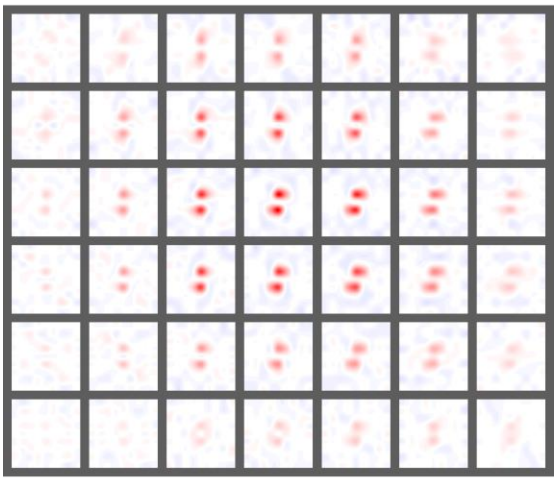
Figure 4.10: LSRC of V1 neuron models

Each square within each figure shows the spike triggered Fourier power for a windowed sub-region of the stimulus. In 2D Fourier power plots, distance from the center represents spatial frequency and the angle around the center represents orientation. In 4.10A and 4.10C the pairs of blobs that are relatively consistent across regions indicate that the orientation tuning is reasonably homogenous within the receptive field. The relative location of the blobs indicates the orientation and spatial frequency of the tuning. The ring-like pattern in 4.10B indicates tuning for all orientations and a particular spatial frequency.

A.



B.



C.

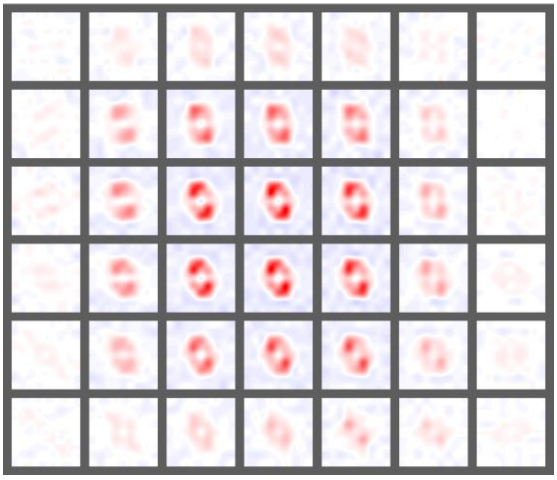


Figure 4.11: LSRC of V2 neuron models

Each square within each figure shows the spike triggered Fourier power for a windowed sub-region of the stimulus. In 2D Fourier power plots, distance from the center represents spatial frequency and the angle around the center represents orientation. Many of the blobs are oblong and curved indicating tuning to a range of orientations. The blobs in 4.11A and 4.11B vary subtly but consistently throughout the receptive field indicating changes in local orientation preferences. The near ring-like structure in 4.11C indicates broad orientation tuning.

Furthermore, the use of many small windows makes it possible to assess substructure within a cell's receptive field, such as regions with different orientation selectivity.

I performed LSRC on the same simulation data used for the STA/STC analysis in section 4.4. The results of LSRC on the V1 neuron models are shown in figure 4.10. Unsurprisingly, the receptive fields are quite homogenous. The pairs of blobs seen in the panels of 4.10A and 4.10C indicate robust orientation tuning, while the rings in 4.10B indicate an unoriented blob. This agrees nicely with the STA/STC results in figure 4.6.

The results of LSRC on the V2 neuron models are shown in figure 4.11. In each of these examples the blobs in each panel have a somewhat oblong curved shape, indicating selectivity for a range of orientations. In 4.11A and 4.11B, there are also subtle shifts of the peak intensity within the blobs across the receptive fields. This indicates that different parts of a receptive field have different orientation preferences. The blobs in figure 4.11C appear almost ring-like, similar to 4.10B. However, from the corresponding filters in figure 3.10C it is evident this neuron is not unoriented but rather tuned to several different orientations. Thus in cases of where a neuron's tuning is complicated and involves several orientations, LSRC may collapse across this complexity and make it appear to exhibit less orientation tuning. To assess the sensitivity of LSRC to orientation tuning inhomogeneity, I calculated the maximum orientation difference across all windows that contained a tuning

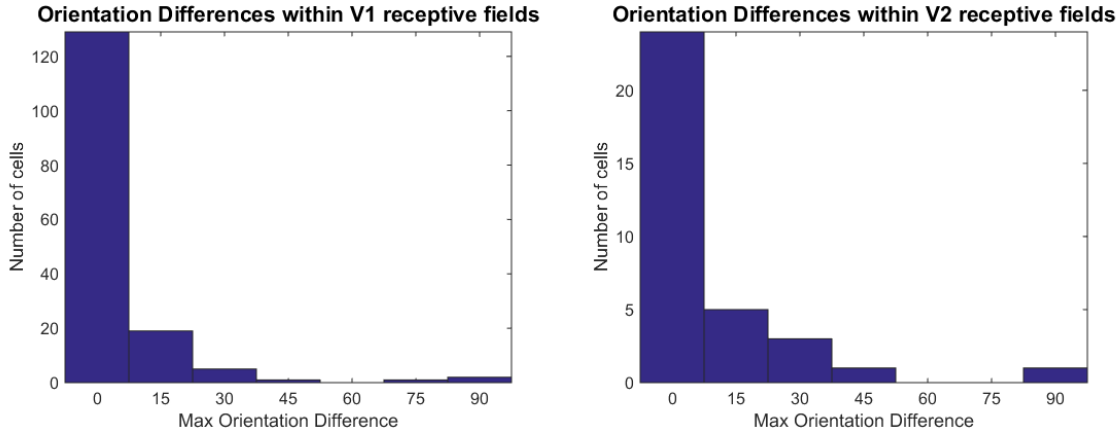


Figure 4.12: Maximum orientation selectivity differences within V1 and V2 receptive fields measured with LSRC

In this analysis both V1 and V2 appear to have largely homogenous receptive fields with the maximum orientation tuning difference within the receptive fields highly peaked at 0. There is not a statistically significant difference between these two distributions, p=0.098.

peak of at least 75% of the maximum. Figure 4.12 plots the distribution of these differences for the V1 and V2 models. In both cases LSRC indicates that the receptive fields are largely homogenous. V2 appears to have a larger proportion of non-zero orientation differences, but the difference is not statistically significant.

In this analysis LSRC does not seem to be especially sensitive to differences in orientation tuning throughout the receptive field. This is likely due to the small size of the receptive fields, and the small number of pixels that define them, limiting effective the spectral and spatial resolution. However, in cases where it succeeds, such as figure 4.11C, LSRC collapses the various filters present in the model (shown in 3.10C) into a blur of Fourier power.

