# Interpretable Machine Learning

# and Sparse Coding for Computer Vision

by

Will Landecker

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Melanie Mitchell, Chair
James Hook
Garrett Kenyon
Feng Liu
Thomas Shrimpton
Martin Zwick

Portland State University
2014

# Abstract

Machine learning offers many powerful tools for prediction. One of these tools, the binary classifier, is often considered a black box. Although its predictions may be accurate, we might never know *why* the classifier made a particular prediction. In the first half of this dissertation, I review the state of the art of *interpretable* methods (methods for explaining *why*); after noting where the existing methods fall short, I propose a new method for a particular type of black box called *additive networks.* I offer a proof of trustworthiness for this new method (meaning a proof that my method does not "make up" the logic of the black box when generating an explanation), and verify that its explanations are sound empirically.

Sparse coding is part of a family of methods that are believed, by many researchers, to *not* be black boxes. In the second half of this dissertation, I review sparse coding and its application to the binary classifier. Despite the fact that the goal of sparse coding is to *reconstruct* data (an entirely different goal than classification), many researchers note that it improves classification accuracy. I investigate this phenomenon, challenging a common assumption in the literature. I show empirically that sparse reconstruction is *not* necessarily the right intermediate goal, when our ultimate goal is classification. Along the way, I introduce a new sparse coding algorithm that outperforms competing, state-of-the-art algorithms for a variety of important tasks.

*To the cow*
*and the silo.*

# Acknowledgements

First and foremost, I am thankful to have had Melanie Mitchell as my advisor. Melanie gave me the creative freedom to explore a wide variety of interesting research topics (which can be seen in this dissertation's breadth) as well as the guidance I needed to bring the research to fruition (which can be seen in this dissertation's completion). I am grateful for the years of academic teamwork and friendship that I enjoyed with Melanie's other students, particularly Mick Thomure and Max Quinn. My research has also been guided and improved by a series of thoughtful, generous, and accomplished collaborators, including Garrett Kenyon, Steven Brumby, Luís Bettencourt, Thomas Shrimpton, Simon DeDeo, and Rick Chartrand.

My research would never have even started — much less resulted in a dissertation — without the early guidance of my parents, who nurtured and encouraged my interests before I even knew what they were. Throughout graduate school I benefited from close friendships and a strong community; there are too many people to name, but I appreciate their love, encouragement, and understanding nonetheless. I am especially thankful to have Hannah and Patrick in my life, whose presence has helped me stay balanced in even the most strenuous stretches of graduate school. I could not ask for a more supportive and loving partner than Caitlin. Caitlin, thank you for sharing your brilliance, your kindness, and your laughter. You inspire me to be not just a better researcher, thinker, and communicator, but a better person. Finally, I am grateful for the companionship of my dog Doogan, who looks good no matter how poorly he is reconstructed in Figure 11.2.

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction and Background

# Introduction

Machine learning is the study of algorithms that attempt to learn patterns from data in order to predict something useful about yet-unseen data. In the *binary classification* problem, the algorithm predicts one bit of information about each datum. Informally, the algorithm attempts to answer one yes-or-no question about each datum, after having seen previous data. For example, previous weather patterns might be analyzed in order to decide whether or not it will rain tomorrow; similarly, a machine-learning algorithm might use previous stock-trading data in order to decide whether or not a particular stock ought to be sold today. As a final example, a computer-vision system might learn from pictures with and without faces in order to predict whether or not a new image contains a face, as in Figure 1.

In the first half of my dissertation, I will focus on explaining the predictions made by one popular family of such algorithms. I consider a family of networks whose predictions are often very accurate on difficult machine-learning tasks; however, it is difficult to determine *why* a network makes a certain prediction. In Figure 1, for example, we might like to ask the black box prediction algorithm (the *classifier*) which aspects of a test image causes it to give one classification or the other (*face* or *no-face*).

After reviewing the state of the art of *interpretable* machine learning (methods

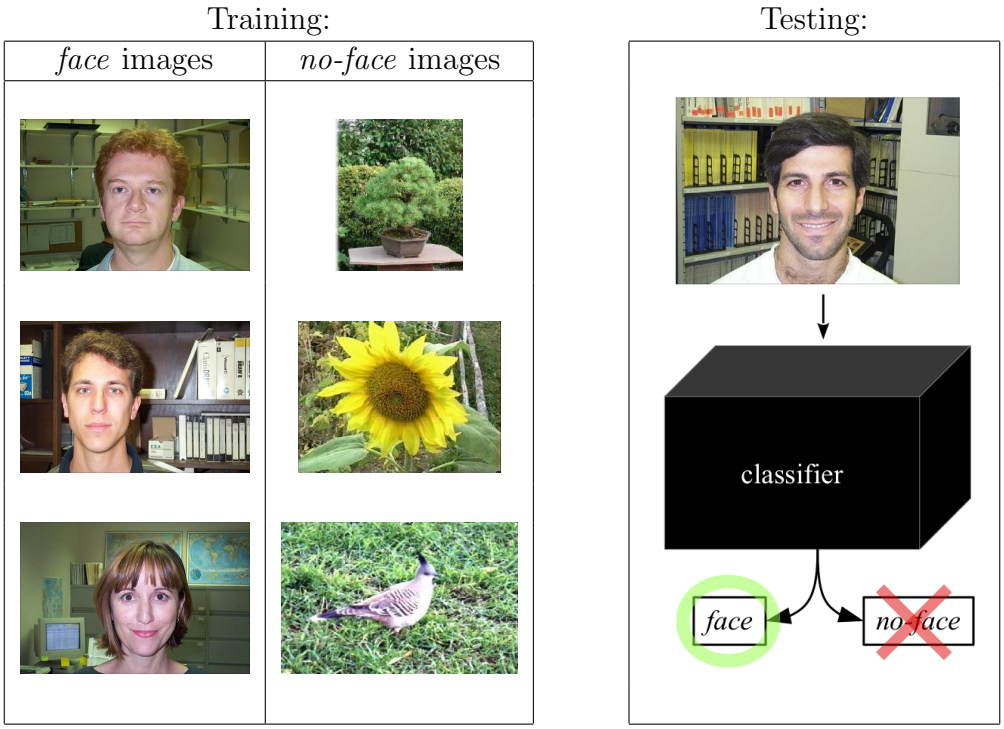



Figure 1: (All figures in this dissertation are best viewed in color.) A simplified illustration of a binary-classification task. First, a learning algorithm is given access to "training" data (left), in which each datum is associated with one of two classes (*face* or *no-face*). The learning algorithm trains a classifier, which attempts to predict the classes of yet-unseen "test" data (right). Example images are taken from the Caltech101 dataset (Fei-Fei et al., 2004).

that attempt to explain which aspects of the datum led to the classification), I will derive an explanation method that can answer these types of questions for a popular family of classifiers called hierarchical networks. Under simplifying assumptions, I will prove that the explanations are *trustworthy*, meaning that the explanation is faithful to the logic behind each decision made by the network. I will empirically test the trustworthiness of my explanation method with a variety of synthetic and natural datasets. I hypothesize that, for some modern datasets, an hierarchical network can achieve high accuracy by exploiting unintended artifacts in the data. For example, even though the data in Figure 1 was gathered in order to train a classifier to detect faces, the same data could be just as easily classified by detecting indoor photographs (note that in the figure, all *face* photos were taken indoors, and all *no-face* photos are outdoors). My hypothesis is that similar unintended strategies exist in benchmark datasets widely used in the machine-learning community, and that hierarchical networks will sometimes classify data with high accuracy while using such an unintended strategy.

Some researchers point to *generative models* as a panacea for the black boxes of machine learning. One of the most widely used generative models is *sparse coding*, in which an algorithm learns how to "reconstruct" the data before learning how to classify it. For example, in Figure 2, a sparse-coding algorithm first learns how to encode and decode data with as little loss as possible (similar to compression). Next, the encoded versions of the data are passed to a classifier for training and testing. Researchers often view sparse coding as an *interpretable* method (as opposed to a black box) because we can manually inspect the reconstructions to verify that the algorithm is encoding information from the relevant parts of the image.

In addition to this interpretable aspect, sparse coding is favored by many re-

searchers because it often increases classification accuracy compared to using the raw, original data. The standard justification for this increase in accuracy comes in two parts: (1) being able to reconstruct the data from a sparse encoding means that the encoding has captured the relevant information, and (2) sparsity (which will be discussed in detail below) is an effective form of regularization[1], which helps increase the classifier's ability to generalize to new data. In the second half of my dissertation, I investigate and challenge this standard justification for the increase in classification accuracy that we often observe with sparse coding. Along the way, I develop a new, highly efficient sparse coding algorithm that achieves state-of-the-art reconstruction performance for a variety of tasks.

In the remainder of Part I, I review the notation, definitions, and relevant background of machine learning that I will refer to later in the dissertation. In Part II, I develop the notion of interpretable classifiers, and I propose and evaluate a novel method for understanding classifications. In Part III, I tease apart two different goals of sparse coding (to reconstruct the data with sparsity, and to increase classification accuracy). I describe a new sparse coding algorithm that performs well on difficult reconstruction tasks in Part IV. In Part V, I review the key points of this dissertation, and I propose directions for future work.

The main results and contributions of this dissertation are the following:

- I develop novel algorithm for explaining individual classifications of hierarchical networks (Section 4.2).

- Using this algorithm, I demonstrate that several popular image datasets (which

[1]Regularization is a common approach to preventing a model from becoming too complex, in hopes of capturing only the aspects of the training data that will generalize well to the test data. Regularization will be discussed more in Part III.

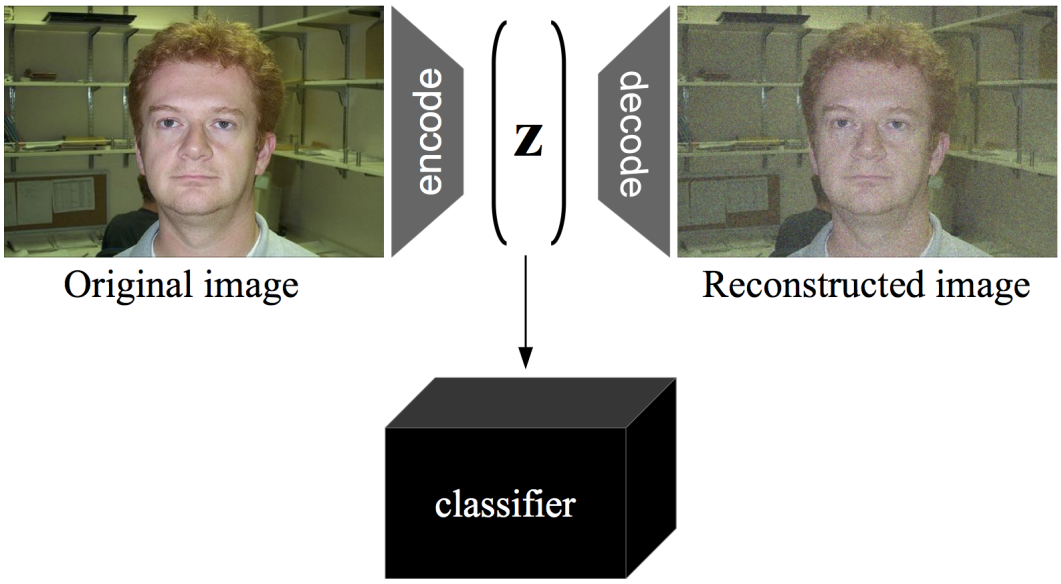



Figure 2: In sparse coding, each datum is encoded into a sparse vector (meaning most of its entries are zero). This vector ($\mathbf{z}$) is chosen such that it reconstructs the original datum well. After such a vector is found, it is sent to the classifier for training or testing. Face image from Caltech101 (Fei-Fei et al., 2004)

are often used for benchmarking computer-vision systems) contain spurious cues in the *backgrounds* of the images that can "give away" the identity of the object in the foreground (Section 5.3.3).

- I provide evidence that the sparse reconstruction performance of sparse coding algorithms is not necessarily tied to the classification difficulty of the dataset (Section 8.2).

- Although sparse coding can increase classification performance, I show that the reason for this increase is not well understood. In particular, I show that there exist sparser codes that reconstruct the data better, but which achieve worse classification accuracy than a competing set of sparse codes (Section 8.3).

- I develop a new sparse-coding algorithm that outperforms state-of-the-art methods when sparsity is low and noise is moderate (Chapter 10).

These results and contributions also appear in Landecker et al. (2013) and Landecker et al. (2014).

# Chapter 1

# Notation and Definitions: Datasets, Classifiers, Features

In this section, I introduce the mathematical notation that will be used in the rest of this dissertation.

Recall the binary classification task. Informally, a binary classification task starts with a dataset (sometimes called *the data*) and a yes-or-no question. For example, the data might be a collection of images, and the question might be "does a given image contain a face?" The task is to analyze the data in such a way that the question can now be answered (as accurately as possible) for yet-unseen data.

More formally, let a *dataset* $\mathcal{X} \subseteq \mathbb{R}^n$ be a set of real vectors of fixed length[1]. An element of a dataset is called a datum, and is denoted $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. I refer to $x_i$ as the $i^{\text{th}}$ *dimension* of $\mathbf{x}$. For much of this dissertation, $\mathbf{x}$ will be an image[2], in which case $x_i$ may be called a *pixel*. When discussing *regions* of $\mathbf{x}$, I mean

[1] I will relax these constraints shortly.

[2] We form a vector from an image by reading the intensities (*i.e.,* the grayscale values) or colors of pixels across the image's rows.

some collection of pixels near each other in an image. When it becomes important to enumerate the data in a dataset, I may write either $\mathbf{x}^i$ or $\mathbf{x}_i$ to indicate the $i^{\text{th}}$ datum in the dataset. Any confusion between indices that enumerate dimensions of a datum and indices that enumerate data in a dataset will be addressed in the text.

Let the *classes* (or *labels*) be a set $\mathcal{Y}$ with two elements. Informally, the elements of $\mathcal{Y}$ are the possible answers to the yes-or-no question we would like to ask about our data. Thus one may choose, for example, $\mathcal{Y} = \{\textit{face}, \textit{no-face}\}$, or $\mathcal{Y} = \{\textit{rain}, \textit{no-rain}\}$. Generally, one formally defines $\mathcal{Y} = \{1, -1\}$ and interprets these numerical values after the analysis[3].

A *labeled dataset* is a set of pairs $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots\}$ where for each $i$, we have $\mathbf{x}_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$, and moreover $y_i$ is the correct class for $\mathbf{x}_i$. Thus a labeled dataset is a subset of $\mathcal{X} \times \mathcal{Y}$.

A *classifier* is a function

$$\hat{y} : \mathcal{X} \to \mathcal{Y}$$

which tries to correctly predict $y_i$ from $\mathbf{x}_i$. Informally, $\hat{y}$ tries to answer the yes-or-no question about new data. The notation $\hat{y}$ is meant to suggest that $\hat{y}(\mathbf{x}_i)$ is an estimate (or an "informed guess") of $y_i$. One usually considers $\hat{y}$ to be parameterized by a set of numerical values. For example, these parameters might be the coefficients of a function computed by $\hat{y}$. Let $\theta$ be an assignment of the parameters, and let $\Theta$ be the set of all possible parameter assignments. I denote the classifier with parameters $\theta$ by $\hat{y}_\theta(\cdot)$.

[3]Some texts define $\mathcal{Y} = \{1, 0\}$. This only changes the equations slightly, and does not represent a serious difference in methodology from what is developed here.

A learning algorithm (which I treat as a function) has the signature

$$\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^+ \to \Theta.$$

In particular, a learning algorithm takes a labeled dataset (with some positive number of labeled data) as input, and returns the parameters $\theta$ that will later be used for classification.

Recall my earlier summary of the binary classification task: to analyze data in order to answer a yes-or-no question about yet-unseen data. Note that this is a somewhat ill-posed problem: if $\mathbf{x}$ has never been seen before, how can we know that $\hat{y}_\theta(\cdot)$ will correctly calculate $y$? In general one cannot be sure that it will, but one might hope that $\hat{y}_\theta(\cdot)$ does well on unseen data if it also did well on the data that it *has* seen. That is to say, one might choose a learning algorithm $\mathcal{L}$ which chooses the parameters $\theta^*$ such that $\hat{y}_{\theta^*}(\mathbf{x}_i) = y_i$ for all $i$ (or for as many $i$ as possible).

Given a datum $\mathbf{x}$, calculating $\hat{y}(\mathbf{x})$ is called *classifying* the data. The proportion of correct classifications of a data set is called the classifier's *accuracy* over that set. The accuracy over the training set is called the *training accuracy*. The unseen data is often called the *test data*, and the accuracy over this set is called the *test accuracy*. Thus I can rephrase the earlier strategy by stating that one might expect the test accuracy to be high if the training accuracy is high. This is called the *inductive learning hypothesis* (Mitchell, 1997).

To solve a new binary classification task, a researcher wishes to find two algorithms: the learning algorithm and the classifier. I will review, at a very high level, some typical classifiers and their associated learning algorithms in Sections 2.1 and 2.2.

It should be noted that there are many machine-learning tasks that do not fall under the umbrella of binary classification, such as clustering, regression, and non-binary classification (Mitchell, 1997). In this dissertation, however, I will focus primarily on binary classification for two reasons. Firstly, binary classification is relatively well-defined and simple in concept. Secondly, binary classification is extremely useful to researchers concerned with prediction. Note that in Part IV of this dissertation, I will shift the discussion to a task more closely related to regression than binary classification.

I conclude this section with a few mathematical definitions that will be used throughout the dissertation. Let $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^n$. The $\ell^p$ norm of $\mathbf{x}$, denoted $\|\mathbf{x}\|_p$, is defined by

$$\|\mathbf{x}\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}.$$

The dot-product, or standard inner product, of $\mathbf{x}$ and $\mathbf{x}'$ is denoted $\langle \mathbf{x}, \mathbf{x}' \rangle$, and is defined by

$$\langle \mathbf{x}, \mathbf{x}' \rangle = \sum_{i=1}^{n} x_i x_i'$$

The cardinality of a set $\mathcal{S}$ is denoted $|\mathcal{S}|$, and is equal to the number of elements in the set. A set $\mathcal{S}$ is *convex* if, given any $\mathbf{s}$ and $\mathbf{s}' \in \mathcal{S}$, we have $t\mathbf{s} + (1-t)\mathbf{s}' \in \mathcal{S}$ for all $0 \leq t \leq 1$. A function $f$ is convex if

$$f(tx + (1-t)x') \leq tf(x) + (1-t)f(x') \quad \text{for all } 0 \leq t \leq 1.$$

The composition of two functions is denoted $f \circ g(x) = f(g(x))$.

## 1.1 Classifier Confidence

Although a binary classifier is a mapping from the data to the discrete set $\mathcal{Y} = \{-1, +1\}$, it is often constructed from a *real-valued* function $c_\theta$, where

$$c_\theta : \mathcal{X} \to \mathbb{R}.$$

One then defines

$$\hat{y}_\theta(\mathbf{x}) = \text{sgn}[c_\theta(\mathbf{x})]$$

where

$$\text{sgn}[x] = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

The function $c_\theta$ is called the *scoring function*, and $c_\theta(\mathbf{x})$ the *score* of the classifier on $\mathbf{x}$. The subset of data for which $y = 1$ are called the *positive* data, and likewise the data for which $y = -1$ are called the *negative* data. The learning algorithm, then, attempts to find parameters $\theta$ for which $c_\theta(\mathbf{x}) \geq 0$ for all (or many) positive data, and $c_\theta(\mathbf{x}) < 0$ for all (or many) negative data[4].

The value $|c_\theta(\mathbf{x})|$ is considered to be the *confidence* of the classifier (Dredze et al., 2008). That is to say, if

$$c_\theta(\mathbf{x}_1) > c_\theta(\mathbf{x}_2) > 0$$

then the classifier will classify both $\mathbf{x}_1$ and $\mathbf{x}_2$ as positive, but it is more confident in its classification of $\mathbf{x}_1$.

In a slight abuse of notation, I will sometimes refer to $c_\theta$ as the "classifier", even

[4]There are often additional criteria required by the learning algorithm. For example, I will review the maximum-margin constraint in Section 2.1.

though I have defined this term to be the thresholded function $\hat{y}_\theta$. This abuse is common in the literature, and should not be confusing: when discussing the discrete set $\mathcal{Y}$ I mean $\hat{y}_\theta$, and when discussing the real-valued output (the confidence) I mean $c_\theta$.

## 1.2 Feature Extraction

So far I have assumed that the data $\mathcal{X}$ is a subset of $\mathbb{R}^n$. However, real-world data does not necessarily come in real vectors of a fixed dimensionality. For instance, a researcher might be asked to classify non-vector objects such as graphs or strings of text. Even when the data are real vectors, the vectors are often of variable length.

For a classifier and learning algorithms to succeed, then, one requires a mapping from the (possibly non-uniform length, possibly non-vector) real-world data to $\mathbb{R}^n$. This mapping is called *feature extraction*. Formally, given a set $\mathcal{X}$ of (real-world) data, feature extraction is a function

$$g : \mathcal{X} \to \mathbb{R}^n$$

that transforms the data into an appropriate format for learning and classification. The output $\mathbf{z}_i = g(\mathbf{x}_i)$ is a vector called the *features* extracted from datum $\mathbf{x}_i$. As an example, when the data are strings of text, the features might be word counts or word frequency for $n$ important words. Features extracted from graphs might include the number of nodes that have $k$ edges for $1 \leq k \leq n$.

The performance of a classifier can depend heavily on the choice of features extracted from the data. Choosing high-performing features often relies heavily on

domain expertise. This extensive hand-tuning can be disappointing for machine-learning researchers who wish to develop methods of automatically learning from data.

In fact, the hand-tuning of features is often beneficial even when the data *are* real vectors of fixed length. That is to say, the accuracy of a machine-learning system can be improved when each datum is transformed in some way that provides task-relevant information to the classifier. In Sections 2.4, 2.5, 5.3 and 7, I will review several common types of feature extraction that are relevant to this dissertation.

# Chapter 2

# Machine Learning Background and Examples

In this Section, I review some common examples of classifiers and features that will be useful later in the document. For notational simplicity, I will use the notation of classifiers $\hat{y}(\cdot)$ that take data $\mathbf{x}$ directly, rather than features $\mathbf{z}$. However, the following classifiers are equally capable of training and testing with feature vectors rather than raw data.

## 2.1 Support Vector Machines

While I have reviewed the abstract definitions of learning algorithms and classifiers, I have not yet given a specific example. In this section, I review the popular Support Vector Machine (SVM) classifier at a high level (Cortes and Vapnik, 1995). The purpose of this review is both to provide a concrete example of a classifier and to prepare the reader for the discussion of SVMs later in this dissertation.

In order to simplify the presentation, let us assume that the data are *linearly separable*. That is to say, the $n$-dimensional data can be separated by an $(n-1)$-dimensional hyperplane such that the positive data lie on one side of the hyperplane and the negative data lie on the other. An example is shown in Figure 2.1.

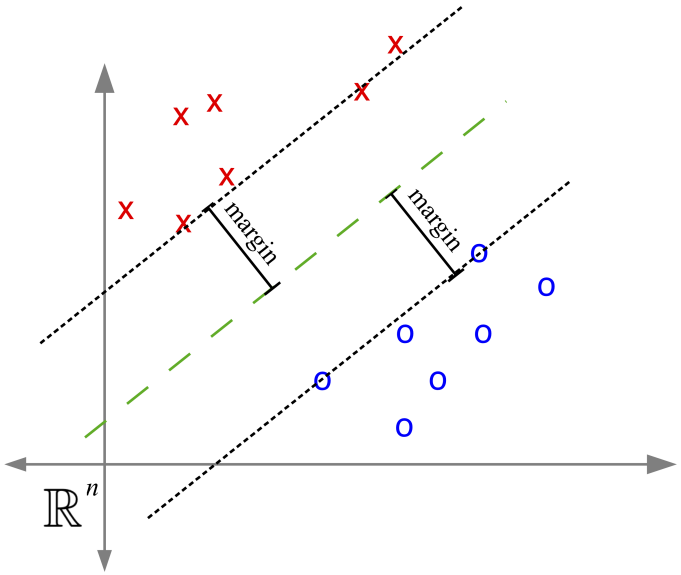



Figure 2.1: Example training data and a linear support vector machine (SVM). The positive data are represented by red x's, and the negative by blue o's. The SVM attempts to learn the separating hyperplane (green, dashed) that maximizes the margin (the distance between the hyperplane and the data).

A linear classifier is of the form

$$\hat{y}_\theta(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$$

where $\theta$ specifies the choice of vector $\mathbf{w}$, "bias" $b \in \mathbb{R}$, and $\langle \cdot, \cdot \rangle$ is the standard dot product. The separating hyperplane (also called the *decision surface*) is defined by the function $\langle \mathbf{w}, \mathbf{x} \rangle + b = 0$.

Clearly, there can be many values for $\theta$ which will separate the data. The goal of the SVM learning algorithm is to choose $\theta^* = (\mathbf{w}^*, b^*)$ that maximizes the distance between the separating hyperplane and the data. This distance is known as the *margin*, and is illustrated in Figure 2.1. Boser et al. (1992) proved that this method produces an optimal classifier for generalizing to (unseen) test data, under certain assumptions of the data.

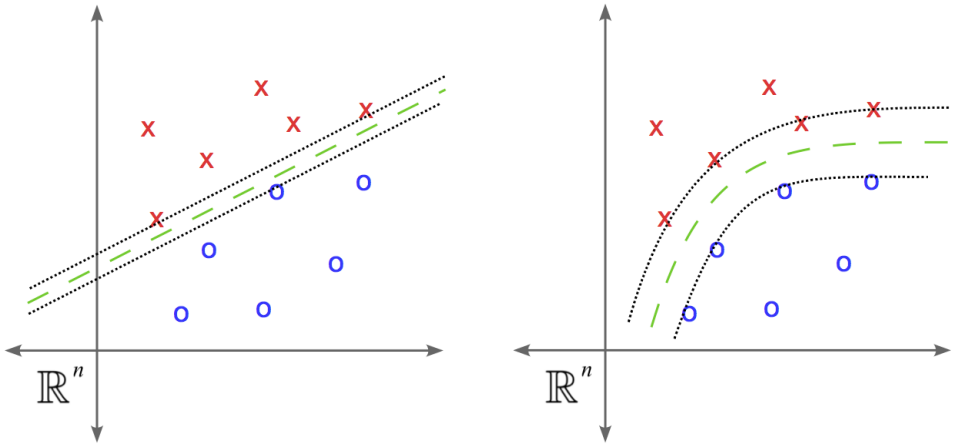
It turns out that, without loss of generality, the margin is exactly $\|\mathbf{w}\|_2$. Therefore, one can find $\mathbf{w}^*$ and $b^*$ by solving the constrained optimization problem

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & \frac{1}{\|\mathbf{w}\|_2} \\ \text{subject to} \quad & 1 - y_i \left( \langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) < 0 \quad \text{for } 1 \leq i \leq N \end{aligned} \tag{2.1}$$

where $N = |\mathcal{X}|$. In the above formulation, $1 - y_i \left( \langle \mathbf{w}, \mathbf{x}_i \rangle + b \right) < 0$ is merely the constraint that each training datum be correctly classified. The constrained minimization problem of (2.1) is known as the *primal* of the SVM (Cortes and Vapnik, 1995).

Training the SVM involves solving this constrained optimization problem in order to find parameters $\mathbf{w}^*$ and $b^*$. Problem (2.1) can be efficiently solved with a variety of linearly-constrained quadratic programming techniques. One of the most popular such methods is the Sequential Minimal Optimization algorithm of Platt (1999). Once the optimal parameters are found, new data are classified with the function

$$\hat{y}_\theta(\mathbf{x}) = \text{sgn}(\langle \mathbf{w}^*, \mathbf{x} \rangle + b^*). \tag{2.2}$$




(A) (B)

Figure 2.2: Linear versus nonlinear (kernel) SVMs. (A) When a linear SVM is fit to the data, the maximum margin is small. (B) When a nonlinear SVM is fit to the same data, the margin can be much larger.

## 2.2 The Kernel Trick

While SVMs offer a well-motivated theory for how to achieve good generalization performance (*i.e.,* choose the hyperplane that maximizes the margin between the two classes), the above examples come with the strict constraint of a linear hyperplane separating the classes. In some cases, however, one might have some *a priori* knowledge that the data is best classified by a nonlinear classifier. The kernel trick allows the same maximum-margin strategy to be applied to this nonlinear case (Shawe-Taylor and Cristianini, 2004).

The kernel trick extends many machine-learning methods (including SVMs) by replacing the dot product $\langle \cdot, \cdot \rangle$ with a new function $\kappa(\cdot, \cdot)$ (called the *kernel function*), specifically tailored to help solve the task at hand. For example, one might choose

the polynomial kernel of degree $d$ with constant $c$, defined by

$$\kappa(\mathbf{x}', \mathbf{x}) = (\langle \mathbf{x}', \mathbf{x} \rangle + c)^d,$$

or the radial basis function (RBF) kernel with scaling parameter $\sigma$, defined by

$$\kappa(\mathbf{x}', \mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}' - \mathbf{x}\|_2^2}{2\sigma^2}\right).$$

While there are mathematical restrictions on the form of $\kappa(\cdot, \cdot)$, I do not detail them here. I refer the interested reader to the lecture notes by Ng (2014, chapter 8). For the purpose of this dissertation, the reader needs only a broad overview of the kernel trick's application to SVMs, after which I will dive deeper into an example kernel called Spatial Pyramid Matching (SPM) which will be used in Section 8.

For SVMs, the kernel trick is enabled by the realization that the vector $\mathbf{w}$ defining the maximum-margin hyperplane lies in the subspace spanned by the training data,

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i. \tag{2.3}$$

for some $\alpha_i \in \mathbb{R}$. (Recall that $n$ is the dimensionality of each $\mathbf{x}_i$.)

This fact allows us to rewrite Equation (2.2) as

$$\begin{aligned} \hat{y}_\theta(\mathbf{x}) &= \operatorname{sgn}\left(\sum_{i=1}^{n} \alpha_i \langle \mathbf{x}_i, \mathbf{x} \rangle + b\right) \\ &= \operatorname{sgn}\left(\sum_{s \in \mathcal{S}} \alpha_s \langle \mathbf{x}_s, \mathbf{x} \rangle + b\right) \end{aligned} \tag{2.4}$$

where $\mathcal{S}$ is the set of indices $i$ for which $\alpha_i \neq 0$, and the set $\{\mathbf{x}_s\}$ are known as the

*support vectors*. Interestingly, it is often the case that $|\mathcal{S}| \ll n$ — that is to say, trained SVMs often have very few support vectors.

Equation (2.4) is known as the *dual* formulation of the SVM classifier (Cortes and Vapnik, 1995). Applying the kernel trick gives us

$$\hat{y}_\theta(\mathbf{x}) = \operatorname{sgn}\left(\sum_{s\in\mathcal{S}} \alpha_s \kappa(\mathbf{x}_s, \mathbf{x}) + b\right), \tag{2.5}$$

Thus training a kernel-SVM involves finding the coefficients $\alpha_s$ that lead to the maximum-margin hyperplane in the space "induced" by the kernel — though this is more detail than necessary for this dissertation.

## 2.3 The Spatial Pyramid Matching Kernel

In Part III of this dissertation, I will use the Spatial Pyramid Matching (SPM) kernel to classify images. Recall that a kernel works by replacing the dot product; thus a kernel yields a new way to compare two pieces of data. The SPM kernel was developed by Lazebnik et al. (2006) as a way to compare the features extracted from two images. I describe the SPM kernel here.

First, let us review the idea of *histogram intersections*. A histogram is a vector (or list of numbers) $\mathbf{h} = [h_1, h_2, \ldots, h_n]$ where each entry $h_i$ in the list represents the number of elements that $\mathbf{h}$ has counted in the $i^{\text{th}}$ "bin." One can generate a new histogram based on the intersection of the first two, $\mathbf{g} = \mathbf{h} \cap \mathbf{h}'$. This new histogram is defined by

$$g_i \stackrel{\texttt{def}}{=} \min(h_i, h'_i).$$

The simple process of generating $\mathbf{g}$ is illustrated in Figure 2.3. One way to measure

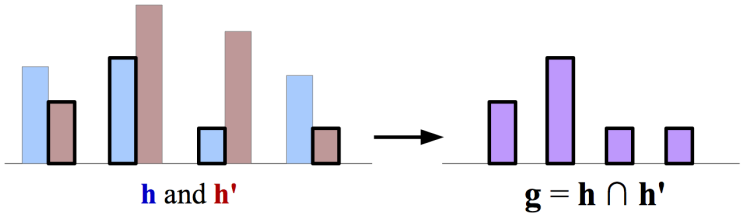



Figure 2.3: Histogram intersection illustrated. Given two 4-dimensional histograms, $\mathbf{h}$ (blue) and $\mathbf{h}'$ (red), histogram intersection yields a new histogram $\mathbf{g}$ (purple), formed by taking the minimum of each entry in $\mathbf{h}$ and $\mathbf{h}'$, $g_i = \min(h_i, h'_i)$. For each $i$, the minimum between $h_i$ and $h'_i$ is outlined in black. All figures in this dissertation are best viewed in color.

the intersection of two histograms with a function $\mathcal{I}$ is to merely sums the dimensions of the intersection,

$$\mathcal{I}(\mathbf{h}, \mathbf{h}') = \sum_{i=1}^{n} \min(h_i, h'_i)$$

The SPM kernel generalizes this type of measure to the case where we want to compare two different sets of points which have not yet been aggregated into histograms. Let $X \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^n$ be two finite sets. Let $c$ be the smallest $n$-dimensional hypercube containing all the elements of both $X$ and $O$. At level $\ell$, SPM "dices" the volume $c$ into $(2^\ell)^n$ equal-sized bins, as demonstrated in Figure 2.4 with $n = 2$ and $\ell = 0, 1, 2$. This results in the histograms $X^\ell$ and $O^\ell$, which count the number of elements of $X$ ($O$, respectively) in each of the bins. The measure computed by the kernel at layer $\ell$ is,

$$\mathcal{I}^\ell(X, O) = \mathcal{I}(X^\ell, O^\ell)$$

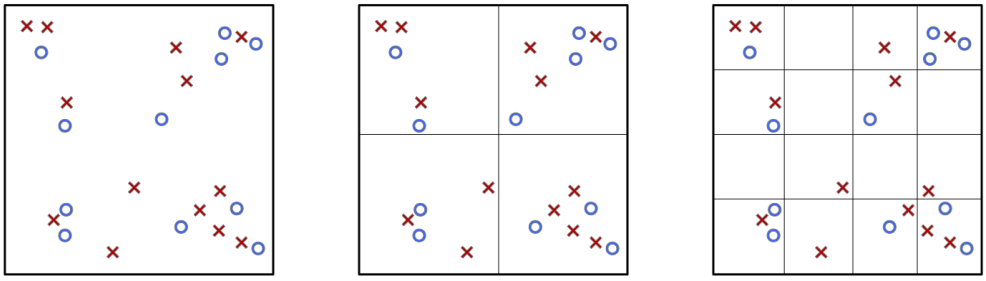



Figure 2.4: The Spatial Pyramid Matching (SPM) kernel generalizes histogram intersection to sets of points in high-dimensional spaces using multiple scales. At level $\ell$, the kernel divides the space into a $2^\ell \times 2^\ell$ grid (in higher-dimensional spaces, an $2^\ell \times 2^\ell \times \ldots \times 2^\ell$-grid), and each bin in the grid is scored. A bin's score is the minimum number of elements from each set (either x or o) in that bin.

The SPM kernel measures the similarity between the sets $X$ and $O$ by computing

$$\kappa_{\mathsf{SPM}}(X, O) \stackrel{\mathsf{def}}{=} \frac{1}{2^L}\mathcal{I}^0(X, O) + \sum_{\ell=1}^{L} \frac{1}{2^{L-\ell-1}}\mathcal{I}^\ell(X, O). \tag{2.6}$$

where $L$ is the number of "levels" in the pyramid. Higher values of $L$ mean that $X$ and $O$ are compared at higher levels of granularity.

Note that larger values of $\ell$ are scored higher by $\kappa_{\mathsf{SPM}}$. This is because the bins are smaller when $\ell$ is larger (see Figure 2.4, right), and thus $\mathcal{I}^\ell$ measures similarity at a finer scale. That is to say, it is less likely for $X$ and $O$ to have points near each other by chance when $\ell$ is larger. In practice, we choose $L = 3$.

I have introduced how an SVM uses a kernel $\kappa(\cdot, \cdot)$ to define a measure of similarity between two pieces of data, $\mathbf{x}$ and $\mathbf{x}'$. I have also discussed how $\kappa_{\mathsf{SPM}}(\cdot, \cdot)$ allows the SVM to measure the similarity of two sets of points, $X$ and $O$. In order to apply $\kappa_{\mathsf{SPM}}$ in an SVM, then, we need a mapping from a datum $\mathbf{x}$ to a set of points $X$. In the next section, I describe a method which does exactly that.

