Fast optimization of non-convex Machine Learning objectives

Nikolaos Nikolaou
Master of Science,
Artificial Intelligence,
School of Informatics,
University of Edinburgh

August 17, 2012

Abstract

In this project we examined the problem of non-convex optimization in the context of Machine Learning, drawing inspiration from the increasing popularity of methods such as Deep Belief Networks, which involve non-convex objectives. We focused on the task of training the Neural Autoregressive Distribution Estimator, a recently proposed variant of the Restricted Boltzmann Machine, in applications to density estimation. The aim of the project was to explore the various stages involved in implementing optimization methods and choosing the appropriate one for a given task. We examined a number of optimization methods, ranging from derivative-free to second order and from batch to stochastic. We experimented with variations of these methods, presenting along the way all the major steps and decisions involved. The challenges of the problem included the relatively large parameter space and the non-convexity of the objective function, the large size of some of the datasets we used, the multitude of hyperparameters and decisions involved in each method, as well as the ever-present danger of overfitting the data. Our results show that second order Quasi-Newton batch methods like L-BFGS and variants of stochastic first order methods like Averaged Stochastic Gradient Descent outshine the rest of the methods we examined.

Acknowledgements

I would like to extend my thanks to my supervisor Dr. Iain Murray for his guidance and patience. Every piece of information he provided was to-the-point and saved me many hours of work. From theoretical background, to implementation details, to writing and typesetting style, the end result would be of far lesser quality, in every aspect, without his feedback.

I also thank my family and all my friends for their love and support. Special thanks go to my friends Costas and Stratis who lended me not only strength but also their computing power for some of the experiments.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Contents

Chapter 1

Introduction and Outline of the Dissertation

1.1 Introduction

Optimization plays a significant role in the field of Machine Learning. "Learning" usually means "finding the values of the parameters of our model for it to best fit the data". This in turn is usually translated mathematically as an optimization problem. In Maximum A Posteriori we maximize a posterior probability. In Maximum Likelihood we maximize a likelihood. In a Support Vector Machine we maximize the gap between two categories (margin) so as to better distinguish between them. In many forms of clustering we minimize a distance. In classification and regression tasks we usually minimize an error function and in cost-sensitive classification (where each type of misclassification has a different cost associated with it) we are minimizing a total cost (the weighted sum or average of the errors of each type weighted by the error types' costs). As a result, improving the speed of the optimization algorithm that lies in the heart of a Machine Learning technique will make the entire procedure faster, as —in most cases— this is where the bottleneck of the method is.

Although the majority of problems commonly faced in Machine Learning are convex, there also exist many methods based on the optimization of non-convex objective functions. In non-convex functions a local minimum is not necessarily the global minimum, so we do not only care about the speed of convergence of the optimization algorithm, but also about the quality of the local optima it finds. Examples of Machine Learning techniques that involve the optimization of non-convex objectives are most types of Deep Belief Networks(DBNs), models that are becoming increasingly popular as of late.

In this project we will use a probabilistic graphical model, the Neural Autoregressive Density Estimator (NADE) which can be used as a building block of DBNs. We will train NADE to perform density estimation, i.e. to estimate the underlying probability density function given a number of datapoints (an unsu-

pervised task). The objective function used for training NADE is non-convex, its parameter space is quite large and the datasets we will train it on contain –in some cases– a large amount of instances. All these characteristics of the problem need to be takin into account when choosing which optimization method we will use.

We will examine a range of different optimization algorithms from simple and fairly inappropriate for convex settings like Steepest Gradient Descent and Coordinate Descent to better-suited ones for the problem such as Stochastic Gradient Descent and L-BFGS. Judging from the size of the parameter space, Coordinate Descent is probably a poor choice, large datasets favor stochastic approaches and the non-convex nature of the objective suggests that stochastic methods and second order batch methods are the most likely to yield good quality optima. We will present all methods in detail, analyzing their theoretical underpinnings, their possible variations and the problems for which they are most appropriate. We will present detailed pseudocode for each and the practical considerations of their implementation.

When designing an optimization algorithm we need to take special considerations to ensure its correctness and its termination. Many optimization algorithms have hyperparameters which need to be adjusted to appropriate values for them to work correctly. Finally, there is a number of heuristics and modifications that can be applied to an optimization algorithm, to boost its performance. And once we have implemented our algorithms we need to compare them not only in terms of quality of optima (for non-convex objectives), but also in terms of speed of convergence and in terms of simplicity (as in "fewer hyperparameters involved"). In this project we will cover all these aspects of implementing and evaluating optimization methods. We will describe both the theoretical background of each aspect as well as practical considerations that needed to be taken into account.

1.2 Outline of the Dissertation

In Chapter 2 we will introduce the basic theoretical background behind this dissertation. Our intention is for it to act as both a theoretical basis for the rest of the chapters and as a source of motivation for studying non-convex optimization in the context of machine learning. We will begin by presenting the general form of an optimization problem and introducing some basic terminology, including the distinction between convex and non-convex optimization problems and the importance of parameter regularization. We will see how often optimization problems arise in machine learning tasks and what distinguishes them from optimization problems in general.

We will then move on to describing the basics of Deep Belief Networks (DBNs), since they represent a successful recent trend in machine learning where the objectives generally are non-convex. We will mention their basic structure, as well as the motivation for using them drawing analogies between the principles that

govern them and how learning appears to work in humans. We will outline the principles of training DBNs and how the non-convexity of their objectives arises. We will then briefly present Restricted Boltzmann Machines and how they are used as building blocks for DBNs. We will only go as deep as introducing their limitations and how other models —ones we will cover in more depth— can overcome them. Throughout these discussions we will offer examples of applications and pointers to bibliography which emphasize the wide range of applications and the power of DBNs.

Finally, we will present the specific models we will be working with, emphasizing on aspects that will be examined throughout the dissertation. Main focus will revolve around their respective advantages and disadvantages over other models, their parameters, their objective functions and their gradients with respect to their parameters and describe in brief the ones we will be comparing our results to, the convexity of their objectives, their computational complexity, the role of overfitting and regularization in practice. The principal focus of this project will be the the Neural Autoregressive Density Estimator (NADE) with the presentation of which, the chapter closes.

Chapter 3 is a combination of theory and application. Here we will delve deeper and more formally into the subject of optimization and more specifically in large scale non-convex optimization problems, as is the task of training NADE with the specific datasets we use. We will compare the benefits and shortcomings of batch versus stochastic methods as well as what we gain and what we lose by using methods of different order.

We will then choose a number of specific methods to examine in more depth, most of which will include both the basic version and variants based on various heuristics. We will present each version of each method in detail, including the motivation behind it, relative benefits and disadvantages over other methods, the role of their hyperparameters and some possible variations. We will then discuss our specific implementation for each version, including justification for each of our decisions and present pseudocodes for all of them.

Chapter 4 will discuss hyperparameter selection in more depth. This chapter continues the theme of combining theory and application. This is so, because there are theoretical and experimental guidelines for this procedure, yet also practical considerations, like keeping the execution time manageable were taken into account. We will discuss termination criteria, choices about which hyperparameters we fixed to specific values and why and which ones we tried to adjust in each method and what subspace of the hyperparameter space we explored. We will discuss how we evaluate our hyperparameter values' choices, how we safeguard against overfitting and how we chose the termination criteria of our methods. In most cases, besides describing the specific hyperparameter selection scheme we used, we will also discuss alternatives, some of which we explored up to some point.

In Chapter 5 we will test some of our techniques by applying them on a toy problem, more specifically on performing binary classification using logistic regression.

We will also present the steps taken to ensure that the gradient and the objective function value computation code for NADE works properly, as well as a couple of very simple baselines for the value of the objective function.

In Chapter 6 we will present the datasets we will train NADE on, discuss the experimental design for obtaining the results and present detailed results of each method on each dataset. We also present baselines produced by other models in other studies on the same datasets for comparisons. We will then examine the ability of the methods studied to reach good local optima as well as the execution time they require. We will analyze how different factors affect the affects the performance of NADE trained under each method and compare the advantages and disadvantages of each one of them in practice.

In Chapter 7 we will present our conclusions from these explorations. We will also discuss paths not traveled that could possibly offer fertile ground for further work.

Chapter 2

Background

2.1 Optimization and its Role in Machine Learning

An *optimization problem* is generally the problem of finding the best solution from all *feasible solutions*. We will briefly discuss the categorizations of optimization problems in the beginning of the next chapter. Here we will restrict ourselves to presenting the general form of continuous optimization problems and their role in Machine Learning. In a continuous optimization problem, our goal is to either minimize or maximize the value of a function $f(\boldsymbol{\theta}) : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}$, called the *objective function*, possibly under a number of *inequality* and/or *equality constraints* ($g_i(\boldsymbol{\theta}) \leq 0$ and $h_i(\boldsymbol{\theta}) = 0$, respectively). The standard definition of a *continuous* optimization problem [Boyd et al. , 2004] is

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{minimize}} \quad & f(\boldsymbol{\theta}) \\ \text{subject to} \quad & g_i(\boldsymbol{\theta}) \leq 0, \ i = 1, \ldots, m_1 \\ & h_i(\boldsymbol{\theta}) = 0, \ i = 1, \ldots, m_2. \end{aligned} \tag{2.1}$$

Any continuous optimization problem can be expressed in the form above. Constraints of the form $g_i'(\boldsymbol{\theta}) \geq 0$ can be rewritten as $-g_i'(\boldsymbol{\theta}) \leq 0$. If the right hand side is not equal to zero, but a constant $b_i \neq 0$, we can simply subtract b_i from both sides to obtain the standard inequality constraint form. Finally, a maximization problem with objective function $f(\boldsymbol{\theta})$ can be stated as the minimization of $-f(\boldsymbol{\theta})$ subject to the same constraints.

Solving optimization problems, usually involves starting at some point in the *parameter space* and searching for close *local optima*. One way to categorize optimization problems is based on whether or not the objective function is *convex*. In convex objective functions, any local optimum has to be a *global optimum*. In *non-convex* objectives[1] we have more than one local optima. As a result, while

[1]From time to time we may refer to the value of the objective function as the "cost" or the

methods that search for the closest local optimum will succeed in finding the global optimum of a convex objective, in non-convex objectives the success is not guaranteed, as we might get trapped in a local optimum. In this sense, non-convex problems are "harder". In Figure 2.1 we can see two univariate functions, a convex and a non-convex one.

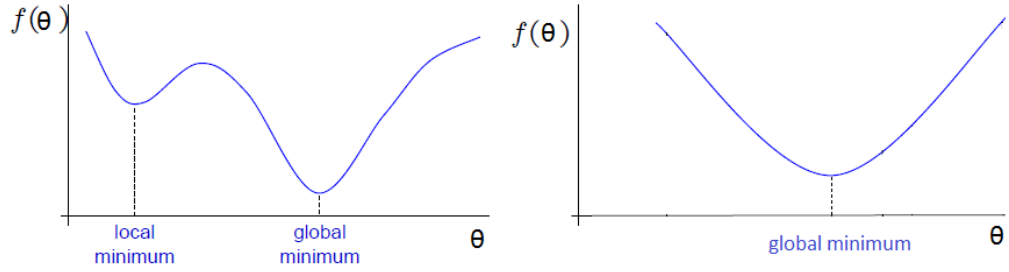

Figure 2.1: A non-convex (left) and a convex (right) function of one variable. Notice in the convex case the local minimum is also the global minimum.
Image adapted from images in lecture 4 of the course *B1 Optimization* from University of Oxford, by Andrew Zisserman
`http://www.robots.ox.ac.uk/~az/lectures/b1/lect4.pdf`.

In the Introduction we stressed the importance of optimization in Machine Learning and gave a few examples to illustrate how common a subproblem in learning algorithms it is. Here we should clarify that optimization in the context of learning is not the same as solving an optimization problem. Our real goal is not to minimize the value of the objective function on the training set, but to train a model that can *generalize* on new data. When this generalization capability is lost because the model has adjusted its parameters so as only to fit the training data, the model suffers from *overfitting*. In practice this often means that some parameters have assumed extremely large values compared to others. *Regularization* is a way to protect training against overfitting. To implement parameter regularization, we can add a term to the objective function that encourages the parameters to assume small values (by penalizing big ones) or that forces many of the parameters to assume zero values (encourages *sparse* solutions). A way to encourage small parameter values for example is $\mathcal{L}2$*-regularization*,

$$f'(\boldsymbol{\theta}) = f(\boldsymbol{\theta}) + \lambda \, ||\boldsymbol{\theta}||_2^2, \quad ||\boldsymbol{\theta}||_2 = \sqrt{\sum_{i=1}^{N} \theta_i^2}. \tag{2.2}$$

Here we formed a new objective function $f'(\boldsymbol{\theta})$, by adding to the original objective function $f(\boldsymbol{\theta})$ the square of the $\mathcal{L}2$*-norm* of $\boldsymbol{\theta}$, weighted by a factor $\lambda \in \mathbb{R}^+$, which controls how harshly we penalize large values (big values of λ mean harsh penalization, thus heavier regularization). Se the new objective balances the old

"error" or simply the "objective" without properly introducing the term. We hope it will be clear that we mean the exact same thing.

one (minimize $f(\boldsymbol{\theta})$ wrt $\boldsymbol{\theta}$) and the regularization term (keep parameter values as small as possible).

A way to encourage sparseness is $\mathcal{L}1$-*regularization*, where we use the $\mathcal{L}1$-*norm* of $\boldsymbol{\theta}$, instead,

$$f'(\boldsymbol{\theta}) = f(\boldsymbol{\theta}) + \lambda \left|\left|\boldsymbol{\theta}\right|\right|_1, \quad \left|\left|\boldsymbol{\theta}\right|\right|_1 = \sum_{i=1}^{N} |\theta_i|. \tag{2.3}$$

In this project we will use unregularized objectives. Our way of preventing overfitting will be by use of a statistical test (see Chapter 3) or in most cases by *early stopping* (see Chapter 4) .

To close this brief introduction to optimization in machine learning, we should note that we usually choose to use convex objectives instead of non-convex ones, whenever we have the chance, as their global optimum is easier to find. Especially since 2006, however, *Deep Belief Networks* have been growing in popularity and —largely due to that— *Neural Networks* have been experiencing a rebirth as well. In both cases, the objective functions for training them are usually non-convex.

2.2 Deep Belief Networks

Deep Belief Networks (DBNs) [Hinton et al., 2006a] are probabilistic generative models that consist of more than one layers of stochastic, latent variables, which typically assume binary values. We call such variables *hidden units* or *feature detectors* and we call the layers they compose *hidden layers*. The inputs of the first hidden layer are the elements of the data vector and the inputs of the second one and on are the activations of its preceding layer. Each hidden layer combines its inputs *nonlinearly* to transform them into new features that will be provided as inputs to the level above. The top two layers have undirected, symmetric connections between them and form an associative memory.

So in DBMs we perform feature extraction at multiple levels by combining the features from the level below in a nonlinear fashion. An example with image data would be the following: Imagine we have T grayscale images of 100×100 pixels and let us assume our initial features are the $D = 10000$ intensity values of the pixels. The D pixels are combined in a nonlinear fashion to form H new features. These features are no longer pixels but nonlinear combinations of pixels, we can think of them as edges, shapes, forms of any kind within an image which represent a different level of abstraction. Then the next layer combines these edges and basic shapes to form new features, which represent an even higher level of abstraction, like entire objects, for instance. As we see, DBNs can extract features of multiple levels of abstraction. This property allows them to approximate any distribution over binary vectors to arbitrary accuracy [Sutskever et al., 2008], [Bengio, 2009a]. In Figure 2.2 we can see features learned at different levels of a Deep Belief Network in [Lee et al., 2011] in image recognition tasks.

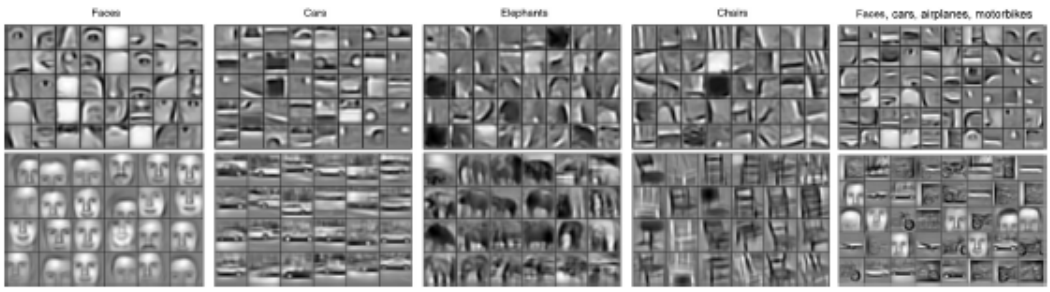

Figure 2.2: An illustration of the features detected at multiple levels from a DBN. Top row shows features learned at the second hidden layer of the convolutional DBN (CDBN) presented in [Lee et al., 2011]. Bottom row shows features learned at the third hidden layer of the network. The first four columns correspond to a training the CDBN on different categories of objects while the last column shows the features learned by training on a mixture of images from all categories. Notice the different levels of abstraction at different layers of the network, the 'intuitiveness' of the features learned (e.g. eyes and mouths of faces, wheels and doors of cars, etc.) as well as the coherence of the features learned even when objects of different categories are presented to the network.
Image from [Lee et al., 2011].

Most likely humans perform similar forms of feature extraction [Morrone et al., 1988], [Watson, 2000]. After all, when looking at an image we observe shapes and distinct regions within it, edges and repeated patterns, not just colors, although the initial information our eyes receive are packets of photons. Similarly we understand language as ideas, concepts and interactions among them, although its building blocks are simple phonemes. In fact, we even use different levels of abstraction. We don't only recognize regions of different color or texture in an image, we recognize parts of objects (like the eyes and nose of a person), entire objects (like faces).

Since the goal of Artificial Intelligence (AI), is to develop techniques that allow machines to perform such tasks at a level comparable to that of humans, it is no surprise that DBNs which can allow for such multilevel abstraction are becoming more and more prevalent in subfields of AI such as Machine Vision (e.g. generating and recognizing images [Bengio et al., 2007], [Ranzato et al., 2007], [Hinton et al., 2006b], [Lee et al., 2011], video sequences [Sutskever et al., 2007], and motion-capture data [Taylor et al., 2007]), Speech [Lee et al., 2009] and Natural Language Processing [Collobert et al., 2009].

The task of training DBMs is not an easy one. First of all, DBMs by definition have many layers and a common practice is to train each layer greedily and move on to the next one. A second difficulty is that in order to search the parameter space of DBNs, the objective functions we need to optimize tend to be non-convex for two main reasons: (i) the outputs of each layer are non-linear transformations of its inputs and (ii) due to symmetries in the networks, we can permute elements

of the weight matrices without changing the network's output. Of course there are some models that use convex objective functions like the one presented in [Deng et al., 2011], but they are the exception, rather than the rule.

And now, an interesting side note. Since we have already made an analogy of the way humans learn to the feature extraction at various abstraction levels performed by DBNs, let us make another one that illustrates the importance of non-convex optimization in learning. When we humans learn, the way we are presented the examples matters. This might imply that learning in humans also involves solving non-convex optimization problems, where different order of presenting the examples will result in ending up in different local optima. In [Bengio, 2009b] the authors stress this analogy, and present a formalization of such strategies for machine learning called "curriculum learning". One possible explanation is that curriculum learning is a special case of *continuation methods*.

Continuation methods are a strategies used in non-convex optimization methods for finding global optima. We start with a smooth relaxation of the original problem and gradually consider less smooth versions of it. Ideally, the global optimum of the relaxation will also be the global optimum of the original problem. The authors suggest that using curriculum learning techniques we can speed up convergence, allow us to reach better local optima, even act as regularizers. In this work we will make use of any of form of curriculum learning or continuation methods. However perhaps in a future work, such ideas could be explored on this problem, or similar ones involving non-convex objectives. We just raised this point to show that non-convex objectives have a great potential in Machine Learning. Analyzing the ways to optimize them and understanding the merits and drawbacks of each —which is the aim of this project— is an important step towards a broader use of non-convex objectives in learning tasks.

2.3 Restricted Boltzmann Machines

Let us now return to DBMs and see how they are constructed. A typical building block of DBMs is the *Restricted Boltzmann Machine (RBM)*. The RBM is a type of stochastic recurrent neural network introduced by [Smolensky, 1986].

In a RBM we have a number of 'observation' or 'visible' units x_d, $d = 1, 2, ..., D$ each of which is connected to a number of hidden units h_k, $k = 1, 2, ..., H$ by a set of weights W_{dk}. Units of the same layer (visible, hidden) have no connections between them but each visible unit is connected to every hidden unit (and vice-versa). So we have a bipartite graph and the weight matrix $\boldsymbol{W}$ is symmetric. The visible units also have a set of bias parameters $\boldsymbol{b} \in \mathbb{R}^D$ and the hidden units a set of bias parameters $\boldsymbol{c} \in \mathbb{R}^H$ associated with them.

We can construct a DBN by 'stacking' RBMs on top of one another [Hinton et al., 2006a], [Sutskever et al., 2008]. After training one RBM, the activations of its hidden units can be treated as the visible units of a higher-level RBM and so on, allowing us to greedily train each layer. We alternatively refer to

the units as 'nodes', 'neurons' or 'variables', following the terminology of Probabilistic Graphical Models in general and specifically Neural Networks. In Figure 2.3 we can see the graphical model of a RBM and a 3-layer DBM constructed by RBMs.

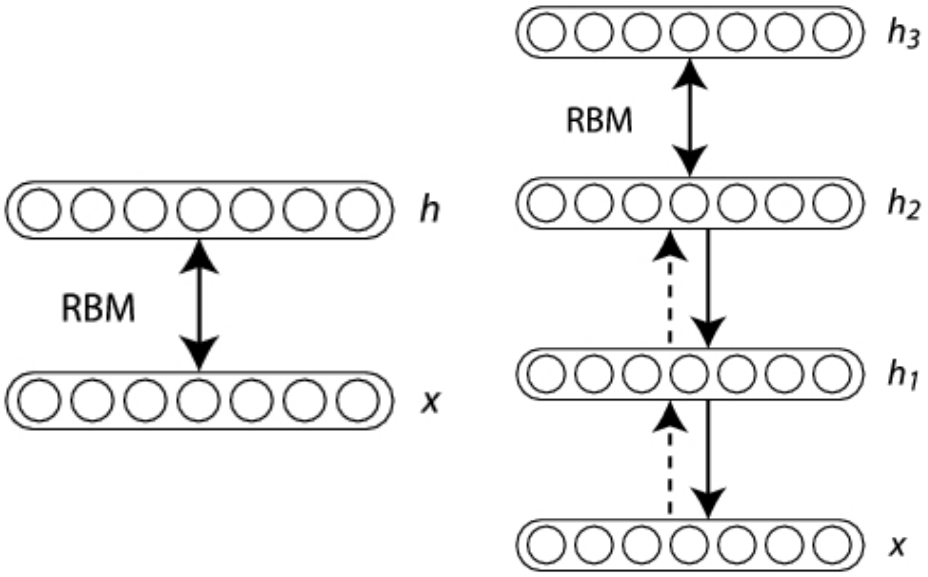

Figure 2.3: Graphical model of a RBM (left) depicted as a bipartite graph among the input $\boldsymbol{x}$ and hidden units $\boldsymbol{h}$. Graphical model of a a 3-layer DBM using RBMs as its building blocks (right). To train the DBN we greedily train each hidden layer (units $\boldsymbol{h_i}$) and move on to the next one (units $\boldsymbol{h_{i+1}}$).
Images from Yoshua Bengio's research webpage:
`http://www.iro.umontreal.ca/~bengioy/yoshua_en/research.html`

To any observation $\boldsymbol{x}$ we assign probabilities,

$$p(\boldsymbol{x}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{x},\boldsymbol{h})}, \tag{2.4}$$

where

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\boldsymbol{b}^T \boldsymbol{W} \boldsymbol{v} - \boldsymbol{x}^T \boldsymbol{h} - \boldsymbol{c}^T \boldsymbol{h} \tag{2.5}$$

is called the 'energy function', with high probability configurations corresponding to low energy. The denominator Z, known as the 'partition function', normalizes the distribution, so that $\sum_{\boldsymbol{v}} p(\boldsymbol{v}) = 1$. Training an RBM means finding configuration of its parameters $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}\}$ that maximizes the likelihood of the model (equivalently minimizes the negative likelihood).

However, in most interesting cases, i.e. when neither the number of visible units D nor the number of hidden units H is very small, the normalization factor Z

becomes intractable. We will not discuss RBMs in more detail here. A useful source of information about RBMs and their training is [Hinton, 2010]. Many of the tricks and heuristics regarding the training of RBMs from the aforementioned paper carry on to the particular model we will make use of many of its general guidelines in Chapters 3 and 4.

The model we will be working in this project is the Neural Autoregressive Distribution Estimator (NADE), presented in [Larochelle et al., 2011]. NADE, transforms the estimation of the partition function to a tractable problem, by converting the RBM into a *Bayesian Network*. The discussion is restricted to binary variables only, however the model can be extended to non-binary cases as well. More precisely, we are going to examine various non-convex optimization techniques in the context of training NADE to perform *density estimation*. NADE can be viewed as an *autoencoder* (it tries to reproduce each data vector from the feature activations that they cause), whose output can be used to assign valid probabilities to the observations. Therefore NADE can be used as a density estimator.

2.4 Density Estimation

Density estimation, the estimation of the distribution $p(\boldsymbol{x})$ given a number of datapoints $\boldsymbol{x}$, is of great importance as a machine learning task. Some of its most notable applications are mentioned in [Zoran et al., 2011] and include estimating missing features (for example 'inpainting' missing pixels in an image), image processing applications like denoising and deblurring, and performing unsupervised feature extraction as a first step to solve other tasks (e.g. in Natural Language Processing, Speech Recognition and Computer Vision that we discussed, in collaborative filtering [Salakhutdinov et al., 2007a], time series classification and anomaly detection [Wulsin et al., 2011], etc.).

RBMs, NADE, DBMs are all generative models, we can therefore use them to learn the distribution $p(\boldsymbol{x})$. Note that we use the terms "density" and "distribution" somewhat loosely. The probability density function applies to continuous random variables, and we will be dealing with discrete ones, so the correct term would be "probability mass function", i.e. the function that gives us the probability that the discrete random variable is exactly equal to some value.

2.5 Models Examined

Logistic Regression

In Chapter 5 we will test some of our optimization methods in a toy application of logistic regression before applying them to the harder task of training NADE. Furthermore, logistic regression is the building block of FVSBN, a model whose

performance we shall compare to that of NADE's. Finally, it is an opportunity to introduce terms and concepts we will be using throughout this dissertation like the definition and behaviour of the logistic sigmoid, the likelihood of a model, why we use loglikelihoods instead and how overfitting works in practice. For all of the reasons above, we shall briefly present the logistic regression model applied to solving a binary classification problem.

We are given T data pairs of the form $(\tilde{\boldsymbol{x}}^{(t)}, y^{(t)})$, where $\tilde{\boldsymbol{x}}^{(t)} \in \mathbb{R}^D$ is the t-th feature vector and $y^{(t)} \in \{0, 1\}$ is the corresponding target label, the class to which the $\tilde{\boldsymbol{x}}^{(t)}$ belongs.

We define $\theta_0 \in \mathbb{R}$ and $\tilde{\boldsymbol{\theta}} \in \mathbb{R}^D$. We will use $\tilde{\theta}_d$, $d = 0, 1, 2, ..., D$ as the parameters of the model, the variables of the model for which we must find the optimal values such that the model fits the data as well as possible. To measure how well the model fits the data, we must first define how we want it to classify an instance. We need a function that assigns to instance $\tilde{\boldsymbol{x}}^{(t)}$ a the probability that it belongs to each class. We shall use a sigmoid function $\sigma : [-\infty, \infty] \rightarrow [0, 1]$. More precisely, we shall use the logistic function. In one dimension it is,

$$\sigma_{\boldsymbol{\theta}}(x) = \frac{1}{1 + e^{-(\beta + \alpha\, x)}}, \tag{2.6}$$

where $\boldsymbol{\theta} = [\beta, \alpha] \in \mathbb{R}^2$. In Figure 2.4 we can see the logistic sigmoid function for $\beta = 0$ and various values of the parameter $\alpha \in \mathbb{R}$, which as we see controls the steepness of the sigmoid.