4.6 Local orientation tuning

A straightforward way to assess orientation tuning substructure is to flash small sinusoidal gratings at different locations within the receptive field (Anzai, Peng, and Van Essen 2007). This study found that 30% of V2 neurons have locations within their receptive fields that differ in orientation tuning by over 30° while none of the V1 cells showed such differences. In this controlled experiment it is also possible to evaluate whether and how different regions of the receptive field interact, by flashing pairs of gratings. The study found that many V2 neurons showed suppressive interactions between receptive field sub-regions that caused the

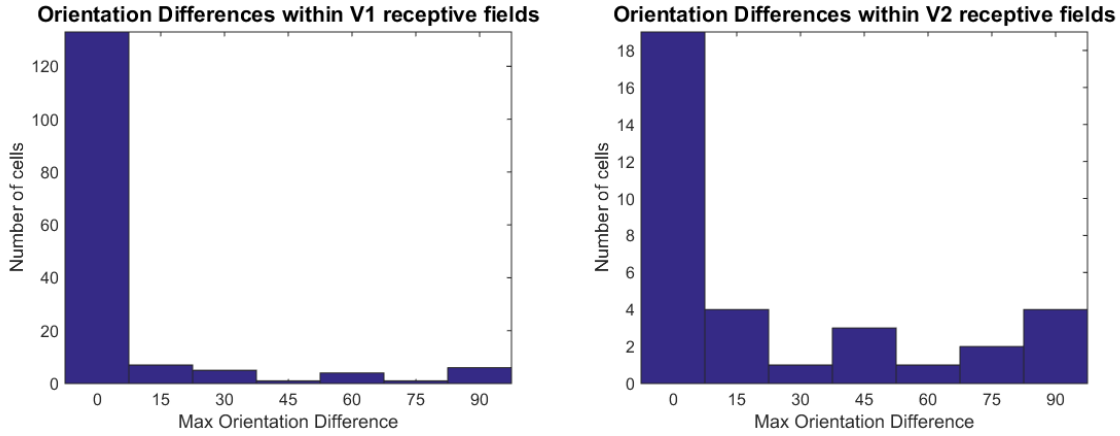


Figure 4.13: Maximum orientation selectivity differences within V1 and V2 receptive fields measured with grating patches

V1 receptive fields are largely homogenous with only 7.6% showing orientation selectivity differences within the receptive field of greater than 30°. And of these, most were artifacts due to unoriented receptive fields. However, in V2 29.4% of the receptive fields contained orientation differences greater than 30°. These distributions are significantly different, p=0.0004.

neurons to be selective for combinations of orientations (Anzai, Peng, and Van Essen 2007).

I repeated this experiment *in silico* on the V1/V2 models and found that 7.6% of the V1 models and 29.4% of the V2 models have locations within their receptive fields that differ in orientation tuning by over 30°. To reduce spurious effects due to small overall responses, locations within the receptive field were only counted if their response exceeded 75% of the maximum response. The percentage of V2 cells exhibiting inhomogeneous orientation tuning, aligns nicely with the original study, but a significant number of V1 cells seemed to have inhomogeneous orientation tuning as well. This turned out to mainly be an artifact due to the presence of unoriented V1 cells in this sample. The distributions of maximum orientation differences within V1 and V2 receptive fields are shown in figure 4.13. For the V1/V2 models, small flashed gratings seem to be a more sensitive method for assessing inhomogeneous orientation tuning than LSRC.

Polar plots of the orientation tuning at each receptive field location are shown for the three example V1 cells in figure 4.14 and for the three example V2 cells in figure 4.15. The first and third V1 examples exhibit strong homogenous orientation tuning as expected. But, interestingly, even the second V1 example exhibits some orientation tuning despite the fact that the filters in the neural network model are all unoriented blobs, as seen in figure 3.9B. This seems to be an artifact due to how the gratings of various orientations overlapped with the small receptive field.

Of the V2 examples, the second in 4.15B shows the most inhomogeneous orientation tuning. Analogously to the original study, I performed an *in silico* interaction experiment at the two labeled locations in 4.15B. The interaction result, which is plotted in 4.15D, closely resembles some of the examples presented in the original study. The two locations seem to interact nonlinearly; the shape of the tuning curve at one location completely changes depending on the orientation presented at the other location. However, it is not obvious how this pattern of apparent interaction could emerge from a model using the elongated filters shown in figure 3.10B. This also calls into question the mechanisms proposed by the original study to explain such interactions, i.e. complex patterns of inhibition between various Gabor-like filters with different orientations (Anzai, Peng, and Van Essen 2007). It is quite possible that probing a nonlinear system in a constrained and systematic way might lead to interpretable behavior that doesn't accurately reflect the natural behavior of the system. This is a danger for interpreting both the original study and this *in silico* experiment on the models.

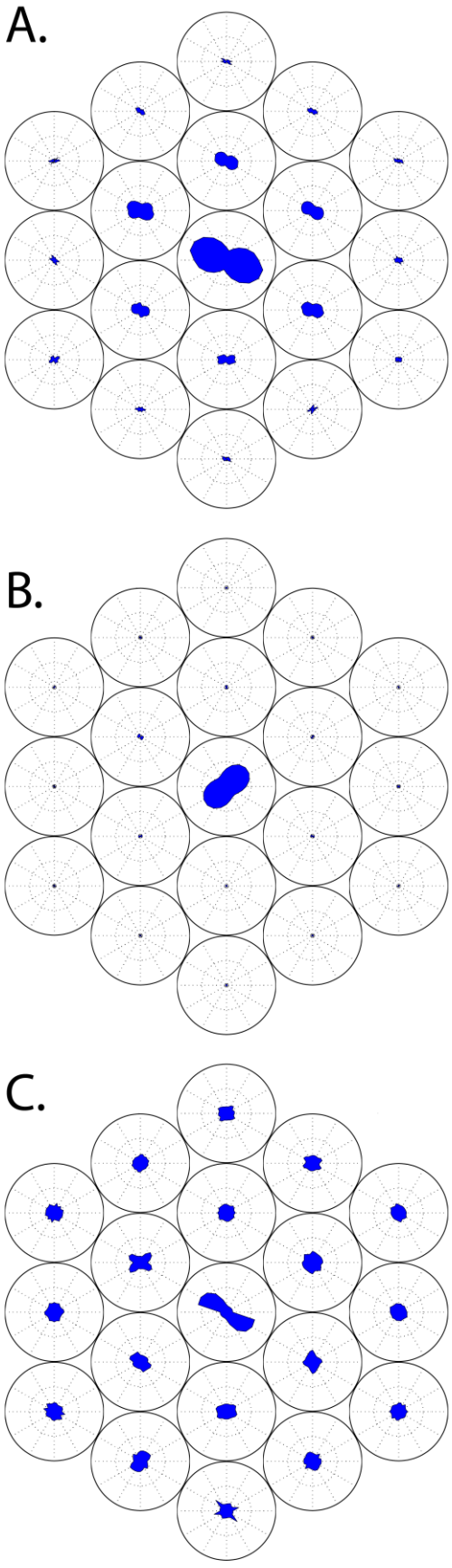


Figure 4.14: Local orientation tuning of V1 models

Each circle represents the location of a grating flashed within the receptive field. Within each circle is a polar plot showing the orientation tuning at that location. The orientation tuning in 4.11A is consistent among the central locations. In the second example, 4.11B, there are only responses at the center location, and there appears to be some orientation tuning. This is likely an artifact since all the filters in the model are unoriented blobs. There is strong orientation tuning in the center of 4.11C and small responses elsewhere, likely due to the apparent noise in the filters seen in 3.9C.

