# Modeling, Inference and Optimization With Composable Differentiable Procedures


## Citation
Maclaurin, Dougal. 2016. Modeling, Inference and Optimization With Composable Differentiable Procedures. Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences.

## Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:33493599

## Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

# Share Your Story
The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

# Modeling, Inference and Optimization with Composable Differentiable Procedures


A DISSERTATION PRESENTED
BY
DOUGAL MACLAURIN
TO
THE DEPARTMENT OF PHYSICS


IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
PHYSICS


HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
APRIL 2016

©2016 – Dougal Maclaurin
all rights reserved.

Thesis advisor: Professor Ryan Prescott Adams Dougal Maclaurin

# Modeling, Inference and Optimization with Composable Differentiable Procedures

## Abstract

This thesis presents five contributions to machine learning, with themes of differentiability and Bayesian inference.

We present Firefly Monte Carlo, an auxiliary variable Markov chain Monte Carlo algorithm that only queries a potentially small subset of data at each iteration yet simulates from the exact posterior distribution.

We describe the design and implementation of Autograd, a software package for efficiently computing derivatives of functions written in Python/Numpy using reverse accumulation mode differentiation.

Using Autograd, we develop a convolutional neural network that takes arbitrary graphs, such as organic molecules, as input. This generalizes standard molecular feature representations and allows end-to-end adaptation of the feature extraction pipeline to particular tasks.

We show how to compute gradients of cross-validation loss with respect to hyperparameters of learning algorithms, with both time and memory efficiency, by chaining gradients backwards through an exactly reversed optimization procedure.

Finally, by accounting for the entropy destroyed by optimization, we show that early stopping and ensembling, popular tricks for avoiding overfitting, can be interpreted as variational Bayesian inference.

# Contents


1 Introduction 1

2 Background 4
- 2.1 Bayesian inference . . . 4
- 2.2 Point estimation . . . 6
- 2.3 Variational inference . . . 7
- 2.4 Markov chain Monte Carlo inference . . . 9
- 2.5 Computing gradients in reverse accumulation mode . . . 13

3 Firefly Monte Carlo 20
- 3.1 Introduction . . . 21
- 3.2 Firefly Monte Carlo . . . 22
- 3.3 Implementation considerations . . . 26
- 3.4 Experiments . . . 33
- 3.5 Conclusion . . . 38

4 Autograd: Automatic Differentiation for Python 41
- 4.1 Mission and design principles . . . 42
- 4.2 Building the computation graph . . . 44
- 4.3 Performing the backward pass . . . 46
- 4.4 Primitive vector-Jacobian products . . . 48
- 4.5 Generalizing to other data types . . . 50
- 4.6 Handling all of Python's syntax . . . 52
- 4.7 Limitations and future work . . . 54
- 4.8 Conclusion . . . 56

5 Convolutional Networks on Graphs 58
- 5.1 Introduction . . . 59
- 5.2 Circular fingerprints . . . 60
- 5.3 Creating a differentiable fingerprint . . . 61
- 5.4 Experiments . . . 63
- 5.5 Limitations . . . 69
- 5.6 Related work . . . 71
- 5.7 Conclusion . . . 73

6 Hyperparameter Optimization Through Reversible Learning **75**
6.1 Introduction . . . . . 76
6.2 Hypergradients . . . . . 78
6.3 Experiments . . . . . 84
6.4 Limitations . . . . . 91
6.5 Related work . . . . . 95
6.6 Extensions and future work . . . . . 96
6.7 Conclusion . . . . . 97

7 Early Stopping as Variational Inference **98**
7.1 Introduction . . . . . 99
7.2 Incomplete optimization as variational inference . . . . . 102
7.3 The entropy of stochastic gradient descent . . . . . 104
7.4 Entropy-friendly optimization methods . . . . . 108
7.5 Experiments . . . . . 110
7.6 Limitations . . . . . 113
7.7 Related work . . . . . 115
7.8 Future work and extensions . . . . . 117
7.9 Conclusion . . . . . 118

8 Conclusion **119**

References **134**

# Acknowledgments

First, a sincere thank you to my advisor, Ryan Adams, for making all of this possible. Ryan is a daily inspiration as a scientist, engineer and entrepreneur. He has created an amazing research group and he cares deeply about his students and postdocs. Thanks for taking a chance on me, believing that I could retool as a computer scientist and welcoming me into your group. And thanks for seeding the quadcopter plague of summer 2015, even though it devastated our office plants.

Thanks to David Duvenaud, my comrade-in-arms. Almost all of the work in this thesis was produced as a partnership: pair-programming, pair-writing, pair-cogitating and pair-throwing-nerf-darts. It was an absolute privilege to have a collaborator with such intelligence, imagination and energy.

Thanks to Matt Johnson, an early evangelist for Autograd, who responds to users' issues faster than I can read them. Thanks for patiently teaching me the importance of linear algebra, and for innumerable computer science gems.

I could wax lyrical about each member of the HIPS group, past and present. You make MD209 feel like a small slice of paradise. Thanks Mike Gelbart, Scott Linderman, Andy Miller, Oren Rippel, Yakir Reshef, Alex Wiltschko, Miguel Hernández-Lobato, Elaine Angelino, Diana Cai, Jasper Snoek, James Zou and Finale Doshi-Velez. Thanks for the camaraderie and the mind-expanding conversations. Particular thanks to Finale for taking the helm of machine learning at Harvard this past year.

Thanks to Alán Aspuru-Guzik and the OLED team: Jorge Aguilera-Iparraguirre, Rafa Gómez-Bombarelli and Tim Hirzel. You have been collaborators of the first order and wonderful people to get to know.

Thanks to Adam Cohen for taking the time to teach me so much, from optics to electronics to machining techniques and for sharing your intellectual breadth and intensity and your indefatigable enthusiasm for science. Thanks to the everyone in the Cohen Lab and to Veena Venkatachalam in particular, my co-conspirator throughout long days skewering cells with glass pipettes in the name of photophysics.

Thanks to the Harvard Physics Department, and to Jacob Barandes and Lisa Cacciabaudo in particular, for your kindness and encouragement and for tirelessly working to help grad students.

Thanks to SAIT, ADI, and the Frank Knox Memorial Fellowship for funding.

Thanks to all of my grad school friends and the Nashton community. Particular thanks to Miriam Huntley for originally nudging me towards machine learning and to Noam Prywes for managing that august institution, the Advanced Society for the Advancement of Advanced Societal Advances. Where else but the ASAASA would I have learned about mushrooms with refractive, image-forming eyes, and heard first-hand accounts of hunting for quasicrystals in Eastern Siberia?

Thanks to my parents and sisters, Marianne, Simon, Elspeth and Lydia, for supporting me my whole life in everything I do. Most importantly, thank you to my wife, Melis. Thanks for being my constant companion, for your unwavering love and support, and for getting high on science with me.

# Further Acknowledgements by Chapter

CHAPTER 3 contains content that has been previously published in:
Maclaurin, D. & Adams, R. P. (2014). Firefly Monte Carlo: Exact MCMC with subsets of data. In *30th Conference on Uncertainty in Artificial Intelligence*.

CHAPTER 4 contains contributions from David Duvenaud and Matt Johnson.

CHAPTER 5 contains content that has been previously published in:
Duvenaud*, D., Maclaurin*, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., & Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Neural Information Processing Systems*.

CHAPTER 6 contains content that has been previously published in:
Maclaurin*, D., Duvenaud*, D., & Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *32nd International Conference on Machine Learning*.

CHAPTER 7 contains content that has been previously published in:
Duvenaud*, D., Maclaurin*, D., & Adams, R. P. (2016). Early stopping as non-parametric variational inference. In *19th International Conference on Artificial Intelligence and Statistics*.

* denotes equal contribution

# 1

# Introduction

In late 2015, Tesla Motors quietly pushed a software update, "Autopilot", to its Model S electric sedans. The update allowed the cars to drive themselves on the highway and change lanes at the flick of a turn signal. It was made possible by a computer vision system which used a convolutional neural network, trained on vast amounts of driving data, to locate lanes and other vehicles from camera images.

Exploiting data to achieve engineering goals like these is the essence of machine learning, a relatively young field with recent high-profile successes in domains such as computer vision, speech recognition and the board game Go. This thesis presents five diverse but interconnected contributions to the field: Firefly Monte Carlo (Chapter 3); Autograd (Chapter 4); Neural Molecular Fingerprints (Chapter 5); gradient-based

hyperparameter optimization (Chapter 6); and a reinterpretation of early stopping as variational inference (Chapter 7).

The common theme linking these chapters is differentiability and the effectiveness of gradient-based optimization. Autograd is a software tool we wrote for efficiently computing gradients of functions written in Python. Chapters 5, 6 and 7 use Autograd for regression on molecules, hyperparameter optimization, and approximate inference respectively. Although Chapter 3, Firefly Monte Carlo, is not directly concerned with differentiation, it belongs to a sub-theme of Bayesian inference, along with Chapter 7.

We begin, in Chapter 2, with background material. We introduce two general-purpose methods for Bayesian inference: Markov chain Monte Carlo (MCMC) and variational inference. We also describe reverse accumulation mode differentiation, the algorithm for efficient gradient computation that underpins Autograd.

In Chapter 3 we introduce the Firefly Monte Carlo algorithm. Firefly tackles the problem of applying MCMC to large data sets, which is usually infeasible because it ordinarily requires examining every data point at each iteration. Firefly Monte Carlo only queries a subset of the data at each iteration, yet maintains the true posterior as its stationary distribution.

Chapter 4 covers the design and implementation of Autograd, an open-source software package for automatically and efficiently computing gradients of functions written in Python/Numpy. Autograd allows functions to be written using native Python syntax, taking advantage of the full expressiveness offered by a modern programming language. Much of machine learning boils down to writing down a loss function and optimizing it with gradients. By making the gradient computation effortless, Autograd makes it easy to rapidly prototype new ideas. It also inspired, and made feasible, the remaining chapters of this thesis.

Chapter 5 tackles the problem of performing regression on graph-structured inputs such as organic molecules. We take a standard feature representation for molecules, the Morgan Circular Fingerprint, and make it differentiable. This allows us to learn, using gradients, a data-driven feature representation tuned to a particular prediction problem. This project was a collaboration with Alán Aspuru-Guzik's group in the Harvard Chemistry Department.

In Chapter 6 we tackle the problem of hyperparameter optimization. Hyperparameters of machine learning algorithms, such as learning rates and regularization parameters, are conventionally tuned using black-box gradient-free optimization of the validation loss. But, since validation loss is ultimately just a function we've expressed as a computer program (albeit an elaborate one that may involve a lengthy optimization as an inner loop) we ought to be able to optimize it using gradients computed with Autograd. There is a technical challenge related to memory consumption, which we solve with "reversible learning", recomputing the optimization trajectory during the reverse pass.

This "reversible learning" has interesting connections to entropy. Optimization, as a many-to-few mapping, is an intrinsically entropy-removing process. Thus, we can't perfectly reverse learning unless we account for the entropy removed in the process. We do this in Chapter 6 with a coding scheme similar to arithmetic coding. But in Chapter 7, we actually take advantage of this entropy accounting, using it to estimate the variational free energy of unconverged optimization. This allows us to give a theoretical account of "early stopping", a popular and empirically effective trick for preventing overfitting.

# 2

# Background

## 2.1 Bayesian inference

The Bayesian approach to modeling data treats observed data, $D$, and model parameters, $\theta$, as random variables, and assumes, *a priori*, a particular joint distribution $p(D, \theta)$, which we usually factorize into a prior, $p(\theta)$ and a likelihood, $p(D|\theta)$. Inferring the model parameters is just a matter of conditioning on the data to obtain the posterior distribution $p(\theta|D)$. Indeed, Bayes' celebrated rule is merely a statement about conditional probabilities:

$$p(\theta|D) = \frac{p(D, \theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{p(D)}. \tag{2.1}$$

Bayesian modeling offers conceptual clarity, a principled methodology for model selection, a generative framework that separates modeling and inference, and well-calibrated uncertainties for prediction and control. Examples of models include Kalman filters, hidden Markov models, latent Dirichlet allocation, Gaussian processes, Markov random fields, probabilistic context-free grammars and many more.

Unfortunately, computing quantities of interest usually requires intractable integrals. These include expectations under the posterior or predictions about new data marginalizing out the model parameters,

$$\mathbb{E}_{p(\theta|D)}\left[f(\theta)\right] = \int d\theta f(\theta) p(\theta|D) \tag{2.2}$$

as well as the marginal likelihood,

$$p(D) = \int d\theta \, p(D, \theta) \tag{2.3}$$

which is required to even evaluate $p(\theta|D)$ and is useful in its own right as a tool for model selection. In special cases, these integrals can have closed-form analytic solutions, and in very low dimensions we can use direct numerical integration (the cost scales exponentially with dimension [81]) but in general we must resort to posterior approximation.

The next three sections describe common approaches to posterior approximation: approximation by a single parameter value (point estimation, Section 2.2), approximation by tractable distributions (variational inference, Section 2.3), and approximation by samples generated from a Markov chain (Markov chain Monte Carlo, Section 2.4).

These approximation methods apply to any distribution for which we can evaluate

the density (and perhaps gradients) pointwise, up to a multiplicative constant. We'll write this unnormalized density as $\tilde{p}(x)$ and the normalized density as $p(x) = \tilde{p}(x)/Z$ where $Z$ is the (unknown) normalization constant.

Variational inference and Markov chain Monte Carlo both originated in statistical physics and it can help to keep the physical interpretation in mind. The canonical ensemble is the archetypal statistical mechanical system, a distribution over the state, $x$, of a system in thermal equilibrium with a heat bath:

$$p(x) = \frac{1}{Z} \exp\left[-E(x)/(k_B T)\right] \tag{2.4}$$

where $E(x)$ is the energy of the state, $T$ is the temperature, $k_B$ is the Boltzmann constant, and $Z$ is the normalizing constant or partition function. As with Bayesian inference, we have now a distribution of interest with an unnormalized density that is easy to evaluate, but a log normalizer, $Z$, that demands a potentially intractable integral. In Bayesian modeling we call $Z$ the marginal likelihood; in physics we call it the partition function. In Bayesian modeling we refer to $\log p(\theta, D)$ as the unnormalized log posterior or log joint density; in physics this is just the (negative) energy (we will usually ignore $k_B T$, although some Bayesian methods, such as annealed importance sampling, make use of it).

## 2.2 Point estimation

The simplest possible approximation to the posterior is to use a single point, most commonly the point of maximum posterior density, knows at the MAP (maximum *a posteriori*) estimate. The MAP estimate can be found with conventional optimization techniques (though non-convexity can make it hard) and it can be a fair approximation,

particularly if the posterior is sharply peaked. One common criticism of MAP estimation is that it is not invariant to reparameterization [63]. It's worth noting that MAP inference can be viewed as an instance of variational inference, using a delta function variational family.

## 2.3 Variational inference

Another approach is to approximate the intractable distribution $p(x)$ with a tractable one. Given a family of distributions, $q_\phi(x)$, parameterized by $\phi$ (for example, Gaussians parameterized by their mean and covariance) and some measure of how well $q_\phi$ approximates $p$, we can frame the problem as an optimization over $\phi$. This general approach is known as variational inference.

A common measure of the goodness of the approximation is the Kullback-Liebler (KL) divergence from $q_\phi$, the approximating distribution, to $p$, the true distribution[1]:

$$\begin{aligned} \mathrm{KL}\left(q_\phi \| p\right) &= \mathbb{E}_{q_\phi}\left[\log q_\phi(x) - \log p(x)\right] \\ &= \mathbb{E}_{q_\phi}\left[\log q_\phi(x) - \log \tilde{p}(x)\right] + \log Z \end{aligned} \tag{2.5}$$

where the expectation is taken with respect to the tractable distribution $q_\phi$.

Since the KL divergence is always nonnegative, we can rewrite Equation 2.5 as:

$$\begin{aligned} \log Z &\geq \underbrace{\mathbb{E}_{q_\phi}\left[-\log q_\phi(x)\right]}_{\text{entropy}} + \underbrace{\mathbb{E}_{q_\phi}\left[\log \tilde{p}(x)\right]}_{\text{-energy}} \\ &\equiv \mathcal{L}(\phi) \end{aligned} \tag{2.6}$$

[1] We don't use the other KL divergence, $\mathrm{KL}\left(p \| q_\phi\right)$, because evaluating it requires computing an expectation with respect to $p$, which is exactly the hard task we are trying to solve.

where $\mathcal{L}$ is known as the variational lower bound. It is a lower bound on the normalizer, or marginal likelihood, $Z$. Maximizing this lower bound with respect to $\phi$ amounts to minimizing $\mathrm{KL}\,(q_\phi \| p)$ and thus optimizes the quality of $q_\phi$ as a posterior approximation. As a bonus, we can use the value of $\mathcal{L}$ as an approximation to the marginal likelihood for other purposes such as model selection.

Notice that $\mathcal{L}$ consists of an energy-like term and an entropy-like term, as labeled in Equation 2.6. Maximizing the (negative) energy term encourages $q_\phi$ to put its probability mass in regions of high $\log \tilde{p}$ (i.e., low energy), while maximizing the entropy term encourages $q_\phi$ to spread out its mass. Combined, the two terms behave like the Helmholtz free energy, trading off entropy and energy. In statistical physics, the quantity $\mathcal{L}$ is known as the (negative) variational free energy.

How we actually approach the minimization depends on the structure of $q$. A very general approach, known as black-box stochastic variational inference (BBSVI) [55] requires only that we be able to sample from $q$ and evaluate its log-density. We can then make a Monte Carlo[2] estimate of the expectations in Equation 2.6 using samples from $q$.

$$\mathcal{L}(\phi) \approx \log q_\phi(x) - \log \tilde{p}(x), \qquad x \sim q_\phi \tag{2.7}$$

In principle, having access to unbiased estimates of $\mathcal{L}(\phi)$ is sufficient, but having access to unbiased estimates of the gradient $\nabla_\phi \mathcal{L}$ is extremely helpful, allowing us to use stochastic gradient optimizers.

To obtain gradients, we need to differentiate through a procedure for sampling from $q_\phi$. One approach is to sample from $q$ by warping (in a $\phi$-dependent way) a sample

[2] Not to be confused with Markov chain Monte Carlo, described in the next section.

drawn from a fixed distribution, $q_0$:

$$x = T(\phi, z), \quad z \sim q_0 \tag{2.8}$$

$$\Rightarrow \qquad \mathcal{L}(\phi) \approx \log q_\phi(T(\phi, z)) - \log \tilde{p}(T(\phi, z)) \tag{2.9}$$

For example, if $q$ is a Gaussian parameterized by its mean and covariance matrix, we can sample from $q$ by drawing a sample from a standard normal, multiplying by the Cholesky decomposition (or other matrix square root) of the covariance matrix and adding the mean.

The performance of variational inference depends on the flexibility of the model class, $q_\phi$, and how well it is able to approximate $p$. Figure 2.1 shows variational approximations with different approximating families.

Variational inference has a long history in statistical physics. Mean field theories (factored approximations) are the classic example (see any statistical mechanics textbook, e.g. Pathria & Beale [76]), and these go back to Pierre Curie's theories on ferromagnetism in the late 19th century. The usual reference for variational inference in machine learning is Wainwright & Jordan [109]. For a very recent review, see Blei et al. [15]. The more general "black-box" variational inference approach treated here was introduced surprisingly recently, by Ranganath et al. [83] and Kucukelbir et al. [55].

## 2.4 Markov chain Monte Carlo inference

Markov chain Monte Carlo (MCMC) is a general procedure for generating approximate samples from a distribution. Given a distribution of interest, $p(x)$, the idea of MCMC is to generate a chain of states, $\{x_1, x_2, ...\}$, such that the distribution of $x_t$ converges to $p$ as $t$ becomes large. The chain satisfies the Markov property: each state only depends

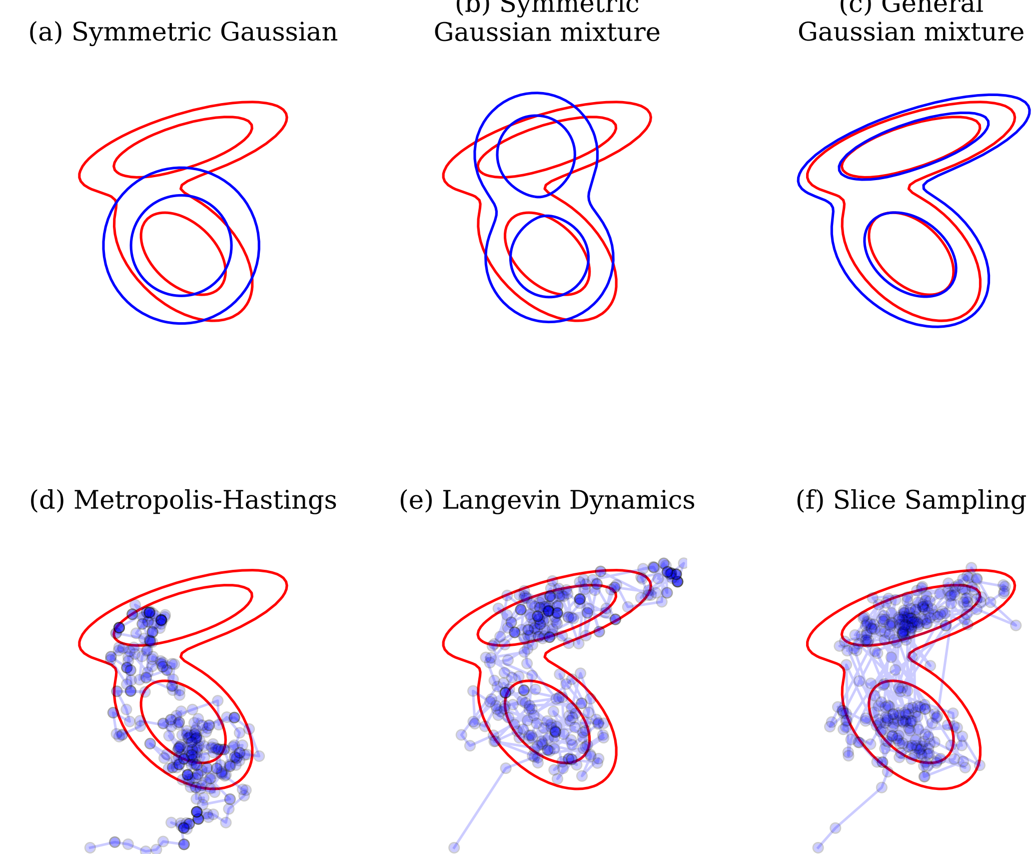



**Figure 2.1:** Illustration of posterior approximation methods. *Top row:* Variational inference, with approximating distributions consisting of (a) a symmetric Gaussian parameterized by its mean and scale (b) a mixture of two symmetric Gaussians, parameterized by their means and scales (c) a mixture of two general Gaussians, parameterized by means and covariance matrices. In this final case we obtain a near-perfect fit, since the target distribution is contained within the class of approximations. (We used a stochastic optimizer so the fit isn't truly perfect.) *Bottom row:* MCMC using (d) Metropolis-Hastings with Gaussian proposals, (e) Metropolis-adjusted Langevin algorithm and (f) slice sampling. Notice that the algorithms differ in their behavior during burn-in and in mixing between the two modes.

on the previous one and not on the rest of the prior history. We can characterize the chain by its transition probability desnsity, $T(x'; x)$, the probability (per unit volume)

of transitioning to a new state $x'$, conditioned on the current state $x$. There are two conditions which together are sufficient to ensure that $x_t$ converges to $p$. The first is that $p$ is an invariant, or stationary, distribution of $T$. That is:

$$\int dx_t \, p(x_t) T(x_{t+1}; x_t) = p(x_{t+1}). \tag{2.10}$$

The second condition is that the chain is ergodic, which roughly means that the entire state space is accessible to the chain. See Meyn & Tweedie [68] for more details.

Since integrating over state space as in Equation 2.10 is often intractable (that's why we're doing MCMC in the first place) we usually prove that $p$ is an invariant distribution of $T$ by satisfying the stronger condition of detailed balance:

$$T(x_b; x_a)p(x_a) = T(x_a; x_b)p(x_b) \tag{2.11}$$

for all $x_a$, $x_b$. That is, when $x$ is distributed according to $p$ and the transition operator $T$ is applied, the number of states transitioning from (a differential region around) $x_a$ to (a differential region around) $x_b$ is the same as the number going the other way, from $x_b$ to $x_a$. Thus the distribution of states remains the same. We now look at some popular transition kernels, illustrated in Figure 2.1.

The *Metropolis-Hastings* [67, 38] transition operator uses a proposal distribution, $q(x'; x)$, (for example, a Gaussian with mean $x$) to propose a new state $x'$ given a current state $x$. In order to satisfy detailed balance, the proposal may be rejected, with rejection probability

$$p_{\text{reject}} = 1 - \min\left\{\frac{p(x')q(x; x')}{p(x)q(x'; x)}, 1\right\}. \tag{2.12}$$

Since the proposal distribution knows nothing about the target distribution, $p$, Metropolis-Hastings MCMC tends to lead to slow diffusion and long mixing times.

The *slice sampling* [72] transition operator works by introducing an auxiliary random variable, $z \in \mathbb{R}^+$, which augments the state space to give a new target distribution:

$$p(z, x) = p(z|x)p(x), \qquad \text{where} \quad z \sim \text{Uniform}(0, p(x)) \tag{2.13}$$

Introducing $z$ does not affect the marginal distribution of $x$, so if we draw samples from this new augmented distribution and ignore the $z$s, we will have samples of $x$ from $p(x)$ as we require. This auxiliary variable trick is very common in designing MCMC algorithms, and we use it to develop the Firefly Monte Carlo algorithm in Chapter 3. In this augmented space, we proceed by alternately sampling $z$ from $p(z|x)$ and $x$ from $p(x|z)$. Thanks to our choice of conditional $p(z|x)$, the conditional distribution of $x$ is the uniform distribution over the set of points where $p(x) > z$, which is easy to sample from along a particular direction, although it requires a few additional tricks.

The *Hamiltonian Monte Carlo* [24, 63] algorithm (HMC) draws inspiration from Hamiltonian mechanics. The transition kernel simulates the dynamics of a massive particle with potential energy $\log p(x)$. As with slice sampling, we introduce an auxiliary variable, $z$, which plays the role of momentum. $z$ has the same number of dimensions as $x$ and has a standard Gaussian distribution, independent of $x$. Perfect simulation of Hamiltonian dynamics leaves $p(x, z)$ invariant but it is not ergodic, since the total energy (kinetic plus potential) never changes. To make a valid MCMC algorithm we could simulate Hamiltonian dynamics for a fixed length of time, then resample the momentum variables (analogous to equilibrating velocities with the thermal bath). Of course, in practice, perfect simulation is impossible. Instead, we simulate approximately

by numerically integrating Hamilton's equations. We use the leapfrog integrator because it is symmetric and symplectic and we correct for any changes in total energy using an accept-reject step in the spirit of the Metropolis-Hastings algorithm. The special case of HMC with only one leapfrog step is known as the Metropolis-Adjusted Langevin algorithm [88, 91]. It describes dynamics dominated by friction rather than inertia and its transition operator looks like gradient descent with added noise.

In contrast to the Victorian origins of variational inference, MCMC wasn't developed until the second half of the 20th century. Sampling-based methods require substantial computational resources and it's no coincidence that the physicist credited with inventing MCMC, Nicholas Metropolis, also designed and built the MANIAC machine at Los Alamos, one of the earliest electronic (rather than electromechanical) computers [3]. Metropolis' 1953 algorithm computed the integrals required for calculating equations of state from two-body interaction potentials. Decades later, Gelfand & Smith [31] popularized MCMC for Bayesian inference and it has been a popular tool in statistics and machine learning ever since [87].

## 2.5 Computing gradients in reverse accumulation mode

Continuous optimization and inference problems become far easier if we have access to the gradient of the objective function or log probability. This is particularly true for functions in high dimensions, $\mathbb{R}^D \to \mathbb{R}$, since each gradient evaluation is equivalent to making $D$ additional function evaluations. With the right evaluation strategy, gradients can be very cheap to compute: the time cost is only a small constant factor more than the cost of evaluating the function itself. In this section, we explain this evaluation strategy, known as *reverse (accumulation) mode differentiation* or, in the neural network community, *backpropagation*.