To simplify the notation we define the matrix

$$\boldsymbol{X} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \tilde{x}_1^{(1)} & \tilde{x}_1^{(2)} & \cdots & \tilde{x}_1^{(T)} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{x}_D^{(1)} & \tilde{x}_D^{(2)} & \cdots & \tilde{x}_D^{(T)} \end{pmatrix}$$

and the parameter vector

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \tilde{\theta}_1 \\ \vdots \\ \tilde{\theta}_D \end{pmatrix}$$

We can now define the probability that $\tilde{\boldsymbol{x}}^{(t)}$ belongs to class 1 as

$$P(y = 1 | \tilde{\boldsymbol{x}}^{(t)}; \boldsymbol{\theta}) = \sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}^{(t)}}} \tag{2.7}$$

So we classify $\boldsymbol{x}^{(t)}$ to class 1 if $P(y = 1 | \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) > 0.5$ and we assign it to class 0 if $P(y = 0 | \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) = 1 - P(y = 1 | \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) > 0.5$,

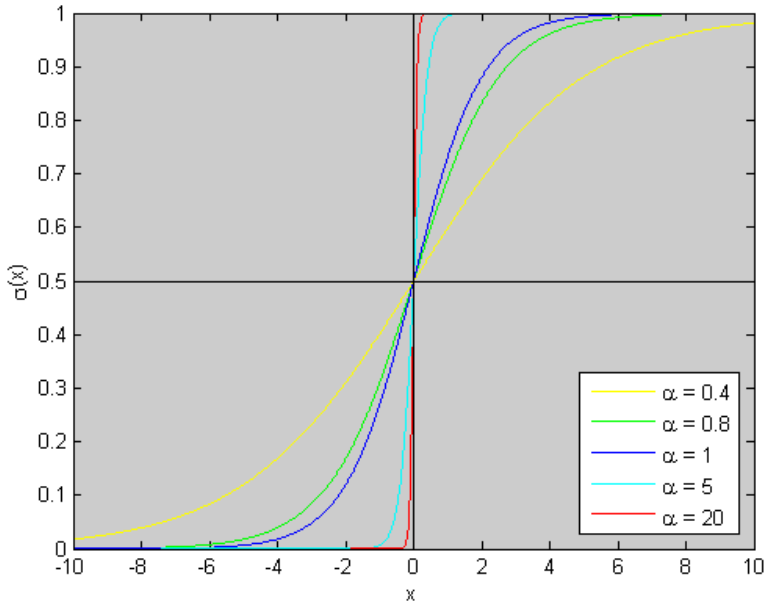

Figure 2.4: The logistic sigmoid function for various values of parameter α. As $|\alpha| \to \infty$, the sigmoid becomes a step function ($\sigma(x) = 0$, for $x < 0$ and $\sigma(x) = 1$, for $x > 0$). As $|\alpha| \to 0$, it becomes $\sigma(x) = 0.5$, $\forall x$. For $\alpha < 0$ we would just have a mirrored image, i.e. it would assign values values $\sigma(x) > 0.5$, for $x < 0$ and $\sigma(x) < 0.5$, for $x > 0$. Finally, for $\beta \neq 0$ we would have a displacement of the sigmoid on the x-axis.

We can measure how well the model fits the data using the likelihood of the logistic regression with parameters $\boldsymbol{\theta}$ under the data. The likelihood is defined as the probability of observing the given data under the model with parameters $\boldsymbol{\theta}$. Assuming the data are independent and identically distributed (iid) the likelihood in our case is

$$J'(\boldsymbol{\theta}) = \prod_{t=1}^{T} P(y = 1|\boldsymbol{x}^{(t)}; \boldsymbol{\theta})^{y^{(t)}} (1 - P(y = 1|\boldsymbol{x}^{(t)}; \boldsymbol{\theta}))^{(1-y^{(t)})} \tag{2.8}$$

This will be our objective function and our goal is to maximize it. It is a convex function. We usually prefer to work with sums rather than products. Very small or very large values which can arise in such products can cause numerical instabilities in an algorithm (underflowing or overflowing variables). Also, adding numbers causes less round-off errors to occur and propagate in the calculations. So we take the logarithm of $J(\boldsymbol{\theta})$ and get

$$\tilde{J}(\boldsymbol{\theta}) = \log(J'(\boldsymbol{\theta})) = \sum_{t=1}^{T} [-y^{(t)} \log(\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)})) - (1 - y^{(t)}) \log(1 - \sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}))] \tag{2.9}$$

We will use the Average Loglikelihood (ALL) of the logistic regression model with parameters $\boldsymbol{\theta}$ given the data,

$$J(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}[-y^{(t)}\log(\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)})) - (1-y^{(t)})\log(1-\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}))] \tag{2.10}$$

We will use the ALL as the objective function to be maximized in the next models we will present as well. $J(\boldsymbol{\theta})$ is a convex function of the parameters θ_d, $d = 0, 1, ..., D$. We will often use its negative, the Average Negative Loglikelihood (ANLL) as an objective to minimize, posing our optimization problem in the standard form we saw in Eq. (2.1).

The partial derivative of $J(\boldsymbol{\theta})$ with respect to parameter θ_d, $d = 0, 1, ..., D$, θ_0 being the intersect (bias) term is

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_d} = \frac{1}{T}\sum_{t=1}^{T}(\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}) - y^{(t)})x_d^{(t)} \tag{2.11}$$

Normally, some sort of regularization is applied to the objective of logistic regression. Notice in Figure 2.4 that the absolute values of the parameters θ_d, $d = 1, 2, ..., D$ control the steepness of the sigmoid in the corresponding dimension (the intersect (bias) term θ_0 just displaces it on the x-axis). The steeper the sigmoid, the farther the probabilities $P(y = 1|\boldsymbol{x}^{(t)}; \boldsymbol{\theta})$ and $P(y = 0|\boldsymbol{x}^{(t)}; \boldsymbol{\theta})$ get from 0.5, so the higher their difference becomes. In a sense, the more confident the classifier is about classifying to the one class or the other after seeing the feature that corresponds to this dimension. If one θ_d has an extremely high (compared to the other parameters) absolute value, the sigmoid essentially becomes a step function in this dimension, thus assigning the examples solely based on their value of x_d. With regularization, we prevent any θ_d from growing to an extreme value, thus dominating $\boldsymbol{\theta}^T\boldsymbol{x}^{(t)}$ and causing the classifier to ignore the rest of the features.

In Algorithm 1 we present the pseudocode for LOGREG(), a subroutine that computes the value of the ALL and its partial derivatives wrt the parameters θ_d, $d = 0, 1, 2, ..., D$ at a given point $\boldsymbol{\theta} = \boldsymbol{\theta^{(0)}}$ in the parameter space, for the given dataset $\boldsymbol{X_{train}}$ with labels $\boldsymbol{y}$ under the logistic regression model we presented above. These partial derivatives form the gradient $\boldsymbol{\nabla J(\theta)}$ of $J(\boldsymbol{\theta})$. We will

introduce the gradient and its use more formally in the next chapter.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{y}$, $\boldsymbol{\theta}$
Result: $J(\boldsymbol{\theta}), \boldsymbol{\nabla J(\theta)}$
$J(\boldsymbol{\theta}) = 0$; $\boldsymbol{\nabla J(\theta)} = ZEROS(SIZE(\boldsymbol{\theta}))$;
$T' = SIZE(\boldsymbol{X_{train}}, ROWS)$;
for $t = 1 : T'$ **do**
  $J(\boldsymbol{\theta^{(0)}}) = J(\boldsymbol{\theta}) + [-y^{(t)} \log(\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)})) - (1 - y^{(t)}) \log(1 - \sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}))]$;
  for $d = 0 : D$ **do**
    $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_d} = \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_d} + (\sigma_{\boldsymbol{\theta}}(\boldsymbol{x}^{(t)}) - y^{(t)}) x_d^{(t)}$;
  end
end
`// Return averages instead of sums:`
$J(\boldsymbol{\theta^{(0)}}) = \frac{1}{T'} J(\boldsymbol{\theta^{(0)}})$;
$\boldsymbol{\nabla J(\theta^{(0)})} = \frac{1}{T'} \boldsymbol{\nabla J(\theta^{(0)})}$

Algorithm 1: The pseudocode for a simple, non-vectorized version of the algorithm that computes the value of the ALL of a logistic regression model and its partial derivatives wrt the parameters θ_d, $d = 0, 1, 2, ..., D$: LOGREG().

Fully Visible Sigmoid Belief Network (FVSBN)

The Fully Visible Sigmoid Belief Network (FVSBN), can be used as a density estimator and it is based on the idea of arranging the variables into a Bayesian Network, that is, a directed acyclic graph (DAG) that represents the joint probability $p(\boldsymbol{x})$. This idea was first proposed in [Frey, 1998] as the logistic autoregressive Bayesian network and then [Bengio et al., 2000] presented a non-linear generalization. Using the chain rule of probability, we decompose $p(\boldsymbol{x})$ into

$$p(\boldsymbol{x}) = \prod_{d=1}^{D} p(x_d | \boldsymbol{x_{<d}}), \tag{2.12}$$

where $\boldsymbol{x_{<d}}$ is the subvector containing all variables $x_i : i < d$, assuming that $x_i : i < d$ are the parents of x_d in the Bayesian Network. Note that the ordering of the features x_d, $d = 1, 2, ..., D$ need not necessarily be the initial one as our notation might indicate. To avoid overloading our notation let us just assume that we have chosen an appropriate or a random ordering of the elements of $\boldsymbol{x}$ before constructing the DAG.

The individual conditional probabilities $p(x_d | \boldsymbol{x_{<d}})$, $d = 1, 2, ..., D$ are tractable (therefore so is $p(\boldsymbol{x})$) and they are given by logistic regression,

$$p(x_d | \boldsymbol{x_{<d}}) = \sigma(b_d + \sum_{j<d} W_{dj} v_j). \tag{2.13}$$

We can use the Average Loglikelihood (ALL) of the model as the objective function to be maximized,

$$J(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}\log(p(\boldsymbol{x}^{(t)})) = \frac{1}{T}\sum_{t=1}^{T}\sum_{d=1}^{D}\log(p(x_d|\boldsymbol{x_{<d}})), \tag{2.14}$$

where the parameter vector is $\boldsymbol{\theta} = [\boldsymbol{b}, \boldsymbol{W(*)}] \in \mathbb{R}^N$, and by $\boldsymbol{W(*)}$ we denote the column vector that we get by concatenating all columns of the weight matrix $\boldsymbol{W}$. The total number of parameters is $N = D + D \times D = D(D+1)$. Note that the objective is convex since it is a sum of convex functions (The FVSBN model is practically D separate logistic regressors). Again usually a regularization term is added to the objective function, typically with a different regularization constant for each of the D regressors.

Neural Autoregressive Density Estimator (NADE)

NADE is another model that can be used for density estimation. It is based on transforming the RBM into a Bayesian Network. Again we use the chain rule to decompose $p(\boldsymbol{x})$ as in Eq. (2.12). In NADE, however, we interject a hidden layer between the input and the output layer. Thus, the model needs connection weights and biases both to map from the input layer to the hidden layer ($\boldsymbol{W} \in \mathbb{R}^{H\times D}$ and $\boldsymbol{b} \in \mathbb{R}^H$, respectively) and synaptic weights and biases to connect the hidden units to the output units ($\boldsymbol{V} \in \mathbb{R}^{D\times H}$ and $\boldsymbol{b} \in \mathbb{R}^D$, respectively). We can use any sigmoid activations but let us use again the logistic sigmoid we have been using all this time. The mapping from input to hidden layer is thus performed by computing

$$\boldsymbol{h_d} = \sigma(\boldsymbol{c} + \boldsymbol{W_{*,<d}}\boldsymbol{x_{<d}}) \tag{2.15}$$

and the mapping from hidden layer to output layer is done by taking

$$p(x_d = 1|\boldsymbol{x_{<d}}) = \sigma(b_d + \boldsymbol{V_{d,*}}\boldsymbol{h_d}) \tag{2.16}$$

The whole procedure corresponds to a Neural Network for each $p(x_d = 1|\boldsymbol{x_{<d}})$, where the weighted connections going in and out of the hidden layer of each mini-neural network are tied. In addition to that, connections are also tied across the individual neural networks, so that the difference between two consequent hidden layer activations is

$$(\boldsymbol{c} + \boldsymbol{W_{*,<d+1}}\boldsymbol{x_{<d+1}}) - (\boldsymbol{c} + \boldsymbol{W_{*,<d}}\boldsymbol{x_{<d}}) = \boldsymbol{W_{*,d+1}}x_{d+1}\,, \tag{2.17}$$

which can be computed in $O(H)$ time and we have D conditionals in the factorization of the joint probability $p(\boldsymbol{x})$. So, computing the $p(\boldsymbol{x})$ under NADE takes $O(HD)$ time overall. Contrast that with the $O(HD^2)$ time complexity

of an approach that would not take into account the weight sharing across the conditionals.

Again, we can use the ALL of this model as the objective function. It is given by Eq. (2.14). The parameter vector is now $\boldsymbol{\theta} = [\boldsymbol{b}, \boldsymbol{W}(*), \boldsymbol{c}, \boldsymbol{V}(*)] \in \mathbb{R}^N$. The total number of parameters is $N = D+H+H\times D+D\times H = D+H+2HD$. Each row of $\boldsymbol{W}$ will correspond to a new feature created by nonlinearly combining the initial features x_d, $d = 0, 1, 2, ..., D$.

In Figure 2.5 we can see illustrations of NADE and FVSBN. In Algorithm 2 we present the pseudocode for NADE(), a subroutine that computes the value of the ALL and its partial derivatives wrt the parameters $\boldsymbol{b}$, $\boldsymbol{W}$, $\boldsymbol{c}$ and $\boldsymbol{V}$ at a given point $(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V}) = (\boldsymbol{b^{(0)}}, \boldsymbol{W^{(0)}}, \boldsymbol{c^{(0)}}, \boldsymbol{V^{(0)}})$ in the parameter space, for the given

dataset $\boldsymbol{X_{train}}$ with labels $\boldsymbol{y}$ under the NADE model.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{b}$, $\boldsymbol{W}$, $\boldsymbol{c}$, $\boldsymbol{V}$
Result: $J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V})$, $\boldsymbol{\nabla J(b)}$, $\boldsymbol{\nabla J(W)}$, $\boldsymbol{\nabla J(c)}$, $\boldsymbol{\nabla J(V)}$
$[T', D] = SIZE(\boldsymbol{X_{train}})$;
for $d = 1 : T'$ **do**
  `// Objective function:`
  $\boldsymbol{\alpha} = \boldsymbol{c}$;
  $J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V}) = 0$;
  for $d = 1 : D$ **do**
    $\boldsymbol{h} = \sigma(\boldsymbol{\alpha})$;
    $p(x_d = 1|\boldsymbol{x_{<d}}) = \sigma(b_d + \boldsymbol{V(d, *)}\, \boldsymbol{h})$;
    $J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V}) = J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V}) + x_d \log(p(x_d = 1|\boldsymbol{x_{<d}})) + (1 - x_d) \log(1 - p(x_d = 1|\boldsymbol{x_{<d}}))$;
    $\boldsymbol{\alpha} = \boldsymbol{\alpha} + \boldsymbol{W(*, d)}\, x_d$;
  end
  `// Gradients:`
  $\boldsymbol{\nabla J(h)} = ZEROS(SIZE(\boldsymbol{c}))$;
  for $d = D : -1 : 1$ **do**
    $\boldsymbol{\alpha} = \boldsymbol{\alpha} - \boldsymbol{W(*, d)}\, x_d$;
    $\boldsymbol{h} = sigmoid(\boldsymbol{\alpha})$;
    $p(x_d = 1|\boldsymbol{x_{<d}}) = \sigma(\boldsymbol{b_d} + \boldsymbol{V(d, *)}\, \boldsymbol{h})$;
    $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_{b_d}} = p(x_d = 1|\boldsymbol{x_{<d}}) - x_d$;
    $(\boldsymbol{\nabla J(V)})(d, *) = (p(x_d = 1|\boldsymbol{x_{<d}}) - x_d)\, \boldsymbol{h}_T$;
    $\boldsymbol{\nabla J(h)} = (p(x_d = 1|\boldsymbol{x_{<d}}) - x_d)\, \boldsymbol{V(d, *)}_T$;
    $(\boldsymbol{\nabla J(W)})(*, d) = \boldsymbol{\nabla J(c)}\, x_d$;
    `// Below ∘ denotes element-wise multiplication:`
    $\boldsymbol{\nabla J(c)} = \boldsymbol{\nabla J(c)} + \boldsymbol{\nabla J(h)} \circ \boldsymbol{h} \circ (\boldsymbol{1} - \boldsymbol{h})$;
  end
  `// Return averages instead of sums:`
  $J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V})^{(t)} = \frac{1}{D} J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V})$;
  $\boldsymbol{\nabla J(b)}^{(t)} = \frac{1}{D} \boldsymbol{\nabla J(b)}$;
  $\boldsymbol{\nabla J(W)}^{(t)} = \frac{1}{D} \boldsymbol{\nabla J(W)}$;
  $\boldsymbol{\nabla J(c)}^{(t)} = \frac{1}{D} \boldsymbol{\nabla J(c)}$;
  $\boldsymbol{\nabla J(V)}^{(t)} = \frac{1}{D} \boldsymbol{\nabla J(V)}$;
end
`// Now average over all instances:`
$J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V}) = \frac{1}{T'} \sum_{t=1}^{(T')} J(\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{V})^{(t)}$;
$\boldsymbol{\nabla J(b)} = \frac{1}{T'} \sum_{t=1}^{(T')} \boldsymbol{\nabla J(b)}^{(t)}$;
$\boldsymbol{\nabla J(W)} = \frac{1}{T'} \sum_{t=1}^{(T')} \boldsymbol{\nabla J(W)}^{(t)}$;
$\boldsymbol{\nabla J(c)} = \frac{1}{T'} \sum_{t=1}^{(T')} \boldsymbol{\nabla J(c)}^{(t)}$;
$\boldsymbol{\nabla J(V)} = \frac{1}{T'} \sum_{t=1}^{(T')} \boldsymbol{\nabla J(V)}^{(t)}$;

Algorithm 2: The pseudocode for a simple, non-vectorized version of the algorithm that computes the value of the ALL of a NADE model and its partial derivatives wrt the parameters parameters $\boldsymbol{b}$, $\boldsymbol{W}$, $\boldsymbol{c}$ and $\boldsymbol{V}$: NADE().

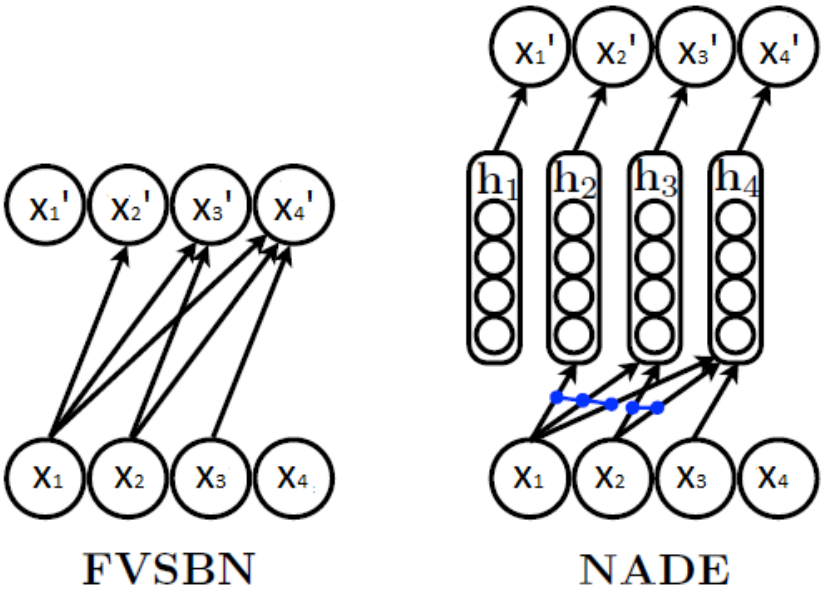

Figure 2.5: An illustration of an arbitrary FVSBN and an arbitrary NADE. The edges connected by blue lines denote connections with tied parameters. Also, we denote with $x'_d = p(x_d = 1|\boldsymbol{x}_{<\boldsymbol{d}})$ are the output probabilities of the density estimator.

Image adapted from [Larochelle et al., 2011].

Note that we can have $H > D$ or $H < D$. In the first case we map the initial feature vectors into a higher dimensional space, before mapping it back to a D-dimensional space. In the second case we map the initial feature vectors into a lower dimensional space, before mapping it back to a D-dimensional space, thus performing a non-linear dimensionality reduction step. The higher the redundancy in the dataset, the less lossy this transformation will be. The same idea extends to RBMs and DBMs in general, of course. For example [Hinton et al., 2006a], [Salakhutdinov et al., 2007b] show that using such a dimensionality reduction, DBMs can learn short binary codes that allow very fast retrieval of documents or images.

Chapter 3

Optimization Methods Examined

We experimented with various optimization methods and variants thereof during the course of this project. Some of these methods, for instance the simple versions of Steepest Gradient Descent and Coordinate Descent are not suited for large scale non-convex settings. However we included them in some experiments to showcase their inadequacy or to use them as a baseline to improve upon and as a means to "measure the difficulty" of training on a particular dataset.

In this section we will present the methods used, the theoretical motivation behind them and a brief discussion about the strengths, weaknesses and some potential variations of each one of them. A brief description of each method is followed by the corresponding pseudocode in its most basic version. Details such as step size decay, termination criteria and tricks to reduce the number of operations (such as indexing and vectoring) were omitted wherever possible in order to keep the pseudocode for each method in its most general and intelligible form. Step size decay and termination criteria (including early stopping) are shared across groups of methods and will be discussed in the following chapter in more detail.

3.1 Categorization of Optimization Methods

Mathematical Optimization is a huge field which encompasses many types of problems and the techniques used for solving them. Categorizations of these techniques include factors such as the number of the objective functions (single-objective vs multi-objective), the existence or not of constraints (constrained vs unconstrained) the properties of the constraints and the objective functions (linear vs non-linear, including special subcategories such as geometric and quadratic programming problems), the separability or lack thereof of the objective functions and the constraints (separable vs non-separable) the properties of the parameters (integer vs real-valued programming, deterministic vs stochastic) the existence or lack thereof of interdependent stages (optimal control vs non-optimal control problems). Another division of optimization techniques can take into account whether a technique is guaranteed to find the optimum or not (exact vs heuristic

optimization techniques). We shall not expand any further on the landscape of optimization techniques. The subject of this dissertation is not to cover optimization methods in general, but merely to apply a number of appropriate ones to the problem in hand: training the NADE model to perform fast and accurate density estimation on a given dataset.

Therefore, we will only deal with optimization methods that are used for solving unconstrained, non-linear, real-valued optimization problems. Both objective functions we include in our study (logistic regression and NADE's average negative log-likelihood) are non-linear, our parameters take real values and there are no constraints involved. In Table 3.1 we can see a simple categorization of the methods studied based on the order of derivative information of the objective function with respect to the parameters they use and whether they need to examine the entire dataset before making a parameter update (Batch Methods), they make an update after seeing a single instance (Stochastic Methods) or a number of instances (Mini-batch Methods).

Table 3.1: A categorization of the methods studied.

	Instances Examined per Update Step		
Order	Batch	Minibatch	Stochastic
Zeroth	CD_basicGD CD_basicNewton		
First	BGD_basic, BGD_heavyball, BGD_bolddriver, BGD_rprop, DIBGD_basic,	MBGD_basic, MBGD_heavyball, MBGD_bolddriver,	SGD_basic, SGD_heavyball, ASGD_basic, IAGD_basic,
Second Quasi-Newton	Newton L-BFGS		

3.2 Order of Optimization Methods

Zeroth order methods do not directly calculate the gradient of the objective function (see next paragraph for more on the gradient). For this reason they are also referred to as non-gradient or derivative free optimization techniques. They perform a line search in a single dimension of the parameter space (can do so iteratively as well), or approximate the gradient using other techniques. Since they need only use simple evaluations of the objective function, each iteration is less expensive computationally compared to first order (or higher) methods. However they are less efficient than the first order methods, since they require more steps to converge. Most of them are easy to program and robust and an important benefit is that they can also be applied to objective functions that are discontinuous or non-differentiable, contrary to higher order methods. For the definitions below we shall assume that the objective function is continuous and k times differentiable, where k is the order of the method we are referring to.

First order methods use only gradient and objective value function information, in other words they only consider up to the first order partial derivatives of the

objective function with respect to the parameters in order to decide how the corresponding parameters will be updated. The gradient $\boldsymbol{\nabla J}$ of the objective function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta} = [\theta_i], i = 1, 2, ...N$ is the vector of all first order partial derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$,

$$\boldsymbol{\nabla J} = \boldsymbol{\nabla J(\theta)} = \left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}\right], i = 1, 2, ...N. \tag{3.1}$$

In second order methods, the parameter updates also use curvature (second order derivative) information. The Hessian $\boldsymbol{H(J(\theta))}$ (or simply $\boldsymbol{H(J)}$ or even $\boldsymbol{H}$ when it is obvious from the context) of the objective function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta} = [\theta_i], i = 1, 2, ...N$ is the matrix whose (i, j)-th element is the second partial derivative of $J(\boldsymbol{\theta})$ with respect to parameters θ_i and θ_j,

$$\boldsymbol{H} = \boldsymbol{H(J(\theta))} = \left[\frac{\partial^2 J(\boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}\right], i, j = 1, 2, ...N. \tag{3.2}$$

The non-diagonal elements of $\boldsymbol{H}$ are called mixed derivatives (generally partial derivatives of second or greater order with respect to two or more different variables are called so). The order of differentiation does not matter for continuous mixed derivatives. In other words, $H_{ij} = H_{ji}, \forall i, j$ and therefore the Hessian is symmetric. Due to this symmetry, instead of N^2 second order partial derivatives we only need to calculate $N(N-1)/2$ to compute the Hessian of $J(\boldsymbol{\theta})$ (remember that N is the dimensionality of the parameter space). So we need $O(N^2)$ operations. Moreover, some 2nd order methods require the inverse of the Hessian $\boldsymbol{H^{-1}}$, the computation of which requires $O(N^3)$ operations. So, second order methods are characterized by a high computational cost.

Some methods have been developed to incorporate curvature information without explicitly using the Hessian itself, but approximations of it or its inverse are computed with the use of first order derivative and previous step information. These methods, called Quasi-Newton Methods, aim to balance the nice convergence properties of the second order methods without suffering the cost of the latter. They are still more expensive computationally per iteration than first order methods though.

There also exist Higher Order methods, which also take into account the third or higher order partial derivatives of the objective function with respect to the parameters in order to decide how the corresponding parameters will be updated. These are rarely used in practice as they involve tensors of higher order partial derivatives, rendering the computational cost per iteration unaffordable. We will not discuss any such methods here.

So the central idea here is that higher order methods are typically more powerful than first order methods, in the sense that they converge in fewer iterations. However, they are more expensive computationally (per iteration) and possibly their computational cost can outweigh their benefits, especially for large parameter

spaces. So we have a convergence rate versus computational (time) complexity tradeoff.

3.3 Batch vs Stochastic Optimization Methods

Batch methods use the entire training set in order to calculate the appropriate update for each of the parameters. As a result, each step is slower compared to that of stochastic and minibatch methods. On the other hand, since it takes into account all the training instances, each step is indeed towards the direction that minimizes the overall error on the training set. The stochastic methods make a parameter update based on a single instance t, therefore it will not necessarily lead to a parameter vector $\boldsymbol{\theta}$ which decreases the average or total value of the objective function (i.e. over all training instances), but for the particular instance.

Comparisons of the relative advantages of batch and stochastic methods can be found in [LeCun et al. , 1998], [Bishop, 1995], [Bengio, 2012] and [Bottou et al. , 2007]. Here we will review the main ideas presented in the aforementioned works. In short, the batch methods have the following advantages:

(i) Their conditions of convergence are well understood.

(ii) Many acceleration techniques like conjugate gradient or 2nd order methods only operate in batch learning (This is not entirely true, there exist second order stochastic methods, however they are not used widely and we will explain later on why).

(iii) Analyzing theoretically their parameter dynamics and convergence rates is simpler.

(iv) They can be parallelized (for example by using map-reduce techniques).

The above advantages are lost in stochastic gradient descent methods, due to the noise their stochasticity adds and to the fact that execution of each step requires the previous step to have been executed. However, the stochastic methods also have their advantages over the batch methods:

(i) They are much faster than batch methods in large-scale learning and in scenarios with redundant data.

(ii) They usually result in better solutions in non-convex objectives.

(iii) They can be used for keeping track of changes to the function modeled.

(iv) They can be used in the context of online learning (Somewhat related to (iii)).

(v) Although the convergence rate of stochastic methods is lower than that of batch methods, there is evidence [Bottou et al. , 2007], [Bengio, 2012] that in the context of learning tasks this convergence benefit loses its importance.

For a simple explanation for (i) let us imagine that our dataset consists of 10 consecutive identical copies of a smaller dataset of, 100 instances. Averaging the gradient over all 1000 instances would yield the same result as computing the gradient based on just the first 100. So, a batch method would need 10 times more time to make a single update to the parameters compared to a stochastic method. This example is —of course— extreme, as identical examples rarely appear in real-world datasets. However similar ones often do. This redundancy is what can make stochastic methods faster than their batch counterparts.

As for (ii), due to the stochasticity of the updates a stochastic method can escape local optima. Advantage (iii) comes into play when the function we model changes over time. Then a stochastic method can capture these changes, as it examines every instance in sequence. Somewhat related to (iii), advantage (iv) comes into play when the instances are presented one at a time (or few at a time) to the training algorithm and we generally need to update our parameters as soon as the new example(s) arrive(s). Here, using batch methods is no longer a viable option as it would be a terrible waste computationally to use our entire old dataset of T instances plus our 1 new example and perform operations on the new $(T+1)$-sized batch. It is also a waste of memory to keep all previous instances stored. Using a stochastic method we can avoid all this computational and memory cost. So this is an extra advantage of the stochastic methods in large-scale optimization.

Finally, regarding (v), the convergence rate of e.g. Stochastic Gradient Descent might be lower than that of Batch Gradient Descent. However, in learning tasks, it has been shown that the convergence rate of the test error (i.e. the value of the objective function on unseen data) is $O(\frac{1}{t})$, i.e. the same as the rate of convergence of SGD [Bottou et al. , 2007], [Bengio, 2012]. So there is no benefit in having a faster convergence rate on the training set as this will not mean having a faster convergence rate on the test set. And remember, in learning we don't really care about minimizing the value of the objective function on the training set, but about achieving generalization.

Miniibatch methods try to strike a balance between the two approaches. If very small batch sizes are used they lean towards the stochastic side, which allows them to be used in online learning and grants them increased ability to escape local optima in non-convex objectives (contrary to batch methods). In addition to that, vectorization can allow each iteration to be almost as fast as a stochastic gradient descent step and compared to stochastic methods, minibatch methods generally need fewer update steps as its steps are less noisy (i.e. each step is based on more training samples, so the direction of the step is towards the local minimum with greater probability). If large batch sizes are used they become less stochastic, but still can have benefits over batch methods e.g provide faster updates.