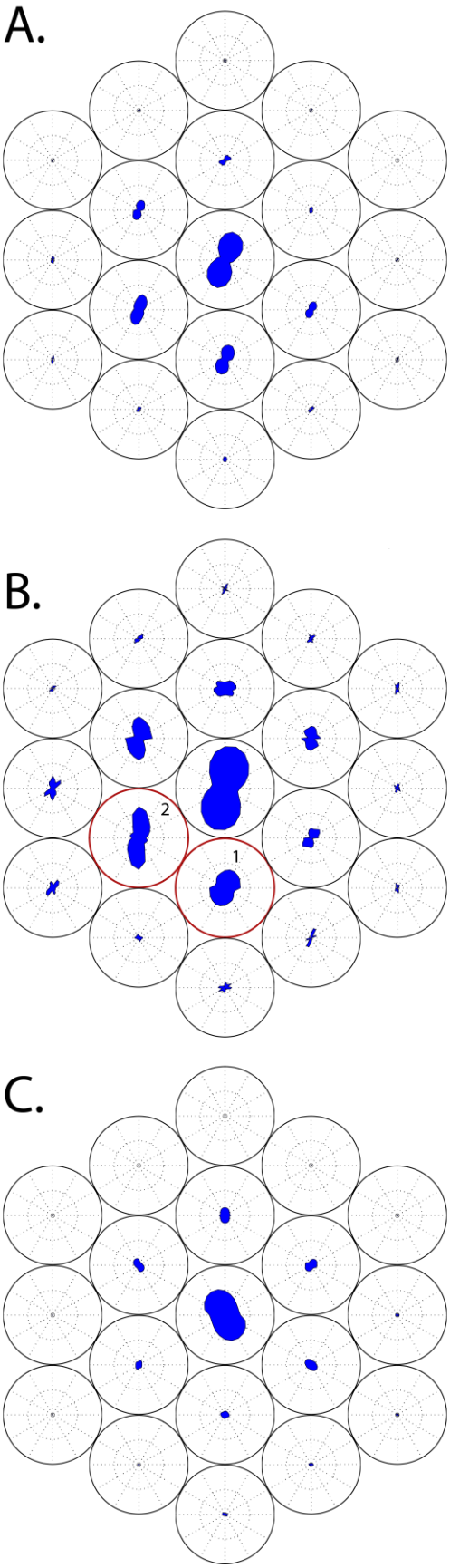
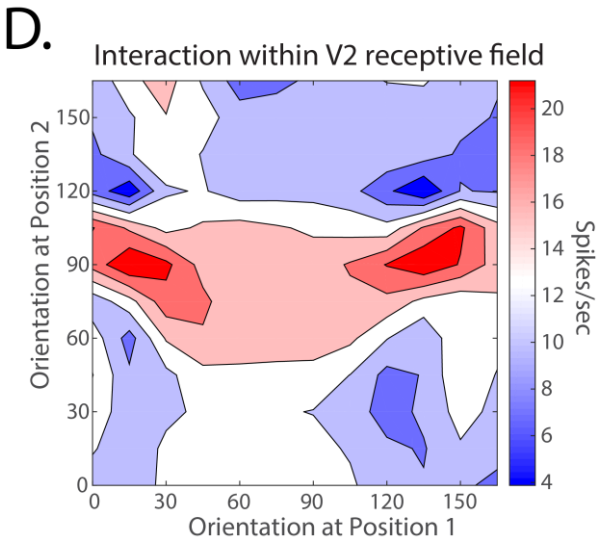


Figure 4.15: Local orientation tuning of V2 models

Each circle represents the location of a grating flashed within the receptive field. Within each circle is a polar plot showing the orientation tuning at that location. In the first and second example, figures 4.15A and 4.15B there are small shifts in orientation preference throughout the receptive field but none over 30°. The second also shows an interesting diversity of tuning curve shapes. The third example, 4.15C, only shows strong responses in the center but shows weak orientation tuning. This is likely due to the fact that the model for this neuron contains filters of several orientations and its true selectivity is not being adequately captured. Figure 4.15D plots the interaction of the positions labeled in 4.15B. These positions interact nonlinearly causing the shape of the tuning curve at one position to depend on the orientation present at the other location. Inhibitory effects (in blue) help create selectivity to particular combinations of orientations.



By the standard of the original study, i.e. a peak orientation tuning difference of over 30°, all of the V2 model examples in figure 4.15 would be considered to have homogenous receptive fields. This lack of sensitivity to interesting receptive field structure is probably partly due to the small receptive fields in this data and the consequent difficulty in effectively sampling different sub-regions using coarsely sampled non-overlapping locations. However, given the diversity of filters present in all of these V2 models, it seems that interesting structure in a large percentage of V2 receptive fields may be masked in this experiment by only considering orientation differences of over 30° as evidence of inhomogeneity.

4.7 Angle selectivity

It has been hypothesized that role of V2 neurons is to combine signals from V1 to generate selectivity to junctions of edges, such as corners. An explicit study of this hypothesis used pairs of parameterized line segments at various relative orientations to probe V2 neurons for angle selectivity (Ito and Komatsu 2004). The study found that many V2 neurons responded selectively to a small set of angles at particular orientations. However, the study also found that in many cases the neurons would respond equally well to one of the two line segments that formed the optimal angle stimuli. These two observations indicate that adding oriented stimuli in some regions of the receptive field inhibits the response to the optimal line segment.