Given a vector-to-scalar function, $\mathbb{R}^D \to \mathbb{R}$, composed of a set of primitive functions $\mathbb{R}^M \to \mathbb{R}^N$ (for various $M$, $N$) with known Jacobians, the gradient of the composition is given by the product of the Jacobians of the primitive functions, according to the chain rule[3]. But the chain rule doesn't prescribe the *order* in which to multiply the Jacobians. From the perspective of computational complexity, the order makes all the difference.

To be concrete, consider $F : \mathbb{R}^D \to \mathbb{R}$ defined as the composition of four primitive functions, $F = D \circ C \circ B \circ A$. We break the function down so that we can refer to intermediate values:

$$
\begin{aligned}
& F(\mathbf{x}) = y && \left(\mathbf{x} \in \mathbb{R}^D, \quad y \in \mathbb{R}\right) \\
& \text{where} \quad y = D(\mathbf{c}), \quad \mathbf{c} = C(\mathbf{b}), \quad \mathbf{b} = B(\mathbf{a}), \quad \mathbf{a} = A(\mathbf{x}) && (2.14)
\end{aligned}
$$

The gradient (or Jacobian[4]) of $F$, $F'$, is then given by

$$
\begin{aligned}
& F'(\mathbf{x}) = \frac{\partial y}{\partial \mathbf{x}} \\
& \text{where} \quad \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{c}} \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}} && (2.15) \\
& \frac{\partial y}{\partial \mathbf{c}} = D'(\mathbf{c}), \quad \frac{\partial \mathbf{c}}{\partial \mathbf{b}} = C'(\mathbf{b}), \quad \frac{\partial \mathbf{b}}{\partial \mathbf{a}} = B'(\mathbf{a}), \quad \frac{\partial \mathbf{a}}{\partial \mathbf{x}} = A'(\mathbf{x}) && (2.16)
\end{aligned}
$$

where $A'$, $B'$, $C'$ and $D'$ are functions that compute the Jacobians of $A$, $B$, $C$, and $D$. We will always use $\mathbf{x} \in \mathbb{R}^D$ as the input and $y \in \mathbb{R}$ as the output. Note that $y$ is a

[3] Why don't we mention the product and quotient rules? These rules are just definitions of the derivatives of "multiply" and "divide". Besides having a different arity, these functions are in the same category as other primitives like "sin" and "exp". The chain rule stands apart. It relates to "compose", which is a higher-order function, taking functions as input and returning a new function as output.

[4] We'll use "Jacobian" and "gradient" interchangeably for functions $\mathbb{R}^D \to \mathbb{R}$, which means treating the gradient as a row vector. This is a nonstandard convention.

scalar while $\mathbf{x}$ is a (possibly enormous) vector.

Since matrix multiplication is associative, we can evaluate the product of Jacobians in Equation 2.15 in any order we choose. Evaluation starting from the left, we call "reverse accumulation mode"; evaluation starting from the right, we call "forward accumulation mode":

$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{c}} \left( \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \underbrace{\left( \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}} \right)}_{\frac{\partial \mathbf{b}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_N}{\partial x_1} & \cdots & \frac{\partial b_N}{\partial x_D} \end{bmatrix}} \right) \qquad \text{Forward accumulation mode} \tag{2.17}$$

$$\frac{\partial y}{\partial \mathbf{x}} = \left( \underbrace{\left( \frac{\partial y}{\partial \mathbf{c}} \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \right)}_{\frac{\partial y}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_N} \end{bmatrix}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \right) \frac{\partial \mathbf{a}}{\partial \mathbf{x}} \qquad \text{Reverse accumulation mode} \tag{2.18}$$

Notice the dramatic difference in the size of the intermediate values computed. In forward mode, these are Jacobians like $\frac{\partial \mathbf{b}}{\partial \mathbf{x}}$. Since $\mathbf{x} \in \mathbb{R}^D$ is a vector, this contains $D$ times as many entries as the corresponding value $b$. In reverse mode, we compute values like $\frac{\partial y}{\partial \mathbf{b}}$. Since $y \in \mathbb{R}$ is a scalar, this contains only as many values as $\mathbf{b}$.

Reverse mode is therefore the more efficient way to evaluate the gradient of a vector-to-scalar function, once we've evaluated the primitive Jacobians $A'(\mathbf{x})$, $B'(\mathbf{a})$, $C'(\mathbf{b})$, and $D'(\mathbf{c})$. But we can do even better: we don't even need to evaluate the primitive Jacobians in the first place. The Jacobians are often extremely sparse and we only ever use them in matrix products. Matrices, after all, are just representations of linear maps,

so rather than instantiate them, we can just directly implement functions that apply the linear maps. That is, for each primitive function, $A : \mathbb{R}^M \to \mathbb{R}^N$, with Jacobian $A' : \mathbb{R}^M \to \mathbb{R}^{N \times M}$ we can define the left-multiplying Jacobian-vector product function (JVP), $J_A : \mathbb{R}^M \to (\mathbb{R}^M \to \mathbb{R}^N)$ as (currying[5])

$$J_A(\mathbf{x}, \mathbf{g}) = A'(\mathbf{x})\mathbf{g} \tag{2.19}$$

and the right-multiplying vector-Jacobian product function (VJP), $J_A^T : \mathbb{R}^M \to (\mathbb{R}^N \to \mathbb{R}^M)$ as

$$J_A^T(\mathbf{x}, \mathbf{g}) = \mathbf{g}A'(\mathbf{x}) \tag{2.20}$$

As an example, consider a function that squares a vector elementwise:

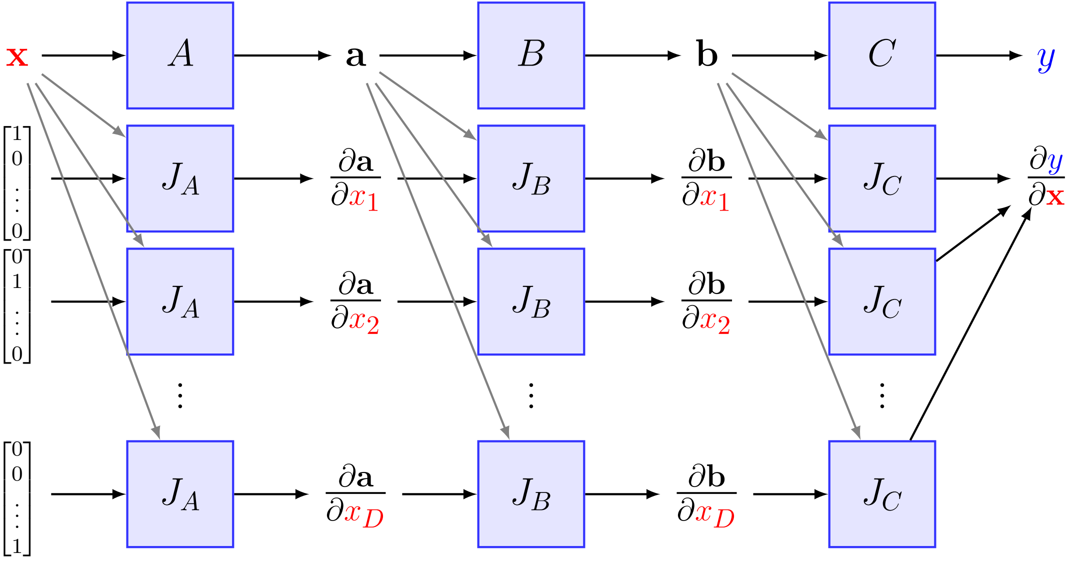
$$\text{ElemSquare}(\mathbf{x}) = \mathbf{x} \odot \mathbf{x} \tag{2.21}$$

where we've used the symbol $\odot$ to denote elementwise multiplication. ElemSquare has a very sparse Jacobian: it's just a matrix with $2x$ on the diagonal and zeros elsewhere. The VJP function is given by

$$J^T_{\text{ElemSquare}}(\mathbf{x}, \mathbf{g}) = 2\mathbf{g} \odot \mathbf{x} \tag{2.22}$$

Since the Jacobian is symmetric, the left-multiplying JVP function $J_{\text{ElemSquare}}$ is the same.

[5] Currying (after Haskell Curry) means we formally only allow functions of a single argument and we simulate multi-argument functions using functions that return functions. Thus, a function with signature $F : X \to (Y \to Z)$ can be applied like $z = F(x)(y)$ (where $x \in X$, $y \in Y$, $z \in Z$). We usually drop the internal parens for convenience, writing just $z = F(x, y)$.



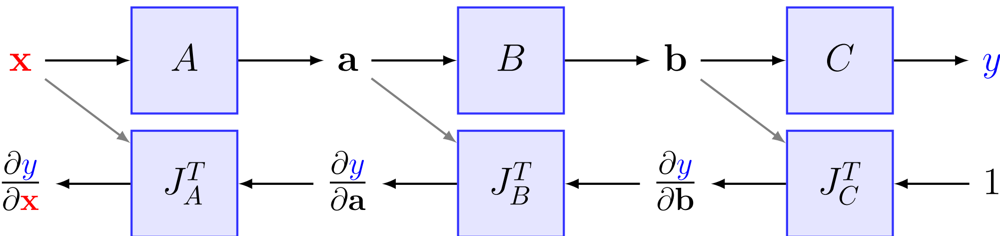




**Figure 2.2:** Illustration of the difference between forward- and reverse-mode differentiation of a composite function $F : \mathbb{R}^D \to \mathbb{R}$, $F = C \circ B \circ A$. Forward-mode accumulates values such as $\frac{\partial \mathbf{a}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{b}}{\partial \mathbf{x}}$, Jacobians of each of the intermediate variables with respect to the *input*, $\mathbf{x}$. It does this by left-multiplying the previous step's accumulated Jacobian by the current primitive Jacobian. Reverse-mode works by multiplying Jacobians in the other direction. It accumulates values such as $\frac{\partial y}{\partial \mathbf{a}}$, $\frac{\partial y}{\partial \mathbf{b}}$, Jacobians of the *output*, $y$, with respect to each of the intermediate variables, by right-multiplying the previous step's accumulated Jacobian by the current primitive Jacobian. If $y \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^D$, with $D$ large, the values accumulated in reverse-mode are a factor of $D$ smaller and the computation is far more efficient.

We can now implement both forward and reverse mode differentiation by chaining JVPs, as shown in Figure 2.2. In forward mode, we apply a JVP for each input dimension

at each step, whereas in reverse mode we only apply a VJP once at each step. Evaluating a JVP or a VJP is usually only a small constant factor (1 - 3) slower than evaluating the primitive function itself[6].

Forward mode differentiation is therefore a factor of $D$ (where $\mathbf{x} \in \mathbb{R}^D$) slower than reverse-mode differentiation, and reverse-mode differentiation is only a small constant factor slower than evaluating the composite function itself. It should be noted that reverse mode differentiation has one major drawback: since we must do a complete forward pass to compute the intermediate values before applying the JVPs on the reverse pass, we need to store all the intermediate values in memory. This can sometimes be prohibitive, as we will discover in Chapter 6.

We have described how to use reverse accumulation mode differentiation to efficiently compute the gradient of a composed *chain* of primitive functions, but in general, a composite function can be described as a directed acyclic *graph* of primitive functions. Luckily, the strategy for chains generalizes quite straightforwardly to graphs. As with chains, we do a complete forward pass to evaluate the function, storing all the intermediate values. Then we traverse the graph in reverse, applying Jacobian-vector products to compute $\frac{\partial y}{\partial \mathbf{z}}$ for each intermediate value $\mathbf{z}$.

There are additional cases we need to handle which don't arise with chain composition: "fan-out", in which a value is used more than once, and "fan-in", in which a function takes multiple inputs. We handle fan-in by defining Jacobian-vector product functions for each of the function's arguments. We handle fan-out, reuse of a variable

[6] This is not an absolute law, but we have not found any violations in practice. It holds for the usual scalar primitives, things like arithmetic operations and trigonometric functions. In the implementation of Autograd (see Chapter 4) we treat higher-level functions as primitive too, functions like matrix multiplication or even Cholesky decomposition. But, since these can be expressed in terms of lower-level primitives, we can always use hand-coded forward- or reverse-mode differentiation to implement their JVP and VJP functions respectively in terms of the JVPs and VJPs of the underlying primitives.

$\mathbf{z}$, by computing $\frac{\partial y}{\partial \mathbf{z}}^{(i)}$ for each branch, $i$, that makes use of $\mathbf{z}$ and summing the results to yield the complete $\frac{\partial y}{\partial \mathbf{z}}$. This puts a constraint on the order in which the graph is traversed, since all the $\frac{\partial y}{\partial \mathbf{z}}^{(i)}$ must be available before continuing. Both these cases are illustrated in Figure 2.3.

Reverse mode differentiation has been independently discovered many times in various quantitative fields [9], and even several times within machine learning [114]. The most famous (re)invention of "backpropagation" is Rumelhart et al. [93]. My own understanding has been shaped heavily by the work of Pearlmutter and Siskind, e.g. [79].

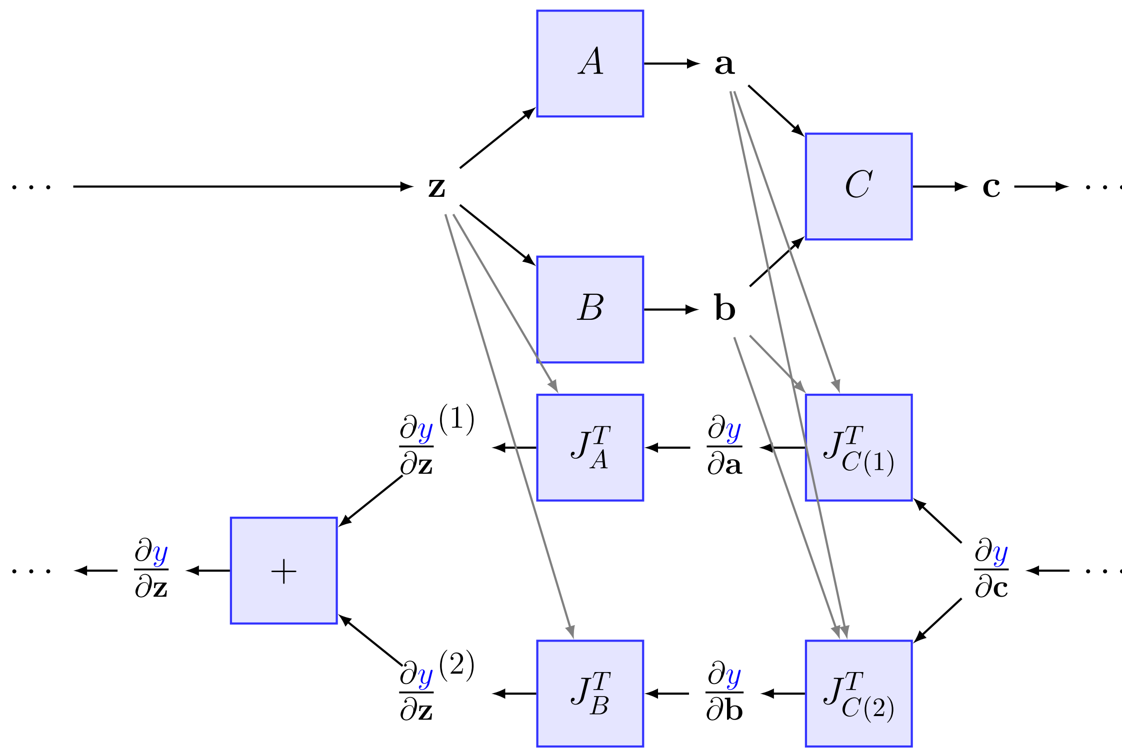



**Figure 2.3:** Illustration of reverse accumulation mode for a computation graph, showing how to handle fan-out (reuse of the value $\mathbf{z}$) and fan-in (the two arguments feeding into function $C$).

# 3

# Firefly Monte Carlo

In this chapter, we present *Firefly Monte Carlo* (Firefly), an auxiliary variable MCMC algorithm that only queries the likelihoods of a potentially small subset of the data at each iteration yet simulates from the exact posterior distribution. Firefly is compatible with a wide variety of modern MCMC algorithms, and only requires a lower bound on the per-datum likelihood factors. In experiments, we find that Firefly generates samples from the posterior more than an order of magnitude faster than regular MCMC, opening up MCMC methods to larger datasets than were previously considered feasible.

This chapter presents work with with Ryan P. Adams. It appeared at UAI 2014 [64] where it won best paper.

## 3.1 Introduction

A common criticism of MCMC for Bayesian inference is that is doesn't scale well to large data sets. The algorithms described in Chapter 2 require evaluating the complete unnormalized posterior at each iteration, which will usually involve inspecting every datum that contributes to the likelihood. This is in contrast to optimization, where people commonly use stochastic gradient methods, which use a subsample of the data to estimate the gradient.

Recent work has shown that approximate transition operators based on subsets of data can be used for predictive prefetching to help parallelize MCMC [5]. Other work uses approximate transition operators directly, for Metropolis-Hastings (MH) and related algorithms [113]. Korattikara et al. [52] and Bardenet et al. [7] have shown that such approximate MH moves can lead to stationary distributions which are approximate but that have bounded error, albeit under strong conditions of rapid mixing.

The *Firefly Monte Carlo* (Firefly) algorithm is in line with these latter efforts to exploit subsets of data to construct transition operators. What distinguishes the approach we present here, however, is that this new MCMC procedure is *exact* in the sense that it leaves the true full-data posterior distribution invariant. Firefly is a latent variable model which introduces a collection of Bernoulli variables - one for each datum – with conditional distributions chosen so that they effectively turn on and off data points in the posterior, hence "firefly". The introduction of these latent variables does not alter the marginal distribution of the parameters of interest. Our only requirement is that it be possible to provide a "collapsible" lower bound for each likelihood term. Firefly can lead to dramatic performance improvements in MCMC, as measured in wallclock time.

## 3.2 Firefly Monte Carlo

The Firefly Monte Carlo algorithm tackles the problem of sampling from the posterior distribution of a probabilistic model. We will denote the parameters of interest as $\theta$ and assume that they have prior $p(\theta)$. We assume that $N$ data have been observed $\{x_n\}_{n=1}^N$ and that these data are conditionally independent given $\theta$ under a likelihood $p(x_n \,|\, \theta)$. Our target distribution is therefore

$$p(\theta \,|\, \{x_n\}_{n=1}^N) \propto p(\theta, \{x_n\}_{n=1}^N) = p(\theta) \prod_{n=1}^N p(x_n|\theta). \tag{3.1}$$

For notational convenience, we will write the $n$th likelihood factor as a function of $\theta$ as

$$L_n(\theta) = p(x_n \,|\, \theta) \, .$$

An MCMC sampler makes transitions from a given $\theta$ to a new $\theta'$ such that posterior distribution remains invariant. Conventional algorithms, such as Metropolis–Hastings, require evaluation of the unnormalized posterior in full at every iteration. When the data set is large, evaluating all $N$ likelihoods is a computational bottleneck. This is the problem that we seek to solve with Firefly.

For each data point, $n$, we introduce a binary auxiliary variable, $z_n \in \{0, 1\}$, and a function $B_n(\theta)$ which is a strictly positive lower bound on the $n$th likelihood: $0 < B_n(\theta) \leq L_n(\theta)$. Each $z_n$ has the following Bernoulli distribution conditioned on the parameters:

$$p(z_n \,|\, x_n, \theta) = \left[ \frac{L_n(\theta) - B_n(\theta)}{L_n(\theta)} \right]^{z_n} \left[ \frac{B_n(\theta)}{L_n(\theta)} \right]^{1-z_n} .$$

We now augment the posterior distribution with these $N$ variables:

$$
\begin{aligned}
p(\theta, \{z_n\}_{n=1}^N \,|\, \{x_n\}_{n=1}^N) &\propto p(\theta, \{x_n, z_n\}_{n=1}^N) \\
&= p(\theta) \prod_{n=1}^N p(x_n \,|\, \theta)\, p(z_n \,|\, x_n, \theta)\,.
\end{aligned}
$$

As in other auxiliary variable methods such as slice sampling, Swendsen-Wang, or Hamiltonian Monte Carlo, augmenting the joint distribution in this way does not damage the original marginal distribution of interest:

$$
\begin{aligned}
&\sum_{z_1} \cdots \sum_{z_N} p(\theta) \prod_{n=1}^N p(x_n \,|\, \theta)\, p(z_n \,|\, x_n, \theta) \\
&= p(\theta) \prod_{n=1}^N p(x_n \,|\, \theta) \sum_{z_n} p(z_n \,|\, x_n, \theta) \\
&= p(\theta) \prod_{n=1}^N p(x_n \,|\, \theta).
\end{aligned}
$$

However, this joint distribution has a remarkable property: to evaluate the probability density over $\theta$, given a particular configuration of $\{z_n\}_{n=1}^N$, it is only necessary to evaluate those likelihood terms for which $z_n = 1$. Consider factor $n$ from the product above:

$$
\begin{aligned}
&p(x_n \,|\, \theta) p(z_n \,|\, x_n, \theta) \\
&= L_n(\theta) \left[ \frac{L_n(\theta) - B_n(\theta)}{L_n(\theta)} \right]^{z_n} \left[ \frac{B_n(\theta)}{L_n(\theta)} \right]^{1-z_n} \\
&= \begin{cases} L_n(\theta) - B_n(\theta) & \text{if } z_n = 1 \\ B_n(\theta) & \text{if } z_n = 0 \end{cases} .
\end{aligned}
$$

The "true" likelihood term $L_n(\theta)$ only appears in those factors for which $z_n = 1$ and we can think of these data as forming a "minibatch" subsample of the full set. If most $z_n = 0$, then transition updates for the parameters will be much cheaper, as these are applied to $p(\theta \,|\, \{x_n, z_n\}_{n=1}^N)$.

Of course, we do have to evaluate all $N$ bounds $B_n(\theta)$ at each iteration. At first glance, we seem to have just shifted the computational burden from evaluating the $L_n(\theta)$ to evaluating the $B_n(\theta)$. However, if we choose $B_n(\theta)$ to have a convenient form, a scaled Gaussian or other exponential family distribution, for example, then the full product $\prod_{n=1}^N B_n(\theta)$ can be computed for each new $\theta$ in $O(1)$ time using the sufficient statistics of the distribution, which only need to be computed once. To make this clearer, we can rearrange the joint distribution in terms of a "pseudo-prior," $\tilde{p}(\theta)$ and "pseudo-likelihood," $\tilde{L}_n(\theta)$ as follows:

$$p(\theta, \{z_n\}_{n=1}^N \,|\, \{x_n\}_{n=1}^N) \propto \tilde{p}(\theta) \prod_{n: z_n = 1} \tilde{L}_n(\theta) \tag{3.2}$$

where the product only runs over those $n$ for which $z_n = 1$, and we have defined

$$\tilde{p}(\theta) = p(\theta) \prod_{n=1}^N B_n(\theta) \qquad\qquad \tilde{L}_n(\theta) = \frac{L_n(\theta) - B_n(\theta)}{B_n(\theta)} \,.$$

We can generate a Markov chain for the joint distribution in (3.2) by alternating between updates of $\theta$ conditional on $\{z_n\}_{n=1}^N$, which can be done with any conventional MCMC algorithm, and updates of $\{z_n\}_{n=1}^N$ conditional on $\theta$ for which we discuss efficient methods in Section 3.3.2. We emphasize that the marginal distribution over $\theta$ is still the correct posterior distribution given in (3.1).

At a given iteration, the $z_n = 0$ data points are "dark": we simulate the Markov chain

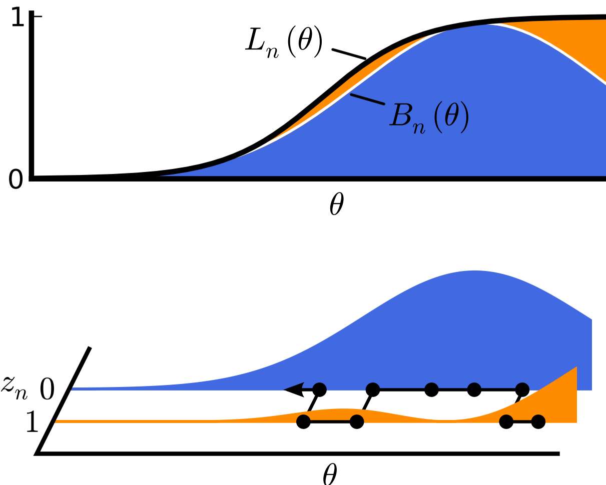



**Figure 3.1:** Illustration of the auxiliary variable representation of a single likelihood for a one-dimensional logistic regression model. The top panel shows how the likelihood function, $L_n(\theta)$, corresponding to a single datum $n$, can be partitioned into two parts: a lower bound, $B_n(\theta)$, shaded blue, and the remainder, shaded orange. The bottom panel shows that we can introduce a Bernoulli random variable $z_n$ and construct a Markov chain in this new, higher dimensional space, such that marginalizing out (i.e. ignoring) the $z_n$ recovers the original likelihood. If $B_n(\theta) \gg L_n(\theta) - B_n(\theta)$, the Markov chain will tend to occupy $z_n = 0$ and we can avoid evaluating $L_n(\theta)$ at each iteration.

without computing their likelihoods. Upon a Markov transition in the space of $\{z_n\}_{n=1}^N$, a smattering of these dark data points become "bright" with their $z_n = 1$, and we include their likelihoods in subsequent iterations. The evolution of the chain evokes an image of fireflies, as the individual data blink on and off due to updates of the $z_n$.

The details of choosing a lower bound and efficiently sampling the $\{z_n\}$ are treated in the proceeding sections, but the high-level picture is now complete. Figure 3.1 illustrates the augmented space, and a simple version of the algorithm is shown in Algorithm 1. Figure 3.2 shows several steps of Firefly Monte Carlo on a toy logistic regression model.

### 3.3 Implementation considerations

In this section we discuss two important practical matters for implementing an effective Firefly algorithm: how to choose and compute lower bounds, and how to sample the brightness variables $z_n$. For this discussion we will assume that we are dealing with a data set consisting of $N$ data points, and a parameter set, $\theta$, of dimension $D \ll N$. We will also assume that it takes at least $O(ND)$ time to evaluate the likelihoods at some $\theta$ for the whole data set and that evaluating this set of likelihoods at each iteration is the computational bottleneck for MCMC. We will mostly assume that space is not an issue: we can hold the full data set in memory and we can afford additional data structures occupying a few bytes for each of the $N$ data.

The goal of an effective implementation of Firefly is to construct a Markov chain with similar convergence and mixing properties to that of regular MCMC, while only evaluating a subset of the data points on average at each iteration. If the average number of "bright" data points is $M$, we would like this to achieve a computational speedup of nearly $N/M$ over regular MCMC.

#### 3.3.1 Choosing a lower bound

The lower bounds, $B_n(\theta)$ of each data point's likelihood $L_n(\theta)$ should satisfy two properties. They should be relatively tight, and it should be possible to efficiently summarize a product of lower bounds $\prod_n B_n(\theta)$ in a way that (after setup) can be evaluated in time independent of $N$.

The tightness of the bounds is important because it determines the number of bright data points at each iteration, which determines the time it takes to evaluate the joint

posterior. For a burned-in chain, the average number of bright data points, $M$, will be:

$$M = \sum_{n=1}^{N} \langle z_n \rangle = \sum_{n=1}^{N} \int p(\theta \,|\, \{x_n\}_{n=1}^{N}) \frac{L_n(\theta) - B_n(\theta)}{L_n(\theta)} \mathrm{d}\theta \,.$$

Therefore it is important that the bounds are tight at values of $\theta$ where the posterior puts the bulk of its mass.