So, to summarize, when optimizing non-convex objectives we favour stochastic or minibatch or second order batch ones. In large datasets, high redundancy within

the data is to be expected so we also favour stochastic methods and minibatch ones with small batch sizes. We also opt to use stochastic methods when we want each update step to be taken as quickly as possible (e.g. right after receiving the next instance). If the parameter space or the dataset is small, perhaps it is a better idea to use second order methods instead.

3.4 Some Intuitions Regarding Our Optimization Problem

So we should take into account many factors when deciding which optimization method to use and how to tailor it to the problem, including the properties of the objective function (e.g. convexity, smoothness, steepness), the dimensionality of the parameter space, the size of the dataset and whether we need to make updates as fast as possible or we can afford to wait. Also, any additional domain knowledge we have, for example, whether a high amount of redundancy in the dataset is to be expected or not, whether some features are correlated and the parameters related to them can be treated as a groups or not, all these can determine which method is appropriate for a given problem.

In our case, the objective function we are trying to optimize is non-convex. So we can deduce that stochastic, minibatch and second order batch methods are the most viable options for achieving good quality optima. Furthermore, we have relatively large datasets, so redundancies are likely to exist within them. Redundancy gives a speed advantage to stochastic methods and minibatch methods where the size of the minibatch is small. We also have large parameter spaces to explore, a fact that translates to high computational cost of second order methods.

So intuitively, in the task of training NADE on the specific datasets, we expect variants of stochastic and minibatch methods to yield better, faster results than the other methods and while second order methods might match (or possibly exceed) their quality of optima, they will require larger execution times.

3.5 Gradient Computation

In Algorithm 3 we can see the common function all methods we examine share: the one that computes the value of the objective function and its gradient with respect to the parameters $\boldsymbol{\theta}$ at $\boldsymbol{\theta^{(0)}}$ on the given training set $\boldsymbol{X_{train}}$. We imaginatively named it COMPUTE_OBJECTIVE_AND_GRADIENT(). To clarify any confusion $\boldsymbol{\theta^{(0)}} \in \mathbb{R}^N$ is just an instantiation of the variable $\boldsymbol{\theta}$. From this point on, however we shall use the two symbols interchangeably since we can easily distinguish if we refer to a parameter or its actual value by the context. Also, in the algorithms that follow we will use J instead of $J(\boldsymbol{\theta^{(0)}})$ and $\boldsymbol{\nabla J}$ instead of $\boldsymbol{\nabla J(\theta^{(0)})}$. Again, what we mean will be clear from the context. Of course

for every model used the objective function and the parameters involved differ, so COMPUTE_OBJECTIVE_AND_GRADIENT() will take the form of NADE() or LOGREG() we presented in the previous section, here we just generalize the concept to include any of these, as well as any other model.

As for the given training data matrix $\boldsymbol{X_{train}}$ its dimensions change according to the actual method use. In the case of batch methods, $\boldsymbol{X_{train}} \in \mathbb{R}^{T \times D}$, as we use all training instances. So N is the number of parameters of our model, T the number of instances (training datapoints) and D the number of features (the dimensionality of each datapoint). For Stochastic Gradient Descent, $\boldsymbol{X_{train}} \in \mathbb{R}^{1 \times D}$, since now we only use one instance to make an update. Finally, for the Dynamically Increasing Batch Size approaches and the Mini-Batch Methods, $\boldsymbol{X_{train}} \in \mathbb{R}^{T_{batch} \times D}$, where T_{batch} is the current minibatch size.

In all cases except for Coordinate Descent, the function outputs $J(\boldsymbol{\theta^{(0)}}) \in \mathbb{R}$ and $\boldsymbol{\nabla J(\theta^{(0)})} \in \mathbb{R}^{\mathbb{N}}$. In Coordinate Descent, we still get a scalar $J(\boldsymbol{\theta^{(0)}})$ but instead of $\boldsymbol{\nabla J(\theta^{(0)})} \in \mathbb{R}^{\mathbb{N}}$ we get $\frac{\partial J(\boldsymbol{\theta^{(0)}})}{\partial \theta_i} \in \mathbb{R}$, since we are working with a single parameter at each iteration. We use $J^{(t)}(\boldsymbol{\theta^{(0)}}) \in \mathbb{R}$ to denote the cost on the t-th instance of the training set and $\boldsymbol{\nabla J^{(t)}(\theta^{(0)})} \in \mathbb{R}^{\mathbb{N}}$ to denote the gradient computed using only the t-th instance of the training set. All this will become more clear in the description of each method.

All methods we implemented call such a function on every iteration. The method we call from the Mark Schmidt's minfunc() package[1] (L-BFGS) also calls such a function on every iteration. Second order methods in addition to the gradient also need the Hessian supplied, so the algorithm below needs to also include the computation of the Hessian in an analogous fashion (i.e. averaging over all instances). Higher order methods would need tensors of higher order partial derivatives.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta^{(0)}}$
Result: $J(\boldsymbol{\theta^{(0)}}), \boldsymbol{\nabla J(\theta^{(0)})}$
$T' = SIZE(\boldsymbol{X_{train}}, ROWS)$;
// Below, OBJFUNCEVAL($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$) and
COMPUTEGRAD($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$) simply compute $J^{(t)}(\boldsymbol{\theta^{(0)}})$ and
$\boldsymbol{\nabla J^{(t)}(\theta^{(0)})}$ respectively and their implementations differ from
model to model:
$J(\boldsymbol{\theta^{(0)}}) = \frac{1}{T'} \sum_{t=1}^{T'}$ OBJFUNCEVAL($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$);
$\boldsymbol{\nabla J(\theta^{(0)})} = \frac{1}{T'} \sum_{t=1}^{T'}$COMPUTEGRAD($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$);

Algorithm 3: The general pseudocode for the algorithm that computes the value of the objective function and its gradient with respect to the parameters: COMPUTE_OBJECTIVE_AND_GRADIENT().

We should note here that instead of taking the average of the $J^{(t)}(\boldsymbol{\theta^{(0)}})$ we could instead simply take their sum since $\frac{1}{T'}$ is just a scaling factor. We would correct

[1] Available at `http://www.di.ens.fr/~mschmidt/Software/minFunc.html`.

the scaling of the actual parameter update with appropriate selection of the step size α to rescale the update step. Similarily, it bears no significance if we optimize regarding to the total cost $\sum_{t=1}^{T'} J^{(t)}(\boldsymbol{\theta^{(0)}})$ or the average cost $\frac{1}{T'}\sum_{t=1}^{T'} J^{(t)}(\boldsymbol{\theta^{(0)}})$ per instance.

Also, although this implementation would work for all kinds of methods, we point out for one last time that this is a general purpose pseudocode, not the actual implementation we used. In fact, we implemented two different versions for this function: one for one for Stochastic Gradient Descent and its variants and one for all the other methods. The benefit to this is that X_{train} degenerates to a D-dimensional row vector in the SGD case, so we can ignore one dimension and avoid additional operations (e.g. avoid taking averages for SGD) gaining some computational benefits (albeit constant ones). We will not dwell any more on this, we just mention it to showcase that a general model can be good for abstraction and theoretical generalization, but in practice, all theoretical tools have to be tailored to their practical implementation.

3.6 Batch Methods

Batch Gradient Descent

The simplest method that falls in the batch category is Batch Gradient Descent, also known as (Steepest) Gradient Descent. It is a First Order Method that is based on taking steps proportional to the negative of the gradient ($-\alpha\,\boldsymbol{\nabla J}$) in order to find a local minimum of $J(\boldsymbol{\theta})$. Gradient descent is based on the observation that $J(\boldsymbol{\theta})$ decreases fastest if we go from $\boldsymbol{\theta^{(0)}}$ in the direction of $-\boldsymbol{\nabla J(\theta^{(0)})}$, so if $\boldsymbol{\theta^{(1)}} = \boldsymbol{\theta^{(0)}} - \alpha\,\boldsymbol{\nabla J(\theta^{(0)})}$, for a small enough α, then $J(\boldsymbol{\theta^{(1)}}) \leq J(\boldsymbol{\theta^{(0)}})$. So for $\boldsymbol{\theta^{(n)}} = \boldsymbol{\theta^{(n-1)}} - \alpha(n)\,\boldsymbol{\nabla J(\theta^{(n-1)})}, n \geq 0$ we will have $J(\boldsymbol{\theta^{(0)}}) \geq J(\boldsymbol{\theta^{(1)}}) \geq ... \geq J(\boldsymbol{\theta^{(n)}})$. Consequently, the sequence $\boldsymbol{\{\theta^{(n)}\}}$ converges to the local minimum. We can see an illustration of the process in Figure 3.1.

As for $\alpha(n) \in \mathbb{R}^+$ is called the *step size*, sometimes also referred to as the *learning rate*. It controls, as the name suggests, the size of the update step of the parameters. It can be constant or vary as a function of the number of iterations passed. The step size is a hyperparameter of the optimization method. More on hyperparameters and hyperparameter selection can be found in the next section. If the goal of the optimization problem is to maximize, rather than minimize the objective function all that changes is that we move towards the direction of the gradient and not its negative, '$-$' signs in the formulae and the pseudocodes below become '+'. To distinguish between the two procedures we call this *Gradient Ascent*. In fact in our implementation we actually perform Gradient Ascent on the Average Log Likelihood of the NADE model instead of Gradient Descent on the Negative Average Log Likelihood.

The variants we will examine here are the following:

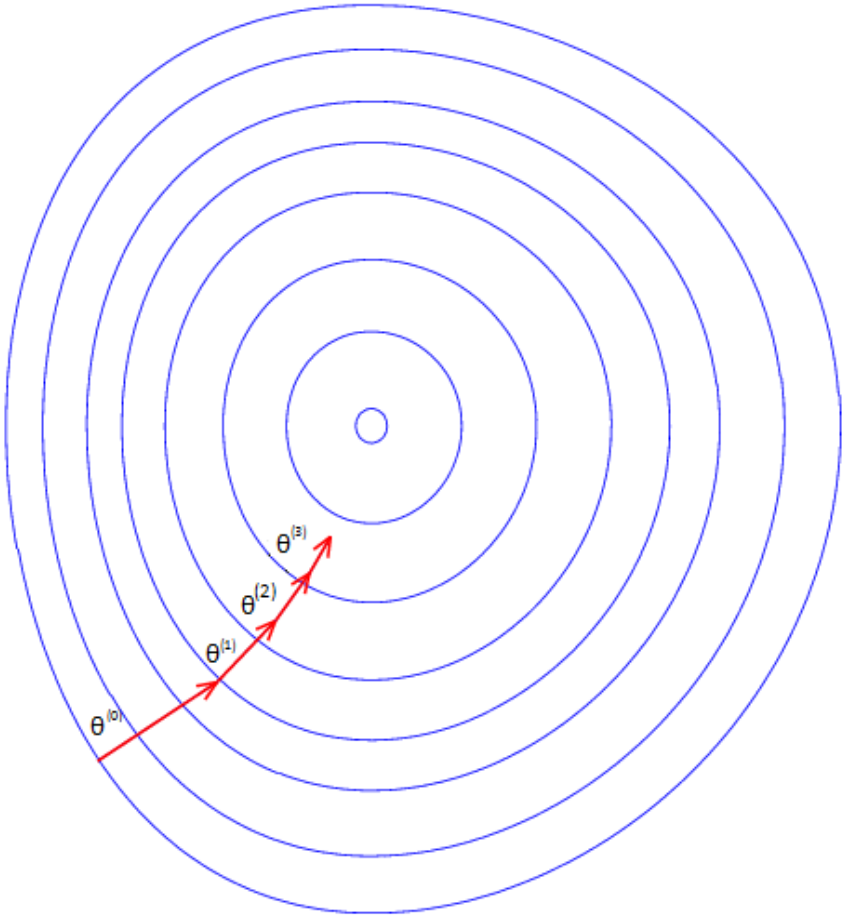

Figure 3.1: An illustration of the convergence of Steepest Gradient Descent to the local minimum. In this case it is the global minimum as the objective is convex. Image adapted from Wikipedia:
`http://en.wikipedia.org/wiki/File:Gradient_descent.png`

BGD_basic: The simple version of the method we discussed above. The pseudocode can be found in Algorithm 4.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}$, α
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}$);
  $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla J}$;
end

Algorithm 4: The core pseudocode for the simple version of Steepest Gradient Descent: BGD_basic.

BGD_heavyball: Batch Gradient Descent using the *Heavy Ball* heuristic introduced in [Polyak, 1964]. The idea is borrowed from the movement of a ball

subject to the laws of classical mechanics. The simple Gradient Descent version can be very slow when the surface of the objective function is highly non-spherical (its curvature varies significantly with each direction). At most points on the error surface, the local gradient will not point directly towards the minimum. The result is a "zig-zagging" behaviour, so Gradient Descent will take many steps to reach the minimum. We can moderate this effect by to adding a contribution from the previous step $\boldsymbol{\Delta\theta_{prev}}$ weighted by a "momentum" term $0 \leq \gamma \leq 1$ to the parameter update , which will smooth the oscillations (the "zig-zagging" mentioned above), and therefore allow us to reach the minimum in fewer steps. However it adds an extra hyperparameter, the momentum term, which we we have to fine tune in addition to α. In Figure 3.2 we can see the zig-zagging behaviour of Steepest Gradient Descent compared to the more smooth convergence of the 'Heavy Ball' heuristic. The pseudocode for this version can be found in Algorithm 5.

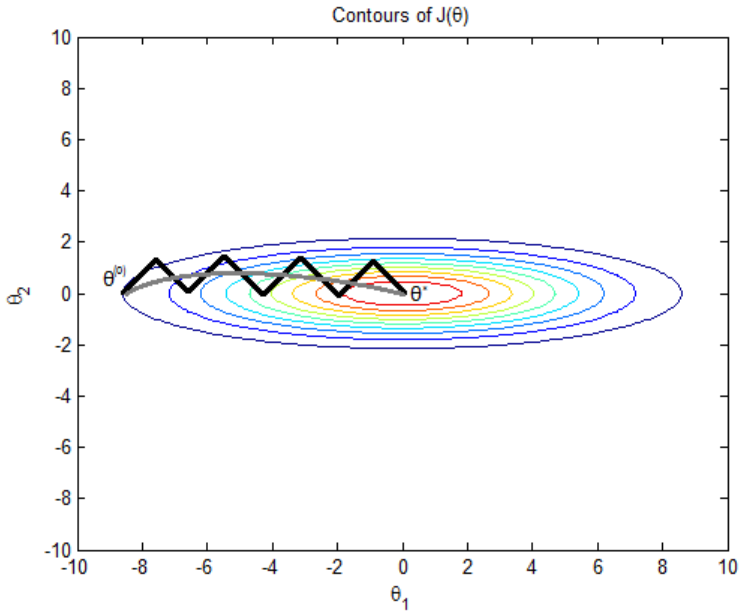

Figure 3.2: An illustration of the "zig-zagging" behaviour the Steepest Gradient Descent might suffer from in 2 or more dimensions. In black we can see the steps of Steepest Gradient Descent towards the direction of the negative gradient and with gray how by using the "Heavy Ball" heuristic the successive steps can almost "cancel out" the movement on the θ_2 coordinate retaining only the movement the on θ_1, thus reaching in fewer iterations the minimum. The "zig-zagging" behaviour was caused due to the fact that we only needed to move along θ_1 to reach the minimum, but in Gradient Descent we move towards the direction of the negative gradient, which does not necessarily point to the minimum.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \gamma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $\boldsymbol{\Delta\theta_{prev}} = \mathbf{0}$;
while *Termination Conditions Are Not Met* **do**
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\,\boldsymbol{\nabla J} - \gamma\boldsymbol{\Delta}\,\boldsymbol{\theta_{prev}}$;
    $\boldsymbol{\Delta\theta_{prev}} = \alpha\,\boldsymbol{\nabla J}$;
end

Algorithm 5: The core pseudocode for the "Heavy Ball" version of Steepest Gradient Descent: BGD_heavyball.

BGD_bolddriver: Batch Gradient Descent using the *Bold Driver* heuristic [Vogl et al., 1988] to adapt the step size after each iteration. The idea is simple: As long as we are far away from the local optimum we must take big steps in order to approach it. Once we are close to it, we need to take smaller steps. But how do we know we are close to the local minimum? A simple way to do this — and the basis of the Bold Driver technique— is that we are close to a local minimum when... we have just overshot it! How do we detect that we overshot the minimum? Simply, the value of the objective function has increased since the last iteration. So in this case, the learning rate proved to be too large to allow us to converge to the local minimum and therefore we decrease it heavily to allow for convergence in the next iterations. On the other hand, we can afford to slightly increase the learning rate in each iteration that we keep detecting a decrease in the value of the objective function based on the idea that "if we haven't overshot it, we are still far from it and need to move faster towards it". Increasing the step size in this case will allow us to get faster near the minimum, and if we overshoot then decrease it considerably to eventually reach it. The fancy name of the algorithm comes from its resemblance to the behaviour of a reckless driver that keeps going faster and faster until there is sufficient indication that this is dangerous, in which case they decrease the speed of the vehicle considerably and instantly. It is a simple and effective strategy to adjust the weights, but perhaps a bad strategy for driving. In Figure 3.3 we can see a visualization of the method. One way to implement it is to compute at each iteration i the value of the objective function on the training set $J^{(i)}$. If in iteration i the value of the objective function has decreased compared to $J^{(i-1)}$, we increase the stepsize slightly, multiplying it by a factor $\rho > 1$, typically around 1.1. If, on the other hand, the value of the objective function has increased compared to $J^{(i-1)}$, we decrease the stepsize severely, multiplying it by a factor $0 < \sigma < 1$, typically around 0.5. We could also use other schemes (e.g. a linear one) to adapt the step size but our implementation used this approach. The constants ρ and σ are additional hyperparameters. In the next section we can see what values we actually used and how we selected them (the above given are just a "rule of thumb" and we gave them just to point out roughly how big the decrease and the increase in weights should be). The pseudocode for this approach can be found in Algorithm 6.

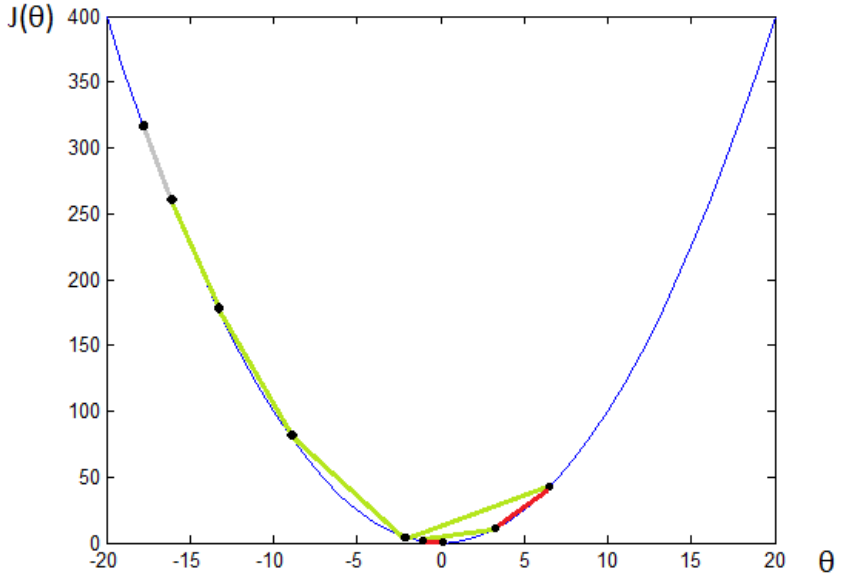

Figure 3.3: An illustration of the Bold Driver Method. In this case the learning rate is decreased considerably, hopefully allowing us to eventually reach the minimum. Steps shown in red indicate that the step size has been decreased, while green ones indicate that it has been increased. The first step is taken using the initial step size alpha.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \rho, \sigma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $J_{prev} = \infty$;
while *Termination Conditions Are Not Met* **do**
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla J}$;
    if $J < J_{prev}$ **then**
        $\alpha = \alpha\, \rho$;
    else if $J > J_{prev}$ **then**
        $\alpha = \alpha\, \sigma$;
    else
        `// Do Nothing...`
    end
    $J_{prev} = J$;
end

Algorithm 6: The core pseudocode for the "Bold Driver" version of Steepest Gradient Descent: BGD_bolddriver.

BGD_rprop: This method is known as '*Resilient Backpropagation* or *R-prop* [Riedmiller et al., 1993]. Here we use a different step size for every parameter. Furthermore, we adapt each step size individually. The idea is that if the sign of a partial derivative has changed from last iteration, we have actually over-

shot the minimum in this dimension, as shown in Figure 3.4, so in this case we should decrease the step size in order to converge to it in next iterations. On the other hand if the sign of a partial derivative remains unchanged in consecutive iterations, it means we have yet to overshoot the minimum in this dimension, so increase the step size. In terms of implementation, we just keep track of (the sign of) each partial derivative we calculated in the previous iteration, and if the sign is the same in the current one, we multiply the step size of this parameter by a factor $\rho > 1$. If the sign is different in the current one, we multiply the step size of this parameter by a factor $0 < \sigma < 1$. Typical values are for these constants are $\rho = 1.1$ and $\sigma = 0.5$. These are additional hyperparameters of the method and in the next section we can see what values we actually used and how we selected them (the above given are again just a "rule of thumb" roughly showing how big the decrease and the increase in weights should be). The pseudocode for this version can be found in Algorithm 7.

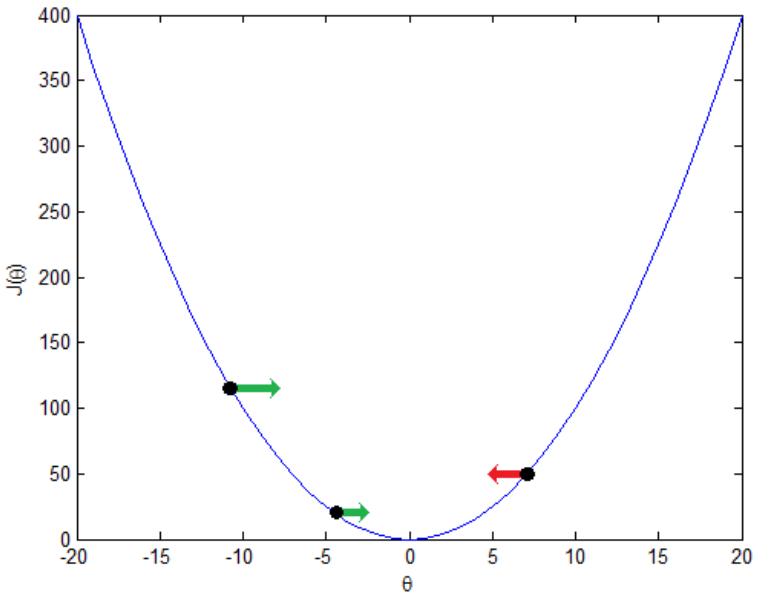

Figure 3.4: An illustration of the basic idea behind the "Resilient Backpropagation" heuristic. The color of the arrows indicates the sign of the partial derivative of the objective function wrt the parameter at the specific point. Green stands for positive, red stands for negative. Once the sign changes we have overshot the minimum.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \rho, \sigma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $\boldsymbol{\nabla J_{prev}} = \mathbf{0}$;
/* Now expand α into a vector of individual parameter stepsizes.
Initially all parameters have the same stepsize: */
$\boldsymbol{\alpha_{vec}} = \alpha \, \boldsymbol{zeros(SIZE(\theta))}$;
while *Termination Conditions Are Not Met* **do**
  $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}$);
  // In the line below $\circ$ denotes element-wise multiplication:
  $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\alpha} \circ \boldsymbol{\nabla J}$;
  for $i = 1 : N$ **do**
    if $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} * (\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i})_{prev} > 0$ **then**
      $\alpha_i = \alpha_i \, \rho$;
    else if $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} * (\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i})_{prev} < 0$ **then**
      $\alpha_i = \alpha_i \, \sigma$;
    else
      // Do Nothing...
    end
  end
  $\boldsymbol{\nabla J_{prev}} = \boldsymbol{\nabla J}$;
end

Algorithm 7: The core pseudocode for the "Resilient Backpropagation" variant of Steepest Gradient Descent: BGD_rprop.

Generally, Batch Gradient Descent is outperformed by other methods. It is a first order method so it needs more iterations to converge compared to second and quasi-Newton methods and it is a batch method which renders it useless in online settings and leaves it at a disadvantage in the task of optimizing non-convex functions, compared to stochastic and mini-batch methods. Even for convex objective functions there are more successful methods one can use such as Conjugate Gradient [Hestenes et al., 1952]. Perhaps only for very small datasets and convex objectives should one pick Batch Gradient Descent.

The Heavy Ball heuristic can counter the zig-zagging behaviour, increasing the speed of convergence of the Batch Gradient Descent in most cases. However this method is still plagued by the problems we mentioned in the previous paragraph. The Bold Driver and Resilient Backpropagation (R-prop) heuristics in practice offer adaptive learning rates providing both speed and quality of solution benefits and appear to work acceptably well even in non-convex settings. In fact, R-prop is still widely used in the fields of Natural Language Processing [Huck et al., 2010], [Hahn et al., 2011] and Speech Recognition [Zweig et al., 2010], [Heigold et al., 2009], mainly due to its simplicity to implement.

Coordinate Descent

Coordinate Descent is technically a zeroth order method as it does not use the gradient vector (although it can be implemented to use partial derivative at each iteration, we do not compute the entire gradient or Hessian). Coordinate Descent considers all but one parameters as constants in each iteration i and optimizes the objective w.r.t the remaining one. So at each iteration $J(\boldsymbol{\theta})$ degenerates to $J(\theta_i)$. We optimize in one direction at each iteration by performing a line search. This approach can be used in stochastic, mini-batch learning, as well as batch learning. Here we only considered the batch variant. In Figure 3.5 we can observe an illustration of the method.

A way to implement the line search is similarly to Steepest Gradient Descent but instead of the entire gradient $\boldsymbol{\nabla J}$ at each iteration i we compute the first partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}, i = 1, 2, ...N$. After executing N iterations cycling through all N parameters we have completed an epoch of the algorithm. We run as many epochs as needed until the termination criteria are met. Line search by gradient descent has the disadvantage that it needs us to set a step size and preferably a different one for each parameter.

Another way to find the minimum in a single dimension is by use of Newton's Method (more on the next subsection). Newton's method is a second order method and in higher dimensions it is computationally expensive, but on just 1 dimension it degenerates into what we call an "inexpensive Newton" as $\boldsymbol{H} = \left[\frac{\partial^2 J(\theta_1)}{\partial \theta_1^2}\right]$, i.e. a scalar, therefore storing it in memory and inverting it is "inexpensive".

We can also perform the line search by direct search methods. We iteratively divide the search space into intervals and then reject the interval that does not contain the minimum. Here we shall not examine such a version but we mention it for completeness.

Due to the coordinate steps the algorithm takes it can become very slow when many parameters are involved. Especially if many of them are not that important in the optimization procedure, we end up wasting time optimizing along them. One of its biggest problems is that it can get stuck at a non-stationary point if the objective function is non-smooth, as shown in Figure 3.6. One solution to this problem could be to use the R-prop technique mentioned earlier (applicable in a batch learning context only) to adjust the individual step sizes.

Coordinate Descent can also take other forms, for instance, instead of cycling through the parameters each epoch we could simply pick a parameter at random to optimize at each iteration. An interesting variant is the Randomized (Block) Coordinate Descent Method [Nesterov, 2010], which extends the concept of updating only one parameter at a time to updating blocks of parameters. Updating the parameters in blocks can increase the speed of convergence by countering the effects of the "coordinate movement" mentioned earlier. An important extension of this method would be finding suitable groups of parameters to update at

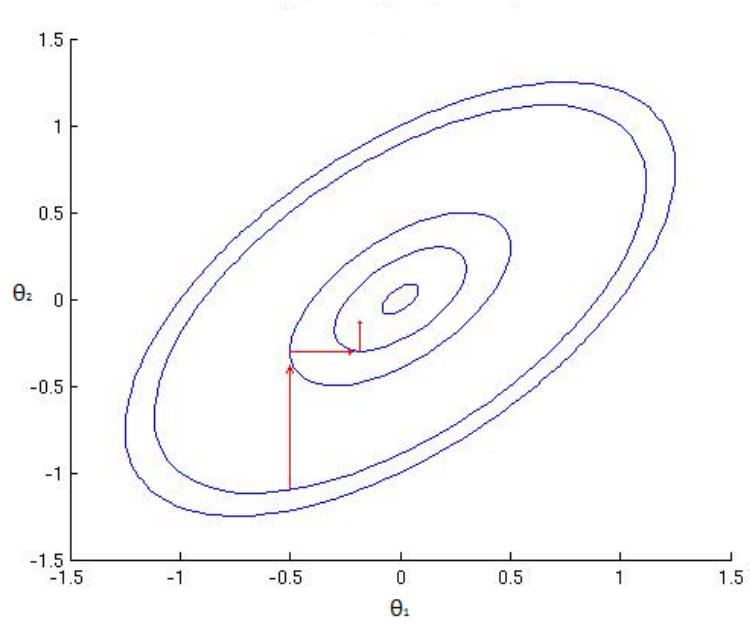

Figure 3.5: Three iterations of the Coordinate Descent method in a 2-dimensional parameter space. On each iteration we move along one dimension of the parameter space.

Image adapted from Wikipedia:
`http://en.wikipedia.org/wiki/File:Coordinate_descent.jpg`

the same step, perhaps by examining similarities over time in the corresponding partial derivatives. We might also consider heuristic approaches that allow us to skip updating certain parameters, if for example we notice small change in them during recent epochs. All these heuristics can help Coordinate Descent become competitive to the other methods presented here.