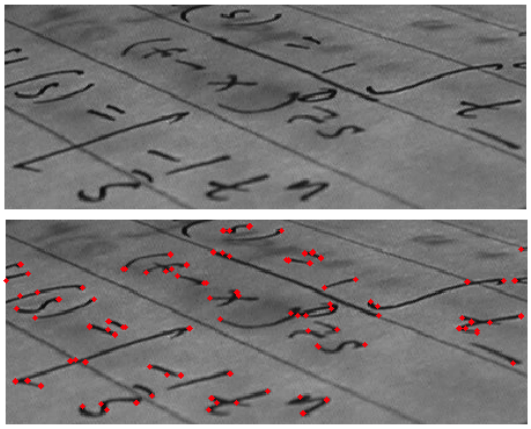


Figure 2.5: An image (left) and the key points extracted by SIFT (right, in red). Images from Wikipedia (2014)

## 2.4 SIFT

This dissertation includes many examples of image classification. Classifying images with good accuracy often relies on good feature extraction methods, as discussed in Section 1.2. In Part III of this dissertation, I will use a feature extraction method called SIFT (Scale Invariant Feature Transform). SIFT is a popular type of feature for classification tasks in computer vision (Lowe, 1999). SIFT features indicate the locations of *key points* in an image. The SIFT features of a single image is a set containing many such key points. These key points are designed to be invariant to rotation, scale, and shift of objects in that image. For example, in Figure 2.5, we see an image and its key points (in red).

There are many types of features extracted by SIFT. They include the gradient of pixel intensity values at multiple orientations in the image to detect high-contrast edges and corners, and differences-of-Gaussians (DoG) to detect small features such as salient dots. The key points are smoothed and various post-hoc analyses are

performed in order to weed out key points that might not be useful for image- or object-recognition.

## 2.5 Generative Models

Part II of this dissertation will focus on methods that explain, given a datum $\mathbf{x}$ and its classification $\hat{y}_\theta(\mathbf{x})$, how each dimension $x_i$ of $\mathbf{x}$ affected the classification. This mapping from the classification back to the datum being classified is reminiscent of a large class of statistical models called *generative models*. These models are capable of generating new data based on features that are learned from the training data. The generation of new data is sometimes referred to as the model "dreaming" (Hinton et al., 2006).

Formally, a generative model defines a distribution over both the input space and model space, $P(\mathbf{x}, \theta)$. With such a distribution, one may choose a particular model (which defines $P(\theta)$), and one can then sample from the distribution

$$P(\mathbf{x}|\theta) = \frac{P(\mathbf{x}, \theta)}{P(\theta)}.$$

Sampling from $P(\mathbf{x}|\theta)$ generates new data from the model itself. This is how a generative model "dreams" new data.

Having such a model generate data allows the user to verify that the model is capturing relevant statistics of the data. Thus generative models, such as deep belief networks (Hinton et al., 2006; Ranzato et al., 2011), deconvolutional networks (Zeiler et al., 2010, 2011) and various latent-variable probabilistic models (Fei-Fei et al., 2004) are often considered to be interpretable. In particular, sampling new data from a

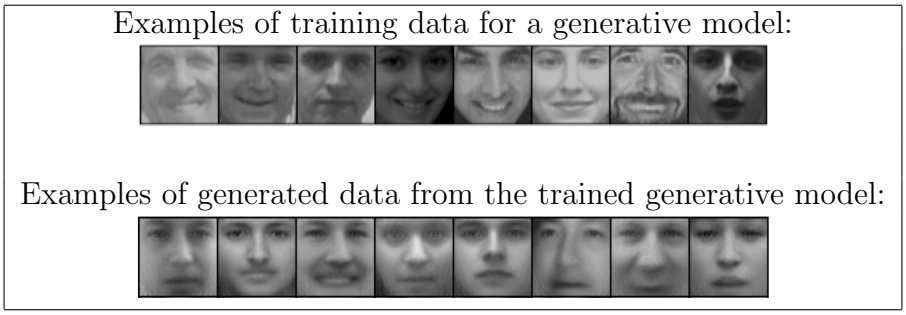



Figure 2.6: Training data and generated data from a deep belief network, a popular type of generative model (Ranzato et al., 2011). The second row of images were sampled from the model's learned distribution over images. The fact that these images are convincingly face-like is good evidence for the model having learned parameters $\theta$ that accurately describe faces.

learned distribution can help us develop some intuition about the learned distribution. In this way, generative models give us some understanding of the statistics learned by the model. While sampling allows for many important types of learning and feedback to the user, I will discuss in Section 3.1 why sampling does not suffice as an explanation method for the purposes of this dissertation.

It should be noted that generative models are not explicitly discriminative — that is to say, they are not necessarily classifiers in their own right. In the work considered in this dissertation, a generative model is typically used to extract features that are then fed to a classifier such as a linear SVM (Zeiler et al., 2010, 2011), much as was shown in Figure 2. In fact, the sparse coding methodology described in that figure is formally a type of generative model. I will discuss this in greater detail in Parts III and IV.

# Part II

# Explaining Classifications

The motivation for this part of the dissertation comes from the black-box nature of many classifiers. Recall the training and testing procedures outlined in Figure 1, in which the test image could be correctly classified as *face* for several different reasons: perhaps the classifier had learned to recognize the geometry of faces; or perhaps it learned to recognize bookshelves or grass. Because machine-learning is gaining use with "real-world" data (meaning datasets that researchers have not carefully manipulated or curated to control for any biases), the machine-learning practitioner is often left to wonder if a classifier's accuracy comes from having learned to solve the intended problem (recognizing faces) or from learning some spurious statistics that have accidentally "leaked through" in the dataset.

Understanding which problem the classifier is *truly* solving should not be considered an extraneous branch of machine learning; it is crucial for many real-world applications. Kononenko (2001) surveyed medical professionals, finding that machine-learning methods were untrusted — even methods that provided higher diagnostic accuracy than physicians — because the methods did not explain how they came to their decision. Poulin et al. (2006) found the same need for interpretable methods among (non-medical) biological researchers. Tickle et al. (1998) note how financial markets are another setting where each individual decision can be critical, and therefore the interpretability of classifiers is crucial for those interested in using machine learning in this domain.

So far, the term "interpretable" has been a vague notion. I will now develop the definition of interpretability in a more concrete way. Intuitively, a classification is interpretable if it comes with some explanation of *what caused* a given classification. More concretely, I desire a *per-instance* explanation method. That is to say, given a single datum $\mathbf{x}$ and a classifier $\hat{y}_\theta$, I want to know how each dimension $x_i$ of the

datum $\mathbf{x}$ affected the classification $\hat{y}_\theta(\mathbf{x})$. I believe that this type of explanation can add a degree of trustworthiness to the machine-learning tools that have not yet been widely adopted in medicine (Kononenko, 2001), financial analysis (Tickle et al., 1998), and other research areas where each individual classification is crucial (Poulin et al., 2006).

For example, in the context of machine-learning applications in medicine, imagine an algorithm trained to detect whether or not a patient has high risk for developing hypertension (the classes are *high risk* or *low risk*) based on features extracted from the patient's record. Even if an algorithm can predict such risk more accurately than physicians, Kononenko (2001) found in a survey that hospitals and healthcare administrators would *not* trust and employ such an algorithm in the absence of some explanation of the resulting predictions. In this context, a per-instance explanation might contain the information, "predict *high risk* for `patient_0589` mostly because `blood_pressure > 160-over-100` and `smoker = true`; additionally, but less importantly, because `age > 65`." Such an explanation could provide an amount of trustworthiness to the prediction, such that the machine-learning algorithms could find more real-world use.

This type of explanation method contrasts with other methods which provide an *aggregate* description of the classifier (Fu, 1994), but which do not explain how the classifier considers an individual datum. An aggregate description might tell us what general rules a classifier uses when detecting high risk (continuing the above example), such as "Look for high blood pressure, history of kidney disease, history of smoking, etc." However, this type of aggregate description does not tell us the reason behind an *individual* classification. While such aggregate descriptions are certainly useful, this dissertation will focus on explaining individual classifications.

Recall that a classifier $\hat{y}$ is a mapping from a datum (we again assume that the data $\mathcal{X} \subseteq \mathbb{R}^n$) to a finite set $\mathcal{Y}$ called the *classes*, parameterized by $\theta \in \Theta$:

$$\hat{y}_\theta : \mathbb{R}^n \to \mathcal{Y}.$$

Let us define a mapping $\mathsf{Explain}$, whose job is to explain a classifier's classification. In particular, $\mathsf{Explain}$ takes a datum $\mathbf{x}$ and a classifier $\hat{y}_\theta$, and returns a vector of the same dimensionality as the datum. Let $\tilde{\mathcal{Y}} = \{\ \hat{y}_\theta \mid \theta \in \Theta\ \}$. Then we have:

$$\mathsf{Explain} : \mathbb{R}^n \times \tilde{\mathcal{Y}} \to \mathbb{R}^n.$$

The output of $\mathsf{Explain}$ will be an explanation of how each dimension of the datum affected the classification. This definition is still fairly vague and nontechnical, and different researchers disagree on the best way to explain a classification (*i.e.,* the best definition of $\mathsf{Explain}$). I review several common definitions below.

# Chapter 3

# Explaining Classifications: Background

## 3.1 The Bayesian Approach to Explanations

In Bayesian machine learning (Bishop, 2006), a system learns parameters $\theta$ in order to calculate an explicit posterior distribution $P_\theta(\mathbf{x}|y)$[1]. This is the distribution of datum $\mathbf{x}$ (treated as a random variable) given the class $y$. A datum $\mathbf{x}$ is classified by

[1]Note that the subscript $\theta$ implies that the distribution is parameterized by the learned model $\theta$; this parameterized distribution is over $\mathbf{x}$ and $y$. For example, under a Gaussian assumption, one might have $\theta = \{(\mu_i, \sigma_i)\}_i$, where the choice of the class $y$ determines which index $i$ is used. Some works write this distribution instead as $P(\mathbf{x}|\theta, y)$.

determining the most likely class $y$ given the datum $\mathbf{x}$,

$$\hat{y}_\theta(\mathbf{x}) = \arg\max_y P_\theta(y|\mathbf{x}) \tag{3.1}$$

$$= \arg\max_y P_\theta(\mathbf{x}|y)P_\theta(y)\frac{1}{P_\theta(\mathbf{x})} \tag{3.2}$$

$$= \arg\max_y P_\theta(\mathbf{x}|y)P_\theta(y) \tag{3.3}$$

where Equation (3.2) comes from Bayes' Theorem, and Equation (3.3) follows from the fact that the prior probability of the data, $P_\theta(\mathbf{x})$, is not a function of $y$.

After classifying a datum as $\hat{y} = \hat{y}_\theta(\mathbf{x})$, a researcher might try to gain some understanding of the classifier by sampling from the distribution

$$P_\theta(\mathbf{X}|\hat{y}). \tag{3.4}$$

where $\mathbf{X}$ is a random variable (which can be sampled), and $\hat{y}$ is the predicted class of the given datum $\mathbf{x}$. In fact, this is exactly the "dreaming" ability of generative models described in Section 2.5. Thus one can "explain" the classification $\hat{y}_\theta(\mathbf{x})$ of a generative model by sampling from the posterior of that class, as in Figure 2.6 (Hinton et al., 2006; Ranzato et al., 2011; Fei-Fei et al., 2004). In the figure, note how sampling functions as a type of explanation: because the model in Figure 2.6 appears to do a good job of sampling *new* images of faces from its learned distribution, a researcher may find confidence that the classifier has learned the relevant information from the data.

When tractable, one might choose to remove the randomization by defining the

function $\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta)$ to calculate the maximum-likelihood estimate from 3.4,

$$\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) = \arg\max_{\mathbf{X}} P_\theta(\mathbf{X}|\hat{y}_\theta(\mathbf{x})), \tag{3.5}$$

or some other statistic of the distribution at hand.

However, sampling from the class's posterior (or calculating a statistic from the posterior, as in Equation (3.5)) is different than asking how each dimension of the datum affected the classification. Sampling (or computing a statistic from) the posterior tells us which data are *likely* given the model of the chosen class; thus two different data classified the same way would give the same explanations (modulo the randomization of sampling)[2]. The type of explanation I seek, however, is not which data are *likely* to appear; rather, I want to determine the importance of the dimensions of a particular datum that is classified by a given classifier.

## 3.2 Approximating the Classifier

Another method to understand the classifications of a black-box classifier is to approximate the classifier with a simpler, interpretable classifier. *Rule generation* is a popular example of this method, in which complicated classifiers (such as multilayer perceptrons (Cybenko, 1989)) are approximated with simpler "rule-based" classifiers (Fu, 1994). Then the classifications of the original black-box classifier are explained based on the simpler classifier.

The simpler "rule-based" classifier is often a decision tree. A decision tree is a

[2]Some methods generate more datum-specific explanations by conditioning the distribution on some mid-level features extracted from the data. While this does allow two different data to have two different explanations, it does not change the fundamental fact that asking "Which data were likely?" is not the question that I pose in this dissertation.

tree where each node is a simple logic statement (*e.g.,* $x_1 < 0$). Traveling from the root of the tree to a leaf results in a long logic statement as well as a classification ("If $x_1 < 0$, then if $x_6 > 4$, then ... then classify as $+1$"). In this case, $\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta)$ is exactly the logic statement that explains how the datum traveled to a leaf of the decision tree. Unfortunately, the depth of these trees can be quite large when applied to a difficult problem (Breslow and Aha, 1997). As a result, this method adds two new difficulties to the explanation problem. First, the explanatory logic statement is often excessively long and cumbersome, making the explanation itself difficult to understand ("The datum was classified as positive because $x_1 < 0$ and $x_6 > 4$ and..."). Second, the explanations are based on an approximation to the original classifier, and thus might include misleading information where the approximation is poor.

Baehrens et al. (2010) perform a similar type of classifier-approximation, and combine the result with a gradient-based explanation. The simpler, approximating classifier is a Gaussian Process Classifier (GPC). GPCs are known to be able to approximate any function (with enough training data). The gradient of the resulting GPC is then used to explain classifications. However, as I will explain in Section 3.4, the gradient does not provide satisfactory explanations with even simple linear classifiers, and there is no reason to expect the gradient of a GPC to be any more illuminating.

A general problem for this class of methods is the extra time needed to learn the approximation of the classifier. This can also require an enormous amount of data in order for the approximation to be close. Moreover, as I noted above, the explanation might be incorrect and misleading when the approximation does not provide a good fit to the original classifier.

## 3.3 Exhaustive Data Search

Given a classifier $\hat{y}_\theta$ and a datum $\mathbf{x}$, some methods explain the importance of feature $x_i$ by modifying $\mathbf{x}$ itself and re-evaluating $\hat{y}_\theta$. For example, Robnik-Šikonja and Kononenko (2008) and Strumbelj et al. (2009) measure the importance of dimension $x_i$ by calculating the expected value of $\hat{y}_\theta(\mathbf{x})$ over all possible changes in the dimensions $x_{j\neq i}$. That is to say, the value $x_i$ is held fixed while the value of all other entries in the vector $\mathbf{x}$ are changed (over a very large range). The idea is that $x_i$ is very important to the positive classification of $\mathbf{x}$ if $\hat{y}_\theta(\mathbf{x})$ is often positive when $x_i$ takes on the given value.

One clear difficulty with this approach is the computational cost of varying all $x_{j\neq i}$ through all possible values. This requires sampling from a set that is exponential in the dimensionality of $\mathbf{x}$. The authors compensate for this by using approximate sampling techniques such as Markov Chain Monte Carlo (Metropolis et al., 1953; Geman and Geman, 1984). However, this introduces an unwanted dependence on the quality of the approximation (to the distribution over which the expected value will be calculated), as I described in Section 3.2.

## 3.4 The Gradient Approach to Explanations

The gradient approach (also called *sensitivity analysis*) defines Explain as follows. Given a datum $\mathbf{x} \in \mathbb{R}^n$ and a classifier $\hat{y}_\theta$,

$$\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) = \begin{bmatrix} \frac{\partial \hat{y}_\theta(\mathbf{x})}{\partial x_1} \\ \frac{\partial \hat{y}_\theta(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial \hat{y}_\theta(\mathbf{x})}{\partial x_n} \end{bmatrix}. \tag{3.6}$$

The gradient approach to explanations follows the logic that $x_i$ was largely responsible for the classification if $\hat{y}_\theta$ is very sensitive to small changes in $x_i$ in a neighborhood around the datum $\mathbf{x}$.

Baehrens et al. (2010) use the gradient approach to explain the classifications of any classifier[3]. Indeed, the benefit of this type of analysis is that it can be applied to *any* differentiable classifier $\hat{y}_\theta$. While this is certainly a desirable trait, let us pause to question whether the information conveyed by the gradient approach truly tells us how each dimension of the datum affected the classification, or whether it tells us something else altogether. Let us look at a pair of examples.

[3]Baehrens et al. (2010) first approximate the classifier as a Gaussian process, then apply the gradient approach to the approximation.

Let $\hat{y}_\theta(\mathbf{x}) = x_1 + 2x_2$, and let $\mathbf{x} = \begin{bmatrix} 5 \\ -2 \end{bmatrix}$. In this case,

$$\begin{aligned} \hat{y}_\theta(\mathbf{x}) &= 1(5) + 2(-2) \\ &= 5 - 4 \\ &= 1, \qquad \text{and} \\ \mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) &= \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \end{aligned}$$

Thus the gradient approach appears to tell us that $x_2$ was twice as important to the classification as $x_1$ was.

Now let $\mathbf{x} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$, and let $\hat{y}_\theta$ be as before. Then we have

$$\begin{aligned} \hat{y}_\theta(\mathbf{x}) &= 1(-3) + 2(2) \\ &= -3 + 4 \\ &= 1, \qquad \text{and} \\ \mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) &= \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \end{aligned} \tag{3.7}$$

Thus the gradient approach still explains that $x_2$ was twice as important to the classification as $x_1$ was.

It would seem that under the gradient approach, Explain omits some useful information. In the first example, the dimension $x_2$ actually pulled the classification toward the *negative* class, but the classification was positive due to the large positive

value of $x_1$. In the second example, these roles were reversed: the dimension $x_1$ pulled the classification toward negative, but was "overpowered" by the large positive value contributed by $x_2$. None of this information is present in the gradient approach's version of Explain.

Recall the type of explanation I seek in this dissertation: given a single datum $\mathbf{x}$, I wish to know how each dimension $x_i$ of $\mathbf{x}$ affected the classification $\hat{y}_\theta(\mathbf{x})$. In general, the gradient approach does not give us this information. Given a datum $\mathbf{x}$ classified by $\hat{y}_\theta$, the gradient approach answers the question "how sensitive is the classifier to each $x_i$?" which is, in general, a different question to ask. Rather than determining how sensitive $\hat{y}_\theta$ is to small changes in $x_i$ (as in the gradient approach), I want to know the responsibility of $x_i$ at exactly the value it assumed in the datum $\mathbf{x}$. Thus the gradient approach is not suitable for my goal.

## 3.5 The Contribution Approach to Explanations

Given a datum $\mathbf{x} \in \mathbb{R}^n$, let the classifier $\hat{y}_\theta$ have the form

$$\hat{y}_\theta(\mathbf{x}) = \operatorname{sgn}\left[\sum_{i=1}^{n} f_i(x_i)\right] \tag{3.8}$$

for real-valued functions $f_i$. Using the terminology of Poulin et al. (2006), a function $\hat{y}_\theta$ in the form of Equation (3.8) is called *additive*[4] and $f_j(x_j)$ is called the *contribution* of dimension $x_j$. Thus the contribution of a variable is exactly the amount that it contributes to the overall sum used in classification.

[4]In an additive classifier, the parameters $\theta$ define the functions $f_i$, meaning $\theta = \{f_1, \ldots, f_n\}$

Under the contribution approach, the explanation of a classification is defined by

$$\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) = \begin{bmatrix} f_1(x_1) \\ f_2(x_2) \\ \vdots \\ f_n(x_n) \end{bmatrix}. \tag{3.9}$$

Thus $x_i$ was important to the classification if $x_i$ contributed a large summand to the overall sum in $\hat{y}_\theta(\mathbf{x})$ (*i.e.,* if $|f_i(x_i)|$ is large).

Recalling our examples from Section 3.4, when $\hat{y}_\theta(\mathbf{x}) = x_1 + 2x_2$ and $\mathbf{x} = \begin{bmatrix} 5 \\ -2 \end{bmatrix}$, we have

$$\begin{aligned} \hat{y}_\theta(\mathbf{x}) &= 1(5) + 2(-2) \\ &= 5 - 4 \\ &= 1. \end{aligned} \tag{3.10}$$

The contribution-based explanation gives us

$$\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) = \begin{bmatrix} 5 \\ -4 \end{bmatrix}.$$

This explanation perfectly captures the intuition in Equation (3.10): namely that $\hat{y}_\theta(\mathbf{x}) = 1$ because $x_1$ added a value of 5 to the overall sum, and $x_2$ added a value of $-4$. Note that this information was lacking in the gradient approach.

Similarly, when $\mathbf{x} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$, we have

$$\begin{aligned} \hat{y}_\theta(\mathbf{x}) &= 1(-3) + 2(2) \\ &= -3 + 4 \\ &= 1, \end{aligned} \tag{3.11}$$

and the contribution approach gives us the explanation

$$\mathsf{Explain}(\mathbf{x}, \hat{y}_\theta) = \begin{bmatrix} -3 \\ 4 \end{bmatrix},$$

which accurately reflects what is calculated in Equation (3.11): $\hat{y}_\theta(\mathbf{x}) = 1$ because $x_1$ contributed a value of $-3$ to the overall sum, and $x_2$ added a value of 4.

In general, the contribution approach tells us very useful information about the responsibility of each $x_i$ in the classification $\hat{y}_\theta(\mathbf{x})$. Given a datum $\mathbf{x}$ and an additive classifier $\hat{y}_\theta(\mathbf{x}) = \sum_j f_j(x_j)$, it is clear that $f_i(x_i)$ is exactly the amount that dimension $x_i$ adds to the overall classification. In other words, the contribution of $x_i$ tells us how the $i^{\text{th}}$ dimension of the datum affected the classification, based on how much it "pulled" the sum toward the positive or negative class. This is exactly the type of information that I would like to capture in order to understand how responsible each dimension $x_i$ was for a classification.

The contribution approach gives us a good explanation about what caused the classification, though it is only defined when the classifier is additive. Note that many common classifiers are additive. Any linear classifier (such as a linear support vector

machine (Cortes and Vapnik, 1995), seen in Section 2.1) is additive by definition. Even Naïve Bayes (Webb et al., 2005) can be written as an additive function. Thus the constraint of additivity still allows us to examine a large cross-section of machine-learning methods.

One challenge with this approach is that the data are usually transformed by *feature extraction*, as discussed in Section 1.2, before being passed to the additive classifier. In this case, the contribution approach tells us how much each dimension of the *feature vector* contributed to the classification, but does not directly explain the contributions of the original datum. Extending the contribution approach to a particular family of popular feature transformations, called hierarchical networks, is exactly the focus of Chapter 4.

# Chapter 4

# Contribution Propagation

In Section 3.5, I showed that *contributions* are a helpful way to understand how each dimension of the datum affected the classification. Recall that given an *additive* classifier, which is of the form

$$\hat{y}_\theta(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n} f_i(x_i)\right), \tag{4.1}$$

the contribution of dimension $j$ is precisely the $j^{\text{th}}$ summand; namely $f_j(x_j)$. Intuitively, it is clear that the contribution is precisely the amount that the $j^{\text{th}}$ dimension of $\mathbf{x}$ affected the overall sum being calculated.

In Section 1.2 I discussed how sometimes the classifier does not take the datum $\mathbf{x}$ as input directly; instead, $\hat{y}_\theta$ may classify some features $\mathbf{z}$ calculated from the datum $\mathbf{x}$. That is to say, when classifying $\mathbf{x}$, one may first calculate the features $\mathbf{z} = g(\mathbf{x})$ (for some chosen function $g$), and then apply the classifier to the features, $\hat{y}_\theta(\mathbf{z})$. In this setting, contributions only tell us the importance of the dimensions of $\mathbf{z}$, yet one may still want to know how the dimensions of $\mathbf{x}$ affected the classification.

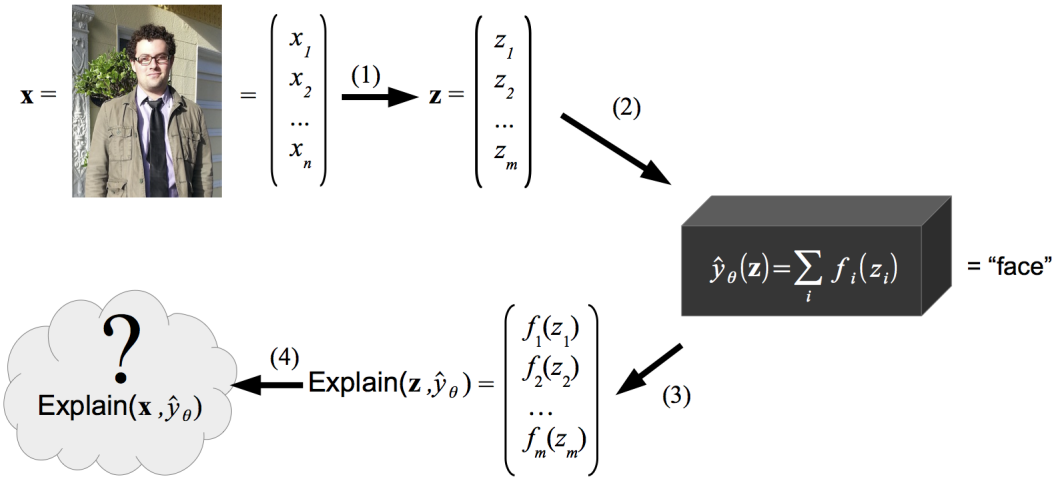



Figure 4.1: Contributions explain how each dimension of the classifier's input affected the classification. When the feature vector $\mathbf{z}$ is extracted from the datum $\mathbf{x}$ (1), the feature vector is given as input to the classifier (2). Contributions can then explain the importance of the different dimensions $z_i$ of the feature vector $\mathbf{z}$ (3). However, contributions alone do not tell us the importance of the dimensions of the original datum (4), which is my ultimate goal.

This problem is illustrated in Figure 4.1, where a classifier attempts to classify the given test image as *face* or *no-face*. Here, $\mathbf{x}$ is an image (each dimension representing one pixel's intensity or grayscale value), and some features $\mathbf{z}$ are extracted from the image. For example, one may calculate the features by convolving the image with different filters known to help detect a face. In such a case, the contributions of Poulin et al. (2006) only tell us how the dimensions of $\mathbf{z}$ affected the classification, but the explanation does not tell us *how the different regions of the image* (dimensions of $\mathbf{x}$) contributed to the classification. A machine-learning practitioner might like to verify, for instance, that the *face* classification is due to pixels from the person's face, rather than from the necktie or some spurious statistics in the background that (unintentionally) correlate with the *face* class in the training data.

In this section, I will develop an extension of contributions that allow us to cal-

culate, for a certain family of feature extraction methods, the contributions of the dimensions of the input $\mathbf{x}$. That is to say, as long as the features $\mathbf{z}$ are calculated in by a certain family of methods (called hierarchical networks), my proposed method will explain how each dimension of the datum $\mathbf{x}$ affected the classification, even though the features $\mathbf{z}$ were the input to the classification function $\hat{y}_\theta(\cdot)$. I will derive this method in Section 4.2, and I will prove several important properties of this method in Section 4.3 before applying it to real machine-learning tasks in Sections 5.1 and 5.3.

## 4.1 Preliminaries

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed, acyclic graph. $\mathcal{V}$ is the set of nodes in the graph, and $\mathcal{E}$ the set of edges. I write

$$(U \to V) \in \mathcal{E}$$

to mean that $U \in \mathcal{V}$, $V \in \mathcal{V}$, and there is an edge from $U$ to $V$. For any node $V \in \mathcal{V}$, let

$$\mathsf{ch}\,(V) \stackrel{\texttt{def}}{=} \{U \in \mathcal{V} : (U \to V) \in \mathcal{E}\}$$

denote its *children* nodes (the set of all nodes with outgoing edges to $V$). I will abuse the notation slightly by sometimes referring to $\mathsf{ch}\,(V)$ as a vector of nodes; the intended use will be clear from context. By convention, when discussing the edges incoming to $V$ from its children, I refer to $V$ as the *parent node*; in general a node can have many parents. Let us further define

$$\mathsf{pa}\,(U) \stackrel{\texttt{def}}{=} \{V \in \mathcal{V} : (U \to V) \in \mathcal{E}\}$$

to be the set (or vector) of parents of $U$.

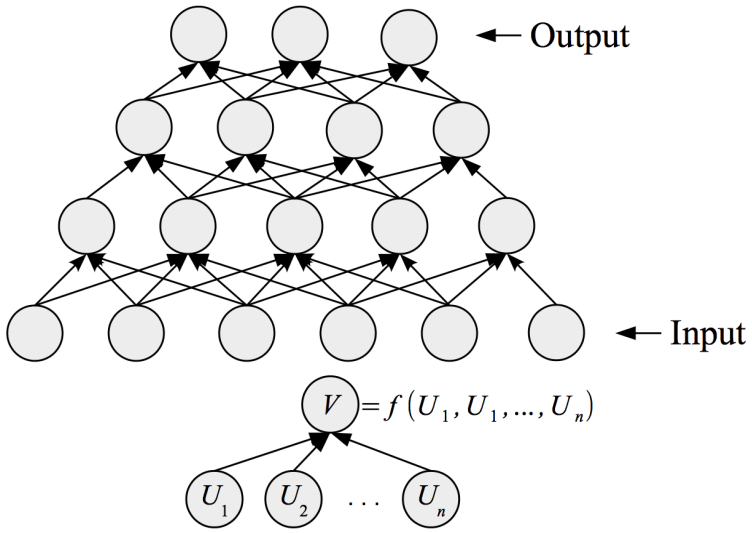



Figure 4.2: A network is a directed, acyclic graph, where each node computes a function of the values of its children. Left: the network is organized into *layers*; the first layer is called the *input*, and the last the *output*. Right: each node in the network computes a function $f$; the arguments of the function are passed in by the children of the node.

Let the graph be organized into *layers*, as in Figure 4.2 (left). Each node in the network computes a simple function whose arguments are given by the children (Figure 4.2, right). When a graph is endowed with this layered topology and the nodes compute such functions (where each function's input is a subset of the previous layer), I refer to the graph as a *hierarchical network* (or *network* for short). The nodes with no incoming edges are called the *input* of the network, and those with no outgoing edges the *output*. In a further abuse of notation, I will sometimes use an

upper-case variable to refer to the identity of a node, as in the statement

$$U_i \in \mathsf{ch}\,(V)\,,$$

and sometimes I will use the same upper-case variable to refer to the value computed by that node's function, as in the statement

$$V = f(U_1, U_2, \ldots, U_n).$$

I will even mix these two notations into a single statement when it simplifies the exposition. For example, one can combine the above two statements into the compact form

$$V = f(\mathsf{ch}\,(V)). \tag{4.2}$$

In Equation (4.2), the $V$ on the right-hand side refers to the identity of a node, and on the left-hand side its computed value.

Networks are often used to extract features from a datum $\mathbf{x}$ by passing each dimension $x_i$ into a separate input node of the network, and using the output of the network as the features $\mathbf{z}$, as in Figure 4.3 (left). This is a common methodology in machine learning; examples include artificial neural networks (Mao and Jain, 1995), the neocognitron (Fukushima, 1980), HMAX (Riesenhuber and Poggio, 1999), deep belief networks (Hinton et al., 2006), convolutional networks (LeCun et al., 1998; Krizhevsky et al., 2012), and deconvolutional networks (Zeiler et al., 2010, 2011).

In fact, using networks for feature extraction is so popular in machine learning that it would be nearly impossible to detail all such methods in an exhaustive list. I will

focus the discussion slightly by considering networks where the function calculated by each node (seen in Figure 4.2, right) is additive. That is to say, every node computes

$$V = \sum_{i=1}^{n} f_i(U_i)$$

where $(U_1, U_2, \ldots, U_n) = \mathsf{ch}(V)$. For the moment, I place no restrictions on the functions $f_i()$. In particular, the $f_i$ do *not* need to be the same for each node $V$. Thus it would be more mathematically rigorous to index $V, f$, and $U$ by their position in the network, as well as the parameters $\theta$ that define the network; however, this notation would quickly become too cumbersome, and so I use the simpler version above. Note that, among others, the networks of LeCun et al. (1998), Krizhevsky et al. (2012), and Zeiler and Fergus (2013) are (or very nearly are) networks of this type. I will refer to these as *additive networks*. Formally, an additive network is a layered network together with the functions $f_i$ computed by each node in the network. To cut down on notation, I will refer to the whole network ensemble (the graph as well as functions) as $\mathcal{G}$.

Recall that an additive classifier has the form

$$\begin{aligned} \hat{y}_\theta(\mathbf{z}) &= \operatorname{sgn}\left[c_\theta(\mathbf{z})\right] \\ &= \operatorname{sgn}\left[\sum_{i=1}^{m} f_i(z_i)\right] \end{aligned}$$

When the classifier's scoring function $c_\theta$ (whose output is called the classifier's "confidence") is also additive (as in Equation (4.1)), it can be considered "just another

node" in the network. In this case, the output layer (say, layer $L$) has only one node (the classifier), and what were previously called the *features* are now calculated by the penultimate layer (layer $L-1$) of the network. This does not change any of the computation; the reorganization is merely conceptual, and will aid in the theory and notation below. This conceptual reorganization is illustrated in Figure 4.3 (right). In what follows, I will refer to this topmost node as $Y$. I will sometimes call $Y$ the *classifier node*, even though technically the classifier incurs an additional $\operatorname{sgn}[\cdot]$ around the output of $Y$.

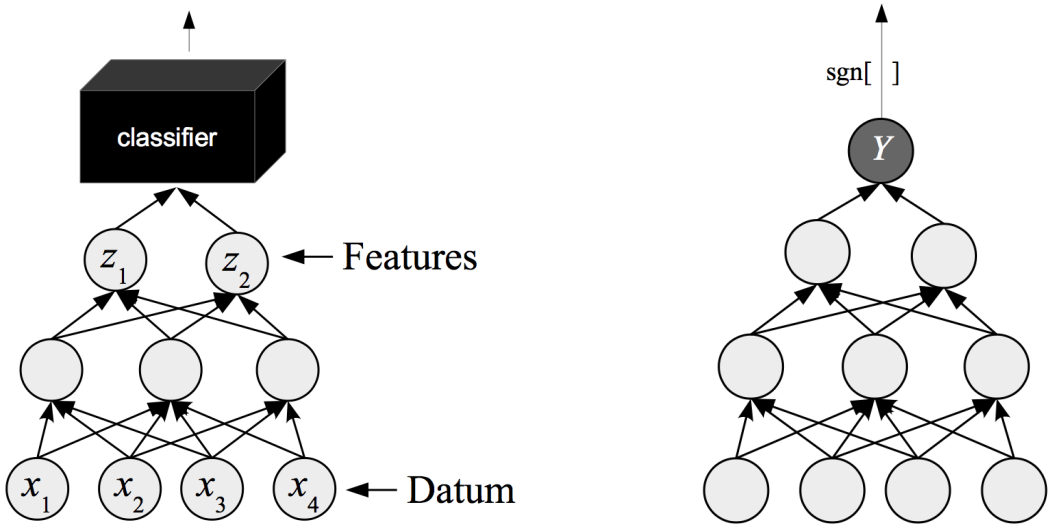



Figure 4.3: Networks are often used for feature extraction in machine learning. Left: the datum $\mathbf{x} = (x_1, x_2, x_3, x_4)$ is passed as input to the network, and the network's output $\mathbf{z} = (z_1, z_2)$ acts as features for classification. Right: when the classifier is additive, the whole of feature extraction and classification can be considered one large network, with one output node. The "sgn [ ]" over the outgoing edge is meant to evoke the final thresholding of the additive function in the classifier.

Recall the problem with using contributions when the classifier takes as input the features extracted from the datum, rather than classifying the datum itself (Figure 4.1). In particular, I seek to determine how each dimension of the datum affected

the classification of that datum. Using the vocabulary of networks, I want to know how each input of the network affected the value of the output node. Contributions (Equation 3.9) give such an explanation for the *features* of the network (layer $L-1$), but not the input (layer 1). In this section, I will extend the notion of contributions from merely explaining the importance of the features to explaining the importance of the input nodes (as well as every other node in the network).

Rather than determining the contribution of each input by treating the entire classifier and network as one large black box, my approach is to analyze each layer of nodes sequentially. Working from the classifier back to the inputs, my method will determine the contributions of feature nodes (layer $L-1$) to the classifier (layer $L$), then the contributions of the nodes in layer $L-2$ to the nodes in layer $L-1$, and so on until we have calculated the contribution of the inputs. I call this process *contribution propagation*.

## 4.2 Deriving Contribution Propagation

The central idea of contribution propagation is that *a node contributes to the classification if it contributes to its parents and its parents contribute to the classification*. This idea is abstract at the moment, but I will make it concrete shortly. In addition to this idea, I list here three properties that should be fulfilled by *any* good explanation method for additive networks:

(i) Given features $\mathbf{z}$ and a classifier $\hat{y}_\theta(\mathbf{z}) = \operatorname{sgn}\left[\sum_i f_i(z_i)\right]$, the contribution of feature $z_i$ is $f_i(z_i)$.

(ii) The sum of the contributions of all nodes in a layer is equal for *all* layers.

(iii) The explanation method is *trustworthy*, meaning that the explanations faithfully explain the logic of the network.

I will clarify and expand on these three properties below.

Property (i) merely states that given an additive classifier $\hat{y}_\theta(\mathbf{z}) = \text{sgn}\left[\sum_j f_j(z_j)\right]$, the contribution of $z_i$ is $f_i(z_i)$, the amount that it influenced the overall sum of $\hat{y}_\theta(\cdot)$. I have already shown concrete examples of this approach in Section 3.5 and in particular Equation (3.9), where I reviewed why these explanations are very informative for the type of question I am asking ("how did each part of the datum affect the classifier?"). Thus property (i) states that whatever equations and algorithm I derive for contribution propagation, they must agree with Equation (3.9) on the contributions of the features.