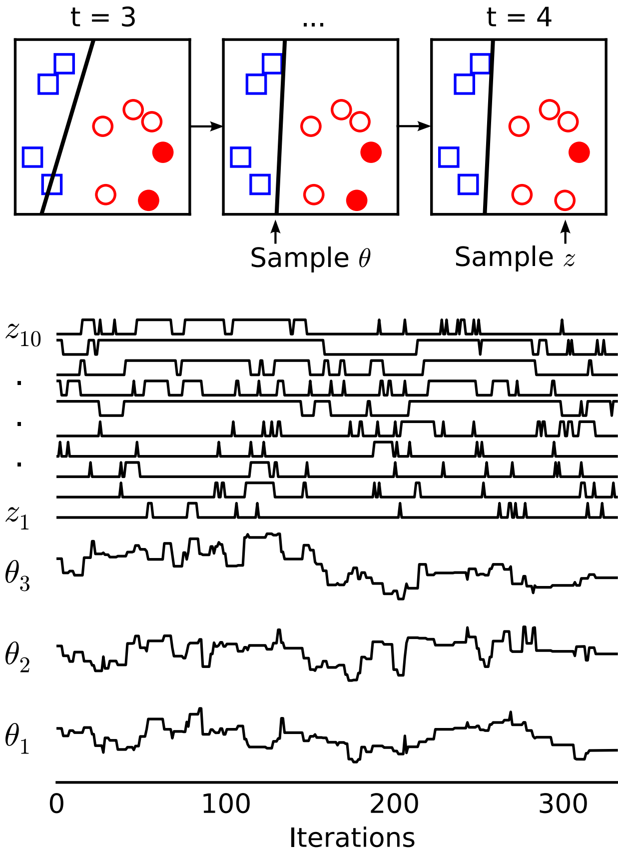



**Figure 3.2:** Illustration of the Firefly algorithm operating on a logistic regression model of a toy synthetic data set, a two-class classification problem in two dimensions (and one bias dimension). The top panel shows a single iteration of Firefly, from $t = 3$ to $t = 4$, which consists of two steps: first we sample $\theta$, represented by the line of equal class probability. Next we sample the $z_n$. In this case, we see one 'bright' (solid) data point become dark. The bottom panel shows the trajectories of all components of $\theta$ and $z$.

The second important property is that the product of the lower bounds must be easy to compute and represent. This property emerges naturally if we use scaled exponential-family lower bounds so that their product can be summarized via a set of sufficient statistics. We should also mention that the individual bounds $B_n(\theta)$ should be easy to compute themselves, since these are computed alongside $L_n(\theta)$ for all the bright points at each iteration. In all the examples considered in this chapter, the rate-limiting step in computing either $L_n(\theta)$ or $B_n(\theta)$ is the evaluation of the dot product of a feature vector with a vector of weights. Once we have computed $L_n(\theta)$ the extra cost of computing $B_n(\theta)$ is negligible.

At this stage it is useful to consider a concrete example. The logistic regression likelihood is

$$L_n(\theta) = \text{logit}^{-1}(t_n \theta^T x_n) = \frac{1}{1 + \exp\{-t_n \theta^T x_n\}} ,$$

where $x_n \in \mathbb{R}^D$ is the set of features for the $n$th data point and $t_n \in \{-1, 1\}$ is its class. The logistic function has a family of scaled Gaussian lower bounds, described in Jaakkola & Jordan [46], parameterized by $\xi$, the location at which the bound is tight:

$$\log(B_n(\theta)) = a(t_n \theta^T x_n)^2 + b(t_n \theta^T x_n) + c$$

where:

$$\begin{aligned} a &= \frac{-1}{4\xi} \left( \frac{e^\xi - 1}{e^\xi + 1} \right) \qquad b = \frac{1}{2} \\ c &= -a * \xi^2 + \frac{\xi}{2} - \log\left(e^\xi + 1\right). \end{aligned}$$

This is the bound shown in Fig. 3.1. The product of these bounds can be computed for a given $\theta$ in $O(D^2)$ time, provided we have precomputed the moments of the data,

**Algorithm 1** Firefly Monte Carlo

*Note: Using simple random-walk MH for clarity.*

```
1: θ_0 ∼ InitialDist                                   ▷ Initialize the Markov chain state.
2: for i ← 1 ... Iters                                  ▷ Iterate the Markov chain.
3:     for j ← 1 ... ⌈N × ResampleFraction⌉
4:         n ∼ RandInteger(1, N)                        ▷ Select a random data point.
5:         z_n ∼ Bernoulli(1 − B_n(θ_{i−1})/L_n(θ_{i−1}))  ▷ Biased coin-flip to determine
   whether n is bright or dark.
6:     end for
7:     θ′ ← θ_{i−1} + η where η ∼ Normal(0, ε²I_D)  ▷ Make a random walk proposal
   with step size ε.
8:     u ∼ Uniform(0, 1)                                ▷ Draw the MH threshold.
9:     if JointPosterior(θ′ ; {z_n}_{n=1}^N) / JointPosterior(θ ; {z_n}_{n=1}^N) > u   ▷ Evaluate MH ratio
   conditioned on auxiliary variables.
10:        θ_i ← θ′                                      ▷ Accept proposal.
11:    else
12:        θ_i ← θ_{i−1}                                 ▷ Reject proposal and keep current state.
13:    end if
14: end for
15:
16: function JointPosterior(θ ; {z_n}_{n=1}^N)          ▷ Modified posterior that
   conditions on auxiliary variables.
17:    P ← p(θ) × ∏_{n=1}^N B_n(θ)  ▷ Evaluate prior and bounds. Collapse of bound
   product not shown.
18:    for each n for which z_n = 1                     ▷ Loop over bright data only.
19:        P ← P × (L_n(θ)/B_n(θ) − 1)                  ▷ Include bound-corrected factor.
20:    end for
21:    return P
22: end function
```

at a one-time setup cost of $O(ND^2)$:

$$\frac{1}{N}\log\prod_{n=1}^{N}B_n(\theta) = a\theta^T\hat{S}\theta + b\theta^T\hat{\mu} + c$$

where

$$\hat{S} = \frac{1}{N}\sum_{n=1}^{N} x_n x_n^T \qquad\qquad \hat{\mu} = \frac{1}{N}\sum_{n=1}^{N} t_n x_n\,.$$

This bound can be quite tight. For example, if we choose $\xi = 1.5$ the probability of a data point being bright is less than 0.02 in the region where $0.1 < L_n(\theta) < 0.9$. With a bit of up-front work, we can do even better than this by choosing bounds that are tight in the right places. For example, we can perform a quick optimization to find an approximate maximum *a posteriori* (MAP) value of $\theta$ and construct the bounds to be tight there. We explore this idea further in Section 3.4.

### 3.3.2 Sampling and handling the auxiliary brightness variables

The resampling of the $z_n$ variables, as shown in lines 3 to 6 of Algorithm 1, takes a step by explicitly sampling $z_n$ from its conditional distribution for a random fixed-size subset of the data. We call this approach *explicit resampling* and it has a clear drawback: if the fixed fraction is $\alpha$ (shown as ResampleFraction in Algorithm 1), then the chain cannot have a mixing time faster than $1/\alpha$, as each data point is only visited a fraction of the time.

Nevertheless, explicit resampling works well in practice since the bottleneck for mixing is usually the exploration of the space of $\theta$, not space of $z_n$. Explicit resampling has the benefit of being a simple, low-overhead algorithm that is easy to vectorize for speed.The variant shown in Algorithm 1 is the simplest: data points are chosen at random, with replacement. We could also sample without replacement but this is slightly harder to do efficiently. Another variant would be to deterministically choose a subset from which to Gibbs sample at each iteration. This is more in line with the traditional approach

of stochastic gradient descent optimization. Such an approach may be appropriate for data sets which are too large to fit into memory, since we would no longer need random access to all data points. The resulting Markov chain would be non-reversible, but still satisfy stationarity conditions.

Explicitly sampling a subset of the $z_n$ seems wasteful if $M \ll N$, since most updates to $z_n$ will leave it unchanged. We can do better by drawing each update for $z_n$ from a pair of tunable Bernoulli proposal distributions $q(z'_n = 1 \,|\, z_n = 0) = q_{d \to b}$ and $q(z'_n = 0 \,|\, z_n = 1) = q_{b \to d}$, and then performing a Metropolis–Hastings accept/reject step with the true auxiliary probability $p(z_n \,|\, x_n, \theta)$. This proposal can be efficiently made for each data point, but it is only necessary to evaluate $p(z_n \,|\, x_n, \theta)$ – and therefore the likelihood function – for the subset of data points which are proposed to change state. That is, if a sample from the proposal distribution sends $z_n = 0$ to $z_n = 0$ then it doesn't matter whether we accept or reject. If we use samples from a geometric distribution to choose the data points, it is not even necessary to explicitly sample all of the $N$ proposals.

The probabilities $q_{b \to d}$ and $q_{d \to b}$ can be tuned as hyperparameters. If they are larger than $p(z_n = 0 \,|\, x_n, \theta)$ and $p(z_n = 1 \,|\, x_n, \theta)$ respectively, then we obtain near-perfect Gibbs sampling. But larger values also require more likelihood evaluations per iteration. Since the likelihoods of the bright data points have already been evaluated in the course of the Markov step in $\theta$ we can reuse these values and set $q_{b \to d} = 1$, leaving $q_{d \to b}$ as the only hyperparameter, which we can set to something like $M/N$. The resulting algorithm, which we call *implicit resampling*, is shown as Algorithm 2.

**Algorithm 2** Implicit $z_n$ sampling

```
1: for n ← 1 ... N                                   ▷ Loop over all the auxiliary variables.
2:     if z_n = 1                                    ▷ If currently bright, propose going dark.
3:         u ~ Uniform(0, 1)                         ▷ Sample the MH threshold.
4:         if q_{d→b} / L̃_n(θ) > u                   ▷ Compute MH ratio with L̃_n(θ) cached from θ update.
5:             z_n ← 0                               ▷ Flip from bright to dark.
6:         end if
7:     else                                          ▷ Already dark, consider proposing to go bright.
8:         if v < q_{d→b} where v ~ Uniform(0, 1)    ▷ Flip a biased coin with probability q_{d→b}.
9:             u ~ Uniform(0, 1)                     ▷ Sample the MH threshold.
10:            if L̃_n(θ) / q_{d→b} < u               ▷ Compute MH ratio.
11:                z_n ← 1                           ▷ Flip from dark to bright.
12:            end if
13:        end if
14:    end if
15: end for
```

### 3.3.3 Data structure for brightness variables

In the algorithms shown so far, we have aimed to construct a valid Markov chain while minimizing the number of likelihood evaluations, on the (reasonable) assumption that likelihood evaluations dominate the computational cost. However, the algorithms presented do have some steps which appear to scale linearly with $N$, even when $M$ is constant. These are steps such as "loop over the bright data points" which takes time linear in $N$. With a well-chosen data structure for storing the variables $z_n$, we can ensure that these operations only scale with $M$.

The data structure needs to store the values of $z_n$ for all $n$ from 1 to $N$, and it needs to support the following methods in $O(1)$ time:

- `Brighten`$(n)$ : Set $z_n = 1$
- `ithBright`$(i)$ : Return $n$, the $i$th bright data point (in some arbitrary ordering).

We similarly require `Darken` and `ithDark`. The data structure should also keep track of how many bright data points there are.

To achieve this, we use the cache-like data structure shown in Figure 3.3. We store two arrays of length $N$. The first is `z.arr`, which contains a single copy of each of the indices $n$ from 1 to $N$. All of the bright indices appear before the dark indices. A variable `z.B` keeps track of how many bright indices there are, and thus where the bright-dark transition occurs. In order to also achieve $O(1)$ assignment of indices, we also maintain a direct lookup table `z.tab` whose $n$th entry records the position in array `z.arr` where $n$ is held. `Brighten`$(n)$ works by looking up int `z.tab` the position of $n$ in `z.arr`, swapping it with the index at position `z.B`, incrementing `z.B`, and updating `z.tab` accordingly.

## 3.4 Experiments

For Firefly to be a useful algorithm it must be able to produce effectively independent samples from posterior distributions more quickly than regular MCMC. We certainly expect it to iterate more quickly than regular MCMC since it evaluates fewer likelihoods per iteration. But we might also expect it to mix more slowly, since it has extra auxiliary variables. To see whether this trade-off works out in Firefly's favor we need to know how much faster it iterates and how much slower it mixes. The answer to the first question will depend on the data set and the model. The answer to the second will depend on these too, and also on the choice of algorithm for updating $\theta$.

We conducted three experiments, each with a different data set, model, and parameter-

update algorithm, to give an impression of how well Firefly can be expected to perform. In each experiment we compared Firefly, with two choices of bound selection, to regular full-posterior MCMC. We looked at the average number of likelihoods queried at each iteration and the number of effective samples generated per iteration, accounting for autocorrelation. The results are summarized in Figure 3.4 and Table 3.1. The broad conclusion is that Firefly offers a speedup of at least one order of magnitude compared with regular MCMC if the bounds are tuned according to a MAP-estimate of $\theta$. In the following subsections we describe the experiments in detail.

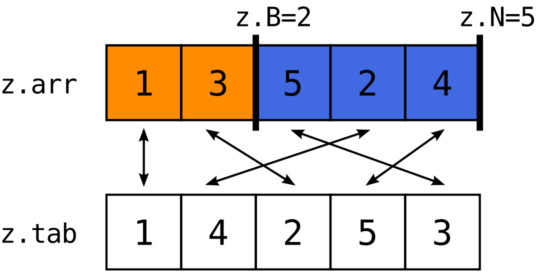



**Figure 3.3:** Illustration of a data structure allowing for efficient operations on the sets of bright and dark data points. Data points 1 and 3 are bright, the rest are dark.

### 3.4.1 Logistic regression

We applied Firefly to the logistic regression task described in [113] using the Jaakkola-Jordan bounds described earlier. The task is to classify MNIST 7s and 9s, using the first 50 principal components (and one bias) as features. We used a Gaussian prior over the weights and chose the scale of that prior by evaluating performance on a held-out test set. To sample over $\theta$, we used symmetric Metropolis-Hasting proposals, with step size chosen to yield an acceptance rate of 0.234 [89], optimized for each algorithm separately. We sampled the $z_n$ using the implicit Metropolis-Hastings sampling algorithm.

| | | Algorithm | Average Likelihood queries per iteration | Effective Samples per 1000 iterations | Speedup relative to regular MCMC |
|---|---|---|---|---|---|
| Data set: | MNIST | Regular MCMC | 12,214 | 3.7 | (1) |
| Model: | Logistic regression | Untuned Firefly | 6,252 | 1.3 | 0.7 |
| Updates: | Metropolis-Hastings | MAP-tuned Firefly | 207 | 1.4 | 22 |
| Data set: | 3-Class CIFAR-10 | Regular MCMC | 18,000 | 8.0 | (1) |
| Model: | Softmax classification | Untuned Firefly | 8,058 | 4.2 | 1.2 |
| Updates: | Langevin | MAP-tuned Firefly | 654 | 3.3 | 11 |
| Data set: | OPV | Regular MCMC | 18,182,764 | 1.3 | (1) |
| Model: | Robust regression | Untuned Firefly | 2,753,428 | 1.1 | 5.7 |
| Updates: | Slice sampling | MAP-tuned Firefly | 575,528 | 1.2 | 29 |

**Table 3.1:** Results from empirical evaluations. Three experiments are shown: logistic regression applied to MNIST digit classification, softmax classification for three categories of CIFAR-10, and robust regression for properties of organic photovoltaic molecules, sampled with random-walk Metropolis–Hastings, Metropolis-adjusted Langevin, and slice sampling, respectively. For each of these, the vanilla MCMC operator was compared with both untuned Firefly and Firefly where the bound was determined from a MAP estimate of the posterior parameters. We use likelihood evaluations as an implementation-independent measure of computational cost and report the number of such evaluations per iteration, as well as the resulting sample efficiency (computed via R-CODA [80]), and relative speedup.

We compared three different algorithms: regular MCMC, untuned Firefly, and MAP-tuned Firefly. For untuned Firefly, we chose $\xi = 1.5$ for all data points. To compute the bounds for the MAP-tuned algorithm, we performed stochastic gradient descent optimization to find a set of weights close the MAP value and gave each data point its own $\xi$ to make the bounds tight at the MAP parameters: $L_n(\theta_{\mathrm{MAP}}) = B_n(\theta_{\mathrm{MAP}})$ for all $n$. For untuned Firefly, and MAP-tuned Firefly we used $q_{d\to b} = 0.1$ and $q_{d\to b} = 0.01$ respectively, chosen to be similar to the typical fraction of bright data points in each case.

The results are shown in Figure 3.4a and summarized in Table 3.1. On a per-iteration basis, the Firefly algorithms mix and burn-in more slowly than regular MCMC by around a factor of two, as illustrated by the autocorrelation plots. Even on a per-likelihood basis,

the naïve Firefly algorithm, with a fixed $\xi$, performs worse than regular MCMC, by a factor of 0.7, despite needing fewer likelihood evaluations per iteration. The MAP-tuned algorithm was much more impressive: after burn-in, it queried only 207 of the 12,2214 likelihoods per iteration on average, giving a speedup of more than 20, even taking into account the slower per-iteration mixing time. We initialized all chains with draws from the prior. Notice that the MAP-tuned algorithm performs poorly during burn-in, since the bounds are less tight during this time, whereas the reverse is true for the untuned algorithm.

### 3.4.2 Softmax classification

Logistic regression can be generalized to multi-class classification problems by softmax classification. The softmax likelihood of a data point belonging to class $k$ of $K$ classes is

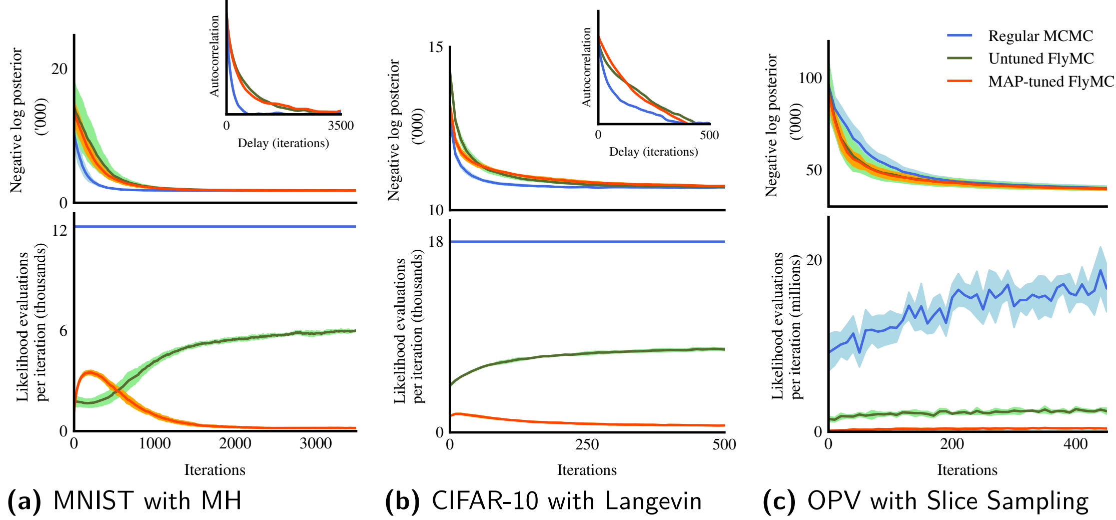
$$L_n(\theta) = \frac{\exp(\theta_k^T x_n)}{\sum_{k'=1}^{K} \exp(\theta_{k'}^T x_n)}$$

Where $\theta$ is now a $K \times D$ matrix. The Jaakkola-Jordan bound does not apply to this softmax likelihood, but we can use a related bound, due to Böhning [16], whose log matches the value and gradient of the log of the softmax likelihood at some particular $\theta$, but has a tighter curvature. Murphy [71] has the result in full in the chapter on variational inference.

We applied softmax classification to a three-class version of CIFAR-10 (airplane, automobile and bird) using 256 binary features discovered by Krizhevsky [53] using a deep autoencoder. Once again, we used a Gaussian prior on the weights, chosen to maximize out-of-sample performance. This time we used the Metropolis-adjusted

Langevin algorithm (MALA, Roberts & Tweedie [91]) for our parameter updates. We chose the step sizes to yield acceptance rates close to the optimal 0.57 [90]. Other parameters were tuned as in the logistic regression experiment.

The softmax experiment gave qualitatively similar results to the logistic regression experiment, as seen in Figure 3.4b and Table 3.1. Again, the MAP-tuned Firefly algorithm dramatically outperformed both the lackluster untuned Firefly and regular MCMC, offering an 11-fold speedup over the latter.




**(a)** MNIST with MH **(b)** CIFAR-10 with Langevin **(c)** OPV with Slice Sampling

**Figure 3.4:** Tuned and untuned Firefly Monte Carlo compared to regular MCMC with three different operators, data sets, and models: (a) the digits 7 and 9 from the MNIST data are classified using logistic regression, with a random-walk Metropolis-Hastings operator; (b) softmax classification on three classes (airplane, automobile, and bird) from the CIFAR-10 image dataset, using Langevin-adjusted Metropolis; (c) robust regression on the HOMO-LUMO gap (as computed by density functional theory calculations) for a large set of organic photovoltaic molecules, using slice sampling. In each subfigure, the top shows the trace of the log posterior density to illustrate convergence, and the bottom shows the average number of likelihoods computed per iteration. One standard deviation is shown around the mean value, as computed from five runs of each. The blue lines are computed using the full-data posterior, and the green and orange lines show the untuned and tuned Firefly MC traces, respectively.

### 3.4.3 Robust sparse linear regression

Linear regression with Gaussian likelihoods yields a closed-form expression for the posterior. Non-Gaussian likelihoods, however, like heavy-tailed distributions used in so-called "robust regression" do not. Our final experiment was to perform inference over robust regression weights for a very large dataset of molecular features and computed electronic properties. The data set, described by Hachmann et al. [35, 36] consists of 1.8 million molecules, with 57 cheminformatic features each [74, 2]. The task was to predict the HOMO-LUMO energy gap, which is useful for predicting photovoltaic efficiency.

We used a student-t distribution with $\nu = 4$ for the likelihood function and we computed a Gaussian lower bound to this by matching the value and gradient of the t distribution probability density function value at some $\xi$ ($\xi = 0$ for the untuned case, $\xi = \theta_{MAP}^T x$ for the MAP-tuned case). We used a sparsity-inducing Laplace prior on the weights. As before, we chose the scales of the prior and the likelihood to optimize out-of sample performance.

We performed parameter updates using slice sampling [72]. Note that slice sampling results in a variable number of likelihood evaluations per iteration, even for the regular MCMC algorithm. Again, we found that MAP-tuned Firefly substantially outperformed regular MCMC, as shown in Figure 3.4c and Table 3.1.

## 3.5 Conclusion

In this chapter, we have presented Firefly Monte Carlo, an algorithm for performing Markov chain Monte Carlo using subsets (minibatches) of data. Unlike other recent proposals for such MCMC operators, Firefly is exact in the sense that it has the true full-data posterior as its target distribution. This is achieved by introducing binary

latent variables whose states represent whether a given datum is bright (used to compute the posterior) or dark (not used in posterior updates). By carefully choosing the conditional distributions of these latent variables, the true posterior is left intact under marginalization. The primary requirement for this to be efficient is that the likelihoods term must have lower bounds that collapse in an efficient way.

There are several points that warrant additional discussion and future work. First, we recognize that useful lower bounds can be difficult to obtain for many problems. It would be helpful to produce such bounds automatically for a wider class of problems. As variational inference procedures are most often framed in terms of lower bounds on the marginal likelihood, we expect that Firefly Monte Carlo will benefit from developments in so-called “black box” variational methods [111, 83]. Second, we believe we have only scratched the surface of what is possible with efficient data structures and latent-variable update schemes. For example, the MH proposals we consider here for $z_n$ have a fixed global $q_{d\to b}$, but clearly such a proposal should vary for each datum. Third, it is often the case that larger state spaces lead to slower MCMC mixing. In Firefly Monte Carlo, much like other auxiliary variable methods, we have expanded the state space significantly. We have shown empirically that the slower mixing can be more than offset by the faster per-transition computational time. In future work we hope to show that fast-mixing Markov chains on the parameter space will continue to mix fast in the Firefly auxiliary variable representation.

Firefly Monte Carlo is closely related to recent ideas in using pseudo-marginal MCMC [4] for sampling from challenging target distributions. If we sampled each of the variables $\{z_n\}$ as a Bernoulli random variable with success probability 0.5, then the joint posterior we have been using becomes an unbiased estimator of the original posterior over $\theta$, up to normalization. Running pseudo-marginal MCMC using this unbiased estimator would

be a special case of Firefly: namely Firefly with $z$ and $\theta$ updated simultaneously with Metropolis-Hastings updates.

# 4

# Autograd: Automatic Differentiation for Python

In Chapter 2 we described reverse-mode differentiation, a procedure for transforming a function $F : \mathbb{R}^D \to \mathbb{R}$ composed of primitive functions into $F'$, an efficient evaluator of the gradient of $F$. Too often, this transformation is done by hand. Painstakingly derived and error-prone gradient expressions occupy a substantial portion of machine learning papers, textbooks and software.

As a purely mechanistic procedure, the transformation can and should be automated. Given a function expressed as a computer program, we should have access to its gradient

without any extra (human) effort. There are some existing software tools for doing this, but they mostly require the user to specify the function using a very limited mini-language. This is better than deriving gradients by hand, but far from effortless.

Dissatisfied with the existing options, we wrote our own automatic differentiation system. We wrote it for the Python programming language and the Numpy numerical computing library and we immodestly gave it the unqualified name "Autograd". Autograd has become quite popular in the machine learning community at large and it has powered all of the work in the rest of this thesis. This chapter describes the implementation of Autograd and the design decisions and trade-offs we made along the way.

I started the Autograd project and wrote the core logic but I was soon joined by David Duvenaud and later Matt Johnson, who deserve much of the credit for extending and maintaining it. The source code can be found at `github.com/HIPS/autograd`.

## 4.1 Mission and design principles

The goal of Autograd is to make gradients effortless. If you can write a loss function, Autograd should be able to give you its gradient. Moreover, in writing that loss function, you should have access to the full expressiveness of a modern high-level programming language (Python) and a mature numerical library (Numpy). Autograd shouldn't get in the way. You should be able to use loops and branches, recursion, function closures, classes, container data structures, numerical data abstractions, and so on.

Beyond this central mission, our design and implementation had a number of guiding principles:

*Support for higher-order derivatives* — It should be possible to take the gradient of a function which itself is composed of gradients. Our core logic has supported this from

the beginning, but maintaining it in practice requires that every time we define a new gradient, we also define the gradients of the primitive functions we use to implement that gradient, and the gradients of those functions ... Thankfully, the recursion usually bottoms out fairly quickly.

*Well-defined scope* — Following the UNIX philosophy, Autograd tries to do one thing (produce gradients) and do it well. We've been tempted (and asked) to offer our own low-level numerical operations and high-level machine learning abstractions but we've mostly managed to resist such mission creep. Having a narrow scope has also allowed us to keep the interface very minimal. For a long time Autograd only exposed a single function, `grad` (we now have a few other high-level derivative operations like `jacobian`). This made it easy for people to use, and allowed us to freely change the underlying implementation with breaking our users' code.

*Systematically tested* — One of the nice things about writing gradient-taking software is that it's easy to verify correctness, since gradient implementations can always be checked against finite differences: $F'(x) \approx (F(x + \epsilon/2) - F(x - \epsilon/2))/\epsilon$. We have test harnesses for easily writing tests for new primitive implementations that check against a battery of input types and values. Our current test suite checks nearly 100,000 different gradients. This has been a great source of confidence and peace of mind.

*Functional style* — Autograd is written in a distinctly functional style. We make heavy use of closures, higher-order functions (even `grad` itself, for example) and we prefer immutable state where possible. An exception is the tape data structure, described below.

## 4.2 Building the computation graph

Reverse-mode differentiation gives us a procedure for transforming the computation graph of the function $F$ into the gradient function $F'$. But first, we need access to the computation graph in some convenient data structure. There are several approaches to this problem.

*Direct specification of computation graph* — One approach is to have the user construct the computation graph directly, specifying a function by manipulating the nodes of the graph as objects. Indeed, some very popular automatic differentiation tools such as Theano [8, 13] take this approach. But node-by-node assembly is a clumsy way to define a function and it's also very limited in its expressiveness. What if you need a loop? Or a branch? You could introduce extra tools for specifying these things, but that amounts to inventing (and asking users to learn) an entire language-within-a-language.