Here we only examined the following basic versions of the method:

CD_basicGD: The simple batch version of the method that cycles through the parameters and updates them individually by performing gradient descent. The pseudocode can be found in Algorithm 8.

CD_basicNewton: The simple batch version of the method that cycles through the parameters and updates them individually using an inexpensive Newton step (a detailed description of Newton's Method will follow shortly). The pseudocode

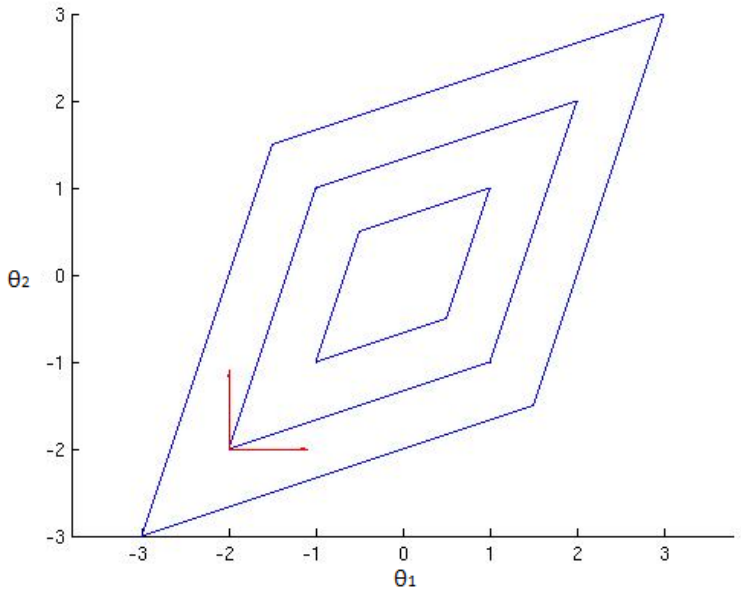

Figure 3.6: An illustration on Coordinate Descent on a non-smooth function. Here Coordinate Descent got trapped at a non-stationary point.
Image adapted from
Wikipedia:`http://en.wikipedia.org/wiki/File:Nonsmooth.jpg`

can be found in Algorithm 9.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  for $i = 1 : N$ **do**
    `// Below, COMPUTE_OBJECTIVE_AND_GRADIENTS(`$\boldsymbol{X_{train}}, \boldsymbol{i}$`) returns only the i-th element of the gradient:`
    $\left[J, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}\right]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}, i$);
    $\theta_i = \theta_i - \alpha \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}$;
  end
end

Algorithm 8: The core pseudocode for the simple batch version of Coordinate Descent where the line search was performed using gradient descent: CD_basicGD.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  for $i = 1 : N$ **do**
    // Below, COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}, i$) returns only the i-th element of the gradient and $COMPUTE\_2ND\_ORDER\_PART\_DER(\boldsymbol{X_{train}}, i)$ computes the second partial derivative of $J(\boldsymbol{\theta})$ wrt θ_i:
    $\left[J, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}\right]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}, \boldsymbol{i}$);
    $\frac{\partial J^2(\boldsymbol{\theta})}{\partial {\theta_i}^2} = COMPUTE\_2ND\_ORDER\_PART\_DER(\boldsymbol{X_{train}}, i)$;
    $\theta_i = \theta_i - \left(\frac{\partial J^2(\boldsymbol{\theta})}{\partial {\theta_i}^2}\right)^{-1} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}$;
  end
end

Algorithm 9: The core pseudocode for the simple batch version of Coordinate Descent where the line search was performed using an inexpensive Newton's step: CD_basicNewton.

Newton's Method

Newton's method (see [Murray, 2010] for a detailed review) is a second order optimization method. Figure 3.7 shows how by utilizing curvature information Newton's method converges faster than gradient descent. We only implemented Newton's method for the logistic regression toy problem, just to showcase it. We also used Newton's method in CD_basicNewton, but this was a trivial case. The reason we did not use Newton's method for training NADE is that in NADE's case the parameter space is too large, therefore computing the objective function's Hessian wrt the parameters would be computationally expensive.

At each iteration i, Newton's method approximates $J(\boldsymbol{\theta})$ by a quadratic function around $\boldsymbol{\theta^{(i-1)}}$, and then takes a step towards the maximum of the quadratic approximation. If $J(\boldsymbol{\theta})$ happens to be a quadratic function, then the exact minimum is found in one step. For each step $i > 1$ we have

$$\boldsymbol{\theta^{(i)}} = \boldsymbol{\theta^{(i-1)}} - H(J(\boldsymbol{\theta^{(i-1)}}))^{-1} \boldsymbol{\nabla J(\theta^{(i-1)})}. \tag{3.3}$$

Usually we modify Newton's method to include a small step size $0 < \alpha < 1$,

$$\boldsymbol{\theta^{(i)}} = \boldsymbol{\theta^{(i-1)}} - \alpha\, H(J(\boldsymbol{\theta^{(i-1)}}))^{-1} \boldsymbol{\nabla J(\theta^{(i-1)})}\,, \tag{3.4}$$

mainly to ensure convergence in the common case where $J(\boldsymbol{\theta})$ is not quadratic. To get an intuition why Newton's method works (and why it takes only one step for a quadratic $J(\boldsymbol{\theta})$), let us examine the case of a 1-dimensional parameter space.

The 2nd order Taylor approximation of $J(\boldsymbol{\theta})$ around $\theta^{(n)}$ is

$$J(\theta) \approx J(\theta^{(n)}) + (\theta - \theta^{(n)})\frac{\partial J(\theta^{(n)})}{\partial \theta} + \frac{1}{2!}(\theta - \theta^{(n)})^2 \frac{\partial J^2(\theta^{(n)})}{\partial \theta^2}. \tag{3.5}$$

Differentiating both sides of Eq. (3.5) we get

$$\frac{\partial J(\theta)}{\partial \theta} \approx \frac{\partial J(\theta^{(n)})}{\partial \theta} + (\theta - \theta^{(n)})\frac{\partial J^2(\theta^{(n)})}{\partial \theta^2} \tag{3.6}$$

Since at the minimum $\theta^{(*)}$ we have $\frac{\partial J(\theta^{(*)})}{\partial \theta} = 0$, by setting $\theta = \theta^{(*)}$ in Eq. (3.6) and rearranging we get

$$\theta^{(*)} \approx \theta^{(n)} - \left(\frac{\partial J^2(\theta^{(n)})}{\partial \theta^2}\right)^{-1} \frac{\partial J(\theta)}{\partial \theta^{(n)}}. \tag{3.7}$$

Note that a point where the gradient of a function is equal to zero is not necessarily a minimum, but a stationary point (maximum, minimum or saddle point in higher dimensions). In order for Newton's method to converge to a minimum, the Hessian needs to be positive definite (all its eigenvalues are positive). In case it is not, the method converges to a saddle point. If $J(\theta)$ is a quadratic function, then "$\approx$" Eq. (3.5) becomes "$=$" and so does in the rest of the equations, thus the single step convergence to $\theta^{(*)}$.

Limited-memory BFGS

The family of Quasi-Newton methods to mimic Newton's Method, hence the term "Quasi-Newton". They use approximations of the Hessian based on gradient and past update step information instead of the Hessian itself. By not explicitly computing the Hessian, these methods are faster per iteration than pure Newton's method. We shall examine here a variation of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, the Limited-memory BFGS (L-BFGS or LM-BFGS) proposed in [Nocedal, 1980], which is a member of the Quasi-Newton family.

Standard BFGS, proposed independently in [Broyden, 1970], [Fletcher, 1970], [Goldfarb, 1970] and [Shanno, 1970], stores a dense $N \times N$ approximation of $\boldsymbol{H}$, where N is the number of parameters. L-BFGS stores only a few vectors that represent the approximation of $\boldsymbol{H}$, hence its name. Also, while standard BFGS computes the Hessian approximation at iteration k, $\boldsymbol{H}^{(k)}$, using all past updates $i = 1, 2, ..., k-1$, L-BFGS only uses the past m updates $i = k-m, ..., k-1$, where m is a small number (usually less than 10). So L-BFGS keeps record of only the vectors $[\boldsymbol{\theta}^{(k-m)}, ..., \boldsymbol{\theta}^{(k-1)}]^T$ and $[\boldsymbol{\nabla J(\theta^{(k-m)})}, ..., \boldsymbol{\nabla J(\theta^{(k-1)})}]^T$ at iteration k.

In Algorithm 10, we give the pseudocode for the *two loop recursion* version of L-BFGS [Nocedal, 1980]. The algorithm gets as input $\boldsymbol{\theta^{(k)}}$ and $\boldsymbol{\nabla J(\theta^{(k)})}$ and

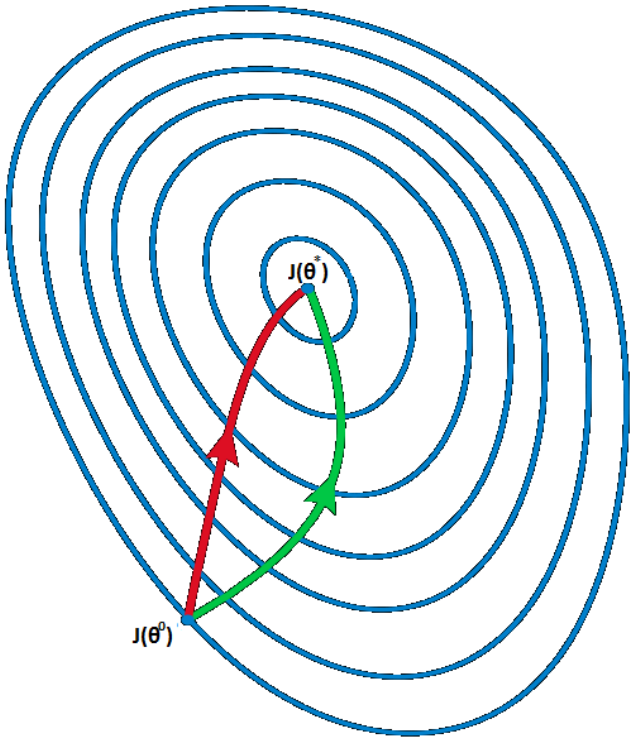

Figure 3.7: A comparison of Newton's Method(red) and gradient descent (green). Newton's method uses curvature information to take a more direct route (notice the "closer contours" in its path which correspond to steeper descent towards the minimum).

Image adapted from Wikipedia: `http://en.wikipedia.org/wiki/File:Newton_optimization_vs_grad_descent.svg`

computes $\boldsymbol{\delta\theta}^{(k)} = \boldsymbol{\theta}^{(k+1)} - \boldsymbol{\theta}^{(k)}$ and $\boldsymbol{\delta\nabla J(\theta^{(k)})} = \boldsymbol{\nabla J(\theta^{(k+1)})} - \boldsymbol{\nabla J(\theta^{(k)})}$ at each iteration. Using these we define $\boldsymbol{p^{(k)}} = \frac{1}{(\boldsymbol{\delta\nabla J(\theta^{(k)})})^T\, \boldsymbol{\delta\theta^{(k)}}}$. The algorithm also requires an initialization of $\boldsymbol{H^{(k)}}$ which we denote with $\boldsymbol{H_0^{(k)}}$. The output of the algorithm is the search direction $\boldsymbol{D}^{(k)} = -\boldsymbol{H^{(k)}}\, \boldsymbol{\delta\nabla J(\theta^{(k)})}$ and we then perform a line search in $\boldsymbol{D}^{(k)}$. This concludes the k-th iteration of L-BFGS. We present this algorithm for completeness. We did not implement L-BFGS, but we did use

the implementation included in Mark Schmidt's minfunc() in our experiments.

Data: $\boldsymbol{\theta}^{(k)}$,$\boldsymbol{\nabla J(\theta^{(k)})}$,$\boldsymbol{H_0^{(k)}}$
Result: $\boldsymbol{D^{(k)}}$
$\boldsymbol{q = \nabla J(\theta^{(k)})}$;
for *i=(k-1):-1:(k-m)* **do**
  $\boldsymbol{\alpha^{(i)} = p^{(i)}\,(\delta\theta^{(i)})^T q}$;
  $\boldsymbol{q = q - \alpha^{(i)}\,\delta\nabla J(\theta^{(i)})}$;
end
$\boldsymbol{D^{(k)} = H_0^{(k)}\,q}$;
for *i=(k-m):(k-1)* **do**
  $\boldsymbol{\beta^{(i)} = p^{(i)}\,(\delta\nabla J(\theta^{(i)}))^T D^{(k)}}$;
  $\boldsymbol{D^{(k)} = D^{(k)} + \delta\theta^{(i)}(\alpha^{(i)} - \beta^{(i)})}$;
end
$\boldsymbol{D^{(k)} = -D^{(k)}}$;

Algorithm 10: The general pseudocode for the two loop recursion version of LBFGS.

L-BFGS is among the most popular general purpose optimization algorithms. Comparative studies to SGD, like the one in [Le et al., 2011], show that it can outperform the latter even in large scale non-convex machine learning problems.

Dynamically Increasing Batch Methods

Finally, we explored the idea of executing batch methods starting with a small subset of the data of size $T' \ll T$ as the initial batch and then dynamically increasing the batch size once there is sufficient evidence that an increase is necessary. This technique was proposed by [Boyles et al., 2011] where it was applied to Coordinate Descent. However the same principle can be applied to any batch method. The criterion that we use in order to decide whether to include more datapoints is the reliability of the computed updates (thus, the reliability of the sign for each computed partial derivative of the gradient). We measure how reliable a computed partial derivative's sign is by performing a statistical test that takes into account how probable it is that the actual sign is not the one we calculated. More precisely, if we have estimated $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} > 0$ then θ_i passes the test if,

$$Pr\left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \leq 0\right] = \Psi\left(\frac{-\mu_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}{\sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}\right) < \epsilon, \tag{3.8}$$

and gets updated. Conversely if we have estimated a negative (mean) partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} < 0$ then θ_i passes the test and gets updated if,

$$Pr\left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \geq 0\right] = 1 - \Psi\left(\frac{-\mu_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}{\sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}\right) < \epsilon. \tag{3.9}$$

We do this for all $i = 1, 2, ...N$ on every iteration. We only update the parameters that pass the statistical test. If all parameters fail the statistical test, then we need to increase the batch size, so as to increase the variance of the sample partial derivatives, using a rule such as: $T' = \lceil T' * M \rceil$, $M \in \mathbb{R}+$. We used a fixed value of $M = 2$ in our implementation. As for the initial batch size value we used $T' = \frac{5 \times T}{100}$ for all datasets. Once all parameters fail their respective partial derivatives' tests with $T' = T$, we terminate the algorithm.

In order for a parameter's (mean) partial derivative to be reliable, we need its absolute value to be large (i.e. to lie far from zero) and its variance to be small. The constant $0 \leq \epsilon \leq 0.5$ is an additional hyperparameter of the algorithm that needs to be adjusted accordingly to optimize performance (see section 4.5 for more on this subject).

The potential benefits of such an implementation are numerous. First of all, by starting with a very small batch we add some stochasticity to the optimizer which could allow it escape local optima early on, much like a stochastic or mini-batch method would do. On the other hand, by the end of the execution we will be running a full batch method, so we asymptotically have the same convergence properties as with batch methods which are better understood compared to stochastic ones. In fact, it can even speed up convergence, since far from the minimum, we need less precision to determine the parameter update than closer to it and this technique takes advantage of it. We only increase the batch size once the ability of the current batch to provide us with statistically reliable updates is exhausted. Once the current batch cannot offer us reliable updates any more, continuing to train on it not only is a waste of time, but can even lead to overfitting. This technique therefore also helps prevent overfitting. As an added bonus, the statistical test provides us with a natural stopping criterion. As a result, there is no further need to use other termination conditions and overfitting safeguards like early stopping (see section 4.5 for details).

However, our results on the task of training the NADE model showed that some of these findings do not carry on to non-convex optimization tasks. It should be noted that in [Boyles et al. , 2011] the authors tested the technique on convex objective functions that included a regularization sub-objective. Thus, possible explanations for the poor performance in our case include the fact that the objective function is convex and the lack of any sort of regularization on our part. *L1*-Regularization makes the objective non-smooth and non-differentiable for any $\theta_i = 0$ so perhaps this was a reason the authors used a derivative-free method like Coordinate Descent in the first place.

For these methods we need the vector of the standard deviations of the partial derivatives,

$$\boldsymbol{\sigma}_{\boldsymbol{\nabla J(\theta)}} = \left[\sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}\right], i = 1, 2, ...N, \tag{3.10}$$

as well, as its mean,

$$\boldsymbol{\mu}_{\boldsymbol{\nabla J(\theta)}} = \left[\mu_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}\right], i = 1, 2, ...N. \tag{3.11}$$

So we modified COMPUTE_OBJECTIVE_AND_GRADIENT() to also output the standard deviation. We aptly named the new version of the function COMPUTE_OBJ_AND_GRAD_MEAN_AND_STDEV(), making a big unimaginative name more of both. It can be seen in Algorithm 11. We remind here that both the mean and the standard deviation are taken over the partial derivatives computed for each instance of the batch and that $\boldsymbol{\nabla J(\theta)} = \boldsymbol{\mu}_{\boldsymbol{\nabla J(\theta)}}$ for our versions of the batch method.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta^{(0)}}$
Result: $J(\boldsymbol{\theta^{(0)}}), \boldsymbol{\nabla J(\theta^{(0)})}, \boldsymbol{\sigma}_{\boldsymbol{\nabla J(\theta^{(0)})}}$
$T' = SIZE(\boldsymbol{X_{train}}, ROWS)$;
// Below, OBJFUNCEVAL($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$) and
COMPUTEGRAD($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$) simply compute $J^{(t)}(\boldsymbol{\theta^{(0)}})$ and
$\boldsymbol{\nabla J^{(t)}(\theta^{(0)})}$ respectively and their implementations differ from
model to model:
$J(\boldsymbol{\theta^{(0)}}) = \frac{1}{T'} \sum_{t=1}^{T'}$ OBJFUNCEVAL($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$);
$\boldsymbol{\nabla J(\theta^{(0)})} = \frac{1}{T'} \sum_{t=1}^{T'}$COMPUTEGRAD($\boldsymbol{X_{train}(t,*)}$, $\boldsymbol{\theta^{(0)}}$);
$\boldsymbol{\sigma \nabla J(\theta^{(0)})} =$
$\sqrt{\frac{1}{T'-1} \sum_{t=1}^{T'} (\boldsymbol{\nabla J(\theta^{(0)})} - COMPUTEGRAD(\boldsymbol{X_{train}(t,*)}, \boldsymbol{\theta^{(0)}}))^2}$;

Algorithm 11: The general pseudocode for the algorithm that computes the value of the objective function and its mean gradient with respect to the parameters, as well as a vector with the standard deviation of its of its elements: COMPUTE_OBJ_AND_GRAD_MEAN_AND_STDEV().

The variants of dynamically increasing batch methods we examine here are the following:

DIBGD_basic: Runs the simple version of Batch Gradient Descent with a dynamically increasing batch size. The pseudocode can be found in Algorithm 12.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}$, $\boldsymbol{\epsilon}$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $terminate = 0$; $grow\_batch = 0$; $\boldsymbol{pass}$ = ZEROS(SIZE($\boldsymbol{\theta}$));
T = SIZE($\boldsymbol{X_{train}}$, ROWS);
$T' = \lceil 0.05\,T \rceil$;
$Batch = \boldsymbol{X_{train(1:T',*)}}$ **while** $terminate == 0$ **do**
  if $grow\_batch == 1$ **then**
    $T' = 2\,T'$;
    $Batch = \boldsymbol{X_{train(1:T',*)}}$;
    if $T' > T$ **then**
      $Batch = \boldsymbol{X_{train(1:T,*)}}$
    end
  end
  $\left[J, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}, \sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}\right] =$
  COMPUTE_OBJ_AND_GRAD_MEAN_AND_STDEV($\boldsymbol{Batch}$, $\boldsymbol{\theta}$);
  for i=1:N **do**
    if $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} > 0$ **then**
      if $\Psi\left(\frac{-\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}{\sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}\right) < \epsilon$ **then**
        $pass(i) = 1$;
      end
    end
    if $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} < 0$ **then**
      if $1 - \Psi\left(\frac{-\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}{\sigma_{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}}}\right) < \epsilon$ **then**
        $pass(i) = 1$;
      end
    end
  end
  `// In the line below ∘ denotes element-wise multiplication,`
  `i.e. update only the parameters that passed their`
  `corresponding test :`
  $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\,\boldsymbol{\nabla J} \circ \boldsymbol{pass}$;
  if $\boldsymbol{pass} == \boldsymbol{0}$ **then**
    $grow\_batch = 1$;
    if $T' == T$ **then**
      $terminate = 1$;
    end
  end
  $\boldsymbol{pass}$ = ZEROS(SIZE($\boldsymbol{\theta}$));
end

Algorithm 12: The general pseudocode for Steepest Gradient Descent with the dynamically increasing batch size technique described in this section: ***DIBGD_basic***.

3.7 Stochastic Methods

We will only examine Stochastic Gradient Descent and some of its variants in this work. There also exist higher order stochastic methods. A simple version of one is to apply a form of Newton Method to one training instance per iteration. Taking this idea a step further we can instead approximate the inverse Hessian and use a Quasi- Newton version of SGD [Bordes et al., 2009]. Most of these however, are complicated, cost more per iteration than simple SGD and it is more difficult to study their convergence properties. Furthermore, what studies have been made [Bordes et al., 2009], [Bottou et al., 2007], point out that at least in the context of training machine learning algorithms, whatever computational benefits are gained are constant and a result of many orthogonal tricks applied to the methods.

Stochastic Gradient Descent

Stochastic Gradient Descent is a first order stochastic optimization method. It can be simply viewed as performing Gradient Descent on only one instance of the training set on each iteration. We can randomly pick an instance on every iteration, or cycle through all T instances within one epoch and run the algorithm for a number of epochs. We chose the second option in our implementation.

Despite its apparent simplicity, in practice SGD and its variants has been very successful for large scale optimization problems, especially in the case of non-convex objectives. Their success, combined with the aforementioned simplicity, has made it the method of choice for many practitioners in large scale and/or non-convex optimization problems in machine learning [Bottou et al., 2004], [Bottou et al., 2007], [Bengio, 2012], [Vishwanathan et al., 2006].

The variants we examine here are the following:

SGD_basic: The simple version of the method we discussed above. The pseudocode can be found in Algorithm 13.

Data: $\boldsymbol{X_{train}}, \boldsymbol{\theta_{initial}}, \alpha$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  for $t = 1 : T$ **do**
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}(t, *)}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla J}$;
  end
end

Algorithm 13: The core pseudocode for the simple version of Stochastic Gradient Descent: SGD_basic.

SGD_heavyball: The "Heavy Ball" version of the method we discussed above. The reasoning behind this heuristic in the SGD setting is to try to moderate the stochastic behaviour of the algorithm by adding contributions from the previous steps to each update. The pseudocode can be found in Algorithm 14.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \gamma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $\boldsymbol{\Delta\theta_{prev}} = \mathbf{0}$;
while *Termination Conditions Are Not Met* **do**
  for $t = 1 : T$ **do**
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}(t, *)}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\, \boldsymbol{\nabla J} - \gamma \boldsymbol{\Delta}\, \boldsymbol{\theta_{prev}}$;
    $\boldsymbol{\Delta\theta_{prev}} = \alpha\, \boldsymbol{\nabla J}$;
  end
end

Algorithm 14: The core pseudocode for the "Heavy Ball" version of Stochastic Gradient Descent: SGD_heavyball.

ASGD_basic: A basic version of the Averaged Stochastic Gradient Descent. Here we use the averaged gradient over all instances visited instead of the gradient of just current instance to compute each step. Again, the idea is to incorporate information from all previous partial derivatives on each step, so as to moderate the stochasticity of SGD. The pseudocode can be found in Algorithm 15.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $T_{passed} = 0$;
while *Termination Conditions Are Not Met* **do**
  for $t = 1 : T$ **do**
    $T_{passed} = ++$;
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}(t, *)}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\mu_{\nabla J}} = \frac{\boldsymbol{\mu_{\nabla J}}(T_{passed}-1)+\boldsymbol{\nabla J}}{T_{passed}}$;
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\, \boldsymbol{\mu_{\nabla J}}$;
  end
end

Algorithm 15: The core pseudocode for the simple version of Averaged Stochastic Gradient Descent: ASGD_basic.

IAGD_basic: A basic version of the Iterate Averaging Stochastic Gradient Descent. Yet another approach for decreasing the stochasticity of the method. We do this by incorporating information from all previous updates on each step

The pseudocode can be found in Algorithm 16.

Data: $\boldsymbol{X_{train}}, \boldsymbol{\theta_{initial}}, \alpha$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $T_{passed} = 0$;
while *Termination Conditions Are Not Met* **do**
  for $t = 1 : T$ **do**
    $T_{passed} = ++$;
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}(t, *)}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\mu_{\Delta\theta}} = \frac{\boldsymbol{\mu_{\Delta\theta}}(T_{passed}-1)+\alpha\,\boldsymbol{\nabla J}}{T_{passed}}$;
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\mu_{\Delta\theta}}$;
  end
end

Algorithm 16: The core pseudocode for the simple version of Iterate Averaging Gradient Descent: IAGD_basic.

All variants we examine are based on the same premise of decreasing stochasticity by keeping a memory of past updates or partial derivatives. The idea of speeding up the convergence of stochastic algorithms via averaging is largely inspired by [Polyak et al., 1992]. Of the methods above, the most successful is ASGD. Evidence for its benefits over SGD (and as an optimization technique in general) can be found in [Le Roux et al., 2012] and [Xu, 2012].

3.8 Mini-Batch Methods

Mini-Batch Gradient Descent

We also included in our experiments some variations of Mini-Batch Gradient Descent. Mini-Batch Gradient Descent, can be thought of as the generalization of Batch Gradient Descent and Stochastic Gradient Descent. Instead of utilizing all T instances of the dataset (as we do in Batch Gradient Descent) or just 1 instance (as we do in Batch Gradient Descent) in order to make an update, we use $1 \leq T_{batch} \leq T$. Obviously for $T_{batch} = T$ Mini-Batch Gradient Descent degenerates into Batch Gradient Descent and for $T_{batch} = 1$ it degenerates into Stochastic Gradient Descent.

The basic procedure is as follows: We split the dataset into B minibatches of size T_{batch}. We then cycle through the minibatches, training our model on each one of them using Batch Gradient Descent and updating its parameters. In our implementation after each epoch (each pass through all the minibatches) we reshuffle the dataset. We continue executing entire epochs until the termination criteria are met (for more details, consult the end of this section).

The variants we examine here are the following:

MBGD_basic: The simple version of the method we discussed above. The pseudocode can be found in Algorithm 17.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  $\boldsymbol{X_{train}}$ = SHUFFLE($\boldsymbol{X_{train}}$, ROWS);
  for $b = 1 : B$ **do**
    $\boldsymbol{Batch} = \boldsymbol{X_{train}}(((\boldsymbol{b}-\boldsymbol{1}) * \boldsymbol{T_{batch}} + \boldsymbol{1}) : (\boldsymbol{b} * \boldsymbol{T_{batch}}), *)$;
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{Batch}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla J}$;
  end
end

Algorithm 17: The core pseudocode for the simple version of Mini-batch Gradient Descent: MBGD_basic.

MBGD_heavyball: Mini-Batch Gradient Descent using the "Heavy Ball" heuristic we presented in Subsection 3.6. We just keep track of the previous step of each parameter and add a contribution of it (i.e. previous step multiplied by the momentum constant) to the new one. The pseudocode for this version can be found in Algorithm 18.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \gamma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$;
while *Termination Conditions Are Not Met* **do**
  $\boldsymbol{X_{train}}$ = SHUFFLE($\boldsymbol{X_{train}}$, ROWS);
  for $b = 1 : B$ **do**
    $\boldsymbol{Batch} = \boldsymbol{X_{train}}(((\boldsymbol{b}-\boldsymbol{1}) * \boldsymbol{T_{batch}} + \boldsymbol{1}) : (\boldsymbol{b} * \boldsymbol{T_{batch}}), *)$;
    $[J, \boldsymbol{\nabla J}]$ = COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{Batch}$, $\boldsymbol{\theta}$);
    $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\nabla J} - \gamma \boldsymbol{\Delta \theta_{prev}}$;
    $\boldsymbol{\Delta \theta_{prev}} = \alpha \boldsymbol{\nabla J}$;
  end
end

Algorithm 18: The core pseudocode for the "Heavy Ball" version of Mini-batch Gradient Descent: MBGD_heavyball.

