Vector Step-size Adaptation for Continual, Online Prediction

by

Andrew Jacobsen

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

In this thesis, we investigate different vector step-size adaptation approaches for continual, online prediction problems. Vanilla stochastic gradient descent can be considerably improved by scaling the update with a vector of appropriately chosen step-sizes. Many methods, including AdaGrad, RMSProp, and AMSGrad, keep statistics about the learning process to approximate a second-order update — a vector approximation of the inverse Hessian. Another family of approaches uses meta-gradient descent to adapt the step-size parameters to minimize prediction error. These meta-descent strategies are promising for non-stationary problems, but have not been as extensively explored as quasi-second order methods. We derive a general, incremental meta-descent algorithm, called AdaGain, designed to be applicable to a broader range of algorithms, including those with semi-gradient updates or even those with accelerations, such as RMSProp. We introduce an instance of AdaGain which combines meta-descent with RMSProp — a method we call RMSGain — which is particularly robust across several prediction problems and is competitive with the state-of-the-art method on a large-scale, time-series prediction problem on real data from a mobile robot.

Preface

This thesis is an extension of a paper we published at the AAAI Conference on Artificial Intelligence (Jacobsen *et al.* 2019a). Chapters 2, 3, and 8 are original works. Chapters 1, 4, and 5 were composed and edited with help from my co-authors in the published work, and then revised and extended by me for this thesis. The experiments in Chapter 6 and Chapter 7 were conducted by me in the original work, and were additionally reproduced and refined for this thesis.

I'm pretty sure you can put whatever the hell you want as your thesis quote.

– Cameron Linke, 2019.

Acknowledgements

I'd first like to thank my two supervisors, Adam White and Martha White. I have received endless support and guidance from them in writing this thesis, as well as during the many other projects we've worked on together over the past year. I'm proud of the amount of work we've accomplished in this short time, and it wouldn't have happened without the opportunities and direction they've provided me. Both have had a profound influence on me and are, more than anyone else, the people I aspire to be like as a research scientist and as a person.

I also owe a great debt of graditude to Matthew Schlegel. I had the fortune of working closely with Matt during my first project with the RLAI lab, and that project eventually branched out into the work that makes up this thesis. Matt has also been a source of support, encouragement, and guidance throughout the past year, and often feels like a third supervisor. In the years to come, I hope that I can inspire new students as much as Matt inspired me.

Perhaps most importantly, thanks to all of the above for all the laughs and beers on Fridays.

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we consider continual, online prediction problems. Consider a learning system whose objective is to learn a large collection of predictions about an agent's future interactions with the world. The predictions specify the value of some signal many steps in the future, given that the agent follows some specific course of action. There are many examples of such prediction learning systems including Predictive State Representations (Littman *et al.* 2001), Observable Operator Models (Jaeger 2000), Temporal-difference Networks (R. S. Sutton and Tanner 2004), and General Value Functions (R. S. Sutton *et al.* 2011). In our setting, the agent continually interacts with the world, making new predictions about the future, and revising its predictions as data is observed. Occasionally, partially due to changes in the world and partially due to changes in the agent's own behavior, the targets may change and the agent must refine its predictions.

Stochastic gradient descent (SGD) is a natural choice for our setting because it is computationally efficient and allows new data to be continually incorporated. The performance of SGD is dependent on the step-size parameter (scalar, vector, or matrix), which scales the gradient to mitigate sample variance and improve data efficiency. Most modern large-scale learning systems make use of optimization algorithms that attempt to approximate stochastic second-order gradient descent to adjust both the direction and magnitude of the descent direction, with early work indicating the benefits of such quasi-second order methods if used carefully in the stochastic case (Bordes *et al.*

2009; N. Schraudolph *et al.* 2007).

Many of these algorithms attempt to approximate the diagonal of the inverse Hessian, which describes the curvature of the loss function, and so maintain a vector of step-sizes—one for each parameter. Starting from AdaGrad (Duchi *et al.* 2011; McMahan and Streeter 2010), several diagonal approximations have been proposed, including RMSProp (Tieleman and Hinton 2012), AdaDelta (Zeiler 2012), vSGD (Schaul *et al.* 2013), Adam (Kingma and Ba 2015) and AMSGrad (Reddi *et al.* 2018). Stochastic quasi-second order updates have been derived specifically for temporal difference learning, with some empirical success (Meyer *et al.* 2014), particularly in terms of parameter sensitivity (Pan *et al.* 2017a; Pan *et al.* 2017b). On the other hand, second-order methods generally assume the loss is fixed (or drawn from a fixed distribution at each time step), and so non-stationary dynamics or drifting targets could be problematic.

A related family of optimization algorithms, called *meta-descent* algorithms, were developed for continual, online prediction problems, such as online supervised learning (Jacobs 1988; A. R. Mahmood *et al.* 2012; A. Mahmood 2010; N. N. Schraudolph 1999; R. S. Sutton 1992a) and reinforcement learning (W. C. Dabney 2014; W. Dabney and Barto 2012; Kearney *et al.* 2018). These algorithms perform meta-gradient descent adapting a vector of step-size parameters to minimize the error of the base learner, instead of approximating the Hessian.

Meta-descent applied to the step-size was first introduced for online least-mean squares methods (Almeida *et al.* 1998; Jacobs 1988; A. R. Mahmood *et al.* 2012; R. S. Sutton 1992a; R. S. Sutton 1992b), including the linear complexity method IDBD (R. S. Sutton 1992a). IDBD was later extended to more general losses (N. N. Schraudolph 1999) and to support (semi-gradient) temporal difference methods (W. C. Dabney 2014; W. Dabney and Barto 2012; Kearney *et al.* 2018). These methods are well-suited to non-stationary problems and have been shown to ignore irrelevant and noisy features (Kearney *et al.* 2018; Kearney *et al.* 2019). The main limitation of several of these meta-descent algorithms, however, is that the derivations are heuristic, making it

difficult to extend to new settings beyond linear temporal difference learning. More general approaches, like Stochastic Meta-Descent (SMD) (N. N. Schraudolph 1999), require the update to be a stochastic gradient descent update and have some issues because they are biased toward smaller step-sizes (Wu *et al.* 2018). It remains an open challenge to make these meta-descent strategies as broadly and easily applicable as the AdaGrad variants.

In this thesis we introduce a new meta-descent algorithm, called AdaGain, that attempts to optimize the stability of the base learner, rather than convergence to a fixed point. AdaGain is built on a generic derivation scheme that allows it to be easily combined with a variety of base-learners including SGD, (semi-gradient) temporal-difference learning and even accelerated SGD updates, like AMSGrad.

Our goal is to investigate the utility of both meta-descent methods and the more widely used quasi-second order optimizers in online, continual prediction problems. We provide an extensive empirical comparison on (1) a canonical optimization problem that is difficult to optimize, requiring the ability to raise the step-size in some regions of the surface and lower it in others to achieve good performance, (2) a finite Markov Decision Process with linear features that cause conventional temporal difference learning to diverge, (3) an online, supervised tracking problem where the optimal step-sizes can be computed, and (4) a high-dimensional time-series prediction problem using data generated from a real mobile robot. In problems with non-stationary dynamics, the meta-descent methods can exhibit an advantage over the quasi-second order methods. On the difficult optimization problems, however, meta-descent methods can fail, which, retrospectively, is unsurprising given the meta-optimization problem for stepsizes is similarly difficult to optimize.

We show that AdaGain can obtain the advantages of both families — performing well on both optimization problems with flat regions as well as non-stationary problems — by selecting an appropriate base learner, such as RMSProp.

The proposed contributions of this thesis are as follows:

1. We introduce a new meta-descent objective which enables meta-descent to be applied to a broader class of base learners than previous methods. We derive a new meta-descent algorithm, AdaGain, which is designed to encourage the stability of learning updates. We provide the derivations for both the full quadratic complexity algorithm and the linear complexity approximation. In addition, we derive the update equations for several base learning algorithms, such as TD(λ), Least Mean Squares updates, and provide a general-purpose finite-difference approximation which can be used without second-order information.

2. We show that the performance of meta-descent methods can be significantly improved by combining them with quasi second-order approaches. We provide an instance of AdaGain which combines meta-descent with RMSProp, called *RMSGain*, and find that it outperforms either of the methods used on their own.

3. We provide an empirical comparison of quasi second-order and meta-descent strategies. Our experiments are designed to test the stability and efficiency of learning in both stationary and non-stationary problems. We find that RMSGain performs significantly better than the existing meta-descent strategies, at least as well as the state-of-the art quasi second-order strategies such as Adam and AMSGrad, and can exhibit lower sensitivity to the its hyperparameters than the competing methods.

This document contains eight chapters. In Chapter 2 we review the fundamental concepts required to understand this thesis and define our notation. In Chapter 3, we motivate adaptive learning methods which attempt to skew the direction and magnitude of the gradient in some meaningful way. Chapter 4 formulates our problem setting and reviews the popular instances of two families of algorithms: quasi second-order methods and meta-descent methods. In Chapter 5 we introduce our meta-descent algorithm, AdaGain. We then continue on to our experimental results in stationary settings and non-stationary settings in Chapter 6 and Chapter 7 respectively. The thesis is concluded with

a summary of the contributions of this work and a discussion of future work in Chapter 8.

Chapter 2

Background and Notation

In this chapter, we review the fundamental concepts required to understand this thesis. We begin by setting up some notation which will be used frequently in this thesis, specifying broadcasting operations and element-wise operations on vectors. We then review the basics of unconstrained numerical optimization. Finally, we review relevant concepts and algorithms from the reinforcement learning literature. The main purpose of this chapter is to define the notation that we will use throughout this document. Our precise problem formulation begins in Chapter 4.

2.1 Broadcasting Operations

Many of the algorithms in this thesis make heavy use of element-wise operations. For ease of notation, we frequently make use of *broadcasting* operations. A scalar operator applied to a vector denotes applying the operator to each of the elements of the vector. For example, for $\mathbf{x} \in \mathbb{R}^d$, $\sqrt{\mathbf{x}}$ denotes the vector $\mathbf{z} \in \mathbb{R}^d$ with $z_i = \sqrt{x_i}$. The same is true of binary scalar operators; for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, we use $\frac{\mathbf{x}}{\mathbf{y}}$ to denote the vector $\mathbf{z} \in \mathbb{R}^d$ with $z_i = \frac{x_i}{y_i}$. We use $\mathbf{x} \circ \mathbf{y}$ to denote element-wise multiplication, to avoid confusion with matrix-vector products. Finally, scalars broadcast to each element of a vector when added, so that for $\epsilon \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{z} \stackrel{\text{def}}{=} \mathbf{x} + \epsilon$ is the vector with $z_i = x_i + \epsilon$. This notation not only makes the update equations look less cluttered, but is also closer to the actual implementation that would be used in software, as most numerical computing libraries have these operations implemented.

2.2 Unconstrained Optimization

Suppose we want to find the minimum of a smooth continuous function $\ell : \mathbb{R}^d \to \mathbb{R}$. That is, we want to find an $\mathbf{x}^* \in \mathbb{R}^d$ such that $\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathbb{R}^d} \ell(\mathbf{x})$. There are many possible directions that this problem could be approached from, but in this thesis we will focus on the method of steepest descent, or *gradient descent*.

Let $\frac{\partial \ell}{\partial x_i}$ denote the partial derivative of $\ell(\mathbf{x})$ with respect to the *ith* component of input vector $\mathbf{x}$, and let $\nabla_{\mathbf{x}}\ell(\mathbf{x}) \stackrel{\text{def}}{=} [\frac{\partial \ell}{\partial x_1}, \ldots, \frac{\partial \ell}{\partial x_d}]^\top$ denote the *gradient* of $\ell(\mathbf{x})$ — that is, a vector containing the partial derivatives of $\ell(\mathbf{x})$ with respect to each of the components of $\mathbf{x}$. We will use ∇ in favor of $\nabla_{\mathbf{x}}$ in cases where it is obvious from context what variable the gradient is being taken with respect to. For a twice differentiable loss function $\ell(\mathbf{x})$, we denote the matrix of second derivatives (the *Hessian* matrix) $\nabla^2\ell(\mathbf{x})$, where $[\nabla^2\ell(\mathbf{x})]_{i,j} = \frac{\partial^2 \ell(\mathbf{x})}{\partial x_j \partial x_i}$.

Gradient descent approaches begin by selecting an initial input vector $\mathbf{x}_0 \in \mathbb{R}^d$. The initial input $\mathbf{x}_0$ may be chosen using some prior knowledge about the function ℓ, but is often chosen randomly. Minimizing ℓ then occurs by iteratively updating $\mathbf{x}$ in the opposite direction of the negative gradient; on each iteration $t \in \mathbb{N}$, $\mathbf{x}$ is updated as

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \nabla \ell(\mathbf{x}_t)$$

where $\alpha_t > 0 \in \mathbb{R}$ is a scalar *step-size* parameter, controlling how far the input $\mathbf{x}$ is perturbed in the direction of the gradient at time t. It is common to just select a constant $\alpha_t = \alpha$ for all time-steps t. Many heuristic approaches exist for selecting α_t, such as decaying an initial $\alpha_0 \in \mathbb{R}$ over time according to some fixed schedule or via line-search methods; for a detailed discussion of these approaches, the reader is directed to Wright and Nocedal (1999) for a clear and engaging development of these approaches.

Oftentimes we will be interested in minimizing loss functions which are *parameterized* by a vector of parameters $\mathbf{w} \in \mathbb{R}^d$, $\ell(\mathbf{x}; \mathbf{w})$. For example, the squared error of a linear predictor could be $\ell(y, \mathbf{x}; \mathbf{w}) = (y - \mathbf{x}^\top \mathbf{w})^2$, where $\mathbf{x}^\top \mathbf{w}$ is a linear approximation of y. In this setting, the procedure is much

the same, except now we iteratively update $\mathbf{w}$ in the direction which decreases $\ell(y, \mathbf{x}; \mathbf{w})$, using the gradient $\nabla_{\mathbf{w}} \ell(y, \mathbf{x}; \mathbf{w})$ and Hessian $\nabla^2_{\mathbf{w}} \ell(y, \mathbf{x}; \mathbf{w})$.

2.3 Stochastic Gradient Descent

In many applications, the loss is defined in terms of an average performance error over a dataset. For example, consider a dataset $\mathcal{D} = \{(\mathbf{x}_t, y_t) : \mathbf{x}_t \in \mathbb{R}^d, y_t \in \mathbb{R}, t = 1, \ldots, N\}$, and suppose we want to fit a linear function $\hat{y}(\mathbf{x}_t) \stackrel{\text{def}}{=} \mathbf{x}_t^\top \mathbf{w} \approx y_t$ to the data. A reasonable objective would be to find the $\mathbf{w} \in \mathbb{R}^d$ which yields the best approximation on average

$$\ell(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{t=1}^{N} (y_t - x_t^\top \mathbf{w})^2,$$

or more generally,

$$\ell(\mathbf{w}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{t=1}^{N} \ell(\mathbf{x}_t, y_t; \mathbf{w})$$

where $\ell(\mathbf{x}_t, y_t; \mathbf{w})$ is the loss on sample $(\mathbf{x}_t, y_t)$.

In practice, however, N can be very large and thus the gradient of the above loss can be prohibitively expensive to compute. Instead, it is common to sample a quantity which is equal to the gradient of the true loss function *in expectation*. It can be shown that using the gradient of the loss on a single sample $(\mathbf{x}_t, y_t)$ is an unbiased estimator of the true gradient

$$\nabla \ell(\mathbf{w}) = \mathbb{E}[\nabla \ell(\mathbf{x}_t, y_t; \mathbf{w})],$$

where the expectation is taken over the dataset $\mathcal{D}$. This expectation can be approximated using a single sample, $\nabla \ell(\mathbf{x}_i, y_i; \mathbf{w})$. Updating $\mathbf{w}$ in the direction of $\nabla \ell(\mathbf{x}_t, y_t; \mathbf{w})$ at each time step can be seen as a *stochastic approximation* to the gradient descent algorithm, and thus is referred to as *Stochastic Gradient Descent* (SGD).

2.4 Linear Least Mean Squares

One of the relevant problem settings in this thesis is the (online) linear Least Mean Squares (LMS) setting. In this setting, a sample $(\mathbf{x}_t, y_t)$ is observed at

each discrete time step t, where $\mathbf{x}_t \in \mathbb{R}^d$ is a given input vector and $y_t \in \mathbb{R}$ is a target signal to be predicted using a linear combination of the inputs, $\hat{y}_t \stackrel{\text{def}}{=} \mathbf{x}_t^\top \mathbf{w}_t$ for parameter weights $\mathbf{w}_t \in \mathbb{R}^d$. The goal of the learner is to minimize the difference between y_t and $\hat{y}_t$ in expectation. The minimization can be formulated as SGD on the loss function $\ell(\mathbf{w}_t) = \frac{1}{2}(y_t - \mathbf{x}_t^\top \mathbf{w}_t)^2$, leading to the update equation

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha_t \nabla \ell(\mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha_t (y_t - \mathbf{x}_t^\top \mathbf{w}_t)\mathbf{x}_t\end{aligned}$$

where $\alpha_t \in \mathbb{R}$ is a scalar step-size parameter, often chosen to be constant.

2.5 Reinforcement Learning

Reinforcement Learning is a framework in which a decision maker (or the *agent*) learns via repeated trial-and-error interaction with an *environment* over a sequence of discrete time steps $t \in \mathbb{N}$. The problem is often formalized as a *Markov Decision Process* (MDP) $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions, $\mathcal{R} \subseteq \mathbb{R}$ the set of rewards. At each time step $t \in \mathbb{N}$, the agent observes the current state S_t and chooses an action A_t according to a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ specifying the distribution over actions in each state, $\pi(a|s) = Pr\{A_t = a|S_t = s\}$. As a result of the agent's action, the environment transitions to the next state $S_{t+1} \in \mathcal{S}$ and the agent receives a *reward* $R_{t+1} \in \mathcal{R}$. The dynamics of the MDP are specified by the function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, specifying the probability of transitioning to a state s' and receiving reward r when taking action a in state s, $p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|, S_t = s, A_t = a\}$.

In this setting, we often want to estimate the quality of a given policy π. This is often done by estimating the *return*

$$G_t \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $\gamma \in [0, 1)$ is a *discount factor* controlling the degree to which immediate rewards are valued compared to later rewards. In practice, we do not know in

advance what the sequence of rewards will be, so the return must be estimated by a function referred to as the *value function*

$$v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}[G_t | S_t = s, A_{t:\infty} \sim \pi].$$

The value function satisfies a recursive relationship specified by the *Bellman equation* $v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a)[r + \gamma v_\pi(s')]$.

2.5.1 Value Function Approximation

In practice, it can be infeasible to approximate the value function. It is often the case that the agent does not have access to the actual states; rather, the agent receives on each time step an incomplete summary of the state in the form of an *feature vector* $\mathbf{x}_t \stackrel{\text{def}}{=} \mathbf{x}(S_t) \in \mathbb{R}^d$. Even if the true states of the environment *are* observed, in most practical applications there are simply too many distinct states to be able to learn $v_\pi(s)$ for every single $s \in \mathcal{S}$, so a feature vector $\mathbf{x}_t \in \mathbb{R}^d$ with $d << |S|$ must be used.

In situations such as these, we attempt to estimate the value function using function approximation. Let $\hat{v}_\pi(s; \mathbf{w})$ be a function parameterized by $\mathbf{w} \in \mathbb{R}^d$ which takes a state $s \in \mathcal{S}$ and produces an estimate of $v_\pi(s)$. In this thesis, our main focus is on algorithms with computational complexity linear in the number of inputs, so we limit our discussion to *linear* value function approximations, $\hat{v}(s; \mathbf{w}) \stackrel{\text{def}}{=} \mathbf{x}(s)^\top \mathbf{w}$, where $\mathbf{w} \in \mathbb{R}^d$.

Fortunately, we have computationally efficient learning algorithms for estimating value functions using linear approximation, one of the most popular being the TD(λ) algorithm. TD(λ) uses an exponentially-decaying memory of previous observations $\mathbf{z}_t$ — the *eligibility trace* — to update the parameter weights on each step. It uses a parameter vector $\lambda \in [0, 1]$ to control the rate at which the trace decays. On each step, the TD(λ) algorithm updates the parameter vector $\mathbf{w}$ using the following update scheme:

$$\begin{aligned}
\delta_t &\stackrel{\text{def}}{=} R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}; \mathbf{w}_t) - \hat{v}_\pi(S_t; \mathbf{w}_t) \\
\mathbf{z}_t &= \lambda \gamma \mathbf{z}_{t-1} + \mathbf{x}_t \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t
\end{aligned}$$

where $\mathbf{z}_{-1} = \mathbf{0} \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$ is a constant scalar step-size parameter, $\gamma \in [0, 1)$ the discount rate, and $\lambda \in [0, 1]$ the trace decay parameter.