A much more natural way to define a function is to use the native function definition syntax of the programming language itself. Arguably, providing a convenient way to specify computational functions is exactly what programming languages are built for!

*Source code inspection* — So how do we obtain the computation graph for a regular function written in a programming language? One way is to directly inspect the source code itself, since we know that tells us everything there is to know about the function. This is feasible with some languages, particularly so-called homoiconic languages like Lisp, in which we can manipulate programs like any other data structure. However, a syntactically rich language like Python is hard to inspect. In principle one can access the abstract syntax tree, but making sense of it would amount to building our own Python interpreter.

*Monitoring function execution* — A third approach, and the one we take with Au-

tograd, is to monitor the function as it is actually being executed, and construct the computation graph on the fly by recording every primitive function call. One can imagine doing this by directly hooking into the interpreter — Python offers mechanisms for this, designed for profiling — but we take a more low-tech approach, which we now describe.

The main idea is to replace every primitive function with a wrapped version. The wrapping layer takes care of the necessary book-keeping, recording the name of the function, the arguments, and the return value. To the user, the wrapped function looks and behaves just like the original.

As well as wrapping functions, we also wrap (or *box*) the return values from each primitive function call. We call these `Node` objects, since they are the nodes in the computation graph, and we use them to record the required information about primitive function calls. Each `Node` contains an underlying value and a recipe for creating that value, consisting of the identity of the function that created it, and a list of the arguments passed to that function, (we call these the `Node`s' parents).

Each time a wrapped primitive is called, it inspects its arguments and unboxes any `Node`s. It then calls its underlying primitive function with these (unboxed) arguments. Finally, it boxes the return value as a new `Node`, along with the recipe (function and arguments) for creating it. (Actually, for performance reasons, we only do this final re-boxing if at least one of the original arguments was boxed.)

Given a particular `Node`, we can now trace its parents, and their parents, and so on, to obtain a complete genealogy, a history of how the node's value was computed. This is a representation of the computation graph, as illustrated in Figure 4.2.

We should note that since this computation graph is created dynamically, it will depend in general on the particular arguments passed to the function. Its topology

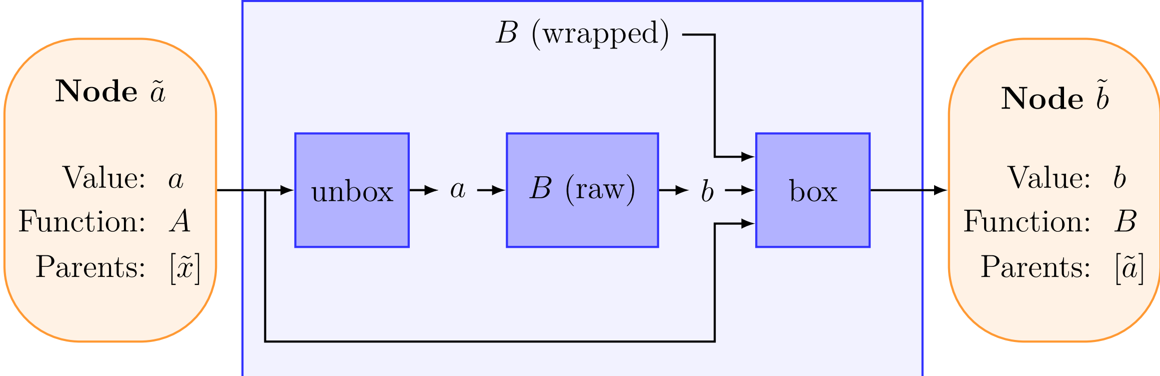



**Figure 4.1:** Primitive function wrapping. When called with a `Node` as an argument, the wrapped version of the primitive function extracts the `Node`'s value and passes it to the underlying primitive function. It boxes the result in a new `Node`, along with pointers to the original argument and the function itself.

may even be different from one call to the next. For example, if the composite function contains a loop, the computation graph will be an unrolled version of this loop and its size will depend on the number of loop iterations. Creating computation graphs dynamically allows Autograd to handle code containing loops and branches without explicitly accounting for them.

## 4.3 Performing the backward pass

We now have a way to obtain a computation graph, represented by linked nodes, given a function composed of wrapped primitives and particular inputs. Finding the derivative of one node, $y$ with respect to another, $x$, should just be a matter of applying the reverse accumulation mode differentiation procedure described in Chapter 2. This means traversing the graph from $y$ back to $x$, applying vector-Jacobian product operators and accumulating values like $\frac{\partial y}{\partial a}$ (for each intermediate value $a$) until we have $\frac{\partial y}{\partial x}$.

The hard part is knowing the order in which to do this. As we explained at the end of Chapter 2, fan-out on the forward pass becomes summation on the reverse pass. On the

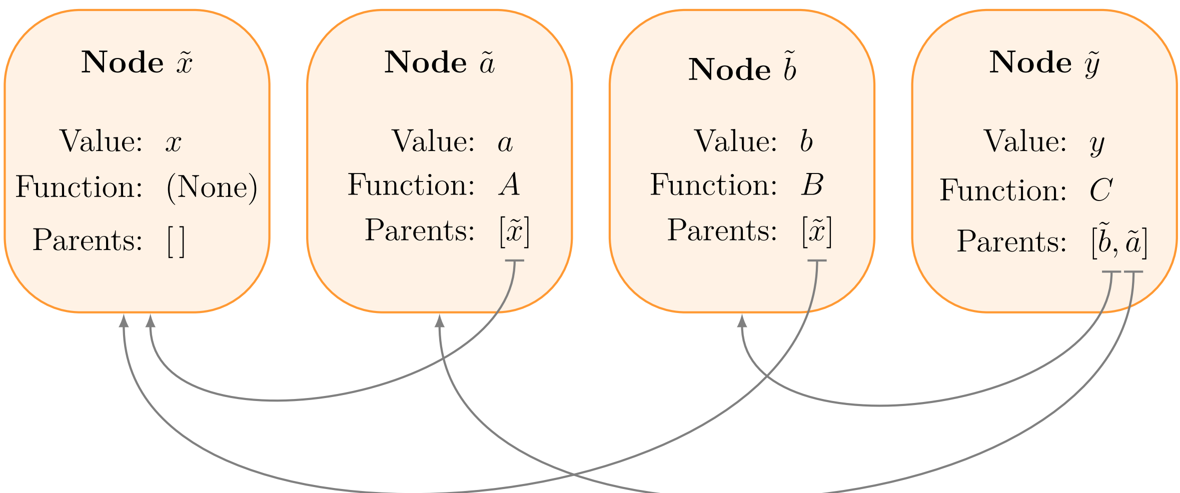



**Figure 4.2:** Representation of computation graph as set of linked nodes. Each node records the function that produces it, and the arguments given to that function, the node's parents. This allows the entire ancestry of any node to be determined.

reverse pass, we can't proceed past a multiply-referenced value until we have traversed all of the paths that lead up to it. What we need is a topological sorting of the graph, an ordering of nodes with children always appearing before their parents.

We could construct a topological sort by doing a preliminary graph traversal before performing the actual gradient accumulation traversal, but there is an easier way. In evaluating the function itself, the interpreter has already done a full graph traversal observing the opposite ordering constraint: parents before children. (The user establishes this order when defining the function by making sure to assign values to variables before using them.) If we can record this order we can simply reverse it for the backward pass.

Recording the order of operations is easy, although it does destroy functional purity. We maintain a list of operations we call the "tape". Every time a new `Node` is created, we append it to the tape. When the function finishes evaluating, the tape is a topologically

sorted list of `Nodes`. We create a separate tape for each variable we want the gradient with respect to. Each `Node` keeps track of the tapes it belongs to, and `Nodes` inherit tapes from their parents. Each tape then represents a (topologically sorted) subgraph of those `Nodes` that are direct descendants of a particular variable, and when we walk backwards through the tape, we only encounter relevant `Nodes`.

## 4.4 Primitive vector-Jacobian products

We've now described the core logic of Autograd, which is actually quite small. The bulk of the work in writing Autograd came in defining the vector-Jacobian products (VJPs) for each primitive function. We started from a small base, just a few basic arithmetic operators, and we've grown to over 200 primitive gradients implemented. The full list at time of writing is in Table 4.1.

The core logic of Autograd permits higher-order derivatives because the mechanism for tracking primitive operations continues to work even if those primitives are being applied on the backward pass. We just need to make sure that we implement VJPs for any primitive we use to implement a VJP. This is a circular requirement, and one might worry that it would lead to a never-ending chain of primitives in need of implementation. In practice, however, we find that primitive functions fall nicely into closed groups, which only need other group members for their gradients. (Consider, for example the basic trigonometric functions: the derivative of sine requires cosine and vice versa.) This also means that gradients are usually able to be implemented at the same level of abstraction as the primitive functions themselves. For example, the VJP for matrix-vector multiplication with respect to the vector argument is itself just a matrix-vector product. This keeps things efficient, as we can continue to take advantage of high-performance libraries that operate at that level of abstraction.

| | |
|---|---|
| Operators | +, -, *, /, (-), **, %, <, <=, ==, !=, >=, > |
| Basic math functions | exp, log, square, sqrt, sin, cos, tan, sinh, cosh, tanh, sinc, abs, fabs, logaddexp, logaddexp2, absolute, reciprocal, exp2, expm1, log2, log10, log1p, arcsin, arccos, arctan, arcsinh, arccosh, arctanh, rad2deg, degrees, deg2rad, radians |
| Complex numbers | real, imag, conj, angle, fft, fftshift, ifftshift, real_if_close |
| Array reductions | sum, mean, prod, var, std, max, min, amax, amin |
| Array reshaping | reshape, ravel, squeeze, diag, roll, array_split, split, vsplit, hsplit, dsplit, expand_dims, flipud, fliplr, rot90, swapaxes, rollaxis, transpose, atleast_1d, atleast_2d, atleast_3d |
| Linear algebra | dot, tensordot, einsum, cross, trace, outer, det, slogdet, inv, norm, eigh, cholesky, sqrtm, solve_triangular |
| Other array operations | cumsum, clip, maximum, minimum, sort, msort, partition, concatenate, diagonal, truncate_pad, tile, full, triu, tril, where, diff, nan_to_num, vstack, hstack |
| Probability functions | t.pdf, t.cdf, t.logpdf, t.logcdf, multivariate_normal.logpdf, multivariate_normal.pdf, multivariate_normal.entropy, norm.pdf, norm.cdf, norm.logpdf, norm.logcdf, dirichlet.logpdf, dirichlet.pdf |
| Special functions | polygamma, psi, digamma, gamma, gammaln, rgamma, multigammaln, j0, y0, j1, y1, jn, yn, erf, erfc |

**Table 4.1:** Primitives with gradients implemented in Autograd.

Not all functions are differentiable. Notably, Boolean-valued comparison operations like “$>$” and “$<$”, which are often used to select branches, are constant with respect

to their inputs almost everywhere (and have undefined derivatives where they aren't). Autograd has a dedicated mechanism to specify these as constant functions so that their descendants aren't tracked.

## 4.5 Generalizing to other data types

In discussing reverse-mode differentiation in Chapter 2, we tacitly assumed that primitive functions are vector-to-vector functions, $\mathbb{R}^M \to \mathbb{R}^N$. But we'd like Autograd to be able to handle all of our favorite numerical types: scalars, multidimensional arrays, and even user-defined types or arbitrary containers such as a tuple of an array, a scalar and a complex scalar.

As long these types map to $\mathbb{R}^D$, it's easy to imagine how to generalize VJP functions, but it helps to introduce some formalism. Consider a primitive function $F : V \to W$ where $V$ and $W$ are vector spaces over $\mathbb{R}$. Since this is a physics thesis, we'll use Dirac notation to represent members of these vector spaces and their duals: kets for vectors, $|x\rangle \in V$, and bras for covectors, $\langle x| \in V^*$ (where $V^*$ is the dual space of $V$, the space of linear maps $V \to \mathbb{R}$). We can generalize the Jacobian-vector product function, $J_F : V \to (V \to W)$ as follows:

$$J_F\big(\,|x\rangle\,,|a\rangle\,\big) = \lim_{\alpha\to 0} \frac{F\big(\,|x\rangle + \alpha\,|a\rangle\,\big) - F\big(\,|x\rangle\,\big)}{\alpha}, \tag{4.1}$$

$$\text{where} \quad \alpha \in \mathbb{R} \quad \text{and} \quad |x\rangle\,,|a\rangle \in V$$

Note that $J_F$ is a linear map $V \to W$ in its second argument. We can obtain the vector-Jacobian product function as the transpose of this linear map, $J_F^T : V \to (W^* \to V^*)$,

defined by:

$$J_F^T\big(\,|x\rangle\,,\langle b|\,\big)\,|a\rangle = \langle b|\,J_F\big(\,|x\rangle\,,|a\rangle\,\big), \tag{4.2}$$
$$\text{where} \quad |x\rangle\,,|a\rangle \in V \quad \text{and} \quad \langle b| \in W^*$$

$J_F$ (in its second argument) maps the dual space $W^*$, to the dual space $V^*$.

Now that we have a definition of a vector-Jacobian product function for general vector spaces, Equation 4.2, we can extend Autograd to handle any data type that can be treated as a vector space. We just need to define and implement the two vector space operations: vector addition and scalar multiplication.

In practice, we also need a way to represent covectors, since we want our primitive Jacobian-vector product functions to take numerical data as arguments rather than functions. We usually represent covectors by members of the original vector space, along with a definition of what it means to apply the linear functional that the covector represents to another member of the original vector space. Usually, this is "multiply the entries elementwise and sum the results". It's also useful in practice to define a basis for any new vector space. This allows us to use Equation 4.2 and 4.1 (with finite $\alpha$) for unit tests of the primitive JVP implementations with specific inputs.

Do we need to do anything special to handle complex numbers? We might be tempted to treat complex values (scalars or arrays) as vector spaces over $\mathbb{C}$ rather than over $\mathbb{R}$. This would work if we were dealing exclusively with holomorphic primitives (functions for which the limit in 4.1, with complex $\alpha$, has the same value regardless of which direction in the complex plane $\alpha$ approaches zero from). However, we would like to be able to handle non-holomorphic primitives, like complex conjugation, as well as primitives that map between complex numbers and real numbers, such as taking the real

part of a complex number. Instead, we can actually treat complex values as members of a vector field over the *reals*. This lets us handle these non-holomorphic primitives, but still gives the same result as the complex scalar field case if the primitive happens to be holomorphic.

This general vector space treatment of the vector-Jacobian function makes it clear that the final result of reverse-mode differentiation of a function $F : V \to \mathbb{R}$ is a covector ($\langle g| \in V^*$) rather than a vector ($|g\rangle \in V$). If we want a vector in $V$, we need a way to map from covectors to vectors, which requires defining an inner product $\langle \cdot, \cdot \rangle_V$ on $V$. In practice, Autograd just returns its internal representations of the covectors and we take them at face value. This corresponds to mapping from $V^*$ to $V$ using the Euclidean inner product (and taking the complex conjugate, in the case of complex numbers). It's a fairly minor transgression, but bearing it in mind could help us avoid type errors, such as the commonly overlooked dimensional inconsistency of gradient descent.

## 4.6 Handling all of Python's syntax

Python offers a rich set of syntactical constructs. Making sure we cover all the possible mechanisms by which a variable's value can be accessed and propagated takes some work. In this section, we examine these mechanisms case by case.

*Numpy functions* — The prototypical way to refer to an object's value is to pass it as an argument to a function. Wrapping functions, as described above, is the basic instrument Autograd uses to build the computation graph. We wrap all Numpy functions by extracting them automatically, and we present the wrapped versions under an `autograd.numpy` namespace. The user can use our wrapped versions rather than the originals just by changing an import, e.g. `import autograd.numpy as np` instead of `import numpy as np`. Thankfully, the great majority of the numerical functions are

under this namespace.

*Built-in and other functions* — Python actually has quite a limited set of built-in functions, and we don't wrap any of them. The few numerical builtins, `sum`, `max`, `min`, are generic functions which dispatch to the operator overloading mechanisms described below.

*Operators* — Python offers the usual set of infix operators +, -, /, *, >, < etc. These are implemented using double-underscore methods (`__add__`, `__sub__` etc.) of the left argument. (If that doesn't exist, Python tries the corresponding right-argument version of the right arguments, `__radd__`, `__rsub__`.) Autograd's `Node` objects implement these double underscore methods, just as they implement other object methods.

*Methods* — Most object methods are accessors for an object's own data. Since we only need wrapped versions of these when the object is boxed, we just have to implement the wrapped versions as methods for Autograd's `Node`. The idea is that Autograd's `Node` objects should look and feel just like the objects they box, and this is done by implementing the same interface. But this does mean creating a type system within Autograd that parallels the underlying type system. Thus, we don't just have a single `Node` type, but an `ArrayNode`, a `FloatNode`, a `TupleNode` and so on.

There are occasional cases, however, when object methods take other objects as additional arguments. This poses a problem if an unboxed object's method is given an Autograd `Node` as an argument. One example is the dot product method, `ndarray.dot`, of Numpy arrays. Another example is the double underscore methods of binary operators (described below). These are hard cases to handle and we only partially succeed. With double underscore methods, if the left argument's `__add__` doesn't recognize the argument type, it will usually call the right arguments `__radd__`. Numpy `ndarrays` have their own convention for handling precedence, the class-level variable

`__array_priority__`, which we take advantage of. In the case of `ndarray.dot`, we decided that the complexity of wrapping it couldn't justify the benefit, and instead we ask users to use `np.dot(A, B)` rather than `A.dot(B)`.

*Indexing* — Indexing syntax, like operator syntax, is implemented with double underscore methods, and we handle it easily by implementing those methods in Autograd `Node`s. Getting the semantics right, however, is harder. As we explain in the next section, Autograd doesn't support assigning to arrays or other state-mutating operations.

*Container syntax* — Finally, Python has direct syntactical constructs for forming container data types: parenthesis for tuples, brackets for lists, and braces for dicts. Although these behave semantically like multi-argument functions, we have no way to intercept the syntax. In most cases this turns out not to be a problem. The containers may hide Autograd nodes, but it only matters when the containers are eventually unpacked, in which case the Autograd nodes are revealed. The few exceptions are where we have primitive functions that take containers as arguments. The wrapping layer can't detect any Autograd nodes smuggled inside. We could have the wrapping layer do a deep inspection of every argument but this would be unacceptably slow. We can usually handle these cases by replacing the container-accepting function with a version that unpacks the container and passes its contents to a multi-argument primitive. See, e.g. `autograd.numpy.concatenate`.

## 4.7 Limitations and future work

Perhaps the biggest limitation of Autograd is that it can't handle in-place modifications of objects, and assignment to arrays in particular. The reason is that the backward pass makes use of the values created on the forward pass. If any of these had been modified in the mean time, the computed gradients would no longer be correct. We could make

copies of all values to use on the backward pass but this would be prohibitively expensive. Alternatively, we could introduce a copy-on-write scheme, whereby we make a copy of an object only when we detect that it is being modified. This would introduce substantial additional complexity but it is a possible enhancement for the future.

Immutability of objects is a hallmark of functional programming, and it isn't as crippling as it might seem. Our experience using Autograd has been that a functional style is a natural fit for the sorts of modeling and inference problems we like to solve in machine learning, and it rarely feels burdensome.

A second limitation is that the overhead of function wrapping can be substantial. We initially wrote Autograd with the assumption that it would only be used for "BLAS-limited" computations: computations dominated by large-scale linear algebra operations which are handled (via Numpy) by ancient optimized FORTRAN libraries like BLAS. Evaluating a feed-forward neural network was always the canonical use-case.

But we (and others) found it useful in many other contexts, often involving smaller matrices and scalars. In these cases, the overhead can be painful. Frustratingly, much of the overhead comes from type checking (the function `isinstance` often shows up as hot spot when profiling). Python's laissez-faire approach to typing, which gives it a wonderful malleability, hurts us here. If we'd chosen a stricter language, like Haskell or Julia, we could have taken advantage of native multiple dispatch and type checking.

Finally, we receive many requests to add GPU support to Autograd. Indeed, if we are targeting computations in the BLAS-limited regime, it would be nice to take advantage of GPUs for highly parallel linear algebra operations. Our response is that building a GPU linear algebra library, or even creating Python bindings to an existing library, is out of scope for Autograd. We hope that Numpy (or another Python project) will build such a library, so that we can wrap it with Autograd. There is a version of Autograd

written for Lua/Torch by a team at Twitter (directly inspired by our Python Autograd) which has native GPU support because it wraps GPU-supporting numerical libraries (`github.com/twitter/torch-autograd`).

A hybrid strategy, which several people have been using already, is to use a GPU-capable library like Theano, to implement computationally expensive parts of a program, the forward evaluation of a neural net for example, then to wrap that as an Autograd primitive in order to use it as part of a larger system. Autograd's interface for defining new primitives makes this simple.

## 4.8 Conclusion

Autograd has had an unexpectedly large impact on my and my co-authors' research. It has inspired projects (such as the ones in this thesis) and it has changed the way we implement machine learning algorithms. We now focus entirely on implementing loss functions (usually log probabilities) and we are free to create and use whatever abstractions are most appropriate for doing that. To give an example, I recently implemented a simple probabilistic programming language for performing BBSVI inference in directed graphical models with arbitrary conditional probabilities. Within it I implemented Haskell-style infinite lazy lists. Together, the libraries are capable of expressing (and doing inference in) models such as an infinite input-output linear dynamical system. Autograd threads through the whole thing effortlessly. To implement this while keeping track of gradients explicitly would have been unthinkable and I would never have dared create those abstractions.

We've also found that using Autograd clarifies many ideas in machine learning and makes them not just easier to implement but also easier to think about. The "examples" directory of the Autograd repository is filled with self-contained scripts implementing

neural networks, graphical models, variational inference, Gaussian processes, mixture models and other ideas in machine learning. With automatic gradients, the implementations are remarkably terse, and the essential features of each algorithm are laid out unobscured. Taking this to an amusing extreme, Ryan Adams managed to fit Black-Box Stochastic Variational Inference into a single tweet[1].

This has been my first experience developing an open-source software package and it has been a pleasure and a thrill to see people around the world using something I've built. Autograd is downloaded more than two thousand times each month, and it has received almost 800 Github stars, the social currency of open-source development.

[1] For posterity, here is the tweet reproduced in its entirety:

**Ryan Adams @ryan_p_adams 7 Nov 2015**
@DavidDuvenaud
```
def elbo(p, lp, D, N):
  v=exp(p[D:])
  s=randn(N,D)*sqrt(v)+p[:D]
  return mvn.entropy(0, diag(v))+mean(lp(s))
gf = grad(elbo)
```

# 5

# Convolutional Networks on Graphs

In this chapter we introduce a convolutional neural network that operates directly on graphs, allowing end-to-end learning of prediction pipelines whose inputs are graphs of arbitrary size and shape. The architecture we present generalizes standard molecular feature extraction methods based on circular fingerprints. We show that these data-driven features are more interpretable, and have better predictive performance on a variety of tasks.

This work was enabled by Autograd, but the historical causality was actually in the other direction. It was the prospect of taking gradients by hand through such a complicated graph-based function that inspired Ryan to write Autograd's predecessor, Kayak.

This chapter presents work with David Duvenaud, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik and Ryan P. Adams. Ryan conceived of the original idea for the convolutional neural network architecture. David and I fleshed out the ideas, wrote the implementation, performed the experiments and wrote the paper together. It was presented at NIPS in 2015 [26].

## 5.1 Introduction

Recent work in materials design uses neural networks to predict the properties of novel molecules by generalizing from examples. One difficulty with this task is that the input to the predictor, a molecule, can be of arbitrary size and shape. Currently, most machine learning pipelines can only handle inputs of a fixed size. The current state of the art is to use off-the-shelf fingerprint software to compute fixed-dimensional feature vectors, and use those features as inputs to a fully-connected deep neural network or other standard machine learning method. This formula was followed by Unterthiner et al. [108], Dahl et al. [21], Ramsundar et al. [82]. During training, the molecular fingerprint vectors were treated as fixed.

In this chapter, we replace the bottom layer of this stack – the function that computes molecular fingerprint vectors – with a differentiable neural network whose input is a graph representing the original molecule. In this graph, vertices represent individual atoms and edges represent bonds. The lower layers of this network are convolutional in the sense that the same local filter is applied to each atom and its neighborhood. After several such layers, a global pooling step combines features from all the atoms in the molecule.

These neural graph fingerprints offer several advantages over fixed fingerprints:

- **Predictive performance.** By using adapting to the task at hand, machine-optimized fingerprints can provide substantially better predictive performance than fixed fingerprints. We show that neural graph fingerprints match or beat the predictive performance of standard fingerprints on solubility, drug efficacy, and organic photovoltaic efficiency datasets.

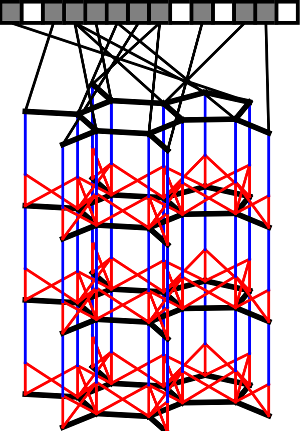
- **Parsimony.** Fixed fingerprints must be extremely large to encode all possible substructures without overlap. For example, [108] used a fingerprint vector of size 43,000, after having removed rarely-occurring features. Differentiable fingerprints can be optimized to encode only relevant features, reducing downstream computation and regularization requirements.

- **Interpretability.** Standard fingerprints encode each possible fragment completely distinctly, with no notion of similarity between fragments. In contrast, each feature of a neural graph fingerprint can be activated by similar but distinct molecular fragments, making the feature representation more meaningful.

## 5.2 Circular fingerprints

The state of the art in molecular fingerprints are extended-connectivity circular fingerprints (ECFP) [92]. Circular fingerprints [32] are a refinement of the Morgan algorithm [70], designed to encode which substructures are present in a molecule.

Circular fingerprints generate each layer's features by applying a fixed hash function to the concatenated features of the neighborhood in the previous layer. The results of these hashes are then treated as integer indices, where a 1 is written to the fingerprint vector at the index given by the feature vector at each node in the graph. Figure 5.1(left) shows a sketch of this computational architecture. Ignoring collisions, each



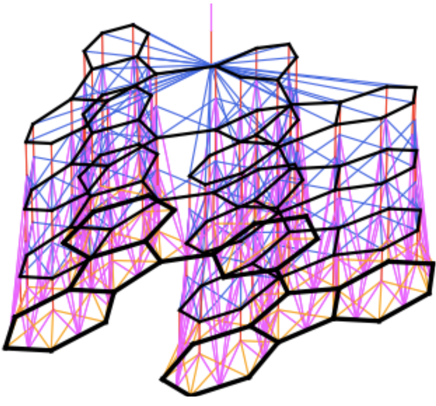


**Figure 5.1:** *Left*: A visual representation of the computational graph of both standard circular fingerprints and neural graph fingerprints. First, a graph is constructed matching the topology of the molecule being fingerprinted, in which nodes represent atoms, and edges represent bonds. At each layer, information flows between neighbors in the graph. Finally, each node in the graph turns on one bit in the fixed-length fingerprint vector. *Right*: A more detailed sketch including the bond information used in each operation.

index of the fingerprint denotes the presence of a particular substructure. The size of the substructures represented by each index depends on the depth of the network. Thus the number of layers is referred to as the 'radius' of the fingerprints.

Circular fingerprints are analogous to convolutional networks in that they apply the same operation locally everywhere, and combine information in a global pooling step.

## 5.3 Creating a differentiable fingerprint

The space of possible network architectures is large. In the spirit of starting from a known-good configuration, we designed a differentiable generalization of circular fingerprints. This section describes our replacement of each discrete operation in circular fingerprints with a differentiable analog.

Hashing The purpose of the hash functions applied at each layer of circular fingerprints is to combine information about each atom and its neighboring substructures. This ensures that any change in a fragment, no matter how small, will lead to a different fingerprint index being activated. We replace the hash operation with a single layer of a neural network. Using a smooth function allows the activations to be similar when the local molecular structure varies in unimportant ways.