MBGD_bolddriver: Mini-Batch Gradient Descent using a version of the "Bold Driver" heuristic we presented in Subsection 3.6. We evaluate at each epoch n for each minibatch $b = 1, ..., B$ the value of the objective function on the training set $J(\theta; n, b)$. If in epoch n the value of the objective function has decreased compared to $J(\boldsymbol{\theta}; n-1, b)$, we increase the stepsize slightly, multiplying it by a factor $\rho > 1$, typically around 1.1. If, on the other hand, the value of the objective function has increased compared to $J(\boldsymbol{\theta}; n-1, b)$, we decrease the stepsize severely, multiplying it by a factor $0 < \sigma < 1$, typically around 0.5. It should be noted

that in this version we don't shuffle the dataset at the beginning of each epoch. This is important, because otherwise direct comparison of the value of the error function on the same minibatch in two consecutive epochs would be impossible. Instead, we only do it once, in the beginning of the execution. The pseudocode for this approach can be found in Algorithm 19.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{\theta_{initial}}, \alpha, \rho, \sigma$
Result: $\boldsymbol{\theta}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $J(\boldsymbol{\theta}; b)_{prev} = \infty, for\ b = 1, 2, ..., B; e = 0$;
$\boldsymbol{X_{train}}$ = SHUFFLE($\boldsymbol{X_{train}}$, ROWS);
while *Termination Conditions Are Not Met* **do**
    e++;
    for $b = 1 : B$ **do**
        $[J(\boldsymbol{\theta}; b), \boldsymbol{\nabla} J(\boldsymbol{\theta}; b)] =$
        COMPUTE_OBJECTIVE_AND_GRADIENTS($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}$);
        $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\, \boldsymbol{\nabla} J(\boldsymbol{\theta}; b)$;
        if $J(\boldsymbol{\theta}; b) < J(\boldsymbol{\theta}; b)_{prev}$ **then**
            $\alpha = \alpha\, \rho$;
        else if $J(\boldsymbol{\theta}; b) > J(\boldsymbol{\theta}; b)_{prev}$ **then**
            $\alpha = \alpha\, \sigma$;
        else
            `// Do Nothing...`
        end
        $J(\boldsymbol{\theta}; b)_{prev} = J(\boldsymbol{\theta}; b)$;
    end
end

Algorithm 19: The core pseudocode for the "Bold Driver" version of Mini-batch Gradient Descent: MBGD_bolddriver.

Chapter 4

Hyperparameter Selection

In the previous section we presented the methods we used in order to optimize the parameters of the model we were working with. However, all these methods have, in turn, parameters of their own, also known as hyperparameters, the choice of which can greatly impact their performance both in terms of speed of convergence and in terms of the quality of the resulting solution. But how do we choose the values of these hyperparameters? After all this is a new optimization problem in its own right. Certainly applying similar techniques to the ones discussed seems too costly. Also, the new optimization problem poses new challenges as some hyperparameters assume values not in $\mathbb{R}$ but in subsets of it e.g. in $\mathbb{Z}$ or in some cases we have constraints involving the hyperparameter values, so such techniques cannot even be used. Thus, we should seek other alternatives for hyperparameter optimization.

So at this point we have to make some choices: Which hyperparameters should we fix and at which values? Which ones should we try to optimize? What search space should we explore and using what optimization technique? How should we evaluate our choices for these hyperparameters? How much of our resources should we allocate for this procedure? In this section we will discuss these choices, as well as the reasons behind each one of these backing them up with theoretical arguments or experimental results wherever appropriate. Main sources of inspiration include [Hinton, 2010], [LeCun et al. , 1998] and [Bengio, 2012].

4.1 Step Size

A hyperparameter that is common to all the methods examined is the step size α. A large step size can result to failure of the method to converge or even lead to divergence. A small step size can lead to very slow convergence. So we need to find a step size that balances between these cases. Although there are techniques that allow us to select the (locally) optimal step size on every iteration (e.g. by performing a line search). In the case of a quadratic objective function, the ideal step size (i.e. individual step size for each partial derivative) would be the inverse

of the Hessian of the objective wrt the parameters $\boldsymbol{H^{-1}}$, as demonstrated by Eq. (3.7). It would allow us to reach the optimum in a single step. If our objective function is an approximation of a quadratic (every function is quadratic in a small enough region) we can still use this technique, however, it requires the expensive computation of the Hessian (or an approximation of it) and in practice more than one step.

A simpler and less time consuming common practice to setting the step size is to start with a relatively big value and let it decrease. The idea is that initially, we are likely to be far from a local minimum, so we have much ground to cover towards reaching it, there is no big danger of overshooting it and thus a large step size is beneficial as it allows us to get close to the minimum in relatively few steps. As the optimization progresses and we get closer and closer to the minimum, however, we should decrease the step size to avoid overshooting it. This approach has the benefits we mentioned over optimizing the step size on every iteration, but can yield poor convergence.

So how do we decide the initial value of the step size? And how should we set its decrease schedule? We will start by answering the second question first. It has been shown [Robbins et al. , 1951] that if we use a step size sequence $\{\alpha(n)\}$ such that

$$\sum_{n=1}^{\infty} \alpha(n) = \infty \tag{4.1}$$

and

$$\sum_{n=1}^{\infty} \alpha^2(n) < \infty \tag{4.2}$$

then $\{\boldsymbol{\theta}(n)\}$ will converge.

In our implementation we chose the following schedule [Bengio, 2012], an illustration of which is shown in Figure 4.1:

$$\alpha(n) = \frac{\alpha_0 \, C}{max\{C, n\}} \tag{4.3}$$

The idea is to keep the step size constant for C iterations and then decrease it inverse proportionately to the number of iterations.

So now we only need to set two hyperparameters: α_0 and C. The most common way to set these hyperparameters is to do a grid search on the 2-D hyperparameter space. In other words, we try a number of discrete combinations of values for the two parameters in a subset of the parameter space, since searching for continuous values would be hopeless. As to what values we should include in the set, for α_0 it makes more sense to use a logarithmic grid e.g. $\{0.01, 0.1, 1, 10, 100\}$ in order

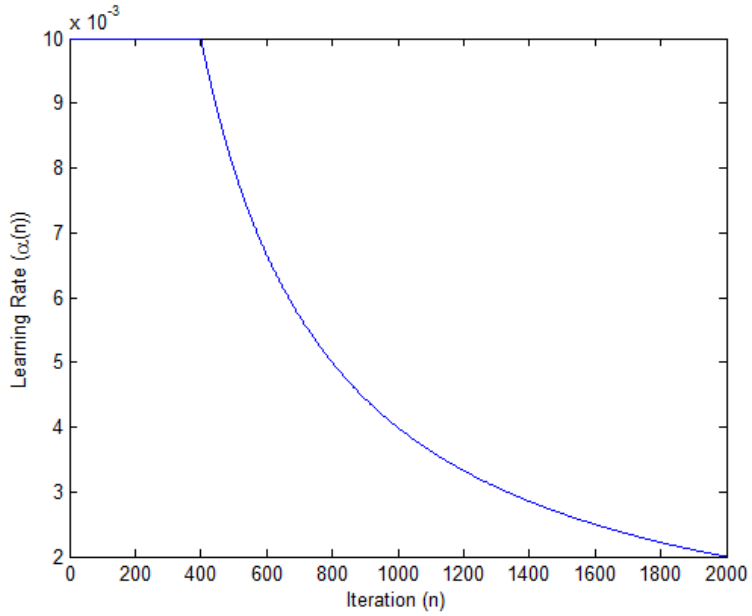

Figure 4.1: An illustration of the learning rate decrease as training progresses Here we chose a starting value of 0.01 and a value of 400 for C. We run the optimizer for 2000 iterations. As we can see, the step size remains constant for the first 400 iterations and then decreases proportionately to the inverse of n.

to capture the order of magnitude of the scaling that needs to be applied by the step size. Once we do that we can choose to adjust it more finely but here we chose not to further optimize it.

In our implementation we searched within $\{5, 0.5, 0.05, 0.005, 0.0005\}$ for α_0. As for C, for batch methods we searched within $\{0, 10, 25, 50, 80\} \times \frac{MAX\_ITERS}{100}$. So we keep the step size constant until $0\%, 10\%, 25\%, 50\%, 80\%$ of the maximum iterations allowed have passed and then start decreasing it. So we range from starting to decrease the learning rate right away (for option 0) to not decreasing (almost) throughout the execution of the algorithm (most do not run for all MAX_ITERS terations). For stochastic methods we use a similar scheme, but modify it a bit to start the decrease after a discrete number of epochs has passed, mainly for aesthetic reasons: $\{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times T}{100}$. We act similarly for minibatch methods searching within $\{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times B}{100}$. We still update the step size on every iteration in both of these cases.

In case we have a tie of two or more parameter configurations (more on how we evaluate them in subsection 4.7), we resolve it by picking the one with the largest α_0 and the smallest C, on the grounds that these will lead to faster convergence. Further ties, e.g. $(\alpha_0(i), C(i)) = (0.1, 1000)$ versus $(\alpha_0(j), C(j)) = (1, 100)$ were resolved by randomly selecting one configuration with equal probability. We mention this for completeness, although such a situation never actually arose in practice.

Here we should mention that there is also the problem of not terminating due

to the updates becoming too small and decreasing without actually becoming equal to zero. As we get closer and closer to a local minimum in a dimension of the parameter space, the partial derivative of the objective function w.r.t. that parameter will tend towards zero. We also haven't mentioned a way to explicitly set the learning rate to zero. So the updates might end up getting smaller and smaller at each iteration but never actually reaching a zero value as this "slowing down" prevents us from reaching the minimum (so $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \neq 0$). We also never necessarily terminate (more precisely, we always terminate due to MAX_ITERS or MAX_EPOCHS criteria), since a decrease on the value of the objective function, albeit a minute one will occur, because ($\alpha(n) \neq 0$). This prevents early stopping from activating (for a detailed explanation see 4.6. We avoid the problem of non-termination by forcing any step size less than 10^{-8} to become zero. Hopefully the values selected for $\alpha(0), C$ will not let this occur often.

4.2 Adaptive and Individual Step Sizes

Another idea for adjusting the step size is to use of adaptive step sizes. This is the basis of heuristics such as Bold Driver and R-prop we described in the previous section. Here, as we saw we have two more hyperparameters to handle: $\rho > 1$ (so as to increase the learning rate when it multiplies it) and $0 < \sigma < 1$ (so as to decrease the learning rate when it multiplies it). We want the increase to be small and the decrease to be considerable. Initially we also included these hyperparameters on the grid search performed, searching in $\{1.3, 1.2, 1.1, 1.01, 1.001\}$ for σ and in $\{0.5, 0.6, 0.7, 0.8, 0.9\}$ for ρ. In case of ties we favoured bigger values for ρ and smaller for σ to encourage faster convergence. However, we ended up choosing to use the fixed values of $\rho = 1.01$ and $\sigma = 0.6$. The reason was that these were the choices of the hyperparameter selection algorithm in most cases anyway and adding two more dimensions in our grid search would make computation time too much to bear, as it would raise the number of dimensions to 4 or even 5.

Unless explicitly stated otherwise (i.e. in all cases except for methods using the R-prop heuristic), each parameter uses the same stepsize. And even in R-prop, we used the same global initial step size value for all parameters. These choices were made mainly to keep the total number of hyperparameters involved small (up to 3) so that the combinatorial search could be performed without adding a substantial overhead to the entire process. Individual learning rates, individual initializations, even individual update schemes for the learning rates are all conceivable options, however. We can even opt for treating the parameters as blocks (e.g. different learning rates for the biases and different for the weights). For the same computational cost-related reasons we only included 5 values as options for each hyperparameter.

4.3 Momentum

In applications of the Heavy Ball heuristic, the momentum term γ is another hyperparameter of the method. We could choose a sequence of momentum values $\{\gamma(n)\}$, i.e. a different one for each iteration n, using an update scheme for γ in a similar fashion we did for the step size α. However this would involve extra hyperparameters, so we opted for a fixed momentum term as most practitioners do. We searched a logarithmic grid on this hyparameter dimension to select the most appropriate value, as well, in particular $\{1, 0.1, 0.01, 0.001, 0.0001\}$. For MBGD_basic, in the final version of our algorithm, we have a fixed $\gamma = 0.01$ to keep the total number of free hyperparameters small.

4.4 Batch Size

In minibatch methods the size of the minibatches is another hyperparameter of the algorithm. For $T_{batch} = T$ we get a batch method and for $T_{batch} = 1$ a stochastic method so its choice can greatly affect the performance of the algorithm. The values we selected the batch size from were $\{2, 4, 5, 8, 10\} \times \frac{T}{100}$. As one can see we focused on relatively small values for T_{batch} compared to T, as we want to lean more towards a "stochastic but with more instances per iteration than actual SGD" approach that would enable us to take advantage of the stochasticity of the updates of SGD but moderate it and allow for vectorization to keep the cost per iteration low.

4.5 Epsilon in Statistical Tests

In the Dynamically Increasing Batch Size approaches the probability $0 \leq \epsilon \leq 0.5$, is another hyperparameter. It is the cutoff below which the probability that the computed sign of a partial derivative is not the one we calculated is considered not statistically significant and the statistical test passes, thus we accept the update. If the probability that the sign is not the one we predicted exceeds ϵ, then we consider it statistically significant and the test fails, thus we do not accept the update. For $\epsilon = 0$ we end up never passing a statistical test, thus never accepting an update while for $\epsilon = 0.5$ we end up never failing a statistical test, therefore always accepting the parameter updates and never terminating (or increasing the initial batch size for that matter). Obviously both of these extreme cases would lead to completely useless optimizers, so we avoided them. We also selected ϵ from a grid of values, in particular from $\{0.05, 0.1, 0.2, 0.3, 0.4\}$ during the hyperparameter selection step.

4.6 Termination Criteria and Overfitting Control

We must specify under what conditions the optimization procedure should terminate. In Dynamically Increasing Batch Size methods, a statistical test provided us with a natural stopping criterion which also safeguarded against overfitting, so in these cases, no further termination condition needed to be specified. However, just to make sure that the optimizer wouldn't get stuck in some extreme cases where the parameters would take a very long time to all fail their respective tests we imposed a slightly stricter condition: if the number of parameters that fail the test does not change for MAX_ITER_SINCE_MOST_FAILED iterations on the entire dataset, then terminate.

For all other methods we needed to impose some other types of termination criteria. One shared across all methods is the maximum iterations allowed. If a batch method has run for MAX_ITERS iterations or a stochastic (or minibatch) method has run for MAX_EPOCHS epochs, then it should terminate. Again, this criterion is more of a safeguard against infinite execution than a refined termination condition. In our implementation we used MAX_ITERS = 400 or MAX_EPOCHS = 400 for all methods examined.

The most meaningful termination method we used in every method implemented except for the Dynamically Increasing Batch Size methods we discussed above (i.e. all save for L-BFGS for which we used the pre-existing minfunc() implementation) was early stopping. At this point we should remember that our actual goal is not optimization in its own right, but learning. And as we know learning implies generalization. Our aim therefore is not to minimize the training error, but the generalization error. In other words, our goal is not to find the parameter values that minimize the value of the objective function on the training set, but the expected loss on an unseen test set.

We use the value of the objective function on the validation set in order to estimate this generalization error. We keep track of the current best solution and stop training once we detect no improvement on the value of the objective function on the validation set over the current best solution for MAX_ITERS_SINCE_BEST iterations (or MAX_EPOCHS_SINCE_BEST epochs where applicable). From this point on, further training would lead to a decrease in training error at the cost of generalization, in other words, in overfitting. In Algorithms 20 and 21 we can find two slightly different implementations of early stopping. The first one was used for batch methods with MAX_ITERS_SINCE_BEST = 5 and the second one for stochastc methods with MAX_EPOCHS_SINCE_BEST = 5 and minibatch methods for MAX_EPOCHS_SINCE_BEST = 5. The vector $\boldsymbol{v}$ is the vector of hyperparameters of the specific optimizer used.

Other possible termination criteria we can use include the following:

(i) Terminate if the parameters do not change substantially. In this scenario, we need to define 'substantially' which might turn out to be tricky since we

can't guarantee that the effect of small changes in a parameter doesn't bear significant changes in the objective function. The surface of the objective function can vary in steepness from dimension to dimension, so our definition of 'substantially' should be different for each parameter to accommodate that.

(ii) Terminate if the value of the objective function does not improve substantially. In this case we also need to define 'substantially' but this task can be more obvious, since the value of the objective function is what we are actually trying to minimize. We can set a tolerance constant of e.g. $TOL = 10^{-3}$ and if $|\frac{J_{train}(i)-J_{train}(i-1)}{J_{train}(i)}| < TOL$ then we terminate. In L-BFGS runs we used this exact criterion. The value of TOL would ideally have to be different for each problem (dataset) and we could have even compared it to $|\frac{J_{valid}(i)-J_{valid}(i-1)}{J_{valid}(i)}|$ instead, for a 'stricter' version of early stopping. We did not do so, however.

Data: $\boldsymbol{X_{train}}$, $\boldsymbol{X_{valid}}$, $\boldsymbol{\theta_{initial}}, \boldsymbol{v}$, MAX_ITERS, MAX_ITERS_SINCE_BEST
Result: $\boldsymbol{\theta_{best}}$
$\boldsymbol{\theta} = \boldsymbol{\theta_{initial}}$; $J_{best} = \infty$; i = 1; terminate = 0; itersSinceBest = 0;
while *i < MAX_ITERS AND terminate == 0* **do**
  $\boldsymbol{\theta_{new}}$ = BATCH_OPTIMIZER($\boldsymbol{X_{train}}$, $\boldsymbol{\theta}, \boldsymbol{v}$);
  $J(\boldsymbol{\theta_{new}})$ = OBJFUNCEVAL($\boldsymbol{X_{valid}}$, $\boldsymbol{\theta_{new}}$);
  if $J(\boldsymbol{\theta_{new}}) < J_{best}$ **then**
    $\boldsymbol{\theta_{best}} = \boldsymbol{\theta_{new}}$;
    $J_{best} = J(\boldsymbol{\theta_{new}})$;
  else
    itersSinceBest++;
    if *itersSinceBest == MAX_ITERS_SINCE_BEST* **then**
      terminate =1;
    end
  end
  i++;
end

Algorithm 20: Early stopping incorporated into a batch optimization method.

Data: $\boldsymbol{X}_{train}$, $\boldsymbol{X}_{valid}$, $\boldsymbol{\theta}_{initial}, \boldsymbol{v}$, MAX_EPOCHS, MAX_EPOCHS_SINCE_BEST
Result: $\boldsymbol{\theta}_{best}$
$\boldsymbol{\theta} = \boldsymbol{\theta}_{initial}$; $J_{best} = \infty$; i = 1; terminate = 0; epochsSinceBest = 0;
while *i < MAX_EPOCHS AND terminate == 0* **do**
  $\boldsymbol{\theta}_{new}$ = RUN_OPTIMIZER_FOR_ONE_EPOCH($\boldsymbol{X}_{train}$, $\boldsymbol{\theta}, \boldsymbol{v}$);
  $J(\boldsymbol{\theta}_{new}; Validation)$ = OBJFUNCEVAL($\boldsymbol{X}_{valid}$, $\boldsymbol{\theta}_{new}$);
  if $J(\boldsymbol{\theta}_{new}) < J_{best}$ **then**
    $\boldsymbol{\theta}_{best} = \boldsymbol{\theta}_{new}$;
    $J_{best} = J(\boldsymbol{\theta}_{new})$;
  else
    epochsSinceBest++;
    if *epochsSinceBest == MAX_EPOCHS_SINCE_BEST* **then**
      terminate =1;
    end
  end
  i++;
end

Algorithm 21: Early stopping incorporated into a mini-batch or a stochastic optimization method. The only difference here is that we check for early stopping after an entire epoch has passed instead of after every iteration. RUN_OPTIMIZER_FOR_ONE_EPOCH() runs an optimization method for an entire epoch and returns the values of parameters after the end of the last iteration of the epoch.

4.7 Evaluation

Finally, let us discuss now how we actually evaluated our choices for the hyperparameters in order to select the most appropriate values. Run the optimization algorithm for each hyperparameter configuration using the entire training set and choosing the one that produces the smallest value of the objective function would be far too costly computationally. For up to 3 free hyperparameters and 5 choices per each we would end up running each method on the entire dataset up to $3^5 = 125$ times. Instead, we randomly selected a small amount of each training set, in particular 10% on which to train for each of these configurations for a small number (20) of iterations or epochs (wherever applicable). We used early stopping with a stricter condition (MAX_ITERS_SINCE_BEST = 2 for batch methods, MAX_EPOCHS_SINCE_BEST = 2 for minibatch and stochastic ones) than in the actual optimization procedure to allow us to drop quickly bad hyperparameter choices and not waste time on them. We end up running each method 125 more times but (i) we do so using the $\frac{1}{10}$-th of the data, (ii) we run for far fewer iterations, (iii) we are more strict in the early stopping condition. As a result the computational cost is far smaller.

Selecting hyperparameters based only on the value of the objective function on the training set will lead to overfitting. Instead of using the training set for both training and hyperparameter selection, we should use a different set for the second task. So we evaluated for each hyperparameter configuration the value of the objective function on the validation set after training and selected the configuration that produced the smallest one among them as the one to use in the actual optimization procedure.

Table 4.1: The free hyperparameters of each method and their range of values in the grid search, as well as those we fixed to specific values after experimentation.

Method	Free Hyperparameters	Fixed Hyperparameters
BGD_basic	$\alpha \in \{5, 0.5, 0.05, 0.005, 0.0005\}$	-
BGD_heavyball	$\alpha \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $\gamma \in \{1, 0.1, 0.01, 0.001, 0.0001\}$,	-
BGD_bolddriver	$\alpha \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $\rho \in \{1.3, 1.2, 1.1, 1.01, 1.001\}$, $\sigma \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$	-
BGD_rprop	${\alpha_0}^{(i)} \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, with ${\alpha_0}^{(i)} = {\alpha_0}^{(j)}\ \forall i, j$, $\rho \in \{1.3, 1.2, 1.1, 1.01, 1.001\}$, $\sigma \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$	-
CD_basicGD	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$	-
L-BFGS	-	(used default minfunc() options)
DIBGD_basic	$\alpha \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $\epsilon \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$	-
SGD_basic	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times T}{100}$	-
SGD_heavyball	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times T}{100}$, $\gamma \in \{1, 0.1, 0.01, 0.001, 0.0001\}$	-
ASGD_basic	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times T}{100}$	-
IAGD_basic	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times T}{100}$	-
MBGD_basic	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times B}{100}$, $batchsize \in \{2, 4, 5, 8, 10\} \times \frac{T}{100}$	-
MBGD_heavyball	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times B}{100}$, $batchsize \in \{2, 4, 5, 8, 10\} \times \frac{T}{100}$	$\gamma = 0.01$
MBGD_bolddriver	$\alpha_0 \in \{5, 0.5, 0.05, 0.005, 0.0005\}$, $C \in \{0, 10, 25, 50, 80\} \times \frac{MAX\_EPOCHS \times B}{100}$, $batchsize \in \{2, 4, 5, 8, 10\} \times \frac{T}{100}$	$\rho = 1.01$, $\sigma = 0.6$

4.8 Alternative Ways to Search the Hyperparameter Space

As a closing note, before settling for a grid search on the hyperparameter space (and keeping its dimensionality $N_\upsilon \leq 4$ by choosing fixed values for the rest of the hyperparameters), we briefly experimented with another hyperparameter selection scheme for methods that involved $N_\upsilon \geq 4$ hyperparameters (thus at least $5^4 = 625$ hyperparameter combinations).

We used a Genetic Algorithm to explore the parameter space, with an initial population of 20 individuals, each of which represented a candidate vector of hyperparameters $\boldsymbol{\upsilon}^{(i)} \in \mathbb{R}^{\mathbb{N}_\upsilon}$ initialized using values selected uniformly at random from the same set of values for each hyper parameter on which we performed the grid search. We used each individual as a hyperparameter vector for the method used and run 20 iterations with an early stopping lookahead MAX_ITERS_SINCE_BEST = 2 for batch methods, MAX_EPOCHS_SINCE_BEST = 2 for minibatch and stochastic ones. To evaluate each candidate hyperparameter combination $\boldsymbol{\upsilon}^{(i)}$, we used as a fitness function $F(\boldsymbol{\upsilon}^{(i)}) = J(\boldsymbol{\theta}; \boldsymbol{\upsilon}^{(i)})/N_{iter}$, where $J(\boldsymbol{\theta}; \boldsymbol{\upsilon}^{(i)})$ is the final value of the objective function on the validation set and N_{iter} the number of iterations until convergence. The 10 fittest individuals would be selected for crossover and would move on to the next generation along with their 10 offspring. For recombination we used single-point crossover among 5 randomly (with equal probability) chosen pairs of the selected individuals. In other words, both parent vectors were cut in the same position ψ selected uniformly at random from $\{1, 2, ..., N_\upsilon\}$ and the four resulting parts were combined to form 2 new individuals of length N_υ each, with one part from each parent. As for the mutation operation, each element of the offspring would switch its value with one of the other 4 possible ones in its corresponding hyperparameter set with equal probability. We used a mutation rate of 0.1. The above describe one generation of the algorithm, we run for 25 generations.

This method was faster than the grid search, since it was designed as a means to reduce running time. The maximum number of iterations of the optimization method would be $20 \times 25 \times 20 < 5^4 \times 20$ of the grid search. However, the resulting hyperparameter choices were usually poor. We found that by fixing certain hyperparameters to specific values we get better results with a grid search on the remaining dimensions and faster(up to $5^3 \times 100$ iterations of the optimization method). The genetic algorithm also has hyperparameters and implementation choices of its own we could have tweaked (mutation rate, definition of mutation and recombination operations, initial population size, allowed values for each element of the individual, selection scheme, fitness function, termination conditions etc). We did not experiment any further with these choices, since we risk entering a vicious cycle. We started off trying to optimize the parameters θ_i of our model, now we seek to optimize the hyperparameters υ_j of the optimization algorithms, it is better if we do not do so using a method that involves hyperparameters of its own in need of optimization.

Chapter 5

Gradient Code Debugging and Verification

Gradient code is particularly error prone and, to make matters worse, these errors can be hard to detect. This is so, because plotting the value of the objective function per iteration usually gives us little information about whether our gradient code is correct. In batch methods this quantity is supposed to be decreasing monotonically per iteration (as long as it does not overshoot the minimum) and in stochastic methods it is supposed to decrease noisily as training progresses (i.e. it is allowed to increase from last iteration as well). Indeed, the same can happen even if we miscalculate the gradient. If we just get its sign right, we will take a step in the correct direction although the size of the step is not the one intended. If we use an individual learning rate per parameter, perhaps this scaling error can be countered (given that the step sizes are allowed to be adapted to appropriate values during training), however using a global step size (which might be the most common practice) scaling errors can complicate matters. Furthermore, especially when the model has many parameters, even missing some partial derivative signs can lead to a decrease in the value of the objective function, since we might have moved to better values in the rest of the parameter dimensions. The point of all this is that e.g. using a batch method, the value of the objective can be decreasing monotonically, the method might appear to be working, but at its core (gradient computation) the code could still be wrong. So we cannot use the value of the objective function as a means of testing our gradients.

5.1 The Finite Differences Method

One method to check whether the code that computes the gradient of the objective function is correct (or better: "agrees with the objective function") is to use the Finite Differences method. This technique is used to estimate derivatives numerically using (first order) finite differences. From the Taylor series we have

$$J(\theta_i^{(0)} + h) = J(\theta_i^{(0)}) + h\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} + \frac{1}{2!}h^2\frac{\partial J^2(\theta_i^{(0)})}{\partial {\theta_i}^2} + \frac{1}{3!}h^3\frac{\partial J^3(\theta_i^{(0)})}{\partial {\theta_i}^3} + R^{(4)} \quad (5.1)$$

and

$$J(\theta_i^{(0)} - h) = J(\theta_i^{(0)}) - h\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} + \frac{1}{2!}h^2\frac{\partial J^2(\theta_i^{(0)})}{\partial {\theta_i}^2} - \frac{1}{3!}h^3\frac{\partial J^3(\theta_i^{(0)})}{\partial {\theta_i}^3} + R^{(4)}, \quad (5.2)$$

where $R^{(k)}$ corresponds to the sum of terms involving derivatives of order k and higher.

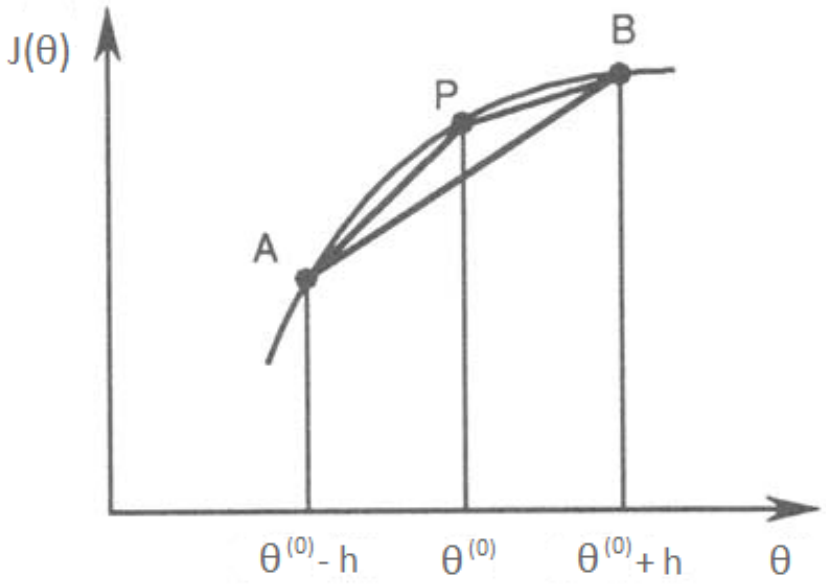

Figure 5.1: We wish to estimate the derivative of $J(\theta)$ wrt θ at θ_0. It is the slope of the curve at point P.