Property (ii) extends the idea of property (i) to all layers of the network. The beauty of the contributions defined in Equation (3.9) is that they divide the classifier's confidence (Section 1.1) into its summands, thus indicating which dimensions provided the evidence for the confidence. By enforcing this property at *all layers in the network*, I hope to have an explanation that is as informative of the nodes at any layer as Equation (3.9) is at the feature layer. In particular, the explanation should explain what portion of the classifier's confidence came from which node, at any layer of the network.

Property (iii) appears to be the least well defined, but perhaps the most important. In short, I do not want an explanation method to "make up" its explanations. The explanations should be faithful to the logic employed by the network when classifying a datum. Proving trustworthiness will require that I formalize this property, meaning that I will ultimately have to express the notion of trustworthiness mathe-

matically. This formalization, as well as a proof that contribution propagation obeys this property, will appear in Theorem 3. For the moment, let us turn our attention back to deriving contribution propagation from the properties listed above.

Recall the core idea of contribution propagation, that *a node contributes to the classification if it contributes to its parents and its parents contribute to the classification.* It follows from this statement that there are two different types of contributions. First, a node will contribute a certain amount to the overall classification. Let us define the signature of a function which will compute this value,

$$\mathcal{C}(\cdot) : \mathcal{V} \to \mathbb{R}, \tag{4.3}$$

Second, the method's core idea implies that a node contributes to its parents. Let us define the signature of a second function which will compute this value,

$$\mathcal{C}(\cdot \to \cdot) : \mathcal{E} \to \mathbb{R}. \tag{4.4}$$

I have not yet defined these two functions, but already their signatures can be used to translate the core idea of contribution propagation into an equation. Given a node $U_i \in \mathcal{V}$, define the contribution of node $U_i$ to be

$$\mathcal{C}(U_i) \stackrel{\texttt{def}}{=} \sum_{V_j \in \mathsf{pa}(U_i)} \mathcal{C}(U_i \to V_j)\, \mathcal{C}(V_j). \tag{4.5}$$

This formula agrees with the core idea of contribution propagation: *if a node contributes to its parents* (the value defined by $\mathcal{C}(U_i \to V_j)$) *and its parents contribute to the classification* (defined by $\mathcal{C}(V_j)$), *then the node itself contributed to the classifica-*

*tion* (defined by $\mathcal{C}(U_i)$). Note as well that one can already begin to see the outline of an algorithm in Equation (4.5), as the $\mathcal{C}(U_i \to V_j)$ terms allow us to propagate the contributions from $V_j$ (in layer $\ell + 1$) down to $U_i$ (in layer $\ell$). It remains to define the functions $\mathcal{C}(\cdot \to \cdot)$ and $\mathcal{C}(\cdot)$ concretely. I will derive their formulae using Equation (4.5) and properties (i) and (ii).

I begin by deriving the formula for $\mathcal{C}(Y)$, where $Y$ is the classifier node (that is to say, when the classifier is considered the topmost node of the network, as in Figure 4.3, $Y$ refers to this node). Let the children of $Y$ (also called the features) be denoted with $Z_i$ ($1 \leq i \leq m$). Because I have assumed that the network is additive, it follows that $Y$ is an additive function of the nodes $Z_i$,

$$Y = \sum_{i=1}^{m} f_i(Z_i). \tag{4.6}$$

Property (i) tells us that

$$\mathcal{C}(Z_i) = f_i(Z_i).$$

Summing over all $Z_i$ in the feature layer, we have

$$\begin{aligned} \sum_i \mathcal{C}(Z_i) &= \sum_i f_i(Z_i) \\ &= Y \end{aligned} \tag{4.7}$$

Thus the contributions of all the nodes $Z_i$ in the feature layer sum to the value output by $Y$ (recall that this value is the confidence of the classifier, as discussed in Section 1). Now recall property (ii), that the contributions of *any* layer must sum to

the same value. Equation (4.7) says that the contributions in the feature layer sum to the value output by $Y$; it follows that the contributions of all nodes in any single layer must sum to the value output by $Y$ as well. Because the classifier node $Y$ is the only node in its layer, it follows that

$$\mathcal{C}(Y) = Y. \tag{4.8}$$

Equation (4.8) defines the contribution of the topmost node in the network. Equation (4.5) defines how to propagate the contribution down through the network, layer by layer. It remains only to define $\mathcal{C}(U_i \to V)$, the contribution of a child node $U_i$ to its parent $V$. Once again considering the topmost node $Y$ and its children $Z_i$, note that each $Z_i$ has only one parent ($Y$), and thus Equation (4.5) becomes

$$\begin{aligned} \mathcal{C}(Z_i) &= \mathcal{C}(Z_i \to Y)\,\mathcal{C}(Y) \\ &= \mathcal{C}(Z_i \to Y)\,Y \\ \Rightarrow f_i(Z_i) &= \mathcal{C}(Z_i \to Y)\,Y \\ \Rightarrow \mathcal{C}(Z_i \to Y) &= \frac{f_i(Z_i)}{Y} \end{aligned} \tag{4.9}$$

Although I have only derived Equation (4.9) for the edges between the features and the classifier node $(Z_i \to Y) \in \mathcal{E}$, I will show in Section 4.3 that the same formula, when applied to *any* edge in the network, allows contribution propagation to satisfy properties (i), (ii) and (iii).

Further note that, in some cases, the denominator of Equation (4.9) may be equal to zero, in which case $\mathcal{C}(Z_i \to Y)$ would be undefined. I will discuss this issue immediately after defining the contribution propagation algorithm in full, which I do

now.

**Definition 1.** (Definition of contribution propagation)

*Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an additive network with $L$ layers. Let $Y$ denote the topmost node in the network (the only node in layer $L$). For* any *node $V \in \mathcal{V}$, let $V$ compute the additive function*

$$V = \sum_{U_i \in \mathsf{ch}(V)} f_i(U_i)$$

*Furthermore, let*

$$\mathcal{C}(Y) \stackrel{def}{=} Y, \tag{4.10}$$

$$\mathcal{C}(U) \stackrel{def}{=} \sum_{V \in \mathsf{pa}(U)} \mathcal{C}(U \to V)\, \mathcal{C}(V), \quad \textit{and} \tag{4.11}$$

$$\mathcal{C}(U_i \to V) \stackrel{def}{=} \frac{f_i(U_i)}{V} \tag{4.12}$$

*Finally, the contribution-propagation algorithm is defined in Figure 4.4.*

```
// Given an additive network G with L layers,
// which classifies a datum x = (x_1, x_2, ..., x_n),
for layer ℓ = L → 1 do
    for all nodes U_i in layer ℓ do
        Calculate C(U_i)
return (C(x_1), C(x_2), ..., C(x_n))  // the contributions of the datum (layer 1).
```

Figure 4.4: The contribution-propagation algorithm. $\mathcal{C}(U_i)$ is the total contribution of node $U_i$, defined by Equations (4.10), (4.11) and (4.12). Recall that $\mathbf{x} = (x_1, \ldots, x_n)$ is the input to the network, thus $\mathcal{C}(x_j)$ is the total contribution of input dimension $x_j$.

Equation (4.12) is defined by a fraction, and thus I must immediately ask whether the denominator can be equal to zero. The denominator is equal to the value of

the parent node, and I have placed no restrictions on this value (other than being computed by an additive function), thus it may indeed be equal to zero in some cases. This presents a problem for Equation (4.12), and it is a shortcoming of this method in the most general setting. However, when I apply contribution propagation to two specific and popular types of networks in machine learning in Sections 5.1 and 5.3, I will show a remedy for the divide-by-zero problem for each of these networks. Until those sections, I will assume that no node outputs a value of zero, and thus Equation (4.12) is well defined.

## 4.3 Theorems about Contribution Propagation

Now that I have defined contribution propagation (Definition 1), a new method for explaining the classifications of an additive network, I will prove that this method satisfies the three properties of a good explanation method stated at the top of Section 4.2. I will prove each of these in turn. It may already be clear that some of these properties are satisfied due to the derivation above; nonetheless, I formally state and prove the three properties of contribution propagation here for completeness.

The central result of this section is Theorem 3, which proves property (iii) by formalizing the notion of a "trustworthy" explanation. The reader who wishes to skip the theorems and proofs may benefit from at least reading the text immediately above, and the theorem statement of, Theorem 3.

I begin by proving property (i), that the contribution of feature $Z_i$ is exactly $f_i(Z_i)$.

**Theorem 1.** (Contribution propagation satisfies property (i))

*Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an additive network. Let $Y \in \mathcal{V}$ be the output node (the*

*classifier), calculating the function* $Y = \sum_{i=1}^{m} f_i(Z_i)$ *where each* $Z_i \in \mathcal{V}$ *is a child of* $Y$ *(a feature). Then*

$$\mathcal{C}(Z_i) = f_i(Z_i)$$

*Proof.* Recalling that each feature-level node $Z_i$ has only the single parent $Y$ (*i.e.*, there is only one classifier), Equation (4.11) becomes

$$\mathcal{C}(Z_i) = \mathcal{C}(Z_i \to Y)\,\mathcal{C}(Y).$$

Plugging in Equations (4.10) and (4.12), we have

$$\begin{aligned}\mathcal{C}(Z_i) &= \frac{f_i(Z_i)}{Y} Y \\ &= f_i(Z_i).\end{aligned}$$

■

Before proving that contribution propagation satisfies property (ii), I will state and prove a lemma that will simplify the following theorem.

**Lemma 1.** *Let* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *be an additive network. Let* $V \in \mathcal{V}$ *be a node which calculates the additive function*

$$V = \sum_{U_i \in \mathsf{ch}(V)} f_i(U_i).$$

*Then*

$$\sum_{U_i \in \mathsf{ch}(V)} \mathcal{C}(U_i \to V) = 1$$

*Proof.* The proof follows straight from the definition in Equation (4.12),

$$\begin{aligned}
\mathcal{C}\left(U_i \to V\right) &= \frac{f_i(U_i)}{V} \\
\Rightarrow \sum_{U_i \in \mathsf{ch}(V)} \mathcal{C}\left(U_i \to V\right) &= \sum_{U_i \in \mathsf{ch}(V)} \frac{f_i(U_i)}{V} \\
&= \frac{1}{V} \sum_{U_i \in \mathsf{ch}(V)} f_i(U_i) \\
&= \frac{1}{V} V \\
&= 1
\end{aligned}$$

■

I will now prove property (ii), that the sum of the contributions of all nodes in a layer is equal to the sum of the contributions of all nodes in any other layer. In fact, I will prove something even more specific: that the sum of any layer's contributions are always equal to the value output by $Y$.

**Theorem 2.** (Contribution propagation satisfies property (ii))

*Let the nodes $U_i$ (where $1 \leq i \leq n$) be all the nodes in layer $\ell$ of the network. Then*

$$\sum_{i=1}^{n} \mathcal{C}\left(U_i\right) = Y \tag{4.13}$$

*Proof.* The proof is by induction. I already have the base case from Equation (4.10). I will prove the inductive step by assuming that Equation (4.13) is true for layer $\ell + 1$, containing the nodes $V_j$ (where $1 \leq j \leq m$). I will then prove that Equation (4.13) is

also true for layer $\ell$, containing nodes $U_i$ (where $1 \leq i \leq n$).

To begin the inductive step, I assume that

$$\sum_{j=1}^{m} \mathcal{C}\left(V_j\right) = Y.$$

In the following equations, I am aided by one extra definition. Let $\mathcal{E}_\ell$ be the set of all edges between layer $\ell$ (with nodes $U_i$) and layer $\ell+1$ (with nodes $V_j$),

$$\mathcal{E}_\ell \stackrel{\texttt{def}}{=} \{U_i \to V_j \in \mathcal{E} \ : \ U_i \in \text{layer } \ell, V_j \in \text{layer } \ell+1\}.$$

Now Equation (4.11) gives us

$$\begin{aligned}
\mathcal{C}\left(U_i\right) &= \sum_{V_j \in \mathsf{pa}(U_i)} \mathcal{C}\left(U_i \to V_j\right) \mathcal{C}\left(V_j\right) \\
\Rightarrow \sum_{i=1}^{n} \mathcal{C}\left(U_i\right) &= \sum_{i=1}^{n} \left( \sum_{V_j \in \mathsf{pa}(U_i)} \mathcal{C}\left(U_i \to V_j\right) \mathcal{C}\left(V_j\right) \right) \\
&= \sum_{(U_i \to V_j) \in \mathcal{E}_\ell} \mathcal{C}\left(U_i \to V_j\right) \mathcal{C}\left(V_j\right) \\
&= \sum_{j=1}^{m} \left( \sum_{U_i \in \mathsf{ch}(V_j)} \mathcal{C}\left(U_i \to V_j\right) \mathcal{C}\left(V_j\right) \right) \\
&= \sum_{j=1}^{m} \mathcal{C}\left(V_j\right) \left( \sum_{U_i \in \mathsf{ch}(V_j)} \mathcal{C}\left(U_i \to V_j\right) \right) \\
&= \sum_{j=1}^{m} \mathcal{C}\left(V_j\right) \qquad \text{(from Lemma (1))} \\
&= Y.
\end{aligned}$$

■

Let us now turn our attention to property (iii), that "the explanation method is *trustworthy*, meaning that the explanations faithfully explain the logic of the network." At first glance, formalizing such a statement presents a serious paradox: in order to state that an explanation method is trustworthy, one needs to know what faithfully explaining the logic of the network looks like; but in order to understand the logic of the network, one needs a trustworthy method for explaining it.

The trick presented in the statement of Theorem 3 is to find a single network that can be defined in two different ways: one in which the logic is known *a priori*, and another in which one can apply contribution propagation. Both of these properties are found in a *linear* network — that is to say, an additive network where the function computed by each node is a linear one. Using a linear network, the function computed by the network can be rewritten in multiple equivalent ways, due to the flexible nature of linear functions.

Consider the linear network in Figure 4.5 (a), with a classifier node $Y$, layer $\ell + 1$ containing nodes $V_j$, and layer $\ell$ containing nodes $U_i$. Because a linear function of linear functions is itself linear, I can rewrite a linear network of arbitrary depth as a single linear function. That is to say, even though the nodes $U_i$ are not the features nodes, I can still write the output $Y$ as a linear function $Y = \sum_i \gamma_i U_i$ (for some $\gamma_i \in \mathbb{R}$). Moreover, a linear function fits perfectly within the framework of Poulin et al. (2006) given in Equation (4.1). Therefore, we know *a priori* that the contribution of node $U_i$ is given by $\gamma_i U_i$ (which is guaranteed by Theorem 1). The theorem then states that the *a priori* explanation is always equal to the explanation generated by contribution propagation (meaning the application of Definition 1 to the full network, including the $V_j$ nodes). That is to say, I will prove that the result of contribution propagation on a (deep) linear network is equal to the contributions

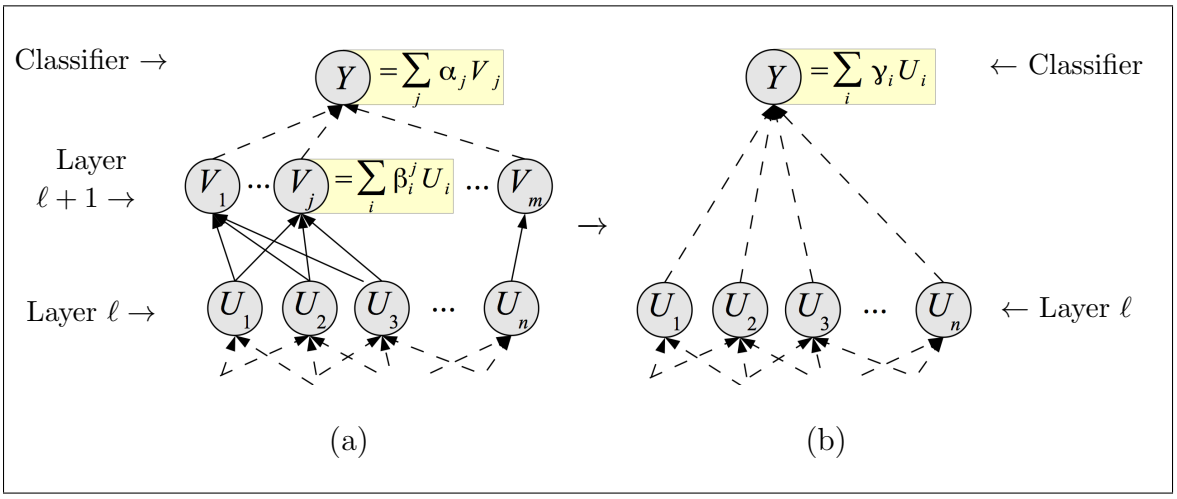



Figure 4.5: Illustration of Theorem 3. Every node in network computes a linear function of its children (a). Because a linear function of linear functions is linear, the output node $Y$ can be rewritten as a function of the layer-$\ell$ nodes $U_i$ (b). This gives us two *equivalent* networks, one in which the nodes $U_i$ are *not* the direct inputs to $Y$ (a) and one in which they are (b). I apply contribution propagation to the "deeper" network (a), and verify that the results are consistent with the contributions defined by Property (i) when applied to the "shallower" network (b). Dashed arrows indicate more layers in the network, omitted from the drawing.

of the (squashed) linear function. This "squashing" of linear networks, which is at the core of Theorem 3, is illustrated in Figure 4.5.

**Theorem 3.** (Trustworthiness of contribution propagation with linear networks.)

*Let $\mathcal{G}$ be a network with $L$ layers. The topmost layer (layer $L$) contains only the classifier node $Y$, and the $\ell^{th}$ layer contains $n$ nodes $U_1, \ldots, U_n$. This network is illustrated in Figure 4.5 (a). Assume each node computes a linear function of its inputs. Because a linear function of linear functions is linear, I can write*

$$Y = \sum_{i=1}^{n} \gamma_i U_i$$

*for some $\gamma_i$ (Figure 4.5, b).*

*Then $\mathcal{C}(U_i) = \gamma_i U_i$, where $\mathcal{C}(U_i)$ is calculated using Definition 1, applied to the full network (Figure 4.5, a).*

*Proof.* The theorem is proved by induction. The base case is covered by Theorem 1. The sketch of the inductive step is as follows. First, I assume that the theorem statement holds true for layer $\ell + 1$ of the network. From this, I will derive the formula for $\gamma_i$. Then Theorem 1 implies that the contribution of $U_i$ must be $\gamma_i U_i$. Finally, I will apply contribution propagation to the full network to calculate $\mathcal{C}(U_i)$, and I will show that these two methods yield the same result.

Moving onto the inductive step in full detail, assume that the theorem statement holds for layer $\ell + 1$, containing $m$ nodes $V_1, \ldots, V_m$. That is to say, assume that the output node $Y$ can be written as a function of the $V_j$ nodes,

$$Y = \sum_{j=1}^{m} \alpha_j V_j \tag{4.14}$$

and, moreover, assume that

$$\mathcal{C}(V_j) = \alpha_j V_j. \tag{4.15}$$

Now I must prove the same to be true of the $\ell^{\text{th}}$ layer, containing $n$ nodes $U_1, \ldots, U_n$. Because the $V_j$ are the parents of the $U_i$, it follows that

$$V_j = \sum_{U_i \in \mathsf{ch}(V_j)} \beta_i^j U_i \tag{4.16}$$

for some fixed coefficients $\beta_i^j \in \mathbb{R}$. Because all functions in this network are linear,

one can also write

$$Y = \sum_{i=1}^{n} \gamma_i U_i \tag{4.17}$$

for some yet-unknown values $\gamma_i$.

In order to define $\gamma_i$, I am aided by rewriting Equation (4.16) as

$$V_j = \sum_{i=1}^{n} \beta_i^j U_i \tag{4.18}$$

with the convention that $\beta_i^j = 0$ if there is no edge from node $U_i$ to $V_j$. The subscript $j = 1, \ldots, n$ indicates that the sum now runs over all nodes in layer $\ell$ (see Figure 4.5), rather than only the children.

This deserves a brief explanation. I previously defined the value of a node $V_j$ in terms of its children $\mathsf{ch}(V_j)$. By extending the definition of $\beta_i^j$ to be 0 when $U_i \notin \mathsf{ch}(V_j)$, I can now describe the same network by considering each layer-$\ell + 1$ node $V_j$ to be a function of all nodes in layer $\ell$. It is important to note that I am not actually changing the network, but merely introducing a mathematical notation that will simplify the proof.

Let us return to the task of defining the $\gamma_i$ of Equation (4.17). To this end, expand

Equation (4.14) into

$$\begin{aligned} Y &= \sum_{j=1}^{m} \alpha_j V_j \\ &= \sum_{j=1}^{m} \alpha_j \sum_{i=1}^{n} \beta_i^j U_i \\ &= \sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_j \beta_i^j U_i \\ &= \sum_{i=1}^{n} \gamma_i U_i \end{aligned} \tag{4.19}$$

where one can finally see that

$$\gamma_i \stackrel{\texttt{def}}{=} \sum_{j=1}^{m} \alpha_j \beta_i^j. \tag{4.20}$$

If $Y = \sum_i \gamma_i U_i$, then Theorem 1 implies that the contribution of $U_i$ must be $\gamma_i U_i$. Plugging in Equation (4.20), it follows that the contribution of $U_i$ is $\sum_{j=1}^{m} \alpha_j \beta_i^j U_i$. This is the *a priori* explanation of the network: I have written $Y$ as a linear function of the $U_i$ nodes, and therefore the contribution of $U_i$ was defined by Theorem 1

It remains only to prove that $\mathcal{C}(U_i) = \sum_{j=1}^{m} \alpha_j \beta_i^j U_i$ as well (where $\mathcal{C}(U_i)$ is calculated using the equations of Definition 1 on the "full" network of Figure 4.5-a, rather than the "squashed" network of Figure 4.5-b). Note that applying Equation (4.12) to the linear functions in Equations (4.14) and (4.18) yields

$$\begin{aligned} \mathcal{C}(V_j \to Y) &= \frac{\alpha_j V_j}{Y}, \quad \text{and} \\ \mathcal{C}(U_i \to V_j) &= \frac{\beta_i^j U_i}{V_j}. \end{aligned} \tag{4.21}$$

Plugging Equations (4.19) and (4.21) into Definition 1,

$$\begin{aligned}
\mathcal{C}(U_i) &= \sum_{V_j \in \mathsf{pa}(U_i)} \mathcal{C}(U_i \to V_j)\,\mathcal{C}(V_j) \\
&= \sum_{V_j \in \mathsf{pa}(U_i)} \mathcal{C}(U_i \to V_j)\,\mathcal{C}(V_j \to Y)\,\mathcal{C}(Y) \\
&= \sum_{V_j \in \mathsf{pa}(U_i)} \frac{\beta_i^j U_i}{V_j} \frac{\alpha_j V_j}{Y} Y \qquad (4.22) \\
&= \sum_{V_j \in \mathsf{pa}(U_i)} \alpha_j \beta_i^j U_i \\
&= \sum_{j=1}^{m} \alpha_j \beta_i^j U_i \qquad (4.23) \\
&= \gamma_i U_i
\end{aligned}$$

Note that, in Equation 4.23, I have again used the convention that $\beta_j^i = 0$ if there is no edge from $U_j$ to $V_i$. This completes the proof. ■

Theorem 3 proves that the explanations generated by contribution propagation are trustworthy (given my mathematical interpretation of the word, which is that a "squashed" network should exhibit the same contributions as a deep network if it calculates the same function) *if the network is linear.* Theorems 1 and 2, however, apply to the much more general case of *additive* networks. I would like to prove an analogous theorem to Theorem 3 in the additive setting as well. Unfortunately, I have no such theorem. The problem is not a lack of proof, but rather a lack of theorem statement: crucial to the statement of Theorem 3 is the fact that one can "squash" the edges between two layers when the network is linear. Because one cannot "squash"

a more general additive function in the same way, the proof of trustworthiness for additive networks must be expressed in some other way. I leave this for future work, but I will show empirical evidence of this method's trustworthiness with additive networks in the sections ahead.

I will briefly summarize Section 4 before moving on. In Section 4.2, I gave three desirable properties of any explanation method for additive networks. Using these three properties, I derived a new method called contribution propagation. In Section 4.3, I proved that the three properties are satisfied by contribution propagation (with the third property only being proved for linear networks). At this point, I have discussed the encouraging theoretical guarantees about the method, but I have yet to implement it. In Sections 5.1 and 5.3, I will apply contribution propagation to two types of networks: a simpler network that is nearly linear, and a more complex network called HMAX (Riesenhuber and Poggio, 1999).

# Chapter 5

# Implementing and Evaluating Contribution Propagation

In Section 5.1, I will introduce a simple network and image-classification task which uses synthetic data. I will apply contribution propagation to this network in Section 5.2; the controlled nature of the data will empirically demonstrate the trustworthiness of the explanations provided by contribution propagation. In Section 5.3, I will apply contribution propagation to a more complex network, and several more complex tasks. The results will provide new insight into how these networks are able to solve difficult problems in computer vision.

## 5.1 The Linear/Max Network

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a network organized into layers. Let $V_k \in \mathcal{V}$ denote a node in an odd-numbered layer, and let $U_i \in \mathcal{V}$ denote a node in an even-numbered layer. Let

nodes in odd-numbered layers compute the linear function

$$V_k = \sum_{U_i \in \mathsf{ch}(V_k)} \beta_i^k U_i \tag{5.1}$$

and let nodes in even-numbered layers compute a maximum operation,

$$U_i = \max_{W_j \in \mathsf{ch}(U_i)} W_j. \tag{5.2}$$

This network is a simplified sketch of a *convolutional neural network* (LeCun et al., 1998; Huang and LeCun, 2006; Kavukcuoglu et al., 2010; Zeiler and Fergus, 2013). Many researchers have justified the alternation between these two operations (or other similar operations), stating that they approximate the circuitry of the mammalian visual cortex (LeCun et al., 1998; Riesenhuber and Poggio, 1999; Serre et al., 2007), though I make no such claims here. However, I will borrow some terminology from the neuro-inspired community, referring to the odd-numbered, linear layers as $S$ layers (from ***S**imple cells*), and the even-numbered *max*-layers as $C$ layers (from ***C**omplex cells*) (Riesenhuber and Poggio, 1999).

My interest in the recipe presented above, as well as elaborations on it, is due to the fact that these types of networks are capable of solving some machine-learning and computer-vision tasks. In general, pattern-matching functions like the dot-product are believed to increase the network's *specificity* (meaning that the $S$ layers help the network recognize specific patterns in the input) whereas the maximum functions help build *invariance* (meaning that the $C$ layers help the network recognize objects despite subtle differences in appearance) (Fukushima, 1980). By alternating between these two types of operations, one hopes to find an algorithm that can recognize

complex objects while being robust to changes in orientation, position, lighting, and other common transformations in the appearance of an object.

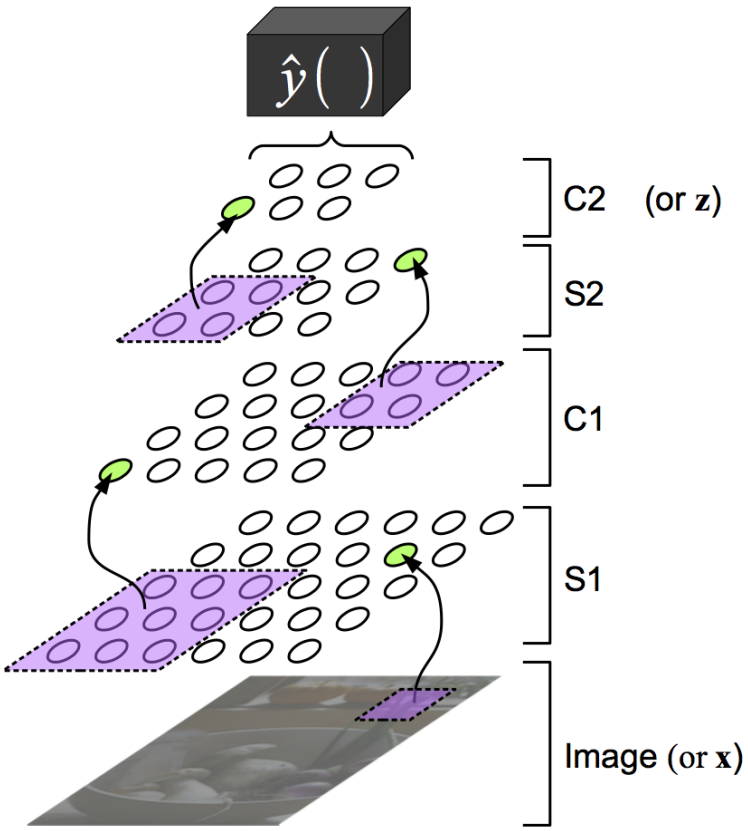



Figure 5.1: A network with alternating layers taking an image as input. Only a small subset of each layer is shown. Each small circle is a node in the network. Processing flows from bottom (*Image*) to top ($\hat{y}$). Arrows illustrate the local connectivity of the network: a small subset of each layer (purple group of nodes) is fed as input to a single node (green) in the following layer. The vector consisting of each $C2$ output is the feature vector used for training and testing the classifier.

Figure 5.1 illustrates a network with five layers: two $S$ layers interleaved with two $C$ layers, and a final classifier. The input of each node is a local region of the previous layer. Associated with each node $V_k$ in an $S$ layer is a vector of coefficients $\boldsymbol{\beta}^k = (\beta_1^k, \ldots, \beta_m^k)$, as in Equation (5.1). In layer $S1$, the coefficients $\boldsymbol{\beta}^k$ are tuned

so that nodes detect edges of various orientations, at all locations of the image. For example, note in Figure 5.2 that $\boldsymbol{\beta}^2$ is tuned to detect vertical edges. This can be seen in the $S1$ outputs of $\boldsymbol{\beta}^2$, where the vertical edges of the input are much brighter (representing a higher output value) than the other lines. Nodes in the second layer $C1$ take a *local* maximum, meaning a maximum over $S1$ outputs which are near each other. This creates a blurring effect in the $C1$ outputs of Figure 5.2.

A node $V_k$ in $S2$ has coefficients $\boldsymbol{\beta}^k$ tuned to recognize junctions of edges in the image (such as corners, or rotated $\top$-shapes) using the *imprinting* method[1], and finally a node in $C2$ performs a maximum over all locations in the image. The classifier is a linear support vector machine (SVM).

## 5.2 Implementing Contribution Propagation with the Linear/Max Network

In order to implement contribution propagation with the network described in Section 5.1, I must adapt Equations (4.10), (4.11) and (4.12) to the functions evaluated in this particular network. The linear nodes $V_k$ fit perfectly within the additive framework, as discussed in the proof of Theorem 3. Recall from Equation 4.12 that the contribution to a linear parent is defined by

$$\mathcal{C}\left(U_i \to V_k\right) = \frac{\beta_i^k U_i}{V_k}. \tag{5.3}$$

The maximum nodes $U_i$, on the other hand, are not additive, and thus I cannot immediately define $\mathcal{C}\left(W_j \to U_i\right)$ which is necessary to implement contribution

[1] I do not discuss the specifics of imprinting here. Details will be given in Section 5.3.

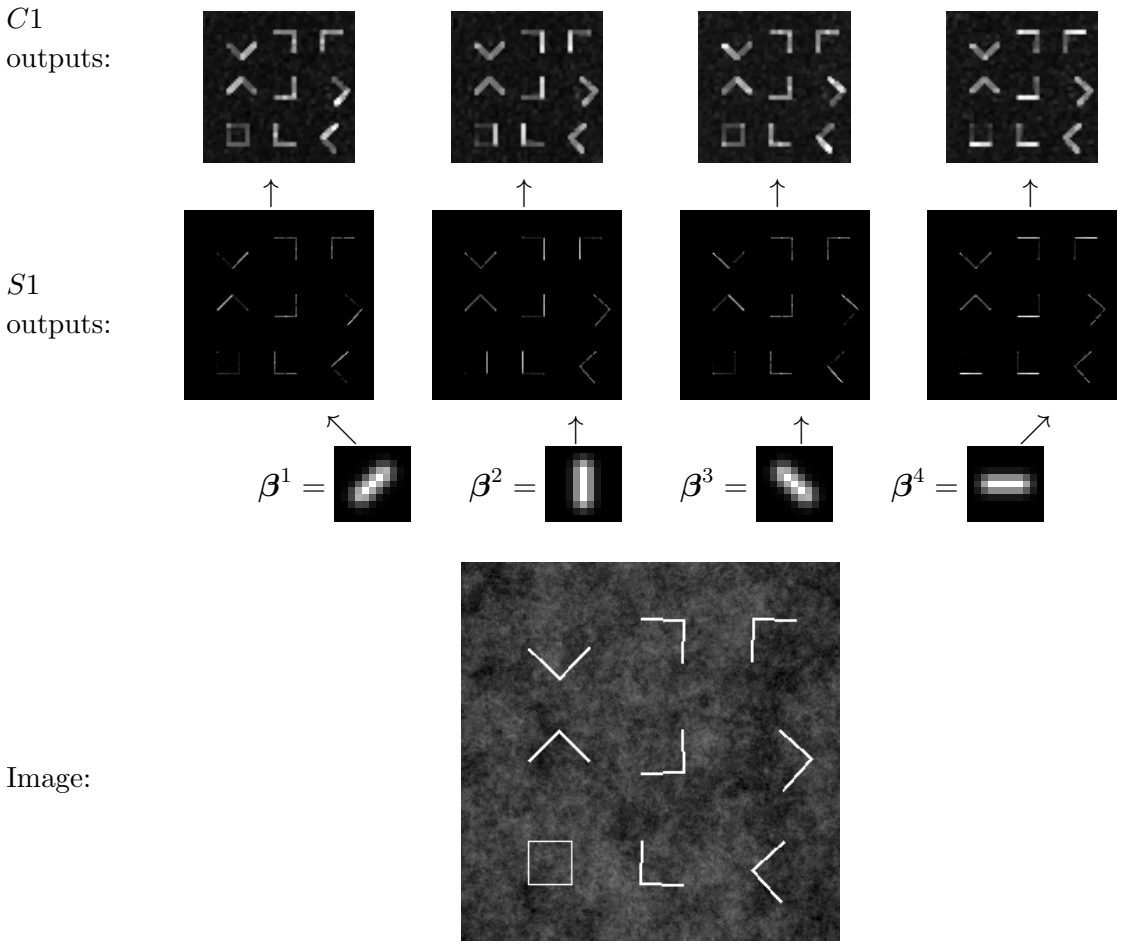



Figure 5.2: Example output from the first two layers of the network. There are four types of nodes at the $S1$ layer, each with coefficients $\boldsymbol{\beta}$ chosen to detect an edge at one of four orientations. Each $C1$ node takes a maximum over a local region of $S1$ nodes, blurring the output and reducing its dimensionality. This alternation between pattern-matching and local pooling is repeated for $S2$ and $C2$ layers (not shown). For visualizing the $S1$ and $C1$ output maps, as well as the coefficients $\boldsymbol{\beta}^k$, a darker colored pixel represents a lower value, and a lighter color represents a larger value. The range of each output map is scaled so that the lowest value is black, and the largest is white.

propagation. To remedy this, I will rewrite the maximum function as

$$U_i = \sum_{W_j \in \mathsf{ch}(U_i)} \delta_j^i W_j \tag{5.4}$$

where $\delta_j^i = 1$ if $W_j$ was the maximum (of all children of $U_i$), and 0 otherwise. Combining Equation (5.4) with Equation (4.12) allows us to define the contribution to a maximum function by

$$\mathcal{C}\left(W_j \to U_i\right) = \delta_j^i. \tag{5.5}$$

This definition for "contribution to a maximum function" is very intuitive: the largest input contributed, and the rest did not[2]. Note that this interpretation and explanation of maximum functions is also used by Zeiler et al. (2011) and Liu and Wang (2012).

The classifier node $Y$ is a linear SVM, taking as input the values of the fourth layer of the network (the features; "C2" in Figure 5.1). I refer to these nodes as $Z_i$ to be consistent with the previous notation for features. A linear classifier is merely a linear function, which is already additive. Thus the linear SVM fits well within the contribution propagation framework. If $Y$ is a linear SVM with support vectors $\mathbf{Z}^s$ and coefficients $\alpha_s$, then we have

$$\mathcal{C}\left(Z_i \to Y\right) \stackrel{\mathtt{def}}{=} \frac{Z_i \left(\sum_{s \in \mathcal{S}} \alpha_s Z_i^s\right)}{Y}. \tag{5.6}$$

The derivation for Equation (5.6) appears in Appendix A. Note that Equation (5.6)

[2]It is possible to have multiple maxima among the inputs (that is to say, multiple $W_j$ may have shared the largest value). To remedy this, I define $\delta_j^i = 1/p$ if $W_j$ was one of the $p$ maxima, and 0 otherwise; in practice, the value of $p$ is almost always 1.

assumes that the SVM has a bias $b$ equal to zero.

Before implementing and evaluating contribution propagation on this network, I must revisit the divide-by-zero problem of Equation (4.12). Because the denominator in that equation could be zero, I must ask whether the above equations are well-defined for all possible inputs. It turns out that, in the case of linear equations, the denominator cancels out in a fortuitous way. Recall Equation (4.23), which shows exactly how the denominator cancels for linear functions. In particular, by rewriting the network as a single linear function, the equation for calculating the contributions of the input nodes (the pixels) simplifies to a form without the possibility of dividing by zero. Thus in what follows, I will use the simplified form of Equation (4.23) in calculating contributions. The details of this process are given in Appendix B.