Indexing Circular fingerprints use an indexing operation to combine all the nodes' feature vectors into a single fingerprint of the whole molecule. Each node sets a single bit of the fingerprint to one, at an index determined by the hash of its feature vector. This pooling-like operation converts an arbitrary-sized graph into a fixed-sized vector. For small molecules and a large fingerprint length, the fingerprints are always sparse. We use the `softmax` operation as a differentiable analog of indexing. In essence, each atom is asked to classify itself as belonging to a single category. The sum of all these classification label vectors produces the final fingerprint. This operation is analogous to the pooling operation in standard convolutional neural networks.

Canonicalization Circular fingerprints are identical regardless of the ordering of atoms in each neighborhood. This invariance is achieved by sorting the neighboring atoms according to their features, and bond features. We experimented with this sorting scheme, and also with applying the local feature transform on all possible permutations of the local neighborhood. An alternative to canonicalization is to apply a permutation-invariant function, such as summation. In the interests of simplicity and scalability, we chose summation.

Circular fingerprints can be interpreted as a special case of neural graph fingerprints having large random weights. To demonstrate that neural fingerprints are a generaliza-

tion of circular fingerprints, in this section we give evidence that circular fingerprints are similar to neural fingerprints having large, randomly-initialized parameter vectors. This is because, in the limit of large input weights, `tanh` nonlinearities approach step functions, which when concatenated form a simple hash function. Also, in the limit of large input weights, the `softmax` operator approaches a one-hot-coded `argmax` operator, which is analogous to an indexing operation.

Algorithms 3 and 4 summarize these two algorithms and highlight their differences. Given a fingerprint length $L$, and $F$ features at each layer, the parameters of neural graph fingerprints consist of a separate output weight matrix of size $F \times L$ for each layer, as well as a set of hidden-to-hidden weight matrices of size $F \times F$ at each layer, one for each possible number of bonds an atom can have (up to 5 in organic molecules).

## 5.4 Experiments

### 5.4.1 Neural fingerprints with large random weights are similar to circular fingerprints

We ran two experiments to demonstrate that neural fingerprints with large random weights behave similarly to circular fingerprints. First, we examined whether distances between circular fingerprints were similar to distances between neural fingerprint-based distances. Figure 5.3 (left) shows a scatterplot of pairwise distances between circular vs. neural fingerprints. Fingerprints had length 2048, and were calculated on pairs of molecules from the solubility dataset Delaney [22]. Distance was measured using a continuous generalization of the Tanimoto (a.k.a. Jaccard) similarity measure, given by

$$\text{distance}(\mathbf{x}, \mathbf{y}) = 1 - \sum_i \min(x_i, y_i) \Big/ \sum_i \max(x_i, y_i). \tag{5.1}$$

**Algorithm 3** Circular fingerprints

```
1: Input: molecule, radius R, fingerprint length S
2: Initialize: fingerprint vector f ← 0_S
3: for each atom a in molecule
4:     r_a ← g(a)                    ▷ lookup atom features
5: end for
6: for L = 1 to R                    ▷ for each layer
7:     for each atom a in molecule
8:         r_1 ... r_N = neighbors(a)
9:         v ← [r_a, r_1, ..., r_N]     ▷ concatenate
10:        r_a ← hash(v)              ▷ hash function
11:        i ← mod(r_a, S)            ▷ convert to index
12:        f_i ← 1                    ▷ Write 1 at index
13:    end for
14: end for
15: Return: binary vector f
```

**Algorithm 4** Neural graph fingerprints

```
1: Input: molecule, radius R, hidden weights H_1^1 ... H_R^5, output weights W_1 ... W_R
2: Initialize: fingerprint vector f ← 0_S
3: for each atom a in molecule
4:     r_a ← g(a)                    ▷ lookup atom features
5: end for
6: for L = 1 to R                    ▷ for each layer
7:     for each atom a in molecule
8:         r_1 ... r_N = neighbors(a)
9:         v ← r_a + Σ_{i=1}^{N} r_i   ▷ sum
10:        r_a ← σ(v H_L^N)           ▷ smooth function
11:        i ← softmax(r_a W_L)       ▷ sparsify
12:        f ← f + i                  ▷ add to fingerprint
13:    end for
14: end for
15: Return: real-valued vector f
```

**Figure 5.2:** Pseudocode of circular fingerprints (*left*) and neural graph fingerprints (*right*). Differences are highlighted in blue. Every non-differentiable operation is replaced with a differentiable analog.

There is a correlation of $r = 0.823$ between the distances. The line of points on the right of the plot shows that for some pairs of molecules, binary ECFP fingerprints have exactly zero overlap.

Second, we examined the predictive performance of neural fingerprints with large random weights vs. that of circular fingerprints. Figure 5.3 (right) shows average predictive performance on the solubility dataset, using linear regression on top of fingerprints. The

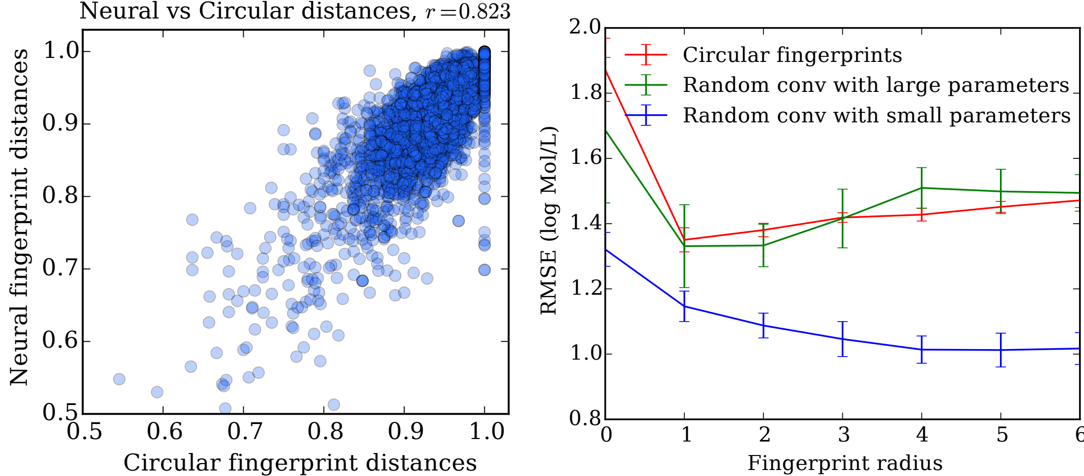



**Figure 5.3:** *Left:* Comparison of pairwise distances between molecules, measured using circular fingerprints and neural graph fingerprints with large random weights. *Right*: Predictive performance of circular fingerprints (red), neural graph fingerprints with fixed large random weights (green) and neural graph fingerprints with fixed small random weights (blue). The performance of neural graph fingerprints with large random weights closely matches the performance of circular fingerprints.

performances of both methods follow similar curves. In contrast, the performance of neural fingerprints with small random weights follows a different curve, and is substantially better. This suggests that even with random weights, the relatively smooth activation of neural fingerprints helps generalization performance.

### 5.4.2 Examining learned features

To demonstrate that neural graph fingerprints are interpretable, we show substructures which most activate individual features in a fingerprint vector. Each feature of a circular fingerprint vector can each only be activated by a single fragment of a single radius, except for accidental collisions. In contrast, neural graph fingerprint features can be activated by variations of the same structure, making them more interpretable, and allowing shorter feature vectors.

Solubility features   Figure 5.4 shows the fragments that maximally activate the most predictive features of a fingerprint. The fingerprint network was trained as inputs to a linear model predicting solubility, as measured in [22]. The feature shown in the top row has a positive predictive relationship with solubility, and is most activated by fragments containing a hydrophilic R-OH group, a standard indicator of solubility. The feature shown in the bottom row, strongly predictive of insolubility, is activated by non-polar repeated ring structures.

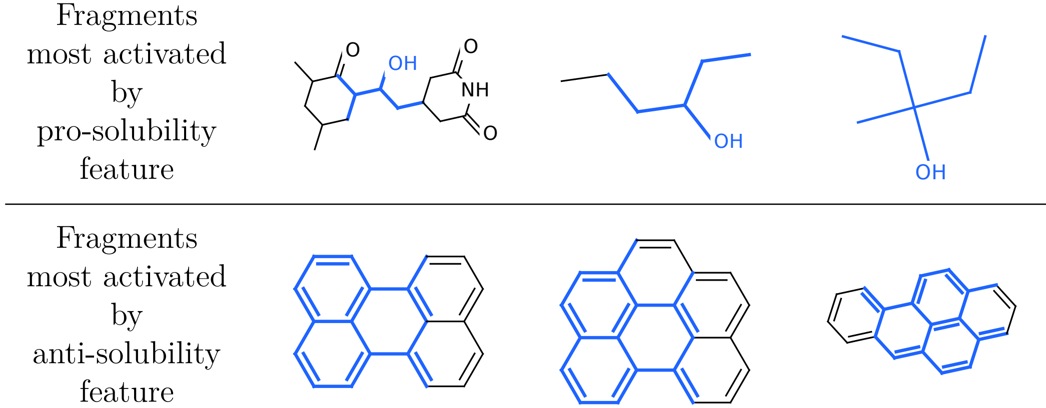



**Figure 5.4:** Examining fingerprints optimized for predicting solubility. Shown here are representative examples of molecular fragments (highlighted in blue) which most activate different features of the fingerprint. *Top row:* The feature most predictive of solubility. *Bottom row:* The feature most predictive of insolubility.

Toxicity features   We trained the same model architecture to predict toxicity, as measured in two different datasets in Tox21 Challenge [106]. Figure 5.5 shows fragments which maximally activate the feature most predictive of toxicity, in two separate datasets.

Unterthiner et al. [107] constructed similar visualizations, but in a semi-manual way: to determine which toxic fragments activated a given neuron, they searched over a hand-made list of toxic substructures and chose the one most correlated with a given neuron.

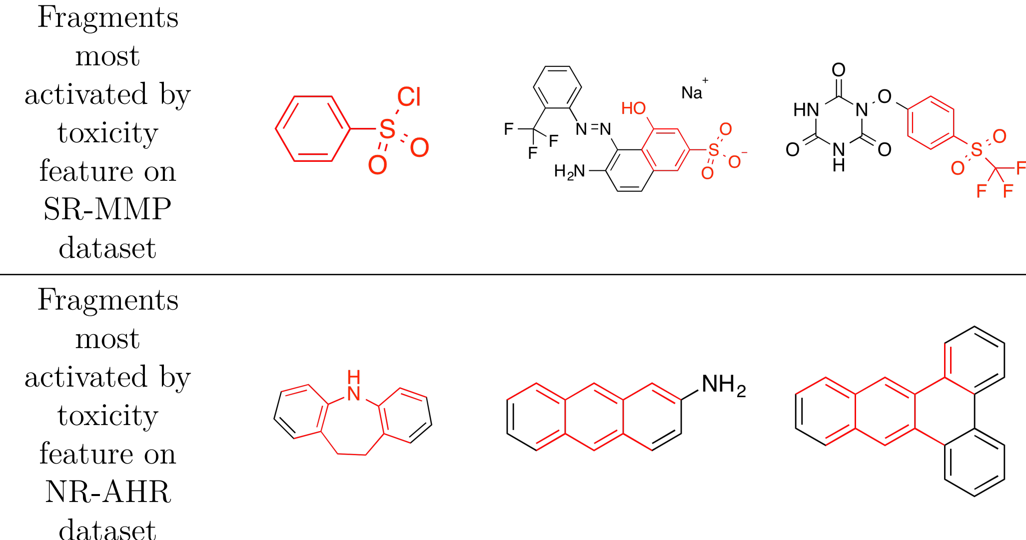



**Figure 5.5:** Visualizing fingerprints optimized for predicting toxicity. Shown here are representative samples of molecular fragments (highlighted in red) which most activate the feature most predictive of toxicity. *Top row:* the most predictive feature identifies groups containing a sulphur atom attached to an aromatic ring. *Bottom row:* the most predictive feature identifies fused aromatic rings, also known as polycyclic aromatic hydrocarbons, a well-known carcinogen.

In contrast, our visualizations are generated automatically, without the need to restrict the range of possible answers beforehand.

### 5.4.3 Predictive Performance

We ran several experiments to compare the predictive performance of neural graph fingerprints to that of the standard state-of-the-art setup: circular fingerprints fed into a fully-connected neural network.

Experimental setup  Our pipeline takes as input the SMILES [112] string encoding of each molecule, which is then converted into a graph using RDKit [86]. We also used RDKit to produce the extended circular fingerprints used in the baseline. Hydrogen atoms were treated implicitly. We implemented the neural graph fingerprints and

the conventional fully-connected neural networks in Python, using Autograd to compute gradients. Code for computing neural fingerprints and producing visualizations is available at `github.com/HIPS/neural-fingerprint`.

In our convolutional networks, the initial atom and bond features were chosen to be similar to those used by ECFP: Initial atom features concatenated a one-hot encoding of the atom's element, its degree, the number of attached hydrogen atoms, and the implicit valence, and an aromaticity indicator. The bond features were a concatenation of whether the bond type was single, double, triple, or aromatic, whether the bond was conjugated, and whether the bond was part of a ring.

Training and Architecture  Training used batch normalization [45]. We also experimented with `tanh` vs `relu` activation functions for both the neural fingerprint network layers and the fully-connected network layers. `relu` had a slight but consistent performance advantage on the validation set. We also experimented with dropconnect [110], a variant of dropout in which weights are randomly set to zero instead of hidden units, but found that it led to worse validation error in general. Each experiment optimized for 10000 minibatches of size 100 using the Adam algorithm [49], a variant of RMSprop that includes momentum.

Hyperparameter Optimization  To optimize hyperparameters, we used random search. The hyperparameters of all methods were optimized using 50 trials for each cross-validation fold. The following hyperparameters were optimized: log learning rate, log of the initial weight scale, the log $L_2$ penalty, fingerprint length, fingerprint depth (up to 6), and the size of the hidden layer in the fully-connected network. Additionally, the size of the hidden feature vector in the convolutional neural fingerprint networks was optimized.

Datasets We compared the performance of standard circular fingerprints against neural graph fingerprints on a variety of domains:

- **Solubility:** The aqueous solubility of 1144 molecules as measured by [22].
- **Drug efficacy:** The half-maximal effective concentration ($EC_{50}$) *in vitro* of 10,000 molecules against a sulfide-resistant strain of *P. falciparum*, the parasite that causes malaria, as measured by Gamo et al. [30].
- **Organic photovoltaic efficiency:** The Harvard Clean Energy Project [35] uses expensive density funtional theory (DFT) simulations to estimate the photovoltaic efficiency of organic molecules. We used a subset of 20,000 molecules from this dataset.

Predictive accuracy We compared the performance of circular fingerprints and neural graph fingerprints under two conditions: In the first condition, predictions were made by a linear layer using the fingerprints as input. In the second condition, predictions were made by a one-hidden-layer neural network using the fingerprints as input. In all settings, all differentiable parameters in the composed models were optimized simultaneously. Results are summarized in Table 5.1.

In all experiments, the neural graph fingerprints matched or beat the accuracy of circular fingerprints, and the methods with a neural network on top of the fingerprints typically outperformed the linear layers.

## 5.5 Limitations

Computational cost Neural fingerprints have the same asymptotic complexity in the number of atoms and the depth of the network as circular fingerprints, but have

| Dataset | Solubility [22] | Drug efficacy [30] | Photovoltaic efficiency [35] |
|---|---|---|---|
| Units | log Mol/L | $EC_{50}$ in nM | % |
| Predict mean | 4.29 ± 0.40 | 1.47 ± 0.07 | 6.40 ± 0.09 |
| Circular FPs + linear layer | 1.71 ± 0.13 | **1.13 ± 0.03** | 2.63 ± 0.09 |
| Circular FPs + neural net | 1.40 ± 0.13 | 1.36 ± 0.10 | 2.00 ± 0.09 |
| Neural FPs + linear layer | 0.77 ± 0.11 | **1.15 ± 0.02** | 2.58 ± 0.18 |
| Neural FPs + neural net | **0.52 ± 0.07** | **1.16 ± 0.03** | **1.43 ± 0.09** |

**Table 5.1:** Mean squared error of neural fingerprints compared to standard circular fingerprints.

additional terms due to the matrix multiplies necessary to transform the feature vector at each step. To be precise, computing the neural fingerprint of depth $R$, fingerprint length $L$ of a molecule with $N$ atoms using a molecular convolutional net having $F$ features at each layer costs $\mathcal{O}(RNFL + RNF^2)$. In practice, training neural networks on top of circular fingerprints usually took several minutes, while training both the fingerprints and the network on top took on the order of an hour on the larger datasets.

Limited computation at each layer How complicated should we make the function that goes from one layer of the network to the next? In this chapter we chose the simplest feasible architecture: a single layer of a neural network. However, it may be fruitful to apply multiple layers of nonlinearities between each message-passing step (as in [95]), or to make information preservation easier by adapting the Long Short-Term Memory [42] architecture to pass information upwards.

Limited information propagation across the graph The local message-passing architecture developed in this chapter scales well in the size of the graph (due to the low degree of organic molecules), but its ability to propagate information across the graph is limited by the depth of the network. This may be appropriate for small graphs

such as those representing the small organic molecules used in this chapter. However, in the worst case, it can take a depth $\frac{N}{2}$ network to distinguish between graphs of size $N$. To avoid this problem, Bruna et al. [17] proposed a hierarchical clustering of graph substructures. A tree-structured network could examine the structure of the entire graph using only $\log(N)$ layers, but would require learning to parse molecules. Techniques from natural language processing [103] might be fruitfully adapted to this domain.

Inability to distinguish stereoisomers Special bookkeeping is required to distinguish between stereoisomers, including enantiomers (mirror images of molecules) and *cis/trans* isomers (rotation around double bonds). Most circular fingerprint implementations have the option to make these distinctions. Neural fingerprints could be extended to be sensitive to stereoisomers, but this remains a task for future work.

## 5.6 Related work

This work is similar in spirit to the neural Turing machine [34], in the sense that we take an existing discrete computational architecture, and make each part differentiable in order to do gradient-based optimization.

Neural nets for quantitative structure-activity relationship (QSAR) The modern standard for predicting properties of novel molecules is to compose circular fingerprints with fully-connected neural networks or other regression methods. [21] used circular fingerprints as inputs to an ensemble of neural networks, Gaussian processes, and random forests. [82] used circular fingerprints (of depth 2) as inputs to a multitask neural network, showing that multiple tasks helped performance.

Neural graph fingerprints The most closely related work is Lusci et al. [60], who build a neural network having graph-valued inputs. Their approach is to remove all cycles and build the graph into a tree structure, choosing one atom to be the root. A recursive neural network [97, 98] is then run from the leaves to the root to produce a fixed-size representation. Because a graph having $N$ nodes has $N$ possible roots, all $N$ possible graphs are constructed. The final descriptor is a sum of the representations computed by all distinct graphs. There are as many distinct graphs as there are atoms in the network. The computational cost of this method thus grows as $\mathcal{O}(F^2N^2)$, where $F$ is the size of the feature vector and $N$ is the number of atoms, making it less suitable for large molecules.

Convolutional neural networks Convolutional neural networks have been used to model images, speech, and time series [58]. However, standard convolutional architectures use a fixed computational graph, making them difficult to apply to objects of varying size or structure, such as molecules. More recently, [48] and others have developed a convolutional neural network architecture for modeling sentences of varying length.

Neural networks on fixed graphs [17] introduce convolutional networks on graphs in the regime where the graph structure is fixed, and each training example differs only in having different features at the vertices of the same graph. In contrast, our networks address the situation where each training input is a different graph.

Neural networks on input-dependent graphs [95] propose a neural network model for graphs having an interesting training procedure. The forward pass consists of running a message-passing scheme to equilibrium, a fact which allows the reverse-mode

gradient to be computed without storing the entire forward computation. They apply their network to predicting mutagenesis of molecular compounds as well as web page rankings. [69] also propose a neural network model for graphs with a learning scheme whose inner loop optimizes not the training loss, but rather the correlation between each newly-proposed vector and the training error residual. They apply their model to a dataset of boiling points of 150 molecular compounds. Our work builds on these ideas, with the following differences: our method replaces their complex training algorithms with simple gradient-based optimization, generalizes existing circular fingerprint computations, and applies these networks in the context of modern QSAR pipelines which use neural networks on top of the fingerprints to increase model capacity.

Unrolled inference algorithms Hershey et al. [41] and others have noted that iterative inference procedures sometimes resemble the feedforward computation of a recurrent neural network. One natural extension of these ideas is to parameterize each inference step, and train a neural network to approximately match the output of exact inference using only a small number of iterations. The neural fingerprint, when viewed in this light, resembles an unrolled message-passing algorithm on the original graph.

## 5.7 Conclusion

We generalized existing hand-crafted molecular features to allow their optimization for diverse tasks. By making each operation in the feature pipeline differentiable, we can use standard neural-network training methods to scalably optimize the parameters of these neural molecular fingerprints end-to-end. We demonstrated the interpretability and predictive performance of these new fingerprints.

Data-driven features have already replaced hand-crafted features in speech recogni-

tion, machine vision, and natural-language processing. Carrying out the same task for virtual screening, drug design, and materials design is a natural next step.

# 6

# Hyperparameter Optimization Through Reversible Learning

Tuning hyperparameters of learning algorithms is hard because gradients are usually unavailable. In this chapter, we show how to efficiently compute exact gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives backwards through the *entire training procedure*. These gradients allow us to optimize thousands of hyperparameters, including step-size and momentum schedules, weight initialization distributions, richly parameterized regularization schemes, and neural network architectures. We do the computation in a memory-efficient way by exactly

reversing the dynamics of stochastic gradient descent with momentum.

This chapter presents work with David Duvenaud and Ryan P. Adams. David and I came up with the ideas, wrote the implementation, performed the experiments and wrote the paper together. It was presented at ICML in 2015 [26].

## 6.1 Introduction

Machine learning systems abound with hyperparameters. These can be parameters that control model complexity, such as $L_1$ and $L_2$ penalties, or parameters that specify the learning procedure itself – step sizes, momentum decay parameters and initialization conditions. Choosing the best hyperparameters is both crucial and frustratingly difficult.

The current gold standard for hyperparameter selection is gradient-free model-based optimization [96, 12, 14, 44]. Hyperparameters are chosen to optimize the validation loss after complete training of the model parameters. These approaches have demonstrated that automatic tuning of hyperparameters can yield state-of-the-art performance. However, in general they are not able to effectively optimize more than 10 to 20 hyperparameters.

Why not use gradients? Reverse-mode differentiation allows gradients to be computed with a similar time cost to the original objective function. This approach is taken almost universally for optimization of elementary[1]parameters. The problem with taking gradients with respect to hyperparameters is that computing the validation loss requires an inner loop of elementary optimization, which makes naïve reverse-mode dif-

[1]Since this chapter is about hyperparameters, we need a term to unambiguously denote the other sort of parameter, the "parameter-that-is-just-a-parameter-and-not-a-hyperparameter". After considering "core", "primal", "elemental", "fundamental", "inner" and "vanilla" we settled on "elementary parameter".

ferentiation infeasible from a memory perspective. Section 6.2 describes this problem and proposes a solution, which is the main technical contribution of this chapter.

Once we have access to gradient with respect to hyperparameters, we can embrace hyperparameters rather than strain to eliminate them, and hyperparameterize our models as richly as they deserve. Just as having a high-dimensional elementary parameterization gives a flexible model, having a high-dimensional hyperparameterization gives flexibility over model classes, regularization, and training methods. Section 6.3 explores these new opportunities.

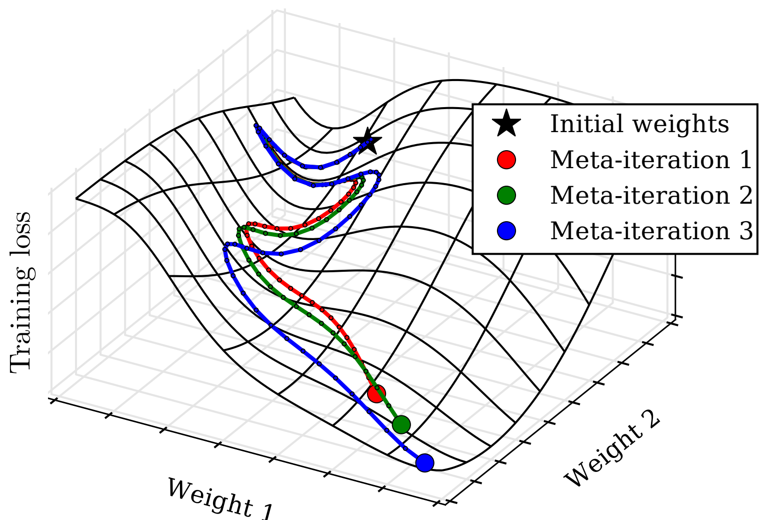



**Figure 6.1:** Hyperparameter optimization by gradient descent. Each meta-iteration runs an entire training run of stochastic gradient descent to optimize elementary parameters (weights 1 and 2). Gradients of the validation loss with respect to hyperparameters are then computed by propagating gradients back through the elementary training iterations. Hyperparameters (in this case, learning rate and momentum schedules) are then updated in the direction of this hypergradient.

### 6.1.1 Contributions

- We give an algorithm that exactly reverses stochastic gradient descent with momentum to compute gradients with respect to all continuous training parameters.

- We show how to efficiently store only the information needed to exactly reverse learning dynamics. For example, when the momentum term is 0.9, this method reduces the memory requirements of reverse-mode differentiation of hyperparameters by a factor of 200.
- We show that these gradients allow optimization of validation loss with respect to thousands of hyperparameters. For example, we optimize fine-grained learning-rate schedules, per-layer initialization distributions of neural network parameters, per-input regularization schemes, and per-pixel data preprocessing.
- We provide insight into learning procedures by examining optimized learning-rate schedules and initialization procedures, comparing them to standard advice in the literature.

## 6.2 Hypergradients

Reverse-mode differentiation (RMD), as described in Chapter 2 has been an asset to the field of machine learning [59]. The RMD method, known as "backpropagation" in the deep learning community, allows the gradient of a scalar loss with respect to its parameters to be computed in a single backward pass. This increases the computational burden by only a small constant factor over evaluating the loss itself, regardless of the number of parameters. Obtaining the same sort of information by either forward-mode differentiation or brute force finite differences would require a separate pass for each parameter and would make deep learning entirely infeasible.

Applying RMD to hyperparameter optimization was proposed by Bengio [10] and Baydin et al. [9], and applied to small problems by Domke [23]. However, the naïve approach fails for real-sized problems because of memory constraints. RMD requires

that intermediate variables be maintained in memory for the reverse pass. Evaluating the validation loss requires training the model, which may require many elementary iterations. Conventional RMD stores this entire training trajectory, $\mathbf{w}_1 ... \mathbf{w}_T$ in memory. In large neural networks, the amount of memory required to store the millions of parameters being trained is typically close to the amount of physical RAM available [102]. If storing the parameter vector takes $\sim$1GB, and the parameter vector is updated tens of thousands of times (the number of mini batches times the number of epochs) then storing the learning history is unmanageable even with physical storage.

Imagine that we could exactly trace a training procedure backwards, starting from the trained parameter values and working back to the initial parameters. Then we could recompute the learning trajectory on the fly during the reverse pass of RMD rather than storing it in memory. This is not possible in general, but we will show that for the popular training procedure of stochastic gradient descent with momentum, we can do exactly this, storing a small number of auxiliary bits to handle finite precision arithmetic.

### 6.2.1 Reversible learning with exact arithmetic

Stochastic gradient descent (SGD) with momentum (Algorithm 5) can be seen as a physical simulation of a system moving through a series of fixed force fields indexed by time $t$. With exact arithmetic this procedure is reversible. This lets us write Algorithm 6, which reverses the steps in Algorithm 5, interleaved with computations of gradients. It outputs the gradient of a function of the trained weights $f(\mathbf{w})$ (such as the validation loss) with respect to the initial weights $\mathbf{w}_1$, the learning-rate and momentum schedules, and any other hyperparameters which affect training gradients.