Image adapted from
`http://staff.unila.ac.id/tiryono/files/2011/09/BedaHingga-FDM.pdf`

We can then approximate each partial derivative $\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i}$ (i.e. the slope at P in Figure 5.1) using the forward difference formula, which corresponds to computing the slope of PB in Figure 5.1,

$$\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} \approx \frac{J(\theta_i^{(0)} + h) - J(\theta_i^{(0)})}{h} \quad (5.3)$$

the backward difference formula, which corresponds to computing the slope of AP in Figure 5.1,

$$\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} \approx \frac{J(\theta_i^{(0)}) - J(\theta_i^{(0)} - h)}{h}. \tag{5.4}$$

or the central difference formula, which corresponds to computing the slope of AB in Figure 5.1,

$$\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} \approx \frac{f(\theta_i^{(0)} + h) - J(\theta_i^{(0)} - h)}{2h} \tag{5.5}$$

We can obtain Eq.(5.3) and Eq.(5.4) by rearranging Eq.(5.1) and Eq.(5.2) respectively and truncating the $R^{(2)}$ term (hence the approximation). To get Eq.(5.4) we add Eq.(5.1) and Eq.(5.2) and truncate the $R^{(2)}$ term. The smaller h is the better the approximation. Rememder, perhaps the most common definition of the derivative of the function $J(\theta_i)$ at $\theta_i^{(0)}$ is

$$\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} = \lim_{h \to 0} \frac{J(\theta_i^{(0)} + h) - J(\theta_i^{(0)})}{h} \tag{5.6}$$

We chose the (1) for simplicity and used $h = 10^{-9}$. We approximated the partial derivatives with respect to each $\theta_i, i = 1, 2, ..., N$ using this procedure. We computed the mean absolute relative error

$$E = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i} - \frac{J(\theta_i^{(0)} + h) - J(\theta_i^{(0)})}{h}}{\frac{\partial J(\theta_i^{(0)})}{\partial \theta_i}} \right| \tag{5.7}$$

of the approximation for parts of our datasets. In every case we examined it was of order $\leq 10^{-7}$ which was deemed acceptable. Some of the models examined had a large number of parameters (e.g. NADE has $H + D + 2 \times H \times D$, with H up to 500 and D at least 112 in our datasets), which meant that using all features (and many hidden variables in NADE) would mean computing partial derivatives with respect to a large number of parameters using the finite differences method. This made the computational cost prohibitive, so instead we checked the approximation using only a few features and $H = 5$ for NADE. Due to its high computational cost, performing gradient checks during execution of the methods was only used at this stage for sanity checks and it was subsequently deactivated during the actual execution of the methods.

5.2 Toy Problem: Optimizing a Logistic Regression Model

We tested some of our optimization methods on the toy problem of performing logistic regression for binary classification on an artificially generated dataset with $D = 2$. We chose a two-dimensional feature space in order to visualize and present on paper more easily the results obtained.The data were generated by two Gaussians, one with mean $\boldsymbol{\mu_1} = (-1, 0)$ and covariance matrix $\boldsymbol{\Sigma} = \boldsymbol{I}$ and another one with mean $\boldsymbol{\mu_0} = (3, -2)$ and covariance also $\boldsymbol{\Sigma} = I$. The first one generated the 1000 samples belonging to class 1 (positive examples, depicted with a red 'X' in the figures below) and the second one the 1000 samples belonging to class 0 (negative examples, depicted with a green 'O' in the figures below).

In Table 5.1 we can see the results after training on $T = 1500$ instances using a test set of 500 instances. We used fixed values for all hyperparameters ($\alpha = 0.1$ in every method, $\gamma = 0.05$ in BGD_heavyball, batchsize = 100 in mbgd_basic) in this case, as the problem was trivial and its aim just to check our algorithms. All methods ran once for 300 iterations/epochs (where applicable) each.

We also included Newton's method in our experiments using a single iteration, just to demonstrate its power relatively to the first order methods. More iterations might actually have lead to overfitting since we took no measures to avoid it. We used the rule based on Eq. (3.4) for Newton's method with a step size of $\alpha = 1$. We calculated the Hessian[1] as

$$H = -\boldsymbol{X_{train}}^T \boldsymbol{B} \boldsymbol{X_{train}}, \tag{5.8}$$

where$\boldsymbol{X_{train}}$ is the data matrix (augmented by a column of 1s to accommodate the bias terms), and $\boldsymbol{B} = \sigma_{\boldsymbol{\theta}}(\boldsymbol{X_{train}}) * (1 - \sigma_{\boldsymbol{\theta}}(\boldsymbol{X_{train}}))^T$. In this scenario, we have only 3 parameters, therefore the Hessian is a 3×3 matrix, which is cheap to compute and work with.

On Figure 5.2 we can see a visualization of the data and the decision boundary on both the training and the testing set before training. On Figures 5.3 to 5.9 we show the datapoints and the decision boundary, on both the training and the testing set, after training with each method.

Note that this is a toy problem. The objective function is convex, the dataset is small and it consists of only two features (so logistic regression needs only three parameters). At this point our goal is not to compare the methods to one another. We did not optimize their hyperparameters to do so, neither did we employ early stopping, but rather to verify that they are working properly.

[1]We could use the finite differences method to check the second partial derivatives as well, taking finite differences of first partial derivatives. So for every partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}$, $i = 1, 2, ..., N$ we must compute the approximate N partial derivatives $\frac{\partial^2 J(\boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}$, $j = 1, 2, ..., N$. Now the finite differences method was already expensive and this procedure will be N times more expensive. We did not perform this check here.

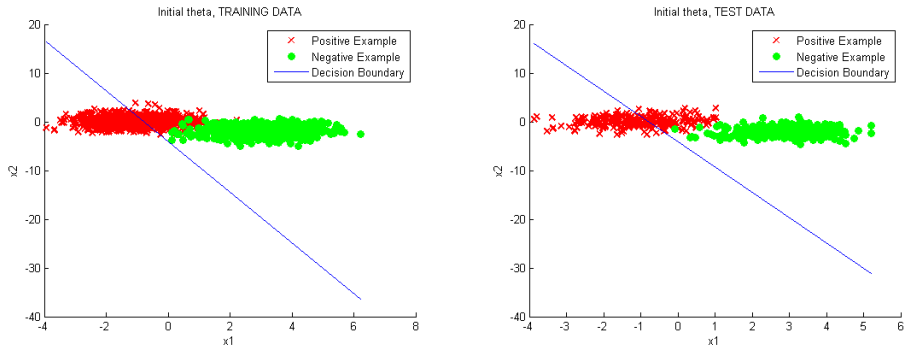

Figure 5.2: Decision boundary before training on Training set (left) and Test set (right).

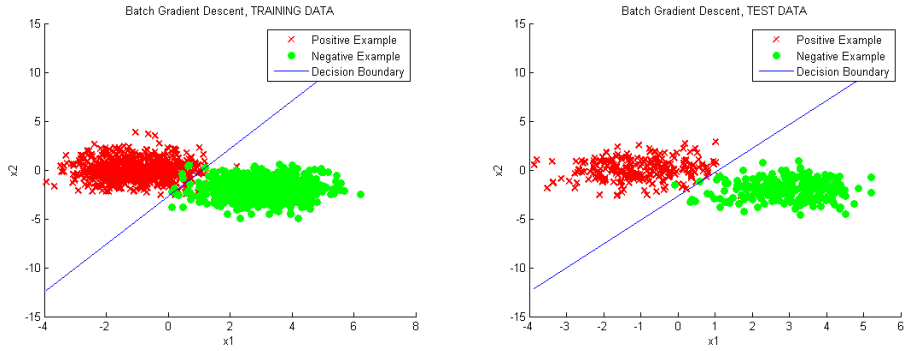

Figure 5.3: Decision boundary after training with BGD_basic on Training set (left) and Test set (right).

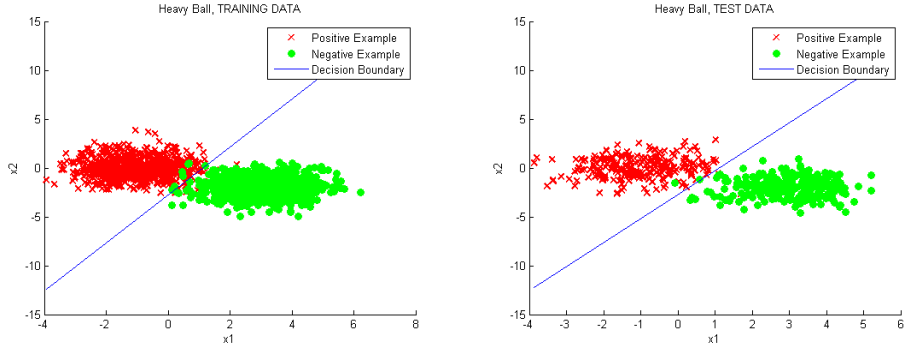

Figure 5.4: Decision boundary after training with BGD_heavyball on Training set (left) and Test set (right).

Notice, however that CD_basicGD performs as well as the other methods. This will not be the case in the optimization of NADE's objective function as we will see in the next section. Also, notice that CD_basicNewton gives a better solution than CD_basicGD. In fact this is so because it is faster. Unfortunately we did not include CD_basicNewton in further studies (i.e. on NADE or on FVSBN) as we only implemented it during the final stages of this dissertation as a viable alternative to CD_basicGD. The results shown here indicate that indeed CD_basicNewton has an advantage over CD_basicGD. As we see from Figures

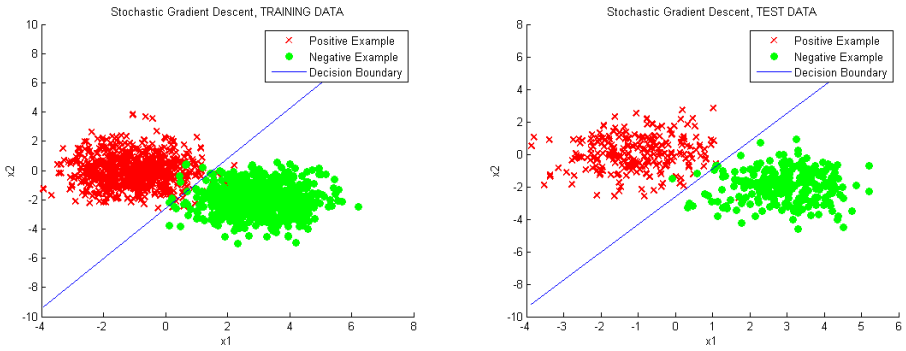

Figure 5.5: Decision boundary after training with SGD_basic on Training set (left) and Test set (right).

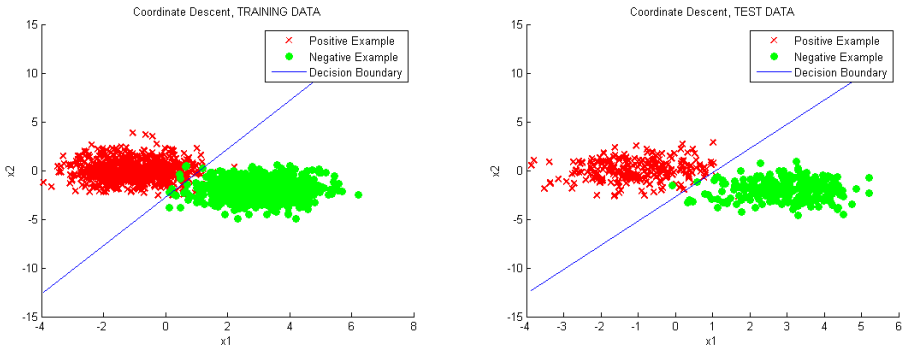

Figure 5.6: Decision boundary after training with CD_basicGD on Training set (left) and Test set (right).

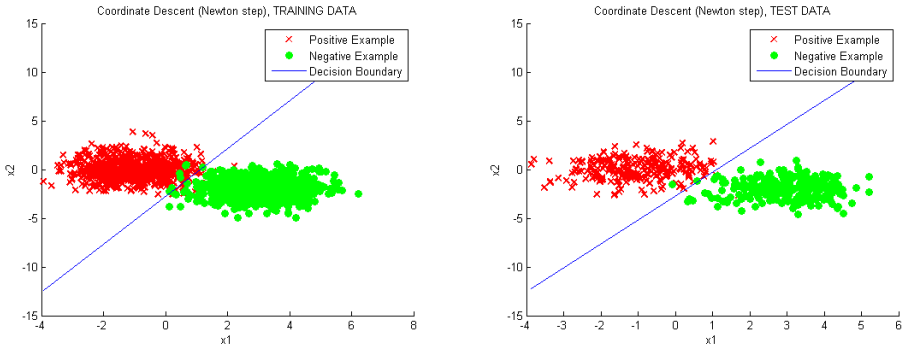

Figure 5.7: Decision boundary after training with CD_basicNewton on Training set (left) and Test set (right).

5.3 to 5.9 and Table 5.1, we get decent results by all methods tested, which is a strong indication that they are working properly.

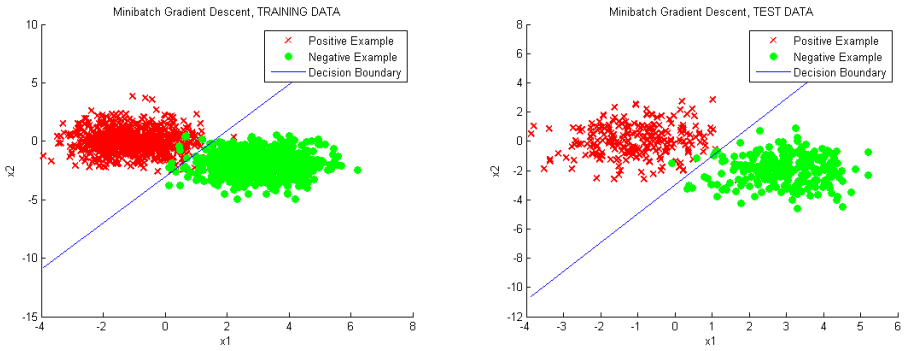

Figure 5.8: Decision boundary after training with MBGD_basic on Training set (left) and Test set (right).

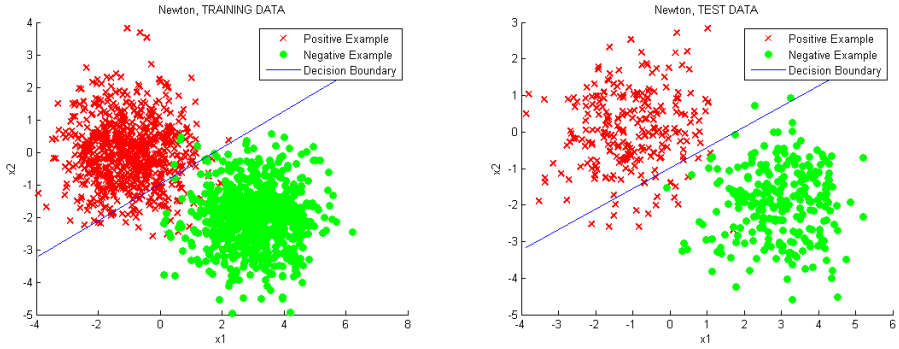

Figure 5.9: Decision boundary after training with Newton's method on Training set (left) and Test set (right). Note that this boundary was achieved after only 1 iteration while the results for all other methods are after 300 iterations.

Table 5.1: Logistic Regression Results

Method	Value of Objective Function on Training Set	Value of Objective Function on Test Set	
Before Optimization	2.006824e+000	2.027101e+000	
BGD_basic	7.808466e-002	6.683343e-002	300
BGD_heavyball	7.684857e-002	6.567788e-002	300
SGD_basic	4.716319e-002	3.426209e-002	300
CD_basicGD	7.828068e-002	6.699035e-002	300
CD_basicNewton	4.744886e-002	3.623695e-002	300
MBGD_basic	4.643823e-002	3.741856e-002	300
Newton	0.2699	0.2662	1

5.3 Simple Baselines for Density Estimation

Until now we discussed the case where the gradients of the parameters with respect to the objective function were miscalculated. How can we detect if we have the objective function right? After all, the finite differences method just verifies

that the objective and the gradient codes "are consistent" with one another. What if we got the objective wrong and consequently wrote correct gradient code for the wrong objective function? Here the answer is somewhat more problem-specific. We have to take into consideration the physical meaning of the objective function: What range can its values assume? What value does it get given particular extreme cases of inputs?

Since we use NADE to perform density estimation we need to make sure that the values it computes for $ln(p(\boldsymbol{x_i})), i = 1, 2, ..., K$, where $K = 2^D$ and $\boldsymbol{x_i} \in \{0, 1\}^D$ assumes all possible values in $\{0, 1\}^D$, corresponded to natural logarithms of probability densities $p(\boldsymbol{x_i}), i = 1, 2, ..., K$. So we verified that indeed $p(\boldsymbol{x_i}) = e^{ln(p(\boldsymbol{x_i}))} \in [0, 1], i = 1, 2, ..., K$ and $\sum_{i=1}^{K} p(\boldsymbol{x_i}) = \sum_{i=1}^{K} e^{ln(p(\boldsymbol{x_i}))} = 1$. Thus $p(\boldsymbol{x_i}), i = 1, 2, ..., K$ are indeed probability densities. But are they "good" ones?

To answer that, at least preliminarily, we can compare NADE to some form of elementary density estimation as a baseline, comparing the average negative log likelihood of the models. The simplest baseline would be beating a "random density estimator". So we assume every bit of instance $x^{(t)}$ can be 0 or 1 with equal probability (i.e. $p(x_d^{(t)} = 0) = p(x_d{}^{(t)} = 1) = 0.5, \forall d$). Under this simplistic assumption the ANLL is

$$J(\boldsymbol{X}; \boldsymbol{\theta}) = \frac{1}{T} \sum_{t=1}^{T} \sum_{d=1}^{D} -ln(p(x_d^{(t)})) = \frac{1}{T} \sum_{t=1}^{T} -\ln(0.5^D) = -\frac{1}{T} T \ln(0.5^D) = -D \ln(0.5) \tag{5.9}$$

We present the values obtained under this baseline for every dataset in Table 5.2 . Usually, our parameter initialization led to average negative log likelihood values in this region, so it is a good indication of the value of our objective function before training. Of course, after training we always beat this unrefined "density estimator".

Yet another model for density estimation we can use as a baseline is to calculate $p(x_d = 0)$ and $p(x_d = 1)$ for each d based on the count of occurrences of bit 1 (c_1) and 0 (c_0) in all T instances. This model can be viewed as a form of Naive Bayes approach for Bernoulli variables. We used pseodocounts instead of actual counts of occurrences so as to avoid the problem of zero probabilities in the case of zero occurrences

$$p(x_d^{(t)} = 0) = \frac{c_0 + a}{T + a + b}, \forall t, \tag{5.10}$$

where $T = c_0 + c_1$ and of course $p(x_d^{(t)} = 1) = 1 - p(x_d^{(t)} = 0), \forall t$.

We can adjust parameters $\alpha \in \mathbb{R}^+$ (this has nothing to do with the learning rate we use in the optimization methods, we just used the same symbol) and $\beta \in \mathbb{R}^+$ to increase or decrease the pseudocounts of each class (0 and 1 respectively), in

essence adjusting the prior probability of each. We chose $\alpha = \beta = 1$, which corresponds to Laplace Smoothing and treats the two classes as equiprobable.

Having done that, we can compute the average negative loglikelihood using these probabilities

$$J(\boldsymbol{X};\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}\sum_{d=1}^{D} -ln(p(x_d^{(t)})) = \sum_{d=1}^{D} -ln(p(x_d^{(t)})) \tag{5.11}$$

In Table 5.2 we can see the ANLL computed under this model for each of the datasets we will train NADE on (the datasets themselves are properly introduced in section 6.1). As we will see, in section 6.3, we also beat this baseline as well. We also calculated the percentage of 1's in each dataset. We will use it here to compare the two baselines, but it can also be thought of as a measure of density of the dataset, with low values corresponding to sparse datasets.

Table 5.2: Baseline values of Average Negative Loglikelihood (ANLL) of simple density estimation techniques. The last column is the percentage of 1's in each dataset. All refer to the corresponding test set.

Dataset	ANLL Random	ANLL Based on Pseudocounts	% of 1's in Dataset
adult	84.5640	25.7602	11.37
mnist	542.7342	205.6319	13.27
connect-4	87.3365	35.3312	33.33
dna	124.7665	100.2841	25.14
mushrooms	77.6325	34.2084	18.75
nips-0-12	346.5736	293.9804	36.65
ocr-letters	88.7228	63.7293	21.87
rcv1	103.9721	58.3684	13.88
web	207.9442	39.7515	3.88

As we see in Figure 5.10. The farther away from 50% the percentage of 1s in the dataset is, the less correct the assumption $p(x_d^{(t)} = 0) = p(x_d{}^{(t)} = 1) = 0.5$, $\forall d$ is so the largest the relative difference $\frac{ANLL_{BASELINE_1} - ANLL_{BASELINE_2}}{ANLL_{BASELINE_1}}$ among these two baselines becomes. This was to be expected, of course.

Having done these checks we are now confident that our code works as it is supposed to and we can move on to applying our methods to training NADE.

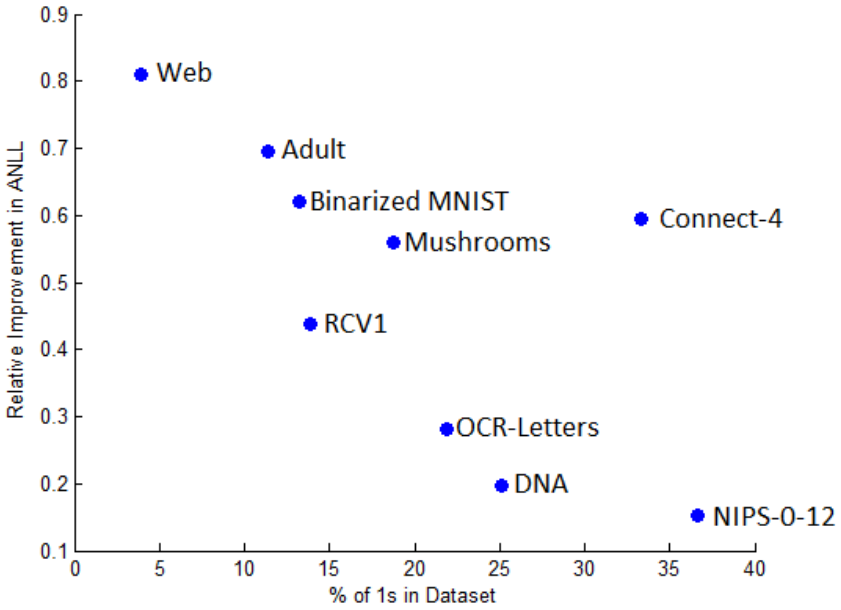

Figure 5.10: Relative difference of the two baselines presented in section 5.3 as a function of the number of 1's in the dataset.

Chapter 6

Experimental Results

6.1 The Datasets

The datasets used in the experiments are all from the LIBSVM database [1] except for *ocr-letter* [2] and *nips-0-12* [3]. They are all versions that have undergone preprocessing (not performed by us, these are versions already available in the sources mentioned above). Features with missing values have been removed and all features,whether discrete or continuous, whether numerical, categorical or ordinal have been converted into an appropriate binary form. The datasets are split in 3 subsets each: Training, Validation and Test. The training set was used during training to optimize the parameters of the model. The validation set was used in hyperparameter selection and in overfitting control by early stopping. The test set was the one used for the final evaluation of this method (the one reported on the tables that follow). In Table 6.1 we can see the size and number of features of each dataset. We already saw that there is a different degree of sparsity in each dataset in Table 5.2.

The datasets we used were the same ones used in [Larochelle et al., 2011]. This way we can verify that we get similar results to those reported by the authors for NADE with H=500 using SGD_basic and compare them to the other baselines presented there. As we see in Table 6.3 we indeed get similar results, yet not exactly the same due to the following reasons:

- The authors in [Larochelle et al., 2011] use a slightly different learning rate decrease scheme than the one used here.
- They use a lookahead of 10 iterations for early stopping. We use 5 instead.
- It is not entirely clear how they performed the hyperparameter selection (e.g. did they use the entire validation set, did they train to the entire training set, did they train until convergence, did they use early stopping

[1] Datasets available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.
[2] Available at `http://ai.stanford.edu/~btaskar/ocr/`.
[3] Available at `http://www.cs.toronto.edu/~roweis/data.html`.

Table 6.1: The datasets we used for our experiments. T is the number of instances, D the number of features.

Dataset Name	Dimensions ($T \times D$) Training	Validation	Test	Brief Description
adult	5000×123	1414×123	26147×123	Census data, consisting of 14 features. (Ver. a5a).
binarized mnist	50000×784	10000×784	10000×784	28×28 images of handwritten digits $0, 1, ..., 9$
connect-4	16000×126	4000×126	47557×126	All legal positions in a game of connect-4 and the player that controls each.
dna	1400×180	600×180	1186×180	Primate splice-junction gene sequences (DNA)
mushrooms	2000×112	500×112	5624×112	*mushrooms* described in terms of 21 physical characteristics.
nips-0-12	1240×500	500×500	400×500	Paper information of NIPS proceedings from volume 0 to 12.
ocr-letters	32152×128	10000×128	10000×128	16×8 images of handwritten English characters.
rcv1	40000×150	10000×150	150000×150	Reuters Corpus Volume I. Contains manually categorized newswire stories.
web	14000×300	3188×300	32561×300	A subset of the ICML-09 URL reputation. dataset. (Ver. w6a).

and if yes was it use the same lookahead as in the final execution of the algorithm or not). Most likely we have made some differences in this procedure, since it involves many decisions. Here, we are exploring a larger range of hyperparameters (5 options per hyperparameter compared to 3 of the authors), but most likely in more "hurried" way (i.e we run for only a few iterations with stricter early stopping and training on only a subset of the training set).

- The number of runs we did to obtain the statistics might differ, we used $N_{runs} = 5$.
- The inherent stochasticity of the entire procedure. In Table 6.2 we can see the sources of variance involved in each method group. In the case of comparing our results to those of [Larochelle et al., 2011] the ones for SGD apply.

Table 6.2: Sources of variance for each method group examined for NADE. Even if we use the same method with the same hyperparameters, these factors will introduce variation among the results of different runs.

Source of Variance	Method Groups				
	BGD L-BFGS	CD	SGD	MBGD	DIBGD
Random initialization of parameters θ_i. (Note: NADE's ALL is non-convex)	√	√	√	√	√
Ordering of features d_i. (Note: NADE computes conditionals $p(x_{d_i}\|x_{<d_i})$)	√	√	√	√	√
Ordering of parameters θ_i		√			
Ordering of instances t_i			√		
Ordering of minibatches b_i				√	
Instances within each minibatch (MBGD) or within initial batch (DIBG)				√	√

6.2 Experimental Design

The final version of our code needs only three inputs specified: the dataset we are working with, the optimization method we will use and the number of hidden variables H. It returns the iterations spent for parameter selection, the iterations needed for the actual execution of the optimization procedure and the Average Log Likelihood (ALL) of the trained NADE model on the Test set.

We initialized the parameters of the model using random values ${\theta_i}^{(0)} \in U(-1, 1)$, $\forall i = 1, 2, ..., N$). We used a random permutation of the columns of the feature vector thus shuffling the order of features. The ordering of the features affects the result of NADE but not of models such as FVSBN. However the authors in [Larochelle et al. , 2011] reported that the effect of the ordering on the ALL is not statistically significant. In addition to that, stochastic and minibatch methods shuffle instances in the beginning of their execution (exceptions MBGD_basic, MBGD_momentum, which shuffle them in the beginning of every epoch) and DIBG randomly selects a small subset of data for the initial batch.

In most cases we executed $N_{runs} = 5$ runs for each method on each dataset (for each choice of $H \in \{5, 50, 500\}$) in order to obtain statistics. We present the average of these runs as well as 95% confidence intervals in the form $\mu \pm \frac{\sigma}{\sqrt{N_{runs}}}$. Each of the N_{runs} runs for all methods for the same dataset and number of hidden variables H was performed using the same parameter initialization and shuffling of features. In methods added in the last stages of this dissertation using the same initialization and shuffling was impossible, since we kept no record of it. Also, in a few cases $N_{runs} < 5$. Finally, for some methods, we only present results on two datasets. These methods were deemed non-competitive so we did not test them on the remaining datasets. For example DIBGD_basic and CD_basicGD gave poor ALL results (we will see possible explanations as to why this happens). MBGD_bolddriver and IAGD_basic appear not to improve upon the basic versions they are supposed to (MBGD_basic amd SGD_basic, respsctively), so we exclude

them from future experiments.

We ran experiments on 3 different machines. In order to compare the value of the objective function after optimization, we present the average (over all runs) value of ALL attained after training NADE as well as 95% confidence intervals. To compare the execution time of each method, we count the number of iterations. In order to get a sense of how fast a method is, we should also factor in the (average) time of 1 iteration for each method, since it changes dramatically from one method to another. We computed the average time of a single iteration for all methods on one of the machines, so as to allow direct comparison among methods. The ALL results were rounded-off to 2 decimal digits and the averages were normalized by subtracting them by the average ALL of the Mixture of Bernoullis (MoB model), shown in 6.4. This was done to keep the same format as in [Larochelle et al., 2011] and facilitate the comparison of the results. The iteration results were rounded to the nearest integer.

6.3 Results: Average Loglikelihood (ALL)

In the tables that follow we present the ALL results attained, as well as baseline results from other models. In this subsection we only compare the methods in terms of the value of the objective function. Therefore when we refer to the "performance" of a method we mean "quality of optima it finds".

Table 6.3: Average Log Likelihood (ALL) of NADE with $H = 500$ hidden units trained by SGD. Results from our work and from [Larochelle et al., 2011] for verification. Averages were normalized by subtracting them by the average ALL of the Mixture of Bernoullis (MoB model), shown in 6.4. Higher values correspond to better density estimation.

NADE Results by	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
Larochelle & Murray	7.25 ±0.05	11.42 ±0.01	13.38 ±0.57	4.65 ±0.04	16.94 ±1.11	13.34 ±0.21	0.93 ±0.11	1.77 ±0.20	48.78 –
Present work	7.24 ±0.04	11.42 ±0.01	13.23 ±0.40	4.65 ±0.04	16.26 ±1.20	13.08 ±0.37	0.92 ±0.09	1.64 ±0.18	42.39* –*
Normalization	−20.44	−23.41	−98.19	−14.46	−290.02	−40.56	−47.59	−30.16	−137.64

* Results for $N_{runs} = 1$ and with $\alpha = 2$ throughout the execution (no hyperparameter optimization applied).