To determine if the models of V2 cells exhibited a comparable pattern of tuning, I generated a similar stimulus set and ran it through the model for each V2 cell. The

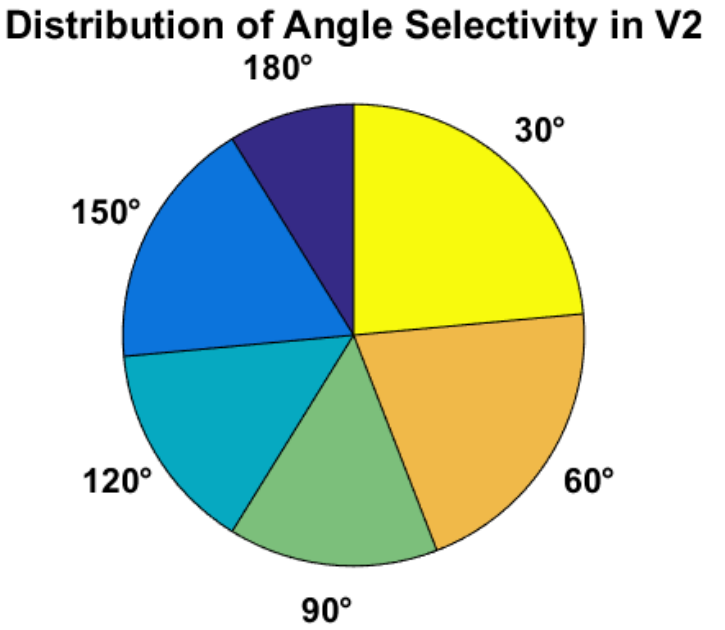


Figure 4.16: Distribution of angle selectivity in models of V2 neurons

This plot shows the distribution of the relative angle between the two line segments. In agreement with the original study I found the preferred angles of the V2 models to be quite evenly distributed with no strong bias. The null hypotheses of a uniform distribution is not rejected, p=0.75.

V2 models had a roughly even distribution of preferred angle widths, which agrees with the result in the original study (see in figure 4.16). Figure 4.17 shows the angle tuning plots for the three example V2 models. The patterns of selectivity closely resemble those seen in the original study and are consistent with the orientation selectivity seen in the V2 model filters in figure 3.10.

However, performing the same *in silico* experiment on the V1 neuron models yielded qualitatively similar results. The distribution of preferred angles over all the V1 models was similar and the angle selectivity profiles of many individual V1 models were quite similar to those seen in figure 4.17 and in the original paper. There was no obvious criterion by which to distinguish V1 and V2 neurons based on this stimulus set. Apparent angle tuning in V1 models could arise even when they lack the necessary filters for true angle tuning due to happenstance alignment

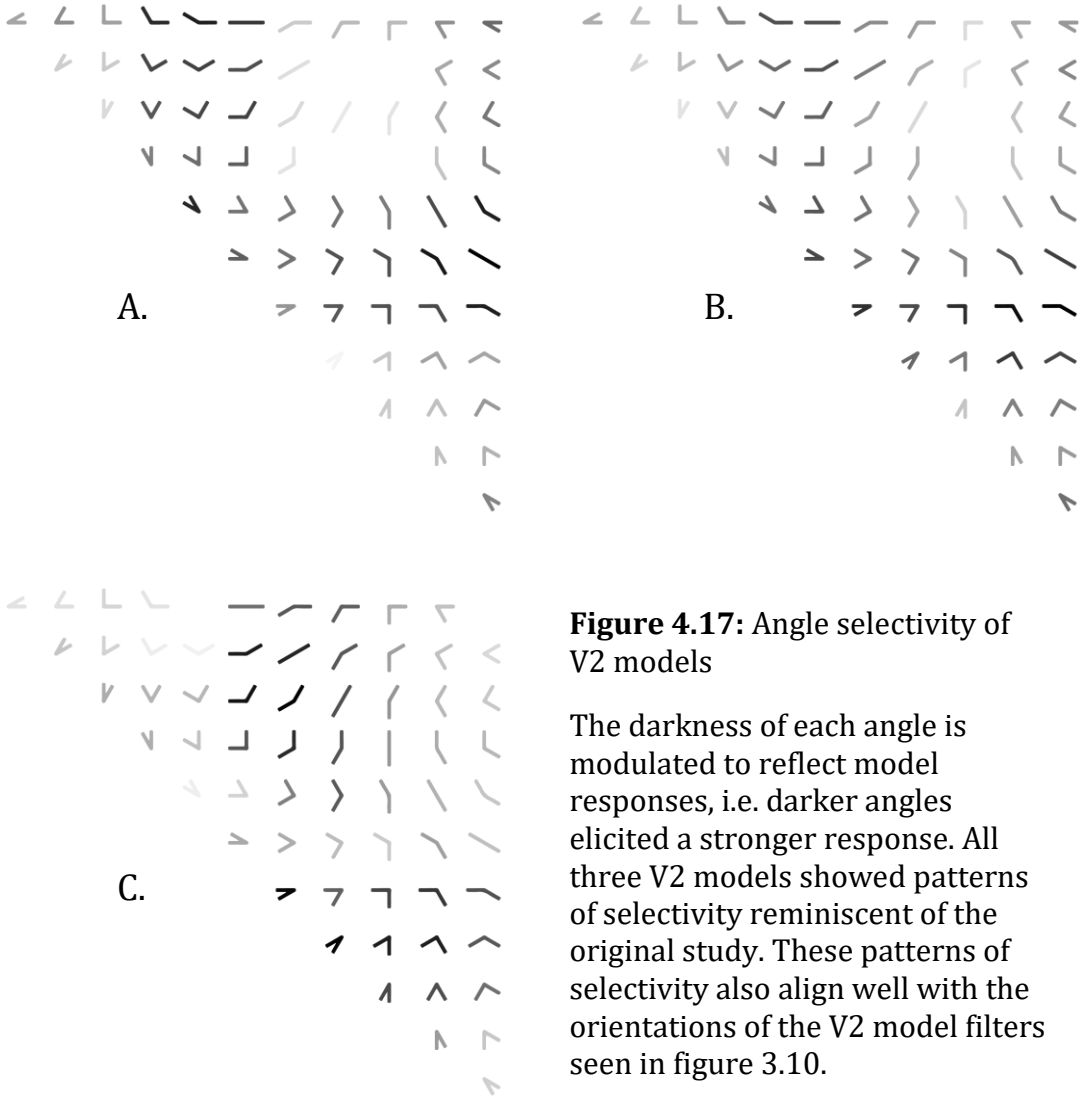


Figure 4.17: Angle selectivity of V2 models

The darkness of each angle is modulated to reflect model responses, i.e. darker angles elicited a stronger response. All three V2 models showed patterns of selectivity reminiscent of the original study. These patterns of selectivity also align well with the orientations of the V2 model filters seen in figure 3.10.

between the stimulus and the receptive field components. While the angle tuning results make sense for the example V2 models, the results in V1 indicate there is a high likelihood for many false negatives and positives. This fact makes interpretation of the results in V2 somewhat questionable. Such "apparent" tuning results are always a danger when using a small artificial stimulus set in only one visual area.