Computations of steps 11 and 12 both require a Hessian-vector product, but these

**Algorithm 5** Stochastic gradient descent with momentum

```
1: input: initial w_1, decays γ, learning rates α, loss function L(w, θ, t)
2: initialize v_1 = 0
3: for t = 1 to T
4:     g_t = ∇_w L(w_t, θ, t)                        ▷ evaluate gradient
5:     v_{t+1} = γ_t v_t − (1 − γ_t) g_t             ▷ update velocity
6:     w_{t+1} = w_t + α_t v_t                       ▷ update position
7: end for
8: output trained parameters w_T
```

**Algorithm 6** Reverse-mode differentiation of SGD

```
 1: input: w_T, v_T, γ, α, train loss L(w, θ, t), loss f(w)
 2: initialize dv = 0, dθ = 0, dα_t = 0, dγ = 0
 3: initialize dw = ∇_w f(w_T)
 4: for t = T counting down to 1
 5:     dα_t = dw^T v_t
 6:     w_{t−1} = w_t − α_t v_t                  ⎫ exactly reverse
 7:     g_t = ∇_w L(w_t, θ, t)                   ⎬ gradient descent
 8:     v_{t−1} = [v_t + (1 − γ_t) g_t] / γ_t    ⎭ operations
 9:     dv = dv + α_t dw
10:     dγ_t = dv^T (v_t + g_t)
11:     dw = dw − (1 − γ_t) dv ∇_w ∇_w L(w_t, θ, t)
12:     dθ = dθ − (1 − γ_t) dv ∇_θ ∇_w L(w_t, θ, t)
13:     dv = γ_t dv
14: end for
15: output gradient of f(w_T) w.r.t w_1, v_1, γ, α and θ
```

can be computed exactly by applying RMD to the dot product of the gradient with a vector [78]. Thus the time complexity of reverse SGD is $\mathcal{O}(T)$, the same as forward SGD.

### 6.2.2 Reversible learning with finite precision arithmetic

In practice, Algorithm 6 fails utterly due to finite numerical precision. The problem is the momentum decay term $\gamma$. Every time we apply step 8 to reduce the velocity, we lose information. Assuming we are using a fixed-point representation, [2] each multiplication by $\gamma < 1$ shifts bits to the right, destroying the least significant bits. This is more than a pedantic concern. Attempting to carry out the reverse training requires repeated multiplication by $1/\gamma$. Errors accumulate exponentially, and the reversed learning procedure ends far from the initial point (and usually overflows). Do we need $\gamma < 1$? Unfortunately we do. $\gamma > 1$ results in unstable dynamics, and $\gamma = 1$, recovers the leapfrog integrator [43], a perfectly reversible set of dynamics, but one that does not converge.

This problem is quite a deep one: optimization necessarily discards information. Ideally, optimization maps all initializations to the same optimum, a many-to-one mapping with no hope of inversion. Put another way, optimization moves a system from a high-entropy initial state to a low-entropy (hopefully zero entropy) optimized final state.

It is interesting to consider the analogy with physical dynamics. The $\gamma$ term is analogous to a drag term in the simulation of Hamiltonian dynamics. Having $\gamma < 1$ corresponds to *dissipative* dynamics which generates heat, increases the entropy of the environment and is not therefore not reversible. But we must have dissipation in order

[2] We assume fixed-point representation to simplify the discussion (and the implementation). Courbariaux et al. [20] show that fixed-point arithmetic is sufficient to train deep networks. Floating point representation doesn't fix the problem, it just defers the loss of information from the division step to the addition step.

for our system to converge to equilibrium.

If we want to reverse the dynamics, there is no choice but to store the extra bits discarded by the $\gamma$ operation. But we can at least try to be parsimonious about the number of extra bits we store. This is what the next section addresses.

### 6.2.3 Optimal storage of discarded entropy

This section gives the technical details of how to efficiently store the information discarded each time the momentum decay operation (Step 8) is applied. If $\gamma = 0.5$, we can simply store the single bit that falls off at each iteration, and if $\gamma = 0.25$ we could store two bits. But for fine-grained control over $\gamma$ we need a way to store the information lost when we multiply by, say, $\gamma = 0.9$, which will be less than one bit on average. Here we give a procedure which achieves exactly this.

We represent the velocity $\mathbf{v}$ and parameter $\mathbf{w}$ vectors with 64-bit integers. With an implied radix point this can be a fixed-point representation of the reals. We represent $\gamma$ as a rational number, $n/d$. When we divide each $v$ by $d$ we use integer division. In order to be able to reverse the process we just need to store the remainder, $v$ modulo $s$, in some "information buffer", $B$. If $B$ were an integer and $n = 2$, the remainder $r$ would just be a single bit, and we could store it in $B$ by left-shifting $B$'s bits and adding $r$. For arbitrary $n$, we can do the base-$n$ analogue of this operation: multiply $B$ by $n$ and add $r$. Eventually, $B$ will overflow. We need a way to either detect this, store the bits, and start a fresh integer, or else we can just use an arbitrary size integer that grows as needed. (Python's "long" integer type supports this). This procedure allows division by $n$ while storing the remainder in $\log_2(n)$ bits on average.

When we multiply by the numerator of $n/d$ we don't need to store anything extra, since integer division will bring us back to exactly the same point anyway. But the

procedure as it stands would store three bits when $\gamma = 7/8$, whereas it should store less than one ($\log_2(8/7) = 0.19$). Our solution is the following: when we multiply $v$ by $n$, there is an opportunity to add a nonnegative integer smaller than $n$ to the result without affecting the reverse process (integer division by $n$). We can get such an integer from the information buffer by dividing it by $n$ and recording $B$ modulo $n$. We are using the velocity $v$ as an information buffer itself! Algorithm 7 illustrates the entire process.

---

**Algorithm 7** Exactly reversible multiplication by a ratio

```
1: Input: Information buffer i, value c, ratio n/d
2: i = i × d                          ▷ make room for new digit
3: i = i + (c mod d)                  ▷ store digit lost by division
4: c = c ÷ d                          ▷ divide by denominator
5: c = c × n                          ▷ multiply by numerator
6: c = c + (i mod n)                  ▷ add digit from buffer
7: i = i ÷ n                          ▷ shorten information buffer
8: return updated buffer i, updated value c
```

---

We could also have used an arithmetic coding scheme for our information buffer [63, Chapter 6]. How much does this procedure save us? When $\gamma = 0.98$, we will have to store only 0.029 bits on average. Compared to storing a new 32-bit integer or floating-point number at each iteration, this reduces memory requirements by a factor of one thousand.

The standard way to save memory in RMD is checkpointing. Checkpointing stores the entire parameter vector on only a fraction of the training steps, and recomputes the missing steps of the training procedure (forwards) as needed during the backward pass. However, this would require too much memory to be practical for large neural nets trained for thousands of minibatches.

## 6.3 Experiments

In typical machine learning applications, only a few hyperparameters (less than 20) are optimized. Since each experiment only yields a single number (the validation loss), the search rapidly becomes more difficult as the dimension of the hyperparameter vector increases. In contrast, when hypergradients are available, the amount of information gained from each training run grows along with the number of hyperparameters, allowing us to optimize thousands of hyperparameters. How can we take advantage of this new ability?

This section shows several proof-of-concept experiments in which we can more richly parameterize training and regularization schemes in ways that would have been previously impractical to optimize.

### 6.3.1 Gradient-based optimization of gradient-based optimization

Modern neural net training procedures often employ various heuristics to set learning rate schedules, or set their shape using one or two hyperparameters set by cross-validation [21, 101]. These schedule choices are supported by a mixture of intuition, arguments about the shape of the objective function, and empirical tuning.

To more directly shed light on good learning rate schedules, we jointly optimized separate learning rates for *every single learning iteration* of training of a deep neural network, as well as separately for weights and biases in each layer. Each meta-iteration trained a network for 100 iterations of SGD, meaning that the learning rate schedules were specified by 800 hyperparameters (100 iterations $\times$ 4 layers $\times$ 2 types of parameters). To avoid learning an optimization schedule that depended on the quirks of a particular random initialization, each evaluation of hypergradients used a different ran-

dom seed. These random seeds were used both to initialize network weights and to choose mini batches. The network was trained on 10,000 examples of MNIST, and had 4 layers, of sizes 784, 50, 50, and 50.

Because learning schedules can implicitly regularize networks [28], for example by enforcing early stopping, for this experiment we optimized the learning rate schedules on the training error rather than on the validation set error. Figure 6.2 shows the results

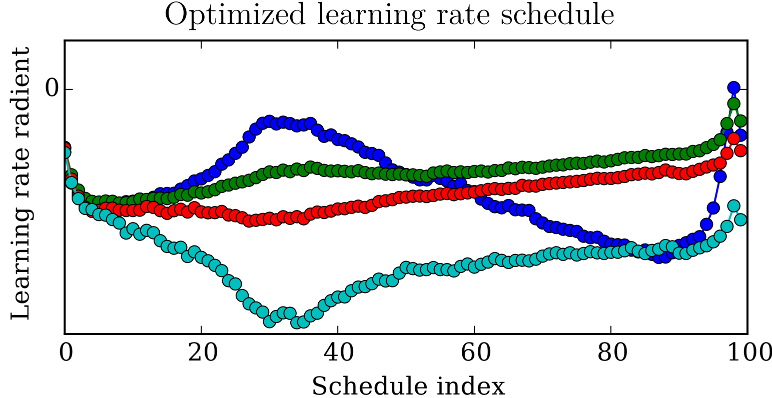



**Figure 6.2:** A learning-rate training schedule for the weights in each layer of a neural network, optimized by hypergradient descent. The optimized schedule starts by taking large steps only in the topmost layer, then takes larger steps in the first layer. All layers take smaller step sizes in the last 10 iterations. Not shown are the schedules for the biases or the momentum, which showed less structure.

of optimizing learning rate schedules separately for each layer of a deep neural network. When Bayesian optimization was used to choose a fixed learning rate for all layers and iterations, it chose a learning rate of 2.4.

Meta-optimization strategies We experimented with several standard stochastic optimization methods for meta-optimization, including SGD, RMSprop [104], and minibatch conjugate gradients. The results in this section used Adam [49], a variant of RMSprop that includes momentum. We typically ran for 50 meta-iterations, and used

a meta-step size of 0.04. Figure 6.3 shows the elementary and meta-learning curves that generated the hyperparameters shown in Figure 6.2.

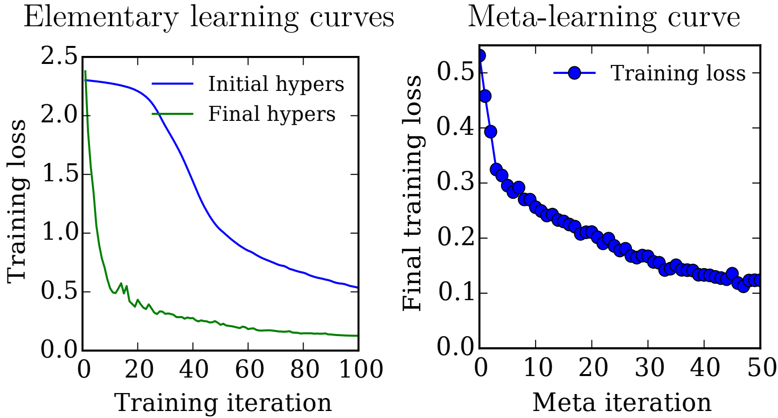



**Figure 6.3:** Elementary and meta-learning curves. The meta-learning curve shows the training loss at the end of each elementary iteration.

How smooth are hypergradients? To demonstrate that the hypergradients are smooth with respect to time steps in the training schedule, Figure 6.4 shows the hypergradient with respect to the step size training schedule at the beginning of training, averaged over 100 random seeds.

Optimizing weight initialization scales We optimized a separate weight initialization scale hyperparameter for each type of parameter (weights and biases) in each layer - a total of 8 hyperparameters. Results are shown in Figure 6.5.

Interestingly, the initialization scale chosen for the first layer weights matches a heuristic which says to choose an initialization scale of $1/\sqrt{N}$, where $N$ is the number of weights in the layer.

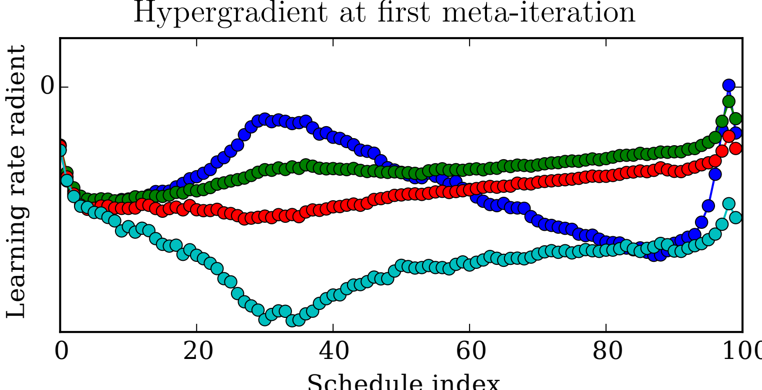



**Figure 6.4:** The initial gradient of the cross-validation loss with respect to the training schedule, averaged over 100 random weight initializations and mini batches. Colors correspond to the same layers as in Figure 6.2.

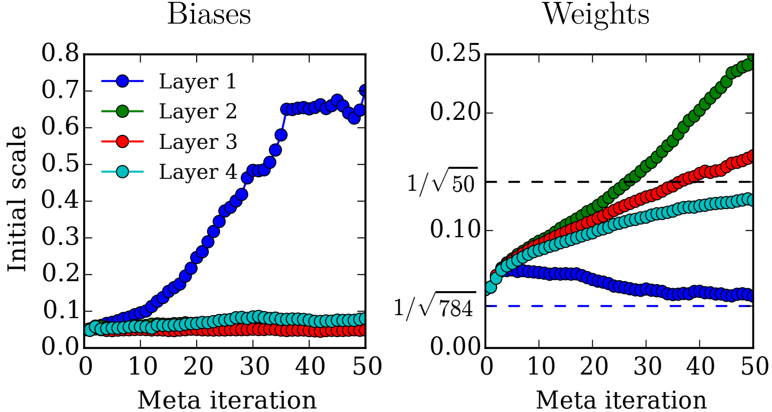



**Figure 6.5:** Meta-learning curves for the initialization scales of each layer in a 4-layer deep neural network. *Left:* Initialization scales for biases. *Right:* Initialization scales for weights. Dashed lines show a heuristic which gives an average total activation of 1. For the first layer it is $(1/\sqrt{784})$ and for subsequent layers $(1/\sqrt{50})$.

### 6.3.2 Optimizing regularization parameters

Regularization is often important for generalization performance. Typically, a single parameter controls a single $L_2$ norm or sparsity penalty on the entire parameter vector of a neural network. Because different types of parameters in different layers play

different roles, it is reasonable to suspect that separate regularization hyperparameter for each parameter type would improve performance. Indeed, Snoek et al. [96] optimized separate regularization parameters for each layer in a neural network, and found that it improved performance.

We can take this idea even further, and introduce a separate regularization penalty for each individual parameter in a neural network. We use a simple model as an example – logistic regression, which can be seen as a neural network without a hidden layer. We choose this model because every weight corresponds to an input-pixel and output-label pair, meaning that these 7,840 hyperparameters might be relatively interpretable. Figure 6.6 shows a set of regularization hyperparameters learned for a logistic regression

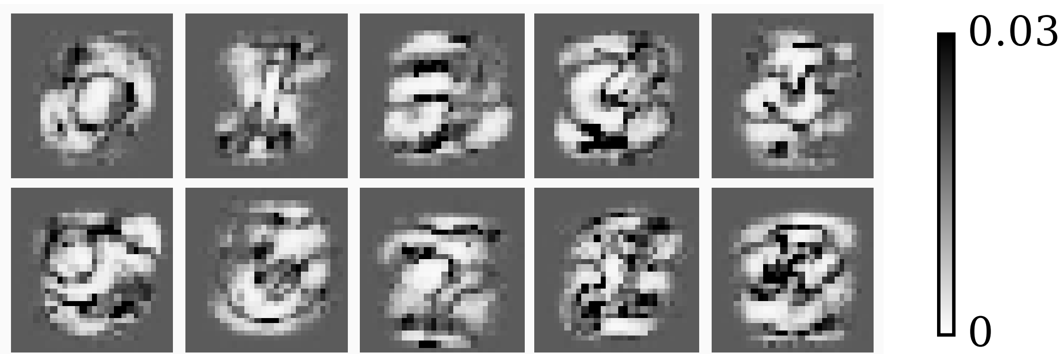



**Figure 6.6:** Optimized $L_2$ regularization hyperparameters for each weight in a logistic regression trained on MNIST. The weights corresponding to each output label (0 through 9 respectively) have been rendered separately. High values (black) indicate strong regularization.

network. Because each parameter corresponds to a particular input, this regularization scheme could be seen as a generalization of automatic relevance determination [62].

### 6.3.3 Optimizing training data

We can use Algorithm 6 to take the gradient with respect to any parameter the training procedure depends on. This includes the training data, which can be viewed as just another set of hyperparameters. By chaining gradients through transformations of the data, we can compute gradients of the validation objective with respect to data preprocessing, weighting, or augmentation procedures.

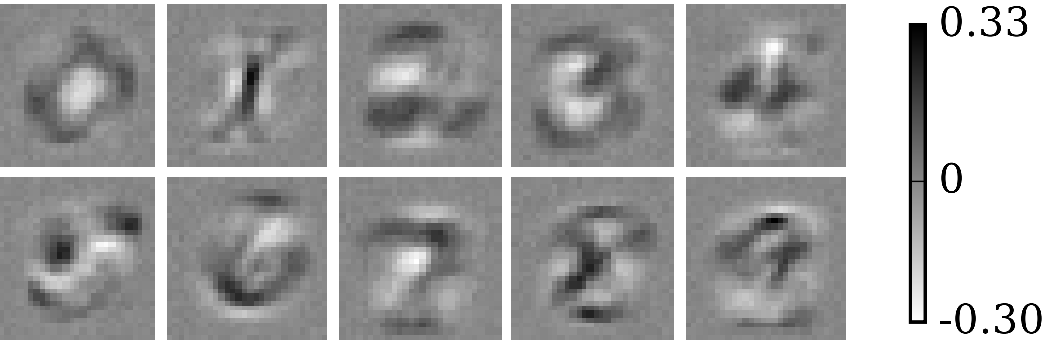



**Figure 6.7:** A dataset generated purely through meta-learning. Each pixel is treated as a hyperparameter, which are all optimized to maximize validation-set performance. Training labels are fixed in order from 0 to 9. Some optimal pixel values are negative.

We demonstrate a simple proof-of-concept where an *entire training set* is learned by gradient descent, starting from blank images. Figure 6.7 shows a training set, the pixels of which were optimized to improve performance on a validation set of 10,000 examples from MNIST. We optimized 10 training examples, each having a different fixed label, again from 0 to 9 respectively. Learning the labels of a larger training set might shed light on which classes are difficult to distinguish and so require more examples.

### 6.3.4 Optimizing initial parameters

The last remaining parameter to SGD is the initial parameter vector. Treating this vector as a hyperparameter blurs the distinction between learning and meta-learning. In the extreme case where all elementary learning rates are set to zero, the training set ceases to matter and the meta-learning procedure exactly reduces to elementary learning on the validation set. Due to philosophical vertigo, we chose not to optimize the initial parameter vector.

### 6.3.5 Learning continuously-parameterized architectures

Many of the notable successes in deep learning have come from novel architectures adapted to particular domains: convolutional neural nets, recurrent neural nets and multitask neural nets. We can think of these architectures as hard constraints that force particular weights to be zero and tie particular pairs of weights together. By softening these hard architectural constraints we can form continuous (but very high-dimensional) parameterizations of architecture. Having access to hypergradients makes learning these softened architectures feasible.

We illustrate this "architecture learning" with a multitask learning problem using the Omniglot data set [56]. This data set consists of 28x28 pixel greyscale images of characters from 50 alphabets with up to 55 characters in each alphabet but only 15 examples of each character. Rather than learning a separate neural net for each alphabet, a multitask approach would be for all the neural nets to share a single first layer, pooling statistical strength to learn generic Gabor-like filters, while maintaining separate higher layers specific to each alphabet.

We can parameterize any architecture based on weight tying or weight absence with

a pairwise quadratic penalty on the weights, $\mathbf{w}^T A \mathbf{w}$, where $A$ is a number-of-weights by number-of-weights matrix. Learning this enormous matrix is clearly infeasible but we can implicitly build such a matrix from lower dimensional structures of manageable size.

For the Omniglot problem, we learn a penalty for each alphabet pair, separately for each neural net layer. Thus, for ten three-layer neural networks, the penalty matrix $A$ is fully described by three ten-by-ten matrices. An architecture with fully independent nets for each alphabet corresponds to three diagonal matrices while an architecture with a mutual lower layer corresponds to two diagonal matrices for the upper layers and a matrix of all ones for the lowest layer (Figure 6.9).

We use five alphabets from the Omniglot set. To see whether our multitask learning system is able to learn high level similarities as well as low-level similarities, we repeat these five alphabets with the images rotated by 90 degrees (Figure 6.8) to make ten alphabets total.

Figure 6.9 shows the learned penalties (normalized by row and column to have ones on the diagonal, akin to a correlation matrix). We see that the lowest layer has been partially shared, across all alphabets equally, with the upper layers much less shared. Interestingly, the top layer penalty learns to share weights between the rotated alphabets.

## 6.4 Limitations

Backpropagation for training neural networks has several pitfalls that were later addressed by analysis and engineering. Likewise, the use of hypergradients also has several apparent difficulties that need to be addressed before it becomes practical. This section explores several issues with this technique that became apparent in our experiments.

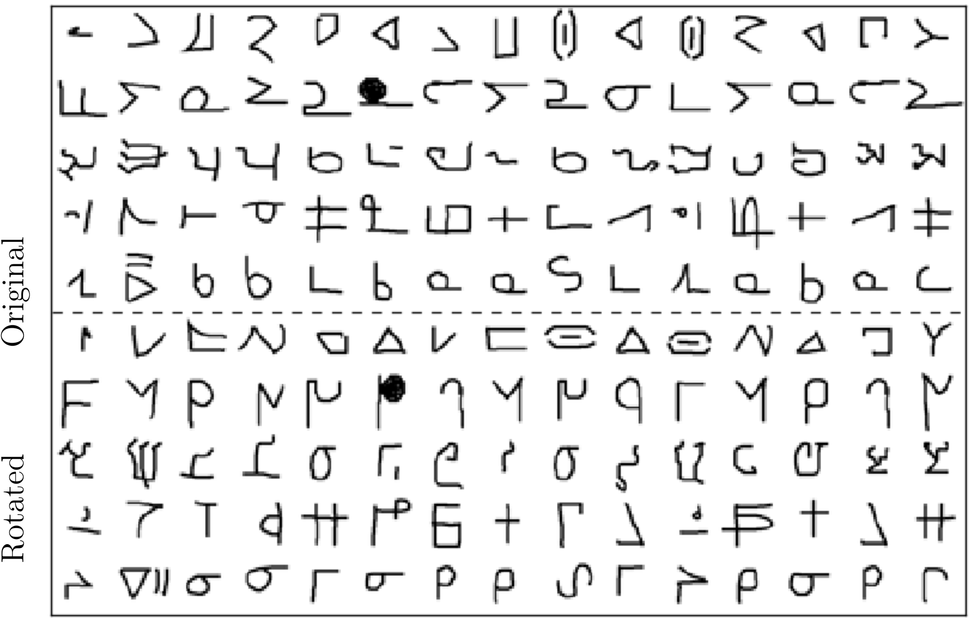



**Figure 6.8:** *Top:* Example characters from 5 alphabets taken from the Omniglot dataset. *Bottom:* Those same alphabets with each character rotated by $90°$. Distinguishing characters within each of these 10 alphabets constitute the 10 tasks in our multi-task learning experiment.

When are gradients meaningful? Bengio et al. [11] noted that "learning long-term dependencies with gradient descent is difficult." Our situation is even worse: We are using gradients to optimize functions which depend on their hyperparameters through hundreds of iterations of SGD. To make things worse, each elementary iteration's gradient itself depends on forward- and then backpropagation through a neural network. Thus the same issues that sometimes make elementary learning difficult are compounded.

For example, Pearlmutter [77, Chapter 4] showed that large learning rates induce chaotic behavior in the learning dynamics, making the gradient uninformative about the medium-term shape of the training objective. This phenomenon is related to the

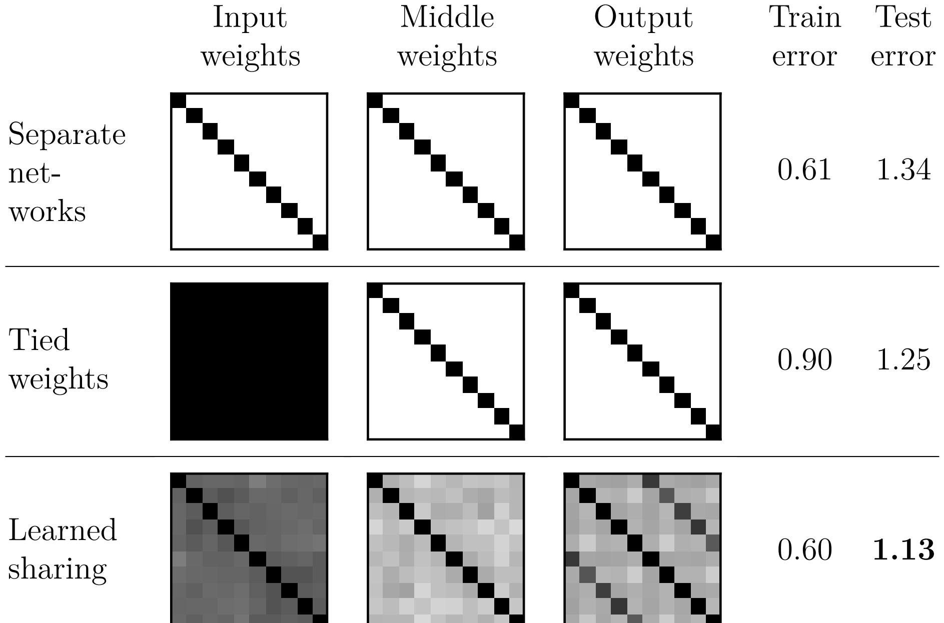



**Figure 6.9:** Results of the Omniglot multitask experiment. Each matrix shows the degree of weight sharing between each pair of tasks for that layer. *Top*: A separate network is trained independently for each task. *Middle*: The lowest-level features were forced to be shared. *Bottom*: The degree of weight sharing between tasks was optimized by hyperparameter optimization.

exploding-gradient problem [75].

Figure 6.10 illustrates this phenomenon when training a neural network having 2 hidden layers for 50 elementary iterations. We partially addressed this problem in our experiments by initializing learning rates to be relatively small, and stopping meta-optimization when the magnitude of the meta-gradient began to grow.

Overfitting How many hyperparameters can we fruitfully optimize? One limitation is overfitting the validation objective, in the same way that optimizing too many parameters can overfit the training objective. However, the same rules of thumb still apply – the size of the validation set, assuming examples are i.i.d., gives a rough guide to how

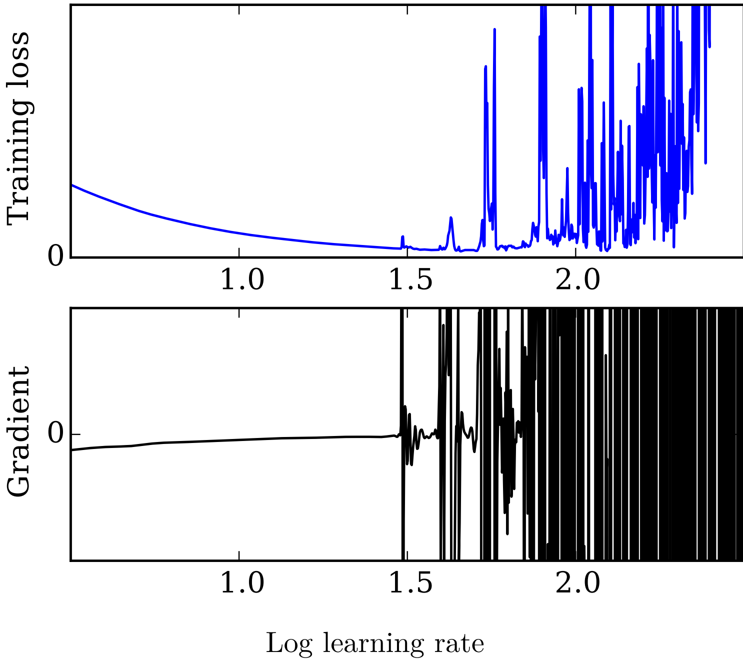



**Figure 6.10:** *Top:* Loss after training as a function of learning rate. *Bottom:* Gradient of loss with respect to learning rate. When the learning rate is high, the gradient becomes uninformative about the medium-term behavior of the function. To maintain stability during meta-learning, we initialize using a small learning rate so as to approach the minimum from the left.

many hyperparameters can be optimized.