In [Larochelle et al., 2011], NADE's performance (in terms of mean ALL value) is compared to that of some other models. We present here their results as well to include them in our comparisons with the results obtained by training NADE using the methods discussed here. These additional models are the following and their mean ALL, as well as 95% confidence intervals, are shown in Figure 6.4 :

- **MoB:** a mixture of multivariate Bernoullis.
- **RBM:** a RBM (briefly described in Chapter 1) made tractable by using only 23 hidden units and trained by contrastive divergence with up to 25 steps of Gibbs sampling.

- **RBM mult:** a RBM where units have been split into groups and within each group only one unit can be active (equal to 1) at any time. This model was proposed in [Larochelle et al. , 2010].
- **RBForest:** an RBM where the activation of hidden units within a group obeys tree constraints. This model was proposed in [Frey, 1998].
- **FVSBN:** A Fully Visible Sigmoid Belief Network, as described in Chapter 1.

Table 6.4: Average Log Likelihood (ALL) for other models in [Larochelle et al. , 2011]. These results were not reproduced by us, but we will use them in our comparisons. Averages were normalized by subtracting them by the average ALL of the Mixture of Bernoullis (MoB model), shown in the first row. Higher values correspond to better density estimation.

Model	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
MOB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	±0.10	±0.04	±0.53	±0.10	±1.12	±0.32	±0.11	±0.23	–
RBM	4.18	0.75	1.29	−0.69	12.65	−2.49	−1.29	0.78	–
	±0.06	±0.02	±0.48	±0.09	±1.07	±0.30	±0.11	±0.20	51.3
RBM mult.	4.15	−1.72	1.45	−0.69	11.25	0.99	−0.04	0.02	–
	±0.06	±0.03	±0.40	±0.05	±1.06	±0.29	±0.11	±0.21	–
RBForest	4.12	0.59	1.39	0.04	12.61	3.78	0.56	−0.15	–
	±0.06	±0.02	±0.49	±0.07	±1.07	±0.28	±0.11	±0.21	–
FVSBN	7.27	11.02	14.55	4.19	13.14	1.26	−2.24	0.81	40.19
	±0.04	±0.01	±0.50	±0.05	±0.98	±0.23	±0.11	±0.20	–
Normalization	−20.44	−23.41	−98.19	−14.46	−290.02	−40.56	−47.59	−30.16	−137.64

We will describe the main findings regarding the average ALL values found in words. The comments there, unless stated otherwise refer to Tables 6.5 and 6.6:

- **Low-order batch methods vs the rest:** As expected, batch methods below second order (BGD variants, DIBG and CD) give optima of lesser quality than those found by L-BFGS, stochastic and minibatch methods. First and lower order batch methods do not use the curvature of the objective function, nor do they gain from the stochasticity of stochastic and minibatch methods to escape local optima. There are, however some datasets that appear to be "easy" (*adult*, *mushrooms*, *connect-4*) to train on. In these cases, the relative difference in ALL between BGD variants and other methods is considerably smaller. In fact for *adult*, the result for BGD and MBGD even overlap. In some "difficult" datasets (binarized mnist, *ocr-letters*, *connect-4*) we can also observe some overlapping between the resulting ALL of batch and minibatch methods.
- **Coordinate descent:** In these runs we used the CD_basicGD version as we only implemented CD_basicNewton in the last stages of the dissertation. As a result the coordinate descent version here was slow. It terminated in most cases due to MAX_ITERS being reached. So it performs worse than all other methods, especially for problems with many parameters (i.e. large H and/or D). An indication for this, is that the relative performance of

Table 6.5: Average Log Likelihood (ALL) for each optimization method training NADE with $H = 5$ hidden units. Averages were normalized by subtracting them by the average ALL of the Mixture of Bernoullis (MoB model), shown in 6.4. Higher values correspond to better density estimation.

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic	5.90	5.92	1.23	1.91	2.27	1.24	0.31	0.19	20.31*
	±0.18	±0.03	±0.46	±0.08	±1.59	±0.58	±0.08	±0.29	±5.12*
BGD_heavyball	6.13	6.18	1.27	2.04	2.31	1.30	0.29	0.19	26.33*
	±0.17	±0.03	±0.45	±0.08	±1.47	±0.60	±0.07	±0.31	±4.73*
BGD_bolddriver	6.16	6.13	1.28	1.98	2.30	1.24	0.28	0.21	24.29*
	±0.23	±0.05	±0.46	±0.08	±1.49	±0.62	±0.08	±0.33	±3.45*
BGD_rprop	6.22	6.14	1.20	1.94	2.30	1.37	0.32	0.21	25.25*
	±0.21	0.05	±0.44	±0.07	±1.63	±0.55	±0.08	±0.30	±4.87*
CD_basicGD	3.93	2.99	–	–	–	–	–	–	–
	±0.24	±0.3	–	–	–	–	–	–	–
L-BFGS	7.12	7.52	5.93	3.24	5.38	7.72	0.61	0.72	31.49*
	±0.11	±0.02	±0.51	±0.06	±1.66	±0.06	±0.29	±0.29	±4.29*
DIBGD_basic	5.09	5.11	–	–	–	–	–	–	–
	±0.26	±0.3	–	–	–	–	–	–	–
SGD_basic	7.08	7.45	5.96	3.22	5.21	7.58	0.57	0.67	30.20*
	±0.06	±0.01	±0.40	±0.06	±1.60	±0.57	±0.07	±0.27	±3.53*
SGD_heavyball	7.08	7.57	6.02	3.20	5.33	7.58	0.60	0.70	29.34*
	±0.05	±0.02	±0.38	±0.06	±1.62	±0.54	±0.05	±0.25	±3.37*
ASGD_basic	7.10	7.59	6.02	3.23	5.36	7.61	0.59	0.72	30.06*
	±0.05	±0.02	±0.33	±0.06	±1.25	±0.58	±0.05	±0.22	±3.40*
IAGD_basic	7.02	7.42	–	–	–	–	–	–	–
	±0.08	±0.03	–	–	–	–	–	–	–
MBGD_basic	6.22	7.01	3.92	1.99	3.84	2.05	0.34	0.55	24.64*
	±0.13	±0.04	±0.37	±0.05	±1.54	±0.59	±0.08	±0.27	±3.81*
MBGD_heavyball	6.27	7.07	3.99	2.03	3.93	2.04	0.33	0.59	25.15*
	±0.14	±0.03	±0.47	±0.08	±1.69	±0.61	±0.08	±0.24	±2.55*
MBGD_bolddriver	6.14	6.88	–	–	–	–	–	–	–
	±0.23	±0.06	–	–	–	–	–	–	–
Normalization	−20.44	−23.41	−98.19	−14.46	−290.02	−40.56	−47.59	−30.16	−137.64

* Results for $N_{runs} = 2$.

CD_basicGD compared to all other methods is lower in the case of *connect-4* than in *adult* and —more convincingly— from the fact that coordinate descent gives worse ALL results for H=50 than for H=5. This result is not normal and it happens only with this method. It is a result of the method not getting enough time to converge. Again, this is perhaps mainly a weakness of the implementation we used and we mention it mainly to explain the results. A different implementation of Coordinate Descent might have been on par with other methods. However, the large parameter space suggests that Coordinate Descent might be a poor choice in these problems in any form that doesn't update parameters in blocks.

- **Dynamically increasing batch Gradient Descent:** Contrary to what we hoped, DIBGD_basic is the second-worst performing (in terms of ALL attained) method. It even performs considerably worse than BGD_basic, the method to which it is supposed to be equivalent asymptotically. An explanation for this is that DIBGD overfits on the initial batch, as we did not perform any sort of regularization. Trapped in a local minimum, the weights grow dramatically and subsequent increases of the minibatch could not counter this effect.

Table 6.6: Average Log Likelihood (ALL) for each optimization method training NADE with $H = 50$ hidden units. Averages were normalized by subtracting them by the average ALL of the Mixture of Bernoullis (MoB model), shown in 6.4. Higher values correspond to better density estimation.

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic	6.53	6.12	4.24	2.84	4.79	4.56	0.36	1.35	27.48*
	±0.07	±0.03	±0.46	±0.05	±1.26	±0.45	±0.19	±0.22	±5.13*
BGD_heavyball	6.78	6.41	4.76	2.93	4.77	4.70	0.34	1.41	26.82*
	±0.07	±0.03	±0.48	±0.05	±1.25	±0.48	±0.18	±0.24	±4.22*
BGD_bolddriver	6.89	6.32	4.72	2.98	4.89	4.50	0.42	1.45	24.46*
	±0.08	±0.04	±0.53	±0.06	±1.28	±0.48	±0.22	±0.26	±6.46*
BGD_rprop	6.74	6.47	4.71	3.06	4.79	4.56	0.33	1.45	28.17*
	±0.09	±0.05	±0.55	±0.06	±1.32	±0.52	±0.21	±0.26	±5.57*
CD_basicGD	2.17	1.41	–	–	–	–	–	–	–
	±0.10	±0.07	–	–	–	–	–	–	–
L-BFGS	7.22	9.15	9.13	4.15	10.79	10.98	0.62	1.79	36.27*
	±0.07	±0.09	±0.46	±0.03	±1.52	±0.56	±0.18	±0.16	±5.27*
DIBGD_basic	4.91	4.16	–	–	–	–	–	–	–
	±0.11	±0.07	–	–	–	–	–	–	–
SGD_basic	7.17	9.18	9.04	4.06	10.60	9.97	0.70	1.75	41.11*
	±0.02	±0.02	±0.30	±0.03	±1.14	±0.32	±0.09	0.09	±3.17*
SGD_heavyball	7.19	9.20	9.08	4.07	10.64	10.02	0.73	1.75	40.07*
	±0.02	±0.02	±0.27	±0.03	±1.17	±0.37	±0.09	±0.11	±3.25*
ASGD_basic	7.20	9.21	9.10	4.07	10.69	10.06	0.73	1.76	46.62*
	±0.02	±0.02	±0.33	±0.03	±1.14	±0.33	±0.10	±0.09	±3.40*
IAGD_basic	7.17	9.07	–	–	–	–	–	–	–
	±0.05	±0.04	–	–	–	–	–	–	–
MBGD_basic	6.79	7.18	6.44	3.56	7.50	6.74	0.46	1.60	32.11*
	±0.05	±0.03	±0.39	±0.04	±1.23	±0.39	±0.14	±0.16	±4.83*
MBGD_heavyball	6.83	7.43	6.49	3.59	7.72	6.78	0.45	1.63	34.10*
	±0.07	±0.02	±0.33	±0.02	±1.26	±0.36	±0.17	±0.15	±3.99*
MBGD_bolddriver	6.17	7.13	–	–	–	–	–	–	–
	±0.09	±0.04	–	–	–	–	–	–	–
Normalization	−20.44	−23.41	−98.19	−14.46	−290.02	−40.56	−47.59	−30.16	−137.64

* Results for $N_{runs} = 2$.

- **Minibatch methods vs the rest:** Minibatch methods achieve in all cases better results than lower order batch methods and worse than stochastic methods and L-BFGS. Thus, it appears that in this non-convex setting, the stochasticity is an important contributor to finding good local optima. A way to showcase this hypothesis more convincingly would have been to plot the ALL value as a function of the batchsize, ranging from 1 (MBGD becomes SGD) to T (MBGD becomes BGD) of MBGD methods for a given dataset. Due to time constraints we did not include that in this study, but it might be a good suggestion for further work. To the defense of these methods (but not of our methodology), the way we used for setting the minibatch size was somewhat naive and restrictive. We used 2%, 4%, 5%, 8%, 10% of the dataset in each batch, but did not factor in other parameters such as the redundancy of the data which would give an advantage to the method. For big datasets even 2% of the dataset might correspond to a large-sized batch, therefore MBGD looses due to averaging any big advantage over batch methods. The decrease of the relative improvement of MBGD over BGD can be observed, to some extent, on the three largest datasets (*binarized mnist*, *rcv1*, *ocr-letters*), where for the two actually overlap.

- **Stochastic methods vs the rest:** Stochastic methods give the best results among all lower order methods examined. They also demonstrate a consistency in these results in that the variance of ALL values reached is smaller than in any other group of methods. We did expect stochastic methods to reach better optima due to the non-convexity of the objective. The ALL of the stochastic methods is in most cases slightly lower than that of L-BFGS, with a few cases even matching or exceeding that.

- **L-BFGS vs the rest:** In terms of ALL results, training NADE with L-BFGS appears to outshine any other method. However, there is a caveat: the variance of the ALL of NADE models trained by L-BFGS is generally higher than that of stochastic methods (though not as high as in lower order batch methods). The reason for this difference in variance is that L-BFGS —at least in our case— is more prone to overfitting. In other words, L-BFGS does indeed (almost) always find better local optima of the objective function than SGD when training (i.e. on the training set), but this can lead it to overfit on the training set and fail to generalize on the test set. In fact, had we shown[4] the results for ALL on both training and test set for L-BFGS and SGD, we would have seen a much larger difference on the training set in terms of ALL of the solution found. This all happens because we took no measures to protect L-BFGS against overfitting. On the other hand, SGD uses early stopping.

- **Effect of heuristics (generally):** The simple versions of each method were in most cases outperformed by versions based on the various heuristics. The "Heavy Ball" version nearly always improves upon the basic version and "Bold Driver" and "R-prop" in BGD improve ALL significantly over that of the basic version in many cases. Exceptions to this rule are MBGD_bolddriver and IAGD_basic. Interestingly, the improvement that the heuristics offer over the basic version is far more pronounced in batch methods. To the other extreme, SGD seems little affected by the heuristic modifications we applied to it, although the effect is consistently positive.

- **Effect of heuristics ("Heavy Ball"):** In nearly every case, a method that uses the "Heavy Ball" heuristic achieves better performance than the basic version. In batch methods this happens because the "memory of the past updates " can help the method avoid bad local optima (certainly not to the extend that noise in SGD does, though) by "rolling over" them, to keep up with the "ball" parallelism. On the other hand in stochastic methods the "Heavy Ball" heuristic diminishes the stochasticity of the updates. In minibatch methods it can potentially do either of these two (i.e. jump over local minima and do a sort of weighted averaging over past updates) depending on the minibatch size.

- **Effect of heuristics ("Bold Driver" and "R-prop"):** Although both "Bold Driver" and "Resilient Backpropagation" generally improve over sim-

[4]Records were not retained to be included in this discussion, but we observed that during our experiments.

ple BGD, they appear to be producing results of higher variance. Since they are using more hyperparameters than BGD to produce adaptive learning rate (and in case of R-prop, individual learning rate for each parameter) the increase in variance is to be expected. MBGD_bolddriver on the other hand appears more unstable, exhibiting high variance in its results and consistently performing below MBGD_basic. An explanation for MBGD_bolddriver's poor performance is the fact that it adjusts the learning rate on each minibatch, therefore it is likely for subsequent increases and especially subsequent decreases, to produce extreme (too high or to low) values for the update. As a result, the algorithm will converge prematurely in either case. Perhaps our decision to fix hyperparameters ρ and σ to particular values also contributed to the bad performance, however this action was necessary at the time as we discussed in Chapter 4.

- **Effect of heuristics (Averaged Stochastic Gradient Descent):** ASGD seems to be the most successful variant of SGD. The slight suppression of its stochasticity seems to benefit SGD and ASGD does this more aggressively than SGD_heavyball. Note at this point that "Heavy Ball" in SGD can be viewed as computing the next update as a weighted average of the updates from previous time steps (plus the new step) with exponentially decaying weights (the further back in time a gradient is, the smaller its contribution). ASGD averages all past gradients and uses the average to compute the update. The price of the decrease in stochasticity seems to be a slight increase in variance of the resulting ALL.

- **Effect of heuristics (Iterate Averaging Gradient Descent):** IAGD is based on the same principle as SGD using the "Heavy Ball" heuristic and ASGD, i.e. "lowering the stochasticity" of the SGD updates. However, although ASGD_basic and SGD_heavyball improve upon SGD_basic, this is not the case with IAGD_basic. This seems strange at first. However, one possible explanation is that, as IAGD averages over previous updates, it also factors in the decrease in the stepsize that occurs over time. As a result, older gradients are given a greater weight than newer ones since $\alpha(n-i) > \alpha(n)$, for all steps n, i. And older gradients generally correspond to points farther away from the optimum than newer ones. On the other hand in SGD_heavyball, the farther back a gradient is the smaller (and exponentially so) its contribution to the new update will be.

- **Effect of the number of hidden units H:** The largest H is the greater the resulting ALL for every individual method (we saw that this does not hold for CD_basicGD and gave an explanation as to why). This is not surprising, since NADE is more flexible when using more features and it can better encode the information of the original data, thus produce better density estimates.

- **Effect of the number of features D:** Generally, the higher D is, the larger H we need to capture the information that lies within the dataset. This is apparent from the fact that on the datasets with the largest num-

ber of features (*binarized mnist*, *nips-0-12*, *web*, *dna*), NADE using fewer hidden units performs considerably worse. On the other hand, on datasets with a small number of features (*adult*, *mushrooms*, *connect-4*) NADE's performance seems little affected by H. Of course D is not the sole factor that determines the number of hidden units needed to encode the dataset with little loss. Another factor is the amount of redundancy (meaning inter-feature dependencies in this case) of the data. As we saw above, *adult* and *mushrooms* seem to be the easiest of the datasets, a fact that is apparent from the small variance of the results obtained, not only per method of training, but per model (as apparent in the results shown in Table 6.4). Apparently the resulting ALL also exhibits small variance with respect to the choice of H, which might imply a high amount of redundancy in these datasets.

- **Effect of the number of instances T:** The only observation that we can make regarding the effect of the size of the dataset on the ALL results is that the advantage of mini-batch methods over batch methods diminishes as T increases. In theory the contrary should happen, due to the stochasticity of the MBGD updates allowing it to escape local optima. This inconsistency in our case is most likely caused by the way we choose the batchsize (for large datasets even 2% of them is still a large dataset).
- **Comparison of NADE's results to those of other models:** If we compare our results with those on 6.4 we will notice that in many cases (e.g. in *adult*) NADE outperforms most of the baseline models there, even trained by a lower order batch method, and even when using only as few as $H = 5$ hidden units. The only exception is FVSBN which generally achieves a performance comparable to that of NADE trained with $H = 500$ hidden units using SGD (and presumably L-BFGS, although we did not train a NADE with $H = 500$ hidden units using L-BFGS in these experiments).
- **Comments on variance:** The variance on ALL attained seems to be greater for batch methods than for minibatch methods and greater for mini-batch methods than for stochastic methods. This gives rise to an interesting observation: The more noisy the updates of an optimization method are, the more robust the method is in a non-convex setting. This appears counter-intuitive at first but is based on the fact that by using noisy updates we can escape poor local optima, thus the possible outcomes in terms of ALL become fewer.

6.4 Results: Execution Times

Now we will present the average number of iterations needed for each method to converge and the number of iterations spent on hyperparameter optimization for each of the datasets. These numbers are not to be directly compared as the computational cost per iteration differs dramatically across methods and across

datasets. It is interesting, however, to compare in each case the number of iterations spent on hyperparameter optimization and on optimization itself. It gives us a nice picture of what price we pay computationally for adding extra hyperparameters.

Below we give results only for the runs for $H = 50$. For $H = 5$ and $H = 500$ the parameter space of the objective function becomes significantly smaller and bigger, respectively. As a result, the number of iterations a method will need until it converges will be different. It is therefore not safe to make any assumptions about the number of iterations needed for $H = 5$ and $H = 500$ based on the results below for $H = 50$. That said, there are some observations we can make regarding the number of iterations which are likely to hold even for different values of H:

- **Observations about methods:** Notice that CD_basicGD always uses the full number of iterations (i.e. terminates due to reaching MAX_ITERS without converging) in its optimization run. This explains its bad performance and we already attributed it to our poor choice for the line search (BGD step).

  BGD_bolddriver and BGD_rprop take fewer iterations than BGD_basic to terminate. They mainly do so due to the learning rate(s) assuming an extreme value. This effect on MBGD_bolddriver might explain its worse average ALL compared to MBGD_basic.

  "Heavy Ball" methods converge in fewer iterations than their "basic" counterparts. The parameter space is large, the objective function non-convex so differences in curvature along different directions are to be expected. The result is the "zig-zagging problem" we discussed in Chapter 3 and the "Heavy Ball" heuristic helps counter that and speed up convergence.

  ASGD_basic and IAGD_basic also converge in fewer iterations than SGD_basic. They require the same numbers of hyperparameters as SGD_basic, contrary to SGD_heavyball that also has the momentum parameter.

  Finally, to state the obvious, L-BFGS requires the least number of iterations to converge, and it requires no hyperparameter optimization which is a huge advantage over the other methods examined. Of course, BGD methods take more iterations than L-BFGS but fewer than MBGD and SGD and SGD methods require the most iterations. Still this means nothing without factoring in the individual costs per iteration.

- **Observations about datasets:** The datasets that need the fewest iterations are *adult*, *mushrooms*, *dna* and *nips-0-12* the first two are apparently "easy" to train on, as a result the optimizers find a good local optimum quickly. The other two appear to be among the most difficult datasets. Here we have fast termination due to getting stuck quickly in —not necessarily good— local optima and early stopping triggering.

  Also, we can notice that in "easy" datasets, the hyperparameter optimiza-

tion procedure takes more time. This happens because the outcome of the optimization in these datasets is less sensitive to the hyperparameters than in more difficult ones. As a result, more hyperparameter configurations are executed all (or at least closer to) 20 times (the MAX_ITERS or MAX_EPOCHS value for hyperparameter optimization) without early stopping being triggered. For example compare the iterations spent in hyperparameter selection for *adult* and *mushrooms* ("easy") to those of *dna* and *nips-0-12* ("difficult") in Table 6.7. This, combined with the fact that the optimization itself requires fewer iterations increases the gap in iterations between hyperparameter selection and the optimization process. See, for instance the extreme cases of BGD_bolddriver and BGD_rprop for *mushrooms*, where the optimizer needs an overhead of about 20× the iterations it will spent in optimization just to set its hyperparameters. A lesson from this observation is that in such "easy" cases, we should not dedicate a large part of our resources in hyperparameter optimization, since (i) a large range of configurations will offer reasonably good results, (ii) hyperparameter optimization will take more iterations than in more "difficult" datasets, since it will quickly reject hyperparameter combinations less often.

Now we present the average time per iteration for each method group on each dataset. In this case, getting a rough estimate of the corresponding average time per iteration for the case of $H = 5$ and $H = 500$ is simpler. Since the cost of training NADE is $O(HD)$ we simply need to multiply the results below by 0.1 and 10 respectively. Again, this is an idealized projection that disregards constant computational overheads, different for each method and subject to implementation details.

Finally, multiplying the number of iterations from Table 6.7 by the corresponding time from Table 6.8 we can get the total time the execution of each method takes on average. The results are shown in Table of Appendix C, where again we show times for hyperparameter selection (top row), times for optimization (middle row) and total times (bottom row). In Table we show only the total times normalized (all times divided by $19021.1438 = maximum \times 10^{-1}$) to enhance readability.

Again, we can see that LBFGS

6.5 Results: Closing Remarks

As expected, L-BFGS and SGD and its variants outperform all other methods in terms of quality of optima they find. L-BFGS gives us the best ALL results its iterations are the slowest among the methods we examined here, but it has the benefit that it does not require fine tuning of hyperparameters such as the step size. This and the fact that it requires fewer iterations than any other method also makes it the fastest among the methods examined. Its main drawback is that it is more prone to overfitting than SGD. Evidence for this is its greater variance in average ALL attained and its larger "gap" between the value of the

Table 6.7: Average number of iterations until termination for each optimization method training NADE with $H = 50$ hidden units. First row shows average number of iterations for the hyperparameter optimization step, second row shows average number of iterations for the parameter optimization step (the execution of the optimizer) and third row is the sum of these two. The numbers were computed based on average iterations or epochs until termination, rounded up to integers and then multiplied by batchsizes and number of instances where appropriate.

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic	56	44	24	61	27	35	44	42	39
	112	276	103	88	81	386	380	309	382
	168	320	127	149	108	421	424	351	421
BGD_heavyball	287	212	197	298	142	187	221	219	228
	99	260	98	79	69	367	347	298	130
	386	472	295	377	211	554	568	517	358
BGD_bolddriver	1379	928	642	1426	727	862	1059	1112	652
	84	228	90	67	73	362	349	275	351
	1463	1156	732	1493	800	1224	1408	1387	1003
BGD_rprop	1254	976	626	1404	703	843	1102	1002	454
	89	194	88	69	59	351	346	263	340
	343	1170	714	1473	762	1194	1448	1265	794
CD_basicGD	44	38							
	400	400	–	–	–	–	–	–	–
	444	438							
L-BFGS	–	–	–	–	–	–	–	–	–
	44	64	33	32	39	97	125	114	137
DIBGD_basic	102	81							
	91	204	–	–	–	–	–	–	–
	193	285							
SGD_basic	125000	304000	21000	62200	17112	546550	313600	856000	725000
	150000	512000	19600	42000	17360	1093168	1680000	812000	2450000
	275000	816000	40600	104200	34472	1639718	1993600	1668000	3175000
SGD_heavyball	575000	1520000	21000	302400	87668	2813125	4256000	4132000	2625000
	145000	49600	94500	40000	14880	1093168	1640000	81200	2400000
	720000	1569600	115500	342400	102548	3906293	5896000	4213200	5025000
ASGD_basic	115000	328000	18200	60800	17484	562625	303800	896000	575000
	135000	432000	18200	34000	13640	996712	1560000	756000	2150000
	250000	760000	36400	94800	31124	1559337	1863800	1652000	2725000
IAGD_basic	115000	336000							
	120000	400000	–	–	–	–	–	–	–
	235000	736000							
MBGD_basic	25500	21500	13000	29380	17760	17940	22460	21240	13020
	1660	2640	1300	1300	1260	5820	4980	3120	5280
	27160	24140	14300	30680	19020	23760	27440	24360	183000
MBGD_heavyball	26500	22500	13500	29040	17640	17840	21800	20300	13120
	1580	2560	1280	1240	1220	5640	4640	3060	5180
	28080	25060	14780	30280	18860	23480	26440	23360	18300
MBGD_bolddriver	23000	20000							
	1240	2040	–	–	–	–	–	–	–
	24240	22040							

objective function after optimization on the training and the test set. More likely this should be the method of choice for training NADE, provided that a heuristic is applied to it to protect it from overfitting, such as early stopping.

SGD is therefore more robust. We could say that L-BFGS is more suitable when the goal is optimization itself, while SGD is better suited for learning tasks , where the goal is to generalize. Pure SGD is fairly slow, but its variants help

Table 6.8: Average time (in seconds $\times 10^{-5}$) for 1 iteration for each method group, on each dataset. These times were all computed on the same machine.

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic BGD_heavyball BGD_bolddriver BGD_rprop CD_basicGD	10814	69831	10806	10490	8698	138826	187445	209378	1453730
L-BFGS	121842	310195	61563	44306	61693	619730	819849	771647	6932125
DIBGD_basic	4796	21816	–	–	–	–	–	–	–
SGD_basic SGD_heavyball ASGD_basic IAGD_basic	33	35	49	30	147	35	40	88	213
MBGD_basic MBGD_heavyball MBGD_bolddriver	10814	3305	806	755	568	8105	12532	21310	103941

Table 6.9: Average time (in seconds $\times 10^{-5}$) until termination for each optimization method training NADE with $H = 50$ hidden units. First row shows average time for the hyperparameter optimization step, second row shows average time for the parameter optimization step (the execution of the optimizer) and third row is the sum of these two. All times are divided by $190211438 = maximum \times 10^{-1}$

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic	28	117	11	7	13	316	539	462	4993
BGD_heavyball	32	173	17	21	14	404	560	569	2736
BGD_bolddriver	83	424	42	82	37	893	1388	1527	7666
BGD_rprop	76	430	41	81	35	871	1427	1392	6068
CD_basicGD	25	161	–	–	–	–	–	–	–
L-BFGS	10	104	7	8	5	307	418	386	3218
DIBGD_basic	5	33	–	–	–	–	–	–	–
SGD_basic	48	150	10	16	27	302	419	772	3555
SGD_heavyball	125	289	30	54	79	719	1240	1949	5627
ASGD_basic	43	140	9	15	24	287	392	764	3051
IAGD_basic	42	132	–	–	–	–	–	–	–
MBGD_basic	1544	420	61	122	57	1012	757	2729	10000
MBGD_heavyball	1597	436	63	120	56	1001	1742	2617	10000
MBGD_bolddriver	327	383	–	–	–	–	–	–	–
Normalization: All times divided by $19021.1438 = maximum \times 10^{-1}$									

speed up its convergence. In particular, ASGD seems to speed up convergence as well as slightly improve the quality of optima found. It is to be preferred over other methods that try to decrease the noisiness of the SGD updates, such as SGD_heavyball and IAGD_basic. SGD_heavyball needs one extra hyperparameter to be set[5] and this proves detrimental to the overall execution times, the improvements it offers on both execution time and the value of the objective

[5]Of course we can also imagine versions of ASGD that use a weighted average of past gradients to make updates, with the weights being controlled by additional hyperparameters. Here we refer to the simple version of ASGD we presented.

function attained are small compared to those that ASGD offers. IAGD on the other hand seems to be problematic, at least in the non-convex, high-dimensional parameter space, relatively large scale task we examine. SGD methods lead to faster optimization than BGD methods and this effect becomes more pronounced the larger the dataset is, as a result of redundancies (i.e. many similar instances) within the dataset.