Chapter 3

Background on Adaptive Learning Methods

In this chapter, we introduce the idea of *adaptive* learning methods — methods which adapt to the local curvature of the loss function in order to make more effective learning updates. While vanilla gradient descent is often used in practice, it can be slow to converge in many settings. To give some intuitions why, let us consider the parabolic function depicted in Figure 3.1 (left), defined by $z(x, y) = x^2 + 10y^2 - 2xy$. The function is convex and has a global minimum

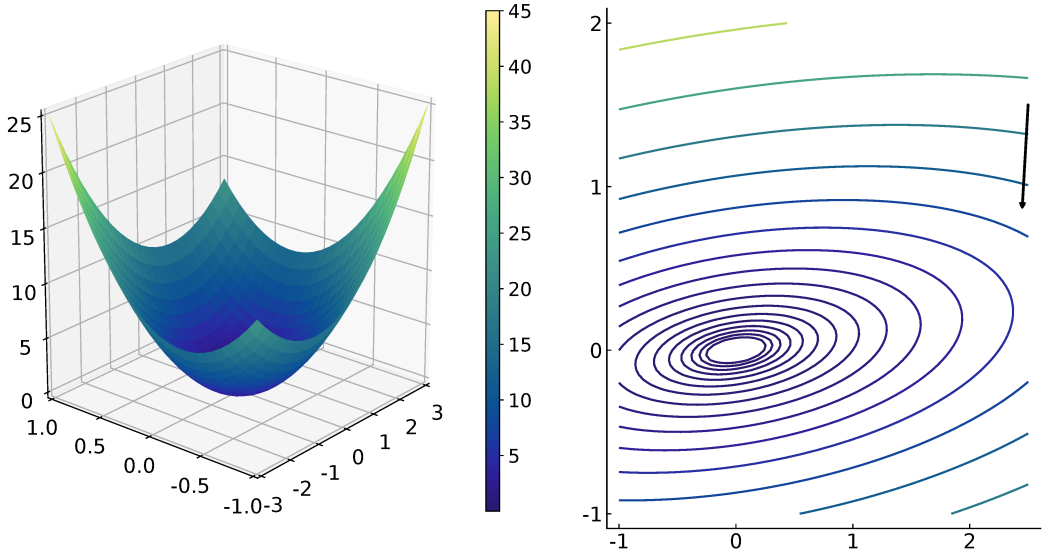

Figure 3.1: (left) Surface plot of the function $f(x, y) = x^2 + 10y^2 - 2xy$. (right) A contour plot of the function $f(x, y)$. Each of the ellipses denotes a set of points where $f(x, y)$ has the same value. A black arrow shows the gradient descent update using step-size $\alpha = 0.025$ from the point $(2.5, 1.5)$.

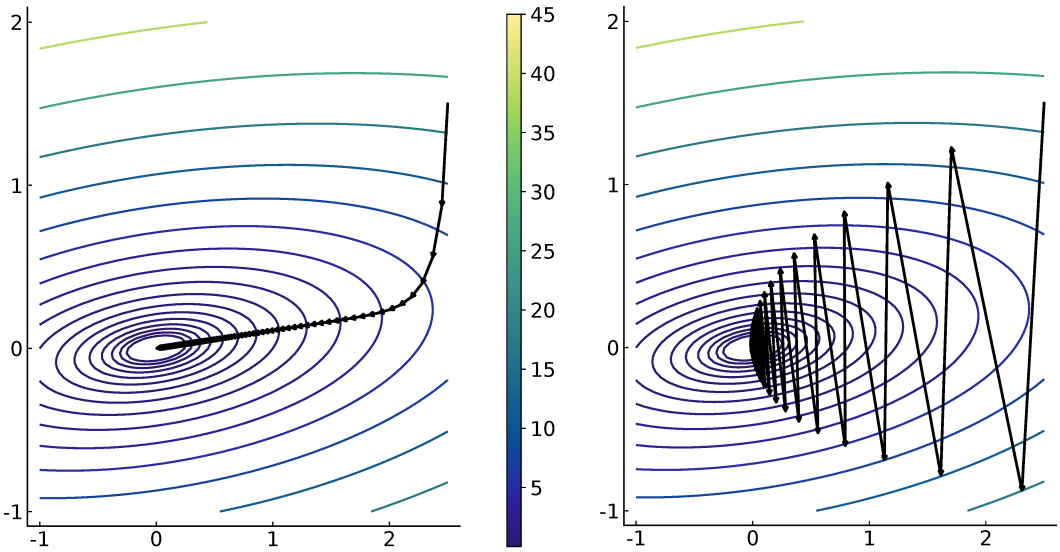

Figure 3.2: Two different steepest descent trajectories. Figure (a) shows the trajectory when using $\alpha = 0.025$, and (b) shows the trajectory using $\alpha = 0.095$.

at $(0, 0)$.

Suppose that at some iteration $t \in \mathbb{N}$ of the optimization process, we end up at the point $(x_t, y_t) = (2.5, 1.5)$, and we take a step in the direction of steepest descent using a step-size of $\alpha = 0.025$, as depicted by the black arrow in Figure 3.1 (right). Clearly, this step is in the correct *general* direction, but is not a step *directly* towards the minima. Examining the gradient $\nabla z(x, y) = [2x - 2y, 20y - 2x]^\top$, we notice that the curvature of this surface is skewed; the function is much steeper along the y-axis than it is along the x-axis. Because of this, steps in the direction of steepest descent will tend to be larger along the y-axis, rather than directly towards the minimum.

Figure 3.2 shows the gradient descent trajectories for two different settings of the step-size. On the left is the trajectory using $\alpha = 0.025$. As noted above, during the beginning of the optimization process, the direction of the gradient does not match the direction of the minimizer. Eventually, the steps do begin to point towards the minima, but now we can see a different issue: the size of the optimization steps have become very small, and it takes a large number of steps to converge to the solution! Along the y-axis, a step-size of $\alpha = 0.025$ is suitable because the gradients are larger in magnitude, but along the x-axis

the gradients are much smaller, necessitating a larger step-size to facilitate rapid progress. On the other hand, Figure 3.2 (right) shows the trajectory when using a larger step-size, $\alpha = 0.095$. Though the larger step-size would be more suitable along the x-axis, it is not well-suited to the y-axis, resulting in the jittery back-and-forth trajectory depicted.

The issues above demonstrate that performing vanilla gradient descent with a constant step-size can lead to updates which are not well-suited to all parts of the optimization surface — the flatter regions of the surface have small-magnitude gradients and require larger step-sizes to make significant progress, whereas the steeper regions have high-magnitude gradients and require smaller step-sizes to prevent large, unstable steps from being taken.

Based on this example, one might intuit that the efficiency of vanilla gradient descent could be improved by a) skewing the descent direction, so as to step more directly towards minima, and b) scale the step-size α in a way which is suitable for the current location on the optimization surface. There is a long history of methods which do exactly this, which we refer to as adaptive methods. In particular, in this thesis, we refer to *adaptive methods* as methods using a *pre-conditioned* gradient descent update:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mathbf{H}_t \nabla \ell(\mathbf{x}_t),$$

where $\mathbf{H}_t \in \mathbb{R}^{d \times d}$ is a *pre-conditioning* matrix, or *preconditioner*, which skews the gradient in some desireable way. When $\mathbf{H}_t$ is a diagonal matrix, the above update can equivalently be written

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mathbf{h}_t \circ \nabla \ell(\mathbf{x}_t),$$

where $\mathbf{h}_t \in \mathbb{R}^d$ is the diagonal of $\mathbf{H}_t$: $[\mathbf{h}_t]_i = [\mathbf{H}_t]_{i,i}$. In the following two sections, we will discuss two classes of adaptive methods which play a significant role in this thesis: second-order and quasi second-order methods.

3.1 Second-order Methods

In the previous section, we saw that certain optimization surfaces can cause gradient descent to take steps which are suboptimal in both direction and

magnitude, leading to less efficient optimization. Note that both of these effects can be explained by the surface having differences in *curvature* along different axes. If a region of the surface is steeper along the y-axis than the x-axis, for example, the gradient will be skewed in the direction of the y-axis. If the surface has both flat and steep regions, a constant step-size will tend to be ill-suited to at least one of those regions. If we could skew the gradient to account for the curvature of the surface, we might be able to alleviate these issues to some degree.

Second-order methods attempt to improve the iterative process by accounting for the local curvature of $\ell(\mathbf{x})$. The idea is to model $\ell(\mathbf{x})$ locally around $\mathbf{x}$ as a quadratic function using a second-order taylor-series expansion

$$\ell(\mathbf{x}) \approx \ell(\mathbf{x}_0) + \nabla\ell(\mathbf{x}_0)^\top(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top\nabla^2\ell(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0), \tag{3.1}$$

where $\mathbf{x}_0$ is a reference point (ideally close to $\mathbf{x}$) and $\nabla^2\ell(\mathbf{x}_0)$ is the *Hessian* of ℓ w.r.t. $\mathbf{x}$ — the matrix of second derivatives, $[\nabla^2\ell(\mathbf{x})]_{i,j} = \frac{\partial^2\ell(\mathbf{x})}{\partial x_i \partial x_j}$ — evaluated at $\mathbf{x}_0$. We denote the quadratic approximation of $\ell(\mathbf{x})$ as $\hat{\ell}(\mathbf{x})$. Taking the gradient of this approximation w.r.t. $\mathbf{x}$, we get:

$$\nabla\hat{\ell}(\mathbf{x}) = \nabla\ell(\mathbf{x}_0) + \nabla^2\ell(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0). \tag{3.2}$$

We can find the $\mathbf{x}$ which minimizes $\hat{\ell}(\mathbf{x})$ by setting this gradient to $\mathbf{0}$ and solving for the $\mathbf{x}$, leading to the solution

$$\mathbf{x} = \mathbf{x}_0 - \nabla^2\ell(\mathbf{x}_0)^{-1}\nabla\ell(\mathbf{x}_0).$$

Thus, iterating this update leads to the popular second-order optimization strategy called *Newton's method*:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \nabla^2\ell(\mathbf{x}_t)^{-1}\nabla\ell(\mathbf{x}_t). \tag{3.3}$$

We refer to an update of this form, using the full Hessian as a preconditioner, as a *Newton update*. Newton's method can be shown to converge quadratically under mild assumptions on the loss function (Wright and Nocedal 1999). When the loss function *is* a quadratic function, Equation 3.1 is no longer an approximation, and the solution in Equation 3.1 is exact — the minima is found in a single step.

3.2 Quasi Second-order Methods

In many applications, accounting for local curvature using a Newton update (Equation 3.3) would be desirable due to the faster rates of convergence, but is often computationally infeasible. Many real-world problems involve high-dimensional inputs, so computing and inverting the $d \times d$ Hessian matrix can be too expensive — especially in systems operating in real time, such as robotics. For this reason, it is often desirable to *approximate* the inverse Hessian. Instead of preconditioning with $(\nabla^2 \ell(\mathbf{x}_t))^{-1}$, *quasi second-order* methods replace the Hessian with an approximation $\mathbf{B}_t \approx \nabla^2 \ell(\mathbf{x}_t)^{-1}$

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mathbf{B}_t \nabla \ell(\mathbf{x}_t).$$

Many of the successful quasi second-order methods — such as BFGS and SR1 (Wright and Nocedal 1999) — attempt to approximate the inverse Hessian of the loss function *incrementally*, avoiding the explicit matrix inversion on each iteration. However, in practice, the quadratic computation required to maintain the approximation of the inverse Hessian can still be too expensive. For this reason, many of the popular approaches in recent years instead use a diagonal approximation, $\mathbf{b}_t \approx \text{diag}(\nabla^2 \ell(\mathbf{x}_t)^{-1})$, allowing the estimate to be updated in linear time (Kingma and Ba 2015; Reddi *et al.* 2018; Tieleman and Hinton 2012; Zeiler 2012).

3.3 Conclusion

In this chapter, we reviewed some of the basic ideas behind adaptive learning methods. The direction of the negative gradient often does not actually point towards the minimum of the function being minimized, which can lead to inefficiencies in the optimization process. Furthermore, it is often the case that smaller step-sizes are required in some regions of the input space, and larger step-sizes in others. Adaptive methods seek to skew the direction and magnitude of the negative gradient in some meaningful way using a preconditioning matrix $\mathbf{H}_t$. One such method is Newton's method, which selects $\mathbf{H}_t$ to be the inverse Hessian of the loss function. Many methods, called quasi second-order

methods, seek to approximate the inverse Hessian incrementally. In the next chapter, we discuss a family of quasi second-order methods which attempts to make this approximation in linear time, using only first-order information.

Chapter 4

Adaptive Learning Methods for Online Prediction

In this thesis we consider *continual, online prediction* problems modeled as dynamical systems. On each discrete time step t, the agent observes the internal state of the system through an imperfect summary vector $\mathbf{o}_t \in \mathcal{O} \in \mathbb{R}^d$ for some $d \in \mathbb{N}$, such as the sensor readings of a mobile robot. On each step, the agent makes a prediction about a target signal $T_t \in \mathbb{R}$. In the simplest case, the target of the prediction is a component i of the observation vector on the next step $T_t = \mathbf{o}_{t+1,i}$—the classic one-step prediction. In the more general case, the target is constructed by mapping the entire future of the observation time series to a scalar, such as the discounted sum formulation used in reinforcement learning: $T_t = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k \mathbf{o}_{t+k+1,i}]$, where $\gamma \in [0, 1)$ discounts the contribution of future observations to the infinite sum. The prediction $P_t \in \mathbb{R}$ is generated by a parametrized function, with modifiable parameter vector $\mathbf{w}_t \in \mathbb{R}^d$.

In continual online prediction problems, the agent's objective is to minimize the error between the prediction P_t given by $\mathbf{w}_t$ and the target T_t before it is observed, over all time steps. The prediction task is performed indefinitely, with new data being observed on each time step; the agent must learn the predictions as the data comes in, updating its predictions (via $\mathbf{w}_t$) with each new sample $\mathbf{o}_t$. This contrasts offline problem settings, in which data-collection and learning happen in separate phases.

Continual online prediction problems are typically solved using stochastic updates to adapt the parameter vector $\mathbf{w}_t$ after each time step t to reduce

the error (retroactively) between P_t and T_t. Generically, for stochastic *update vector* $\Delta_t \in \mathbb{R}^d$, the weights are modified as

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \boldsymbol{\alpha}_t \circ \Delta_t \tag{4.1}$$

for a vector step-size $\boldsymbol{\alpha}_t$, where the operator $\circ$ denotes element-wise multiplication. Given an update vector, the goal is to select $\boldsymbol{\alpha}_t$ to reduce error, into the future. Semi-gradient methods like temporal difference learning follow a similar scheme, but Δ_t is not the gradient of an objective function. Throughout this chapter, we will specify learning algorithms in terms of Δ_t and $\boldsymbol{\alpha}_t$, and assume the weights are updated according to Equation 4.1.

This setting leads to three natural restrictions on the algorithms that we consider.

1. First, we require that the algorithms have computational complexity linear in the number of input parameters, $d \in \mathbb{N}$. Many problems of practical importance have high-dimensional input spaces and a fine discretization of time, sometimes having only milliseconds between successive inputs. In these settings, quadratic-time algorithms are simply too expensive to be used in real time.

2. The second restriction we impose is that the methods can be used fully online and incrementally, processing data on a sample-by-sample basis. Oftentimes the stream of observations is non-stationary, or drifting over time; we want methods which can continuously integrate new data as it is observed.

3. Finally, we restrict ourselves to step-size adaptation algorithms which avoid decreasing the step-sizes to zero over time. As the step-sizes decay to zero, the learning process converges to a fixed solution; this is problematic in settings where the target signal is non-stationary, since the learner will no longer be able to track the target as it changes over time.

Luckily, there are at least two approaches which meet these restrictions: the meta-descent methods, and a family of quasi second-order methods derived from the AdaGrad algorithm. In the following sections, we discuss each of these approaches in turn, starting with the more widely-known quasi second-order methods.

4.1 Quasi Second-order Methods

Step-size adaptation for the stationary setting is often based on estimating second-order updates. The idea is to estimate the loss function $\ell : \mathbb{R}^d \to \mathbb{R}$ locally around the current weights $\mathbf{w}_t$ using a second-order Taylor series approximation—which requires the Hessian $\mathbf{H}_t \stackrel{\text{def}}{=} \nabla^2 \ell(\mathbf{w}_t)$. A closed-form solution can then be obtained for the approximation, because it is a quadratic function, giving the next candidate solution $\mathbf{w}_{t+1} = \mathbf{w}_t - (\mathbf{H}_t)^{-1} \nabla \ell(\mathbf{w}_t)$. If instead the Hessian is approximated—such as with a diagonal approximation—then we obtain *quasi second-order* updates. Taken to the extreme, with the Hessian approximated by a scalar, as $\mathbf{H}_t = \alpha_t^{-1}\mathbf{I}$, we obtain first-order gradient descent with a step-size of α_t. For the batch setting, the gains from second-order methods are clear, with a convergence rate of $O(1/t^2)$, as opposed to $O(1/t)$ for first-order descent.

These gains are not as clear in the stochastic setting, but diagonal approximations appear to provide an effective balance between computation and convergence rate improvements (Bordes *et al.* 2009). These approaches, however, still require $O(d^2)$ computation per step, making them less useful in practice for many online prediction scenarios. For example, online prediction in robotics applications often have high dimensional inputs (see Section 7.2), and the robot's clock-cycle is often on the order of milliseconds. Making quadratic-time parameter updates at each step is simply infeasible in such learning systems. Instead, we focus on quasi second-order methods which have linear time approximations. In particular, we focus on a family of quasi second-order methods which evolved from the AdaGrad algorithm (Duchi *et al.* 2011; McMahan and Streeter 2010).

4.1.1 AdaGrad

Many of the most widely-used approaches to adapting a vector of step-sizes are descendents of AdaGrad (Duchi *et al.* 2011; McMahan and Streeter 2010). The full quadratic-complexity AdaGrad algorithm updates parameters using

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{G}_t^{-1} \nabla \ell(\mathbf{w}_t)$$

where $\eta \in \mathbb{R}$ is a global learning rate shared by all parameters, and $\mathbf{G}_t \in \mathbb{R}^{d \times d}$ is the preconditioning matrix defined by

$$\mathbf{G}_t = \sqrt{\sum_{i=1}^{t} \nabla \ell(\mathbf{w}_i) \nabla \ell(\mathbf{w}_i)^\top}.$$

In practice, the quadratic-time algorithm can be too computationally expensive, as it involves inverting the $d \times d$ matrix $\mathbf{G}_t$ on each time step. Instead, a linear-time approximation to the AdaGrad algorithm is often used, where $\mathbf{G}_t$ is approximated by its diagonal entries, allowing the inverse to be computed in linear time. Let $\mathbf{v}_t \in \mathbb{R}^d$ be the vector $[\mathbf{v}_t]_i = [\mathbf{G}_t]_{i,i}$ for $i = 1, \ldots, d$. Using this approximation, $\mathbf{v}_t$ can be both updated and inverted in linear time, as desired. This leads to updates of the form

$$\begin{aligned}
\mathbf{v}_t &= \mathbf{v}_{t-1} + \nabla \ell(\mathbf{w}_t)^2 \\
\boldsymbol{\alpha}_t &\stackrel{\text{def}}{=} \frac{\eta}{\sqrt{\mathbf{v}_t} + \epsilon} \\
\Delta_t &\stackrel{\text{def}}{=} -\nabla \ell(\mathbf{w}_t)
\end{aligned}$$

where $\mathbf{v}_0 \stackrel{\text{def}}{=} \mathbf{0}$ and $\epsilon \in \mathbb{R}$ is a small positive constant to prevent division by zero.

At a high-level, AdaGrad might be seen as trying to "even out" how much each of the features contributes to the overall learning process. In vanilla SGD, features which are frequently active and/or often produce high-magnitude gradient components disproportionately influence the learning process. This can effectively drown out the influence of rare but otherwise informative features. By setting the step-sizes $\boldsymbol{\alpha}_t$ to be inversely proportional to $\sum_{i=1}^{t} \nabla \ell(\mathbf{w}_i)^2$, AdaGrad prevents over-active features from dominating the learning process.

4.1.2 RMSProp and AdaDelta

The step-size adaptation rule of Adagrad has some notable drawbacks: the first is that learning rates monotonically decay to zero over time, causing learning progress to eventually halt. This makes AdaGrad ill-suited to continual learning problems with non-stationary dynamics. The second drawback

is that AdaGrad can be sensitive to initial conditions; parameters with large error gradients at the beginning of training will be stuck with small learning rates for the rest of the training.