4.8 Conclusions

I have shown that neural network models of V1 and V2 behave similarly to real neurons in a range of standard neurophysiology experiments. However, having the ground truth of the model makes it possible to assess the strengths and weaknesses of various experimental paradigms. I have shown that standard neurophysiology experiments are often relatively insensitive to complex structure in V2 receptive fields and tend to collapse over interesting variation. Furthermore, I have shown that small, artificial stimulus sets designed to test specific hypotheses have the potential to lead to spurious conclusions. I believe it is for all these reasons that there has been only slow progress in characterizing V2. The most important result from running these *in silico* simulations turned out not to be a better understanding of the computations performed by the neural network, but rather new insights into the limitations of current experimental techniques. There is thus a great need for new experimental stimuli that can stimulate V2 in controlled but ecologically valid ways. By using these neural network models of V2, researchers will be able rapidly prototype future experiments in order to avoid methods that conceal the complexity of V2 responses.

Chapter 5

Conclusion

Chapter 5

Conclusion

5.1 Summary of contributions

The main contributions of this dissertation are:

- I have introduced STRFlab, an object-oriented MATLAB toolbox for Spatio-Temporal Receptive Field (STRF) estimation. I have incorporated into STRFlab state-of-the-art linearizing transforms and statistical learning algorithms in order to give a wide range neuroscientists easy access to the most powerful STRF estimation methodologies. I have described the principles behind the underlying machinery for fitting and regularizing a STRF model and worked through realistic STRF estimation examples so that new users can get STRFlab working on their own problems as quickly as possible.
- I created a general method for fitting high order Volterra series models to large high dimensional data sets called boosted inhomogeneous polynomial kernel networks (IPKN). This new method makes use of connections between neural networks and kernel regression and leverages the powerful stochastic gradient boosting algorithm. This method extends and generalizes the traditional STA and STC techniques (since they are equivalent to the first and second order Volterra coefficients) by enabling the simultaneous estimation of first and second order terms using natural stimuli with the Poisson error function. Furthermore, the method facilitates the fitting of previously intractable high order models. I have shown that this method performs well on both receptive field estimation tasks and general system identification problems.
- Recent developments in the neural network community have made neural networks far more attractive for STRF estimation. I have tested and evaluated many recently developed techniques, as well as created some new extensions, in order to find a robust procedure for fitting STRFs using deep time-delay neural networks. Deep time-delay neural networks are an attractive modeling framework because they make no assumptions about the underlying data and can in principle model any function. I have shown that given adequate data and proper regularization, deep time-delay neural networks can substantially outperform state-of-the-art linearized regression techniques for STRF estimation. Additionally I have shown the first pixel-space visualization of feature tuning in V2 neurons.

- Finally, I have shown that deep neural network models of V1 and V2 neurons behave similarly to real neurons in a series of *in silico* experiments. Because the feature space of the neural network models can be directly visualized, each of the experimental techniques could be assessed based on its ability to recover information about this feature space. These experiments exposed some of the inherent limitations of standard electrophysiology experiments. Many of the standard techniques could find large scale differences in V2 receptive field architecture relative to V1, but tended to obscure finer scale detail and structure. More worrisome, however, was the tendency for some techniques to produce interpretable results that did not reflect the most important computations performed by the model. I plan to make the neural network models available to neuroscience community to aid in constructing more powerful future experiments.

5.2 Open questions and future directions

This work leaves open several important issues both in terms of methods and in the interpretation of results. There are especially many opportunities for continued research using the deep neural network framework. The new pooling function described in section 3.9 would enable single hidden units in neural networks to learn either selectivity or invariance to their inputs, greatly enhancing the representational capacity of the whole network. This would have application both for STRF modeling and in the wider neural network community. Additionally, it would be interesting to incorporate the inhomogeneous polynomial kernel function described in section 2.3 into a deep architecture. Each layer of the deep network would thus compute first and second order transformations of their input. Given how well V1 neurons can be modeled using first and second order models, it is interesting to consider whether stacking such a representation into a deep architecture could effectively model the transformations between visual areas.

Another path for future investigation involves interpreting the neural networks fit to neural data. Interpreting neural networks in general is an active area of research. A promising path forward is interpreting the neural networks in terms of the feature space learned in the first layer. While these features can be directly visualized, the network performs nonlinear combinations of their outputs. Quantifying the extent of these interactions and determining their functional relevance is important for understanding both the networks and the neural computations that they model.

5.3 Closing remarks

Properly characterizing the brain's internal representations of the world is crucial for a deep understanding of the workings of our minds. We are, however, only at the beginning of the road to grasping these representations. To make progress we must fully embrace the complexity of neural computation and be wary of simple explanations. We must accept the possibility that neural computations may not be accessible to intuition and can only be understood mathematically. For these reasons, this dissertation has focused more on rigorous quantitative characterizations of neural computations than on creating simple and intuitive interpretations.

Bibliography

Albrecht, D G, D B Hamilton, and N N O David. 1982. “Striate Cortex of Monkey and Cat: Contrast Response Function.” *J Neurophysiol* 48 (1): 217–37.

Anzai, A, X Peng, and D C Van Essen. 2007. “Neurons in Monkey Visual Area V2 Encode Combinations of Orientations.” *Nature Neuroscience* 10 (10): 1313–21. doi:10.1038/nn1975.

Appelle, S. 1972. “Perception and Discrimination as a Function of Stimulus Orientation: The ‘Oblique Effect’ in Man and Animals.” *Psychological Bulletin* 78: 266–78. doi:10.1037/h0033117.

Beck, A, and M Teboulle. 2009. “A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems.” *SIAM Journal on Imaging Sciences*. doi:10.1137/080716542.

Bengio, Y, and O Delalleau. 2011. “On the Expressive Power of Deep Architectures.” In *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6925 LNAI:18–36. doi:10.1007/978-3-642-24412-4_3.

Bottou, L. 1991. “Stochastic Gradient Learning in Neural Networks.” *Proceedings of Neuro-Nımes*.

———. 2012. “Stochastic Gradient Descent Tricks.” In *Neural Networks: Tricks of the Trade*, 421–36.

Breiman, L. 1996. “Bagging Predictors - Springer.” *Machine Learning*. Kluwer Academic Publishers.

Bühlmann, P, and B Yu. 2006. “Sparse Boosting.” *The Journal of Machine Learning Research* 7 (December). JMLR.org: 1001–24.

Chichilnisky, E J. 2001. “A Simple White Noise Analysis of Neuronal Light Responses.” *Network (Bristol, England)* 12 (2). Systems Neurobiology, The Salk Institute, La Jolla, CA 92037-1099, USA. ej@salk.edu: 199–213.

Conway, B R. 2001. “Spatial Structure of Cone Inputs to Color Cells in Alert Macaque Primary Visual Cortex (V-1).” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 21 (8): 2768–83.

Conway, B R, and M S Livingstone. 2006. “Spatial and Temporal Properties of Cone Signals in Alert Macaque Primary Visual Cortex.” *The Journal of Neuroscience :*

The Official Journal of the Society for Neuroscience 26 (42): 10826–46. doi:10.1523/JNEUROSCI.2091-06.2006.