Discrete parameters Of course, gradients are not necessarily useful for optimizing discrete hyperparameters such as the number of layers, or hyperparameters that affect discrete changes such as dropout regularization parameters. Some of these difficulties could be addressed by parameterizing apparently discrete choices in a continuous manner. For instance, the per-hidden-unit regularization of section 6.3.2 is an example of a continuous way to choose the number of hidden units.

## 6.5 Related work

The most closely-related work is Domke [23], who derived algorithms to compute reverse-mode derivatives of gradient descent with momentum and L-BFGS, using them to update the hyperparameters of CRF image models. However, his approach relied on naïve caching of all parameter vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_T$, making it impractical for large models with many training iterations.

Larsen et al. [57], Eigenmann & Nossek [27], Chen & Hagan [19], Bengio [10], Abdel-Gawad & Ratner [1], and Foo et al. [29] showed that gradients of regularization parameters are available in closed form when training has converged exactly to a local minimum. In contrast, our procedure can compute exact gradients of any type of hyperparameter, whether or not learning has converged.

Support vector machines Chapelle et al. [18] introduced a differentiable bound on the SVM loss in order to be able to compute derivatives with respect to hundreds of hyperparameters, including weighting parameters for each input dimension in the kernel. However, this bound was not tight, since optimizing the SVM objective requires a discrete selection of training points.

Bayesian methods For Bayesian models with a closed-form marginal likelihood, gradients with respect to all continuous hyperparameters are usually available. For example, this ability has been used to construct complex kernels for Gaussian process models [85, Chapter 5]. Variational inference also allows gradient-based tuning of hyperparameters in Bayesian neural-network models such as deep Gaussian processes [39]. However, it does not provide gradients with respect to training parameters.

Gradients with respect to Markov chain parameters Salimans et al. [94] tune the step-size and mass-matrix parameters of Hamiltonian Monte Carlo by chaining gradients from a lower bound on the marginal likelihood through several iterations of leapfrog dynamics. Because they used only a small number of steps, all intermediate values could be stored naïvely. Our reversible-dynamics memory-tape approach could be used to dramatically extend the number of HMC iterations used in this approach.

## 6.6 Extensions and future work

Bayesian optimization with gradients Hypergradients could be used with parallel, model-based optimization of hyperparameters. For example, Gaussian-process-based optimization methods could incorporate gradient information [99]. Such methods could make use of parallel evaluations of hypergradients, which might be too slow to evaluate in a sequential manner.

Reversible elementary computation Recurrent neural network models can require so much memory to differentiate that checkpointing is required simply to compute their elementary gradients [66]. Reversible computation might offer memory savings for some architectures. For example, evaluations of Long Short-Term Memory [42] or a Neural Turing Machines [34] rely on long chains of mostly-small updates of parameters. Exactly reversing these dynamics might allow more memory-efficient elementary gradient evaluations of their outputs on very long input sequences.

Exactly reversing other learning methods The memory saving trick from Section 6.2.3 could presumably be applied to other momentum-based variants of SGD such as RMSprop [104] or Adam [49].

## 6.7 Conclusion

In this chapter, we derived a computationally efficient procedure for computing gradients through stochastic gradient descent with momentum. We showed how the approximate reversibility of learning dynamics can be used to drastically reduce the memory requirement for exactly backpropagating gradients through hundreds of training iterations.

We showed how these gradients allow the optimization of validation loss with respect to thousands of hyperparameters, something which was previously infeasible. This new ability allows the automatic tuning of most details of training neural networks. We demonstrated the tuning of detailed training schedules, regularization schedules, and neural network architectures.

# 7

# Early Stopping as Variational Inference

In the previous chapter we used reversible learning as a trick for performing reverse mode differentiation without storing the intermediate states and we were forced to account for the entropy reduction in order to do the reversal exactly. But keeping track of entropy this way can be a powerful tool in itself. In this chapter we use this entropy accounting to estimate the variational lower bound of the distribution associated with an optimization procedure.

We show that unconverged stochastic gradient descent can be interpreted as a procedure that samples from a nonparametric approximate posterior distribution. This distribution is implicitly defined by the transformation of an initial distribution by a sequence of optimization steps. By tracking the change in entropy over these distribu-

tions during optimization, we form a scalable, unbiased estimate of a variational lower bound on the log marginal likelihood. This bound can be used to optimize hyper-parameters instead of cross-validation. This Bayesian interpretation of SGD suggests improved, overfitting-resistant optimization procedures, and gives a theoretical foundation for early stopping and ensembling. We investigate the properties of this marginal likelihood estimator on neural network models.

This chapter presents work with David Duvenaud and Ryan P. Adams. David and I came up with the ideas, wrote the implementation, performed the experiments and wrote the paper together. It will be presented at AISTATS in May 2016 [25].

## 7.1 Introduction

In much of machine learning, the central computational challenge is optimization: we try to minimize some training-set loss with respect to a set of model parameters. If we treat the training loss as a negative log-posterior, this amounts to searching for a maximum *a posteriori* (MAP) solution. Paradoxically, over-zealous optimization can yield worse test-set results than incomplete optimization due to the phenomenon of *over-training*. A popular remedy to over-training is to invoke "early stopping" in which optimization is halted based on the continually monitored performance of the parameters on a separate validation set. However, early stopping is both theoretically unsatisfying and incoherent from a research perspective: how can one rationally design better optimization methods if the goal is to achieve something "powerful but not *too* powerful"? A related trick is to ensemble the results from multiple optimization runs from different starting positions. Similarly, this must rely on imperfect optimization, since otherwise all optimization runs would reach the same optimum.

We propose an interpretation of incomplete optimization in terms of variational

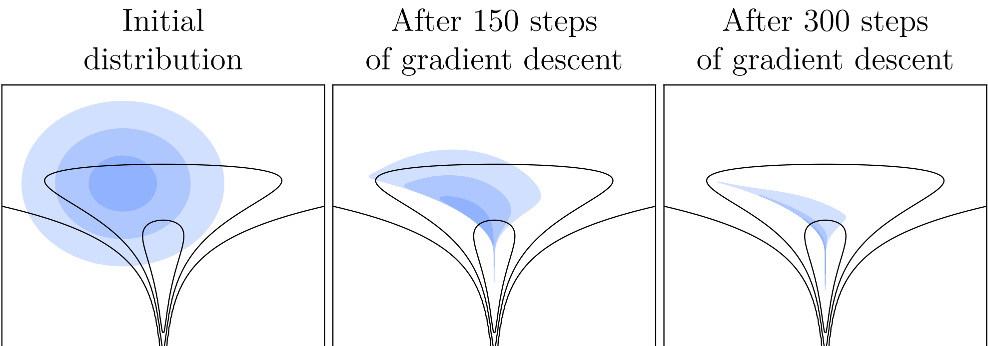



**Figure 7.1:** A series of distributions (blue) implicitly defined by gradient descent on an objective (black). These distributions are defined by mapping each point in the initial distribution through a fixed number of iterations of optimization. These distributions have nonparametric shapes, and eventually concentrate around the optima of the objective.

Bayesian inference, and provide a simple method for estimating the marginal likelihood of the approximate posterior. Our starting point is a Bayesian posterior distribution for a potentially complicated model, in which there is an empirical loss that can be interpreted as a negative log likelihood and regularizers that have interpretations as priors. One might proceed with MAP inference, and perform an optimization to find the best parameters. The main idea of this chapter is that such an optimization procedure, initialized according to some distribution that can be chosen freely, generates a sequence of distributions that are implicitly defined by the action of the optimization update rule on the previous distribution. We can treat these distributions as variational approximations to the true posterior distribution. A single optimization run for $N$ iterations represents a draw from the $N$th such distribution in the sequence. Figure 7.1 shows contours of these approximate distributions on an example posterior.

With this interpretation, the number of optimization iterations can be seen as a variational parameter, one that trades off fitting the data well against maintaining a broad (high entropy) distribution. Early stopping amounts to optimizing the variational lower

bound (or an approximation based on a validation set) with respect to this variational parameter. Ensembling different random restarts can be viewed as taking independent samples from the variational posterior.

To establish whether this viewpoint is helpful in practice, we ask: can we efficiently estimate the marginal likelihood implied by unconverted optimization? We tackle this question in section 7.2. Specifically, for stochastic gradient descent (SGD), we show how to compute an unbiased estimate of a lower bound on the log marginal likelihood of each iteration's implicit variational distribution. We also introduce an 'entropy-friendly' variant of SGD that maintains better-behaved implicit distributions.

We also ask whether model selection based on these marginal likelihood estimates picks models with good test-time performance. We give some experimental evidence in both directions in section 7.5. A related question is how close the variational distributions implied by various optimization rules approximate the true posterior. We briefly address this question in section 7.6.

### 7.1.1 Contributions

- We introduce a new interpretation of optimization algorithms as samplers from a variational distribution that adapts to the true posterior, eventually collapsing around its modes.

- We provide a scalable estimator for the entropy of these implicit variational distributions, allowing us to estimate a lower bound on the marginal likelihood of any model whose posterior is twice-differentiable, even on problems with millions of parameters and data points.

- In principle, this marginal likelihood estimator can be used for hyperparameter

selection and early stopping without the need for a validation set. We investigate the performance of these estimators empirically on neural network models, and show that they have reasonable properties. However, further refinements are likely to be necessary before this marginal likelihood estimator is more practical than using a validation set.

## 7.2 Incomplete optimization as variational inference

As explained in Chapter 2, variational inference aims to approximate an intractable posterior distribution, $p(\theta|D)$, with more tractable distribution, $q(\theta)$, by maximizing the variational lower bound:

$$\mathcal{L}[q] \equiv \underbrace{\mathbb{E}_q\left[-\log q(\theta)\right]}_{\text{Entropy } S[q]} - \underbrace{\mathbb{E}_q\left[-\log p(\theta, D)\right]}_{\text{Energy } E[q]}. \tag{7.1}$$

We propose a nonparametric variational family, $q(\theta)$, defined as follows. Consider a general procedure to minimize the energy $(-\log p(\theta, D))$ with respect to $\theta \in \mathbb{R}^D$. The parameters $\theta$ are initialized according to some distribution $q_0(\theta)$ and updated at each iteration according to a transition operation $T : \mathbb{R}^D \to \mathbb{R}^D$:

$$\theta_0 \sim q_0(\theta)$$
$$\theta_{t+1} = T(\theta_t).$$

Our variational family consists of the sequence of distributions $q_0, q_1, q_2, \ldots$, where $q_t(\theta)$ is the distribution over $\theta_t$ generated by the above procedure. These distributions don't have a closed form, but we can exactly sample from $q_t$ by simply running the optimizer for $t$ steps starting from a random initialization.

As shown in (7.1), $\mathcal{L}$ consists of an energy term and an entropy term. The energy term measures how well $q$ fits the data and the entropy term encourages the probability mass of $q$ to spread out, preventing overfitting. As optimization of $\theta$ proceeds from its $q_0$-distributed starting point, we can examine how $\mathcal{L}$ changes. The negative energy term grows, since the goal of the optimization is to reduce the energy. The entropy term shrinks because the optimization converges over time. Optimization thus generates a sequence of distributions that range from underfitting to overfitting, and the variational lower bound captures this tradeoff.

We cannot evaluate $\mathcal{L}[q_t]$ exactly, but we can obtain an unbiased estimator. Sampling $\theta_0$ from $q_0$ and then applying the transition operator $t$ times produces an exact sample $\theta_0$ from $q_t$, by definition. Since $\theta_t$ is an exact sample from $q_t(\theta)$, $\log p(\theta_t, D)$ is an unbiased estimator of the energy term of (7.1). The entropy term is trickier, since we do not have access to the density $q(\theta)$ directly. However, if we know the entropy of the initial distribution, $S[q_0(\theta)]$, then we can estimate $S[q_t(\theta)]$ by tracking the change in entropy at each iteration, calculated by the change of variables formula.

To compute how the volume shrinks or expands due to an iteration of the optimizer, we require access to the Jacobian of the optimizer's transition operator, $J(\theta)$:

$$S[q_{t+1}] - S[q_t] = \mathbb{E}_{q_t(\theta_t)}\big[\log|J(\theta_t)|\big]\,. \tag{7.2}$$

Note that this analysis assumes that the mapping $T$ is bijective. Combining these terms, we have an unbiased estimator of $\mathcal{L}$ at iteration $T$, based on the sequence of parameters,

$\theta_0, \ldots, \theta_T$, from a single training run:

$$\mathcal{L}[q_T] \approx \underbrace{\log p(\theta_T, D)}_{\text{Energy}} + \underbrace{\sum_{t=0}^{T-1} \log |J(\theta_t)| + S[q_0]}_{\text{Entropy}} . \tag{7.3}$$

## 7.3 The entropy of stochastic gradient descent

In this section, we give an unbiased estimate for the change in entropy caused by SGD updates. We'll start with a naïve method, then in section 7.3.1, we give an approximation that scales linearly with the number of parameters in the model.

Stochastic gradient descent is a popular and effective optimization procedure with the following update rule:

$$\theta_{t+1} = \theta_t - \alpha \nabla L(\theta), \tag{7.4}$$

where the $L(\theta)$ the objective loss (or an unbiased estimator of it e.g. using minibatches) for example $-\log p(\theta, D)$, and $\alpha$ is a 'step size' hyperparameter. Taking the Jacobian of this update rule gives the following unbiased estimator for the change in entropy at each iteration:

$$S[q_{t+1}] - S[q_t] \approx \log |I - \alpha H_t(\theta_t)| \tag{7.5}$$

where $H_t$ is the Hessian of $-\log p_t(\theta, D)$ with respect to $\theta$.

Note that the Hessian does not need to be positive definite or even non-singular. If some directions in $\theta$ have negative curvature, as on the crest of a hill, it just means that optimization near there spreads out probability mass, increasing the entropy. There

**Algorithm 8** stochastic gradient descent with entropy estimate

1: **input:** Weight initialization scale $\sigma_0$, step size $\alpha$, twice-differentiable negative log-likelihood $L(\theta, t)$
2: **initialize** $\theta_0 \sim \mathcal{N}(0, \sigma_0 \mathbf{I}_D)$
3: **initialize** $S_0 = \frac{D}{2}(1 + \log 2\pi) + D \log \sigma_0$
4: **for** $t = 1$ **to** $T$
5: $\quad S_t = S_{t-1} + \log |\mathbf{I} - \alpha H_{t-1}|$ ▷ Update entropy
6: $\quad \theta_t = \theta_{t-1} - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ Update parameters
7: **end for**
8: **output** sample $\theta_T$, entropy estimate $S_T$

are, however, restrictions on $\alpha$. If $\alpha\lambda_i = 1$, for any $i$, where $\lambda_i$ are the eigenvalues of $H_t$, then the change in entropy will be undefined (infinitely negative). This corresponds to a Newton-like update where multiple points collapse to the optimum in a single step giving a distribution with zero variance in a particular direction. However, gradient descent is unstable anyway if $\alpha\lambda_{\max} > 2$, where $\lambda_{\max}$ is the largest eigenvalue of $H_t$. So if we choose a sufficiently conservative step size, such that $\alpha\lambda_{\max} < 1$, this situation should not arise. Algorithm 8 combines these steps into an algorithm that tracks the approximate entropy during optimization.

So far, we have treated SGD as a deterministic procedure even though, as the name suggests, the gradient of the loss at each iteration may be replaced by a stochastic version. Our analysis of the entropy is technically valid if we fix the sequence of stochastic gradients to be the same for each optimization run, so that the only randomness comes from the parameter initialization. This is a tendentious argument, similar to arguing that a pseudorandom sequence of numbers has only as much entropy as its seed. However, if we do choose to randomize the gradient estimator differently for each training run (e.g. choosing different minibatches) then the expression for the change in entropy, Equation 7.5, remains valid as a *lower bound* on the change in entropy and the

subsequent calculation of $\mathcal{L}$ remains a true lower bound on the log marginal likelihood.

### 7.3.1 Estimating the Jacobian in high dimensions

The expression for the change in entropy given by (7.5) is impractical for large-scale problems since it requires an $\mathcal{O}\left(D^3\right)$ determinant computation. Fortunately, we can make a good approximation using just two Hessian-vector products, which can usually be performed in $\mathcal{O}\left(D\right)$ time using reverse-mode differentiation [78].

The idea is that since $\alpha\lambda_{\text{max}}$ is small, the Jacobian is just a small perturbation to the identity, and we can approximate its determinant using traces as follows:

$$\begin{aligned}\log|I-\alpha H| &= \sum_{i=0}^{D}\log\left(1-\alpha\lambda_i\right)\\ &\geq \sum_{i=0}^{D}\left[-\alpha\lambda_i-(\alpha\lambda_i)^2\right] \qquad (7.6)\\ &= -\alpha\text{Tr}\left[H\right]-\alpha^2\text{Tr}\left[HH\right]\,. \qquad (7.7)\end{aligned}$$

The bound in (7.6) is just a second order Taylor expansion of $\log(1-x)$ about $x=0$ and is valid if $\alpha\lambda_i < 0.68$. As we argue above, the regime in which SGD is stable requires that $\alpha\lambda_{\text{max}} < 1$, so again choosing a conservative learning rate keeps this bound in the correct direction. For sufficiently small learning rates, this bound becomes tight.

The trace of the Hessian can be estimated using inner products of random vectors [6]:

$$\text{Tr}\left[H\right]=\mathbb{E}\left[\mathbf{r}^T H\mathbf{r}\right], \qquad \mathbf{r}\sim\mathcal{N}(0,I)\,. \qquad (7.8)$$

To see that this is a valid estimator, consider a basis in which $H$ is diagonal. We use

**Algorithm 9** linear-time estimate of log-determinant of Jacobian of one iteration of stochastic gradient descent

1: **input:** step size $\alpha$, current parameter vector $\theta$, twice-differentiable negative log-likelihood $L(\theta)$
2: **initialize** $\mathbf{r}_0 \sim \mathcal{N}(0, \sigma_0 \mathbf{I}_D)$
3: $\mathbf{r}_1 = \mathbf{r}_0 - \alpha \mathbf{r}_0^{\mathsf{T}} \nabla\nabla L(\theta, t)$
4: $\mathbf{r}_2 = \mathbf{r}_1 - \alpha \mathbf{r}_1^{\mathsf{T}} \nabla\nabla L(\theta, t)$
5: $\hat{\mathcal{L}} = \mathbf{r}_0^{\mathsf{T}} \left(-2\mathbf{r}_0 + 3\mathbf{r}_1 - \mathbf{r}_2\right)$
6: **output** $\hat{\mathcal{L}}$, an unbiased estimate of a parabolic lower bound on the change in entropy.

this estimator to derive algorithm 9. In high dimensions, the exact evaluation of the determinant in step 5 should be replaced with the approximation given by algorithm 9.

Note that the quantity we are estimating (7.5) is well-conditioned, in contrast to the related problem of computing the log of the determinant of the Hessian itself. This arises, for example, in making the Laplace approximation to the posterior [61]. This is a much harder problem since the Hessian can be arbitrarily ill-conditioned, unlike our small Hessian-based perturbation to the identity.

### 7.3.2 Parameter initialization, priors, and objective functions

What initial parameter distribution should we use for SGD? The marginal likelihood estimate given by (7.3) is valid no matter which initial distribution we choose. We could conceivably optimize this distribution in an outer loop using the marginal likelihood estimate itself.

However, using the prior as the initialization distribution has several advantages. First, it is usually designed to have broad support. Since SGD usually decreases entropy, starting with a high-entropy distribution is a good heuristic.

The second advantage has to do with our choice of objective function. One option

is to use the unnormalized log-posterior, but we can use any function we like. A more sensible choice is the negative log-likelihood. Variational distributions only differ from the initial distribution to the extent that the posterior differs from the prior. This difference is just the log-likelihood.

One nice implication of using the log-likelihood as the objective function is that the entropy estimate will be exactly correct for parameters that don't affect the likelihood, since their gradient (and corresponding rows of the Hessian) will always be zero. Because of these favorable properties, we use the prior as the initial distribution and log-likelihood as the objective in our experiments.

## 7.4 Entropy-friendly optimization methods

SGD optimizes the training loss, not the variational lower bound. In some sense, if this optimization happens to create a good intermediate distributions, it's only by accident! Why not design a new optimization method that produces good variational lower bounds? In place of SGD, we can use any optimization method for which we can approximate the change in entropy, which in practice means any optimization for which we can compute Jacobian-vector products.

An obvious place to start is with stochastic update rules inspired by Markov chain Monte Carlo (MCMC). Procedures like Hamiltonian Monte Carlo [73] and Langevin dynamics MCMC [113] look very much like optimization procedures but actually have the posterior as their stationary distribution. This is exactly the approach taken by Salimans et al. [94]. One difficulty with using stochastic updates, however, is that calculating the change in entropy at each iteration requires access to the current distribution over parameters. As an example, consider that convolving a delta function with a Gaussian yields an infinite entropy increase, whereas convolving a broad uniform

distribution with a Gaussian yields only a small increase in entropy. Welling & Teh [113] handle this by learning a highly parameterized "inverse model" which implicitly models the distribution over parameters. The downside of this approach is that the parameters of this model must be learned in an outer loop.

Another approach is to try to develop deterministic update rules that avoid some of the pathologies of update rules like SGD. This could could be a research agenda in itself, but we give one example here of a modification to SGD which can improve the variational lower bound. One problem with SGD in the context of posterior approximation is that SGD can collapse the implicit distribution into low-entropy filaments, shrinking in some directions to be orders of magnitude smaller than the width of the true posterior. A simple trick to prevent this is to apply a nonlinear, parameter-wise warping to the gradient, such that directions of very small gradient do not get optimized all the way to the optimium. For example, the modified gradient (and resulting modified Jacobian) could be

$$g' = g - g_0 \tanh\left(g/g_0\right) \tag{7.9}$$

$$J' = \left(1 - \cosh^{-2}(g/g_0)\right) J \tag{7.10}$$

where $g_0$ is a "gradient threshold" parameter that sets the scale of this shrinkage. The effect is that entropy is not removed from parameters which are close to their optimum. An example showing the effect of this entropy-friendly modification is shown in Figure 7.2.

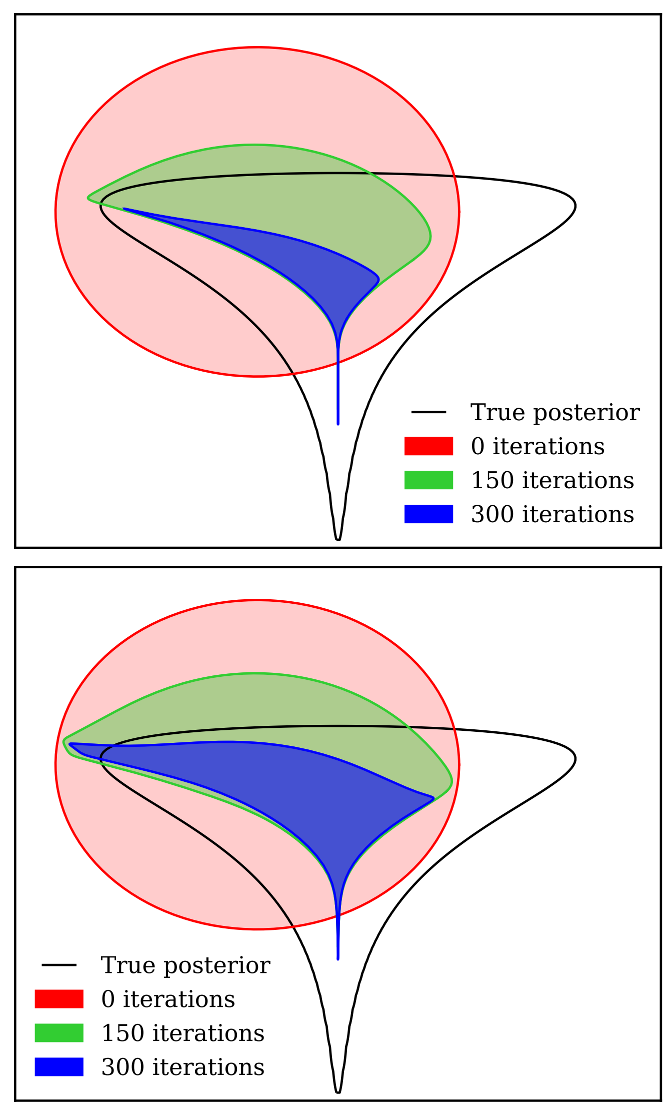



**Figure 7.2:** *Left*: The distribution implied by standard gradient descent. *Right*: The distribution implied by the modified, "entropy-friendly", gradient descent algorithm. The entropy-friendly distributions are slower to collapse into low-entropy filaments, causing the marginal likelihood to remain higher.

## 7.5 Experiments

In this section we show that the marginal likelihood estimate can be used to choose when to stop training, to choose model capacity, and to optimize training hyperparam-

eters without the need for a validation set. We are not attempting to motivate SGD variational inference as a superior alternative to other procedures; we simply wish to give a proof of concept that the marginal likelihood estimator has reasonable properties. Further refinements are likely to be necessary before this marginal likelihood estimator is more practical than simply using a validation set.

### 7.5.1 Choosing when to stop optimization

As a simple demonstration of the usefulness of our marginal likelihood estimate, we show that it can be used to estimate the optimal number of training iterations before overfitting begins. We performed regression on the Boston housing dataset using a neural network with one hidden layer having 100 hidden units, sigmoidal activation functions, and no regularization. Figure 7.3 shows that marginal likelihood peaks at a similar place to the peak of held-out log-likelihood, which is where early stopping would occur when using a large validation set.

### 7.5.2 Choosing the number of hidden units

The marginal likelihood estimate is also comparable between training runs, allowing us to use it to select model hyperparameters, such as the number of hidden units.

Figure 7.4 shows marginal likelihood estimates as a function of the number of hidden units in the hidden layer of a neural network trained on 50,000 MNIST handwritten digits. The largest network trained in this experiment contains 2 million parameters.

The marginal likelihood estimate begins to decrease for more than 30 hidden units, even though the test-set likelihood in maximized at 300 hidden units. We conjecture that this is due to the marginal likelihood estimate penalizing the loss of entropy in parameters whose contribution to the likelihood was initially large, but were made

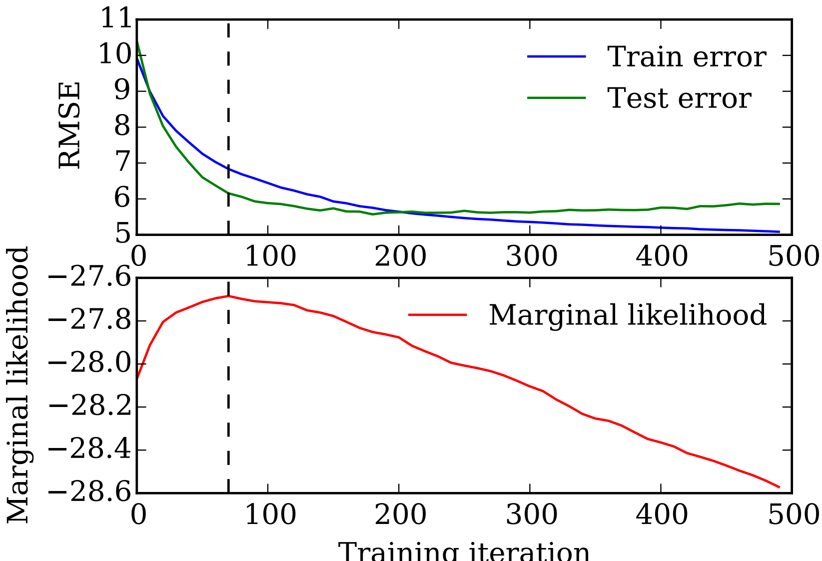



**Figure 7.3:** *Top*: Training and test-set error on the Boston housing dataset. *Bottom*: Stochastic gradient descent marginal likelihood estimates. The dashed line indicates the iteration with highest marginal likelihood. The marginal likelihood, estimated online using only the training set, and the test error peak at a similar number of iterations.

irrelevant later in the optimization.