The RMSProp algorithm (Tieleman and Hinton 2012) seeks to address these problems by instead accumulating the squared gradients over a fixed-size window, approximated as an exponentially decaying average of the squared gradients $\mathbf{v}_t$:

$$\begin{aligned}\mathbf{v}_t &= (1-\beta)\mathbf{v}_{t-1} + \beta\nabla\ell(\mathbf{w}_t)^2 \\ \boldsymbol{\alpha}_t &\stackrel{\text{def}}{=} \frac{\eta}{\sqrt{\mathbf{v}_t}+\epsilon} \\ \Delta_t &\stackrel{\text{def}}{=} -\nabla\ell(\mathbf{w}_t)\end{aligned}$$

where $\beta \in (0,1)$ is a fixed smoothing parameter[1] and $\mathbf{v}_0 \stackrel{\text{def}}{=} \mathbf{0}$.

AdaDelta (Zeiler 2012) uses the same preconditioner as RMSProp — approximating the sum of squared gradients with a exponentially decaying average — but adds an additional unit correction term, $\mathbf{m}_t$. The idea is that if we assume that $\mathbf{w}_t$ has some hypothetical units, the update $\boldsymbol{\alpha}_t \circ \Delta_t$ applied to $\mathbf{w}_t$ ought to have those same units. The authors propose using an exponentially decaying average of the update vectors Δ_t to make this correction, resulting in the update equations

$$\begin{aligned}\mathbf{v}_t &= (1-\beta)\mathbf{v}_{t-1} + \beta\nabla\ell(\mathbf{w}_t)^2 \\ \boldsymbol{\alpha}_t &\stackrel{\text{def}}{=} \frac{\sqrt{\mathbf{m}_t}}{\sqrt{\mathbf{v}_t}+\epsilon} \\ \Delta_t &\stackrel{\text{def}}{=} -\nabla\ell(\mathbf{w}_t) \\ \mathbf{m}_{t+1} &= (1-\beta)\mathbf{m}_t + \beta\Delta_t^2.\end{aligned}$$

The nice thing about this unit correction term is that it is used in place of the parameter η in the RMSProp algorithm, resulting in one less hyperparameter to be tuned. In practice, however, the ability to tune a global learning rate tends to result in better performance from RMSProp.

[1] We use the convention that β is small (near 0, rather than near 1) throughout this thesis for consistency. The opposite convention (as used in Kingma and Ba (2015), Tieleman and Hinton (2012), and Zeiler (2012)) can be used by making the change of variables $\rho \stackrel{\text{def}}{=} 1-\beta$.

Finally, it is interesting to note the relationship between the RMSProp-style preconditioner and local curvature of the loss function. The term $\mathbf{v}_t$ is an exponentially decaying average of recent squared gradients, so the preconditioner $\boldsymbol{\alpha}_t = \frac{\eta}{\sqrt{\mathbf{v}_t}+\epsilon}$ scales each component inversely proportional to its squared magnitude in the local region. This is intuitively similar to the behaviour that would result from using $\boldsymbol{\alpha}_t \stackrel{\text{def}}{=} \text{diag}(\nabla^2 \ell(\mathbf{w}_t))^{-1}$ — each component of the gradient is scaled based on how "steep" the loss function is within a local region of $\mathbf{w}_t$. In fact, many works suggest that RMSProp-style preconditioners are incremental approximations of a diagonal Hessian (Dauphin *et al.* 2015; Hazan *et al.* 2007; Kingma and Ba 2015; Martens 2014; Pascanu and Bengio 2013). For this reason, we consider methods which use preconditioners of this sort to be quasi second-order methods.

4.1.3 Adam and AMSGrad

Like RMSProp, Adam (Kingma and Ba 2015) locally estimates the decaying average of previous squared gradients $\mathbf{v}_t$, and adapts the learning rates according to it:

$$\begin{aligned}\mathbf{v}_t &= (1-\beta_2)\mathbf{v}_{t-1} + \beta_2 \nabla \ell(\mathbf{w}_t)^2 \\ \boldsymbol{\alpha}_t &\stackrel{\text{def}}{=} \frac{\eta}{\sqrt{\mathbf{v}_t}+\epsilon}\end{aligned}$$

for global step-size $\eta \in \mathbb{R}$, decay parameter $\beta_2 \in (0,1)$, stability constant ϵ, and $\mathbf{v}_0 \stackrel{\text{def}}{=} \mathbf{0}$.

Unlike RMSProp and AdaDelta, Adam adds a form of momentum: instead of using the error gradient $\nabla \ell(\mathbf{w}_t)$ directly, Adam uses an exponentially decaying average of the previous gradients.

$$\mathbf{m}_t = (1-\beta_1)\mathbf{m}_{t-1} + \beta_1 \nabla \ell(\mathbf{w}_t)$$

for decay parameter $\beta_1 \in (0,1)$, and $\mathbf{m}_0 \stackrel{\text{def}}{=} \mathbf{0}$. The two estimates $\mathbf{v}_t$ and $\mathbf{m}_t$ are additionally bias-corrected to account for the fact that their initialization

biases them towards zero.

$$\hat{\mathbf{m}}_t \stackrel{\text{def}}{=} \frac{\mathbf{m}_t}{1-(1-\beta_1)^t}$$
$$\hat{\mathbf{v}}_t \stackrel{\text{def}}{=} \frac{\mathbf{v}_t}{1-(1-\beta_2)^t}.$$

Using these definitions, the update rules are then:

$$\boldsymbol{\alpha}_t \stackrel{\text{def}}{=} \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t}+\epsilon}$$
$$\Delta_t \stackrel{\text{def}}{=} -\hat{\mathbf{m}}_t$$

where η is again a global learning rate and ϵ is a small constant added for numerical stability.

Reddi *et al.* (2018) point out an issue with the convergence proof of Adam, leading them to a slightly different preconditioner. AMSGrad uses (mostly) the same updates as the Adam algorithm, but instead of preconditioning with $\frac{1}{\sqrt{\mathbf{v}_t}+\epsilon}$ they use a running estimate of the *maximum* $\mathbf{v}_t$ encountered so far

$$\widetilde{\mathbf{v}}_t = \max(\widetilde{\mathbf{v}}_{t-1}, \mathbf{v}_t)$$
$$\boldsymbol{\alpha}_t \stackrel{\text{def}}{=} \frac{\eta}{\sqrt{\widetilde{\mathbf{v}}_t}+\epsilon}$$

where $\widetilde{\mathbf{v}}_0 = \mathbf{0}$.

Of the algorithms discussed so far, only AdaGrad and AMSGrad have rigorous convergence guarantees. However, these guarantees hold only when the global step-size η decays to zero over time (*e.g.* $\eta_t = \frac{\eta}{\sqrt{t}}$). Thus, these convergence results (and equivalent regret bounds) are not entirely relevant to our continual online prediction setting, since we generally require the ability to track a non-stationary target T_t continually. We thus do not consider convergence or regret bounds further in this thesis.

4.2 Meta-descent Methods

The *meta-descent* strategies directly learn step-sizes that minimize the same objective as the base learner. A simpler set of such methods, called *hypergradient* methods (Almeida *et al.* 1998; Baydin *et al.* 2018; Jacobs 1988), adjust

the step-size based on its impact on the weights on a single step. Hypergradient Descent (HD) (Baydin *et al.* 2018) takes the gradient of the loss $\ell(\mathbf{w})$ w.r.t. a scalar step-size $\alpha > 0$, to get the meta-gradient for the step-size as $\partial \ell(\mathbf{w}_t)/\partial \alpha = -\nabla_{\mathbf{w}} \ell(\mathbf{w}_{t-1})^\top \nabla_{\mathbf{w}} \ell(\mathbf{w}_t)$. The update simply requires storing the vector $\nabla_{\mathbf{w}} \ell(\mathbf{w}_{t-1})$ and updating $\alpha_{t+1} = \alpha_t + \eta \nabla_{\mathbf{w}} \ell(\mathbf{w}_{t-1})^\top \nabla_{\mathbf{w}} \ell(\mathbf{w}_t)$, for a meta step-size $\eta > 0$. More generally, meta-descent methods, like IDBD (R. S. Sutton 1992a) and SMD (N. N. Schraudolph 1999), try to control the loss function indirectly, using the step-sizes. The step-sizes are adapted by optimizing a *meta-objective*

$$\min_{\boldsymbol{\alpha} > 0} \mathbb{E}[\ell(\mathbf{w}_t(\boldsymbol{\alpha}))|\mathbf{w}_0].$$

Intuitively, the idea is that the step-sizes influence the trajectory of the parameters $(\mathbf{w}_0, \ldots, \mathbf{w}_t)$, and the parameters influence the loss function, so the loss function can be indirectly controlled using the step-sizes. The meta-objective is optimized via SGD by considering the impact of the step-sizes back in time, through the weights:

$$\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} = \sum_j^d \frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i}, \tag{4.2}$$

where $w_{t,j}(\boldsymbol{\alpha})$ denotes j-th element in vector $\mathbf{w}_t(\boldsymbol{\alpha})$. The goal is to approximate this gradient efficiently, usually using a recursive strategy. Below, we show the derivation for the SMD algorithm for concreteness, as well as to correct an error in the original derivation[2].

4.2.1 Stochastic Meta-Descent

Our main focus for meta-descent methods is *Stochastic Meta-descent* (SMD) (N. N. Schraudolph 1999), which is a generalization of many of its predecessors (N. Schraudolph 1998; R. S. Sutton 1992a; R. S. Sutton 1992b). Given a twice-differentiable loss function $\ell : \mathbb{R}^d \to \mathbb{R}$, SMD attempts to adapt a vector of step-sizes $\boldsymbol{\alpha} \in \mathbb{R}^d$ using stochastic gradient descent, minimizing the loss

[2]The discrepency between the proposed algorithm and the SMD derivation has been noticed in prior works (A. Mahmood 2010), but it was not shown why this discrepency is an error; we do so by re-deriving the algorithm.

with respect to the step-sizes. We compute the gradient of the loss function $\ell(\mathbf{w}_t(\boldsymbol{\alpha}))$, w.r.t. step-size. We derive the full quadratic-complexity algorithm to start, and then introduce approximations to obtain a linear-complexity algorithm. For step-size α_i as the ith element in the vector $\boldsymbol{\alpha}$, we can write the i^{th} component of $\nabla_{\boldsymbol{\alpha}}\ell(\mathbf{w}_t(\boldsymbol{\alpha}))$ as

$$\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} = \sum_j^d \frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i}.$$

Define the following two vectors, for $w_{t,j}$ the j-th element in vector $\mathbf{w}_t(\boldsymbol{\alpha})$,

$$\Delta_{t,j}(\mathbf{w}_t(\boldsymbol{\alpha})) \stackrel{\text{def}}{=} -\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial w_{t,j}} \in \mathbb{R}^d \qquad \text{the gradient update} \tag{4.3}$$

$$\boldsymbol{\psi}_{t,i} \stackrel{\text{def}}{=} \frac{\partial \mathbf{w}_t(\boldsymbol{\alpha})}{\partial \alpha_i} \in \mathbb{R}^d. \tag{4.4}$$

For notational simplicity, we supress the arguments of Δ_t and $\mathbf{w}_t$ unless relevant for computing derivatives. We can obtain vector $\boldsymbol{\psi}_{t,i}$ recursively as

$$\begin{aligned}
\boldsymbol{\psi}_{t+1,i} &= \frac{\partial}{\partial \alpha_i}(\mathbf{w}_t + \boldsymbol{\alpha} \circ \Delta_t) = \frac{\partial \mathbf{w}_t(\boldsymbol{\alpha})}{\partial \alpha_i} + \boldsymbol{\alpha} \circ \frac{\partial \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ \sum_j \frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ (\mathbf{H}_t \boldsymbol{\psi}_{t,i}) + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= (\mathbf{I} - \operatorname{diag}(\boldsymbol{\alpha})\mathbf{H}_t)\boldsymbol{\psi}_{t,i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix}.
\end{aligned}$$

The resulting generic updates for quadratic-complexity SMD, with meta step-size η, are:

$$\begin{aligned}
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(\eta \boldsymbol{\alpha}_{t-1} \circ \boldsymbol{\Psi}_t^\top \Delta_t\right) \\
\mathbf{H}_t &\stackrel{\text{def}}{=} \nabla^2_{\mathbf{w}_t} \ell(\mathbf{w}_t) \\
\boldsymbol{\Psi}_{t+1} &= (\mathbf{I} - \operatorname{diag}(\boldsymbol{\alpha}_t)\mathbf{H}_t)\boldsymbol{\Psi}_t + \operatorname{diag}(\Delta_t).
\end{aligned} \tag{4.5}$$

where $\boldsymbol{\Psi} \in \mathbb{R}^{d \times d}$ is the matrix with $[\boldsymbol{\Psi}_t]_{:,i} = \boldsymbol{\psi}_{t,i}$, $\boldsymbol{\psi}_{0,i} = \mathbf{0}$ and $\boldsymbol{\alpha}_0 = \alpha_0$ for some initial value $\alpha_0 \in \mathbb{R}$.

For the linear-complexity algorithm, we set entries $(\boldsymbol{\psi}_{t,i})_j = 0$ for $i \neq j$. Let $\mathbf{H}_{t,i}$ be the ith column of the Hessian. This results in the simplification

$$\begin{aligned}\boldsymbol{\psi}_{t+1,i} &= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ \sum_j^d \mathbf{H}_{t,j}(\boldsymbol{\psi}_{t,i})_j + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}\\ \mathbf{0}\end{bmatrix} \\ &= \boldsymbol{\psi}_{t,i} - \boldsymbol{\alpha} \circ \mathbf{H}_{t,i}(\boldsymbol{\psi}_{t,i})_i + \begin{bmatrix}\mathbf{0}\\ \Delta_{t,i}\\ \mathbf{0}\end{bmatrix}.\end{aligned}$$

Further, since we will then assume that $(\boldsymbol{\psi}_{t+1,i})_j = 0$ for $i \neq j$, we need not compute the full vector $\mathbf{H}_{t,i}(\boldsymbol{\psi}_{t,i})_i$. Instead, we only need to compute the ith entry, i.e., for $\frac{\partial \Delta_{t,i}(\mathbf{w}_t)}{\partial w_{t,i}}$. We can then instead define $\hat{\psi}_{t,i}$ to be a scalar approximating $\frac{\partial w_{t,i}(\boldsymbol{\alpha})}{\partial \alpha_i}$, with $\hat{\boldsymbol{\psi}}_t$ the vector of these, and the diagonal of the Hessian

$$\hat{\mathbf{h}}_t \stackrel{\text{def}}{=} \left[\frac{\partial^2 \ell(\mathbf{w}_t)}{\partial w_{t,1}^2}, \ldots, \frac{\partial^2 \ell(\mathbf{w}_t)}{\partial w_{t,d}^2}\right] \tag{4.6}$$

to define the recursion as $\hat{\boldsymbol{\psi}}_{t+1} \stackrel{\text{def}}{=} \hat{\boldsymbol{\psi}}_t - \boldsymbol{\alpha} \circ \hat{\mathbf{h}}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t$, with $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$. The gradient using this approximation, with off-diagonals zero, is:

$$\begin{aligned}\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} &= \sum_j^d \frac{\partial \ell(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} \\ &\approx \frac{\partial \ell(\mathbf{w}_t)}{\partial w_{t,i}} \frac{\partial w_{t,i}(\boldsymbol{\alpha})}{\partial \alpha_i} \\ &= \hat{\psi}_{t,i} \Delta_{t,i}.\end{aligned}$$

The resulting update to the step-size is

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} + \eta \hat{\boldsymbol{\psi}}_t \circ \Delta_t \\ \hat{\boldsymbol{\psi}}_{t+1} &= \left(\mathbf{I} - \boldsymbol{\alpha}_t \circ \hat{\mathbf{h}}_t\right) \circ \hat{\boldsymbol{\psi}}_t + \Delta_t.\end{aligned} \tag{4.7}$$

In practice, it's common for meta-descent approaches to constrain the step-sizes $\boldsymbol{\alpha}$ to be an exponential function, $\boldsymbol{\alpha}_t = \exp(\boldsymbol{\beta}_t)$ for $\boldsymbol{\beta}_t \in \mathbb{R}^d$. This allows the step-sizes to remain strictly positive, while also allowing the step-sizes $\boldsymbol{\alpha}$ to be updated using geometric steps. The updates are similar to the above, except now we minimize the loss w.r.t. $\boldsymbol{\beta}$

$$\frac{\partial \ell(\mathbf{w}_t(\boldsymbol{\alpha}(\boldsymbol{\beta})))}{\partial \beta_i} = \sum_j^d \frac{\partial \ell(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} \frac{\partial \alpha_i(\boldsymbol{\beta})}{\partial \beta_i}. \tag{4.8}$$

making a change of variables $\hat{\boldsymbol{\psi}}_t' \stackrel{\text{def}}{=} \hat{\boldsymbol{\psi}}_t \circ \boldsymbol{\alpha}$, the update to $\boldsymbol{\alpha}$ becomes

$$\begin{aligned}\boldsymbol{\alpha}_t &= \exp(\boldsymbol{\beta}_t) = \exp(\boldsymbol{\beta}_{t-1} + \eta\hat{\boldsymbol{\psi}}_t' \circ \Delta_t) \\ &= \exp(\boldsymbol{\beta}_{t-1}) \exp(\eta\hat{\boldsymbol{\psi}}_t' \circ \Delta_t) = \boldsymbol{\alpha}_{t-1} \exp(\eta\hat{\boldsymbol{\psi}}_t' \circ \Delta_t),\end{aligned}$$

and $\hat{\boldsymbol{\psi}}_t'$ is updated by $\hat{\boldsymbol{\psi}}_t' = \left(\mathbf{I} - \boldsymbol{\alpha}_t \circ \hat{\mathbf{h}}_t\right) \circ \hat{\boldsymbol{\psi}}_t' + \boldsymbol{\alpha}_t \circ \Delta_t$. After rearranging terms in the $\hat{\boldsymbol{\psi}}_t'$ update, we arrive at update equations similar to the original work (N. N. Schraudolph 1999)

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \circ \exp(\eta\hat{\boldsymbol{\psi}}_t' \circ \Delta_t) \tag{4.9}$$

$$\hat{\boldsymbol{\psi}}_{t+1}' = \hat{\boldsymbol{\psi}}_t' + \boldsymbol{\alpha}_t \circ \left(\Delta_t - \hat{\mathbf{h}}_t \circ \hat{\boldsymbol{\psi}}_t'\right). \tag{4.10}$$

Difference to original SMD algorithm: Now, surprisingly, the above algorithm differs from the algorithm given for SMD. But, the original derivation appears to have a flaw, where the gradients of weights taken w.r.t. to a vector of step-sizes is assumed to be a vector. Rather, with the off-diagonal approximation we use, it should be a diagonal matrix, resulting in a diagonal Hessian. For completeness, we include the original algorithm, which uses a full Hessian-vector product.

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(\eta\hat{\boldsymbol{\psi}}_t \circ \Delta_t\right) \\ \hat{\boldsymbol{\psi}}_{t+1} &= \hat{\boldsymbol{\psi}}_t + \boldsymbol{\alpha}_t \circ \left(\Delta_t - \mathbf{H}_t\hat{\boldsymbol{\psi}}_t\right).\end{aligned}$$

Note that a follow-up paper that tested SMD (Wu *et al.* 2018) uses this update, but does not have an error, because they use a *scalar* step size. In fact, in the SMD paper, if the step size had been a scalar, then their derivation would be correct.

4.2.2 IDBD

The IDBD algorithm (R. S. Sutton 1992a) is a special case of SMD, arising from linear LMS updates. Recall from Section 2.4 that the loss function for linear LMS problems is of the form $\ell(\mathbf{w}_t) = \frac{1}{2}(y_t - \mathbf{x}_t^\top \mathbf{w}_t)^2$ for target signal

$y_t \in \mathbb{R}$, input vector $\mathbf{x}_t \in \mathbb{R}^d$, and parameter weights $\mathbf{w}_t \in \mathbb{R}^d$. To apply the SMD update, we compute $\Delta_t = -\nabla_{\mathbf{w}_t}\ell(\mathbf{w}_t) = (y_t - \mathbf{x}_t^\top \mathbf{w}_t)\mathbf{x}_t$, and $\hat{\mathbf{h}}_t = \text{diag}(\nabla^2_{\mathbf{w}_t}\ell(\mathbf{w}_t)) = \mathbf{x}_t \circ \mathbf{x}_t$. Plugging these quantities into Equations 4.9 and 4.10 gives the updates

$$\begin{aligned}
\delta_t &\stackrel{\text{def}}{=} (y_t - \mathbf{x}_t^\top \mathbf{w}_t) \\
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \circ \exp(\eta \delta_t \mathbf{x}_t \circ \hat{\boldsymbol{\psi}}'_t) \\
\hat{\boldsymbol{\psi}}'_{t+1} &= \hat{\boldsymbol{\psi}}'_t + \boldsymbol{\alpha}_t \circ \left(\delta_t \mathbf{x}_t - \mathbf{x}_t \circ \mathbf{x}_t \circ \hat{\boldsymbol{\psi}}'_t\right).
\end{aligned}$$

The original IDBD proposed in (R. S. Sutton 1992a) rearranges the update for $\hat{\boldsymbol{\psi}}'$ and additionally adds positive bounding operation to give $\hat{\boldsymbol{\psi}}'$ the interpretation of a decaying memory of the recent weight changes.

$$\hat{\boldsymbol{\psi}}'_{t+1} = \hat{\boldsymbol{\psi}}'_t \circ \max\left(\mathbf{0}, \mathbf{1} - \mathbf{x}_t \circ \mathbf{x}_t \circ \boldsymbol{\alpha}_t\right) + \delta_t \boldsymbol{\alpha}_t \circ \mathbf{x}_t \tag{4.11}$$

In Chapters 6 and 7, we implement IDBD using the update for $\hat{\boldsymbol{\psi}}'$ in Equation 4.11.