Conway, B R, S Moeller, and D Y Tsao. 2007. "Specialized Color Modules in Macaque Extrastriate Cortex." *Neuron* 56 (3): 560–73. doi:10.1016/j.neuron.2007.10.008.

Dahl, G E, T N Sainath, and G E Hinton. 2013. "Improving Deep Neural Networks for LVCSR Using Rectified Linear Units and Dropout." *Acoustics*.

David, S V, and J L Gallant. 2005. "Predicting Neuronal Responses during Natural Vision." *Network: Computation in Neural Systems* 16 (2-3): 239–60. doi:10.1080/09548980500464030.

David, S V, B Y Hayden, and J L Gallant. 2006. "Spectral Receptive Field Properties Explain Shape Selectivity in Area V4." *Journal of Neurophysiology* 96 (6): 3492–3505. doi:10.1152/jn.00575.2006.

David, S V, W E Vinje, and J L Gallant. 2004. "Natural Stimulus Statistics Alter the Receptive Field Structure of v1 Neurons." *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 24 (31): 6991–7006. doi:10.1523/JNEUROSCI.1422-04.2004.

De Boer, R, and P Kuyper. 1968. "Triggered Correlation." *IEEE Transactions on Bio-Medical Engineering* 15: 169–79. doi:10.1109/TBME.1968.4502561.

Dodd, T J. 2002. "A New Solution to Volterra Series Estimation." *CD-Rom Proc 2002 IFAC World Congress*.

Dow, B M, R G Vautin, and R Bauer. 1985. "The Mapping of Visual Space onto Foveal Striate Cortex in the Macaque Monkey." *Journal of Neuroscience* 5 (4): 890–902.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12: 2121–59.

Efron, B. 1982. "The Jackknife, the Bootstrap and Other Resampling Plans."

Efron, B, T Hastie, I Johnstone, R Tibshirani, H Ishwaran, K Knight, JM Loubes, et al. 2004. "Least Angle Regression." *Annals of Statistics* 32: 407–99. doi:10.1214/009053604000000067.

Engel, Y, S Mannor, and R Meir. 2004. "IEEE Xplore - The Kernel Recursive Least-Squares Algorithm." *Signal Processing*.

Favier, G, A Y Kibangou, and T Bouilloc. 2012. “Nonlinear System Modeling and Identification Using Volterra-PARAFAC Models.” *International Journal of …* 26 (1): 30–53. doi:10.1002/acs.1272.

Foster, K H, J P Gaska, M Nagler, and D A Pollen. 1985. “Spatial and Temporal Frequency Selectivity of Neurones in Visual Cortical Areas V1 and V2 of the Macaque Monkey.” *The Journal of Physiology* 365: 331–63.

Franz, M O, and B Schölkopf. 2006. “A Unifying View of Wiener and Volterra Theory and Polynomial Kernel Regression.” *Neural Computation* 18 (12). mof@tuebingen.mpg.de: 3097–3118.

Freeman, J, C M Ziemba, D J Heeger, E P Simoncelli, and J A Movshon. 2013. “A Functional and Perceptual Signature of the Second Visual Area in Primates.” *Nature Neuroscience* 16 (7). Center for Neural Science, New York University, New York, New York, USA. freemanj11@janelia.hhmi.org: 974–81.

Friedman, J. 1999. “Greedy Function Approximation: A Gradient Boosting Machine.” *Ann Statist* 29 (5). Institute of Mathematical Statistics: 1189–1232.

———. 2002. “Stochastic Gradient Boosting.” *Computational Statistics & Data Analysis* 38 (4): 367–78.

Friedman, J, T Hastie, and R Tibshirani. 2010a. “Regularization Paths for Generalized Linear Models via Coordinate Descent.” *Journal of Statistical Software* 33: 1–22. doi:10.1359/JBMR.0301229.

———. 2010b. “A Note on the Group Lasso and a Sparse Group Lasso,” January.

Friedman, J, and B E Popescu. 2003. “Gradient Directed Regularization for Linear Regression and Classification.”

Gallant, J L, C E Connor, S Rakshit, J W Lewis, and D C Van Essen. 1996. “Neural Responses to Polar, Hyperbolic, and Cartesian Gratings in Area V4 of the Macaque Monkey.” *Journal of Neurophysiology* 76 (4): 2718–39.

Glorot, X, A Bordes, and Y Bengio. 2011. “Deep Sparse Rectifier Neural Networks.” *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011* 15: 315–23.

Gottschalk, A, M G Sexton, and G Roschke. 2004. “Multiplicative Neural Noise Can Favor an Independent Components Representation of Sensory Input.” *Network (Bristol, England)* 15 (4): 291–311.

Gulcehre, C, K Cho, R Pascanu, and Y Bengio. 2013. “Learned-Norm Pooling for Deep Feedforward and Recurrent Neural Networks,” November.

Heeger, D J. 1992. “Normalization of Cell Responses in Cat Striate Cortex.” *Visual Neuroscience* 9 (2): 181–97.

Hegdé, J, and D C Van Essen. 2000. “Selectivity for Complex Shapes in Primate Visual Area V2.” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 20 (5). Department of Anatomy and Neurobiology, Washington University School of Medicine, St. Louis, Missouri 63110, USA.: RC61.

Hinton, G E, N Srivastava, A Krizhevsky, I Sutskever, and R Salakhutdinov. 2012. “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors.” *arXiv.org*. Cornell University Library.

Holt, G R, and C Koch. 1997. “Shunting Inhibition Does Not Have a Divisive Effect on Firing Rates.” *Neural Computation* 9 (5): 1001–13.

Hsu, A, A Borst, and F E Theunissen. 2004. “Quantifying Variability in Neural Responses and Its Application for the Validation of Model Predictions.” *Network: Computation in …*. Informa UK Ltd UK.

Hubel, D H, and T N Wiesel. 1959. “Receptive Fields of Single Neurones in the Cat’s Striate Cortex.” *The Journal of Physiology* 148: 574–91. doi:10.1113/jphysiol.2009.174151.

Huber, P J. 1964. “Robust Estimation of a Location Parameter.” *The Annals of Mathematical Statistics* 35 (1). Institute of Mathematical Statistics: 73–101.

Ito, M, and H Komatsu. 2004. “Representation of Angles Embedded within Contour Stimuli in Area V2 of Macaque Monkeys.” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 24 (13). Division of Sensory and Cognitive Information, Department of Information Physiology, National Institute for Physiological Sciences, Okazaki, Aichi 444-8585, Japan. minami@nips.ac.jp: 3313–24.

Jones, J P, and L A Palmer. 1987. “An Evaluation of the Two-Dimensional Gabor Filter Model of Simple Receptive Fields in Cat Striate Cortex.” *Journal of Neurophysiology* 58 (6). Department of Anatomy, University of Pennsylvania School of Medicine, Philadelphia 19104-6058.: American Physiological Society: 1233–58.