### 5.2.1 Methodology

Now that the contribution propagation equations are defined for all nodes in the network, I evaluate the contribution-propagation method by training and testing this network on synthetic data. The data are generated so as to constrain the logic of the network. That is to say, there are no spurious statistical correlations in this simple dataset. Examples of training data are shown in Figure 5.3 (top). Positive training images contain a $\vee$ shape, and negative training images a $\wedge$ shape. The shape is placed in a random location of the image, and randomly rotated within $\pm 5$ degrees and scaled within $\pm 20\%$. I generated the data and implemented the network and contribution propagation in Matlab.

I trained the network using twenty positive and twenty negative images. There were four types of $S1$ nodes, responding to edges at orientations of $0°/180°$, $45°/225°$,

$90°/270°$ and $135°/315°$ in every location of the image. $C1$ nodes pooled over a $4 \times 4$ neighborhood of $S1$ units at a single orientation; each $C1$ node is centered on every other $S1$ node, which reduces the number of $C1$ nodes needed to cover the whole image. The output of the $S1$ and $C1$ layers for a test image is shown in Figure 5.2. A single $S2$ node receives as input a $5 \times 5$ region of each $C1$ output map, allowing the $S2$ node to find patterns that have components at any orientation. Each $S2$ node is centered on every other $C1$ node, further reducing the number of nodes required. There are two types of $S2$ nodes at every such location: those that respond to a $\vee$ shape, and those that respond to a $\wedge$ shape. There are only two $C2$ nodes: one taking a maximum of all $S2$ outputs for the $\vee$ shape, and the other for the $\wedge$ shape. These two $C2$ outputs comprise the entirety of the feature vectors passed to the SVM.

### 5.2.2 Results

Figure 5.3 shows four example training images (A) and two example test images (B). Figure 5.3 also shows the results of applying contribution propagation (whose algorithm is defined in Figure 4.4) to the classification of a test image (C). Note that there is no correct class for the test image: it contains both the positive shape ($\vee$) and the negative shape ($\wedge$). Nonetheless, contribution propagation tells us how strongly each pixel pulled the image toward *positive* or *negative* classification; this value is exactly $\mathcal{C}(x_i)$.

The results of contribution propagation are shown in Figure 5.3 (C) using false color. Over a low-contrast version of the original image, I add red if the contribution of the pixel was positive, and blue if it was negative. The saturation of the color is proportional to the magnitude of the contribution, as shown in the legend. The

results are perfectly consistent with the only logical strategy that exists in the data: the pixels belonging to the $\vee$ pulled the classification toward positive (red), the pixels belonging to the $\wedge$ pulled the classification toward negative (blue), and the rest did not contribute at all (grey).

It may be surprising that the red and blue colors do not evenly color the $\vee$ and $\wedge$ shapes in Figure 5.3 (C), indicating that some pixels of the $\vee$ and $\wedge$ shapes had a significantly larger contribution than others. There are two reasons for this. The first is that in each layer, some nodes have more parents than others, and therefore have more opportunities to contribute to the classification. For instance, in Section 5.2.1 I described how an $S2$ node pools over a $5 \times 5$ neighborhood of $C1$ nodes, and the $S2$ nodes are centered on *every other* $C1$ node. This means that a $C1$ node can have as many as nine parents, and an adjacent $C1$ node can have as few as four parents. Nodes with more parents have more opportunities to have a large contribution (which can be seen in Equation 4.11), and thus some $C1$ nodes will inherently influence the classification more than others. Also, some $S1$ nodes do not contribute since they were not a local maximum, which further causes the "splotchy" appearance of the contributions.

The results in Figure 5.3 provide empirical evidence that validates Theorem 3. In particular, when asked to distinguish between $\vee$ shapes and $\wedge$ shapes, the pixels belonging to the $\vee$ and the $\wedge$ are solely responsible for the decision. The fact that contribution propagation provides such a clear explanation despite noise in the data (variance in the appearance in the shapes, random noise in the background, clutter in the test images) speaks to both its robustness and its utility.

Ultimately, the above experiment was meant only to empirically demonstrate the trustworthiness of contribution propagation in a controlled setting. The data was

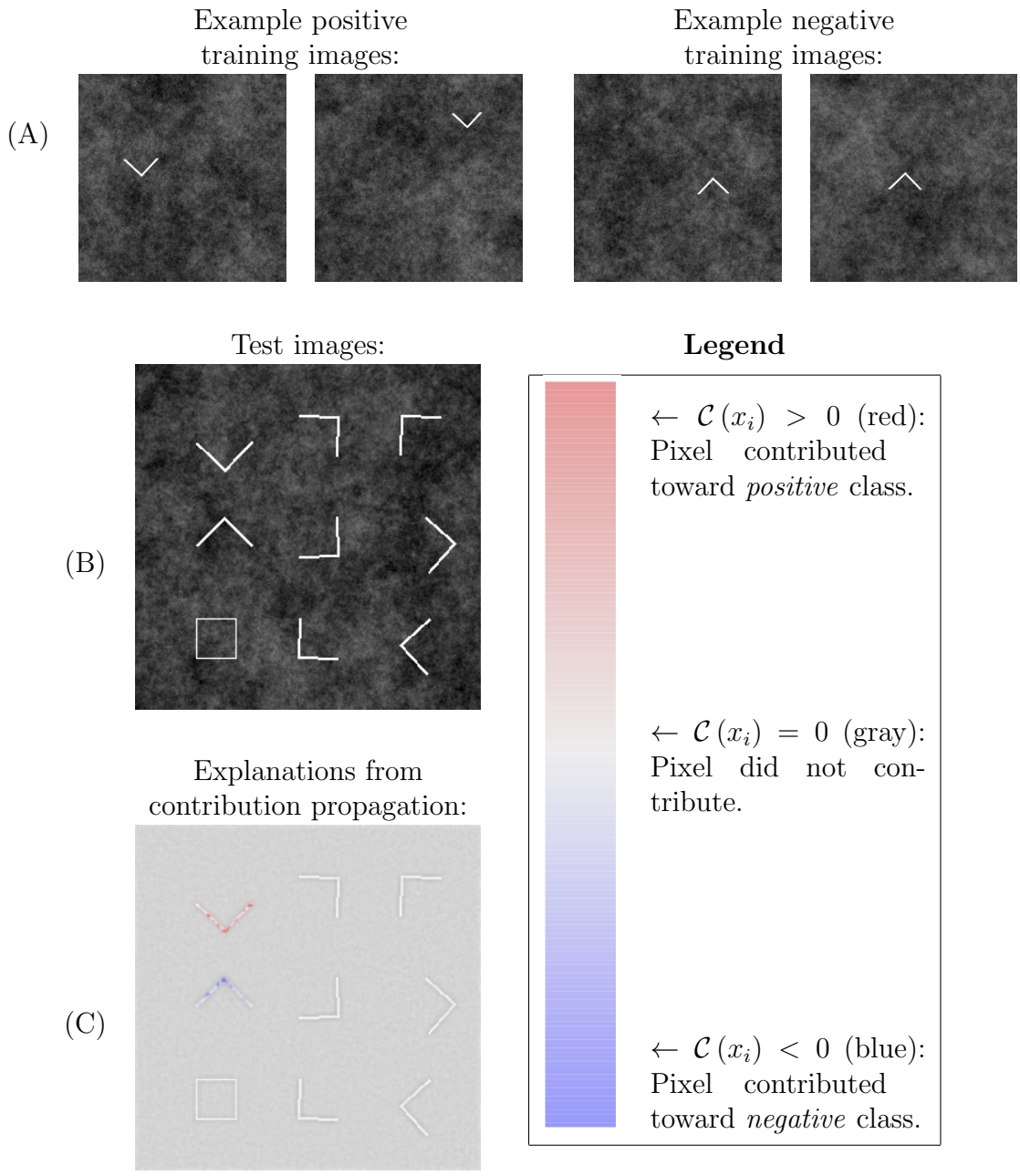



Figure 5.3: Example showing that contribution propagation provides trustworthy explanations. Positive training images contain a $\vee$ shape in a random location, rotated randomly within $\pm 5°$, and randomly scaled within $\pm 20\%$ (A). Negative training images contain a $\wedge$ shape randomized in the same way. A test image contains both shapes, as well as other new shapes (B). All image backgrounds are $1/f$ noise. Contribution propagation (C) explains which pixels $x_i$ pulled the classification toward positive (red), which toward negative (blue), and which did not contribute to the classification (grey).

chosen so that there was only one possible strategy for good classification (*i.e.,* there were no spurious statistics in the background), and thus I was able to verify that the explanations provided by contribution propagation look *trustworthy*. However, the task being solved ($\vee$ vs. $\wedge$) is by no means a challenging one. In the next section, I will apply contribution propagation to a more challenging classification task, requiring a more elaborate network.

## 5.3 HMAX

This section applies contribution propagation to a popular type of network called HMAX (Riesenhuber and Poggio, 1999; Serre et al., 2007). I will evaluate HMAX with both controlled synthetic data as well as two well-known computer vision datasets. Finally, I will use contribution propagation to shed light on the behavior of the HMAX system. Much of the research presented in this section appeared in Landecker et al. (2013).

I begin by briefly reviewing the architecture of the HMAX network. Details of the network's implementation are given in Section 5.3.2. As in Section 5.1, I will examine a 5-layered network: two interleaved $S$ and $C$ layers, and a final linear SVM. In a trained HMAX network, a node $V_j$ in layer 1 or layer 3 (the $S$ layers) computes the radial basis function (RBF)

$$V_j = \exp(-\alpha \parallel \mathsf{ch}\,(V_j) - \mathbf{P}_j \parallel^2), \tag{5.7}$$

where $\mathbf{P}_j$ and $\alpha > 0$ are parameters of the model. The vector $\mathbf{P}_j$ is called the *prototype* of node $V_j$. Equation (5.7) is one of the primary differences between HMAX and the

linear/max network of Section 5.1. Both the RBF and the linear function (5.1) are meant to recognize specific patterns in their inputs. It is thought that the nonlinearity of the RBF allows HMAX to recognize more complex patterns than a simple linear function (Riesenhuber and Poggio, 1999).

As in the linear/max network, nodes $U_i$ in layers 2 or 4 (the $C$ layers) compute a maximum of their inputs:

$$U_i = \max_{V_j \in \mathsf{ch}(U_i)} V_j.$$

Figure 5.1 shows a network with two $S$ layers and two $C$ layers, whose input $\mathbf{x}$ consists of the gray-scale pixel values of an image. The output of the network (i.e., the output of the $C2$ layer) is the feature vector given to a linear SVM. This illustration depicts HMAX just as well as it depicts the linear/max network of the previous section.

### 5.3.1 The Contribution-Propagation Algorithm for HMAX

Recall that the *contribution-propagation* algorithm starts with the contribution $\mathcal{C}(Y)$ of the classifier (which is equal to the confidence of the classifier). The algorithm then iteratively descends through the layers of the network, calculating the contributions of each node in layer $\ell$, for $\ell = L - 1$ down to 1.

In order to complete the description of contribution propagation for HMAX, I need to adapt the contribution propagation equations for the types of nodes in HMAX. I have already derived the equations for the *max*-nodes and linear SVM in Section 5.1, so it remains only to define $\mathcal{C}(U_i \to V_j)$ for nodes that calculate the RBF. Recall that the definition for $\mathcal{C}(U_i \to V_j)$, in Equation (4.12), assumes that $V_j$ computes an additive function. Clearly, the RBF is not additive! Nonetheless, I will derive a

formula for propagating contributions based on an approximation to the RBF.

Consider a node $V_j$ in an $S$-layer. For convenience, let $\mathbf{U} = \mathsf{ch}(V_j)$ and $\mathbf{P} = \mathbf{P}_i$. Then the function computed by $V_j$ is the RBF

$$V_j = \exp(-\alpha \parallel \mathbf{U} - \mathbf{P} \parallel^2). \tag{5.8}$$

The immediate goal is to define a function $\mathcal{C}_{\text{RBF}}(U_i \to V_j)$ which faithfully describes the degree to which $U_i$ contributed to the value of $V_j$.

Equation (5.8) is a measure of distance between the vectors $\mathbf{V}$ and $\mathbf{P}$. Because $\alpha > 0$, a closer distance yields a larger RBF value, and a further distance yields a smaller value. Thus $\mathcal{C}_{\text{RBF}}(U_i \to V_j)$ should be higher if $U_i$ is closer to $P_i$, meaning when $(U_i - P_i)^2$ is smaller[3] . Moreover, the distance calculated by the RBF is tuned by the function $s(x) = \exp(-\alpha x)$. Thus, let us define $\mathcal{C}_{\text{RBF}}(U_i \to V_j)$ as:

$$\begin{aligned} \mathcal{C}_{\text{RBF}}(U_i \to V_j) &= \frac{s((U_i - P_i)^2)}{Z} \\ &= \frac{\exp(-\alpha(U_i - P_i)^2)}{Z}, \end{aligned} \tag{5.9}$$

where $Z$ is a yet-undefined normalization term.

Recall Lemma (1), which tells us that $\sum_i \mathcal{C}(U_i \to V_j) = 1$. This lemma is required in order to prove that contribution propagation satisfies properties (ii) and (iii). To ensure that this lemma still holds, I simply set the denominator $Z$ in Equation (5.9) to equal the sum over all children of the RBF, and we arrive at the full definition:

$$\mathcal{C}_{\text{RBF}}(U_i \to V_j) \stackrel{\mathsf{def}}{=} \frac{\exp(-\alpha(U_i - P_i)^2)}{\sum_{U_k \in \mathsf{ch}(V_j)} \exp(-\alpha(U_k - P_k)^2)}. \tag{5.10}$$

[3]Note that $U_i$ and $P_i$ refer to the $i^{\text{th}}$ entries in the vectors $\mathbf{U}$ and $\mathbf{P}$, respectively.

Equation (5.10) captures some of our intuition about a radial basis function: those children $U_i$ that are close to their target $P_i$ have a higher contribution. Moreover, note that the denominator in Equation (5.10) can never be zero, since the exponential function is strictly positive. Therefore, Equation (5.10) is always well-defined, regardless of the value output by $V_j$.

Despite these facts, I stress that Equation (5.10) attempts to explain, in an additive way, contributions to a non-additive function. Thus it formally falls outside of the guarantees in Section 4.3. I will train and test an HMAX network on the simple task shown in Section 5.1 to obtain empirical evidence of the trustworthiness of the definitions give in this section. These experiments are reported in Section 5.3.3.

### 5.3.2 HMAX Implementation

The code that implements HMAX and its variant of contribution propagation was written by Mick Thomure; the code was implemented based on the equations I derived for contribution propagation (both the general equations from Section 4.1 as well as the HMAX-specific equations from Section 5.3.1). The subsequent experiments were run by me. I describe here the parameters for HMAX that were used in the experiments.

The code implements a four-layer network (Figure 5.1), based on the network of Serre et al. (2007). The input image is preprocessed to form a $256 \times 256$ gray-scale image with local contrast enhancement. An S1 prototype is an $11 \times 11$-pixel Gabor filter (a type of edge-detector). Using Equation (5.8), the S1 layer applies a battery of Gabors at 8 orientations, 2 phases, and 4 scales, with $\alpha = 1$ for all S1 nodes. For each Gabor configuration, an S1 node is centered at every other pixel, resulting in

a set of 64 S1 output maps, each of size $128 \times 128$. A C1 node pools over the two phases and a $5 \times 5$ spatial neighborhood of S1 outputs, again centered at every other S1 output. This results in 32 C1 output maps, each of size $64 \times 64$. For an S2 node, the input is a $7 \times 7$ neighborhood of C1 nodes at all orientations, but at a single scale. The input vector and the prototype of each S2 node are each scaled to unit length ($\|\mathbf{U}\|_2 = \|\mathbf{P}\|_2 = 1$). I use $\alpha = 5.0$ for every S2 node. For each prototype, there is a corresponding S2 node centered at every other C1 node, resulting in multiple $32 \times 32$ S2 output maps, one for each prototype and scale. Finally, a C2 node applies a *max* operation to all locations and all scales of a single prototype's S2 map. Thus the output of the C2 layer is a vector with one component per S2 prototype. This *feature vector* is passed to the linear SVM. I use the SVM$^{light}$ package (Joachims, 1999) with an unbiased SVM ($b = 0$). This allows a simpler derivation of contribution propagation, without impacting the accuracy of the network.

Each S2 node is parameterized by a vector $\mathbf{P}$, called the *prototype*. There are a variety of methods for learning the prototypes in the literature (Thomure et al., 2013); I use the *imprinting* method for its simplicity and noted performance (Serre et al., 2007). To imprint a prototype, the S1 and C1 features are first extracted from a training image. The S2 prototype is imprinted by setting $\mathbf{P}$ equal to some cropped region of the C1 outputs. This region may be chosen explicitly or randomly. Thus in the full HMAX network, the S2 nodes are comparing the C1 outputs of test images to the C1 outputs of regions in training images.

### 5.3.3 Experiments and Results

#### Simple Shapes

In the first experiment, I use synthetic data similar to that described in Section 5.1 to verify that the approximations made for the RBF have not compromised the trustworthiness of my explanation method. The data from this experiment is illustrated in Figure 5.4. Although the dataset is very similar to Section 5.1, I describe it here for completeness. Each training image contains a simple shape, either an 'L' shape (Figure 5.4 (B), positive class) or an inverted 'L' shape (Figure 5.4 (C), negative class). Noise is added by rotating the shape uniformly randomly within $\pm 5$ degrees and translating the shape to a random location, and $1/f$ noise is added to the background. The noise ensures that the learned classifier is nontrivial.

Figure 5.4 (A) shows the two imprinted S2 prototypes around the vertex of the 'L' and inverted 'L' shapes. I train the SVM with 20 training images (10 positive and 10 negative) by giving the images to the network, and using the resulting feature vectors for training.

The test images contain 9 possible shapes (Figure 5.4 (D)), including both an 'L' and inverted 'L', each placed at a random position in a $3 \times 3$ grid and rotated randomly within $\pm 5$ degrees. Again, $1/f$ noise is added to the background. As in Section 5.1, because both the positive and negative objects are present in the test image, I do not expect one classification over the other. The test images were designed to illustrate the trustworthiness of the contribution-propagation algorithm rather than to test the classification accuracy of HMAX (accuracy will be addressed in the next experiment). All test images in this toy example were very near the decision boundary, which is reasonable given that both the positive and negative classes are present in each test image.

I used contribution propagation to explain a test image's classification using false

color[4] as follows. First, the contribution propagates down through the layers of the network (Figure 5.4E-H) using the algorithm presented in Figure 4.4. Figure 4.4 (E) shows the contribution of each node in the $S2$ layer, Figure 4.4 (F) shows the contribution of each node in the $C1$ layer, and so on. This results in the calculation of the contribution $\mathcal{C}(x_i)$ of each pixel $x_i$ (Figure 5.4 (H)). Red-colored pixels contributed to a positive classification ('L'); blue-colored pixels contributed to a negative classification (inverted 'L'); pixels that did not contribute to the classification are drawn in green.

The visualization in Figure 5.4 (H) provides empirical evidence that contribution propagation faithfully explains the logic behind the classifications of HMAX. In particular, the image regions matching the imprinted prototypes are colored red around the 'L' shape, and blue around the inverted 'L' shape. The algorithm thus explains the classification of "undecided": there was a nearly equal "pull" between the pixels surrounding the 'L' (toward positive classification) and those surrounding the inverted 'L' (toward negative classification). This pixel-level explanation of how the image is interpreted by the network and classifier was provided automatically by my contribution-propagation algorithm, and gives evidence for the trustworthiness of this algorithm, even though approximations were made due to the non-additive RBF function.

**Real-World Images**

Next, I use the Caltech101 data set (Fei-Fei et al., 2004) to train the network and a linear, unbiased SVM in a binary classification task using categories of "chair" (*positive* class, corresponding to red in the visualizations) and "dalmatian" (*negative*

[4]The color scheme used in this section's visualizations is different than the one in Section 5.1, but the idea behind the colors is the same.

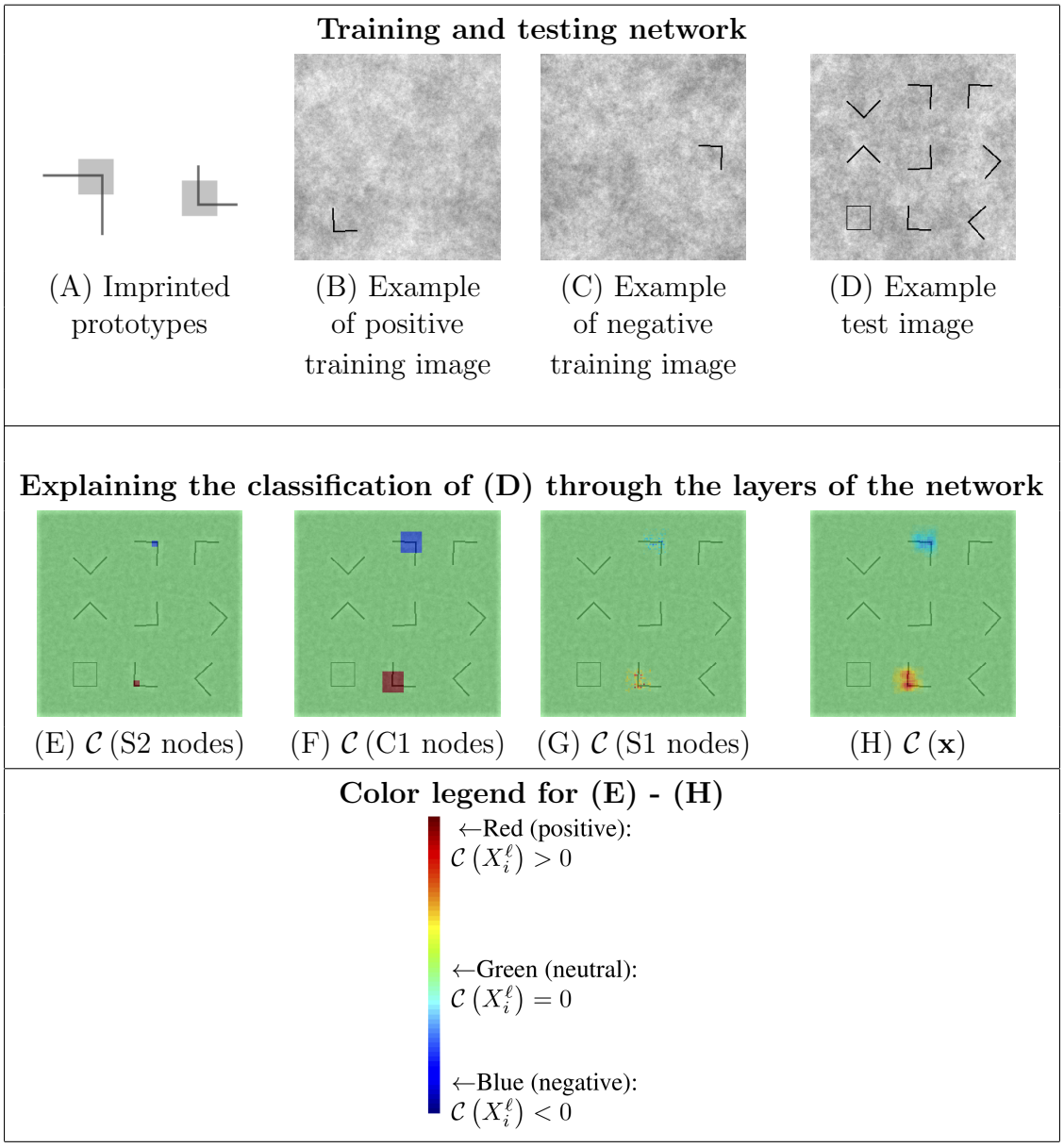



Figure 5.4: Using contribution propagation to visualize HMAX's classification of images containing simple shapes. The contribution of the nodes in each layer verifies the trustworthiness of my algorithm. Two S2 prototypes are used (shaded squares, A). An unbiased linear SVM is trained on images containing either an 'L' shape (B, positive class) or an inverted 'L' (C, negative class). Given a test image (D), contribution propagation gives the contribution of every node at all layers (E-H). Note that the image is drawn in the background of (E-H) in order to better explain the contribution of each region. Colors correspond to each pixel's contribution, as shown in the legend at the bottom. These visualizations give evidence for the trustworthiness of my contribution-propagation algorithm.

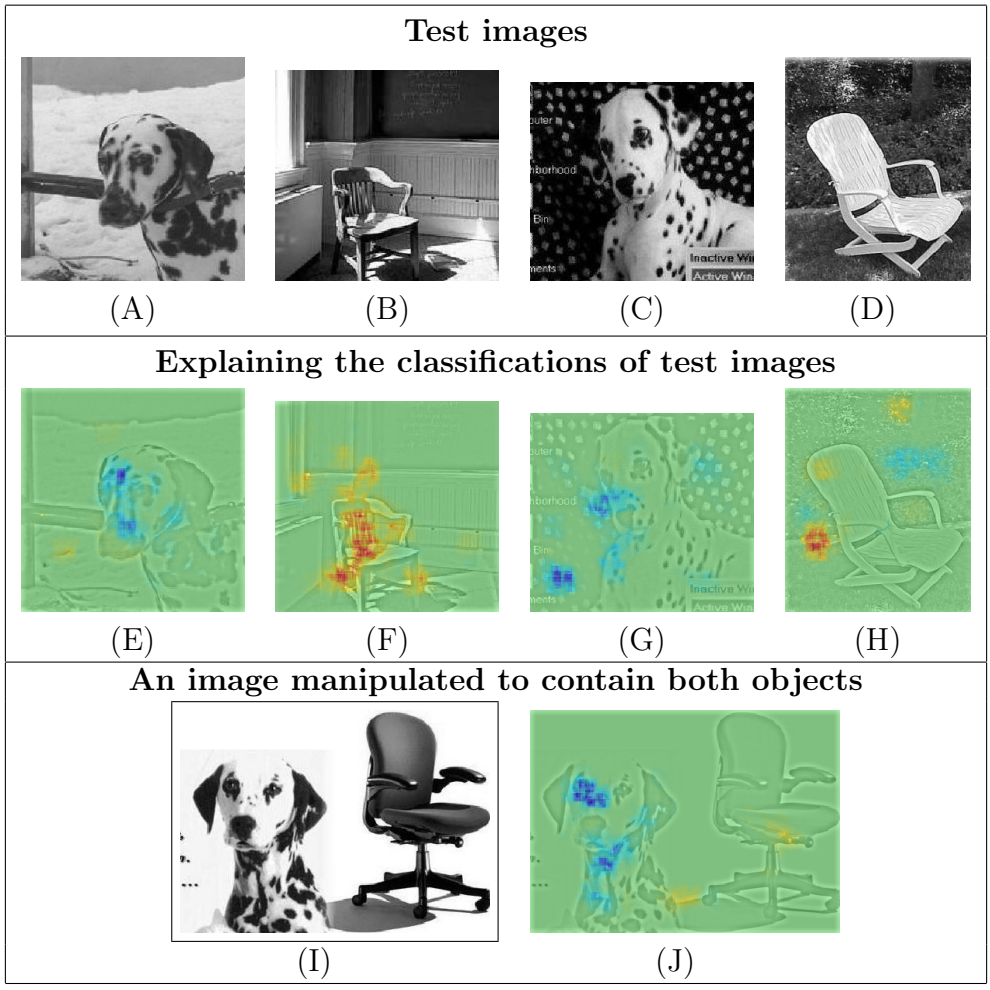



Figure 5.5: Using contribution propagation to visualize HMAX's classification of chairs (positive) vs. dalmatians (negative), from the Caltech101 database. Colors correspond to the legend in Figure 5.4 (bottom): positive contribution (toward *chair*) is denoted with red, and negative contribution (toward *dalmatian*) with blue. Some images (A, B) are correctly classified because of the contribution of pixels that belong to the object being classified (E, blue on dalmatian; F, red on chair). Other images (C, D) are still correctly classified, but partially due to the contribution of background pixels (G, blue on background; H, red on background). An image manipulated to contain both objects (I) is classified as *dalmatian*, and this classification is intuitively explained by the contribution-propagation algorithm (J).

class, corresponding to blue). The categories contain 60 images each. Using 10 splits for cross validation, I randomly choose 30 training images and 30 test images from each category. Following Serre et al. (2007), the network imprints 1000 S2 prototypes randomly from the S2 inputs of the training set, and the SVM is trained on the resulting network's output for each training image. Test images are classified with an average accuracy of 94%, with a 3% standard deviation (for comparison, a biased SVM achieved 93% accuracy with 1.2% standard deviation, so the reader should not be concerned about the lack of bias term).

In Figure 5.5, note that some images (A, B) are correctly classified primarily due to the pixels of the object itself (E, F). However, the explanations also reveal some surprising behavior of the network and classifier (G, H): it appears that some images were correctly classified due to features extracted primarily from the image's background. In Figure 5.5 (G), this may be less surprising, as the background is quite similar to the dalmatian. However, in Figure 5.5 (H), it is unclear why the background (dark red) was taken as evidence for the presence of a chair (or, possibly, absence of a dalmatian). Such an unexpected explanation offered by contribution propagation can be useful to the user who is trying to create a system that will generalize well; the user can see that, at least in some cases, the network is basing its classification on features that are not relevant to the general task, due to either deficiencies in the network or spurious correlations in the data set.

A natural question is how often a correct classification is "surprising" (that is to say, a correctly classified image where the background appears to contribute more than the object). Formulating a metric to define such a surprising classification is beyond the scope of the present work. However, a subjective visual inspection of the classified images reveals 5 of the 60 classifications of test images to be of this nature.

As a final application of contribution propagation, I edited an image to include both a dalmatian and a chair (Figure 5.5 (I)). This image was classified by the network as negative (dalmatian), and the contribution propagation algorithm explains this classification as follows. Figure 5.5 (J) shows that, although there were features associated with the chair class on the right side of the image (yellow, light red), the features extracted from the pixels belonging to the dalmatian were weighted more heavily (deep blue).

Some readers may feel that the small number of training images used may cast doubt on the validity of the trained classifier. However, note that researchers often benchmark their computer-vision system by measuring its performance on the Caltech101 dataset using 30 images per class as training data (Bosch et al., 2007). This experiment was thus designed to mimic a benchmarking process familiar to many computer-vision researchers. In this light, the surprising results presented in Figure 5.5 hint at an important question to the computer-vision community: Does high performance on this dataset indicate a system's capacity for object recognition, or merely for learning spurious statistical (background) cues? The widespread use of this dataset in the computer-vision literature makes this question all the more pressing.

**AnimalDB**

As a final application of contribution propagation, I trained HMAX (and the SVM classifier) using the *AnimalDB* dataset from Serre et al. (2005). This dataset consists of 1200 images taken in a variety of locations, half of which contain an animal. The task, then, is to predict whether or not an image contains an animal. Both the scenery and the type and appearance of the animal vary widely, as seen in Figure 5.6. This is arguably the dataset that most helped catapult HMAX into some degree of popularity

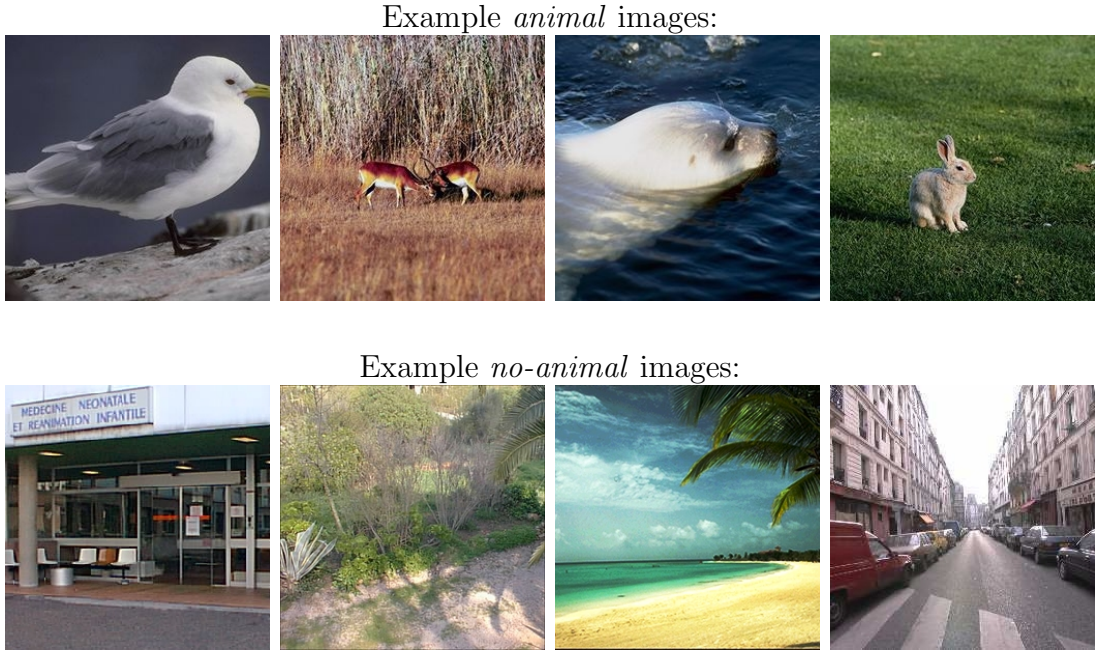



Figure 5.6: Example images from the *AnimalDB* dataset, from Serre et al. (2005)

among computational neuroscientists. In particular, the fact that HMAX achieves accuracy comparable to humans and primates on this dataset (when presented with an image for a very short time period) has been taken as evidence that HMAX mimics the architecture of the primate visual cortex (Serre et al., 2005, 2007).

Our implementation of HMAX achieved approximately 78% test accuracy (comparable to the 80% achieved by the implementation of Serre et al. (2007)). Figure 5.7 shows the results of applying contribution propagation to some test images. In some cases (Figure 5.7, E, F), the correct classification of the image was due to the contribution of pixels belonging to the animal. However, in other cases (Figure 5.7, G, H), pixels belonging to the *background* of the image had the largest contribution to the classification.

These results are surprising. It appears that there are spurious statistics present

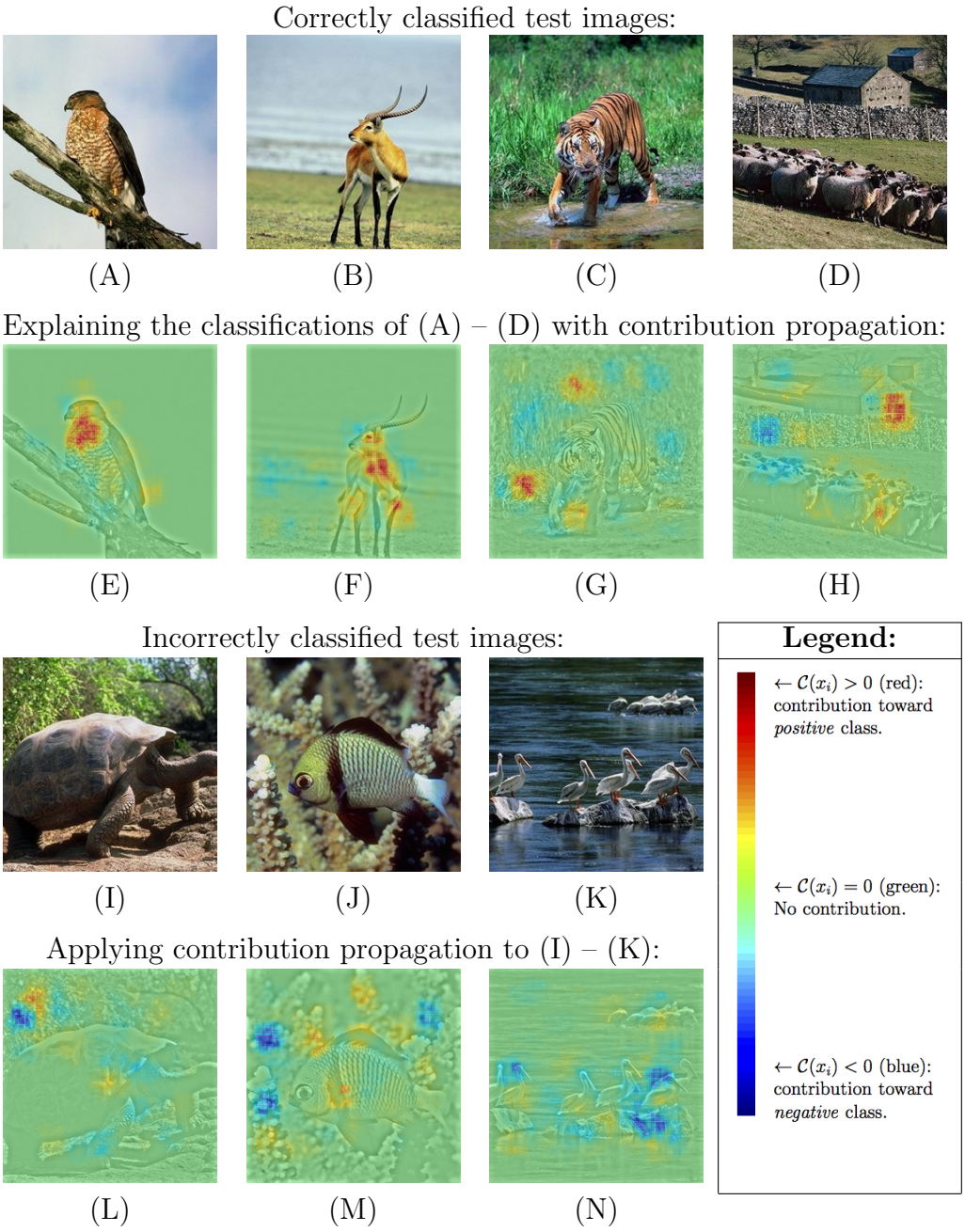



Figure 5.7: Examples of results from applying contribution propagation to HMAX with the *AnimalDB* dataset. (A – D) Four correctly classified *animal* test images and (E – H) the explanation provided by contribution propagation. In some cases, the primary evidence of the *animal* classification came from pixels belonging to the animal itself (red spots on animal in E,F). In others, it appears to be the pixels in the background that cause the *animal* classification (red spots on background in G,H). Contribution propagation also allows us to see what caused the misclassification of images (I – N).

in the background of the images which give away whether or not an animal is present in the foreground. A closer inspection of the dataset reveals that many *animal* images have blurry backgrounds, whereas the *no-animal* images tend to be in focus everywhere. This type of bias in the image is reasonable, given that all the photos were taken by professional photographers. The results of contribution propagation show us how easily an unintended bias can sneak into a dataset.