4.2.3 TIDBD

TIDBD (Kearney *et al.* 2018) is the temporal-difference learning counterpart to IDBD. Like IDBD, TIDBD can similarly be derived as a stochastic meta-descent algorithm. The derivation proceeds using a loss function $\ell(\mathbf{w}_t) = \frac{1}{2}\delta_t^2$, where $\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}; \mathbf{w}_t) - \hat{v}(S_t; \mathbf{w}_t)$ is the one-step *TD error* at time t (see Section 2.5). The main deviation from the typical SMD derivation is the use of a *semi-gradient* update, where the term $\hat{v}(S_{t+1}; \mathbf{w}_t)$ is treated as if it were part of the target signal and constant w.r.t. $\mathbf{w}_t$, leading to $\nabla\ell(\mathbf{w}_t) = \delta_t \mathbf{x}_t$. Additionally, TIDBD updates $\hat{\boldsymbol{\psi}}_t$ using an update analogous to Equation 4.11. The derivation then follows using steps analogous to those in Section 4.2.1, resulting in update equations

$$\begin{aligned}
\mathbf{z}_t &= \lambda\gamma \mathbf{z}_{t-1} + \mathbf{x}_t \\
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(\eta \delta_t \mathbf{x}_t \circ \hat{\boldsymbol{\psi}}'_t\right) \\
\hat{\boldsymbol{\psi}}'_{t+1} &= \hat{\boldsymbol{\psi}}'_t \circ \max(\mathbf{0}, \mathbf{1} - \boldsymbol{\alpha}_t \circ \mathbf{x}_t \circ \mathbf{z}_t) + \delta_t \boldsymbol{\alpha}_t \circ \mathbf{z}_t.
\end{aligned}$$

4.3 Conclusion

In this chapter, we introduced our problem setting — continual online prediction — and two approaches to adapting a vector of step-sizes in this setting: a family of quasi second-order methods and the meta-descent methods.

It is interesting to note that although the quasi second-order methods have become strongly associated with supervised learning and training deep neural networks in recent years, these methods were originally derived from an online convex optimization (OCO) perspective. In the OCO setting, the agent proposes a solution $\mathbf{w}_t \in \mathbb{R}$ at each time step. After the solution has been proposed, a convex loss function ℓ_t is revealed and the agent receives a loss of $\ell_t(\mathbf{w}_t)$. This is somewhat similar to our setting, insofar as one could imagine that the $\mathbf{w}_t$ proposed by the agent parameterizes some function approximation (such as $\hat{y}_t \stackrel{\text{def}}{=} \mathbf{x}_t^\top \mathbf{w}_t$), and the loss returned is an error metric, such as $\ell_t(\mathbf{w}_t) \stackrel{\text{def}}{=} (y_t - \hat{y}_t)^2$. Yet despite being designed for similar problem settings, there has yet to be an in-depth empirical comparison of the meta-descent and quasi second-order strategies presented in this chapter. We provide the first such comparison in Chapter 6 and Chapter 7.

In our discussion of meta-descent methods, we omit various extensions to the base algorithms. The IDBD algorithm is extended in (A. R. Mahmood *et al.* 2012) to include heuristics for normalizing the step-size and avoiding overshooting. TIDBD also has an analogous extension (Kearney *et al.* 2019). However, normalization and overshooting heuristics could, in theory, be applied to any of the adaptive algorithms we've discussed, including the quasi second-order methods. We omit such extensions from further consideration, focusing instead on the base algorithms.

Chapter 5

Adaptive Gain for Stability

Tracking — continually updating the weights with recent experience — contrasts the typical goal of convergence. Much of the previous algorithm development for step-size adaptation, however, has been towards the aim of convergence, with algorithms like AdaGrad and AMSGrad that decay step-sizes over time. Assuming finite representational capacity, there may be aspects of the problem that can never be accurately modeled or predicted by the agent. In these partially observable problems tracking and thus treating the problem as if it were non-stationary can improve prediction accuracy compared with methods that converge (R. Sutton *et al.* 2007). In continual online prediction, we assume the agent's task is partially observable in this way and develop a new step-size method that can facilitate tracking.

We treat the learning system as a dynamical system—where the weight update is based on stochastic updates known to suitably track the targets—and consider the choice of step-size as the inputs to the system to maintain *stability*. Such a view has been previously considered under adaptive gain for least-mean squares (LMS) (Benveniste *et al.* 1990, Chapter 4), where weights are treated as state following a random drift. To generalize this idea to other incremental algorithms, we propose a general criterion based on the magnitude of the update vector.

A criteria for $\boldsymbol{\alpha}$ to maintain stability in the system is to keep the norm of the update vector Δ_t small

$$\min_{\boldsymbol{\alpha}>0} \mathbb{E}\left[\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2 \;\middle|\; \mathbf{w}_0\right]. \tag{5.1}$$

The update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ on this time step is dependent on the step-size $\boldsymbol{\alpha}$ because that step-size influences the parameter weights $\mathbf{w}_t$ and past updates. The expected value is over all possible update vectors $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ for the given step-size and assuming the system started with some $\mathbf{w}_0$. If the system is ergodic, $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ does not depend on the initial $\mathbf{w}_0$, and is only driven by the underlying state dynamics and the choice of $\boldsymbol{\alpha}$. The step-size can be seen as a control input for this system, with the goal to maintain a stable dynamical system by minimizing $\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2$ over time.

We derive an algorithm to estimate $\boldsymbol{\alpha}$ for this dynamical system, which we call AdaGain: Adaptive Gain for Stability. The algorithm is derived for a generic update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$ that is differentiable w.r.t. the weights $\mathbf{w}_t(\boldsymbol{\alpha})$. To simplify the notation, we suppress the arguments of Δ_t and $\mathbf{w}_t$ where they are unneeded.

5.1 AdaGain with Quadratic Complexity

We derive the full quadratic-complexity algorithm to start, and then introduce approximations to obtain a linear-complexity algorithm. To minimize (5.1), we use stochastic gradient descent, and thus need to compute the gradient of $\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2$ w.r.t. the step-size $\boldsymbol{\alpha}$. For step-size α_i as the ith element in the vector $\boldsymbol{\alpha}$, and $w_{t,j}$ the j-th element in vector $\mathbf{w}_t$, we can write the i^{th} component of the gradient as

$$\begin{aligned}\frac{\frac{1}{2}\partial\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2}{\partial\alpha_i} &= \Delta_t^\top \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial\alpha_i} \\ &= \Delta_t^\top \sum_j^d \frac{\partial\Delta_t(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial\alpha_i}.\end{aligned}$$

The key, then, is to track how a change in the weights impacts the update and how changes in the step-size impact the weights. The first term can be computed instantaneously on this step. For the second term, however, the impact of the step-size on the weights goes back further to previous updates. Let $\boldsymbol{\psi}_{t,i} \stackrel{\text{def}}{=} \frac{\partial\mathbf{w}_t(\boldsymbol{\alpha})}{\partial\alpha_i} \in \mathbb{R}^d$. We show how to obtain a recursive form for this

step-size gradient as follows:

$$\begin{aligned}
\boldsymbol{\psi}_{t+1,i} &= \frac{\partial \mathbf{w}_{t+1}}{\partial \alpha_i} = \frac{\partial}{\partial \alpha_i}(\mathbf{w}_t + \boldsymbol{\alpha} \circ \Delta_t) \\
&= \frac{\partial \mathbf{w}_t(\boldsymbol{\alpha})}{\partial \alpha_i} + \boldsymbol{\alpha} \circ \frac{\partial \Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ \sum_j \frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ (\mathbf{G}_t \boldsymbol{\psi}_{t,i}) + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\
&= (\mathbf{I} + \operatorname{diag}(\boldsymbol{\alpha})\mathbf{G}_t)\boldsymbol{\psi}_{t,i} + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix}, \qquad (5.2)
\end{aligned}$$

where $\mathbf{G}_{t,j} \stackrel{\text{def}}{=} \frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,j}} \in \mathbb{R}^d$, $\mathbf{G}_t \stackrel{\text{def}}{=} [\mathbf{G}_{t,1}, \ldots, \mathbf{G}_{t,d}] \in \mathbb{R}^{d \times d}$. Therefore, $\boldsymbol{\psi}_{t+1,i}$ represents a sum of updates, with a recursive weighting on previous $\boldsymbol{\psi}_{t,i}$ adjusting the weight of previous updates in the sum.

We can approximate the gradient using this recursive relationship, without storing all previous samples. Though the above updates are exact, we obtain an approximation when implementing such a recursive form in practice. When using $\boldsymbol{\psi}_{t-1,i}$ computed on the last time step $t-1$, this gradient estimate is in fact w.r.t. the previous step-size $\boldsymbol{\alpha}_{t-2}$, rather than $\boldsymbol{\alpha}_{t-1}$. Thus, for many steps into the past, the accumulated gradients in $\boldsymbol{\psi}_{t,i}$ are likely inaccurate. To improve the approximation, and forget old gradients, we introduce a forgetting parameter $0 < \beta < 1$, which focuses the accumulation of gradients in $\boldsymbol{\psi}_{t,i}$ to a more recent window (see Equation 5.4).

The gradient update to the step-size also needs to ensure that the step-sizes remain positive. Similarly to IDBD, we use an exponential form for the step-size, where $\boldsymbol{\alpha} = \exp(\boldsymbol{\beta})$ and $\boldsymbol{\beta} \in \mathbb{R}^d$ is updated with (unconstrained) stochastic gradient descent. Conveniently, we do not need to maintain this auxiliary variable, and can simply directly update $\boldsymbol{\alpha}$ using $\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t \circ \exp(-\eta \boldsymbol{\alpha}_t \circ \mathbf{g}_t)$, where all operations are taken element-wise and $\mathbf{g}_t = \frac{\frac{1}{2}\partial \|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2}{\partial \boldsymbol{\alpha}_t}$. This update is derived in Section 5.3, along with a discussion of other possible approaches to maintaining non-negative step-sizes.

The resulting generic updates for quadratic-complexity AdaGain, with meta step-size η, are

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \circ \exp\left(-\eta \boldsymbol{\alpha}_{t-1} \circ (\boldsymbol{\Psi}_t^\top \mathbf{G}_t^\top \Delta_t)\right) \tag{5.3}$$

$$\boldsymbol{\psi}_{t+1,i} = (1-\beta)\boldsymbol{\psi}_{t,i} + \beta \left(\boldsymbol{\alpha}_t \circ (\mathbf{G}_t \boldsymbol{\psi}_{t,i}) + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \right) \tag{5.4}$$

where the exponential is applied element-wise, $\boldsymbol{\psi}_{0,i} = \mathbf{0}$, $\boldsymbol{\alpha}_0 = \alpha_0$ for some initial value $\alpha_0 \in \mathbb{R}$, and $(\boldsymbol{\Psi}_t)_{:,i} = \boldsymbol{\psi}_{t,i}$ with $\boldsymbol{\Psi}_t \in \mathbb{R}^{d \times d}$. For computational efficiency to avoid matrix-matrix multiplication, the order of multiplication for $\boldsymbol{\Psi}_t^\top \mathbf{G}_t^\top \Delta_t$ should start from the right, as $\boldsymbol{\Psi}_t^\top (\mathbf{G}_t^\top \Delta_t)$. The key complexity in deriving an AdaGain update, then, is simply in computing the Jacobian $\mathbf{G}_t$; given this, the remainder of the algorithm is fixed. For each update $\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))$, the Jacobian will be different, but is straightforward to compute.

5.2 AdaGain with Linear Complexity

Maintaining the entire matrix $\boldsymbol{\Psi}_t$ can be prohibitively expensive. As was done in IDBD (R. S. Sutton 1992a), one way to avoid maintaining this matrix is to assume that $\frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} = 0$ for $i \neq j$. This heuristic reflects that α_i is likely to have the largest impact on $w_{t,i}$, and less impact on the other entries in $\mathbf{w}_t$.

The modification above for this heuristic is straightforward, simply by setting entries $(\boldsymbol{\psi}_{t,i})_j = 0$ for $i \neq j$. This results in the simplification of Equation 5.2 to

$$\begin{aligned} \boldsymbol{\psi}_{t+1,i} &= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ \sum_j^d \mathbf{G}_{t,j} (\boldsymbol{\psi}_{t,i})_j + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix} \\ &= \boldsymbol{\psi}_{t,i} + \boldsymbol{\alpha} \circ \mathbf{G}_{t,i} (\boldsymbol{\psi}_{t,i})_i + \begin{bmatrix} \mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0} \end{bmatrix}. \end{aligned}$$

Further, since we will then assume that $(\boldsymbol{\psi}_{t+1,i})_j = 0$ for $i \neq j$, there is no purpose in computing the full vector $\mathbf{G}_{t,i}(\boldsymbol{\psi}_{t,i})_i$. Instead, we only need to compute the ith entry, i.e., for $\frac{\partial \Delta_{t,i}(\mathbf{w}_t)}{\partial w_{t,i}}$. We can then instead define $\hat{\psi}_{t,i}$ to be a scalar approximating $\frac{\partial w_{t,i}(\boldsymbol{\alpha})}{\partial \alpha_i}$, with $\hat{\boldsymbol{\psi}}_t$ the vector of these, and $\hat{\mathbf{j}}_t \stackrel{\text{def}}{=}$

$\left[\frac{\partial \Delta_{t,1}(\mathbf{w}_t)}{\partial w_{t,1}}, \ldots, \frac{\partial \Delta_{t,d}(\mathbf{w}_t)}{\partial w_{t,d}}\right]$ to define the recursion as $\hat{\boldsymbol{\psi}}_{t+1} \stackrel{\text{def}}{=} \hat{\boldsymbol{\psi}}_t + \boldsymbol{\alpha} \circ \hat{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t$, with $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$. The gradient using this approximation, with off-diagonals zero, is

$$\begin{aligned}
\frac{\frac{1}{2}\partial \|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|_2^2}{\partial \alpha_i} &= \Delta_t^\top \sum_j^d \frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,j}} \frac{\partial w_{t,j}(\boldsymbol{\alpha})}{\partial \alpha_i} \\
&\approx \Delta_t^\top \frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,i}} \frac{\partial w_{t,i}(\boldsymbol{\alpha})}{\partial \alpha_i} \\
&= \hat{\psi}_{t,i} \mathbf{G}_{t,i}^\top \Delta_t
\end{aligned}$$

To compute this approximation, for all i, we still need to be able to compute $\mathbf{G}_t^\top \Delta_t$. In some cases this is straightforward, as is the case for linear TD (found in Section 5.4.2). More generally, we can use R-operators (Pearlmutter 1994) to compute this Jacobian-vector product, or a simple finite difference approximation, as we do in Section 5.4.1. Therefore, because we can compute this Jacobian-vector product in linear time, the only approximation is to $\hat{\boldsymbol{\psi}}_t$. The update, with forgetting parameter $\beta \in [0, 1]$, is

$$\begin{aligned}
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(-\eta \; \boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top \Delta_t)\right) \\
\hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_t + \beta\left(\boldsymbol{\alpha}_t \circ \hat{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t\right).
\end{aligned} \tag{5.5}$$

These approximations parallel diagonal approximations for second-order techniques, which similarly assume off-diagonal elements are zero. Further, $\mathbf{G}_t$ itself is a gradient of the update w.r.t. the weights, where this update was already likely the gradient of the loss w.r.t. the weights. Thus, this $\mathbf{G}_t$ contains similar information as the Hessian. The AdaGain update, therefore, contains some information about curvature but allows for updates that are not necessarily (true) gradient updates.

This AdaGain update is generic but does require computing the Jacobian of a given update, which could be onerous in certain settings. We provide an update based on finite differences in Section 5.4.1 which only requires differences between updates. We have found that this approach also works well in practice.

5.3 Maintaining Non-negative Stepsizes

One straightforward option to maintain non-negative step-sizes is to define a constraint on the step-size. We can prevent the step-size from going below a small threshold ϵ (e.g., $\epsilon = 0.001$), ensuring positive step-sizes. The projection onto this constraint set after each gradient descent step simply involves applying the operator $(\cdot)_\epsilon$, which thresholds any values below $\epsilon > 0$ to ϵ. The drawback to this simple approach, however, is that it introduces another hyperparameter to tune: the threshold value ϵ.

Another option—and the one we use in this work—is to use an exponential form for the step-size, $\boldsymbol{\alpha} = \exp(\boldsymbol{\beta})$, so that it remains positive. The algorithm, with or without an exponential form, remains essentially identical to the thresholded version, because

$$\frac{\frac{1}{2}\partial\|\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}(\boldsymbol{\beta})))\|_2^2}{\partial\beta_i} = \Delta_t^\top \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial\alpha_i}\frac{\partial\alpha_i(\boldsymbol{\beta})}{\partial\beta_i}.$$

Therefore, we can still recursively estimate the gradient with the same approach, regardless of how the step-size $\boldsymbol{\alpha}$ is constrained. For the thresholded form, we simply use the gradient $\Delta_t^\top \frac{\partial\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial\alpha_i}$ and then project (i.e., threshold). For the exponential form, the gradient update for $\boldsymbol{\alpha}$ is simply used within an exponential function, as described below.

Consider directly maintaining $\boldsymbol{\beta}$, which is unconstrained. For the function form $\alpha_i = \exp(\beta_i)$, the partial derivative $\frac{\partial\alpha_i(\boldsymbol{\beta})}{\partial\beta_i}$ is simply equal to α_i and so the gradient update includes an additional α_i in front. This can more explicitly be maintained, without an additional variable, by noticing that for gradient $g_i = \alpha_i\Delta_t^\top \frac{\Delta_t(\mathbf{w}_t(\boldsymbol{\alpha}))}{\partial\alpha_i}$ for $\beta_{t,i}$, the step-size $\alpha_{t,i}$ can be updated using:

$$\begin{aligned}
\alpha_{t+1,i} &= \exp(\beta_{t+1,i}) \\
&= \exp(\beta_{t,i} - \eta g_i) \\
&= \exp(\beta_{t,i})\exp(-\eta g_i) \\
&= \alpha_{t,i}\exp(-\eta g_i).
\end{aligned}$$

Therefore, we can still directly maintain $\boldsymbol{\alpha}$. The resulting update to $\boldsymbol{\alpha}$ is

simply

$$\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \exp\left(-\eta \boldsymbol{\alpha}_t \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top \Delta_t)\right). \tag{5.6}$$

Other multiplicative updates are also possible. SMD (N. N. Schraudolph 1999) uses an exponential update, but uses an approximation with a maximum, to avoid the expensive computation of the exponential function. Hypergradient descent (Baydin *et al.* 2018) uses a similar multiplicative update, but without a maximum.

5.4 Specific AdaGain Updates

In this section, we provide a selection of concrete AdaGain updates. We begin with a general-purpose finite-difference approximation which can be computed without the matrix-vector product $\mathbf{G}^\top \Delta_t$. We then derive the update equations for a common learning algorithm: semi-gradient TD(λ). AdaGain for LMS updates falls out of the semi-gradient TD(λ) updates as a special case. Finally, we give concretely the update equations for a variant of AdaGain which uses an RMSProp base learner, combining a quasi second-order update with meta-descent.

5.4.1 Finite-difference AdaGain

One advantage of AdaGain is that it is derived generically, allowing extensions to many online algorithms, unlike IDBD, and variants which are derived specifically for the squared TD-error. To avoid requiring knowledge about the algorithm update and its derivatives, we can provide an approximation to the Jacobian-vector product and the diagonal of the Jacobian, using finite differences. As long as the update function for the algorithm can be queried multiple times, this algorithm can be easily applied to any update.