Generally, the various heuristics seem to improve upon the basic versions and do so both on speed of convergence and on the quality of optima found. "Heavy Ball" method appears to be more effective on batch methods and minibatch methods with large batch sizes, than on SGD. "Bold Driver" and "R-prop" help mainly in terms of speeding up convergence, in some cases significantly. In fact, in some datasets the fastest methods prove to be BGD_rprop (more often) and BGD_bolddriver. Of course they do not match the quality of optima found by SGD and L-BFGS.

We had hopes that MBGD would match the performance of SGD. In general they did not, however a different choice of batchsizes, might have helped them become more competitive. They do improve upon batch methods, though only in terms of the optima found. Their execution times, however, suffer from the additional hyparameter that needs to be set (batchsize). Had we fixed, however, the batchsize to any of the values of the grid, MBGD would most likely still perform better than BGD both in terms of optima found (see in Tables 6.5 and 6.6 the big difference in ALL between BGD and MBGD methods) and in terms of execution times (compare only the optimization times in Table C.1 of Appendix C, ignoring hyperparameter optimization, generally, the larger the dataset, the faster MBGD methods are compared to BGD).

To close our discussion regarding the methods, DIBGD_basic, although promising as an idea, seems to have not been properly implemented here. No measure was taken to prevent it from overfitting and the non-convexity of the objective function of NADE appears to make the optimizer get stuck early in local optima.

As for the hyperparameters selected in each case, although we did not keep proper records of the hyperparameter vectors selected we made some general observations during the experimentations that might be useful in further studies. In "easy" datasets (the extremes being *adult* and *mushrooms*) we observed more variation in the choice of hyperparameters per run than in "difficult" ones (the extremes being *dna* and *nips-0-12*). The (initial) learning rate α_0 and to a lesser extent parameter C that controls its decay exhibited the highest variation among datasets. Experimentation showed that further tweaking the learning rate to values not on the grid we searched, could lead to improved performance (faster convergence/better optima). The 20 epoch/20 iteration limit might have been harsher for batch methods, forcing them to terminate hyperparameter optimization runs far away from the optimum, as a result choosing suboptimal learning rates and particularly favouring larger values. The momentum hyperparameter γ appeared to be less important than the stepsize and vales within the same scale did not affect considerably the end result. As we said, the way we picked

the batchsize in MBGD was probably not the ideal. In retrospect, by fixing the batchsize to –say– 2% of the full batch, MBGD methods could have achieved similar or better performance in terms of ALL, without paying the heavy computational price of an extra free hyperparameter. Similarly the hyperparameters σ and ρ of 'Bold Driver" and "R-prop" in the BGD methods should have been set to the "rule of thumb" values $\rho = 1.1$ and $\sigma = 0.5$ to remove the hyperparameter optimization overhead for BGD_bolddriver and BGD_rprop. However these too methods were very sensitive to σ and ρ so we should first experiment to select appropriate values for them on each dataset separately.

In this project we did not go into a detailed study of the properties of the individual datasets beforehand, as our goal was to build a framework that could be applied to any dataset, allowing us to compare the methods. Hence the broad range of hyperparameters on the grid (5 choices per each hyperparameter) and the "universality" of the termination criteria across datasets. It is generally a good idea to "get a feel" of the dataset first, perhaps use only parts of it to see how easy it is to train on (how sensitive it is to the choice of method or the setting of hyperparameters, how different results we get by training on different subsets of it or by using different initializations). If we detect high variances in the results, this is an indication that we need careful fine-tuning of the hyperparameters. If the variances are small we could perhaps afford to dedicate little resources to hyperparameter optimization. Early stopping conditions in "difficult to optimize" hyperparameter spaces should be more forgiving, allowing for more iterations without exceeding the best solution found so far, while for "easy" datasets we can be stricter. The size of the parameter space (in many models, such as in NADE this depends on the dimensionality of the dataset) should also affect the parameter initialization.

Having seen how the individual methods perform, we suggest as a first option to use L-BFGS, since it is faster and it does not require any hyperparameters to be set. Ideally, measures should be taken to prevent it from overfitting. In the case of large datasets (judging from our results this means ≥ 30000 instances) SGD methods might be a better choice, especially ASGD. In any case each dataset is different (so each parameter space is different) and other factors —such as redundancy— can give an advantage to SGD. In large scale non-convex problems we should probably avoid using BGD.

Chapter 7

Conclusion and Future Work

In this study we examined optimization in the context of machine learning. We covered the basic steps of implementing and using optimization algorithms from understanding the ideas behind the basic algorithms themselves, to optimizing the hyperparameters involved, from checking the gradient code and protecting the methods from overfitting to evaluating methods and choosing the most suitable among them. We focused on a non-convex objective function, with a high dimensional parameter space and used datasets of small to fairly large size, however, most of this work carries on to general optimization problems in machine learning.

Among the methods we examined the ones better suited for our task are L-BFGS and SGD. L-BFGS is faster in most cases and always finds slightly better local optima, but is more likely to overfit the training data than SGD. The most successful variant of SGD was ASGD improving both the speed, in large datasets being even faster than LBFGS and the quality of optima found by SGD without adding any additional hyperparameters.

We also verified the power of the NADE model showing that even using first order batch methods, despite the objective function being non-convex, even with a small number of hidden units it can outperform simpler models. An interesting factor to explore in further studies could be finding the minimal value for H in NADE that leads to the optimal density estimation results on each of the dataset. The optimal value of H could also demonstrate the amount of redundancy (interdependencies among features) within the datasets.

This study was far from exhaustive. We did not cover higher order stochastic methods such as Stochastic Quasi-Newton [Bordes et al., 2009]. We also did not cover many popular batch methods such as Conjugate Gradient and members of the Quasi-Newton family like Barzilai-Borwein [Barzilai et al., 1988] (Barzilai-Borwein approximates the Hessian by a diagonal matrix to obtain a step size for gradient descent). Both are implemented in the minfunc() package and comparisons could be easily extended to include them as well.

There are possible directions to explore even with the methods we experimented

with. We could combine different methods in different stages of the optimization. For example we could start with stochastic methods and once the training progresses (and we are nearing a local optimum), then we can switch to a batch method for the few remaining iterations. A simple way to mimic this approach could be to use a minibatch method with batch sizes that increase in size every few epochs. The Dynamically Increasing Batch size methods are doing something similar but as we saw they did not perform well in our task. We hypothesized that the poor performance was due to the absence of regularization combined with the non-convexity of the objective function.

An interesting experiment here would be to use DIBG_basic to optimize a convex objective function. If the method proves successful, we would have another indication that our hypothesis that it gets trapped in local optima and overfits the training data on a non-convex setting is correct. Furthermore, we could also explore other methods of the Dynamically Increasing Batch Size family by applying its idea of growing the batch size based on the result of a statistical test on other batch methods. These could be methods we already examined here, e.g. BGD_rprop or L-BFGS, which proved better suited to the problem than simple BGD.

Another heuristic we can use, is executing the optimization algorithms starting from various initializations and keeping the best one as evaluated on the validation set. This idea could be used to give us a good initialization of the parameters and we could also compare how well different methods perform as "initializers" for others. Another unexplored idea would be to use a minibatch version of L-BFGS, this might help improve both speed (by exploiting redundancy in the dataset) and quality of optima (by adding some amount of stochasticity to the updates).

Finally, since our results indicated that LBFGS is prone to overfitting, a good idea would be to try to modify L-BFGS to include some form of overfitting control. A simple example would be to use early stopping. Another approach would be to use a regularized objective (using e.g. *L1*-regularization) and combine it with a variant of L-BFGS like Orthant-wise limited-memory quasi-Newton (OWL-QN). OWL-QN is specifically designed for fitting L1-regularized models [Galen et al., 2007].

We also saw that variants of stochastic gradient descent such as ASGD work very well and appear to offer more consistent results. However, they are in most cases slower than L-BFGS and require fine-tuning their learning rate. We could modify ASGD to increase its speed by exploiting the sparsity of the datasets (and the resulting update vectors). A version of ASGD that takes sparsity into account is presented in [Xu, 2012]. Another interesting variant to explore is the version of SGD proposed in [Schaul et al., 2012] which automatically adjusts the step size on every iteration so as to minimize the expected error after the next update. The same method can be applied to individual (or even grouped) learning rates per parameter. Using this version of SGD, there is no longer need to fine-tune the step size (and parameter C that controls its decay in our version). This would effectively remove all need for hyperparameter optimization in our SGD

implementation.

As for hyperparameter optimization in general, we saw that for many of the methods examined, it plays an important role. We also saw that performing a combinatorial search in a grid of the hyperparameter space can become the bottleneck of the entire procedure (for more than 2 hyperparameters with 5 options for each). In [Bergstra et al., 2012] the authors propose random search as an alternative to grid-search. They argue that only a few of the hyper-parameters really matter for most data sets (not the same in all). When doing a —say— $n \times n$ grid search we only examine n values per hyperparameter. If on the other hand we examine n^2 random hyperparameter configurations we can cover up to n^2 different values per hyperparameter at the same computational cost. Thus, by doing a random search we can find hyperparameter configurations that are as good or better than those found by a grid search, but much faster. In [Bergstra et al., 2011] the authors present two greedy sequential hyperparameter optimization techniques capable of dealing with both discrete and continuous hyperparameters. Both methods take into account priors over the hyperparameters and are based in the optimization of the criterion of Expected Improvement [Jones, 2001]. When compared to random search in the hyperparameter space and the manual grid-aided search they outperform both. It would be a good idea to explore alternative schemes such as this one to automatically select the optimal hyper-parameter configuration.

Bibliography

[Barzilai et al., 1988] Barzilai J. & Borwein J. M. (1988) *Two-point step size gradient methods.* IMA Journal of Numerical Analysis, 8 (1988), 141–148.

[Bengio et al., 2007] Bengio Y. , Lamblin P. , Popovici P. & Larochelle H. (2007) *Greedy Layer-Wise Training of Deep Networks.* Advances in Neural Information Processing Systems 19, MIT Press, Cambridge, MA.

[Bengio, 2009a] Bengio Y. (2009). *Learning deep architectures for AI.* Foundations and Trends in Machine Learning, vol. 2, iss. 1, pp. 1 – 127.

[Bengio, 2009b] Bengio Y. , Louradour J. , Collobert R. & Weston J. (2009). *Curriculum Learning.* Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML 2009), ACM.

[Bengio et al., 2000] Bengio, Y. & Bengio, S. (2000). *Modeling high-dimensional discrete data with multi-layer neural networks.* Advances in Neural Information Processing Systems 12 (NIPS 1999) pp. 400 – 406. MIT Press.

[Bengio, 2012] Bengio Y. 2012. *Practical Recommendations for Gradient-Based Training of Deep Architectures.* CoRR abs/1206.5533.

[Bergstra et al., 2011] Bergstra J. , Bardenet R. , Bengio Y. and Kegl B. (2011). *Algorithms for Hyper-parameter Optimization.* NIPS 2011.

[Bergstra et al., 2012] Bergstra J. & Bengio Y. (2012). *Random Search for Hyper-Parameter Optimization.* Journal of Machine Learning Research, 13(281 – 305).

[Bishop, 1995] Bishop C. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press.

[Bordes et al., 2009] Bordes A. , Bottou L. & Patrick G. (2009). *SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent.* Journal of Machine Learning Research, 10: 1737 – 1754.

[Bottou et al., 2004] Bottou L. & LeCun Y. (2004). *Large Scale Online Learning.* Advances in Neural Information Processing Systems 16, MIT Press, Cambridge, MA.

[Bottou et al., 2007] Bottou L. & Bousquet O. (2007). *The Tradeoffs of Large Scale Learning.* In Proceedings of NIPS 2007.

[Broyden, 1970] Broyden, C. G. (1970). *The convergence of a class of double-rank minimization algorithms.* Journal of the Institute of Mathematics and Its Applications 6: 76 – 90.

[Boyd et al., 2004] Boyd S. P. & Vandenberghe L. (2004). *Convex Optimization.* Cambridge University Press. p. 129. ISBN 978-0-521-83378-3.

[Boyles et al., 2011] Boyles L. , Korattikara A. , Ramanan, D. and Welling, M. (2011). *Statistical Tests for Optimization Efficiency.* NIPS 2011.

[Collobert et al., 2009] Collobert R. & Weston J. (2009). *A unified architecture for natural language processing: Deep neural networks with multitask learning.* Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML 2009), ACM.

[Deng et al., 2011] Deng L. & Yu D. (2011). *Deep Convex Network: A Scalable Architecture for Deep Learning.* Interspeech 2011, pp. 2285–2288.

[Fletcher, 1970] Fletcher, R. (1970). *A New Approach to Variable Metric Algorithms.* Computer Journal 13 (3): 317 – 322.

[Frey, 1998] Frey B. J. (1998). *Graphical models for machine learning and digital communication.* MIT Press.

[Galen et al., 2007] Galen A. & Jianfeng G. (2007). *Scalable Training of L1-Regularized Log-Linear Models.* 24th Annual International Conference on Machine Learning (ICML 2007).

[Goldfarb, 1970] Goldfarb, D. (1970). *A Family of Variable Metric Updates Derived by Variational Means.* Mathematics of Computation 24 (109): 23 – 26.

[Hahn et al., 2011] Hahn S. , Lehnen P. & Ney H. (2011). *Powerful extensions to CRFS for grapheme to phoneme conversion.* In Proceedings of ICASSP. 2011, 4912 – 4915.

[Heigold et al., 2009] Heigold G. , Rybach D. , Schlüter R. & Ney H. (2009). *Investigations on Convex Optimization Using Log-Linear HMMs for Digit String Recognition.* Interspeech 2009.

[Hestenes et al., 1952] Hestenes M. R. & Stiefel E. (1952). *Methods of Conjugate Gradients for Solving Linear Systems.* Journal of Research of the National Bureau of Standards 49 (6).

[Hinton et al., 2006a] Hinton G. E. & Salakhutdinov R. R. (2006) *Reducing the Dimensionality of Data with Neural Networks.* Science, 28 July 2006, Vol. 313. no. 5786, pp. 504 – 507.

[Hinton et al., 2006b] Hinton G. E. , Osindero S. & Teh, Y. (2006) *A fast learning algorithm for deep belief nets.* Neural Computation 18, pp 1527 – 1554.

[Hinton, 2010] Hinton G. E. (2010). *A Practical Guide to Training Restricted Boltzmann Machines.* Technical report.

[Huck et al., 2010] Huck M. , Ratajczak M. , Lehnen P. & Ney H. (2010). *A comparison of various types of extended lexicon models for statistical machine translation.* AMTA 2010: the Ninth conference of the Association for Machine Translation in the Americas.

[Jones, 2001] Jones D. R. (2001). *A taxonomy of global optimization methods based on response surfaces.* Journal of Global Optimization, 21:345–383.

[Larochelle et al., 2010] Larochelle, H. , Bengio, Y. & Turian, J. (2010). *Tractable multivariate binary density estimation and the restricted Boltzmann forest.* Neural Computation, 22, pp. 2285 – 2307.

[Larochelle et al., 2011] Larochelle H. & Murray I. (2011). *The Neural Autoregressive Distribution Estimator.* JMLR: W&CPl 15, pp. 29 – 37.

[Le et al., 2011] Le Q. V. , Ngiam J. , Coates A. , Lahiri A. , Prochnow B. & Ng A. Y. (2011). *On optimization methods for deep learning.* ICML 2011: 265 – 272.

[LeCun et al., 1998] LeCun Y. , Bottou, L. , Orr G. & Muller K. (1998). *Lecture Notes in Computer Science In Neural Networks–Tricks of the Trade.* Vol. 1524, pp. 5 – 50.

[Lee et al., 2009] Lee H. , Largman Y. , Pham P. & Ng A. Y. (2009). *Unsupervised feature learning for audio classification using convolutional deep belief networks.* Advances in Neural Information Processing Systems 23 (NIPS 2009).

[Lee et al., 2011] Lee H. , Grosse R. , Ranganath R. & Ng A. Y. (2011) *Unsupervised learning of hierarchical representations with convolutional deep belief networks.* Commun. ACM 54(10): 95 – 103.

[Le Roux et al., 2012] Le Roux N. , Schmidt M. & Bach, F. (2012). *A Stochastic Gradient Method with an Exponential Convergence Rate for Strongly-Convex Optimization with Finite Training Sets.* Submitted.

[Morrone et al., 1988] Morrone M. C. & Burr D. C. (1988). *Feature detection in human vision: a phase dependent energy model.* Proc. R. Soc. (Lond) B235, 221 – 245.

[Murray, 2010] Murray W. (2011). *Newton-Type Methods.* Wiley Encyclopedia of Operations Research and Management Science.

[Nesterov, 2010] Nesterov Y. (2010). *Efficiency of coordinate descent methods on huge-scale optimization problems.* CORE Discussion Paper, 2010/2.

[Nocedal, 1980] Nocedal J. (1980). *Updating Quasi-Newton Matrices with Limited Storage.* Mathematics of Computation 35, pp. 773 – 782.

[Polyak, 1964] Polyak B. T. (1964). *Some methods of speeding up the convergence of iteration methods.* Z. Vycisl. Mat. i Mat. Fiz. , 4, pp. 1 – 17.

[Polyak et al., 1992] Polyak B. T. & Juditsky A. B. (1992). *Acceleration of Stochastic Approximation by Averaging.* SIAM Journal on Control and Optimization 30, No. 4. pp. 838 – 855.

[Ranzato et al., 2007] Ranzato M. , Huang F. J. , Boureau Y. & LeCun Y. (2007) *Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition.* Proc. of Computer Vision and Pattern Recognition Conference (CVPR 2007).

[Riedmiller et al., 1993] Riedmiller M. & Braun H. (1993). *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm.* IEEE International Conference On Neural Networks, pp. 586 – 591.

[Robbins et al., 1951] Robbins H. & Monro S. (1951). *A Stochastic Approximation Method.* Annals of Mathematical Statistics 22, #3 , pp. 400 – 407.

[Salakhutdinov et al., 2007b] Salakhutdinov R. R. & Hinton G. E. (2007). *Semantic Hashing.* In Proceedings of the SIGIR Workshop on Information Retrieval and Applications of Graphical Models, Amsterdam.

[Salakhutdinov et al., 2007a] Salakhutdinov R. R. , Mnih A. & Hinton G. E. (2007). *Restricted Boltzmann Machines for Collaborative Filtering.* International Conference on Machine Learning (ICML 2007).

[Schaul et al., 2012] Schaul T. , Zhang S. & LeCun Y. (2012) *No More Pesky Learning Rates.* Technical Report, arXiv:1206.1106v1

[Shanno, 1970] Shanno D. F. (1970). *Conditioning of quasi-Newton methods for function minimization.* Mathematics of Computation 24 (111): 647 – 656.

[Smolensky, 1986] Smolensky P. (1986). *Information processing in dynamical systems: Foundations of harmony theory.* Parallel Distributed Processing: Volume 1: Foundations, pages 194–281. MIT Press, Cambridge, MA.

[Sutskever et al., 2007] Sutskever I. & Hinton, G. E. (2007) *Learning Multilevel Distributed Representations for High-dimensional Sequences.* AI and Statistics, 2007.

[Sutskever et al., 2008] Sutskever I. & Hinton G. E. (2008). *Deep Narrow Sigmoid Belief Networks are Universal Approximators.* Neural Computation, Vol 20, pp 2629 – 2636.

[Taylor et al., 2007] Taylor G. W. , Hinton G. E. & Roweis S. (2007) *Modeling human motion using binary latent variables.* Advances in Neural Information Processing Systems 19, MIT Press, Cambridge, MA.

[Vishwanathan et al., 2006] Vishwanathan S. , Schraudolph N. , Schmidt M. & Murphy K. (2006). *Accelerated Training of Conditional Random Fields with Stochastic Gradient Methods.* ICML 2006.

[Vogl et al., 1988] Vogl T. P. , Mangis J. W. , Rigler A. K. , Zink W. T. & Alkon D. L. (1988). *Accelerating the convergence of the back-propagation method.* Biological Cybernetics 59, pp. 257 – 63.

[Watson, 2000] Watson A. B. (2000). *Visual detection of spatial contrast patterns: Evaluation of five simple models* Optics Express, (6):12 – 33.

[Wulsin et al., 2011] Wulsin D. F. , Gupta J. R. , Mani R. , Blanco J. A. & Litt B. (2011). *Modeling electroencephalography waveforms with semi-supervised deep belief nets: fast classification and anomaly measurement.* J Neural Eng. 2011 Apr 28;8(3):036015.

[Xu, 2012] Xu W. (2011) *Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.* CoRR abs/1107.2490.

[Zoran et al., 2011] Zoran D. & Weiss Y. (2011). *From learning models of natural image patches to whole image restoration.* ICCV, pp. 479 – 486.

[Zweig et al., 2010] Zweig G. & Nguyen P. (2010). *SCARF: a segmental conditional random field toolkit for speech recognition.* Interspeech 2010: 2858 – 2861.

Appendix A

Notational Conventions

- Vectors are denoted by lowercase boldface letters or sometimes (mainly in pseudocodes) variables written in bold starting with a lowercase letter (e.g. $\boldsymbol{\theta}$, $\boldsymbol{pass}$).
- Matrices are denoted by uppercase boldface letters or sometimes (mainly in pseudocodes) variables written in bold starting with an uppercase letter (e.g. $\boldsymbol{H}$, $\boldsymbol{Batch}$).
- We use the notation $\boldsymbol{A(1,*)}$ in pseudocodes and $\boldsymbol{A_{1,*}}$ in formulae to refer to the of the first row of $\boldsymbol{A}$ (i.e. $\boldsymbol{A(1,*)} \equiv \boldsymbol{A_{1,*}}$ and is a vector).
- We use the notation $\boldsymbol{x}^{(T)}$ to denote the T-th instance $\boldsymbol{x^{(t)}}$. We use $\boldsymbol{x}^T$ to denote the transpose of a vector $\boldsymbol{x}$.
- When defining a D-dimensional vector $\boldsymbol{a}$ by "$\boldsymbol{a} \in \mathbb{R}^D$" without specifying whether it is a column or a row vector, assume it is whichever of the two is required by the operations $\boldsymbol{a}$ is involved in to be defined.
- We use $\{a(n)\}$ to refer to the series $a(0), a(1), a(2), ..., a(n)$.
- We use $f(x)$ interchangeably with $f(x_0)$, when it is clear from the context that we refer to the value of $f(x)$ for $x = x_0$. Similarly, we might even just use f in some cases to denote the same thing (especially in pseudocodes) when doing so adds no ambiguity. This notational slip carries on to partial derivatives, the gradient and the Hessian of f as well.
- In pseudocodes, constants are usually written in plain uppercase characters (e.g. N, MAX_ITERS). Their values are always explained in the main text and —whenever deemed necessary— in the algorithms themselves in commends or by an explicit value assignment. So if some symbol or term is not specified in a pseudocode, then it is a constant and it is bound to be explained in the main text, most likely in the subsection describing the pseudocode.
- In pseudocodes, functions are also written in uppercase (e.g. OBJFUNCEVAL(), COMPUTE_OBJECTIVE_AND_GRADIENT()). Most of these are

defined in other pseudocodes. Exceptions include ZEROS(), SIZE() and SHUFFLE().

- In pseudocodes, we assume predefined functions ZEROS() and SIZE() that work the same way their counterparts in Matlab/Octave do. So ZEROS(m,n) will create a $m \times n$ matrix with all its elements equal to zero. SIZE($\boldsymbol{\theta}$) returns the length of vector $\boldsymbol{\theta}$ and SIZE($\boldsymbol{A}$) the dimensions of matrix $\boldsymbol{A}$. We use SIZE($\boldsymbol{A}$, ROWS) to get the number of rows of $\boldsymbol{A}$ and SIZE($\boldsymbol{A}$, COLUMNS) to get the number of columns of $\boldsymbol{A}$. Conversely, we use SHUFFLE($\boldsymbol{A}$, ROWS) to get a random permutation of the rows of $\boldsymbol{A}$ and SHUFFLE($\boldsymbol{A}$, COLUMNS) to get a random permutation of the columns of $\boldsymbol{A}$.
- We use $\boldsymbol{A(*)}$ to denote the column vector that we get by concatenating all columns of matrix $\boldsymbol{A}$.
- We use the notation $1:T$ to denote $1, 2, ..., T$ just like in Matlab/Octave.
- We use the notation $a = ++$ as a shorthand for $a = a + 1$, as in many programming languages (but not Matlab).
- We use the "=" operator to denote value assignment and the "==" operator to denote equality, as is done in most programming languages.

Appendix B

Table of Method Names

Table B.1: The methods discussed here, a brief description of each and a reference to the algorithm that describes it.

Method	Brief Description	Algorithm	Implemented	Comments
BGD_basic	The simple version of batch gradient descent	4	Yes	
BGD_heavyball	Batch gradient descent using the "Heavy Ball" heuristic	5	Yes	
BGD_bolddriver	Batch gradient descent using the "Bold Driver" heuristic	6	Yes	
BGD_rprop	Batch gradient descent using the "Resilient Backpropagation" heuristic	7	Yes	
CD_basicGD	Coordinate Descent using a batch gradient descent step to update each parameter	8	Yes	
CD_basicNewton	Coordinate Descent using an inexpensive Newton step to update each parameter	9	Yes	Only used in logistic regression toy problem
Newton	Newton's Method	Not given. Update step described by Eq. (3.4)	Yes	Only used in logistic regression toy problem
L-BFGS	Limited Memory BFGS	10	No. Used minfunc() implementation	
DIBGD_basic	Dynamically increasing batch size version of batch gradient descent	12	Yes	
SGD_basic	The simple version of stochastic gradient descent	13	Yes	
SGD_heavyball	Stochastic gradient descent using the "Heavy Ball" heuristic	14	Yes	
ASGD_basic	Averaged stochastic gradient descent descent	15	Yes	
IAGD_basic	Iterate averaging stochastic gradient descent	16	Yes	
MBGD_basic	The simple version of minibatch gradient descent	13	Yes	
MBGD_heavyball	Minibatch gradient descent using the "Heavy Ball" heuristic	18	Yes	
MBGD_bolddriver	A version of minibatch gradient descent using the "Bold Driver" heuristic	19	Yes	

Appendix C

Average Execution Times Table

Table C.1: Average time (in seconds $\times 10^{-5}$) until termination for each optimization method training NADE with $H = 50$ hidden units. First row shows average time for the hyperparameter optimization step, second row shows average time for the parameter optimization step (the execution of the optimizer) and third row is the sum of these two.

Method	*adult*	*connect-4*	*dna*	*mushrooms*	*nips-0-12*	*ocr-letters*	*rcv1*	*web*	*binarized mnist*
BGD_basic	605584	3072564	259344	639890	234846	4858910	8247580	8793876	56695470
	1211168	19273356	1113018	923120	704538	53586836	71229100	64697802	555324860
	1816752	22345920	1372362	1563010	939384	58445746	79476680	73491678	612020330
BGD_heavyball	3103618	14804172	2128782	3126020	1235116	25960462	41425345	45853782	331450440
	1070586	18156060	1058988	828710	600162	50949142	65043415	62394644	188984900
	4174204	32960232	3187770	3954730	1835278	76909604	106468760	108248426	520435340
BGD_bolddriver	14912506	64803168	6937452	14958740	6323446	119668012	198504255	232828336	947831960
	908376	15921468	972540	702830	634954	50255012	65418305	57578950	510259230
	15874952	80724636	7909992	15661570	6958400	169923024	263922560	290407286	1458091190
BGD_rprop	13560756	68155056	6764556	14727960	6114694	117030318	206564390	209796756	659993420
	962446	13547214	950928	723810	513182	48727926	64855970	55066414	494268200
	14523202	81702270	7715484	15451770	6627876	165758244	271420360	264863170	1154261620
CD_basicGD	475816	2653578							
	4325600	27932400	–	–	–	–	–	–	–
	4801416	30585978							
L-BFGS	–	–	–	–	–	–	–	–	–
	5361048	19852480	2031579	1417792	2406027	60113810	102481125	87967758	949701125
DIBGD_basic	489192	1767096							
	436436	4450464	–	–	–	–	–	–	–
	925628	6217560							
SGD_basic	4125000	10640000	1029000	1866000	2515464	19129250	12544000	75328000	154425000
	4950000	17920000	960400	1260000	2551920	38260880	67200000	71456000	521850000
	9075000	28560000	1989400	3126000	5067384	57390130	79744000	146784000	676275000
SGD_heavyball	18975000	53200000	1029000	9072000	12887196	98459375	170240000	363616000	559125000
	4785000	1736000	4630500	1200000	2187360	38260880	65600000	7145600	511200000
	23760000	54936000	5659500	10272000	15074556	136700000	235840000	370761600	1070325000
ASGD_basic	3795000	11480000	891800	1824000	2570148	19691875	12152000	78848000	122475000
	4455000	15120000	891800	1020000	2005080	34884920	62400000	66528000	457950000
	8250000	26600000	1783600	2844000	4575228	54576795	74552000	145376000	580425000
IAGD_basic	3795000	11088000							
	4200000	14000000	–	–	–	–	–	–	–
	7995000	25088000							
MBGD_basic	275800000	71100000	10500000	22200000	10100000	145400000	81500000	452600000	1353300000
	17951240	8725200	1047800	981500	715680	47171100	62409360	66487200	548808480
	293751240	79825200	11547800	23181500	10815680	192571100	143909360	519087200	1902108480
MBGD_heavyball	286600000	74400000	10900000	21900000	10000000	144600000	273200000	432600000	1363700000
	17086120	8460800	1031680	936200	692960	45712200	58148480	65208600	538414380
	303686120	82860800	11931680	22836200	10692960	190312200	331348480	497808600	1902114380
MBGD_bolddriver	248722000	66100000							
	13409360	6742200	–	–	–	–	–	–	–
	62131360	72842200							