Kagan, I, M Gur, and D M Snodderly. 2002. “Spatial Organization of Receptive Fields of V1 Neurons of Alert Monkeys: Comparison with Responses to Gratings.” *Journal of Neurophysiology* 88 (5): 2557–74. doi:10.1152/jn.00858.2001.

Kobatake, E, and K Tanaka. 1994. “Neuronal Selectivities to Complex Object Features in the Ventral Visual Pathway of the Macaque Cerebral Cortex.” *Journal of Neurophysiology* 71 (3): 856–67.

Korenberg, M J. 1991. “Parallel Cascade Identification and Kernel Estimation for Nonlinear Systems.” *Annals of Biomedical Engineering* 19 (4): 429–55. doi:10.1007/BF02584319.

Korenberg, M J, and I W Hunter. 1996. “The Identification of Nonlinear Biological Systems: Volterra Kernel Approaches.” *Annals of Biomedical Engineering* 24. doi:10.1007/BF02648117.

Kusunoki, M, K Moutoussis, and S M Zeki. 2006. “Effect of Background Colors on the Tuning of Color-Selective Cells in Monkey Area V4.” *Journal of Neurophysiology* 95 (5): 3047–59. doi:10.1152/jn.00597.2005.

Li, B, M R Peterson, and R D Freeman. 2003. “Oblique Effect: A Neural Basis in the Visual Cortex.” *Journal of Neurophysiology* 90: 204–17. doi:10.1152/jn.00954.2002.

Maas, A L, A Y Hannun, and A Y Ng. 2013. “Rectifier Nonlinearities Improve Neural Network Acoustic Models.” *ICML Workshop on Deep Learning for …*.

Mackay, D. 1995. “Probable Networks and Plausible Predictions — a Review of Practical Bayesian Methods for Supervised Neural Networks.” *Network: Computation in Neural Systems*. doi:10.1088/0954-898X/6/3/011.

Mahon, L E, and R L De Valois. 2001. “Cartesian and Non-Cartesian Responses in LGN, V1, and V2 Cells.” *Visual Neuroscience* 18 (6). Cambridge University Press: 973–81.

Mairal, J, F Bach, J Ponce, and G Sapiro. 2010. “Online Learning for Matrix Factorization and Sparse Coding.” *The Journal of Machine Learning Research* 11. JMLR.org: 19–60.

Marmarelis, P. 1978. *Analysis of Physiological Systems : The White-Noise Approach*. New York: Plenum Press.

Marmarelis, V Z, and X Zhao. 1997. “Volterra Models and Three-Layer Perceptrons.” *IEEE Transactions on Neural Networks / a Publication of the IEEE Neural Networks Council* 8 (6): 1421–33.

Møller, M F. 1993. “A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning.” *Neural Networks* 6 (4): 525–33.

Nelder, J A, and R W M Wedderburn. 1972. “Generalized Linear Models.” *Journal of the Royal Statistical Society, Series A* 135: 370–84. doi:10.2307/2344614.

Nishimoto, S, and J L Gallant. 2011. “A Three-Dimensional Spatiotemporal Receptive Field Model Explains Responses of Area MT Neurons to Naturalistic Movies.” *Journal of Neuroscience*. doi:10.1523/JNEUROSCI.6801-10.2011.

Nishimoto, S, T Ishida, and I Ohzawa. 2006. “Receptive Field Properties of Neurons in the Early Visual Cortex Revealed by Local Spectral Reverse Correlation.” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 26 (12). Graduate School of Frontier Biosciences, Osaka University, Osaka 560-8531, Japan.: 3269–80.

Nordhausen, C T, P J Rousche, and R A Normann. 1994. “Optimizing Recording Capabilities of the Utah Intracortical Electrode Array.” *Brain Research* 637: 27–36. doi:10.1016/0006-8993(94)91213-0.

Olshausen, B A, and D J Field. 1997. “Sparse Coding with an Overcomplete Basis Set: A Strategy Employed by V1?” *Vision Research* 37 (23). Department of Psychology, Cornell University, Ithaca, NY 14853, USA. bruno@redwood.ucdavis.edu: 3311–25. doi:10.1016/S0042-6989(97)00169-7.

Park, I M, and J W Pillow. 2011. “Bayesian Spike-Triggered Covariance Analysis.” *NIPS*.

———. 2013. “Bayesian Inference for Low Rank Spatiotemporal Neural Receptive Fields.” In *Advances in Neural Information Processing Systems*, 2688–96.

Pasupathy, A, and C E Connor. 2001. “Shape Representation in Area V4: Position-Specific Tuning for Boundary Conformation.” *Journal of Neurophysiology* 86 (5): 2505–19.

Perrone, M P, and L N Cooper. 1992. “When Networks Disagree: Ensemble Methods for Hybrid Neural Networks.”

Picard, R R, and R D Cook. 1984. “Cross-Validation of Regression Models.” *Journal of the American Statistical Association* 79 (387). Taylor & Francis Group: 575–83.

Pillow, J W. 2005. “Prediction and Decoding of Retinal Ganglion Cell Responses with a Probabilistic Spiking Model.” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 25 (47): 11003–13.

Prenger, R J. 2008. “Encoding and Decoding of Natural Images by the Mid-Level Visual System.”

Prenger, R J, M C-K Wu, S V David, and J L Gallant. 2004. “Nonlinear V1 Responses to Natural Scenes Revealed by Neural Network Analysis.” *Neural Networks : The Official Journal of the International Neural Network Society* 17 (5-6): 663–79. doi:10.1016/j.neunet.2004.03.008.

Purpura, K P, J D Victor, and E Katz. 1994. “Striate Cortex Extracts Higher-Order Spatial Correlations from Visual Textures.” *Proceedings of the National Academy of Sciences of the United States of America* 91 (18). Department of Neurology and Neuroscience, Cornell University Medical College, New York, NY 10021.: 8482–86.

Ranganathan, A, M-H Yang, and J Ho. 2011. “Online Sparse Gaussian Process Regression and Its Applications.” *IEEE Transactions on Image Processing : A Publication of the IEEE Signal Processing Society* 20 (2). Honda Research Institute, Mountain View, CA 94041, USA. aranganathan@honda-ri.com: 391–404.

Ranzato, M A, F J Huang, Y L Boureau, and Y LeCun. 2007. “Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition.” In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. doi:10.1109/CVPR.2007.383157.

Richards, J R, and J P Congalidis. 1987. “Feedforward and Feedback Control of a Copolymerization Reactor,” 286–306.

Riedmiller, M, and H Braun. 1992. “RPROP - A Fast Adaptive Learning Algorithm.” In *Proceedings of the International Symposium on Computer and Information Science VII*.