To compute the Jacobian-vector product, we use the fact that this corresponds to a directional derivative. Notice that $\mathbf{G}_t^\top \Delta_t$ corresponds to the vector of directional derivatives for each component (function) in the update Δ_t, in the direction of $\mathbf{u} = \Delta_t$, because the dot-product separates in $\mathbf{G}_{t,1}^\top \mathbf{u}, \ldots, \mathbf{G}_{t,d}^\top \mathbf{u}$. Therefore, for update function $\Delta : \mathbb{R}^d \to \mathbb{R}^d$ (such as the gradient of the loss),

we get the approximation

$$\mathbf{G}_t^\top \Delta_t \approx \frac{\Delta(\mathbf{w} + r\mathbf{u}) - \Delta(\mathbf{w} - r\mathbf{u})}{2r}, \tag{5.7}$$

where $r \in \mathbb{R}$ is a small constant value. For the diagonal of the Jacobian, we can again use finite differences. An efficient finite difference computation is proposed within the simultaneous perturbation stochastic approximation algorithm (Spall 1992), which uses a random perturbation vector $\mathbf{p}$ to compute the centered difference $\frac{(\Delta(\mathbf{w}+r\mathbf{p})-\Delta(\mathbf{w}-r\mathbf{p}))_i}{2r\mathbf{p}_i}$. This formula provides an approximation to the gradient of the i entry in the update Δ_t with respect to weight i; when computed for all i, this approximates the diagonal of the Jacobian $\hat{\mathbf{j}}_t$. To avoid additional computation, we can re-use the above difference with perturbation $\mathbf{u}$, rather than a random vector $\mathbf{p}$. To avoid division by zero, if $\mathbf{u}$ contains a zero entry, we threshold the normalization with a small constant ϵ to give

$$\hat{\mathbf{j}}_t \approx \frac{\Delta(\mathbf{w} + r\mathbf{u}) - \Delta(\mathbf{w} - r\mathbf{u}))}{2r} \circ (1/\text{sign}(\mathbf{u}) \max(\epsilon, |\mathbf{u}|)) \tag{5.8}$$

where division is element-wise. Another approach would be to sample a random direction $\mathbf{p}$ for this finite difference and use $\Delta(\mathbf{w}+\mathbf{p}) - \Delta(\mathbf{w})$, divided by the absolute value of each element of $\mathbf{p}$. We found empirically that using the same direction as Δ_t was actually more effective, and more computationally efficient, so we use that approach.

Using these approximations, we can compute the update to the step-size as in Equation (5.5), repeated here for easy reference:

$$\begin{aligned}
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(-\eta \boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ (\mathbf{G}_t^\top \Delta_t)\right) \\
\hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_t + \beta\left(\boldsymbol{\alpha}_t \circ \hat{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t\right).
\end{aligned}$$

5.4.2 AdaGain for Linear TD(λ)

In this section, we derive the updates for a particular algorithm, namely semi-gradient linear TD(λ). We first provide the AdaGain updates for linear TD(λ), and then provide the derivation below. Recall from Section 2.5 that for TD(λ),

the update is

$$\begin{aligned}\delta_t &\stackrel{\text{def}}{=} r_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t \\ \Delta_t &\stackrel{\text{def}}{=} \delta_t \mathbf{z}_t \\ \mathbf{z}_t &\stackrel{\text{def}}{=} \lambda\gamma \mathbf{z}_{t-1} + \mathbf{x}_t.\end{aligned}$$

for discount rate γ, bootstrapping parameter λ, and $\mathbf{z}_0 \stackrel{\text{def}}{=} \mathbf{0}$. Using these definitions, AdaGain for linear TD(λ) updates the step-sizes using:

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp(-\eta(\Delta_t^\top \mathbf{z}_t)\boldsymbol{\alpha}_{t-1} \circ \mathbf{d}_t \circ \hat{\boldsymbol{\psi}}_t) \\ \hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_{t,i} + \beta\left(\boldsymbol{\alpha}_t \circ \mathbf{z}_t \circ \mathbf{d}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t\right)\end{aligned} \tag{5.9}$$

where $\hat{\boldsymbol{\psi}}_0 = \mathbf{0}$.

To derive the update for $\boldsymbol{\alpha}$, we need to compute the gradients of the updates, particularly $\frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,i}}$, or for the full algorithm, the Jacobian $\mathbf{G}$ as follows:

$$\begin{aligned}\frac{\partial \Delta_t(\mathbf{w}_t)}{\partial w_{t,i}} &= \mathbf{z}_t \frac{\partial \delta_t(\mathbf{w}_t)}{\partial w_{t,i}} \\ &= \mathbf{z}_t \frac{\partial}{\partial w_{t,i}}(r_{t+1} + \gamma_{t+1}\mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t) \\ &= \mathbf{z}_t \left(\gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t\right)_i.\end{aligned}$$

Letting $\mathbf{d}_t \stackrel{\text{def}}{=} \gamma_{t+1}\mathbf{x}_{t+1} - \mathbf{x}_t$, the Jacobian is $\mathbf{G}_t = \mathbf{z}_t \mathbf{d}_t^\top$ and the diagonal approximation is $\mathbf{g}_t \stackrel{\text{def}}{=} \mathbf{z}_t \circ \mathbf{d}_t$. The quadratic complexity algorithm uses $\mathbf{G}_t$ as given:

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp(-\eta(\Delta_t^\top \mathbf{d}_t)\boldsymbol{\alpha}_{t-1} \circ (\boldsymbol{\Psi}_t^\top \mathbf{z}_t)) \\ \boldsymbol{\psi}_{t+1,i} &= (1-\beta)\boldsymbol{\psi}_{t,i} + \beta\left(\boldsymbol{\alpha}_t \circ (\mathbf{z}_t \mathbf{d}_t^\top \boldsymbol{\psi}_{t,i}) + \begin{bmatrix}\mathbf{0} \\ \Delta_{t,i} \\ \mathbf{0}\end{bmatrix}\right).\end{aligned}$$

The linear complexity algorithm uses $\mathbf{g}_t$ to update $\hat{\boldsymbol{\psi}}_t$, giving the step-size update in (5.9):

$$\begin{aligned}\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp(-\eta(\Delta_t^\top \mathbf{d}_t)\boldsymbol{\alpha}_{t-1} \circ \mathbf{z}_t \circ \hat{\boldsymbol{\psi}}_t) \\ \hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_{t,i} + \beta\left(\boldsymbol{\alpha}_t \circ \mathbf{z}_t \circ \mathbf{d}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t\right).\end{aligned}$$

5.4.3 AdaGain for Linear LMS Updates

Another common update is the Linear Least Mean Squares update. In this setting, $\Delta_t = (y_t - \mathbf{x}_t^\top \mathbf{w}_t)\mathbf{x}_t$ for a given target signal $y_t \in \mathbb{R}$. Notice that the LMS Update equation falls directly out of the TD(λ) update equation by letting $\gamma = \lambda = 0$. Thus, the update equations for AdaGain applied to LMS updates can be found by simply replacing λ and γ everywhere they appear. In particular, the trace $\mathbf{z}_t = \lambda\gamma\mathbf{z}_{t-1} - \mathbf{x}_t$ becomes $-\mathbf{x}_t$, and $\mathbf{d}_t = \gamma\mathbf{x}_{t+1} - \mathbf{x}_t$ becomes $-\mathbf{x}_t$ as well. The updates therefore become

$$\begin{aligned}
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp(-\eta(\Delta_t^\top \mathbf{x}_t)\boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ \mathbf{x}_t) \\
\hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_{t,i} + \beta\left(\boldsymbol{\alpha}_t \circ \mathbf{x}_t \circ \mathbf{x}_t \circ \hat{\boldsymbol{\psi}}_t + \Delta_t\right).
\end{aligned}$$

5.4.4 AdaGain with RMSProp

Step-size adaptation is often desirable when working with difficult optimization surfaces. It is often the case that the objective being optimized is sensitive to some components of the parameter vector $\mathbf{w}_t$, making smaller step-sizes desireable. Likewise, the objective may be insensitive to other components of the parameter vector, requiring larger step-sizes in order to significantly influence the objective. Unsurprisingly, these difficulties extend to the *meta-objective* in meta-descent strategies, since these methods use the step-sizes $\boldsymbol{\alpha}$ to influence the weights $\mathbf{w}_t$, thereby indirectly influencing the objective. Intuitively, if the objective is difficult to control directly using the parameters $\mathbf{w}_t$, it will be similarly difficult to control the objective through $\mathbf{w}_t(\boldsymbol{\alpha})$ using the step-sizes. Because of this, the meta-optimization can be ineffective, resulting in ineffective step-size adaptation.

To help alleviate these difficulties, we propose using meta-descent methods in conjunction with quasi second-order preconditioners. The idea is that the conditioning of AdaGain's meta-objective can be improved by minimizing the norm of a better-conditioned update vector. For a given base learning update Δ_t, we can apply AdaGain to a *preconditioned* update vector $\widetilde{\Delta}_t \stackrel{\text{def}}{=} \text{diag}(\mathbf{H}_t)^{-1} \circ \Delta_t$ to minimize the objective $\mathbb{E}[\|\widetilde{\Delta}_t(\mathbf{w}_t(\boldsymbol{\alpha}))\|^2 | \mathbf{w}_0]$. In particular, using the RMSProp preconditioner equates to applying AdaGain to an RMSProp base

learner.

Other preconditioners are possible, of course; one could apply AdaGain to any of the popular quasi second-order updates. However, each of the quasi second-order updates has its own hyperparameters to tune in addition to those of AdaGain. This would be an infeasible number of hyperparameters to thoroughly tune, so we constrain the quasi second-order update to share the hyperparameters used by AdaGain. For example, an RMSProp update has a global step-size η and a smoothing parameter β; we constrain these values to be the same as AdaGain's meta step-size and forgetting parameter respectively. In our preliminary experiments, we found that applying AdaGain to an RMSProp base learner generally worked better than the other quasi second-order approaches given the constraints on hyperparameters. The update equations for AdaGain with RMSProp, which we refer to as *RMSGain*, are:

$$\begin{aligned}
\mathbf{v}_t &= (1-\beta)\mathbf{v}_{t-1} + \beta\Delta_t^2 \\
\widetilde{\Delta}_t &= \frac{\Delta_t}{\sqrt{\mathbf{v}_t} + \epsilon} \\
\boldsymbol{\alpha}_t &= \boldsymbol{\alpha}_{t-1} \exp\left(-\eta\ \boldsymbol{\alpha}_{t-1} \circ \hat{\boldsymbol{\psi}}_t \circ (\widetilde{\mathbf{G}}_t^\top \widetilde{\Delta}_t)\right) \\
\hat{\boldsymbol{\psi}}_{t+1} &= (1-\beta)\hat{\boldsymbol{\psi}}_t + \beta\left(\boldsymbol{\alpha}_t \circ \widetilde{\mathbf{j}}_t \circ \hat{\boldsymbol{\psi}}_t + \widetilde{\Delta}_t\right)
\end{aligned}$$

where $\mathbf{v}_0 = \mathbf{0}$ and ϵ is a small constant to prevent division by zero. $\widetilde{\mathbf{G}}_t$ and $\widetilde{\mathbf{j}}_t$ are the jacobian and its diagonal approximation, respectively, of $\widetilde{\Delta}_t$. We do not take the gradient through the preconditioner $\frac{1}{\sqrt{\mathbf{v}_t}+\epsilon}$, treating it instead as a constant w.r.t. $\mathbf{w}_t$.

5.5 Conclusion

In this chapter, we introduced a new meta-descent algorithm called AdaGain. AdaGain is built on a generic update scheme and can be applied to a wide variety of base learners. While previous meta-descent methods attempt to adapt the step-sizes in a way that minimizes the loss function, AdaGain attempts to adapt the step-sizes in a way which minimizes the norm of the base learner's stochastic update vector Δ_t. In this way, AdaGain can be seen as adapting

the step-sizes to optimize learning stability. Furthermore, this objective gave us a principled way to combine meta-descent methods with quasi-second order methods: one can simply apply AdaGain to a preconditioned update vector. In particular, when using the RMSProp preconditioner, we refer to the resulting algorithm as *RMSGain*. In the following chapters, we demonstrate that RMSGain can lead to significant performance improvements over the existing meta-descent methods.

Chapter 6

Experiments in Stationary Settings

We conducted a series of experiments to investigate the performance characteristics of quasi second-order and meta-descent methods in settings with stationary dynamics. We begin with two difficult stationary problems: minimizing the Rosenbrock function, and off-policy state-value learning in a variation of Baird's "star" counterexample.

The goal in this chapter is to investigate the performance of each algorithm on difficult stationary tasks, before moving on to non-stationary tasks in Chapter 7. This allows us to first build intuitions about these approaches in isolation from the additional complexities of non-stationarity. The Rosenbrock function has both steep regions and flat regions, requiring the algorithms to be able to efficiently raise and lower their step-sizes when in different regions of the input space. Outside of the flat regions, the steep surfaces additionally pose a challenge to learning stability, as the gradients are very large. On the other hand, Baird's counterexample is an off-policy state-value prediction problem that generates notoriously unstable learning iterates in TD methods, leading to divergence for any scalar step-size. The ability to gracefully handle unstable learning iterates is a desirable property in continual online prediction problems, since we expect the algorithms to run reliably for undetermined lengths of time.

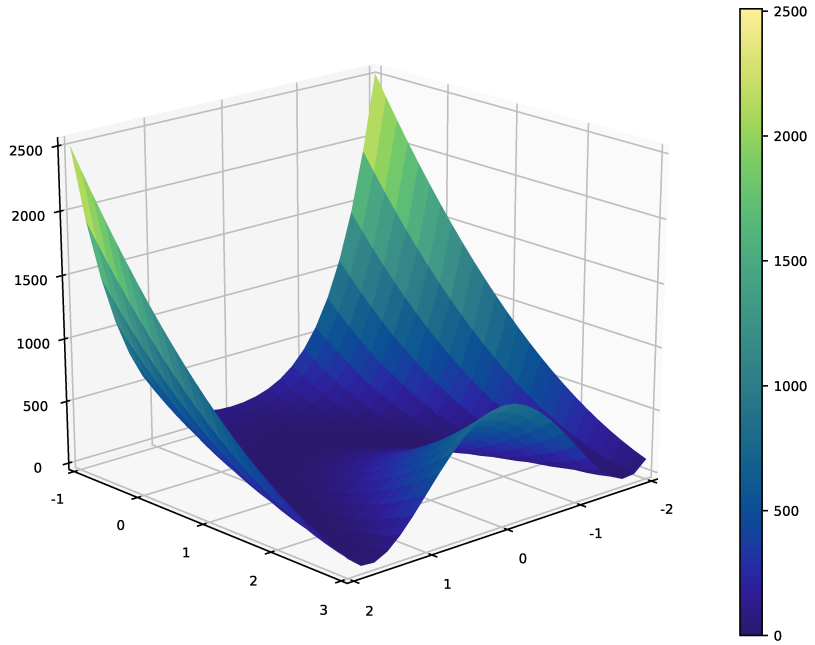

Figure 6.1: A surface plot of the Rosenbrock function.

6.1 Unconstrained Optimization

In our first experiment, we investigate performance on a stationary optimization task: finding the minimum of a difficult optimization surface. The Rosenbrock function is depicted in Figure 6.1, and is defined by :

$$Rosenbrock(x, y) = 100\left(y - x^2\right)^2 + (1 - x)^2 . \tag{6.1}$$

It has a minimum located at $(1, 1)$, in the middle of a flat parabolic region. Outside the flat parabolic region, the function swiftly increases in magnitude, reaching values in the thousands within a few unit steps. The difficulty in minimizing this function comes from the radically different step-sizes required in these two regions. Inside the flat region, the magnitude of the gradient is exceptionally small, so a large step-size is required to make significant progress. On the other hand, outside this region the gradient has a high magnitude, requiring small step-sizes so as to not over-shoot the region containing the minima. Thus, when the starting coordinates (x_0, y_0) are not known in advance, an optimization algorithm must be able to efficiently increase/decrease its step-sizes in the flat and steep regions respectively.

We investigated four variants of AdaGain: 1) the linear complexity AdaGain with RMSProp (*RMSGain*), 2) the linear finite-difference approximation of RMSGain (*ApproxRMSGain*), 3) the quadratic complexity implementation of RMSGain (*QuadraticRMSGain*), and 4) the linear complexity AdaGain without RMSProp (*AdaGain*). For the RMSGain and its variants, the initial learning rate α_0, meta step-size η, and forgetting parameter β were each tuned over the values $\{2^{-i} : i = 0, \ldots, 12\}$. For vanilla AdaGain, the meta step-size η and forgetting parameter β were tuned over the same values, but the inital step-size α_0 had to be tuned over the range $\{2^{-i} : i = 10, \ldots, 22\}$ in due to instability in initial learning.

We compared the performance of AdaGain against several baselines including SGD, AMSGrad, RMSProp, and SMD. AMSGrad has three hyperparameters, which were tuned over the same range as RMSGain's parameters. RMSProp and SMD's two hyperparameters were both tuned over the range $\{1.4^{-i} : i = 1, \ldots, 46\}$. The step-size for SGD was tuned over a linear spacing of 2197 values in the range $[2^{-22}, 1]$.

The performance was measured using Equation 6.1 over 6000 optimization steps, and averaged over 100 independent runs. On each run, we selected the initial position on the surface (x_0, y_0) randomly from the set $\{(x, y) \in \mathbb{R}^2 : x, y \in [-5, 5]\}$. The hyperparameters for each algorithm were chosen according to the minimum area under the learning curve in order to capture both good initial performance and final performance.

Figure 6.2 shows the learning curves for this problem. The left graph shows a comparision of the different AdaGain variants. RMSGain, ApproxRMSGain, and Quadratic RMSGain all perform similarly on this problem, while AdaGain performs significantly worse on average. The right graph shows the comparison of the best-performing AdaGain variant, RMSGain, and the rest of the baselines. RMSGain and AMSGrad both learn faster on average than all other baseline methods considered, and achieve similar final performance. Further, two meta-descent methods, SMD and AdaGain without RMSProp perform poorly. SMD, in particular, performs about as well as SGD in this problem; this is because the best-performing instance of SMD on this problem is the

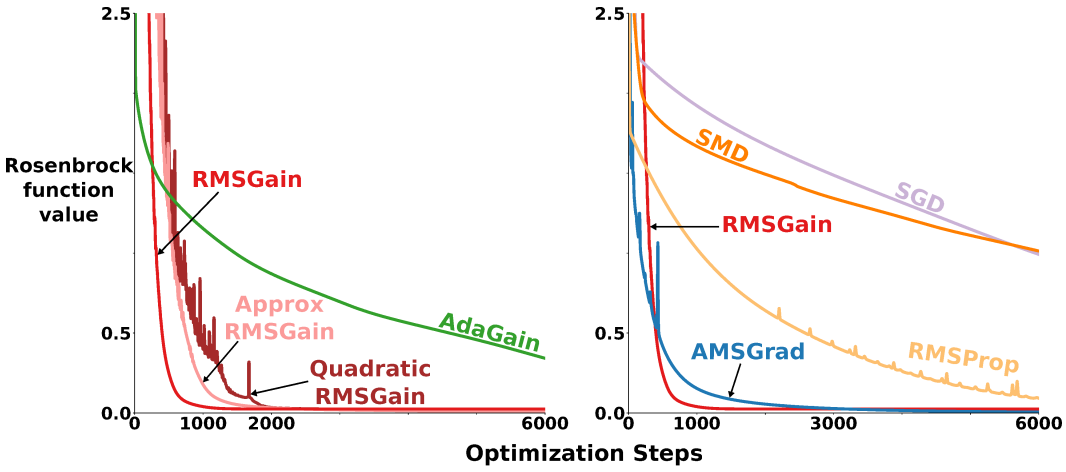

Figure 6.2: Learning curves for minimizing the Rosenbrock function. The left graph shows a comparison of the different AdaGain variants. The right graph shows a comparison of the best-performing AdaGain variant, RMSGain, and the baseline algorithms. The value of Equation 6.1 at each optimization step is shown on the y-axis. Results are averaged over 100 randomly selected initial positions (x_0, y_0).

one that barely adapts the step-sizes $\boldsymbol{\alpha}_t$ at all, suggesting that the meta-level optimization is sensitive to the initial conditions.

The result highlights issues with applying meta-descent approaches without considering the optimization surface, and the importance of having an algorithm like AdaGain which can be combined with quasi-second order methods. Furthermore, the result suggests that meta-descent and quasi second-order approaches can have complementary effects; RMSGain learns faster and gets closer to the minimum on average than both AdaGain and RMSProp on their own. Finally, the result suggests that there is not a significant loss in using the finite-difference approximation of RMSGain; the two implementations perform fairly similarly in this problem.

To better understand the average performance curves in Figure 6.2, we plotted the learning curve of each individual runs of the experiment. Figure 6.3 shows learning curves for RMSGain, AdaGain, SMD, SGD, RMSProp, and AMSGrad. In each graph, the colored lines show the learning curve for a single run of the experiment over the first 1500 steps. The learning curve averaged over runs is shown in black.