Moreover, the visualizations in Figure 5.7 call into question the claim, made by Serre et al. (2007), that HMAX implements a good approximation of the logic and circuitry found in the mammalian visual cortex. The evidence for this claim, that HMAX performs similarly to mammals when performing the *animal* / *no-animal* task over a very short time scale, is lessened when one considers the spurious statistics in the dataset found by contribution propagation. It is certainly still possible that the claim of Serre et al. (2007) is true, but the logic employed by our implementation of HMAX and explained by contribution propagation does not appear to be the same logic used by mammals when performing this task.

# Chapter 6

# Summary

I have presented *contribution propagation*, a novel method for explaining the classifications of additive networks, having reviewed why traditional approaches such as a sensitivity analysis fail to give satisfactory explanations. My method extends the contribution-based explanations of Poulin et al. (2006), and determines the contribution of each input based on the internal calculations performed by the network during classification. I proved the trustworthiness of my explanation method for linear networks, and proved other important properties of the method for the more general class of additive networks.

I applied contribution propagation to a simple network consisting of linear and *max*-nodes in order to empirically verify the trustworthiness of the resulting explanations. The synthetic data was generated such that there was only one possible way that the different dimensions of the data could have affected the classifier. The explanations provided by contribution propagation confirmed the method's trustworthiness. Although this linear/max network was conceptually simple, it contains many similarities to modern computer vision systems that achieve state-of-the-art accuracy

on a variety of difficult tasks (Kavukcuoglu et al., 2010; Zeiler and Fergus, 2013). A promising direction for future work will be to apply contribution propagation to these networks.

I reported the results of applying contribution propagation to HMAX, a hierarchical network containing RBF nodes which are *not* additive. In order to derive the relevant equations for contribution propagation, I approximated the RBF in an additive way. I tested the resulting explanations with controlled data (similar to the linear/max network) to empirically verify that the explanations were still trustworthy. I also applied my method to binary classification tasks using some well-known sets of natural images, revealing surprising artifacts in the way that some images are classified. In particular, we see that some images are correctly classified because of the contribution of pixels belonging to the image's background (Figure 5.5 (G), (H); Figure 5.7 (G), (H)). This information is surprising when the task is completed with high accuracy, and is very useful to the user of the machine-learning algorithm. Such information provided by my method can help the user to tune the algorithm for better generalizability, as well as to create data sets without spurious artifacts, so as to encourage the network and classifier to solve the intended problem.

# Part III

# Sparse Coding and Image Classification

In Part II of my dissertation, I discussed the interpretability of binary classifiers. Given a classifier that accurately classifies an image, I asked what the model was really "seeing" (the object or the background). In Parts III and IV of the dissertation I will consider models whose primary goal is to "see" all of the data by creating a good reconstruction of their inputs; I will ask what aspects of these models also lead to good classification accuracy. To be slightly more concrete, I will investigate the classification accuracy of features produced by a family of generative models known as *sparse coding*.

In the following chapters, I will begin by reviewing the sparse coding problem. I will review how sparse coding is used to create features for binary classification, and I will measure the performance of sparse coding in a variety of ways in order to tease apart how the method is useful for classification.

# Chapter 7

# Sparse Coding Background

Sparse coding is the study of how to represent a given object as a combination of known elements; in particular, one would like to encode the given object as a combination of *few* elements out of a large possible set. In the brain, for example, one may ask how a concept (such as *grandmother* or *watermelon-flavored bubble gum*) is encoded by neural firing patterns. Under the principle of sparse coding, each concept is encoded by a small subset of neurons firing strongly (Földiak, 1990). To be clear, for any particular concept, most neurons do not fire; but for each concept, a unique, small subset will fire strongly. Olshausen and Field (1996) showed how a particular mathematical interpretation of sparse coding can give rise to an encoding scheme that mimics some properties of certain neurons in the mammalian visual cortex. Figure 7.1 shows the collection of elements used for encoding (collectively referred to as *the dictionary*) that Olshausen and Field's algorithm learned for the purposes of sparse-coding natural images[1]. That is to say, the dictionary shown in Figure 7.1 consists of elements that were tuned for the purpose of representing *any*

[1] "Natural images" here refers to images of nature.

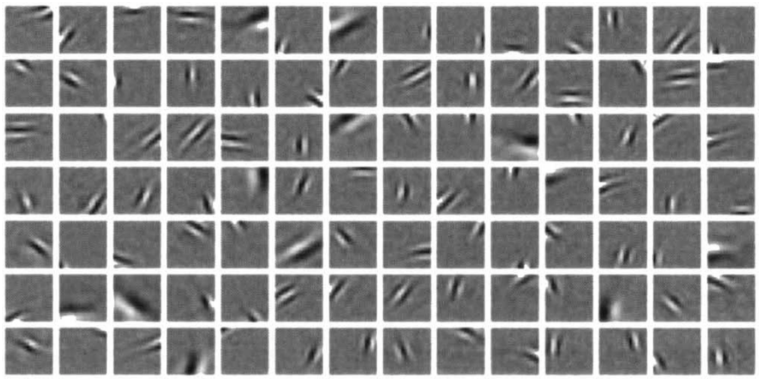
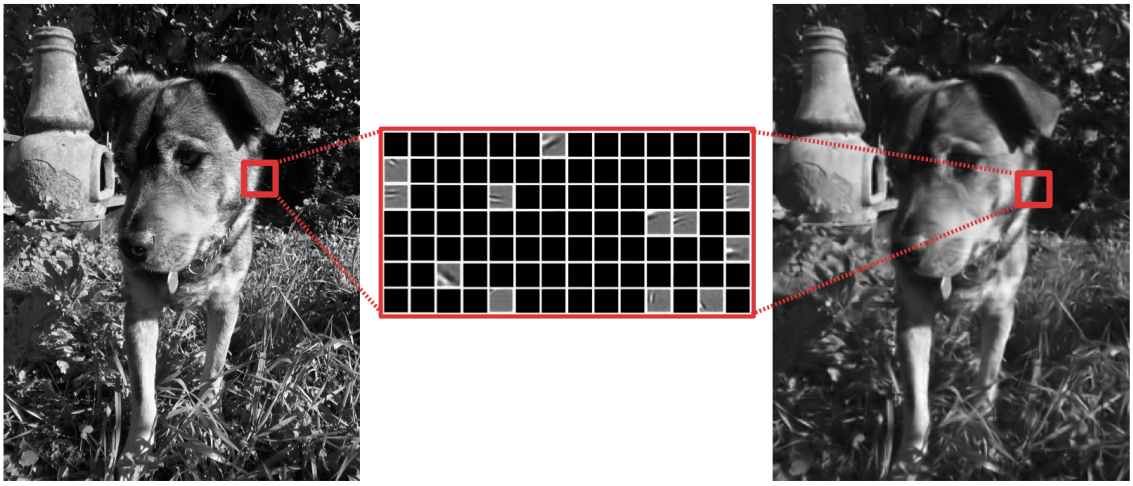


Figure 7.1: Example dictionary of elements used to sparse code images. This dictionary was tuned to enable a sparse encoding of natural images. Taken from Olshausen and Field (1996).

natural image as a combination of only a few elements from this dictionary.

Figure 7.2 shows how an image can be encoded with a small combination of these learned dictionary elements. In this example, each small image patch is encoded independently. For one image patch (A, outlined in red), only a few elements from the dictionary are needed to encode the patch (B). This encoding creates a representation of the original image patch. In Figure 7.2 (C), we see that the encoding *reconstructs* the original image (with some noise introduced in the process). For this reason, the literature often describes the sparse coding of images as "reconstructing an image using a small number of dictionary elements."

For sparse coding to work well, an algorithm clearly must have a particular type of dictionary that allows it to reconstruct the observation (*e.g.*, the image of the dog, in Figure 7.2) using only a few elements of the dictionary. For encoding natural images, a "good dictionary" is usually a collection of edges at various orientations and scales, as shown in Figure 7.1. Learning such a dictionary from a large dataset (of natural images, in this example) is a large task unto itself, and for the remainder of this



(A) (B) (C)

Figure 7.2: Example of sparse coding. Using the dictionary from Figure 7.1 (Olshausen and Field, 1996), an image of a dog is sparse coded. (A) A small patch of the image (red square) can be represented using only a few dictionary elements (B). Applying this sparse coding principle to every image patch yields a "sparse reconstruction" of the image (C).

dissertation I will assume that the dictionary is given. I refer the reader interested in dictionary learning to Engan et al. (1999); Olshausen and Field (1996); Aharon et al. (2006) and Mairal et al. (2009), as I will discuss dictionaries at only a high level. I will focus, instead, on the different tasks for which sparse coding is often used in machine learning and computer vision, once the dictionary is already given. In the following sections, I will dive deeper into the mathematics behind sparse coding, and I will introduce the measures of performance that are relevant to the sparse-coding tasks presented later in this dissertation.

## 7.1 How Sparse Coding Works

Given an *observation* $\mathbf{x} \in \mathbb{R}^n$, one wishes to encode $\mathbf{x}$ as a small linear combination of some collection of known vectors $\phi_i \in \mathbb{R}^n$, for $1 \leq i \leq m$. That is to say, a sparse-coding algorithm searches for the coefficients $z_i \in \mathbb{R}$ such that

$$\mathbf{x} \approx \sum_{i=1}^{m} z_i \phi_i$$

where *most $z_i$ are equal to zero.* The fact that most $z_i$ are zero is analogous to the principle of "few neurons firing for any particular concept," discussed above. In Figure 7.2 (A) the small image patch in red is $\mathbf{x}$; Figure 7.2 (B) shows the dictionary elements $\phi_i$ (where $\phi_i$ is blacked out if $z_i$ is zero); and Figure 7.2 (C) shows the reconstruction $\sum_{i=1}^{m} z_i \phi_i$.

To simplify the notation, define the dictionary $\Phi$ to be the matrix whose columns are the different $\phi_i$,

$$\Phi \stackrel{\texttt{def}}{=} [\phi_1 \ \phi_2 \ \cdots \ \phi_m] .$$

Now I can rewrite the task of sparse coding as the search for the vector $\mathbf{z} = (z_1, \ldots, z_m)$, called the *sparse code*, such that

$$\mathbf{x} \approx \Phi\mathbf{z} \quad \text{and} \quad \|\mathbf{z}\|_0 \leq s \tag{7.1}$$

for some small number $s$ (Tibshirani, 1996). Note that $\|\cdot\|_0$ is the $\ell^0$ penalty function[2], returning the number of nonzero coefficients in its vector input.

From this formulation, it is clear why a field like signal processing would take interest in sparse coding. In particular, if $s < n$, then $\mathbf{z}$ is a compressed representation of the original signal $\mathbf{x}$ (Donoho, 2006). Crucial to this compression is that $\Phi$ be fixed and agreed-upon by all parties (those encoding $\mathbf{x}$ to $\mathbf{z}$, and those decoding $\mathbf{z}$ back to $\mathbf{x}$). Several theoretical guarantees of the quality of the compression have been proven in the case where $\Phi$ is "random" (meaning each entry $\Phi_{i,j}$ is sampled randomly from the standard normal distribution) (Candes, 2008). In particular, it has been shown that having $m \gg n$ (meaning that there are far more dictionary elements than needed to form a basis) allows for a good reconstruction of $\mathbf{x}$ with small $s$ (Olshausen and Field, 1996, 1997), and I shall assume that $m \gg n$ for the remainder of this dissertation. Empirically, learning a task-based dictionary $\Phi$ (meaning, for example, a dictionary that has been tuned to allow for encoding and decoding with few elements, as in Figure 7.1) has been shown to improve the performance of sparse coding for a variety of tasks over random dictionaries (Aharon et al., 2006; Engan et al., 1999; Mairal et al., 2009).

I refer to the calculation of $\Phi\mathbf{z}$ as *reconstruction*, since with the correct choice of $\mathbf{z}$

[2]Some texts refer to this function as a norm, or pseudo-norm, or a "norm" (with quotations); however, it is not truly a norm, nor is it a pseudo-norm, nor a quasi-norm. It is merely a penalty used to regularize the vector $\mathbf{z}$, thus I call it a *penalty function*.

this calculation "reconstructs" the observation $\mathbf{x}$. In many cases, one cannot expect to *perfectly* reconstruct $\mathbf{x}$ when $s$ is very small; instead, one might allow for a small amount of error in the reconstruction, meaning $\|\mathbf{x} - \Phi\mathbf{z}\|_2 \leq \delta$ for some small $\delta > 0$. This changes (7.1) to

$$\|\mathbf{x} - \Phi\mathbf{z}\|_2 \leq \delta \quad \text{and} \quad \|\mathbf{z}\|_0 \leq s. \tag{7.2}$$

However, the algorithm searching for $\mathbf{z}$ might not know the desired values of $\delta$ and $s$ ahead of time. Instead, one may prefer to ask the algorithm to minimize both of these values together, to achieve some balance between sparsity (low $s$) and reconstruction (low $\delta$). In this case, one can formulate the sparse coding problem as

$$\arg\min_{\mathbf{z}} \|\mathbf{x} - \Phi\mathbf{z}\|_2 + \lambda\|\mathbf{z}\|_0. \tag{7.3}$$

where $\lambda > 0$ controls the tradeoff between reconstruction and sparsity. Recall the goal of sparse coding, to "reconstruct $\mathbf{x}$ using few dictionary elements." It is clear how (7.3) formalizes this idea: with an appropriate choice of $\lambda$, the $\lambda\|\mathbf{z}\|_0$ term drives most dimensions of $\mathbf{z}$ to zero, while the $\|\mathbf{x} - \Phi\mathbf{z}\|_2$ term forces $\mathbf{z}$ to reconstruct $\mathbf{x}$ as well as possible.

Because solving Equation (7.3) directly is known to be NP-hard (Natarajan, 1995), some approaches relax the non-convex $\ell^0$ penalty to the convex $\ell^1$ norm (Tibshirani, 1996; Efron et al., 2004). In this case, Equation (7.3) becomes

$$\arg\min_{\mathbf{z}} \|\mathbf{x} - \Phi\mathbf{z}\|_2 + \lambda\|\mathbf{z}\|_1. \tag{7.4}$$

Recall that the $\ell^1$ norm is defined by $\|\mathbf{z}\|_1 = \sum_{i=1}^{n} |z_i|$.

The minimization problem in Equation (7.4) is known as the *lasso* (Tibshirani,

1996), and is numerically solvable using linear programming. Donoho and Elad (2003) showed that, in many cases, solving Equation (7.4) in fact yields the same solution $\mathbf{z}^*$ that solves Equation (7.3). There are a variety of other approaches to sparse coding which replace the $\ell^0$ penalty with any number of $\ell^p$ (quasi-)norms for $0 < p \leq 1$ (Chartrand and Yin, 2008; Boyd et al., 2011; Chartrand and Wohlberg, 2013).

The above formulations of sparse coding are merely different mathematical interpretations of the original principle: a sparse coding algorithm searches for a vector $\mathbf{z}$ which reconstructs $\mathbf{x}$ (as well as possible) using only a few dictionary elements. There are countless algorithms that try to solve the sparse coding problem. I will divide these algorithms into two categories: greedy and non-greedy algorithms. Each of these families will play a part in the upcoming analysis, and both have seen very good success at solving the sparse coding problem.

Examples of greedy sparse coding algorithms include Least Angle Regression (LARS) (Efron et al., 2004) and Matching Pursuit (MP) and its variants (Tropp and Gilbert, 2007; Donoho et al., 2012). These algorithms are characterized by initializing the sparse code $\mathbf{z}$ to be a vector of zeros; at each iteration, the algorithm adds one or more non-zeros to the vector $\mathbf{z}$. The heuristic for choosing which dimension to make nonzero, and the value assigned to the nonzero dimensions, define the algorithm. The defining property of these algorithms is that they never let a nonzero $z_i$ fall back to zero. This is a very restrictive property, forcing the algorithm to be careful when choosing a dimension of $\mathbf{z}$ to become nonzero.

Examples of non-greedy sparse coding algorithms include Subspace Pursuit (SP) (Dai and Milenkovic, 2009), Iterative Hard Thresholding (IHT) and its variants (Blumensath and Davies, 2010, 2009; Blumensath, 2012), and Iterative Soft Thresholding and its variants (Daubechies et al., 2004; Beck and Teboulle, 2009). These algorithms

are characterized by keeping track of an *active set* of nonzero dimensions of $\mathbf{z}$, which can grow *or* shrink at any iteration. Thus, unlike with greedy algorithms, a nonzero $z_i$ at iteration $t$ may become zero at iteration $t+1$ of a non-greedy algorithm. As a vast generalization, this family of algorithms tends to come with stronger guarantees (*i.e.,* non-greedy sparse-coding algorithms tend to come with tighter provable bounds between their results and the solution to Equations (7.3) and (7.4)), but I emphasize that algorithms from *both* families are often used with good success. Note that the Difference Map, which I will apply to sparse coding in Part IV of this dissertation, is a non-greedy algorithm.

## 7.2 Sparse Coding for Reconstruction

In the previous sections, I introduced the principle of sparse coding: to reconstruct the observation $\mathbf{x}$ using few columns of a matrix $\Phi$. Sometimes, $\mathbf{x}$ is a vector representing the pixel values of an image (*i.e.,* the grayscale intensities of an image patch) as in Figure 7.2. Other times, $\mathbf{x}$ might be the features extracted from an image. For example, Yang et al. (2009) set $\mathbf{x}$ equal to the SIFT features of an image (discussed in Section 2.4), and then search for a sparse encoding $\mathbf{z}$ that gives a good reconstruction of the extracted SIFT features.

But what exactly is meant by "a good reconstruction?" To formalize this idea, I will give two common mathematical definitions that are meant to measure reconstruction quality. When a sparse coding algorithm terminates with the solution $\mathbf{z}^*$, define the *reconstruction* or *estimate* by

$$\hat{\mathbf{x}} \stackrel{\texttt{def}}{=} \Phi\mathbf{z}^*.$$

One common measure of reconstruction quality is the normalized root mean squared error (NRMSE), defined by

$$\text{NRMSE}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{\|\mathbf{x} - \hat{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2}.$$

Clearly the NRMSE is minimized when $\hat{\mathbf{x}} = \mathbf{x}$. Thus a lower NRMSE is a better reconstruction of $\mathbf{x}$.

Another measure of reconstruction quality is the signal-to-noise ratio (SNR), defined by

$$\text{SNR}(\mathbf{x}, \hat{\mathbf{x}}) = 10 * \log_{10}\left(\frac{\text{var}_{\mathbf{x}}}{\|\mathbf{x} - \hat{\mathbf{x}}\|_2^2/n}\right)$$

where $\text{var}_{\mathbf{x}}$ is the variance measured over the dimensions of the vector $\mathbf{x}$, and $n$ is the dimensionality of $\mathbf{x}$. Unlike the NRMSE measure, SNR gives a *higher* value for higher-quality approximations. Moreover, SNR is unbounded, whereas NRMSE is bounded below by 0.

Ultimately, both of these measures revolve around the difference between the observation and the approximation, $\mathbf{x} - \hat{\mathbf{x}}$. This difference is typically called the *residual*, and plays an important role in many of the sparse coding algorithms. Given that both NRMSE and SNR appear to be two different ways to measure the residual, one may ask why we need these measures at all. The answer is that the two measures exist because, historically, they are favored by different research communities. In the various experiments reported below, I will sometimes use NRMSE, and sometimes SNR. This is because I will report results that are meant to appeal to a wide range of researchers. As a general rule, I will use SNR when measuring the noise or reconstruction error of image pixels, and when adding noise artificially to make an experiment

more challenging. I use NRMSE when measuring everything else.

In addition to its reconstruction error, a sparse code $\mathbf{z}$ is often measured by its *sparsity*. Recall the neural origins of the sparse coding principle: a concept is encoded by the strong activation of few neurons. Thus the sparsity of a sparse code $\mathbf{z}$ is usually measured with the $\ell_0$ penalty function, where $\|\mathbf{z}\|_0$ indicates the number of nonzero dimensions in $\mathbf{z}$.

## 7.3 Sparse Coding for Classification

In Section 2.5, I briefly mentioned that sparse coding can be used as a type of feature extraction for a classification task. In particular, given a datum $\mathbf{x}$, one might sparse-code $\mathbf{x}$ to find a sparse representation $\mathbf{z}$, and then use $\mathbf{z}$ for training and testing. In the case where $\mathbf{x}$ is already a set of features extracted from the datum (such as SIFT (Lowe, 1999)), then $\mathbf{z}$ would reconstruct those features; one might consider sparse coding a "second feature extraction" in this setting.

In the previous section, I discussed how a sparse code $\mathbf{z}^*$ is often measured by how well it reconstructs the observation $\mathbf{x}$ as well as its sparsity. When the ultimate goal is to train or test a classifier, though, there is an additional important measure to consider: classification error. Classification error is defined by

$$\text{classification error} = 1 - \text{classification accuracy}.$$

Thus higher classification accuracy is the same as lower classification error. Just as I previously measured classification accuracy on a *test* set (see Section 1), I will also measure the classification error achieved by sparse-coded test data in the upcoming

sections.

Previous research has indicated that treating a sparse code $\mathbf{z}$ as features extracted from a datum $\mathbf{x}$ often decreases classification error compared to training and testing with the vector $\mathbf{x}$ directly (Yang et al., 2009; Zeiler et al., 2010, 2011). Researchers explain this phenomenon by noting that $\mathbf{z}$ contains the same information as $\mathbf{x}$ (since $\mathbf{z}$ can be used to reconstruct $\mathbf{x}$); moreover, the constraint that $\mathbf{z}$ be *sparse* is known to be an effective type of regularization[3] to help the classifier generalize to new data (Raina et al., 2007; Bansal et al., 2010; Coates and Ng, 2011). I will refer to this explanation, that *sparse coding reduces classification error because the features reconstruct the data sparsely*, as the **sparse reconstruction hypothesis**. This hypothesis appears to be widely assumed in the literature (Ng, 2004; Raina et al., 2007; Coates and Ng, 2011).

When asking why sparse coding decreases classification error, an alternative to the sparse reconstruction hypothesis is that sparse coding could act as a type of feature *selection* (Lee et al., 2007). Briefly, feature selection is an analysis performed on the features of the training data, but before (or concurrent with) training the classifier. Once features are extracted, some of the features are essentially culled from the dataset altogether, regarded as irrelevant or harmful for generalization. Many different criteria are used for feature selection, including information gain (Guyon and Elisseeff, 2003), $\ell^p$ regularization (Ng, 2004), and heuristic search such as genetic algorithms (Siedlecki and Sklansky, 1989; Yang and Honavar, 1998). A key aspect of feature selection is that the choice of "active" features is guided by the classification error of the resulting features. In this sense, feature selection is quite different than sparse coding, where the sparsity is thought of primarily as a parameter that affects

[3]Loosely speaking, regularization is a way of penalizing overly complex solutions to problems. In this case, "complex" means "too many nonzero dimensions."

reconstruction. Moreover, feature selection will "turn off" the same feature for the entire dataset, whereas sparse coding sets different $z_i$ (and therefore $\phi_i$) to zero for different data. Thus I do not find the link between feature selection and sparse coding to be a strong one, and I will focus the remainder of Part III on evaluating the **sparse reconstruction hypothesis** instead.

# Chapter 8

# Evaluating the Sparse Reconstruction Hypothesis

Recall the **sparse reconstruction hypothesis**, which states that sparse coding can decrease classification error because the sparse codes $\mathbf{z}$ reconstruct the original data (or features) $\mathbf{x}$, and they are sparse. In this section, I investigate whether the **sparse reconstruction hypothesis** adequately explains the decrease in classification error seen with sparse coding. I hypothesize that the sparse reconstruction hypothesis is *not* correct. In particular,

> **My hypothesis** is that there exist some sparse codes that reconstruct a dataset with less error *and* are sparser, but which still lead to poorer classification accuracy than a competing set of sparse codes.

In order to test my hypothesis, I will use sparse coding algorithms to create sparse codes $\mathbf{z}$ with various sparsity levels $\|\mathbf{z}\|_0$. I will measure both the classification error and reconstruction error as a function of the sparsity for a variety of datasets and

sparse coding algorithms. It follows from the **sparse reconstruction hypothesis** that if one set of sparse codes is sparser *and* reconstructs the data with less error, it should achieve lower classification error as well. I hypothesize that this is *not* always true.

## 8.1 Methodology

In order to test my hypothesis, I require a method for classifying data based on their sparse codes. One of the most popular such methods for computer vision comes from Yang et al. (2009), whose method achieves low classification error on a variety of standard computer-vision datasets.

Following Yang et al. (2009), I first convert all images to grayscale. I then extract a dense grid of SIFT features (Lowe, 1999) from $16 \times 16$ pixel patches, tiled over each image with a stride of 8 (meaning two adjacent $16 \times 16$-patches overlap half of each other's pixels). The SIFT features are then sparse coded using a $128 \times 1024$ dictionary $\Phi$ from Yang et al. (2009), which was learned from the Caltech101 dataset (Fei-Fei et al., 2004). To be clear: because the SIFT features are being sparse coded, the sparse codes will sparsely reconstruct the SIFT features (rather than the pixel values of an image). This process (sparse coding SIFT features) has achieved low classification error on a variety of modern datasets (Yang et al., 2009) and includes a sparse-coding process that will allow me to test my hypothesis.

To perform sparse coding, I choose one representative algorithm from both the greedy and non-greedy families described in Section 7.1. From the greedy family algorithms I use LARS (Efron et al., 2004), which adds exactly one non-zero dimension to $\mathbf{z}$ at each iteration; and from the non-greedy family of algorithms I use Subspace

Pursuit (Dai and Milenkovic, 2009) which has strong theoretical guarantees and is also one of the fastest sparse coding methods (the speed of various algorithms is evaluated later in Chapter 10). I evaluate my hypothesis with these two, different algorithms in order to verify that my conclusions are not due to an artifact in one single algorithm.

The sparse codes are used as features to be classified by (or to train) an SVM. The SVM implements the spatial pyramid match (SPM) kernel, as discussed in Section 2.2. Additionally, the same sparse codes are used to reconstruct the original SIFT features using the dictionary $\Phi$. When reconstructing, the reconstruction error is measured as normalized root mean squared error (NRMSE). Recall from Section 7.2 that when trying to reconstruct the signal $\mathbf{x}$ with the sparse code $\mathbf{z}^*$, the reconstruction $\hat{\mathbf{x}} = \Phi\mathbf{z}^*$ has an NRMSE calculated by $\|\mathbf{x} - \hat{\mathbf{x}}\|_2 / \|\mathbf{x}\|_2$. As a rough heuristic guideline, a good NRMSE is $10^{-.1} \approx 0.8$, and a very good NRMSE is $10^{-.4} \approx 0.4$.

## 8.2 Synthetic Datasets

In addition to testing my hypothesis with multiple sparse-coding algorithms, I will test it with multiple datasets. Pinto et al. (2011) created a synthetic dataset with two classes, *cars* and *planes*. Each image contains either a car or a plane, placed over a background of "natural imagery." To vary the difficulty of the task, variation is introduced in the rotation, location and scale of the object (the car or the plane). This variation yields seven datasets, with *pinto_synthetic_0* exhibiting the least amount of variation, and *pinto_synthetic_6* the most. Sample car images from all 7 variation levels are shown in Figure 8.1. These 7 variation levels allow the researcher to test the robustness of their computer vision system by varying the difficulty of the

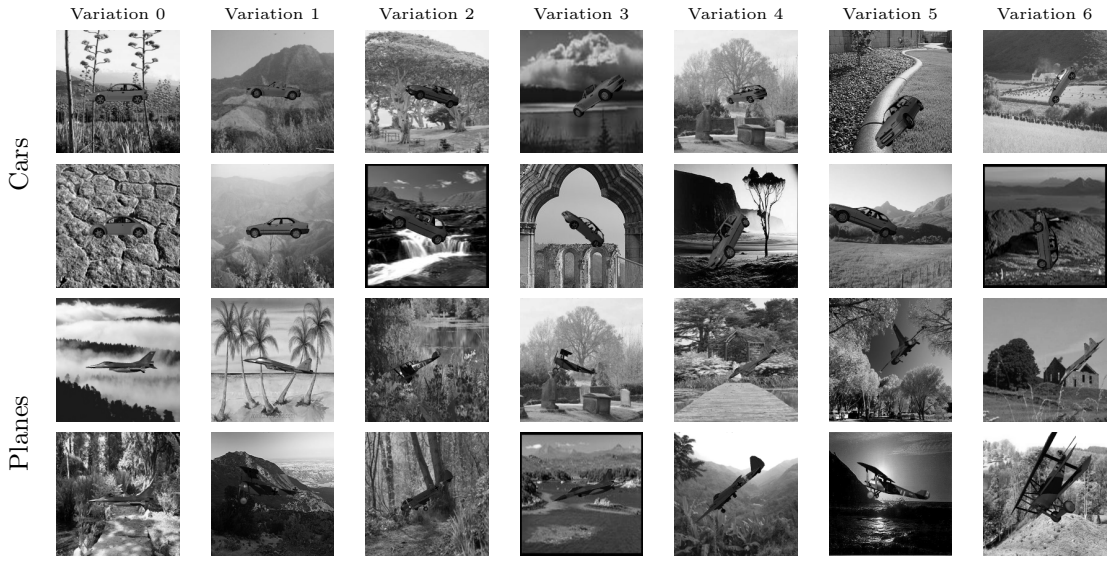



Figure 8.1: Examples from the images of the `pinto_synthetic` datasets (Pinto et al., 2011). The higher variation numbers contain images where the distribution of the object's location, scale and rotation have larger variance.

classification task. The random backgrounds ensure that no spurious correlations exist between the image background and the object to be identified: if an algorithm performs well on these datasets, it can only be because it has identified the features belonging to the object itself.

There are 130 images per class, per variation. I extract SIFT features from each image, and then sparse-code the SIFT features. Using the sparse codes, I train an SPM-kernel SVM with 100 images per class, and test on the remaining 30, for ten splits of cross validation. I measure both the test classification error and the reconstruction error of the test data for each variation level independently.

For this first experiment, I perform sparse coding with Subspace Pursuit (SP) (Dai and Milenkovic, 2009) because it is known to converge to a good reconstruction

extremely quickly for problems of this size[1], and it allows the user the dictate the sparsity of the returned vector, $\mathbf{z}$. This latter fact, that the user controls the sparsity of $\mathbf{z}$, lets me measure how reconstruction error and classification error vary in relation to the common parameter of sparsity. I vary the sparsity logarithmically between 1 and 32. The results are plotted in Figure 8.2.

There are several trends in Figure 8.2. First, the reconstruction error (green line) decreases with less sparsity in $\mathbf{z}$ (meaning larger $\|\mathbf{z}\|_0$, or more non-zeros in $\mathbf{z}$). To see this, note that the green line decreases monotonically in all plots. This makes sense: when reconstructing a signal $\mathbf{x}$, it is generally helpful to use more columns of $\Phi$. Moreover, this trend is true in *all* variation levels: the green curve looks roughly identical in all plots in Figure 8.2. This also makes sense, since the variation in object appearance (Figure 8.1) was designed to make the objects harder to recognize, not to make the images more difficult to compress.

Turning our attention to the classification error (blue lines in Figure 8.2), it is clear that sparse coding does indeed reduce classification error: the solid blue lines (classification error with sparse codes $\mathbf{z}$) are lower than the dashed blue lines (classification error with raw SIFT features $\mathbf{x}$). It is also clear that object variation has a profound effect on classification error, as it was designed to: the blue lines are, overall, higher in the plots with higher variation numbers. Thus classification error and reconstruction error respond very differently to object variation.

Moreover, the two measures also respond differently to the sparsity level $\|\mathbf{z}\|_0$. While reconstruction error decreases with larger $\|\mathbf{z}\|_0$, classification error appears to be lowest with a medium amount of sparsity. It is unclear whether classification error

[1]The code for Subspace Pursuit came directly from the website of the authors, Dai and Milenkovic (2009). I give a thorough comparison between a representative sample of state-of-the-art sparse coding methods in Chapters 11 and 12.

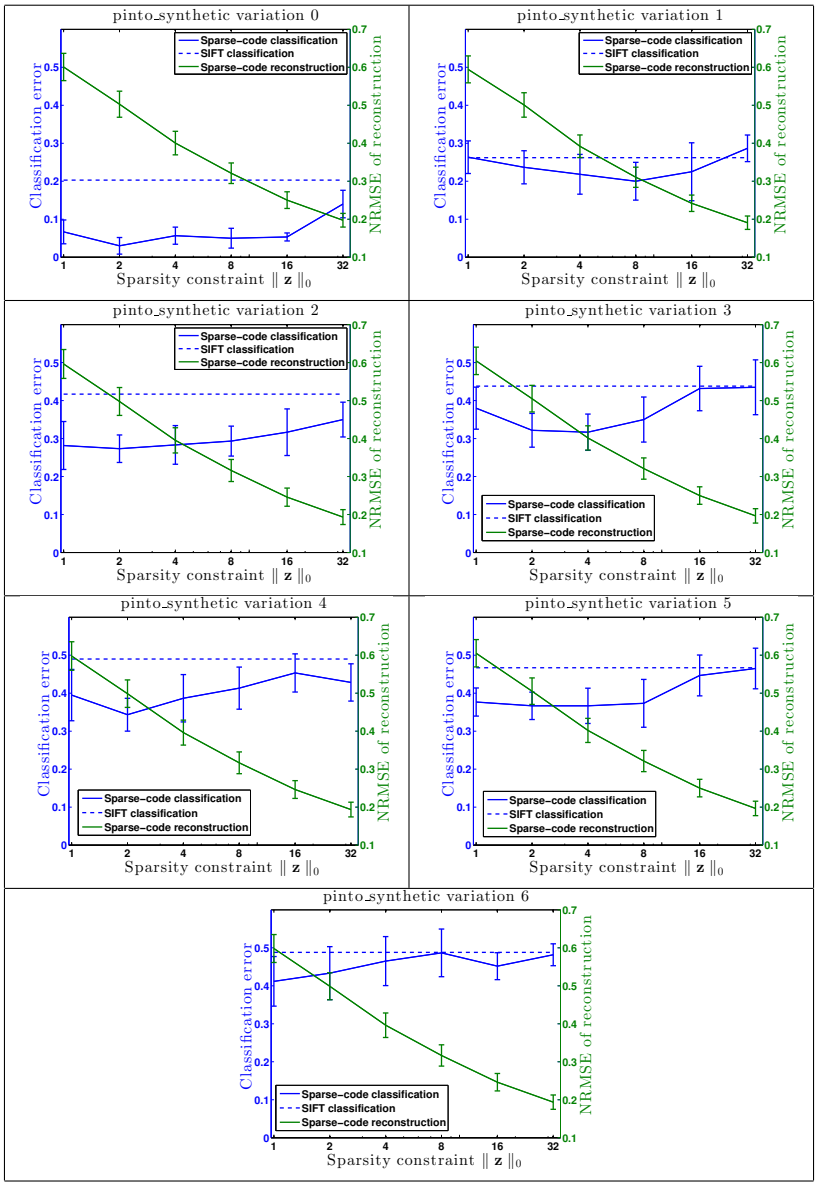



Figure 8.2: Results from sparse-coding the *pinto_synthetic* datasets with Subspace Pursuit. Variation number indicates the variation level of each dataset. SIFT codes are extracted from each image, and then sparse-coded. The reconstruction error of the sparse codes are shown in green; the classification error of the sparse codes in solid blue; and the classification error of the (raw, not sparse coded) SIFT features in dashed blue. Classification error is strongly affected by variation in the dataset, whereas reconstruction error is affected only by the sparsity $\|\mathbf{z}\|_0$. The higher the variation level (of the *pinto_synthetic* dataset), the more variation in the objects in the images, and therefore the more challenging the classification problem. The variation number was described in Figure 8.1.

is minimized with $\|\mathbf{z}\|_0 = 2$, 4, 8, or 16, due to the variance between cross-validation trials; however, the general trend is that the classification error at $\|\mathbf{z}\|_0 = 1$ and $\|\mathbf{z}\|_0 = 32$ are higher than the middle values[2]. In fact, we will see more evidence of this trend momentarily, with other sparse coding algorithms and other datasets.

Next, I perform the same analysis using a greedy sparse-coding algorithm to verify that the results generalize. From the set of greedy algorithms I chose LARS (Efron et al., 2004). Like SP, LARS allows the user to dictate the sparsity level $\|\mathbf{z}\|_0$. I vary the sparsity logarithmically between 2 and 64, for all variation levels of the same dataset. The results are plotted in Figure 8.3.