Riesenhuber, M, and T Poggio. 1999. “Hierarchical Models of Object Recognition in Cortex.” *Nature Neuroscience* 2: 1019–25. doi:10.1038/14819.

Ringach, D L. 2002. “Spatial Structure and Symmetry of Simple-Cell Receptive Fields in Macaque Primary Visual Cortex.” *Journal of Neurophysiology*, 455–63.

Rumelhart, D E, G E Hinton, and R J Williams. 1986a. “Learning Representations by Back-Propagating Errors.” *Nature*. doi:10.1038/323533a0.

———. 1986b. “Learning Internal Representations by Error Propagation.” In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1:318–62.

Rust, N C, O Schwartz, J A Movshon, and E P Simoncelli. 2005. “Spatiotemporal Elements of Macaque v1 Receptive Fields.” *Neuron* 46 (6): 945–56. doi:10.1016/j.neuron.2005.05.021.

Sceniak, M P, D L Ringach, M J Hawken, and R M Shapley. 1999. “Contrast ’ S Effect on Spatial Summation by Macaque V1 Neurons.” *America*, 733–39.

Schmid, A M, K P Purpura, I E Ohiorhenuan, F Mechler, and J D Victor. 2009. “Subpopulations of Neurons in Visual Area v2 Perform Differentiation and Integration Operations in Space and Time.” *Frontiers in Systems Neuroscience* 3 (January): 15. doi:10.3389/neuro.06.015.2009.

Schwartz, O, J W Pillow, N C Rust, and E P Simoncelli. 2006. “Spike-Triggered Neural Characterization.” *Journal of Vision* 6 (4): 484–507. doi:10.1167/6.4.13.

Sermanet, P, S Chintala, and Y LeCun. 2012. “Convolutional Neural Networks Applied to House Numbers Digit Classification.” *Proceedings of International Conference on Pattern Recognition ICPR12*, 10–13.

Shampine, L F. 1994. *Numerical Solution of Ordinary Differential Equations, Volume 4*. CRC Press.

Skottun, B C, R L De Valois, D H Grosof, J A Movshon, D G Albrecht, and A B Bonds. 1991. “Classifying Simple and Complex Cells on the Basis of Response Modulation.” *Vision Research* 31: 1079–86. doi:10.1016/0042-6989(91)90033-2.

Sollich, P, and A Krough. 1996. “Learning with Ensembles: How over-Fitting Can Be Useful.” *NIPS*.

Soni, Abhishek. 2006. “Control-Relevant System Identification Using Nonlinear Volterra and Volterra-Laguerre Models,” June.

Srivastava, N, G E Hinton, A Krizhevsky, I Sutskever, and R Salakhutdinov. 2014. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” *Journal of Machine Learning Research* 15: 1929–58.

Stein, R B. 1967. “Some Models of Neuronal Variability.” *Biophysical Journal* 7 (1): 37–68. doi:10.1016/S0006-3495(67)86574-3.

Sutskever, I, J Martens, G E Dahl, and G E Hinton. 2013. “On the Importance of Initialization and Momentum in Deep Learning.” *JMLR W&CP* 28: 1139–47. doi:10.1109/ICASSP.2013.6639346.

Talebi, V, and C L Baker. 2012. “Natural versus Synthetic Stimuli for Estimating Receptive Field Models: A Comparison of Predictive Robustness.” *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 32 (5): 1560–76. doi:10.1523/JNEUROSCI.4661-12.2012.

Theunissen, F E, S V David, N C Singh, A Hsu, W E Vinje, and J L Gallant. 2001. "Estimating Spatio-Temporal Receptive Fields of Auditory and Visual Neurons from Their Responses to Natural Stimuli." *... in Neural Systems* 12 (3). Informa UK Ltd UK : 289–316.

Tibshirani, R. 1996. "Regression Shrinkage and Selection Via the Lasso." *J R Stat Soc, Ser B* 58: 267–88.

Tieleman, T, and G E Hinton. 2012. "Coursera: Neural Networks for Machine Learning."

Touryan, J, G Felsen, and Y Dan. 2005. "Spatial Structure of Complex Cell Receptive Fields Measured with Natural Images." *Neuron* 45 (5): 781–91. doi:10.1016/j.neuron.2005.01.029.

Tsang, I W-H, J T-Y Kwok, and J M Zurada. 2006. "Generalized Core Vector Machines." *IEEE Transactions on Neural Networks / a Publication of the IEEE Neural Networks Council* 17 (5). Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. ivor@cs.ust.hk: 1126–40.

Tukey, J W. 1958. "Bias and Confidence in Not-Quite Large Samples." *The Annals of Mathematical Statistics* 29: 614.

Wager, S, S Wang, and P Liang. 2013. "Dropout Training as Adaptive Regularization." *Advances in Neural Information ...*, 1–11.

Waibel, A, T Hanazawa, G E Hinton, K Shikano, and K J Lang. 1989. "Phoneme Recognition Using Time-Delay Neural Networks." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37 (3): 328–39.

Wan, L, M D Zeiler, S Zhang, Y L Cun, and R Fergus. 2013. "Regularization of Neural Networks Using DropConnect." *Proceedings of the ...*, 1058–66.

Wilcox, Rand R, and H J Keselman. 2003. "Modern Robust Data Analysis Methods: Measures of Central Tendency." *Psychological Methods* 8: 254–74. doi:10.1037/1082-989X.8.3.254.

Willmore, B D B, R J Prenger, and J L Gallant. 2010. "Neural Representation of Natural Images in Visual Area V2." *The Journal of Neuroscience : The Official Journal of the Society for Neuroscience* 30 (6): 2102–14. doi:10.1523/JNEUROSCI.4099-09.2010.

Wray, J, and G G R Green. 1994. “Calculation of the Volterra Kernels of Nonlinear Dynamic Systems Using an Artificial Neural Network.” *Biological Cybernetics*. doi:10.1007/BF00202758.

Yuan, M, and Y Lin. 2006. “Model Selection and Estimation in Regression with Grouped Variables.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68 (1): 49–67. doi:10.1111/j.1467-9868.2005.00532.x.

Zeiler, M D. 2012. “ADADELTA: An Adaptive Learning Rate Method,” December.

Zeki, S M. 1973. “Colour Coding in Rhesus Monkey Prestriate Cortex.” *Brain Research* 53 (2): 422–27.

Zhang, T, and B Yu. 2005. “JSTOR: The Annals of Statistics, Vol. 33, No. 4 (Aug., 2005), Pp. 1538-1579.” *Annals of Statistics*.

Zou, H, and T Hastie. 2005. “Regularization and Variable Selection via the Elastic Net.” *Journal of the Royal Statistical Society. Series B: Statistical Methodology* 67: 301–20. doi:10.1111/j.1467-9868.2005.00503.x.