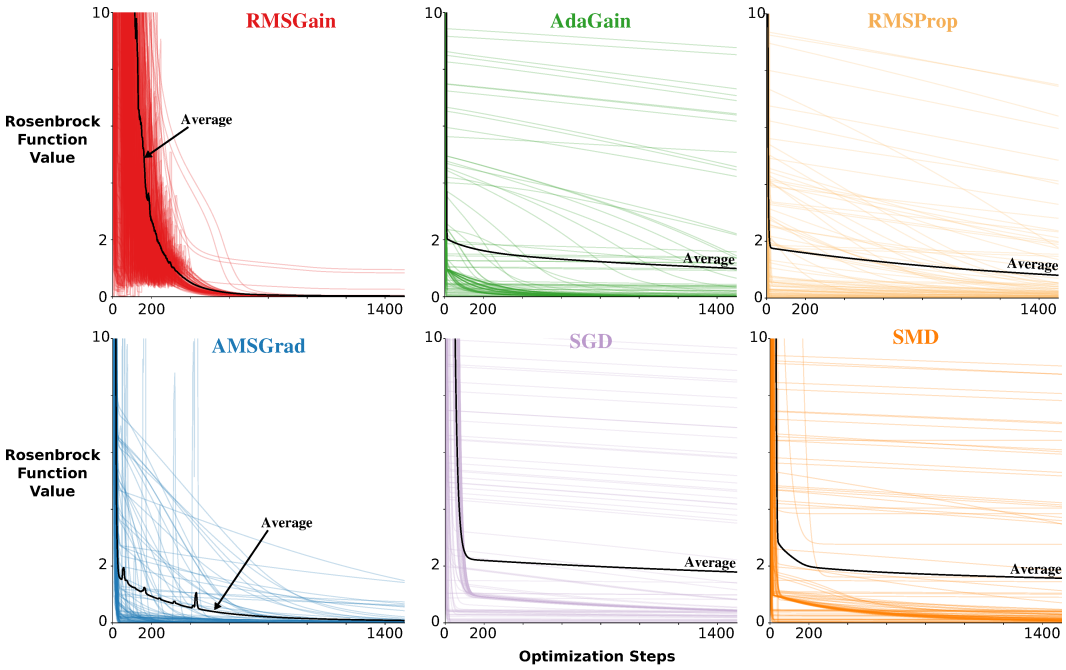

Figure 6.3: Individual learning curves for minimizing the Rosenbrock function. The learning curve for each of the 100 runs is shown in color, with the average of these curves shown in black. A randomly selected initial position (x_0, y_0) was selected in each run. The per-run learning curves for ApproxRMSGain and QuadraticRMSGain were similar to RMSGain and omitted in the interest of space.

The individual runs reveal that AdaGain, RMSProp, SGD, and SMD all have a significant number of runs in which the learning progress plateaus. This tends to occur when the algorithm gets into the flat parabolic region but is unable to adequately increase its step-sizes (see Figure 6.4). AMSGrad and RMSGain are fairly consistent in their behavior across runs, with AMSGrad a little less so. The individual learning curves of AMSGrad reveal several large performance spikes; these spikes indicate overshooting due to momentum and demonstrate one of the potential drawbacks of momentum-based strategies. Interestingly, RMSGain appears to be the only algorithm which isn't able to immediately minimize the function on at least one of the runs. This does make intuitive sense, however, since RMSGain's estimate $\mathbf{v}_t$ is noisy at the beginning of the experiment, which affects the estimate $\hat{\boldsymbol{\psi}}_t$ and in turn the step-sizes $\boldsymbol{\alpha}_t$. Despite these noisy initial estimates, RMSGain ends up consistently minimizing the function across runs.

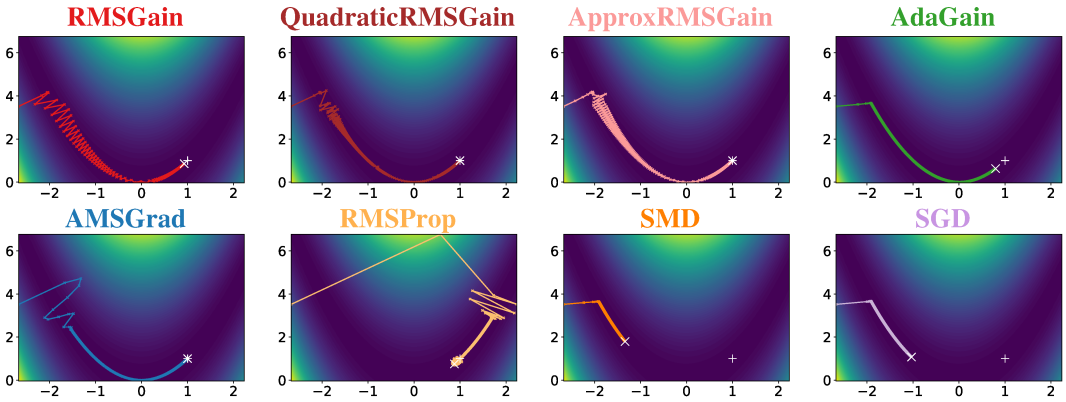

Figure 6.4: Optimization trajectories of a single run (with tuned meta-parameters) on the Rosenbrock function. A white × symbol denotes where in the input space the algorithm converged. The paths represent how each algorithm changes the weights while searching for the minimum. The white + symbol indicates the optimal value for the weights — if the × and + symbol overlap, the algorithm has reached the global minimum of the function.

Figure 6.4 shows the optimization trajectories for a run of the experiment which demonstrates many of the behaviors implied by Figure 6.3. At the beginning of the experiment, AMSGrad's momentum causes it to overshoot a few times before settling into the flat parabolic region. SMD and SGD get to the parabolic region quickly but make little progress within it, unable to significantly increase their learning rates. AdaGain fares a bit better but does not reach the global minima. RMSProp steps wildly during the initial learning, caused by the lack of bias correction on its preconditioner. The RMSGain variants all display a similar jittering pattern, in which optimization steps stay safely within the flat region, while still making significant learning progress.

Figure 6.5 shows a parameter sensitivity graph. For each algorithm, the plot shows a circle for each hyperparameter setting. The y-axis shows the value of Equation 6.1 at the end of the experiment, averaged over 100 runs. The circles are randomly perturbed on the x-axis to better see the density of points around each value on the y-axis. A threshold of 1.0 is set on the y-axis; settings which yield performance above the threshold are grouped together at the top of the plot, and the percentage of parameter settings that were above the threshold are shown as a percentage at the top (high percentages are bad).

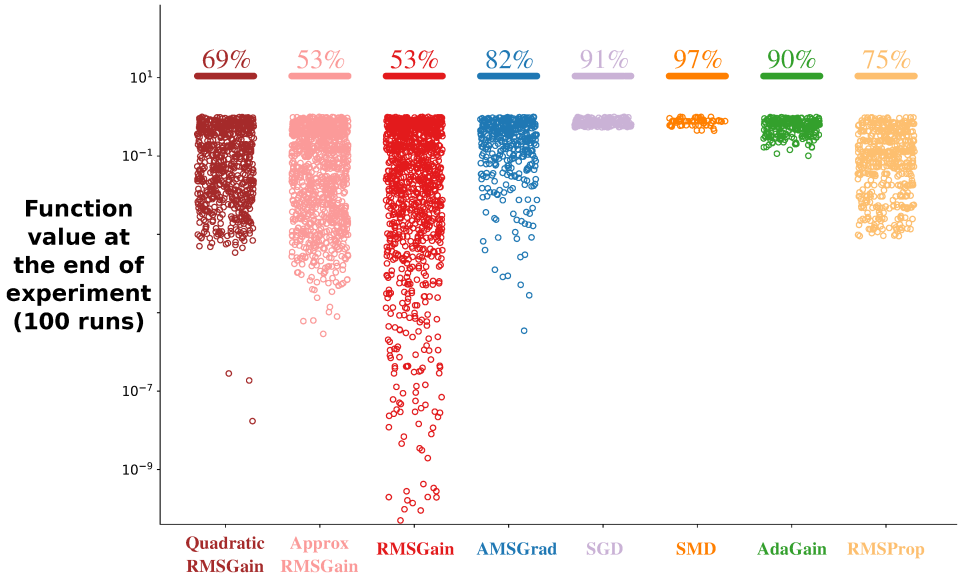

Figure 6.5: A waterfall plot for the final performance in the unconstrained optimization experiment. Each circle on the graph denotes the final value of the Rosenbrock function after 6000 optimization steps, averaged over 100 runs, for a particular parameter setting of the algorithm labeled on the x-axis. The points are randomly perturbed about the center of each column to give a visualization of the clustering of performances. A threshold of 1.0 is set on the y-axis; circles which are above this threshold are grouped together at the top, along with the percentage of all parameter settings which exceeded the threshold.

The plot shows that the algorithms which do not use an RMSProp-style preconditioner have the lowest density of parameter settings below the threshold, and generally achieve worse average final performance than the algorithms that use the RMSProp-style preconditioner. The RMSGain variants all have a higher density of settings below the threshold than the quasi second-order methods AMSGrad and RMSProp. Furthermore, the RMSGain variants all have at least one parameter setting which outperforms all parameter settings of the quasi second-order methods. This result suggests that RMSGain's combination of meta-descent and quasi second-order methods lead to more robust performance and lower sensitivity in terms of final performance on this problem.

The result also demonstrates that the linear RMSGain variants are generally less sensitive to their hyperparameters than the quadratic variant on

this problem. The sensitivity of Quadratic RMSGain is further demonstrated by three outlier parameter settings which reach a performance less than 10^{-6}, suggesting that the best-performing instance of this algorithm can be difficult to find. A possible explanation for this sensitivity is that the Hessian is generally ill-conditioned in this problem; using the full Hessian is likely making the updates more poorly conditioned, leading to greater sensitivity to the algorithm's hyperparameters.

Overall, the results in this experiment demonstrate that the combination of meta-descent methods and quasi second-order methods results in an approach to step-size optimization which 1.) is less sensitive to initial conditions, performing well across many varied initial points (x_0, y_0), 2.) can perform competitively with quasi second-order methods, and generally improves over both AdaGain and RMSProp on their own, and 3.) can be more robust to its hyperparameters.

6.2 Baird's Counterexample

Our next experiment investigates the ability of each optimization algorithm to handle unstable learning iterates. Baird's "star" counterexample is an off-policy state-value prediction problem which causes $TD(\lambda)$ to diverge for any

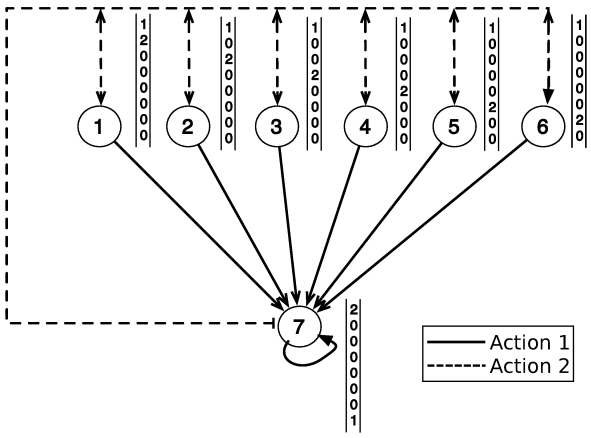

Figure 6.6: A variation of Baird's "star" counterexample. To the right of each state is the feature vector observed in that state. Figure adapted from White (2015).

scalar step-size (Baird 1995). We use a variation of the problem (Maei 2011), depicted in Figure 6.6. The MDP consists of seven states, and two actions available in each state: the first action leads deterministically to state 7 from any state (depicted by the solid arrows). The second action leads to one of states $1-6$ with equal probability (depicted by the dashed arrows). All transitions yield a reward of 0.

The task is an off-policy learning problem in which the agent must estimate the value function $v_\pi(\cdot)$ with discount rate $\gamma = 0.99$ and target policy $\pi(s) = 1$ (transition to state 7) in all states s. The behaviour policy chooses actions 1 and 2 with equal probability. The value function is approximated linearly as $\hat{v}_\pi(s) = \mathbf{w}^\top \mathbf{x}(s)$, where $\mathbf{w} \in \mathbb{R}^8$ is a weight vector and $\mathbf{x}(s) \in \mathbb{R}^8$ is the feature vector observed in state s (depicted to the right of each state in Figure 6.6).

The solution to this problem is simple: all transitions result in a reward of 0, so the value function under any policy π is $v_\pi(\cdot) = 0$. Likewise, to solve the problem we need to find a weight vector $\mathbf{w}^*$ such that $\hat{v}_\pi(s) = \mathbf{x}(s)^\top \mathbf{w}^* = 0$ for all states s. The feature vectors $\mathbf{x}(s)$ are all linearly independent, so this is an overdetermined system of equations with infinite possible solutions. In particular, $\mathbf{w}^* = c[-2, 1, 1, 1, 1, 1, 1, 4]$ for any $c \in \mathbb{R}$ exactly estimates $v_\pi(\cdot)$.

The difficulty of this problem comes in when a bad initialization of the weights is chosen. Initializing the weight vector as $\mathbf{w}_0 = [1, 1, 1, 1, 1, 1, 1, 10]^\top$, for example, leads to pathogenic behavior in TD(λ). First, note that any transitions to any of the states $1-6$ result in an importance sampling ratio $\rho_t = 0$, so $\mathbf{w}$ is updated only when transitioning to state 7. Suppose that the first transition to state 7 occurs from state $s \in \{1, \ldots, 6\}$, then the TD error $\delta_0 = 0 + \gamma \hat{v}_\pi(7) - \hat{v}_\pi(s) = 0.99 \cdot 12 - 3 > 0$, so weights w_0 and w_s will be increased, increasing the value of $\hat{v}_\pi(s) = w_0 + 2w_s$. Note that w_0 is also shared with state 7, so $\hat{v}_\pi(7) = 2w_0 + w_7$ increases as well. Thus $\frac{6}{7}$ of the transitions result in increasing the value of $\hat{v}_\pi(s)$ and $\hat{v}_\pi(7)$. In fact, the only time values are ever decreased is when transitioning from state 7 to itself, so values are increased much more often than they are decreased, leading to unbounded growth of the value estimates.

Suppose, however, that updates were made using a vector of step-sizes,

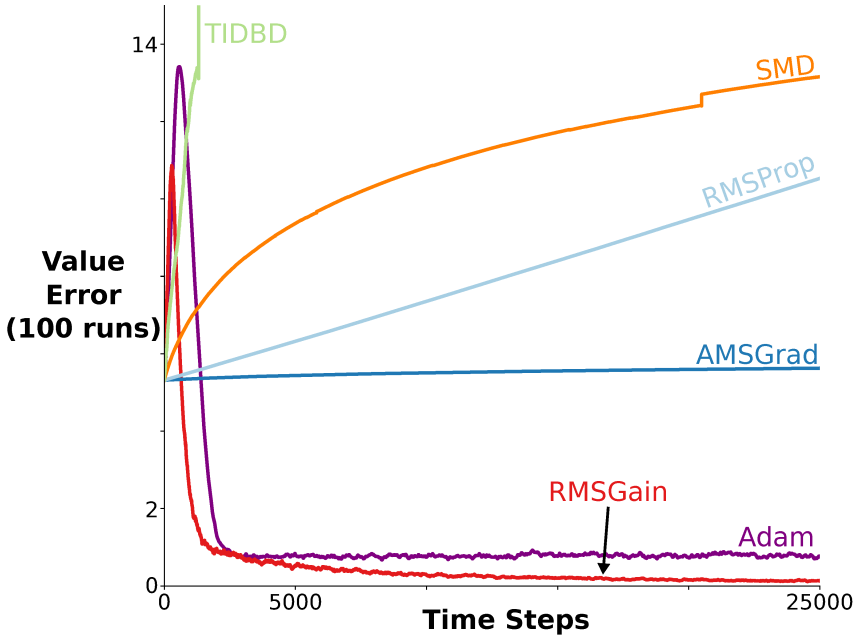

Figure 6.7: Learning curves in a variation of Baird's "star" counterexample. The y-axis shows the MSVE recorded at each time step, averaged over 100 independent runs of the experiment.

instead of a scalar step-size α. This allows several possible ways to avoid the pathogenic behavior in this problem. For example, one possible solution would be to cut the learning rate of w_0 to zero making w_0 constant, effectively removing the pathogenic generalization between states. In this experiment, we investigate whether the optimization algorithms can effectively adjust the step-sizes to handle the unstable learning iterates of this problem.

We applied Adam, AMSGrad, SMD, TIDBD , and RMSGain to this problem. Performance was measured for $25,000$ steps in terms of Mean Squared Value Error $\text{MSVE}(\mathbf{w}) = \frac{1}{|\mathcal{S}|}\sum_{s=1}^{|\mathcal{S}|}(\hat{v}_\pi(s;\mathbf{w}) - v_\pi(s;\mathbf{w}))^2$, and the results averaged over 100 independent runs of the experiment. The hyperparameters of AMSGrad, SMD, and RMSGain were tuned over the same values as in Section 6.1. Adam and TIDBD were tuned over the same values as AMSGrad and SMD respectively. The best-performing instance of each algorithm was chosen according to the setting which produced the smallest area under the learning curve.

Figure 6.7 shows the learning curves of each method. Only RMSGain,

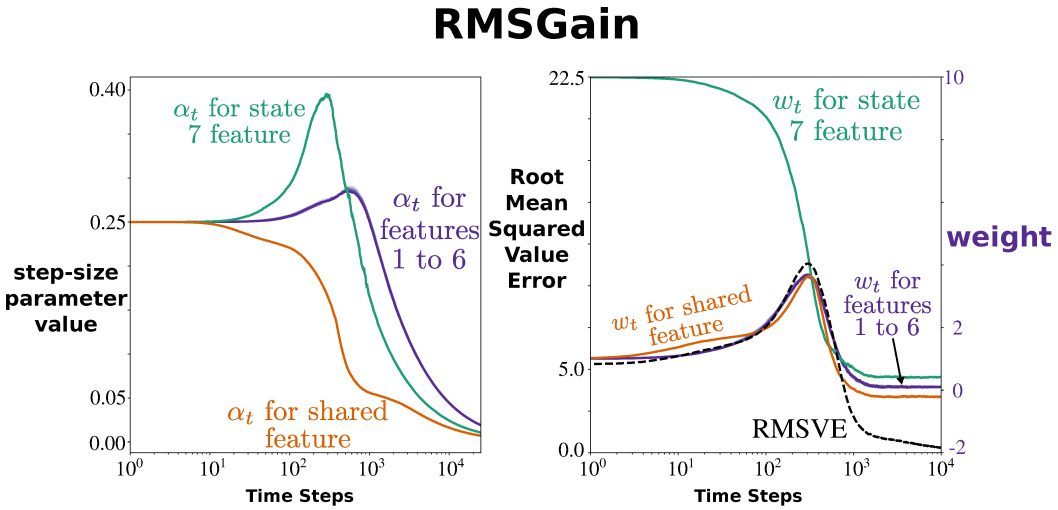

Figure 6.8: The step-size parameter values over time (left), and the corresponding weights (right) learned by RMSGain in Baird's counterexample, with results averaged over 1000 independent runs.

Adam, and AMSGrad are able to prevent divergence. SMD and RMSProp's performance is typical of Baird's counterexample: the hyperparameter search simply found parameters that caused slower divergence. RMSGain learns significantly faster than Adam and achieves lower error. AMSGrad fails to make any significant learning progress.

It is interesting to note that RMSProp quickly diverges, but Adam does not. Recall from Chapter 4 that these two methods both use the same preconditioning term, $\frac{1}{\sqrt{\mathbf{v}_t}+\epsilon}$ but differ in that Adam additionally uses a form of momentum while RMSProp does not. The fact that RMSProp diverges and Adam does not suggests that the preconditioner term is not what's responsible for preventing divergence in this problem — it's the momentum. In other words, Adam may be successful in this problem due to update averaging rather than step-size adaptation.