Comparing Figures 8.3 and 8.2, one can immediately notice some similar trends. Sparse coding still decreases classification error compared to the raw SIFT features. Classification error is strongly affected by the object variation level, but reconstruction error is not. And, as was the case with SP, increasing $\|\mathbf{z}\|_0$ decreases the reconstruction error monotonically (note the green lines moving down and to the right in all plots of Figure 8.3). We also see the same "Goldilocks zone" for classification error[3] as a function of $\|\mathbf{z}\|_0$. In particular, the lowest classification error is usually achieved by $\|\mathbf{z}\|_0 = 8$ or 16, and increases on either side of these values[4].

The evidence so far indicates that sparse coding can decrease classification error. This trend appears to be true for almost all sparsity levels, and for a variety of difficulties for the classification task. Note that choosing the sparsity $\|\mathbf{z}\|_0$ that yields

[2]The greatest exception appears to be variation level 6, which is not terribly informative since the classifier appears to be guessing at this variation level (note that the classification error is about 0.5).

[3]By "Goldilocks zone," I only mean that the classification error is minimized in the middle of the plots. That is to say, the classification error plots are roughly U-shaped.

[4]Again, there is an exception with variation 6, but the fact that the classifier is essentially guessing (classification error close to 50%) means that the classification error levels here are not very informative.

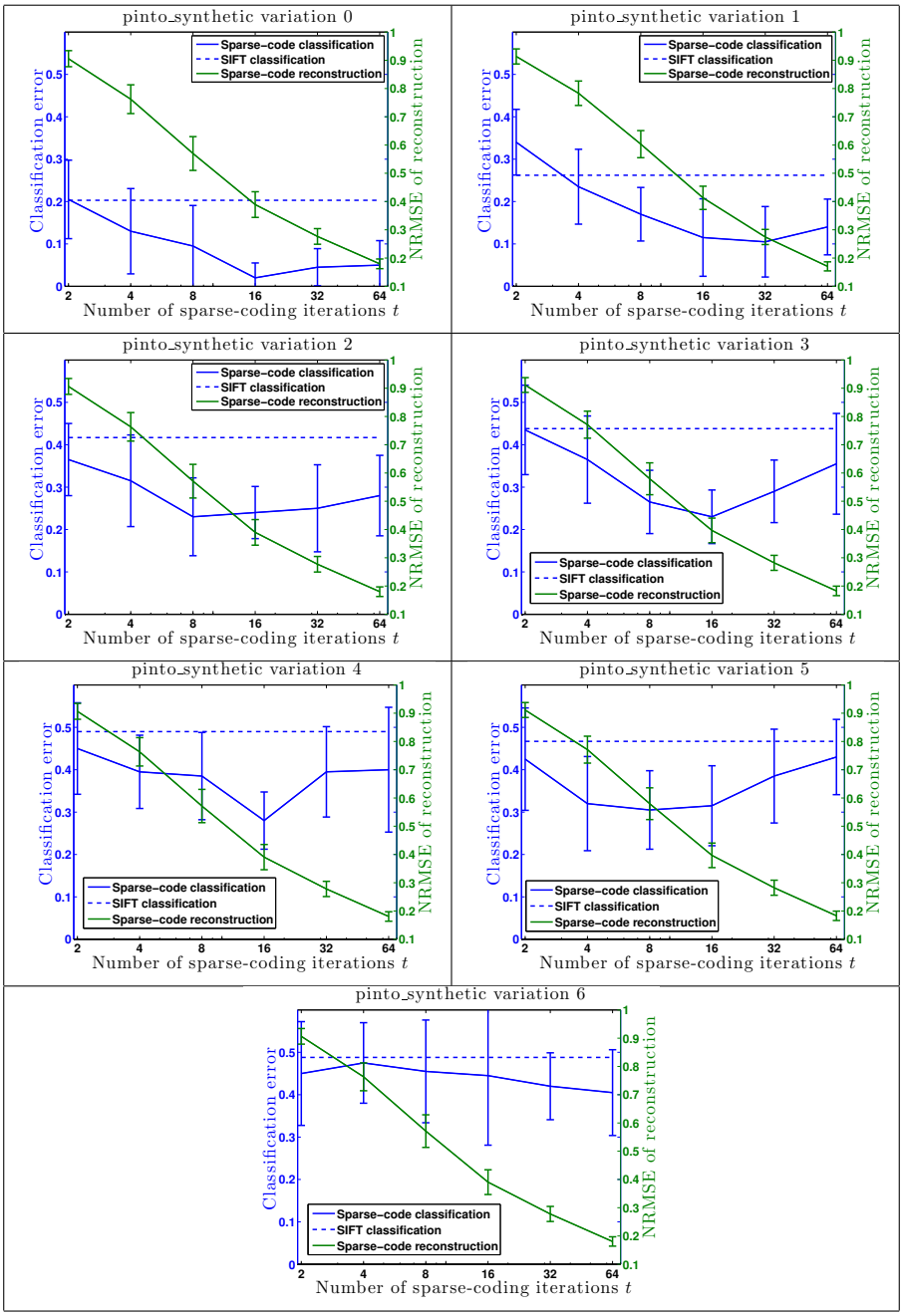



Figure 8.3: Sparse coding the `pinto_synthetic` dataset with LARS. Reconstruction error decreases monotonically with more nonzero coefficients in $\mathbf{z}$. However, the classification error increases at the largest values of $\|\mathbf{z}\|_0$. For all but the highest variation level, classification error appears to be U-shaped, though the large error bars diminish the significance of this trend.

the lowest reconstruction error leads to sub-optimal classification error. The power of this evidence is diminished by the size of the error bars on all blue lines (which is due to the small size of the test set), making it difficult to conclude anything for certain. But the trend appears to be that we should not tune our sparse-coding parameters solely to minimize reconstruction, if our ultimate goal is classification. That is to say, the success of sparse coding must not solely be due to the code's ability to reconstruct.

Recall the **sparse reconstruction hypothesis**, which states that sparse coding decreases classification error because the codes reconstruct the data *and* they are sparse. Already, though, we see the connection between sparse reconstruction and classification error begin to erode. In particular, note how the reconstruction error follows roughly the same curve in all plots in Figure 8.2 and 8.3, while the classification error responds dramatically to the variation level of the dataset. It is clear that for both the SP and LARS algorithms, sparse reconstruction performance is nearly identical for all variation levels of the *pinto_synthetic* data. Classification error, on the other hand, is not.

However, this alone does not prove **my hypothesis**. I hypothesized that there exist some sparse codes that reconstruct a dataset with less error *and* are sparser, but which still lead to poorer classification accuracy than a competing set of sparse codes. Due largely to the size of the error bars on the above plots, we cannot evaluate my hypothesis here. To evaluate my hypothesis, we need datasets with more images, which will decrease the size of the error bars.

## 8.3 Natural Datasets

In this section, I will describe results from the same experiments using a variety of natural images[5]. In particular, these experiments apply the sparse-code classification methodology to the *Caltech101* (Fei-Fei et al., 2004), *graz02* (Marszatek and Schmid, 2007), and *scenes* (Lazebnik et al., 2006) datasets. Unlike the synthetic data from Section 8.2, there is no natural way to smoothly change the difficulty of these datasets. However, by using multiple independent datasets, we can be more confident in persistent patterns in the experimental results.

These three datasets are each chosen to be challenging in different ways. *Caltech101* (Fei-Fei et al., 2004) contains images of objects which have each been cropped and rotated so that the object is centered and oriented in a consistent way throughout the dataset. Example images from Caltech101 are shown in Figure 8.4. The consistent object appearance makes recognition easier. However, there are 100 object categories in the dataset (and one "background" category); such a large number of categories makes the task more difficult.

The *graz02* dataset (Marszatek and Schmid, 2007) contains only three categories (*bikes, cars,* and *people*). However, the appearance of the objects varies widely within each class, as can be seen in the examples given in Figure 8.5. This large variance in object appearance, which makes for a much more challenging task, leads researchers to often use this dataset for object detection, rather than image classification. The object detection task is to locate the object in each image; this contrasts with image classification, which asks only if a given object is present somewhere in the image. I mention this only as a testament to the difficulty of the dataset — I will still use

[5]In this context, I use the term *natural images* to denote images that were not computer-generated.

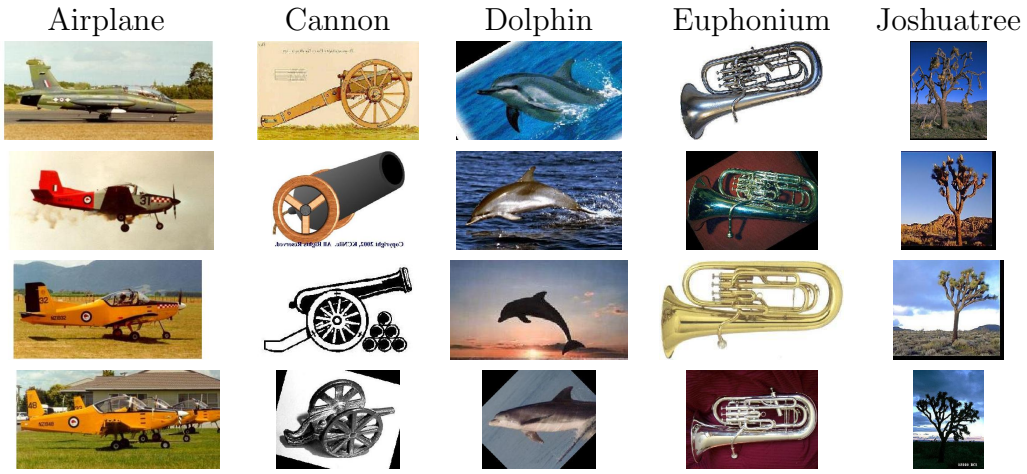



Figure 8.4: Example images from 5 of the 101 categories in the *Caltech101* dataset. The task is made simpler by the fact that the objects appear at a consistent scale and orientation in the center of the image. However, it is made more challenging by the large number of categories.

this dataset (as well as all others) for the image classification task, as I will the other datasets.

Finally, the *scenes* dataset (Lazebnik et al., 2006) does not contain "objects" in the way that the previous datasets do. Instead, each image contains one of fifteen different scenes to be recognized based on the overall structure of the picture. The fifteen scenes include *bedroom, suburb, industrial, kitchen, living room, coast, forest, highway, inside-city, mountain, open-country, street, tall-building, office,* and *store*, examples of which can be seen in Figure 8.6. This is a different type of challenge than the previous datasets, in that the category cannot necessarily be predicted based on the presence of any particular object.

Because *Caltech101* contains 101 categories, the *guessing error rate* (the classification error achieved by random guessing) is more than 99%. The state of the art performance on this dataset was recently achieved by Zeiler and Fergus (2013), with

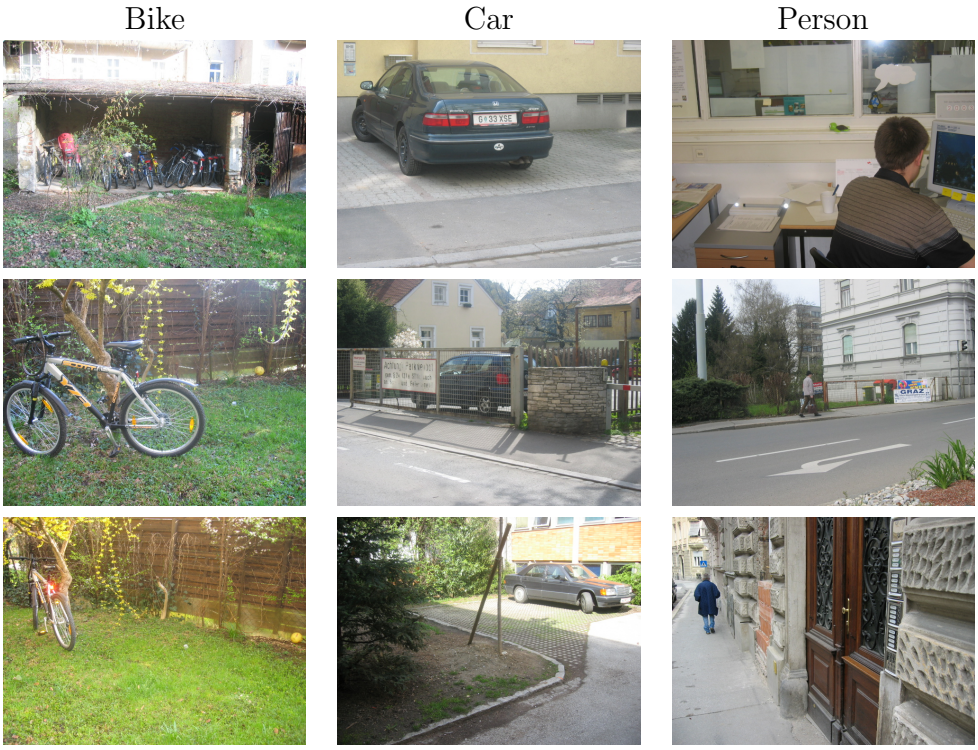



Figure 8.5: Example images from the *graz02* dataset. This dataset is typically used for object *detection* (locating the object in the image), rather than image classification (deciding whether the image contains the object). As a result of the high variance in object appearance, this is a very challenging dataset for object recognition.

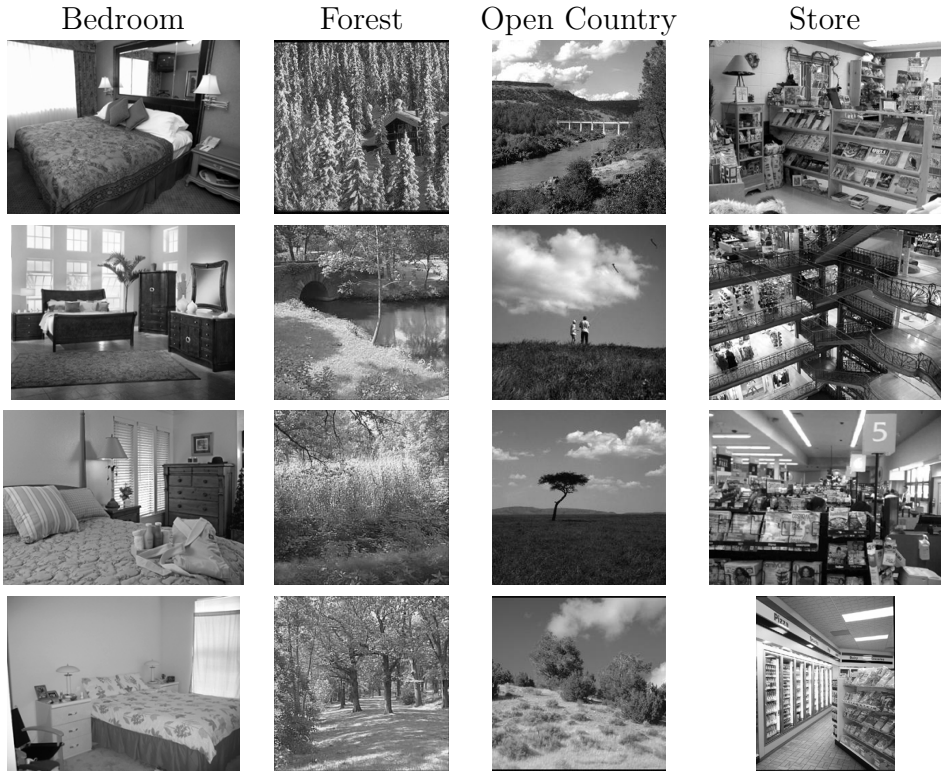



Figure 8.6: Example images from the *scenes* dataset, containing the 15 categories *bedroom, suburb, industrial, kitchen, living room, coast, forest, highway, inside-city, mountain, open-country, street, tall-building, office,* and *store*. Compared to the previous datasets, this task does not rely on the presence or absence of a single object (*e.g.,* a car or an airplane); instead, the whole image is used to convey the scene.

13.5% classification error. Because the *scenes* dataset contains 15 categories, the guessing error rate is approximately 92.3%, whereas Lazebnik et al. (2006) report 18.6% error on this dataset. The *graz02* dataset contains 3 categories, leading to a 66.6% guessing error rate, whereas Marszatek and Schmid (2007) achieve approximately 10% error.

Figure 8.7 gives the results of my sparse-coding experiments on these datasets. For all three datasets, it can be seen that the reconstruction error decreases monotonically as more nonzero coefficients are allowed in the sparse codes ($\|\mathbf{z}\|_0$), as one would expect.Moreover, on all plots, sparse coding is seen to decrease classification error compared with the raw SIFT features: note that the solid blue lines (the classification error of sparse codes) are lower than the dashed blue lines (the classification error of raw SIFT features) in almost every case. More interestingly, we see again that classification error find its minimum with a moderate value of $\|\mathbf{z}\|_0$. This "Goldilocks zone" of minimal classification error occurs around $\|\mathbf{z}\|_0 = 2$ or 4 for SP, and $\|\mathbf{z}\|_0 = 8$ or 16 for LARS, depending on the dataset.

Recall once more the **sparse reconstruction hypothesis**: sparse coding decreases classification error because sparsity and reconstruction are both helpful for classification. At first glance, the results presented in Figure 8.7 agree with this reasoning, in that we see the lowest classification error in the middle of the plots (the valley in the blue solid lines). Therefore, we might conclude, on either side of this valley we must be violating either the need for sparsity (to the right of the valley), or the need for reconstruction (to the left of the valley).

A more interesting story emerges when I compare the lowest classification error of each algorithm (SP and LARS) on each dataset. In Table 8.1, I show the sparsity and reconstruction error associated with lowest classification error achieved by each

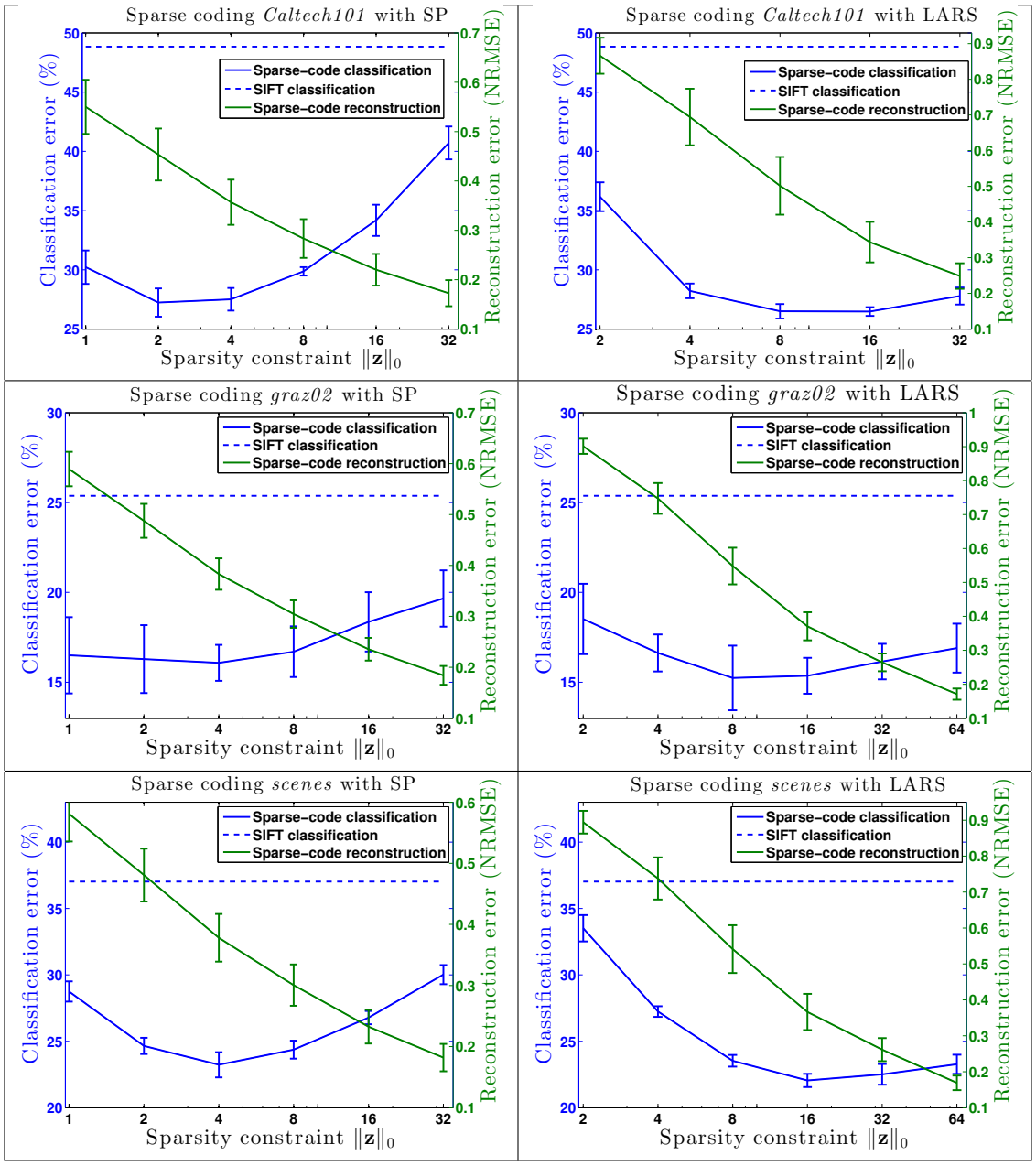



Figure 8.7: Classification error and reconstruction error react differently to the sparsity parameter $\|\mathbf{z}\|_0$. Reconstruction error always decreases with larger $\|\mathbf{z}\|_0$; classification error, on the other hand, is minimized with $\|\mathbf{z}\|_0 = 8$ or 16 for LARS, and 2 or 4 for SP.

algorithm. Note that the sparsity level $\|\mathbf{z}\|_0$ that yields the lowest classification error is different for each algorithm: SP achieves its lowest classification error with a sparsity of 2 or 4, whereas LARS achieves its lowest classification error with a sparsity of 8 or 16. At the indicated level of sparsity (which minimizes the classification error), I also show the algorithm's corresponding reconstruction error. In all cases the sparse codes coming from SP are sparser *and* have a lower reconstruction error. The **sparse reconstruction hypothesis** would imply, then, that the sparse codes from SP should also have lower classification error. They do not. *On each dataset, LARS achieves lower classification error despite the fact that SP generates codes which are sparser and which reconstruct better.* The strength of this statement is diminished by the fact that, in several cases, each algorithm's reconstruction error and classification error are within each other's standard deviation. Nonetheless, the trend is present in the data, and casts doubt on the **sparse reconstruction hypothesis**.

My hypothesis is that there exist some sparse codes that reconstruct a dataset with less error *and* are sparser, but which still lead to poorer classification accuracy than a competing set of sparse codes. Having tested two very popular sparse coding algorithms on a variety different datasets, I have shown that over a variety of datasets, the lowest classification error achieved by SP is with codes that are sparse and reconstruct better than those of LARS; and yet LARS achieves even lower classification error. This confirms my hypothesis, and casts suspicion on the **sparse reconstruction hypothesis**. Of course, there is always room to further test my hypothesis by changing other variables (*e.g.*, the size of the dictionary $\Phi$, methods for learning $\Phi$, different kernels and classifiers). Performing a comprehensive analysis over all possible variations would be very difficult and is beyond the scope of this dissertation.

Table 8.1: Improving sparse reconstruction does *not* improve classification error. For each of the datasets examined, the table gives the sparsity and reconstruction error associated with the lowest classification error achieved by each sparse coding algorithm. In each case, LARS has the lowest classification error (in boldface), but SP gives sparser (in boldface) codes with lower reconstruction error (in boldface). Each experiment was repeated ten times; here I report the mean (with standard deviation in parentheses).

Best performance on *Caltech101*

| Algorithm: | $\|\mathbf{z}\|_0$ | Reconstruction error: | Classification error: |
|---|---|---|---|
| SP | **2** | **0.4536** (0.0527) | 27.24% (1.19%) |
| LARS | 8 | 0.5015 (0.0808) | **26.50%** (0.49%) |

Best performance on *graz02*

| Algorithm: | $\|\mathbf{z}\|_0$ | Reconstruction error: | Classification error: |
|---|---|---|---|
| SP | **4** | **0.3831** (0.0308) | 16.08% (1.00%) |
| LARS | 8 | 0.5483 (0.0541) | **15.24**% (1.80%) |

Best performance on *scenes*

| Algorithm: | $\|\mathbf{z}\|_0$ | Reconstruction error: | Classification error: |
|---|---|---|---|
| SP | **4** | **0.3006** (0.0338) | 24.37% (0.68%) |
| LARS | 16 | 0.3664 (0.0503) | **22.05%** (0.50%) |

Note that I am not the first to question how or why to use sparse coding for classification. Many algorithms have been proposed which each advocate for different methods when sparse coding (Yang et al., 2009; Lee et al., 2007; Gao et al., 2010) . More than finding yet another sparse coding algorithm and parameter choice heuristic, Coates and Ng (2011) question whether sparse coding is the right way to encode data at all: they consider alternative ways to encode a signal $\mathbf{x}$ with a matrix $\Phi$ such as k-nearest-neighbors (where $z_i = 1$ if the column of $\phi_i$ is one of the $k$ closest columns to $\mathbf{x}$, and $z_i = 0$ otherwise), and the simple projection $\Phi^\top \mathbf{x}$ (with small values set to zero). Coates and Ng find many of these alternate encoding methods to yield classification results that are competitive with sparse coding. Moreover, many of these alternate encodings are orders of magnitude faster than sparse coding.

If one observes the same decrease in classification error with faster methods, this begs the question, *why is there any interest in sparse coding at all?* Part III of my dissertation has shown that we do not fully understand why sparse coding often decreases classification error when compared with raw features. However, just because we do not fully understand this phenomenon does not take away from the utility of the phenomenon itself: sparse coding *does* often decrease the classification error, and is therefore a useful tool for machine-learning practitioners. Moreover, as I shall discuss in Part IV, classification is only one of many uses for sparse coding. In the remainder of this dissertation, I will introduce some other popular uses for sparse coding, as well as a new sparse-coding algorithm that excels at these challenging tasks.

# Part IV

# An Improved Sparse Coding Algorithm

In Part III, I showed that sparse coding can decrease classification error, but that the common explanation for this phenomenon does not appear to be true. Classification, however, is only one of many uses for sparse coding. Sparse coding (and the closely-related problem of compressed sensing, to be discussed shortly) has given rise to algorithms that are regularly used for a wide variety of tasks such as image compression and denoising (Elad and Aharon, 2006), dictionary learning (Mairal et al., 2009), regression (Tibshirani, 1996; Efron et al., 2004), and medical imaging (Lustig et al., 2008).

I also showed in Part III that the different algorithms that perform sparse coding can produce very different results. This turns out to be especially true for larger values of $\|\mathbf{z}\|_0$, in which case the NP-hardness of Equation 7.3 requires that algorithms make strong assumptions in order to converge in a reasonable amount of time. The different assumptions made by each algorithm can, in turn, yield very different approximations of the original problem. This is a challenge for the above mentioned fields (image denoising, medical imaging, etc.), where sometimes $\|\mathbf{z}\|_0$ is large and no known algorithm consistently offers a "good" solution $\mathbf{z}^*$.

In Part IV of my dissertation, I will introduce a new sparse-coding algorithm that excels with larger values of $\|\mathbf{z}\|_0$. This algorithm is based on the Difference Map, which was pioneered by Elser et al. (2007) for a variety of NP-hard optimization problems. However, I am the first to successfully develop a sparse-coding algorithm based on the Difference Map[6]. In order to describe the Difference Map, I will begin by introducing the *constraint intersection problem,* which I will use as a new framework

[6]Qiu and Dogandžić (2011) used the Difference Map when solving a problem closely related to sparse coding, though I became aware of this work only after completing my research. Moreover, the code supplied by Qiu and Dogandžić did not perform competitively with any of the algorithms discussed in the following experiments, as I will explain in Section 10.1.

for sparse coding. I will then describe my new algorithm, and I will show how it outperforms the state-of-the-art sparse-coding algorithms for a variety of tasks.

# Chapter 9

# Background: The Constraint Intersection Problem

The Difference Map was developed to solve the constraint intersection problem (Elser et al., 2007). I will review this problem here, and discuss how sparse coding can be cast in this new setting.

Given sets $A$ and $B$ and distance-minimizing projections $P_A$ and $P_B$[1], respectively, the *constraint intersection problem* is to find a point $\mathbf{z}^* \in A \cap B$. In sparse coding, for example, we might consider the two constraints of sparsity ($A$) and good reconstruction ($B$). Clearly, a point $\mathbf{z}^*$ that obeys both of these constraints is also a good sparse code for the purpose of reconstruction. Let us gain some intuition around the general constraint intersection problem before diving deeper into its application to sparse coding.

Figure 9.1 shows two sets $A$ and $B$. At the top of the figure is a point $\mathbf{z}$ and

[1]By distance-minimizing projection, I mean that $P_A(\mathbf{z}_0) = \arg\min_{\mathbf{z}} \|\mathbf{z}_0 - \mathbf{z}\|_2$ subject to $\mathbf{z} \in A$, and likewise for $P_B$.

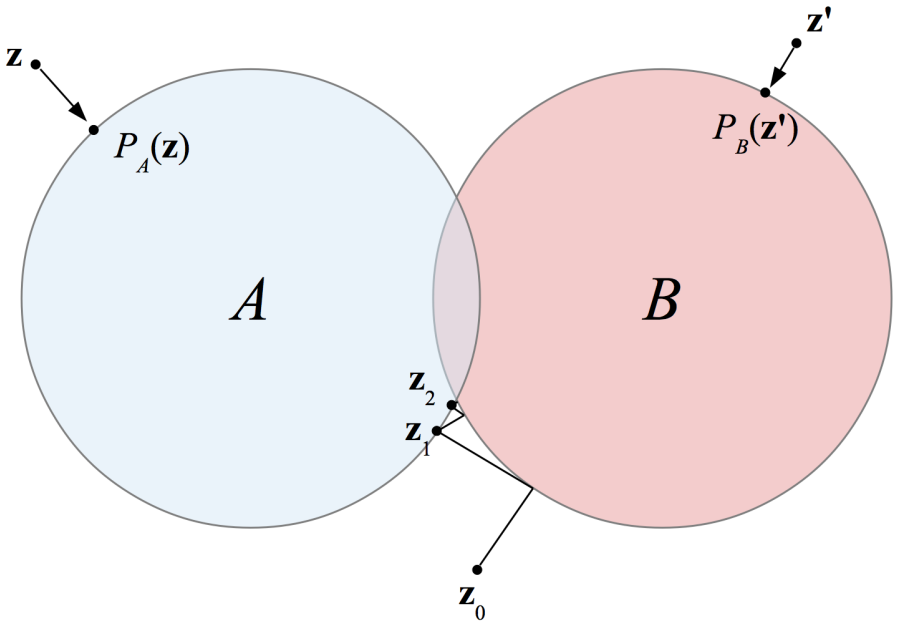



Figure 9.1: Minimum-distance projections and the constraint intersection problem. Top of figure: two points, $\mathbf{z}$ and $\mathbf{z}'$, and their minimum-distance projections, $P_A(\mathbf{z})$ and $P_B(\mathbf{z}')$. Bottom of figure: the Alternating Map (AM), defined by $\mathbf{z}_{t+1} \leftarrow P_A(P_B(\mathbf{z}_t))$, which is guaranteed to converge to a point $\mathbf{z}^* \in A \cap B$ if both $A$ and $B$ are convex.

its minimum-distance projection onto $A$, denoted $P_A(\mathbf{z})$. Similarly, the point $\mathbf{z}'$ is shown along with its minimum-distance projection onto $B$, denoted $P_B(\mathbf{z}')$. When $A$ and $B$ are well-behaved convex sets (as in the figure), these minimum-distance projections can be simple to derive and compute. For less well-behaved sets, defining the projections can be a difficult problem unto itself. As stated earlier, the goal of the constraint intersection problem is to find a point $\mathbf{z}^* \in A \cap B$. Elser et al. (2007) showed that many interesting problems can be cast in this framework, including $k$-SAT, protein folding, and optical reconstruction.

A good first attempt at solving this problem might be the Alternating Map (AM), defined by

$$\mathbf{z}_{t+1} \leftarrow P_A(P_B(\mathbf{z}_t)).$$

Thus as $t$ increases, AM projects back and forth between the two sets. This algorithm is illustrated at the bottom of Figure 9.1, where we see an initial guess $\mathbf{z}_0$ bouncing back and forth between the two sets until it finds a point in their intersection[2].

AM is guaranteed to solve the constraint-intersection problem (meaning the sequence defined by $\mathbf{z}_{t+1} \leftarrow P_A(P_B(\mathbf{z}_t))$ converges to a point $\mathbf{z}^* \in A \cap B$) if $A$ and $B$ are both convex. (Recall that a set $A$ is convex iff, for any $a, a' \in A$, we also have $ta + (1-t)a' \in A$ for all $0 \leq t \leq 1$.) However, if either $A$ or $B$ is not convex, AM may get stuck in a local minimum, as illustrated in Figure 9.2. It turns out that one of the core constraints in sparse coding is not convex, and thus I require a more sophisticated approach than AM in order to solve the sparse-coding problem in this framework.

[2]Though in practice, one might stop when $\|\mathbf{z}_{t+1} - \mathbf{z}_t\|$ becomes sufficiently small.

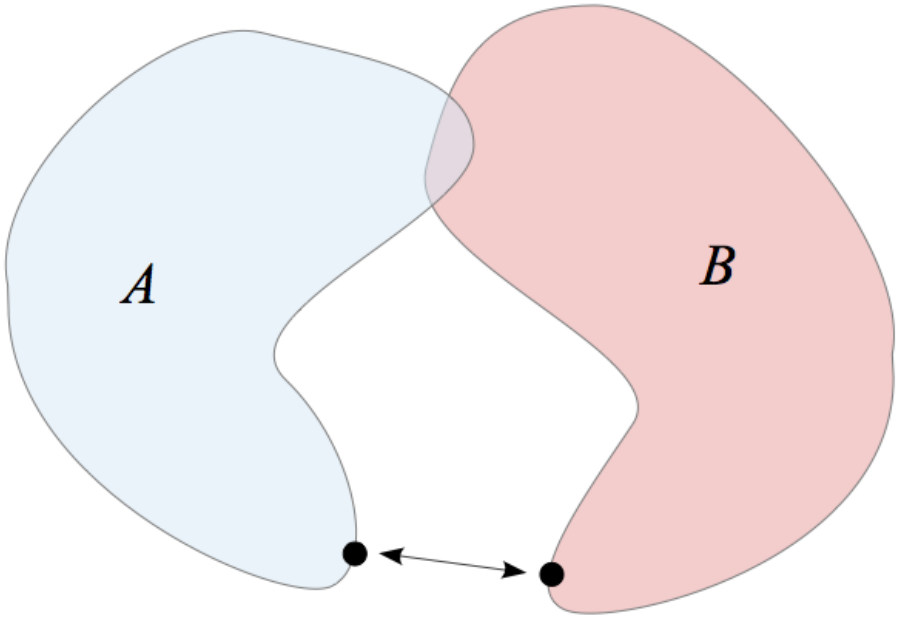



Figure 9.2: When either constraint is not convex, the Alternating Map (AM) will not necessarily solve the constraint-intersection problem. The arrows show a fixed point (meaning AM has converged to a point $\mathbf{z}^* = P_A(P_B(\mathbf{z}^*))$ ) which is not a solution (meaning $\mathbf{z}^* \notin A \cap B$).

# Chapter 10

# The Difference Map for Sparse Coding

The Difference Map (Elser et al., 2007) is a general method for solving the constraint-intersection problem. It was originally applied to a wide variety of NP-hard optimization problems such as protein-folding, 2D and 3D packing problems, and optical reconstruction. Importantly, Elser et al. did *not* apply the Difference Map to sparse coding. In this section, I will introduce the fundamentals of the Difference Map (DM), and I will show how DM can be used for sparse coding[1].

One iteration of DM is defined by $\mathbf{z} \leftarrow D(\mathbf{z})$, where

$$D(\mathbf{z}) \stackrel{\texttt{def}}{=} \mathbf{z} + \beta \left[ P_A \circ f_B(\mathbf{z}) - P_B \circ f_A(\mathbf{z}) \right] \tag{10.1}$$

[1]A large portion of the research presented in Chapters 10 through 12.2 appeared in Landecker et al. (2014).

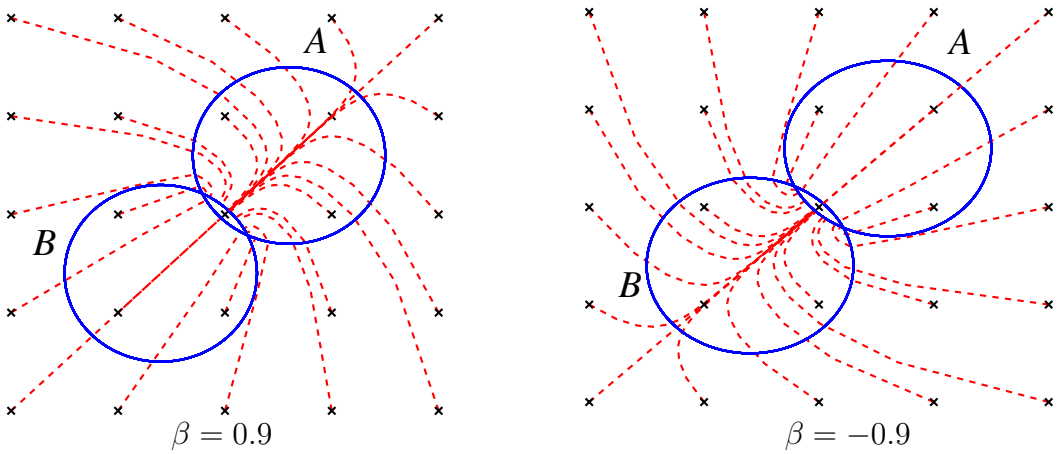



Figure 10.1: DM solving the constraint-intersection problem. Starting with a $5 \times 5$ grid of points (black x's), each point iterates with $\mathbf{x} \leftarrow DM(\mathbf{x})$ until it reaches the intersection of $A$ and $B$. DM uses the parameter $\beta = 0.9$ (left), and $\beta = -0.9$ (right).

and

$$f_A(\mathbf{z}) \stackrel{\text{def}}{=} P_A(\mathbf{z}) - (P_A(\mathbf{z}) - \mathbf{z})/\beta$$
$$f_B(\mathbf{z}) \stackrel{\text{def}}{=} P_B(\mathbf{z}) + (P_B(\mathbf{z}) - \mathbf{z})/\beta$$

and $\beta \in \mathbb{R}, \beta \neq 0$. One can test for convergence by monitoring the value

$$|P_A \circ f_B(\mathbf{z}) - P_B \circ f_A(\mathbf{z})|,$$

which vanishes when a solution is found[2].