### 7.5.3 Optimizing training hyperparameters

We can also use marginal likelihoods to optimize training parameters such as learning rates, initial distributions, or any other optimization parameters. As an example, Figure 7.5 shows the marginal likelihood estimate as a function of the gradient threshold in the entropy-friendly SGD algorithm from section 7.4 trained on 50,000 MNIST handwritten digits.

As the level of thresholding increases, the training and test error get worse due to under-fitting. However, for intermediate thresholds, the lower bound increases. Because it is a lower bound, its increase means that the estimate of the marginal likelihood is

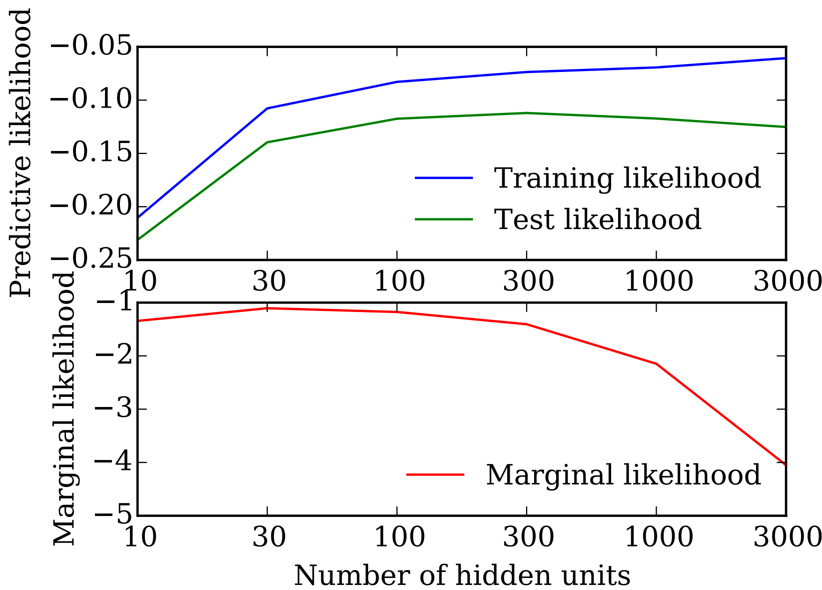



**Figure 7.4:** *Top*: Training and test-set likelihood as a function of the number of hidden units in the first layer of a neural network. *Bottom*: Stochastic gradient descent marginal likelihood estimates. In this case, the marginal likelihood over-penalizes high numbers of hidden units.

becoming *more accurate*, even though the actual model happens to be getting worse at the same time.

### 7.5.4 Implementation details

To compute Hessian-vector products in our models, we used autograd, a reverse-mode automatic differentiation package for Python capable of arbitrary-order derivatives.

## 7.6 Limitations

In practice, the marginal likelihood estimate we present might not be useful for several reasons. First, using only a single sample to estimate both the expected likelihood as well as the entropy of an entire distribution will necessarily have high variance under

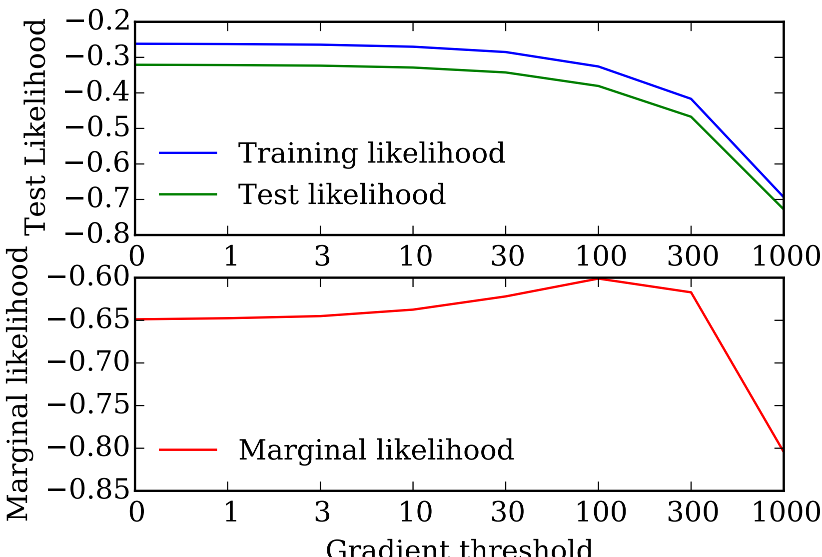



**Figure 7.5:** *Top*: Training and test-set likelihood as a function of the gradient threshold. *Bottom*: Marginal likelihood as a function of the gradient threshold. A gradient threshold of zero corresponds to standard SGD. The increased lower bound for non-zero thresholds indicates that the entropy-friendly variant of SGD is producing a better implicit variational distribution.

some circumstances. These problems could conceivably be addressed by ensembling, which has an interpretation as taking multiple exact independent samples from the implicit posterior.

Second, as parameters converge, their entropy estimate (and true entropy) will continue to decrease indefinitely, making the marginal likelihood arbitrarily small. However, in practice there is usually a limit to the degree of overfitting possible. This raises the question: when are marginal likelihoods a good guide to predictive accuracy? Presumably the marginal likelihood is more likely to be correlated with predictive performance when the implicit distribution has moderate amounts of entropy. In section 7.4 we modified SGD to be less prone to produce regions of pathologically low entropy, but a more satisfactory solution is probably possible.

Third, if the model includes a large number of parameters that do not affect the predictive likelihood, but which are still affected by a regularizer, their convergence will penalize the marginal likelihood estimate even though these parameters do not affect test set performance. This is why in section 7.3.2 we recommend optimizing only the log-likelihood, and incorporating the regularizer directly into the initialization procedure. More generally however, entropy could be underestimated if a large group of parameters are initially constrained by the data, but are later "turned off" by some other parameters in the model.

Finally, how viable is optimization as an inference method? Standard variational methods find the best approximation in some class, but SGD doesn't even try to produce a good approximate posterior, other than by seeking the modes. Indeed, Figure 7.1 shows that the distribution implied by SGD collapses to a small portion of the true posterior early on, and mainly continues to shrink as optimization proceeds. However, the point of early stopping is not that the intermediate distributions are particularly good approximations, but simply that they are better than the point masses that occur when optimization has converged.

## 7.7 Related work

Estimators for early stopping Stein's unbiased risk estimator (SURE) [100] provides an unbiased estimate of generalization performance under broad conditions, and can be used to construct a stopping rule. Raskutti et al. [84] derived a SURE estimate for SGD in a regression setting. Interestingly, this estimator depends on the 'shrinkage matrix' $\prod_{t=0}^{T} (\mathbf{I} - \alpha_t H_T)$, which is just the Jacobian of the entire SGD procedure along a particular path. However, this estimator depends on an estimate of the noise variance, and is restricted to the i.i.d. regression setting. It's not clear if this stopping rule could

also be used to select other training parameters or model hyperparameters.

Reversible learning  Optimization is an intrinsically information-destroying process, since a (good) optimization procedure maps any initial starting point to one or a few final optima. We can quantify this loss of information by asking how many bits must be stored in order to reverse the optimization, as in Maclaurin et al. [65]. We can think of the number of bits needed to exactly reverse the optimization procedure as the average number of bits 'learned' during the optimization.

From this perspective, stopping before optimization converges can be seen as a way to limit the number of bits we try to learn about the parameters from the data. This is a reasonable strategy, since we don't expect to be able to learn more than a finite number of bits from a finite dataset. This is also an example of reducing the hypothesis space to improve generalization.

MCMC for variational inference  Our method can be seen as a special case of Salimans et al. [94], who showed that any set of stochastic dynamics, even those not satisfying detailed balance, can be used to implicitly define a variational distribution. However, to provide a tight variational bound, one needs to estimate the entropy of the resulting implicit distribution. Salimans et al. [94] do this by defining an inverse model which estimates backwards transition probabilities, and then optimizes this model in an outer loop. In contrast, our dynamics are deterministic, and our estimate of the entropy has a simple fixed form.

Bayesian neural networks  Variational inference has been performed in Bayesian neural-network models [33, 39, 40]. Kingma & Welling [50] show how neural networks having unknown weights can be reformulated as neural networks having known weights

but stochastic hidden units, and exploit this connection to preform efficient gradient-based inference in Bayesian neural networks.

Black-box stochastic variational inference Kucukelbir et al. [54] introduce a general scheme for variational inference using only the gradients of the log-likelihood of a model. However, they constrain their variational approximation to be Gaussian, as opposed to our free-form variational distribution.

SGD as an estimator Hardt et al. [37] give theoretical results showing that the smaller the number of training epochs, the better the generalization performance of models trained using SGD. Toulis et al. [105] examine the properties of SGD as an estimator, and show that a variant that averages parameter updates has improved statistical efficiency.

## 7.8 Future work and extensions

Optimization with momentum One obvious extension would be to design an entropy estimator of momentum-based optimizers such as stochastic gradient descent with momentum, or refinements such as Adam [49]. However, it is difficult to track the entropy change during the updates to the momentum variables.

Gradient-based hyperparameter optimization Optimizing marginal likelihood rather than training loss lets us choose both training and regularization parameters without using a validation set. However, optimizing more than a few hyperparameters is difficult without gradients. Following Domke [23] and Maclaurin et al. [65], we could compute exact gradients of the variational lower bound with respect to all variational parameters using reverse-mode differentiation through SGD. Chaining gradients through

SGD would allow one to set all hyperparameters using gradient-based optimization without the need for a validation set.

Stochastic dynamics One possible method to deal with over-zealous reduction in entropy by SGD would be to add noise to the dynamics. In the case of Gaussian noise, we would recover Langevin dynamics [73]. However, estimating the entropy is more difficult in this case. Welling & Teh [113] introduced stochastic gradient Langevin dynamics for doing inference with minibatches, but do not track the entropy of the implicit distribution.

More generally, we are free to design optimization algorithms that do a better job of producing samples from the true posterior, as long as we can track their entropy. The gradient-thresholding method proposed in this chapter is a simple first example of a refinement to SGD that maintains a tractable entropy estimate while improving the quality of the intermediate distributions.

## 7.9 Conclusion

Most regularization methods have an interpretation as approximate inference in some Bayesian model. This chapter shows that early stopping and ensembling can also be interpreted this way, sampling from an implicit nonparametric distribution.

We introduced a variational lower bound on the marginal likelihood of these implicit distributions. We showed how to produce an unbiased estimate of this variational lower bound by approximately tracking the entropy change at each step of optimization. Our estimator is compatible with using data minibatches and scales linearly with the number of parameters, making it suitable for large-scale problems. This inexpensive calculation turns standard gradient descent into an inference algorithm.

In principle this bound could be used to choose model and training hyperparameters without a validation set, however in practice it doesn't beat the gold standard of cross-validation.

# 8

# Conclusion

This thesis has presented five contributions to machine learning: two algorithms for approximate Bayesian inference, Firefly Monte Carlo (Chapter 3) and early stopping as variational inference Chapter (7); a differentiable version of chemical fingerprints for learning data-driven molecular features (Chapter 5); a procedure for taking gradients with respect to hyperparameters by exactly reversing optimization dynamics (Chapter 6) and software for effortlessly producing gradients, Autograd (Chapter 4).

We can identify three recurring motifs. The first is the effectiveness of gradient-based optimization with flexible, highly parameterized models. This has been the central ingredient in the success of neural networks for supervised learning and saw it in action in our neural molecular fingerprints and in our gradient-based hyperparameter opimiza-

tion.

The second motif has been practical Bayesian inference. The Bayesian approach to modeling has many compelling merits, but it is often held back by a perception of computational difficulty. We need to make tools for approximate Bayesian inference that are easy to use and scale well to large data sets. Firefly and early stopping as inference were attempts in this direction.

The third motif has been the mind-expanding power of programming tools that present the right abstraction. The gradient operator is an excellent abstraction, and we use it freely and to great effect in symbolic mathematics. But having Autograd, a practical implementation of the gradient operator in an actual programming language, has been invaluable.

A promising research direction that combines these threads is to use differentiable, highly parameterized models as conditional variational distributions. We could build an abstraction for composing conditional probabilites that would allow us to specify both a generative process and a variational family that can approximately condition on observed variables. These ideas are in the air. See, for example, Kingma & Welling [51], Kucukelbir et al. [55] and very recent work from my own colleagues, Johnson et al. [47].

# References


[1] Abdel-Gawad, A. & Ratner, S. (2007). *Adaptive optimization of hyperparameters in L2-regularised logistic regression*. Technical report.

[2] Amador-Bedolla, C., R. Olivares-Amaya, R., Hachmann, J., & Aspuru-Guzik, A. (2013). Organic photovoltaics. In K. Rajan (Ed.), *Informatics for Materials Science and Engineering* (pp. 423–440).: Elsevier, Amsterdam.

[3] Anderson, H. L. (1986). Metropolis, Monte Carlo, and the MANIAC. *Los Alamos Science*.

[4] Andrieu, C. & Roberts, G. O. (2009). The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics*, (pp. 697–725).

[5] Angelino, E., Kohler, E., Waterland, A., Seltzer, M., & Adams, R. P. (2014). Accelerating MCMC via parallel predictive prefetching. In *Proceedings of the 30th Conference on Uncertainy in Artificial Intelligence*.

[6] Bai, Z., Fahey, G., & Golub, G. (1996). Some large-scale matrix computation problems. *Journal of Computational and Applied Mathematics*, 74(1), 71–89.

[7] Bardenet, R., Doucet, A., & Holmes, C. (2014). Towards scaling up MArkov chain Monte Carlo : an adaptive subsampling approach. In *Proceedings of the 31st International Conference on Machine Learning*.

[8] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., & Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

[9] Baydin, A. G., Pearlmutter, B. A., & Radul, A. A. (2015). Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767.

[10] Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, 12(8), 1889–1900.

[11] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2), 157–166.

[12] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., et al. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*.

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

[14] Bergstra, J., Yamins, D., & Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning* (pp. 115–123).

[15] Blei, D. M., Kucukelbir, A., & McAuliffe, J. D. (2016). Variational inference: A review for statisticians. *arXiv preprint arXiv:1601.00670*.

[16] Böhning, D. (1992). Multinomial logistic regression algorithm. *Annals of the Institute of Statistical Mathematics*, 44(1), 197–200.

[17] Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*.

[18] Chapelle, O., Vapnik, V., Bousquet, O., & Mukherjee, S. (2002). Choosing multiple parameters for support vector machines. *Machine learning*, 46(1-3), 131–159.

[19] Chen, D. & Hagan, M. T. (1999). Optimal use of regularization and cross-validation in neural network modeling. In *International Joint Conference on Neural Networks*, volume 2 (pp. 1275–1280).: IEEE.

[20] Courbariaux, M., Bengio, Y., & David, J.-P. (2014). Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*.

[21] Dahl, G. E., Jaitly, N., & Salakhutdinov, R. (2014). Multi-task neural networks for QSAR predictions. *arXiv preprint arXiv:1406.1231*.

[22] Delaney, J. S. (2004). ESOL: Estimating aqueous solubility directly from molecular structure. *Journal of Chemical Information and Computer Sciences*, 44(3), 1000–1005.

[23] Domke, J. (2012). Generic methods for optimization-based modeling. In *International Conference on Artificial Intelligence and Statistics* (pp. 318–326).

[24] Duane, S., Kennedy, A., Pendleton, B., & Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters*, 195, 216.

[25] Duvenaud, D., Maclaurin, D., & Adams, R. P. (2016). Early stopping as non-parametric variational inference. In *19th International Conference on Artificial Intelligence and Statistics*.

[26] Duvenaud, D., Maclaurin, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., & Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Neural Information Processing Systems*.

[27] Eigenmann, R. & Nossek, J. A. (1999). Gradient based adaptive regularization. In *Proceedings of the 1999 IEEE Signal Processing Society Workshop on Neural Networks* (pp. 87–94).: IEEE.

[28] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., & Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11, 625–660.

[29] Foo, C.-s., Do, C. B., & Ng, A. Y. (2008). Efficient multiple hyperparameter learning for log-linear models. In *Advances in neural information processing systems* (pp. 377–384).

[30] Gamo, F.-J., Sanz, L. M., Vidal, J., de Cozar, C., Alvarez, E., Lavandera, J.-L., Vanderwall, D. E., Green, D. V., Kumar, V., Hasan, S., et al. (2010). Thousands of chemical starting points for antimalarial lead identification. *Nature*, 465(7296), 305–310.

[31] Gelfand, A. & Smith, A. (1990). Sampling-based approaches to calculating marginal densities. *Journal of the Americal Statistical Association*, 85, 385.

[32] Glem, R. C., Bender, A., Arnby, C. H., Carlsson, L., Boyer, S., & Smith, J. (2006). Circular fingerprints: flexible molecular descriptors with applications from physical chemistry to ADME. *IDrugs: the investigational drugs journal*, 9(3), 199–204.

[33] Graves, A. (2011). Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems* (pp. 2348–2356).

[34] Graves, A., Wayne, G., & Danihelka, I. (2014). Neural Turing machines. *arXiv preprint arXiv:1410.5401*.

[35] Hachmann, J., Olivares-Amaya, R., Atahan-Evrenk, S., Amador-Bedolla, C., Sánchez-Carrera, R. S., Gold-Parker, A., Vogt, L., Brockway, A. M., & Aspuru-Guzik, A. (2011). The Harvard clean energy project: large-scale computational screening and design of organic photovoltaics on the world community grid. *The Journal of Physical Chemistry Letters*, 2(17), 2241–2251.

[36] Hachmann, J., Olivares-Amaya, R., Jinich, A., Appleton, A. L., Blood-Forsythe, M. A., Seress, L. R., Roman-Salgado, C., Trepte, K., Atahan-Evrenk, S., Er, S., Shrestha, S., Mondal, R., Sokolov, A., Bao, Z., & Aspuru-Guzik, A. (2014). Lead candidates for high-performance organic photovoltaics from high-throughput quantum chemistry - the Harvard clean energy project. *Energy Environ. Sci.*, 7, 698–704.

[37] Hardt, M., Recht, B., & Singer, Y. (2015). Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*.

[38] Hastings, W. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57, 97.

[39] Hensman, J. & Lawrence, N. D. (2014). Nested variational compression in deep Gaussian processes. *arXiv preprint arXiv:1412.1370*.

[40] Hernández-Lobato, J. M. & Adams, R. P. (2015). Probabilistic backpropagation for scalable learning of Bayesian neural networks. *Arxiv preprint arXiv:1502.05336*.

[41] Hershey, J. R., Roux, J. L., & Weninger, F. (2014). Deep unfolding: Model-based inspiration of novel deep architectures. *arXiv preprint arXiv:1409.2574*.

[42] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.

[43] Hut, P., Makino, J., & McMillan, S. (1995). Building a better leapfrog. *Astrophysical Journal, Part 2 - Letters*, 443, L93–L96.

[44] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *Proceedings of LION-5*, volume 6683 (pp. 507–523).: Springer.

[45] Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[46] Jaakkola, T. S. & Jordan, M. I. (1997). A variational approach to Bayesian logistic regression models and their extensions. In *Workshop on Artificial Intelligence and Statistics*.

[47] Johnson, M. J., Duvenaud, D., Wiltschko, A. B., Datta, S. R., & Adams, R. P. (2016). Structured vaes: Composing probabilistic graphical models and variational autoencoders. *arXiv preprint arXiv:1603.06277*.

[48] Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A convolutional neural network for modelling sentences. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics.*

[49] Kingma, D. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[50] Kingma, D. & Welling, M. (2014a). Efficient gradient-based inference through transformations between bayes nets and neural nets. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (pp. 1782–1790).

[51] Kingma, D. P. & Welling, M. (2014b). Auto-encoding variational Bayes. In *International Conference on Learning Representations.*

[52] Korattikara, A., Chen, Y., & Welling, M. (2014). Austerity in MCMC Land: Cutting the Metropolis-Hastings Budget. In *Proceedings of the 31st International Conference on Machine Learning.*

[53] Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images.* Technical report, Department of Computer Science, University of Toronto.

[54] Kucukelbir, A., Ranganath, R., Gelman, A., & Blei, D. (2014). Fully automatic variational inference of differentiable probability models. In *NIPS Workshop on Probabilistic Programming.*

[55] Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., & Blei, D. M. (2016). Automatic differentiation variational inference. *arXiv preprint arXiv:1603.00788.*

[56] Lake, B. M. (2014). *Towards more human-like concept learning in machines: Compositionality, causality, and learning-to-learn*. PhD thesis, Massachusetts Institute of Technology.

[57] Larsen, J., Svarer, C., Andersen, L. N., & Hansen, L. K. (1998). Adaptive regularization in neural network modeling. In *Neural Networks: Tricks of the Trade* (pp. 113–132). Springer.

[58] LeCun, Y. & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361.

[59] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1, 541–551.

[60] Lusci, A., Pollastri, G., & Baldi, P. (2013). Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. *Journal of chemical information and modeling*, 53(7), 1563–1575.

[61] MacKay, D. J. (1992). A practical Bayesian framework for backpropagation networks. *Neural computation*, 4(3), 448–472.

[62] MacKay, D. J. & Neal, R. M. (1994). Automatic relevance determination for neural networks. In *Technical Report*. Cambridge University.

[63] MacKay, D. J. C. (2003). *Information theory, inference, and learning algorithms*. Cambridge University press.

[64] Maclaurin, D. & Adams, R. P. (2014). Firefly Monte Carlo: Exact MCMC with subsets of data. In *30th Conference on Uncertainty in Artificial Intelligence*.

[65] Maclaurin, D., Duvenaud, D., & Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *32nd International Conference on Machine Learning*.

[66] Martens, J. & Sutskever, I. (2012). Training deep and recurrent networks with Hessian-free optimization. In *Neural Networks: Tricks of the Trade* (pp. 479–535). Springer.

[67] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., & Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21, 1087.

[68] Meyn, S. P. & Tweedie, R. L. (2012). *Markov chains and stochastic stability*. Springer Science & Business Media.

[69] Micheli, A. (2009). Neural network for graphs: A contextual constructive approach. *Neural Networks, IEEE Transactions on*, 20(3), 498–511.

[70] Morgan, H. (1965). The generation of a unique machine description for chemical structure. *Journal of Chemical Documentation*, 5(2), 107–113.

[71] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. Cambridge, MA: MIT Press.

[72] Neal, R. M. (2003). Slice sampling. *The Annals of Statistics*, 31(3), 705–767.

[73] Neal, R. M. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2.

[74] Olivares-Amaya, R., Amador-Bedolla, C., Hachmann, J., Atahan-Evrenk, S., Sanchez-Carrera, R. S., Vogt, L., & Aspuru-Guzik, A. (2011). Accelerated com-

putational discovery of high-performance materials for organic photovoltaics by means of cheminformatics. *Energy Environ. Sci.*, 4, 4849–4861.

[75] Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. *arXiv preprint arXiv:1211.5063*.

[76] Pathria, R. & Beale, P. (1996). *Statistical Mechanics*. Elsevier Science.

[77] Pearlmutter, B. (1996). *An investigation of the gradient descent process in neural networks*. PhD thesis, Carnegie Mellon University.

[78] Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural computation*, 6(1), 147–160.

[79] Pearlmutter, B. A. & Siskind, J. M. (2008). Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2), 7.

[80] Plummer, M., Best, N., Cowles, K., & Vines, K. (2006). CODA: Convergence diagnosis and output analysis for MCMC. *R News*, 6(1), 7–11.

[81] Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 3 edition.

[82] Ramsundar, B., Kearnes, S., Riley, P., Webster, D., Konerding, D., & Pande, V. (2015). Massively multitask networks for drug discovery. *arXiv:1502.02072*.

[83] Ranganath, R., Gerrish, S., & Blei, D. M. (2014). Black box variational inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*.

[84] Raskutti, G., Wainwright, M. J., & Yu, B. (2014). Early stopping and non-parametric regression: an optimal data-dependent stopping rule. *The Journal of Machine Learning Research*, 15(1), 335–366.

[85] Rasmussen, C. E. & Williams, C. K. (2006). *Gaussian Processes for Machine Learning*, volume 38. The MIT Press, Cambridge, MA, USA.

[86] RDKit, 2011. RDKit: Open-source cheminformatics. www.rdkit.org. [accessed 11-April-2013].

[87] Robert, C. & Casella, G. (2004). *Monte Carlo statistical methods*. Springer Science & Business Media.

[88] Roberts, G. & Stramer, O. (2003). Langevin diffusions and metropolis-hastings algorithms. *Methodology and Computing in Applied Probability*, 4, 337.

[89] Roberts, G. O., Gelman, A., & Gilks, W. R. (1997). Weak convergence and optimal scaling of random walk Metropolis algorithms. *Annals of Applied Probability*, 7, 110–120.

[90] Roberts, G. O. & Rosenthal, J. S. (1998). Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60(1), 255–268.

[91] Roberts, G. O. & Tweedie, R. L. (1996). Exponential convergence of Langevin distributions and their discrete approximations. *Bernoulli*, (pp. 341–363).

[92] Rogers, D. & Hahn, M. (2010). Extended-connectivity fingerprints. *Journal of Chemical Information and Modeling*, 50(5), 742–754.

[93] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.

[94] Salimans, T., Kingma, D. P., & Welling, M. (2014). Markov chain Monte Carlo and variational inference: Bridging the gap. *arXiv preprint arXiv:1410.6460*.

[95] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *Neural Networks, IEEE Transactions on*, 20(1), 61–80.

[96] Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems 25*.

[97] Socher, R., Huang, E. H., Pennin, J., Manning, C. D., & Ng, A. Y. (2011a). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems* (pp. 801–809).

[98] Socher, R., Pennington, J., Huang, E. H., Ng, A. Y., & Manning, C. D. (2011b). Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (pp. 151–161).: Association for Computational Linguistics.

[99] Solak, E., Murray Smith, R., Leithead, W., Leith, D., & Rasmussen, C. E. (2003). Derivative observations in Gaussian process models of dynamic systems. *Advances in Neural Information Processing Systems*, (pp. 1057–1064).

[100] Stein, C. M. (1981). Estimation of the mean of a multivariate normal distribution. *The Annals of Statistics*, 9(6), 1135–1151.

[101] Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (pp. 1139–1147).

[102] Sutskever, I., Vinyals, O., & Le, Q. V. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27* (pp. 3104–3112). Curran Associates, Inc.

[103] Tai, K. S., Socher, R., & Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.

[104] Tieleman, T. & Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning.

[105] Toulis, P., Tran, D., & Airoldi, E. M. (2015). Stability and optimality in stochastic gradient descent. *arXiv preprint arXiv:1505.02417*.

[106] Tox21 Challenge (2014). National center for advancing translational sciences. http://tripod.nih.gov/tox21/challenge. [Online; accessed 2-June-2015].

[107] Unterthiner, T., Mayr, A., Klambauer, G., & Hochreiter, S. (2015). Toxicity prediction using deep learning. *arXiv preprint arXiv:1503.01445*.

[108] Unterthiner, T., Mayr, A., ünter Klambauer, G., Steijaert, M., Wenger, J., Ceulemans, H., & Hochreiter, S. (2014). Deep learning as an opportunity in virtual screening. In *Advances in Neural Information Processing Systems*.

[109] Wainwright, M. J. & Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2), 1–305.

[110] Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., & Fergus, R. (2013). Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*.

[111] Wang, C. & Blei, D. M. (2013). Variational inference in nonconjugate models. *The Journal of Machine Learning Research*, 14(1), 1005–1031.

[112] Weininger, D. (1988). SMILES, a chemical language and information system. *Journal of chemical information and computer sciences*, 28(1), 31–36.

[113] Welling, M. & Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 681–688).

[114] Widrow, B. & Lehr, M. A. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9), 1415–1442.