To understand how RMSGain prevents divergence consider Figure 6.8. The left graph shows the step-size values as they evolve over time, and the right graph shows the corresponding weights. Recall that the weight for feature seven is initialized to a high value. RMSGain initially increases feature seven's step-size causing weight seven to quickly fall. In parallel RMSGain reduces the step-size for the redundant feature, preventing incorrect generalization. Over

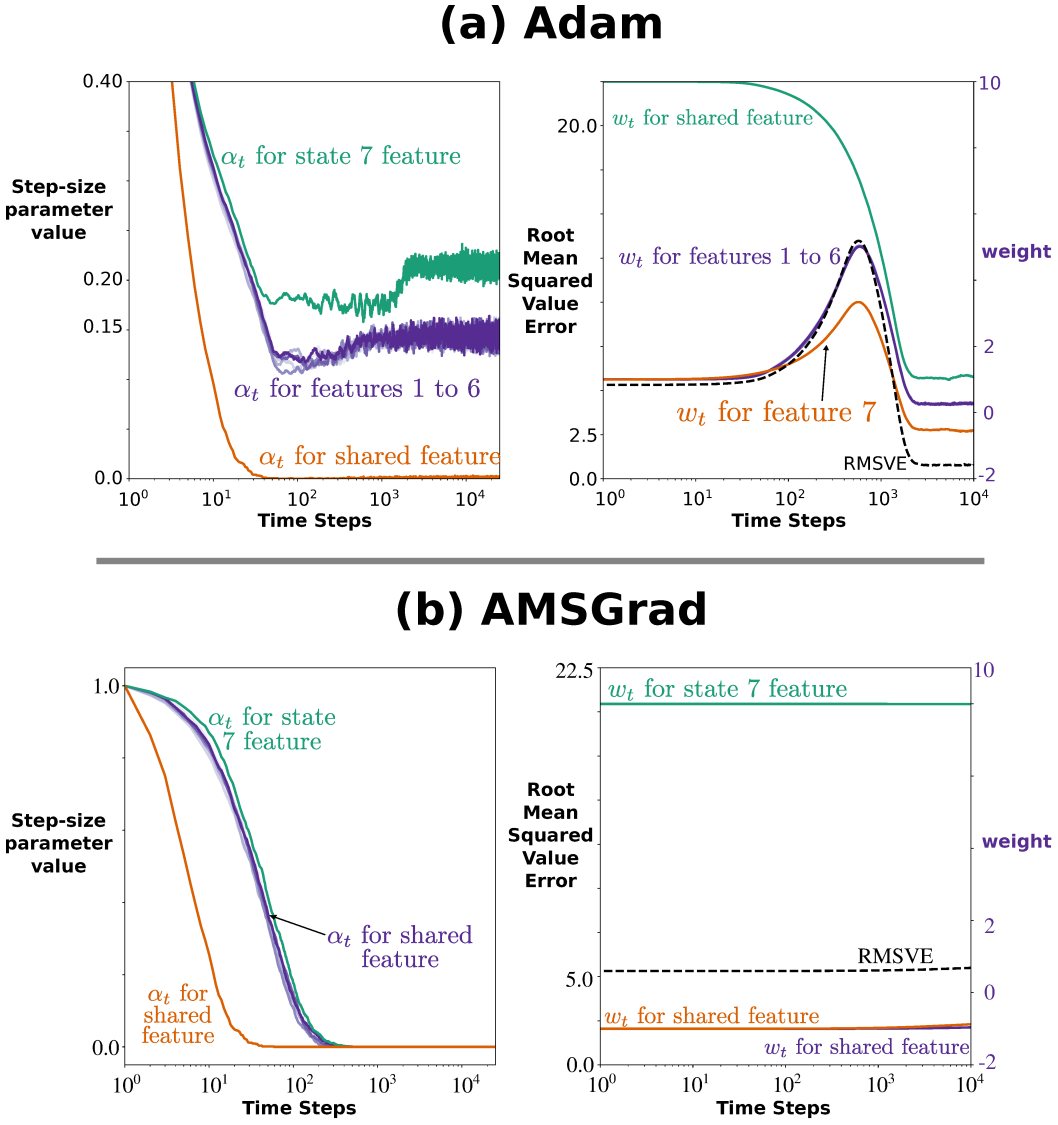

Figure 6.9: The step-size parameter values over time (left), and the corresponding weights (right) learned by a) Adam, and b) AMSGrad in Baird's counterexample. The results are averaged over 1000 independent runs.

time the weights converge to one of many valid solutions, and the value error, plotted in black on the right side, converges to zero.

The left plot of Figure 6.9 (a) show the evolution of the weights and step-sizes for Adam. Adam is successful in reducing the step-size for the redundant feature; however, the step-sizes of the other features decay quickly and then begin growing again, preventing convergence to low value error. In contrast, Figure 6.9 (b) shows the progression of the weights and step-sizes for AMS-

Grad. Unlike Adam, AMSGrad's preconditioner term can only decrease in magnitude, and all of the step-sizes are quickly pushed to zero, preventing learning[1]. This demonstrates that although AMSGrad has better convergence guarantees than Adam, it may be ill-suited to continual online prediction problems, as unstable learning iterates can cause it to stop learning completely.

The results in this experiment suggest that both RMSGain and Adam are suitable for handling problems with unstable learning iterates. RMSGain's meta-objective appears to enable adapting the step-sizes in a way that stabilizes the learning iterates, whereas Adam's use of momentum and the RMSProp preconditioner seem to "smooth out" the instabilities while keeping the step-sizes above zero.

6.3 Conclusion

In this chapter, we performed two experiments in environments with stationary dynamics. The first experiment highlighted that meta-descent methods are not robust to the shape of the optimization surface. AdaGain's generic update scheme enabled us to overcome this issue by applying AdaGain to a preconditioned update vector. In particular, the combination of AdaGain and RMSProp — an algorithm we call RMSGain — was generally more robust to the optimization surface than previous meta-descent strategies in this experiment.

In our second experiment, we considered a Markov Decision Process in which TD methods diverge for any scalar step-size. Both RMSGain and Adam are able to converge to low value error in this task, while all other algorithms either diverge or learn nothing at all. The ability to stabilize the learning process using the step-sizes suggests that both RMSGain and Adam could be valuable algorithms in many problems of interest.

[1]In the original publication of our paper (Jacobsen *et al.* 2019a), the result for Adam was mistakenly reported as AMSGrad due to a bug in the code. This result was updated in the arXiv version of the paper (Jacobsen *et al.* 2019b) and the mistake noted in the errata.

Chapter 7

Experiments in Non-Stationary Settings

In this chapter, we conduct experiments in settings with non-stationary dynamics. The experiments in this chapter are aimed at understanding how well each algorithm can adapt in problems where the solution is changing over time. We begin with a simple LMS problem in which the dynamics change every so often and the optimal step-size can be computed analytically. Our experiments are concluded with a time-series prediction problem on real data from a mobile robot.

The first experiment investigates whether each of the algorithms is able to adapt the step-size in an optimal way. This experiment is additionally used to assess whether each of the algorithms is able to remain stable during a large number of learning iterates; the experiment is run for 3×10^9 iterations, corresponding to roughly 24 hours of operation. To achieve this an algorithm must be robust to the many sudden changes in the distribution of the target without becoming unstable.

In our final experiment, we compare the performance of our meta-descent strategy and a quasi second-order strategy when applied to real data generated from the sensor readings of a mobile robot. The task has many of the difficulties of real-world problems: the targets can be very high magnitude, the data is non-stationary, and the input is high-dimensional. The high-magnitude targets result in high-magnitude and high-variance updates, posing a threat to learning stability. The non-stationarity requires that the algorithms be able to

track the target as its distribution changes over time. The high-dimensional inputs result in significant computational expense, necessitating some compromise in terms of hyperparameter tuning.

7.1 Stateless Tracking

We begin with an experiment in a simple non-stationary domain, introduced by R. S. Sutton (1981). Consider a simple stateless tracking problem driven by two interacting Gaussians:

$$Y_t \stackrel{\text{def}}{=} Z_t + \mathcal{N}(0, \sigma^2_{Y,t})$$
$$Z_{t+1} \leftarrow Z_t + \mathcal{N}(0, \sigma^2_{Z,t}).$$

The agent only observes the sequence $(Y_1, Y_2, \ldots)$. The objective is minimize mean squared error (MSE) between a scalar prediction $P_t = w_t$ and the target $T_t = Y_{t+1}$. This problem is non-stationary because $\sigma_{Y,t}$ and $\sigma_{Z,t}$ change periodically, and the agent has no knowledge of the schedule. Since $\sigma_{Y,t}$ and $\sigma_{Z,t}$ govern how quickly the mean Z_t drifts and the sampling variance in Y_t, the agent must adjust its step-size accordingly: larger $\sigma_{Z,t}$ requires larger step-size, larger $\sigma_{Y,t}$ requires a smaller step-size (see Figure 7.1 for an illustrative example). The agent must continually change its scalar step-size value in order to

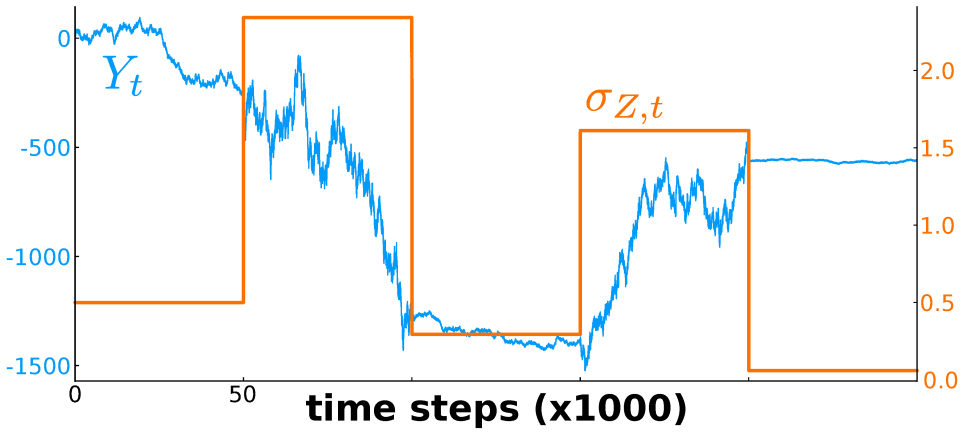

Figure 7.1: An illustrative example sequence of targets Y_t in the stateless tracking problem. In this example, $\sigma_{Z,t}$ was changed every $50,000$ steps, while $\sigma_{Y,t}$ remains fixed at 0.5. When $\sigma_{Z,t}$ is high the distribution of targets drifts at a faster rate, necessitating a greater degree of tracking by a learning algorithm.

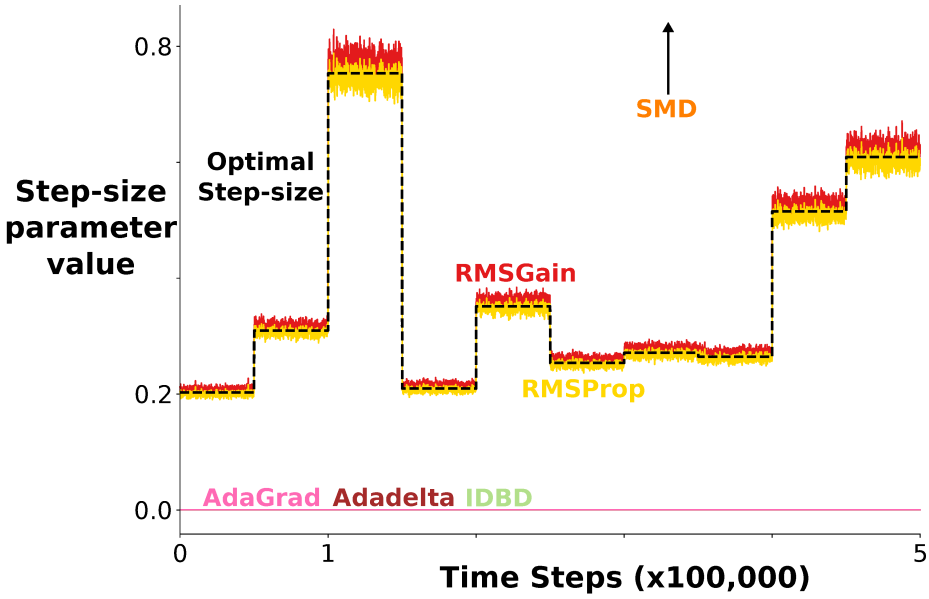

Figure 7.2: Step-sizes in the stateless tracking experiment. Depicted is the last 500,000 steps out of 3×10^9. The black dashed line shows the optimal step-size at each time step.

achieve low MSE. The optimal constant scalar step-size can be computed as $\alpha^* = \frac{-\sigma_Z^2 + \sqrt{\sigma_Z^4 + 4\sigma_Y^2 \sigma_Z^2}}{2\sigma_Y^2}$ in this simple domain (R. S. Sutton 1981), and is shown by the black dashed line in Figure 7.2.

We compared the step-sizes learned by several well-known quasi-second order methods (AdaGrad, RMSProp, AdaDelta) and three meta-descent strategies (RMSGain, SMD, IDBD). Each of RMSGain's three hyperparameters were tuned over the range $\{2^{-i} : i = 0, \ldots, 13\}$. The rest of the algorithms have either one or two hyperparameters, which were tuned over a linear spacing between the endpoints $[2^{-13}, 1]$, with the spacing chosen such that each algorithm was tuned over 2744 settings. The best performing instance of each algorithm was chosen according to the lowest MSE between the algorithm's step-size and the optimal step-size over $500,000$ steps of the experiment. The experiment was re-run for over 24 hours with the best instances of each algorithm to test the robustness of these methods in a long-running continual prediction task. The distribution of the target values was changed every $50,000$ steps by sampling $\sigma_{Z,t}$ uniform random from the range $[0, 2.5]$.

Figure 7.2 shows the step-sizes for each of the algorithms over the last

$500,000$ steps of the experiment, averaged over 10 runs. The optimal step-size is plotted as a black dashed line. RMSGain and RMSProp were the only algorithms to optimally adapt the step-size throughout the entirety of the experiment on all runs.

AdaGrad is unable to successfully solve this problem due to its decreasing step-sizes. This demonstrates how methods which decrease the step-sizes to zero over time can cause issues when an agent is faced with non-stationarity. Interestingly, AdaDelta is also unsuccessful at solving this problem. Recall from Section 4.1.2 that AdaDelta adds a unit correction term to the numerator of the RMSProp preconditioner. In this problem setting, the parameter updates are small on average, so the numerator term ends up pushing the AdaDelta's step-sizes down towards zero. Similarly, on every run of the experiment IDBD pushes its step-size to near-zero at some point, and is unable to bring it back up[1]. The largest number of steps IDBD performs before decaying the step-size to zero is $800,002,275$ steps — roughly a quarter of the way through the experiment. SMD gets destabilized and diverges to infinity on step $2,206,527,012$ of run 4 of the experiment, roughly three quarters through the experiment. The failure of both SMD and IDBD demonstrates the inherent difficulty of continual, long-running tasks with non-stationary dynamics, and the importance of optimizing for stability.

In addition to learning stability, sensitivity to parameter settings is also important. To help better understand these methods, we constructed a parameter sensitivity graph (Figure 7.3). For each algorithm, the plot shows a circle for each hyperparameter setting. y-axis shows the average squared error between the algorithm's step-size and the optimal step-size. A threshold is set on the y-axis; settings which yield performance above the threshold are grouped together at the top of the plot, and the percentage of parameter settings that are above the threshold are shown as a percentage at the top.

The plot shows that RMSGain has a number of settings that outperform

[1]In our previous work (Jacobsen *et al.* 2019a), IDBD was implemented with a lower bound $\boldsymbol{\beta} = \max(\boldsymbol{\beta}, -10)$ (see Section 4.2.2). In this work, we removed this lower bound for better consistency between the meta-descent method implementations.

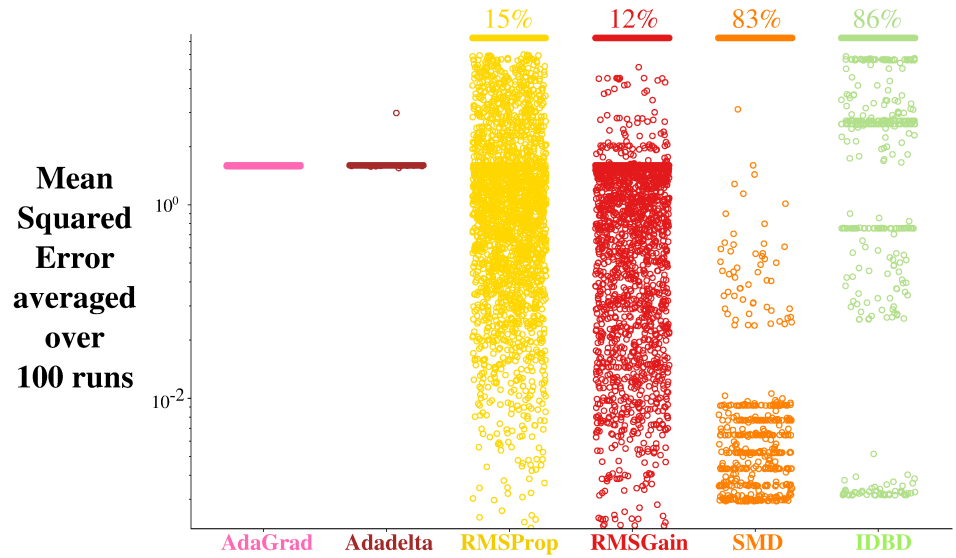

Figure 7.3: Parameter sensitivity plot for the first 500,000 steps of the stateless tracking problem. Each circle denotes the average MSE for a single parameter combination of an algorithm. The parameter combinations and respective performance are grouped in vertical columns for each method. The circles in each column are randomly offset within the column horizontally for a visualization of the clustering. Circles near the bottom of the plot represent low MSE. Circles arranged in a line in the top-most part of the plot are parameter combinations that either diverged or exceeded a minimum performance threshold, with the percentage of such parameter combinations given in the graph.

both IDBD and SMD, but that IDBD and SMD have a higher density of settings near their optimal performances. On the other hand, the majority of settings of both IDBD and SMD are above the threshold. RMSProp has similar sensitivity to RMSGain but has a higher density of parameter settings which do worse than AdaGrad (*i.e.* worse than decaying the step-sizes to zero). Many of the parameter combinations allowed RMSGain and RMSProp to achieve low error, suggesting a simple manual parameter tuning of these algorithms is more likely to achieve good performance on this problem, while the other methods may require more in-depth parameter sweeps.

The results in this experiment demonstrate that there are meta-descent methods and quasi second-order methods which are capable of optimally tracking a non-stationary target. Notably, classic meta-descent methods such as IDBD and SMD are capable of tracking the target, but may not be ideal for

long-running tasks; neither method was able to continue reliably performing the task for a large number of time steps. Furthermore, despite the simplicity of this task, both SMD and IDBD were very sensitive to their hyperparameters, having only a minority of parameter settings reach an acceptable level of performance. On the other hand, RMSProp and RMSGain both performed reliably during the long-running continual learning task, adapting the step-size effectively across all runs of the experiment without diverging or having the step-sizes become unreasonably small. Both RMSProp and RMSGain were additionally less sensitive to their hyperparameters than the rest of the algorithms.

7.2 Predicting Robot Sensor Readings

In our previous experiments, we focused mainly on simple domains in which the dynamics of the environment and the behavior of each of the algorithms could be well understood. While these domains were useful in terms of building understanding, they are simpler than the problem settings that we'd expect to see in the wild. Our final experiment on poses many of the problems we associate with online continual prediction problems. The problem uses the sensor readings from a real mobile robot; the data is noisy, and the magnitudes of the predictions can be very large — in the millions — making learning stability an issue. The input to the system is very high-dimensional, necessitating linear-time algorithms. Some of the sensor readings have a slow drift to them, requiring the ability to track carefully. Other sensors have regions in which the distribution of the inputs and targets changes suddenly, requiring the algorithms to aggressively revise their predictions to remain accurate.

The data was generated as a mobile robot interacted with its environment following a fixed behavior policy. Time is discretized into time steps of 100 milliseconds, and the readings from each of the robot's 53 sensors were recorded at each time step. The robot's environment was a square pen with an area of 4 squared meters and had a light source on one edge of the pen. The behavior policy was a fixed stochastic policy which caused the robot to traverse along

the edge of the pen in a loop, with the pen's wall at the robot's right side. The data was generated over $144,000$ time steps, corresponding to approximately 3.4 hours of runtime on the robot.

We used the same tile-coding (R. S. Sutton and Barto 2018) of the sensor inputs described in the original work, giving 6065 binary feature components for use as a linear representation. The tile-coding strategy consists of a mixture of joint and single tilings, where a tiling is a discretization of the output space of a group of sensors. For example, given two sensors $s^{(1)}$ and $s^{(2)}$, a tiling could be defined by a two-dimensional grid overlaid on the two-dimensional space of possible joint sensor readings. If the sensor readings $(s_t^{(1)}, s_t^{(2)})$ at time t fall into cell (i, j) of the grid, then the corresponding entry in the feature vector is set to 1. In particular, the tile-coding strategy in this experiment uses 457 such tilings, resulting in feature vectors with 457 active binary features on each time step.

We recreate the robotic Nexting experiment (Modayil *et al.* 2014), using TD(λ) to make dozens of predictions about the future values of robot sensor readings. We formulate each prediction as estimating the discounted sum of future sensor readings, treating each sensor as a reward signal with a discount factor of $\gamma = 0.9875$, corresponding to approximately 8-second predictions. We incrementally processed the sensor data on each step constructing a feature vector from the vector of sensor readings and making one prediction for each sensor. At the end of learning, the returns $G_t^{(s)}$ for each sensor s were computed, and we measured the symmetric mean absolute percentage error $\text{SMAPE}(T, s) \stackrel{\text{def}}{=} \frac{1}{T} \sum_{t=1}^{T} \frac{|\hat{v}^{(s)}(S_t) - G_t^{(s)}|}{|\hat{v}^{(s)}(S_t)| + |G_t^{(s)}|}$ between the prediction $\hat{v}^{(s)}(\cdot)$ on sensor s and the corresponding return $G_t^{(s)}$.