Let us recall the mathematical formulation of sparse coding. Given a matrix $\Phi \in \mathbb{R}^{n \times m}$ (where $n < m$) and a datum $\mathbf{x} \in \mathbb{R}^n$, I wish to find a sparse vector

[2]The fact that this value vanishes is clear when one plugs $\mathbf{z}^* = D(\mathbf{z}^*)$ into Equation (10.1).

$\mathbf{z}^* \in \mathbb{R}^m$ that reconstructs $\mathbf{x}$ and is sparse, meaning

$$\mathbf{x} = \Phi\mathbf{z} \quad \text{and} \quad \|\mathbf{z}\|_0 \leq s$$

for some positive integer $s$. I apply DM to this problem by defining the constraint sets

$$A = \{\mathbf{z} \in \mathbb{R}^m : \|\mathbf{z}\|_0 \leq s\},$$
$$B = \{\mathbf{z} \in \mathbb{R}^m : \Phi\mathbf{z} = \mathbf{x}\},$$

In order to use DM for sparse coding, I need only to derive the minimum-distance projections $P_A$ and $P_B$ onto the sets $A$ and $B$, respectively. These definitions, along with Equation (10.1), will fully define how one can solve the sparse coding problem with DM.

The minimum-distance projection onto $A = \{\mathbf{z} \in \mathbb{R}^m : \|\mathbf{z}\|_0 \leq s\}$ is known as *hard thresholding*, and is defined by

$$P_A(\mathbf{z}) = [\mathbf{z}]_s, \tag{10.2}$$

where $[\mathbf{z}]_s$ is obtained by setting to zero the $m - s$ dimensions of $\mathbf{z}$ having the smallest absolute values (where $m$ is the dimensionality of $\mathbf{z}$). This is demonstrated in Figure 10.2.

The minimum-distance projection onto $B$ is given by the formula

$$P_B(\mathbf{z}) = \mathbf{z} - \Phi^+(\Phi\mathbf{z} - \mathbf{x}), \tag{10.3}$$

$$\mathbf{x} = \begin{bmatrix} 1.3 \\ 0.1 \\ -0.2 \\ 0.9 \\ -1.5 \end{bmatrix} \xrightarrow{P_A} [\mathbf{x}]_3 = \begin{bmatrix} 1.3 \\ 0 \\ 0 \\ 0.9 \\ -1.5 \end{bmatrix}$$

Figure 10.2: A minimum-distance projection onto the set $A$, with $s = 3$. All but the 3 largest absolute values are set to zero.

where $\Phi^+ = \Phi^T(\Phi\Phi^T)^{-1}$ is the Moore-Penrose pseudo-inverse of $\Phi$. Equation (10.3) was first derived, in the context of the present work, by Chartrand (2013). The derivation is a standard application of constrained optimization and linear algebra, and is an important element of sparse coding with the Difference Map, so I present it here for completeness.

Equation (10.3) is derived as follows. Note that the minimum-distance projection onto $B$ is defined by the linearly-constrained quadratic program (LCQP):

$$P_B(\mathbf{z}_0) = \arg\min_{\mathbf{z}\in\mathbb{R}^n} \tfrac{1}{2}\|\mathbf{z} - \mathbf{z}_0\|_2^2 \;\text{ such that }\; \Phi\mathbf{z} = \mathbf{x}. \tag{10.4}$$

The Lagrangian of this LCQP is

$$\mathcal{L}(\mathbf{z}, \lambda) = \tfrac{1}{2}\|\mathbf{z} - \mathbf{z}_0\|_2^2 + \lambda(\Phi\mathbf{z} - \mathbf{x}). \tag{10.5}$$

where $\lambda \in \mathbb{R}$ is the Lagrange multiplier (Boyd and Vandenberghe, 2004). The $\mathbf{z}$ that solves the LCQP (10.4) also minimizes $\mathcal{L}$, and is found by setting $\nabla_{\mathbf{z}}(\mathcal{L}) = 0$, which yields

$$\mathbf{z} = \mathbf{z}_0 + \Phi^\top\lambda. \tag{10.6}$$

Plugging (10.6) into $\mathbf{x} = \Phi\mathbf{z}$ and solving for $\lambda$ gives

$$\lambda = (\Phi\Phi^\top)^{-1}(\Phi\mathbf{z}_0 - \mathbf{x}).$$

Finally, pluging this into (10.6) gives

$$\mathbf{z} = \mathbf{z}_0 - \Phi^\top(\Phi\Phi^\top)^{-1}(\Phi\mathbf{z}_0 - \mathbf{x}),$$

as in (10.3).

Equation (10.3) provides a minimum distance onto the set $B = \{\mathbf{z} : \Phi\mathbf{z} = \mathbf{x}\}$. However, recall that it is often more realistic to allow for an imperfect reconstruction, meaning

$$\|\mathbf{x} - \Phi\mathbf{z}\|_2 \leq \delta$$

for some $\delta > 0$. In this case, it makes sense to redefine the set $B$ as

$$B = \{\mathbf{z} : \|\Phi\mathbf{x} - \mathbf{z}\|_2 \leq \delta\}.$$

The minimum-distance projection onto $B$ is defined by the quadratically-constrained quadratic program (QCQP)

$$P_B(\mathbf{z}_0) = \arg\min_{\mathbf{z}\in\mathbb{R}^n} \|\mathbf{z} - \mathbf{z}_0\|_2 \text{ such that } \|\Phi\mathbf{z} - \mathbf{x}\|_2 \leq \delta. \tag{10.7}$$

However, there are two issues with solving (10.7). The first is that we need to know the noise level $\delta$. In practice, we will not know the number $\delta$, though we might estimate it using a technique like simulated annealing. The second problem is that

solving a QCQP is very costly, with a runtime of approximately $O(n^3)$ where $n$ is the dimensionality of $\mathbf{x}$. Because the QCQP occurs frequently, inside each iteration of DM, solving such a QCQP would be prohibitively expensive.

Luckily, it turns out that defining $P_B$ with Equation (10.3) gives very good (and fast) results for the noisy case (*i.e.,* when $\delta > 0$), even though it was derived from the assumption that $\delta = 0$. Figure 10.3 shows that the linearly constrained $P_B$ using Equation (10.3) (LCQP, in the legend) allows DM to converge much more quickly than the quadratically constrained $P_B$ which uses (10.7) instead (QCQP, in the legend), even when given noisy observations. In this experiment, I generate a random $\Phi \in \mathbb{R}^{400\times1000}$ and $\|\mathbf{z}\|_0 = 150$ (see Chapter 12 for details on constructing $\Phi$ and $\mathbf{z}$). I calculate the noiseless $\mathbf{x} = \Phi\mathbf{z}$ and the noisy $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon \cdot \mathcal{N}(0, 1)$ such that $\text{SNR}(\mathbf{x}, \tilde{\mathbf{x}}) = 20$ dB, which ensures that $\delta > 0$. The Difference Map is then given $\Phi$ and $\tilde{\mathbf{x}}$, and asked to recover $\mathbf{z}$ using either the quadratically constrained $P_B$ (QCQP) or the linearly constrained $P_B$ (LCQP). The computationally expensive QCQP at each iteration causes DM to converge much more slowly. Thus I only consider the LCQP version of DM for the remainder of this work.

## 10.1 Comparison to Other Algorithms

In the following chapters, I compare the reconstruction error and convergence rate of the Difference Map to a representative sample of commonly used algorithms for sparse coding: Least Angle Regression (LARS) (Efron et al., 2004), Fast Iterative Soft Thresholding Algorithm (FISTA) (Beck and Teboulle, 2009), Stagewise Orthogonal Matching Pursuit (StOMP) (Donoho et al., 2012), Accelerated Iterative Hard Thresholding (AIHT) (Blumensath, 2012), Subspace Pursuit (SP) (Dai and

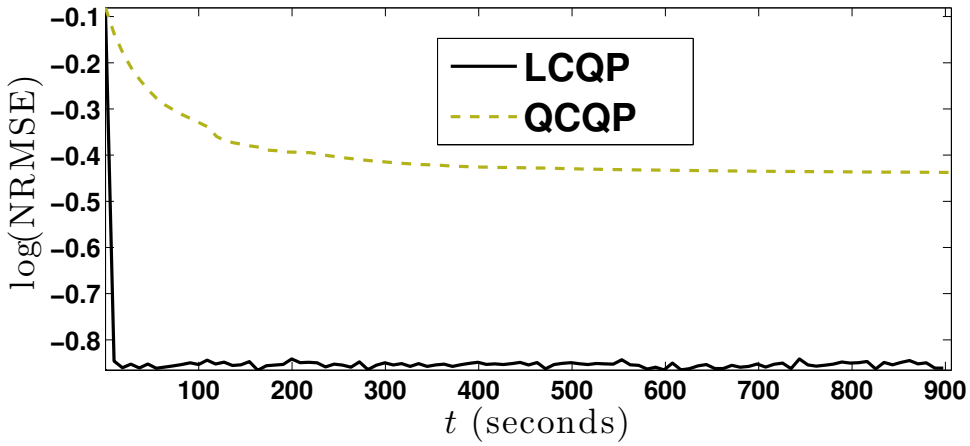



Figure 10.3: Comparing the noisy and noise-free versions of $P_B$. The noise-free, linearly constrained approximation (LCQP) of $P_B$ allows the Difference Map to recover the signal $\mathbf{z}$ much more quickly than the noisy, quadratically constrained (QCQP) version of $P_B$, even when given the noisy observation $\tilde{\mathbf{x}} = \Phi\mathbf{z} + \epsilon \cdot \mathcal{N}(0, 1)$. I measure the log(NRMSE) of the estimate $\hat{\mathbf{z}}$, computed by $\log(\|\mathbf{z} - \hat{\mathbf{z}}\|_2 / \|\mathbf{z}\|_2)$. See text immediately prior to Section 10.1 for additional details.

Milenkovic, 2009), Iteratively Reweighted Least Squares (IRLS) (Chartrand and Yin, 2008) and Alternating Direction Method of Multipliers (ADMM) (Boyd et al., 2011; Chartrand and Wohlberg, 2013).

The projection $P_A$, defined in Equation (10.2) and illustrated in Figure 10.2, is an important part of many sparse-coding algorithms (Blumensath, 2012; Blumensath and Davies, 2009, 2010; Dai and Milenkovic, 2009; Qiu and Dogandžić, 2010, 2011). The projection $P_B$ defined in Equation (10.3) also appears in the "expectectation-conditional maximize either" (ECME) algorithm (Qiu and Dogandžić, 2010, 2011). Normalized Iterative Hard Thresholding (NIHT) (Blumensath and Davies, 2009) uses a calculation similar to $P_B$, replacing the pseudo-inverse with $\mu_t \Phi^\top$ for an appropriately chosen scalar $\mu_t$.

Given that many sparse-coding algorithms consider the same types of projections

as DM, any advantage achieved by DM must not come from the individual projections $P_A$ and $P_B$, but rather the way in which DM combines the two projections into a single iterative procedure. This is particularly true when comparing DM to the simple alternating map. Alternating between projections is guaranteed to find a point at the intersection of the two constraints if both are convex; however, if either of the constraints is not convex, it is easy for this scheme to get stuck in a local minimum that does not belong to the intersection, as was illustrated in Figure 9.2.

While many of the theoretical questions about DM remain open, it does come with a crucial guarantee: even on nonconvex problems, a fixed point (meaning $D(\mathbf{z}) = \mathbf{z}$) implies that the algorithm has found a solution (meaning a point in $A \cap B$). To see this, note that $D(\mathbf{z}) = \mathbf{z}$ implies

$$P_A \circ f_B(\mathbf{z}) = P_B \circ f_A(\mathbf{z}). \tag{10.8}$$

Thus if it reaches a fixed point, the algorithm has found a point that exists in both $A$ and $B$. (Note that the left-hand side of Equation (10.8) is in $A$, and the right-hand side in $B$, and the two are equal.) As a result, I believe that in some cases DM will converge to a sparse solution with lower reconstruction error than competing algorithms.

Note that Qiu and Dogandžić (2011) apply DM to the ECME algorithm (a variant of expectation maximization) in order to improve upon that algorithm's sparse-coding performance. Although one of ECME's two projections uses DM *internally*, ECME continues to combine the two projections in a simple alternating fashion, and is thus a flavor of the Alternating Map (AM). This is in stark contrast to my proposed algorithm, which uses DM *externally* to the individual projections as a more intricate

way of combining them. Qiu and Dogandžić's resulting algorithm, called DM-ECME, is capable of finding only *non-negative* signals (*i.e.,* $z_i > 0$), which is a strict subset of the problems that I consider in this work. Moreover, even on non-negative signals, I have found that DM-ECME did not perform competitively with any of the algorithms mentioned above, and thus I do not include it in the experiments described below.

The implementation details of the Difference Map (as well as the method for tuning the parameters of the competing algorithms) are given in Appendix C.

# Chapter 11

# Evaluating the Difference Map on Image Reconstruction

In this section, I will evaluate DM and a handful of state-of-the-art algorithms on a standard sparse-coding problem: reconstructing images. Each algorithm will be given the same dictionary $\Phi$, the desired sparsity level $\|\mathbf{z}\|_0$, and the same observation (image) $\mathbf{x}$, and will attempt to find a $\mathbf{z}$ that minimizes the reconstruction error subject to the given sparsity level. This process will be repeated for a variety of sparsity levels $\|\mathbf{z}\|_0$. Some algorithms, like DM, consider the individual projections onto the two constraint sets; others combine the two constraints into a single objective, as in Equation (7.4).

Unlike Part III, where the sparse-coding algorithms reconstructed the SIFT features extracted from an image, in the remainder of this dissertation the algorithms reconstruct the pixel intensities (grayscale values) of the image. When reconstructing a large image, I treat each $w \times w$ patch of grayscale pixel intensities as an independent signal $\mathbf{x}$ to reconstruct. Thus each algorithm reconstructs the image by reconstructing

each patch independently. Recall that DM has an expensive startup cost of calculating the pseudo inverse of the dictionary, $\Phi^+$. Because the dictionary $\Phi$ is constant for each image patch, the algorithm only needs to compute the pseudo-inverse in (10.3) once. By amortizing the cost of the pseudo-inversion over all patches, this effectively allows DM to converge more quickly (compared to re-computing the pseudo-inverse at each iteration). I amortize the cost of pre-computation for other algorithms as well (most notably ADMM and IRLS)[1].

In order to test the performance of the algorithms when reconstructing natural images, the algorithms require a dictionary $\Phi$ learned for sparse image reconstruction. Dictionary learning is not the focus of this dissertation, but I present the dictionary-learning method in Appendix D for completeness. The learned dictionary contains the typical combination of high- and low-frequency edges, at various orientations and scales. Some examples of dictionary elements are shown in Figure 11.1.

With the dictionary $\Phi$ in hand, the algorithms sparse-code and then reconstruct several natural images. I measure the quality of the sparse reconstruction as a function of time. At time $t$, I measure the reconstruction quality of patch $\mathbf{x}$ as follows. First, I perform hard-thresholding on the algorithm's current guess $\mathbf{z}_t$, setting the $m - s$ smallest absolute values to zero, yielding the $s$-sparse vector $[\mathbf{z}_t]_s$. I then calculate the reconstruction

$$\mathbf{x}_t = \Phi[\mathbf{z}_t]_s$$

and measure the SNR of $\mathbf{x}$ (the true image patch) to $\mathbf{x}_t$. (Recall from Section 7.2 that

[1]By amortizing the pre-computation, I mean that these algorithms require expensive calculations before they can reconstruct an image patch, but the expensive calculation need not be repeated for the remainder of the image patches. By dividing the cost of the expensive calculation by the number of image patches reconstructed, one estimates the expected computation time required per image patch.

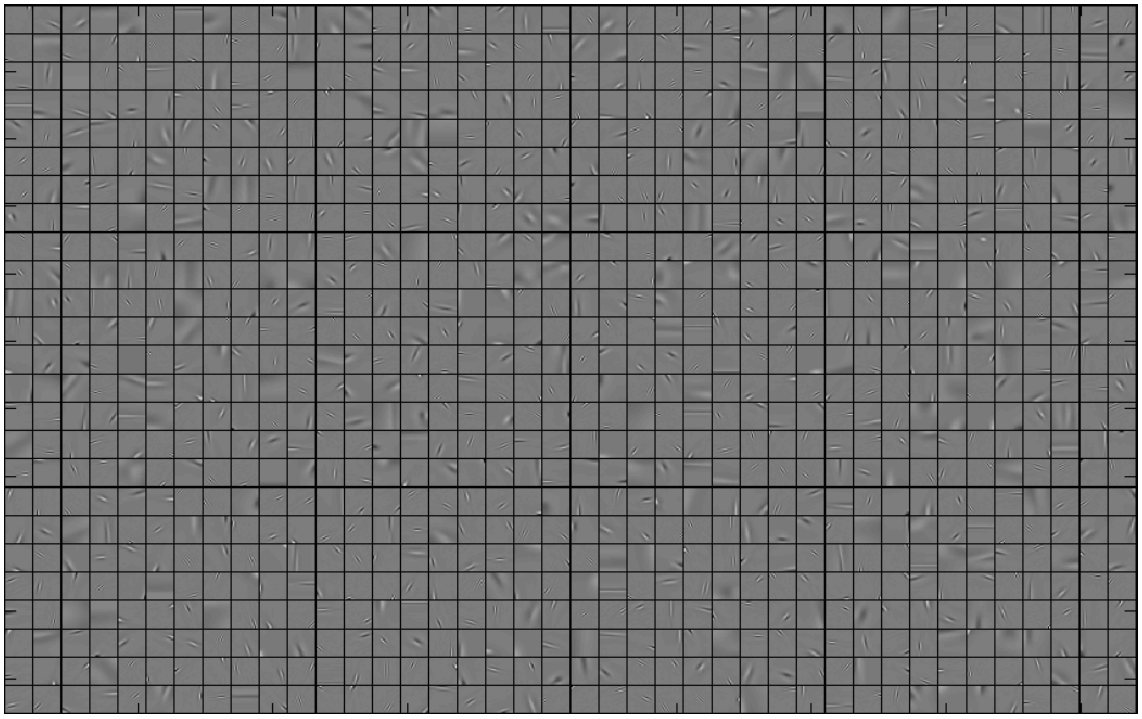


Figure 11.1: Example elements from the dictionary $\Phi \in \mathbb{R}^{400\times1000}$ used for reconstruction. The dictionary contains elements of size $20 \times 20$ pixels, learned from 10 million image patches from the *person* and *hill* categories of ImageNet (Deng et al., 2009).

I use SNR to measure the reconstruction of image pixels, and that a high SNR implies low reconstruction error.) Thus I am measuring how well, at time $t$, the algorithm can create a *sparse* reconstruction of $\mathbf{x}$. Note that algorithms returning a solution that is *sparser* than required will not be affected by the hard-thresholding step.

Each algorithm reconstructs a $320 \times 240$ image of a dog, seen in Figure 11.2, using the $400 \times 1000$ dictionary from Figure 11.1. I measure runtime instead of iterations, as the time required per iteration varies widely for the algorithms considered. Additionally, the pre-computation for DM is the longest of any algorithm, requiring the pseudo-inverse of the dictionary. The amortized cost of this pre-computation is included in the timekeeping. I measure results for both $s = 100$ and $s = 200$, as well

Table 11.1: Signal to noise ratio (SNR, in decibels) of the reconstructed image from Figure 11.2. I test various sparsity levels $s$ and various runtimes $t$ (seconds per entire image). The difference map consistently achieves high SNR. Bold entries indicate the highest SNR for each value of $s$ and $t$.

| | $s = 100$ | | | $s = 200$ | | |
|---|---|---|---|---|---|---|
| | $t = 10$ | $t = 20$ | $t = 30$ | $t = 10$ | $t = 20$ | $t = 30$ |
| Diff. Map | 15.91 | **17.55** | **17.45** | **20.28** | **22.38** | **23.34** |
| FISTA | 4.80 | 12.04 | 17.21 | 4.82 | 12.13 | 21.71 |
| ADMM | 15.51 | 16.71 | 17.31 | 19.80 | 21.96 | 23.00 |
| IRLS | 9.62 | 13.52 | 14.92 | 13.47 | 18.38 | 20.62 |
| Sub. Pursuit | **16.47** | 16.78 | 16.84 | 16.94 | 16.88 | 16.87 |
| LARS | 10.62 | 12.66 | 14.16 | 10.63 | 12.68 | 14.24 |
| AIHT | 14.71 | 15.62 | 16.18 | 18.72 | 19.91 | 20.69 |
| StOMP | 15.55 | 15.50 | 15.50 | 17.65 | 17.92 | 17.93 |

as $t = 10, 20$ and 30 seconds[2]. The results in Table 11.1 show that DM consistently achieves a very good SNR of the reconstruction. As would be expected, increasing $s$ and $t$ tend to improve each algorithm's reconstruction performance.

The highest quality reconstructions, achieved with $s = 200$ and $t = 30$, are shown in Figure 11.2. While some algorithms fail to reconstruct details in the animal's fur and the grass, many algorithms reconstruct the image well enough to make it difficult to find errors by mere visual inspection. I show the difference between the reconstructions and the original image (Figure 11.2, bottom row), where a neutral gray color in the difference image corresponds to a perfect reconstruction of that pixel; white and black are scaled to a difference of 0.3 and -0.3, respectively (the original image was scaled to the interval [0,1]).

The advantage of DM over other algorithms, when sparsely reconstructing images,

[2]I measure time in seconds per full-image reconstruction, which is actually performed independently for each $20 \times 20$ patch. Thus $t = 10, 20$ and 30 correspond to approximately $0.05, 0.1$, and 0.15 seconds per patch, respectively.

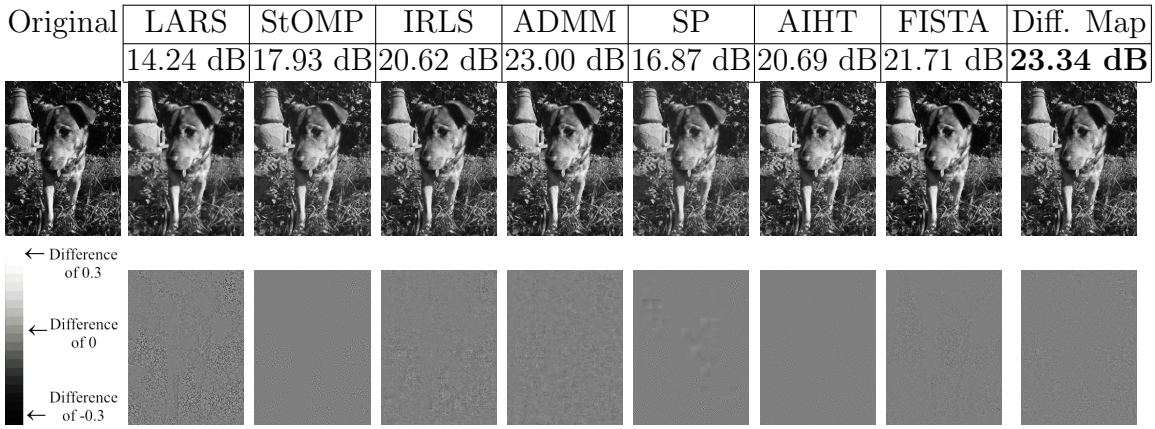
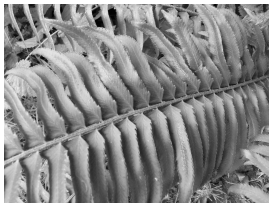



Figure 11.2: Reconstructing a natural image. The Difference Map outperforms the other algorithms (SNR shown in decibels, top row) when reconstructing a $320 \times 240$ image of a dog (reconstructions shown in middle row). Difference images (bottom row) show the difference between the reconstruction and the original image, which ranges from -0.3 (black) to 0.3 (white) – original grayscale values are between 0 (black) and 1 (white). Results for $s = 200$ and $t = 30$. Difference images are best seen by zooming in.

can be seen with a large variety of images. In Figure 11.3, we see that DM consistently achieves the best reconstruction.

| **Diff. Map** | **23.19** |
|---|---|
| FISTA | 19.49 |
| ADMM | 22.16 |
| IRLS | 20.58 |
| LARS | 14.51 |
| SP | 17.85 |
| StOMP | 19.32 |
| AIHT | 20.37 |



| **Diff. Map** | **24.84** |
|---|---|
| FISTA | 18.96 |
| ADMM | 23.78 |
| IRLS | 23.31 |
| LARS | 18.02 |
| SP | 20.81 |
| StOMP | 23.23 |
| AIHT | 22.65 |

| **Diff. Map** | **22.79** |
|---|---|
| FISTA | 19.00 |
| ADMM | 21.92 |
| IRLS | 20.51 |
| LARS | 14.81 |
| SP | 17.89 |
| StOMP | 18.81 |
| AIHT | 20.08 |

| **Diff. Map** | **24.80** |
|---|---|
| FISTA | 19.31 |
| ADMM | 24.06 |
| IRLS | 23.28 |
| LARS | 18.00 |
| SP | 20.38 |
| StOMP | 21.13 |
| AIHT | 22.57 |

Figure 11.3: The Difference Map regularly outperforms other algorithms in finding sparse reconstructions of a variety of images. Each algorithm is evaluated by measuring the SNR in decibels between the reconstruction and the original image (left column). Images are scaled to $320 \times 240$ pixels ($240 \times 320$ for horizontal images). Reconstructions have sparsity $s = 200$, and are completed in 30 seconds per image (approximately 0.15 seconds per $20 \times 20$ patch). The dictionary $\Phi$ is the same as in Figure 11.1.

# Chapter 12

# Evaluating the Difference Map on Compressed Sensing

Compressed sensing is very closely related to sparse coding. In compressed sensing (Donoho, 2006), we are again given $\mathbf{x}$ and $\Phi$ and asked to find a sparse $\mathbf{z}$ such that

$$\mathbf{x} = \Phi\mathbf{z}$$

or, in the noisy case, such that

$$\|\mathbf{x} - \Phi\mathbf{z}\|_2 \leq \delta.$$

The same algorithms used for sparse coding can also be used for compressed sensing, but the physical interpretation and measure of success are different. While in sparse coding we consider $\mathbf{z}$ a sparse encoding of the signal $\mathbf{x}$, in compressed sensing we consider $\mathbf{x}$ to be a projection or measurement of the true signal $\mathbf{z}$. In both cases we

are given $\mathbf{x}$ and asked to find $\mathbf{z}$, but the two problems disagree on which variable ($\mathbf{x}$ or $\mathbf{z}$) is "the signal."

For example, in sparse coding, $\mathbf{x}$ may be an image (the signal), and $\mathbf{z}$ would be a compressed or encoded version of this image. In that setting, $\mathbf{x}$ is a signal generated by a physical phenomenon (photons hitting a light receptor), and $\mathbf{z}$ is just a compressed encoding of that signal. In compressed sensing, entries in $\mathbf{x}$ may be the measurements from an MRI, which have been diffused and scattered through a body before being recorded. Finding $\mathbf{z}$, in this case, is akin to inferring what was actually being scanned by the MRI (*i.e.*, recovering the position of the body from noisy measurements). Thus $\mathbf{z}$ is the physical phenomenon or signal (what was inside the MRI), and $\mathbf{x}$ is a randomized measurement of the signal.

The difference between sparse coding and compressed sensing is slippery to grasp and, in the end, nearly inconsequential in this dissertation. The only relevant difference, for the purpose of this dissertation, is that the disagreement between which variable represents the "true signal" ($\mathbf{x}$ in sparse coding, and $\mathbf{z}$ in compressed sensing) leads to two different measures of success. When sparse coding I measured the reconstruction error of $\mathbf{x}$, whereas in compressed sensing I will measure the reconstruction error of $\mathbf{z}$. I will measure the reconstruction error of $\mathbf{z}$ with NRMSE; when adding artificial noise to make the task more challenging, I will measure the added noise with SNR.

The idea of "measuring the reconstruction error of $\mathbf{z}$" is puzzling when considering the previous experiments, in which there was no "true" $\mathbf{z}$. Again, this is the core difference between sparse coding and compressed sensing. In the latter, there *is* a true $\mathbf{z}$, though the algorithm is still only given $\mathbf{x}$ and $\Phi$ as input. To measure the reconstruction of $\mathbf{z}$, then, I will require a different experimental methodology than

before.

One of the most common ways to measure a compressed sensing algorithm is by randomly generating $\Phi$, $\mathbf{z}$, and $\mathbf{x}$ as follows. Given positive integers $m, n$, and $s$, I generate the matrix $\Phi \in \mathbb{R}^{m \times n}$ with entries sampled randomly from $\mathcal{N}(0, 1)$. I then ensure that columns have zero mean and unit variance. This matrix is called *a random dictionary*. I generate the $s$-sparse vector $\mathbf{z} \in \mathbb{R}^n$ whose $s$ nonzero entries are sampled from $\mathcal{N}(0, 1)$. I then calculate $\mathbf{x} = \Phi\mathbf{z}$, and the noisy "observation" $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon \cdot \mathcal{N}(0, 1)$. Finally, I ask each algorithm to reconstruct $\mathbf{z}$ given only $\Phi$ and $\tilde{\mathbf{x}}$.

Note that, in this setting, I am able to measure the reconstruction error of the algorithm's estimate $\hat{\mathbf{z}}$ of $\mathbf{z}$, because I generated $\mathbf{z}$. This was not the case in the previous chapter, where $\mathbf{x}$ was a patch of an image, and there was no "true $\mathbf{z}$". Again, this is the core difference between compressed sensing and sparse coding, but the algorithms solving the problem are the same.

In the remainder of this chapter, I compare the performance of DM to the same algorithms from Chapter 11. As an extra algorithm for comparison, I will include the Alternating Map (AM), which was introduced in Chapter 9; I implement AM with the same projections $P_A$ and $P_B$ defined as in Equations (10.2) and (10.3), respectively. This formulation of AM closely resembles the ECME algorithm for known sparsity levels (Qiu and Dogandžić, 2010). I test each algorithm with a wide variety of matrix sizes, sparsity, and noise levels.

## 12.1 Compressed Sensing Experiments

In the first experiment, each algorithm attempts to reconstruct $\mathbf{z}$ as I vary the sparsity level $s$. I choose $\epsilon$ so that the SNR is close to 20 dB. The results in Figure 12.1

demonstrate that for small values of $s$ (Figure 12.1 A), meaning sparser signals, most algorithms are able to recover $\mathbf{z}$ almost equally well. As I increase the value of $s$ (Figure 12.1 B,C), the signals become less sparse and other algorithms converge to undesirable minima. The Difference Map, however, continues to get very close to recovering $\mathbf{z}$.

In the next experiment, each algorithm attempts to reconstruct $\mathbf{z}$ as I vary the noise by changing $\epsilon$. I fix $s$ at 150. The results in Figure 12.2 show that with very little noise (A) and very high noise (D), the Difference Map performs as well as several algorithms at recovering the true signal $\mathbf{z}$, though it requires more time. For moderate amounts of noise (B,C), the Difference Map is able to get closer to recovering the signal than any other algorithm.

Note that DM and AM start "late" in all plots from Figures 12.1 and 12.2 because their pre-computation time is the longest (calculating $\Phi^+$). Each run of the algorithm is given a new random dictionary $\Phi$, which requires computing a new $\Phi^+$. Hence there can be no amortization of the cost of calculating $\Phi^+$, as there was in Chapter 11. Despite using the same projections, there is a large disparity in performance between DM and AM when $s > 75$. Because the two algorithms both use the same two projections $P_A$ and $P_B$, this performance gap shows the power of combining two simple projections in a more elaborate way than simply alternating between them.

From the results in Figures 12.1 and 12.2, I hypothesize that DM has a significant advantage with moderately noisy (SNR of approximately 20dB), less sparse signals (higher $s$); with these types of problems, other state-of-the-art compressed sensing algorithms get stuck in local minima or require a large amount of time to reach a good solution. Figure 12.3 shows the results of tests of this hypothesis with a variety of different matrix sizes and sparsity ratios, each time with an SNR of approximately

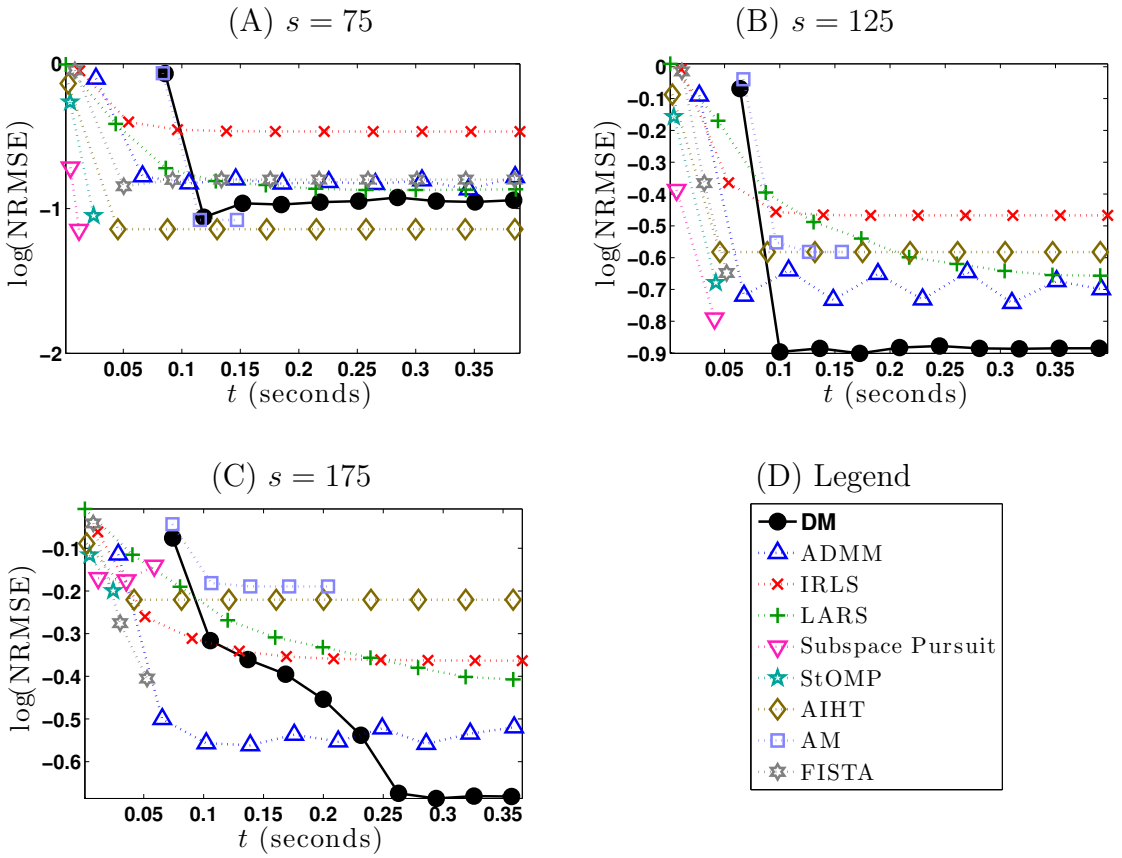



Figure 12.1: Reconstructing signals with various levels of sparsity $s$. Given $\mathbf{x}$ and $\Phi$, an algorithm tries to recover $\mathbf{z}$ such that $\mathbf{x} = \Phi\mathbf{z}$ and $\|\mathbf{z}\|_0 \leq s$. I measure the normalized root mean squared error (NRMSE) at time $t$ by estimating $\mathbf{z}_t$ and calculating $\|\mathbf{z} - \mathbf{z}_t\|/\|\mathbf{z}\|$. With sparser signals (A), most algorithms get equally close to recovering the true signal. With less sparse signals (B,C), the Difference Map gets closer than other algorithms to recovering the signal. Each plot is averaged over ten runs, with $\epsilon$ chosen to give an SNR of approximately 20 dB, and $\Phi \in \mathbb{R}^{400\times1000}$.

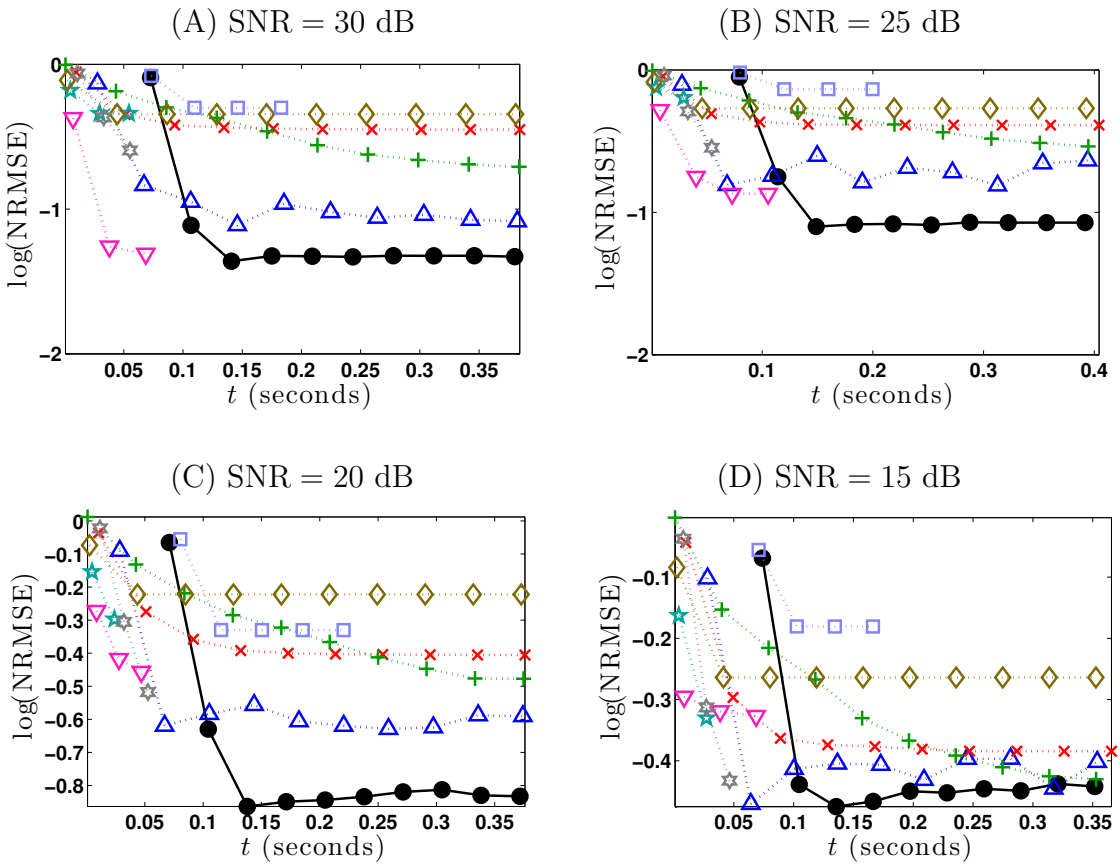



Figure 12.2: Reconstructing signals with various levels of noise $\epsilon$. Legend is the same as Figure 12.1. With very little noise (A) and large amounts of noise (D), the Difference Map recovers the signal as well as the best algorithms, though requiring more time. With moderate amounts noise (B,C), the Difference Map gets closer than other algorithms to recovering the signal. Each plot is averaged over ten runs, with $s = 150$ and $\Phi \in \mathbb{R}^{400 \times 1000}$.

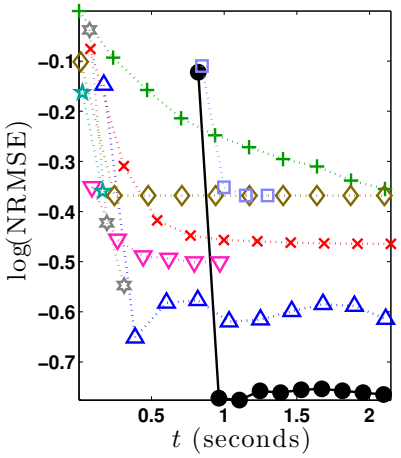
20 dB. The results show that DM does indeed outperform other algorithms in this setting, for all cases tested.

For all of the experiments reported above, the Difference Map's $\ell^0$ constraint (from (10.2)) was the same as the true $s$ used to generate the data. In many settings, however, the true $s$ is unknown. I measure the robustness of DM in this setting by fixing the true value $s$ (used to generate $\mathbf{z}$) while varying the $\ell^0$ constraint in (10.2). I then measure the log-NRMSE of the reconstructed signal $\hat{\mathbf{z}}$. The results in Figure 12.4 show that when the true $s$ (used to generate $\mathbf{z}$) is fixed at 150, DM continues to recover $\mathbf{z}$ better than any other algorithm for an $\ell^0$ constraint down to 90 and up to 190. Thus DM appears quite robust to the specific $\ell^0$ constraint value used when implementing the algorithm. Note that this "unknown $s$" setting was explored in more detail in Chapter 11 in the context of reconstructing natural images.

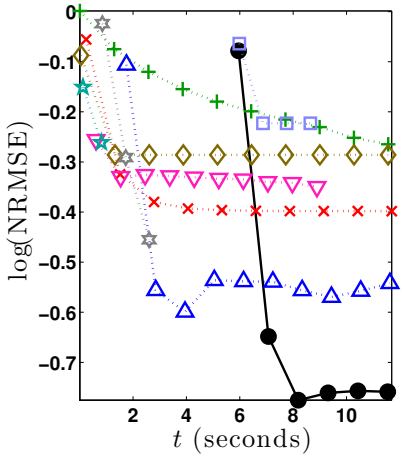
## 12.2 Summary of the Difference Map's performance

I have presented the Difference Map, a method of finding a point in the intersection of two constraint sets, and I have applied DM to the problems of sparse coding and compressed sensing. The constraint-set formulation is a natural fit for sparse-recovery problems, in which we have two competing constraints for $\mathbf{z}$: to reconstruct the observation $\mathbf{x}$ and to be sparse.

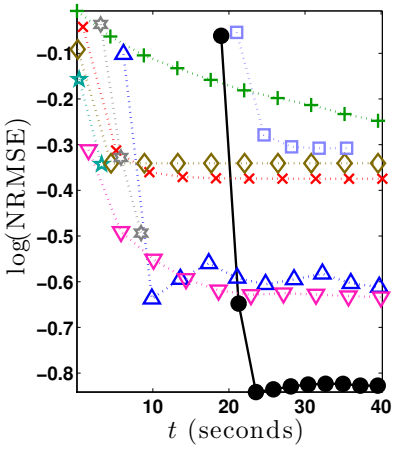
When the solution $\mathbf{z}$ is very sparse and the observation $\tilde{\mathbf{x}}$ is not too noisy, DM takes more time in finding the same solution as competing algorithms. However, when the solution $\mathbf{z}$ is *less* sparse and when the observation $\tilde{\mathbf{x}}$ is noisy, DM outperforms state-of-the-art sparse recovery algorithms. The noisy, less sparse setting corresponds well to reconstructing natural images, which can often require a large number of dictionary




(A) $\Phi \in \mathbb{R}^{1024 \times 2048}$, $s = 400$




(B) $\Phi \in \mathbb{R}^{2000 \times 5000}$, $s = 800$




(C) $\Phi \in \mathbb{R}^{3000 \times 8000}$, $s = 1050$

Figure 12.3: The difference map outperforms other algorithms at recovering $\mathbf{z}$ from a noisy observed signal with a wide variety of matrix sizes $\mathbb{R}^{m \times n}$, when sparsity is high ($s \approx n/3$). Legend is the same as Figure 12.1. The noisy observation $\tilde{\mathbf{x}} = \Phi\mathbf{z} + \epsilon \cdot \mathcal{N}(0, 1)$ has an SNR of approximately 20 dB. Each plot is averaged over ten runs.

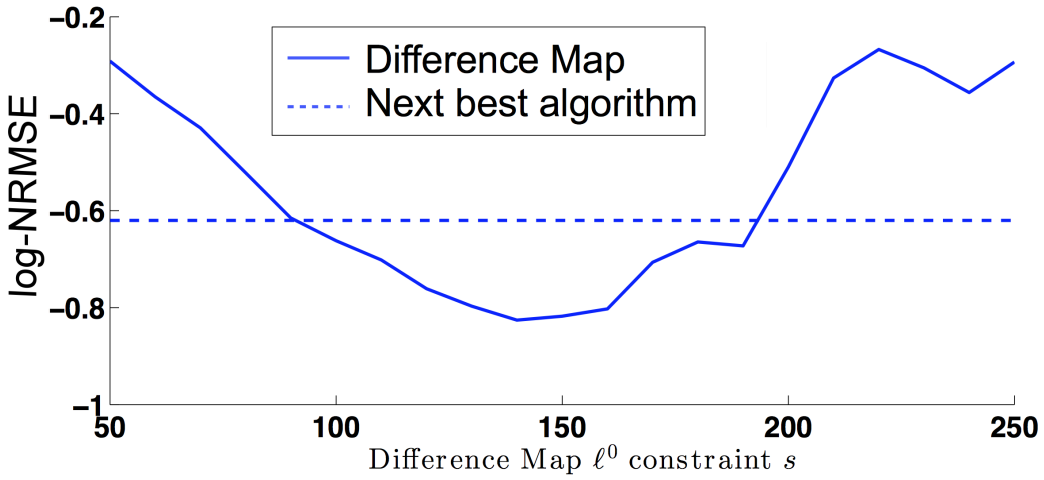



Figure 12.4: The Difference Map outperforms other algorithms and recovering $\mathbf{z}$, even when $s$ is unknown, for a wide range of values. Using a random $\Phi \in \mathbb{R}^{400\times1000}$, $s = 150$, and $\epsilon$ chosen to give an SNR of 20 dB, I vary the Difference Map $\ell^0$ constraint. The next best algorithm achieves a log-NRMSE of -0.62; the Difference Map outperforms this for any $\ell^0$ constraint between 90 and 190.

elements in order to accurately reconstruct. The experiments I have reported above show that DM performs favorably in reconstructing a variety of images, with a variety of parameter settings.

Parameter tuning can present a laborious hurdle to the researcher. DM requires tuning only a single parameter $\beta$. For all experiments performed with DM (natural image reconstruction for various images; reconstruction with random matrix dictionaries of various sizes, with varying amounts of sparsity and noise), I found DM to work almost equally as well for all $-0.9 \leq \beta \leq -0.1$. The robustness of DM under such a wide variety of parameter values and problems makes DM a very competitive choice for sparse coding and compressed sensing.

The robustness of DM comes from how it combines two simple projections into a single iterative procedure. The Alternating Map (AM) combines the same projections

in a simple alternating fashion, and performs poorly in almost all experiments. The gap in performance between these two methods demonstrates the power of combining multiple constraints in a more perspicacious way.

Finally, recall that performance in all experiments was measured as a function of time, which would seem to put DM at a natural disadvantage to other algorithms: DM requires the pseudo-inverse of the dictionary, computing which requires more time than any other algorithm's pre-computation. Despite this, DM consistently outperforms other algorithms.

# Part V

# Conclusions and Future Work

This dissertation examines the link between classifying an image and seeing an object. Part II of this dissertation asks, *if a classifier does a good job classifying images, does it necessarily see the objects that were intended to be recognized?* In order to answer this question, I developed a novel method of explaining a popular type of classifier (additive networks), called *contribution propagation* (Landecker et al., 2013).

Contribution propagation is based on the original idea of contributions by Poulin et al. (2006), which tell us how each dimension of a feature vector affected a classification. However, when the features $\mathbf{z}$ are extracted from a datum $\mathbf{x}$, we may still want an explanation at the level of the datum $\mathbf{x}$ that provides the same type of information as contributions do at the level of $\mathbf{z}$. Contribution propagation was designed to do exactly this, when $\mathbf{z}$ is the output of an additive network.

In Section 4.2, I gave three desirable properties for explaining the classifications performed by additive networks. In Section 4.3, I proved that contribution propagation satisfies all three properties. (It is important to note that the third property was proved only for linear networks.) Empirical testing with synthetic data, described in Sections 5.1 and 5.3, gave strong evidence that my method performs as desired. Finally, applying contribution propagation to a network trained with real-world data, described in Section 5.3, explained which parts of the images contributed most to the classifications. In response to the question, *If a classifier does a good job classifying images, does it necessarily see the objects that were intended to be recognized?* these results seem to tell us *no* for some cases reported in recent literature.

Part III of this dissertation asks, *If a model can sparsely reconstruct the data, will it also classify the data well?* In order to answer this question, I investigate the popular method of *sparse coding* as a means of feature extraction. Sparse coding is a method of representing a signal (or datum) as a sparse linear combination of known

elements. The resulting *sparse codes* (the coefficients of the sparse linear combination) are often treated as the features for classification as well as for image reconstruction.

In many settings, using the sparse codes for training and testing results in better classification performance. It appears to be widely assumed that this is due to two factors: (1) the fact that sparse codes will reconstruct the data means that the codes contain the same information as the original data, and (2) the sparse codes are sparse, which is an effective form of regularization to increase the classifier's ability to generalize (Bansal et al., 2010; Coates and Ng, 2011). In short, these claims amount to what I called the **sparse reconstruction hypothesis**: the best encoding of data for classification will give the best possible reconstruction at some level of sparsity.

I investigated this hypothesis by varying the sparsity level of several sparse-coding algorithms, and measuring classification error and reconstruction error. My results indicate that sparse coding does improve classification accuracy in most cases. However, the results of Section 8.3 are contrary to the sparse reconstruction hypothesis. For multiple datasets I showed two sets of sparse codes, one of which is sparser and reconstructs better (produced by the SP algorithm (Dai and Milenkovic, 2009)), the other of which achieves lower classification error (produced by the LARS algorithm (Efron et al., 2004)). This is not consistent with the sparse reconstruction hypothesis.

In Part IV of this dissertation, I described a new sparse coding method based on the Difference Map of Elser et al. (2007), which outperforms competing methods when applied to fairly complicated signals (meaning moderately noisy signals that require more dictionary elements to reconstruct). While Part III of this dissertation indicates that we do not adequately understand why sparse coding can decrease classification error, sparse coding (and the closely related problem of *compressed sensing*) is also very useful for other tasks including image compression, medical imaging, sig-

nal recovery, and denoising. I expect the Difference Map to be a strong contender for researchers choosing among algorithms in these fields.

While the results summarized above add numerous contributions to the fields of machine learning and computer vision, they also open up new avenues for future research. In Section 4.3, I proved that contribution propagation provides *trustworthy* explanations of how each dimension of the input affected the classification performed by a linear network. However, I proposed contribution propagation in the context of the more general additive networks. A proof of trustworthiness in this more general case is currently missing, and would greatly add to the small but growing literature on explaining the classifications of machine learning algorithms. Additionally, some very recent developments in *deep learning* have yielded networks which almost perfectly fit my definition of "linear networks" (Krizhevsky et al., 2012; Zeiler and Fergus, 2013). Applying contribution propagation to these networks would be interesting, particularly considering that they have been setting records for classification accuracy on a variety of difficult datasets.

I found evidence, in Section 8, that sparse reconstruction is not necessarily the right intermediate goal if the ultimate goal is classification. While the results were consistent across several sparse-coding algorithms and datasets, they all used the same feature extraction method (SIFT) and sparse-coding dictionary $\Phi$ (from Yang et al. (2009)). The sparse-coding community would benefit from testing whether or not these patterns persist with different features and with different dictionaries.

The Difference Map (DM) algorithm for sparse coding, presented in Section 10, outperforms state-of-the-art sparse-coding algorithms when the sparsity level was high. However, it is unclear *why* DM achieves such success. A thorough analysis of the convergence of DM would contribute greatly to our understanding of the algorithm,

as well as the problem that it solves.

# Bibliography

M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, 2006.

D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K. Müller. How to explain individual classification decisions. *Journal of Machine Learning Research*, 11:1803–1831, 2010.

V. Bansal, O. Libiger, A. Torkamani, and N. Schork. Statistical analysis strategies for association studies involving rare variants. *Nature Reviews Genetics*, 11(11): 773–785, 2010.

A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

C. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer New York, 2006.

T. Blumensath. Accelerated iterative hard thresholding. *Signal Processing*, 92(3):752 – 756, 2012.

T. Blumensath and M. Davies. Iterative hard thresholding for compressed sensing. *Applied and Computational Harmonic Analysis*, 27(3):265–274, 2009.

T. Blumensath and M. Davies. Normalized iterative hard thresholding: Guaranteed stability and performance. *IEEE Journal of Selected Topics in Signal Processing*, pages 298–309, 2010.

A. Bosch, A. Zisserman, and X. Munoz. Representing shape with a spatial pyramid kernel. In *International Conference on Image and Video Retrieval*, 2007.

B. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM, 1992.

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

L. Breslow and D. Aha. Simplifying decision trees: A survey. Technical report, Navy Center for Applied Research in Artificial Intelligence, 1997.

E. Candes. The restricted isometry property and its implications for compressed sensing. *Comptes Rendus Mathematique*, 346(9):589–592, 2008.

R. Chartrand. Nonconvex splitting for regularized low-rank + sparse decomposition. *IEEE Transactions on Signal Processing*, 60:5810–5819, 2012.

R. Chartrand. Personal communication, 2013.

R. Chartrand and B. Wohlberg. A nonconvex ADMM algorithm for group sparsity with sparse groups. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2013.

R. Chartrand and Wotao Yin. Iteratively reweighted algorithms for compressive sensing. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3869–3872, 2008.

A. Coates and A. Ng. The importance of encoding versus training with sparse coding and vector quantization. In *Proceedings of the 28th International Conference on Machine Learning*, pages 921–928, 2011.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3), 1995.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

W. Dai and O. Milenkovic. Subspace pursuit for compressive sensing signal reconstruction. *IEEE Transactions on Information Theory*, 55(5):2230–2249, 2009.

I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications in Pure and Applied Mathematics*, 57(11):1413–1457, 2004.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

D. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52(4): 1289–1306, 2006.

D. Donoho and M. Elad. Optimally sparse representation in general (nonorthogonal) dictionaries via $\ell$-1 minimization. *Proceedings of the National Academy of Sciences*, 100(5):2197–2202, 2003.

D. Donoho, Y. Tsaig, I. Drori, and J. Starck. Sparse solution of underdetermined systems of linear equations by stagewise orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 58(2):1094–1121, 2012.

M. Dredze, K. Crammer, and F. Pereira. Confidence-weighted linear classification. In *International Conference on Machine Learning*, 2008.

B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Annals of Statistics*, 32:407–499, 2004.

M. Elad and M. Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Transactions on Image Processing*, 15(12):3736–3745, 2006.

V. Elser, I. Rankenburg, and P. Thibault. Searching with iterated maps. *Proceedings of the National Academy of Sciences*, 104(2):418–423, 2007.

K. Engan, S. Aase, and Hakon H. Method of optimal directions for frame design. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 2443–2446. IEEE, 1999.

L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. In

*IEEE Computer Vision and Pattern Recognition, Workshop on Generative-Model Based Vision*, 2004.

P. Földiak. Forming sparse representations by local anti-Hebbian learning. *Biological Cybernetics*, 64(2):165–170, 1990.

L. Fu. Rule generation from neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, 24(8), 1994.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

S. Gao, I. Tsang, L. Chia, and P. Zhao. Local features are not lonely–laplacian sparse coding for image classification. In *Computer Vision and Pattern Recognition*, pages 3555–3561. IEEE, 2010.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):721–741, 1984.

I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

G. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

F. Huang and Y. LeCun. Large-scale learning with SVM and convolutional nets for generic object categorization. In *Computer Vision and Pattern Recognition*, 2006.

T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

K. Kavukcuoglu, P. Sermanet, Y. Bourreau, K. Gregor, M. Mathieu, and Y. LeCun. Learning convolutional feature hierarchies for visual recognition. In *Advances in Neural Information Processing*, 2010.

I. Kononenko. Machine learning for medical diagnosis: History, state of the art and perspective. *Artificial Intelligence in Medicine*, 23:89–109, 2001.

A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 1, page 4, 2012.

W. Landecker. Matlab code for sparse coding with the difference map. `http://web.cecs.pdx.edu/~mm/DM_Reconstruct.m`, 2013.

W. Landecker, M. Thomure, G. Kenyon, M. Mitchell, L. Bettencourt, and S. Brumby. Interpreting classifications of hierarchical networks. In *Computational Intelligence in Data Mining (CIDM), Symposium on Interpretable Machine Learning*, 2013.

W. Landecker, R. Chartrand, and S. DeDeo. Robust compressed sensing and sparse coding with the difference map. *European Conference in Computer Vision*, 2014. To appear.

S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition*, volume 2, pages 2169–2178, 2006.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

H. Lee, A. Battle, R. Raina, and A. Ng. Efficient sparse coding algorithms. *Advances in Neural Information Processing Systems*, 19:801, 2007.

L. Liu and L. Wang. What has my classifier learned? Visualizing the classification rules of bag-of-feature model by support region detection. In *Computer Vision and Pattern Recognition*, pages 3586 –3593, 2012.

D. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, volume 2, pages 1150–1157. IEEE, 1999.

M. Lustig, D. Donoho, J. Santos, and J. Pauly. Compressed sensing MRI. *Signal Processing Magazine, IEEE*, 25(2):72–82, 2008.

J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 689–696. ACM, 2009.

J. Mao and A.K. Jain. Artificial neural networks for feature extraction and multivariate data projection. *IEEE Transactions on Neural Networks*, 6(2):296–317, Mar 1995. ISSN 1045-9227.

M. Marszatek and C. Schmid. Accurate object localization with shape masks. In *Computer Vision and Pattern Recognition*, pages 1–8, 2007.

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

B. Natarajan. Sparse approximate solutions to linear systems. *SIAM Journal on Computing*, 24(2):227–234, 1995.

A. Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *International Conference in Machine Learning*, page 78, 2004.

A. Ng. Stanford CS229 lecture notes. `http://cs229.stanford.edu/notes/cs229-notes3.pdf`, April 2014.

B. Olshausen and D. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

B. Olshausen and D. Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, 1997.

N. Pinto, Y. Barhomi, D. Cox, and J. DiCarlo. Comparing state-of-the-art visual features on invariant object recognition tasks. In *IEEE Workshop on Applications of Computer Vision*, pages 463–470, 2011.

J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.

B. Poulin, R. Eisner, D. Szafron, P. Lu, R. Greiner, D.S. Wishart, A. Fyshe, B. Pearcy, C. MAcDonell, and J. Anvik. Visual explanation of evidence in additive classifiers. In *Proceedings of 18th Conference on Innovative Applications of Artificial Intelligence*, 2006.

K. Qiu and A. Dogandžić. Double overrelaxation thresholding methods for sparse signal reconstruction. In *Information Sciences and Systems*, pages 1–6. IEEE, 2010.

K. Qiu and A. Dogandžić. Nonnegative signal reconstruction from compressive samples via a difference map ECME algorithm. In *IEEE Workshop on Statistical Signal Processing*, pages 561–564, 2011.

R. Raina, A. Battle, H. Lee, B. Packer, and A. Ng. Self-taught learning: Transfer learning from unlabeled data. In *Proceedings of the 24th International Conference on Machine Learning*, pages 759–766, 2007.

M. Ranzato, J. Susskind, V. Minh, and G. Hinton. On deep generative models with applications to recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2011.

M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11), 1999.

M. Robnik-Šikonja and I. Kononenko. Explaining classifications for individual instances. *IEEE Transactions on Knowledge and Data Engineering*, 20:589–600, 2008.

T. Serre, A. Oliva, and T. Poggio. A theory of object recognition: computations and circuits in the feedforward path of the ventral stream in the primate visual cortex. Technical Report 036, MIT, 2005.

T. Serre, A. Oliva, and T. Poggio. A feedforward architecture accounts for rapid

categorization. *Proceedings of the National Academy of Sciences*, 104(15):6424–6429, 2007.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

W. Siedlecki and J. Sklansky. A note on genetic algorithms for large-scale feature selection. *Pattern Recognition Letters*, 10(5):335–347, 1989.

V. Stodden, L. Carlin, D. Donoho, I. Drori, D. Dunson, M. Elad, S. Ji, J. Starck, J. Tanner, V. Temlyakov, Y. Tsaig, and Y. Xue. SparseLab Matlab Software. `http://sparselab.stanford.edu/`, 2007.

E. Strumbelj, I. Kononenko, and M. Robnik-Šikonja. Explaining instance classifications with interactions of subsets of feature values. *Data and Knowledge Engineering*, 68(10):886–904, 2009.

M. Thomure, M. Mitchell, and G. Kenyon. On the role of shape prototypes in hierarchical models of vision. In *The International Joint Conference on Neural Networks*, pages 1–6. IEEE, 2013.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

A. Tickle, R. Andrews, M. Golea, and J. Diederich. The truth will come to light: Directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6), 1998.

J. Tropp and A. Gilbert. Signal recovery from random measurements via orthogonal

matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655–4666, 2007.

G. Webb, J. Boughton, and Z. Wang. Not so naive Bayes: Aggregating one-dependence estimators. *Machine Learning*, 58(1), 2005.

Wikipedia. Scale-invariant feature transform. `http://en.wikipedia.org/wiki/Scale-invariant_feature_transform`, 2014.

J. Yang and V. Honavar. Feature subset selection using a genetic algorithm. *IEEE Intelligent Systems*, 13(2):44–49, 1998.

J. Yang, K. Yu, Y. Gong, and T. Huang. Linear spatial pyramid matching using sparse coding for image classification. In *Computer Vision and Pattern Recognition (Computer Vision and Pattern Recognition), 2009.*, pages 1794–1801. IEEE, 2009.

M. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. *Computing Research Repository (The arXiv)*, abs/1311.2901, 2013.

M. Zeiler, D. Kirshan, G. Taylor, and R. Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition*, 2010.

M. Zeiler, G. Taylor, and R. Fergus. Adaptive deconvolutional networks for mid and high level learning. In *International Conference on Computer Vision*, 2011.

# Appendix A

# Deriving the Contribution Propagation Equation for Linear SVMs

Let $\mathbf{Z}^i = (Z_1^i, Z_2^i, \ldots, Z_m^i) \in \mathbb{R}^m$ be the $i^{\text{th}}$ training vector, where the superscript $i$ is the index into the training dataset and the subscripts $1, 2, \ldots, m$ are the indices into the dimensions of the vector. Let $\mathcal{S}$ be the set of indices (into the training dataset) indicating the support vectors, as discussed in Section 2.1. Let $\mathbf{Z}$ be the test datum will be classified. Then Equation (2.4) becomes

$$\begin{aligned}
\hat{y}_\theta(\mathbf{Z}) &= \operatorname{sgn}\left[\sum_{s\in\mathcal{S}} \alpha_s \langle \mathbf{Z}, \mathbf{Z}^s \rangle + b\right] \\
&= \operatorname{sgn}\left[\sum_{s\in\mathcal{S}} \alpha_s \sum_{i=1}^{m} Z_i Z_i^s + b\right] \\
&= \operatorname{sgn}\left[\sum_{i=1}^{m} Z_i \left(\sum_{s\in\mathcal{S}} \alpha_s Z_i^s\right) + b\right]
\end{aligned}$$

Recall that in the case of a network whose topmost node is an additive classifier, I refer to this topmost node as $Y$, which computes the score of the classifier,

$$Y = \sum_{i=1}^{m} Z_i \left( \sum_{s \in \mathcal{S}} \alpha_s Z_i^s \right) + b$$

Plugging into Equation (4.12) yields

$$\mathcal{C}\left(Z_i \to Y\right) = \frac{Z_i \left( \sum_{s \in \mathcal{S}} \alpha_s Z_i^s \right)}{Y}$$

as desired. Note that I have assumed that the bias $b = 0$, as was the case for the networks analyzed in this dissertation.

# Appendix B

# Obviating Division-by-Zero with Linear/Max Networks

In this appendix, I will show how the contribution propagation formulae for linear/-max networks, presented in Section 5.1, combine in such a way that there is no possibility for division by zero. Recall that theorem 3 demonstrates how the function computed by a linear network can be written as a single linear function of the network's inputs; moreover the formula for the inputs' contributions in such a network (Equation (4.23)) have no denominator, and therefore no concern of division by zero. Although a linear/max network is not a linear network due to the *max*-nodes, the same trick applies.

To simplify notation, I will redefine $\mathsf{ch}(V)$ to be the *indices* of the children nodes, rather than the children nodes themselves. To be clear, I have previously defined

$$\mathsf{ch}(V) = \{U_i \in \mathcal{V} \ : \ U_i \to V \in \mathcal{E}\}$$

where $\mathcal{E}$ is the set of edges in the network. However, the following equations will be greatly simplified by momentarily redefining $\mathsf{ch}\,(V)$ as

$$\mathsf{ch}\,(V) \stackrel{\mathsf{def}}{=} \{i \ : \ U_i \to V \in \mathcal{E}\}.$$

Recall that there are five layers in the network discussed Section 5.1 (including the classifier node, which is a layer unto itself), which alternate between linear and maximum functions. The function computed by network with output $Y$ and input $x_1, \ldots, x_n$ is fully defined by

$$Y = \sum_{i \in \mathsf{ch}(Y)} \beta_i \sum_{j \in \mathsf{ch}(U_i)} \delta_j^i \sum_{k \in \mathsf{ch}(V_j)} \beta_k^j \sum_{l \in \mathsf{ch}(U_k)} \delta_l^k \sum_{h \in \mathsf{ch}(V_l)} \beta_h^l x_h \tag{B.1}$$

where $\beta_i$ are the coefficients of the C2 nodes to the classifier, $\delta_j^i \in \{0, 1\}$ implements the maximum function of the C2 nodes (as in Equation (5.4)), $\beta_k^j$ are the S2 coefficients, $\delta_l^k$ implements the maximum function of the C1 nodes, and $\beta_h^l$ are the S1 coefficients. Rearranging the equation, we have

$$\begin{aligned}
Y &= \sum_{i \in \mathsf{ch}(Y)} \sum_{j \in \mathsf{ch}(U_i)} \sum_{k \in \mathsf{ch}(V_j)} \sum_{l \in \mathsf{ch}(U_k)} \sum_{h \in \mathsf{ch}(V_l)} \beta_i \delta_j^i \beta_k^j \delta_l^k \beta_h^l x_h && \text{(B.2)} \\
&= \sum_{i \in \mathsf{C2}} \sum_{j \in \mathsf{S2}} \sum_{k \in \mathsf{C1}} \sum_{l \in \mathsf{S1}} \sum_{h=1}^{n} \beta_i \delta_j^i \beta_k^j \delta_l^k \beta_h^l x_h && \text{(B.3)} \\
&= \sum_{h=1}^{n} \sum_{i \in \mathsf{C2}} \sum_{j \in \mathsf{S2}} \sum_{k \in \mathsf{C1}} \sum_{l \in \mathsf{S1}} \beta_i \delta_j^i \beta_k^j \delta_l^k \beta_h^l x_h && \text{(B.4)} \\
&= \sum_{h=1}^{n} \gamma_h x_h. && \text{(B.5)}
\end{aligned}$$

For Equation (B.3), I define

$$\mathsf{C2} \stackrel{\mathtt{def}}{=} \{i \ : U_i \in \text{layer C2}\}$$

and similarly for $\mathsf{S2}, \mathsf{C1}$ and $\mathsf{S1}$. I further extend the definition of $\delta_j^i$ to be 0 if $j \neq \mathsf{ch}\,(U_i)$, and similarly for the $\beta$ variables. (As discussed in the proof of Theorem 3, this extension is merely conceptual, and does not change the computation performed by the network.) Equation (B.4) comes from swapping the order of the summations, and Equation (B.5) yields the definition

$$\gamma_h \stackrel{\mathtt{def}}{=} \sum_{i \in \mathsf{C2}} \sum_{j \in \mathsf{S2}} \sum_{k \in \mathsf{C1}} \sum_{l \in \mathsf{S1}} \beta_i \delta_j^i \beta_k^j \delta_l^k \beta_h^l.$$

Applying Theorem 3 to Equation (B.5), the contribution of an input (or pixel) $x_l$ is given by

$$\begin{aligned} \mathcal{C}\,(x_h) &= \gamma_h x_h \\ &= \sum_{i \in \mathsf{C2}} \sum_{j \in \mathsf{S2}} \sum_{k \in \mathsf{C1}} \sum_{l \in \mathsf{S1}} \beta_i \delta_j^i \beta_k^j \delta_l^k \beta_h^l x_h \end{aligned} \tag{B.6}$$

I use Equation (B.6) to calculate the contributions of the pixels in Section 5.2.2. Importantly, there are no denominators in this form, and therefore there is no possibility for division by zero.

# Appendix C

# Implementation Details of the Sparse Coding Algorithms

Given an integer $s$, a signal $\mathbf{x} \in \mathbb{R}^n$, and the dictionary $\Phi \in \mathbb{R}^{n \times m}$, all algorithms search for a $\mathbf{z}$ such that

$$\mathbf{x} \approx \Phi\mathbf{z} \quad \text{and} \quad \|\mathbf{z}\|_0 \leq s.$$

Some algorithms, such as the Difference Map (DM), consider each of two constraints separately; others combine them into a single objective, as in Equation (7.3) or (7.4).

I implemented the Difference Map in Matlab (Landecker, 2013). All experiments were performed on a computer with a 3 GHz quad-core Intel Xeon processor, running Matlab R2011a. I obtained Matlab implementations of LARS and StOMP from SparseLab v2.1 (Stodden et al., 2007). Implementations of AIHT (Blumensath, 2012) and Subspace Pursuit (Dai and Milenkovic, 2009) were found on the websites of the papers' authors. I also obtained Matlab implementations of ADMM (Chartrand and Wohlberg, 2013) and IRLS (Chartrand and Yin, 2008) directly from the authors of

the cited papers.

The implementations of LARS, SP, AIHT, and StOMP are parameter-free[1]. It was necessary to tune a single parameter ($\beta$) for DM, and two parameters each for ADMM and IRLS. I tuned the parameters in two iterations of grid search. ADMM and IRLS required different parameters for the two different experiments presented in the next sections (sparse coding with natural images, and compressed sensing with random measurements). Interestingly, DM performed well with the same parameter value for both types of experiments.

I use training matrices of the same dimension, sparsity, and noise level as the ones appearing in the subsequent experiments in order to tune parameters. I chose parameters to minimize the NRMSE of the estimate $\hat{\mathbf{x}}$, averaged over all training problems. When tuning parameters for natural image reconstruction, I used a training set of 1000 image patches taken from the *person* and *hill* categories of ImageNet (Deng et al., 2009), providing a good variety of natural scenery. Example images are shown in Figure D.1.

When tuning $\beta$ for DM, I first perform grid search with an interval of 0.1, between $-1.2$ and $1.2$[2]. Next, in a radius of 0.5 around the best $\beta$, I performed another grid search with an interval of 0.01. Surprisingly, all $\beta$ in the interval $[-0.9, -0.1]$ appeared to be equally good for all problems reported in this dissertation. I chose $\beta = -0.14$ because it performed slightly better during my experiments, but the advantage over other $\beta \in [-0.9, -0.1]$ was not significant.

I used logarithmic grid search to tune the two parameters for ADMM and IRLS.

[1]This is somewhat of a simplification. SP, LARS and StOMP ask for the desired sparsity level $\|\mathbf{z}\|_0$. However, I will treat $\|\mathbf{z}\|_0$ as hyper-parameters, which all algorithms will have access to.

[2]Elser et al. (2007) claim that the natural range for the parameter $\beta$ is [-1,1] (excluding 0), but that occasionally values outside of this interval work well.

First, I searched parameter values by powers of ten, meaning $10^{\alpha}$, for $\alpha = -5, -4, \ldots, 5$. I then searched in the neighborhood of best exponent $c$ by $\frac{1}{10}$ powers of ten, meaning $10^{c+\alpha}$ for $\alpha = -0.5, -0.4, \ldots, 0.5$.

For random measurements (the experiments in Section 12), this results in parameter values $\mu = 1.26 \times 10^2, \lambda = 3.98 \times 10^{-1}$ for ADMM, and $\alpha = 3.16 \times 10^{-3}; \beta = 2.51 \times 10^{-1}$. For natural image reconstruction (Section 11), we found $\mu = 1.58 \times 10^2, \lambda = 1.0 \times 10^{-1}$ for ADMM and $\alpha = 2.5 \times 10^{-4}, \beta = 5 \times 10^{-3}$ for IRLS. Note that the $\beta$ parameter for IRLS has nothing to do with the $\beta$ parameter for DM. I refer to both as $\beta$ only to remain consistent with the respective bodies of literature about each algorithm, but in the rest of the dissertation I refer to the parameter for DM. IRLS is capable of addressing the $\ell^p$ quasi-norm for a variety of values $0 < p \leq 1$, while ADMM uses modifications of the $\ell^p$ quasi-norm designed to have a simple proximal mapping (Chartrand, 2012). In both cases I tried $p = \frac{1}{2}$ and $p = 1$, and found $p = \frac{1}{2}$ to perform better.

# Appendix D

# Dictionary Learning for Sparse Image Reconstruction

Rick Chartrand and I decided on the following details of the dictionary. After discussing these details, Rick Chartrand performed the actual dictionary learning. ADMM was used as the sparse-coding algorithm[1].

The dictionary is trained with 10 million $20 \times 20$ image patches, and we choose to learn 1000 atoms, resulting in a dictionary of size $400 \times 1000$. The dictionary is trained with patches from the *person* and *hills* category of ImageNet (Deng et al., 2009), which provide a variety of natural scenery (see Figure D.1 for images). The training alternates sparse coding using 20 iterations of ADMM using $p$-shrinkage with $p = 1/2$ (see Chartrand (2012) for details), with a dictionary update using the method of optimal directions (Engan et al., 1999).

Using 1,000 processors, the dictionary converged in about 2.5 hours. The training

[1]This does not give an unfair advantage to ADMM, because the reconstructed images presented in this paper are separate from the dataset used to train the dictionary.


Figure D.1: Examples images from the *person* and *hill* categories of ImageNet (Deng et al., 2009), used to learn the dictionary $\Phi$ in this Section, and to estimate parameters for various algorithms, as described in Appendix C.

patches were reconstructed by the dictionary with an average of 29 nonzero components (out of 1000), and the reconstruction of the training images had a relative error of 5.7%.