The learning curves were aggregated using the median. A small number of the sensors in this experiment lead to high-magnitude returns, resulting in update targets in the millions. the prediction error on these sensors is generally much higher than the rest of the sensors. Because of this, the median across sensors is a more representative measure of aggregate performance than the mean.

For this experiment we reduced the number of algorithms, using AMSGrad as the representative quasi second-order method and RMSGain as the representative meta-descent algorithm. Because of the computational cost of this experiment, the hyperparameters were tuned in two stages: first, all hyperparameters were pre-tuned on one of the robot's light sensors. The pre-tuning was used to identify reasonable values for each of the hyperparameters. Once the best setting was selected, the meta step-size η of each algorithm was re-tuned for performance across all sensors.

In the pre-tuning phase, AMSGrad's three hyperparameters were both pre-tuned over the values $\{2^{-i} : i = 0, \ldots, 12\}$. For RMSGain, we fixed the initial step-size $\boldsymbol{\alpha}_0 = \mathbf{1}$, and pre-tuned the decay parameter β and meta-step-size η over the values $\{1.4^{-i} : i = 1, \ldots, 45\}$. The light-sensor data was incrementally processed as described above, and the SMAPE was calculated. The best performing parameter setting was selected in terms of the minimum area under the learning curve. In the second phase of tuning, the meta step-size η of each algorithm was re-tuned for performance over all sensors. For each $\eta \in \{2.5^{-i} : i = 0, \ldots, 20\}$, the data for each sensor was incrementally processed, and the resulting learning curves were aggregated using the median. The best parameter setting was selected according to the minimum area under the median learning curve.

As a baseline, we additionally include the performance of an optimal static baseline, computed offline by solving a system of equations offline (as in Modayil *et al.* (2014)). The optimal solution makes use of only the first $40,000$ data points for each sensor, reflecting a realistic scenario of computing predictions from a limited batch of data, and later using the offline solution for online prediction.

The median SMAPE across all 53 sensors is shown in Figure 7.4 (left). As to be expected, the SMAPE for the offline optimal predictions is low on the training data (first $40,000$ time steps), and much higher on later data due to non-stationarity in the data. The learning curves also show that RMSGain and AMSGrad perform similarly in terms of aggregate error over all predictions, with RMSGain perfoming slightly better at the start of learning. Inspecting

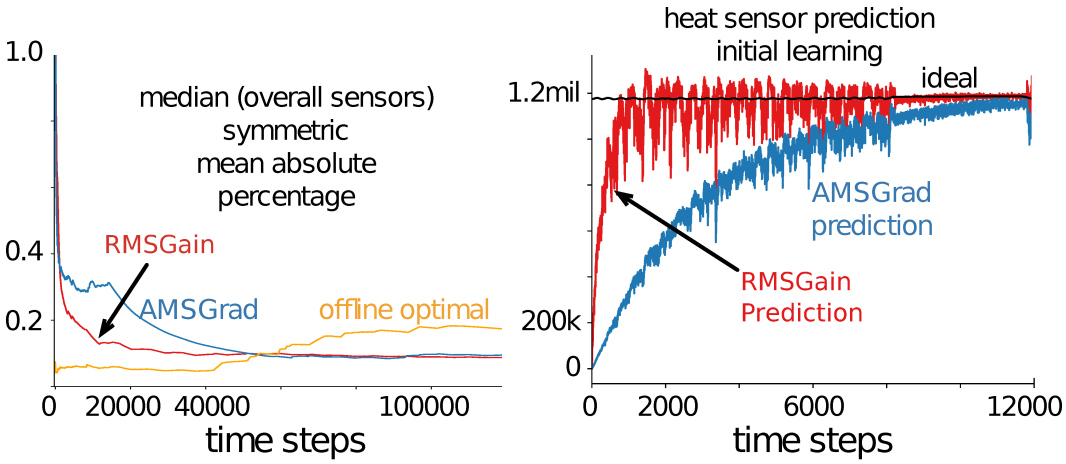

Figure 7.4: (left) The median symmetric mean absolute percentage error (SMAPE) across all 53 sensors, (right) plot of the predictions for the heat sensor verses the ideal prediction. The SMAPE provides a measure of prediction accuracy compared with the ideal predictions (always plotted in black). The ideal predictions are computed offline using all future data (as described in Modayil *et al.* (2014)), but the predictions are learned online and incrementally.

the predictions of one of the heat sensors (Figure 7.4 (right)) reveals a possible explanation. In early learning, RMSGain more quickly increases the prediction to near the ideal prediction, whereas AMSGrad more slowly reaches this point — over 12000 steps. This is particularly notable because the heat sensor targets are unnormalized, obtaining values over 1 million. RMSGain and AMSGrad then both track the ideal heat prediction similarly, and so obtain similar error for the remainder of learning.

The advantage in early learning is also demonstrated in Figure 7.5. The top plot shows the predictions for one of the light sensors during an unexpected event: the robot over-heated, causing it to stall directly in front of the light. This unexpected event is a case in which the distribution of the prediction targets shifts suddenly, requiring the agent to quickly revise its predictions. RMSGain adapts its predictions more quickly than AMSGrad in response to this event. The remaining plots show the predictions for the light sensor and a magnetic sensor during normal operation, and demonstrate that RMSGain and AMSGrad otherwise perform similarly on this task.

Overall, this experiment serves as a sanity check for RMSGain, validating

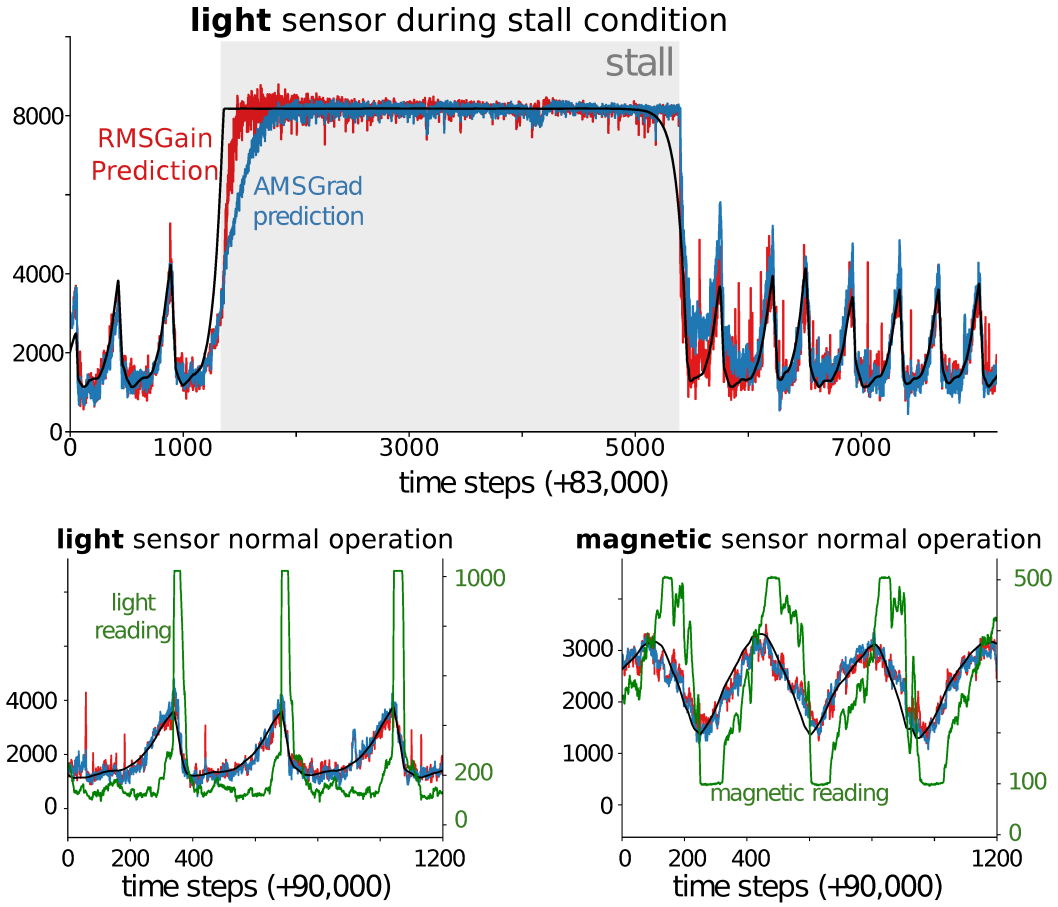

Figure 7.5: Three snapshots of the predictions learned by RMSGain and AMSGrad compared with the offline ideal predictions. Each of the three plots highlights a different part of the dataset to give an alternative perspective on the accuracy of the learned predictions. In the top plot we see a situation where the robot stalled unexpectedly directly in front of a bright light source, saturating the light sensor. Due to this sudden unpredictable event, the predictions of both RMSGain and AMSGrad became incorrect. RMSGain more quickly adapts learning to adjust its predictions to reflect the new reality, matching the ideal predictions (black line). Otherwise, these plots show that, in general, RMSGain and AMSGrad can track the ideal prediction similarily.

that RMSGain can scale to more realistic problems and remains stable in the face of high levels of noise and high-magnitude prediction targets. The results demonstrate that RMSGain is able to perform comparatively with a well-established quasi second-order method when applied to real data, and suggest that meta-descent methods can be as effective as the state-of-the-art quasi second-order methods in practice.

7.3 Conclusion

In this chapter, we performed two experiments in environments with non-stationary dynamics. In the first experiment, we performed a simple stateless tracking task in which the optimal step-size can be computed. While the meta-descent methods could optimally adapt the step-size parameter, only RMSGain was able to reliably perform the task for a large number of iterations without issue. The experiment also suggests that the meta-descent methods IDBD and SMD can be sensitive to their hyperparameters compared to RMSGain, which had sensitivity similar to that of the best-performing quasi second-order method RMSProp.

In our second experiment, we compared the performance of a meta-descent method and a quasi second-order method making predictions about the future sensor readings of a mobile robot. The results suggest that RMSGain can scale up to real-world problems in which the prediction targets are high-magnitude and high-variance, and can perform comparatively with a well-established quasi second-order method on problems involving real data.

Chapter 8

Conclusion and Future Work

In this work, we proposed a new general meta-descent strategy to adapt a vector of step-sizes for continual, online prediction problems. We defined a new meta-descent objective which enables a broader class of incremental updates for the base learner, generalizing beyond work specialized to least-mean squares, temporal difference learning, and vanilla stochastic gradient descent updates. We derived a recursive update for the step-sizes and provide a linear-complexity approximation.

In our first experiment, we highlighted that meta-descent strategies can be sensitive to the shape of the optimization surface. The ability to use AdaGain for generic updates enabled us to overcome this issue by layering AdaGain on RMSProp, a simple quasi-second order approach. We then showed that the combination of meta-descent and quasi second-order methods can perform better than either method alone. Further experiments demonstrated that the combination of AdaGain with RMSProp can have benefits in terms of learning stability, and was less sensitive to the its hyperparameters than the competing meta-descent and quasi second-order approaches.

The primary next steps are to better understand the generality of the method and characterize theoretical properties. While our experiments do provide some empirical success, it is unclear what possible regret bounds — if any — this algorithm could have. The simplest next-step would be to analyze the regret bounds of AdaGain in the online convex optimization setting, so that the bounds can be compared more directly with those of AdaGrad and

AMSGrad. Furthermore, the literature on measures of regret in non-stationary settings is lacking; none of the methods discussed in this thesis have proven regret bounds in the non-stationary setting. Yet, given the ubiquitous nature of non-stationarity in real-world online prediction problems, investigating regret bounds in this setting is a subject of interest for future work.

In this work, our main focus was on linear function approximation; we did not explore the application of AdaGain to nonlinear approximators such as neural networks. It remains to be seen what advantages optimizing for stability in neural networks could lead to. Preliminary experiments in this regard indicate that AdaGain can work well when applied to neural networks, but the benefits over using existing algorithms — such as AMSGrad, for example — are not clear. One possible direction for future work would thus be to perform an in-depth empirical analysis of the performance of meta-descent methods such as AdaGain when applied to neural networks.

All of the adaptive methods studied in this thesis introduce one or more additional hyperparameters. While our experiments suggest that some methods are less sensitive to these hyperparameters than others, they all nonetheless required thorough tuning to attain good performance. Designing performant parameter-free learning algorithms is a subject of interest in nearly all disciplines of machine learning, and is an exciting direction for future work.

Meta-descent has recently been used to adapt hyperparameters other than the step-size, such as the discount rate γ and the bootstrapping parameter λ in deep reinforcement learning (Xu *et al.* 2018). A possible direction for future work is to investigate whether these parameters could also be leveraged for learning stability, using an objective analogous to the objective used by AdaGain.

Finally, we note that AdaGain is only a single, simple instance of the idea of optimizing for stability in online prediction problems. An interesting direction for future work would be to revisit the analyses that lead to the popular quasi second-order methods, and attempt to account for stability. For example, Gupta *et al.* (2017) introduce a unified framework for adaptive regularization in online learning, in which a preconditioning ma-

trix $\mathbf{H}_t$ is selected at each time-step, and updates are made according to $w_{t+1} = w_t - \mathbf{H}_t \nabla_{\mathbf{w}} \ell(\mathbf{w}_t)$. The preconditioning matrix is selected by solving $\mathbf{H}_t = \min_{\mathbf{H}>0} \left\{ \sum_{t=1}^{T} \|\nabla_{\mathbf{x}} \ell_t(\mathbf{x})\|_{\mathbf{H}}^2 + \Psi(\mathbf{H}) \right\}$, where ℓ_t is the loss function revealed at iteration t, and Ψ is a parameter of the framework called the *potential function*. Selecting $\Psi(\mathbf{H}) \stackrel{\text{def}}{=} \text{trace}(\mathbf{H}^{-1})$ leads to the AdaGrad algorithm, for example. Investigating potential functions which emphasize the stability of learning updates could potentially be an interesting direction for exploring adaptive learning algorithms for continual, online prediction problems.

References

[1] L. B. Almeida, T. Langlois, J. D. Amaral, and A. Plakhov, "On-line learning in neural networks," in, 1998, ch. Parameter Adaptation in Stochastic Optimization. 2, 24

[2] L. Baird, "Residual algorithms: reinforcement learning with function approximation," in, Elsevier, 1995. 51

[3] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online Learning Rate Adaptation with Hypergradient Descent," in *International Conference on Learning Representations*, 2018. 24, 25, 37

[4] A. Benveniste, M. Metivier, and P. Priouret, *Adaptive Algorithms and Stochastic Approximations*. Springer, 1990. 31

[5] A. Bordes, L. Bottou, and P. Gallinari, "SGD-QN: Careful quasi-Newton stochastic gradient descent.," *Journal of Machine Learning Research*, 2009. 1, 20

[6] W. C. Dabney, "Adaptive Step-sizes for Reinforcement Learning," PhD thesis, University of Massachusetts - Amherst, 2014. 2

[7] W. Dabney and A. G. Barto, "Adaptive step-size for online temporal difference learning.," in *AAAI*, 2012. 2

[8] Y. N. Dauphin, H. de Vries, and Y. Bengio, "Equilibrated adaptive learning rates for non-convex optimization," *arXiv:1502.04390*, 2015. 23

[9] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 2011. 2, 20

[10] V. Gupta, T. Koren, and Y. Singer, "A unified approach to adaptive regularization in online and stochastic optimization," *arXiv:1706.06569*, 2017. 68

[11] E. Hazan, A. Agarwal, and S. Kale, "Logarithmic regret algorithms for online convex optimization," *Machine Learning*, 2007. 23

[12] R. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, 1988. 2, 24

[13] A. Jacobsen, M. Schlegel, C. Linke, T. Degris, A. White, and M. White, "Meta-descent for online, continual prediction," in *AAAI Conference on Artificial Intelligence*, 2019. iii, 55, 59

[14] ——, "Meta-descent for online, continual prediction," *arXiv:1907.07751*, 2019. 55

[15] H. Jaeger, "Observable Operator Processes and Conditioned Continuation Representations," *Neural Computation*, 2000. 1

[16] A. Kearney, V. Veeriah, J. B. Travnik, R. S. Sutton, and P. M. Pilarski, "Tidbd: adapting temporal-difference step-sizes through stochastic meta-descent," *arXiv:1804.03334*, 2018. 2, 29

[17] A. Kearney, V. Veeriah, J. Travnik, P. M. Pilarski, and R. S. Sutton, "Learning feature relevance through step size adaptation in temporal-difference learning," *arXiv:1903.03252*, 2019. 2, 30

[18] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Machine Learning*, 2015. 2, 16, 22, 23

[19] M. L. Littman, R. S. Sutton, and S. Singh, "Predictive representations of state," in *Advances in Neural Information Processing Systems*, 2001. 1

[20] H. R. Maei, "Gradient temporal-difference learning algorithms," 2011. 51

[21] A. R. Mahmood, R. S. Sutton, T. Degris, and P. M. Pilarski, "Tuning-free step-size adaptation," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2012. 2, 30

[22] A. Mahmood, "Automatic step-size adaptation in incremental supervised learning," 2010. 2, 25

[23] J. Martens, "New insights and perspectives on the natural gradient method," *arXiv:1412.1193*, 2014. 23

[24] H. B. McMahan and M. Streeter, "Adaptive Bound Optimization for Online Convex Optimization," in *International Conference on Learning Representations*, 2010. 2, 20

[25] D. Meyer, R. Degenne, A. Omrane, and H. Shen, "Accelerated gradient temporal difference learning algorithms," in *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2014. 2

[26] J. Modayil, A. White, and R. S. Sutton, "Multi-timescale nexting in a reinforcement learning robot," *Adaptive Behavior*, 2014. 62–64

[27] Y. Pan, E. S. Azer, and M. White, "Effective sketching methods for value function approximation.," in *Conference on Uncertainty in Artificial Intelligence*, 2017. 2

[28] Y. Pan, A. White, and M. White, "Accelerated Gradient Temporal Difference Learning," in *International Conference on Machine Learning*, 2017. 2

[29] R. Pascanu and Y. Bengio, "Revisiting Natural Gradient for Deep Networks," *arXiv:1301.3584*, 2013. 23

[30] B. A. Pearlmutter, "Fast exact multiplication by the hessian," *Neural computation*, 1994. 35

[31] S. J. Reddi, S. Kale, and S. Kumar, "On the Convergence of Adam and Beyond," in *International Conference on Learning Representations*, 2018. 2, 16, 24

[32] T. Schaul, S. Zhang, and Y. LeCun, "No More Pesky Learning Rates," in *International Conference on Artificial Intelligence and Statistics*, 2013. 2

[33] N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent," *International Conference on Artificial Neural Networks*, 1999. 2, 3, 25, 28, 37

[34] N. Schraudolph, J. Yu, and S. Günter, "A stochastic quasi-Newton method for online convex optimization," in *International Conference on Artificial Intelligence and Statistics*, 2007. 2

[35] N. Schraudolph, "Online local gain adaption for multi-layer perceptions," *Technical report/IDSIA*, 1998. 25

[36] J. C. Spall, "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation," *IEEE Transactions on Automatic Control*, 1992. 38

[37] R. S. Sutton, *"adaptation of learning rate parameters." in: goal seeking components for adaptive intelligence: an initial assessment*, 1981. 57, 58

[38] ——, "Adapting bias by gradient descent: An incremental version of delta-bar-delta," in *AAAI Conference on Artificial Intelligence*, 1992. 2, 25, 28, 29, 34

[39] ——, "Gain Adaptation Beats Least Squares?" In *Seventh Yale Workshop on Adaptive and Learning Systems*, 1992. 2, 25

[40] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. MIT Press, 2018. 62

[41] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. Pilarski, A. White, and D. Precup, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *International Conference on Autonomous Agents and Multiagent Systems*, 2011. 1

[42] R. S. Sutton and B. Tanner, "Temporal-Difference Networks," in *Advances in Neural Information Processing Systems*, 2004. 1

[43] R. Sutton, A. Koop, and D. Silver, "On the role of tracking in stationary environments," in *International Conference on Machine Learning*, 2007. 31

[44] T. Tieleman and G. Hinton, "RmsProp: Divide the gradient by a running average of its recent magnitude," in *COURSERA Neural Networks for Machine Learning*, 2012. 2, 16, 22

[45] A. White, "Developing a predictive approach to knowledge," 2015. 50

[46] S. Wright and J. Nocedal, "Numerical optimization," *Springer Science*, 1999. 7, 15, 16

[47] Y. Wu, M. Ren, R. Liao, and R. B. Grosse, "Understanding Short-Horizon Bias in Stochastic Meta-Optimization.," in *International Conference on Learning Representations*, 2018. 3, 28

[48] Z. Xu, H. P. van Hasselt, and D. Silver, "Meta-Gradient Reinforcement Learning," in *Neural Information Processing Systems*, 2018. 68

[49] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *arXiv: 1411.4000v2*, 2012. 2, 16, 22