Université de Montréal

Model-Based Hyperparameter Optimization

par

Paul Crouther

Department of Computer Science and Operations Research
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Computer Science

April 30, 2023

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Model-Based Hyperparameter Optimization

présenté par

Paul Crouther

a été évalué par un jury composé des personnes suivantes :

Glen Berseth

(président-rapporteur)

Pierre-Luc Bacon

(directeur de recherche)

Irina Rish

(membre du jury)

Résumé

L'objectif principal de ce travail est de proposer une méthodologie de découverte des hyperparamètres. Les hyperparamètres aident les systèmes à converger lorsqu'ils sont bien réglés et fabriqués à la main. Cependant, à cette fin, des hyperparamètres mal choisis laissent les praticiens dans l'incertitude, entre soucis de mise en œuvre ou mauvais choix d'hyperparamètre et de configuration du système. Nous analysons spécifiquement le choix du taux d'apprentissage dans la descente de gradient stochastique (SGD), un algorithme populaire. Comme objectif secondaire, nous tentons de découvrir des points fixes en utilisant le lissage du paysage des pertes en exploitant des hypothèses sur sa distribution pour améliorer la règle de mise à jour dans SGD. Il a été démontré que le lissage du paysage des pertes rend la convergence possible dans les systèmes à grande échelle et les problèmes difficiles d'optimisation de la boîte noire. Cependant, nous utilisons des gradients de valeur stochastiques (SVG) pour lisser le paysage des pertes en apprenant un modèle de substitution, puis rétropropager à travers ce modèle pour découvrir des points fixes sur la tâche réelle que SGD essaie de résoudre. De plus, nous construisons un environnement de gym pour tester des algorithmes sans modèle, tels que Proximal Policy Optimization (PPO) en tant qu'optimiseur d'hyperparamètres pour SGD. Pour les tâches, nous nous concentrons sur un problème de jouet et analysons la convergence de SGD sur MNIST en utilisant des méthodes d'apprentissage par renforcement sans modèle et basées sur un modèle pour le contrôle. Le modèle est appris à partir des paramètres du véritable optimiseur et utilisé spécifiquement pour les taux d'apprentissage plutôt que pour la prédiction. Dans les expériences, nous effectuons dans un cadre en ligne et hors ligne. Dans le cadre en ligne, nous apprenons un modèle de substitution aux côtés du véritable optimiseur, où les hyperparamètres sont réglés en temps réel pour le véritable optimiseur. Dans le cadre hors ligne, nous montrons qu'il y a plus de potentiel dans la méthodologie d'apprentissage basée sur un modèle que dans la configuration sans modèle en raison de ce modèle de substitution qui lisse le paysage des pertes et crée des gradients plus utiles lors de la rétropropagation.

mots clés: apprentissage par renforcement, optimisation des hyperparamètres, contrôle optimal, l'apprentissage en profondeur, méta-apprentissage, apprentissage par renforcement basé sur un modèle, optimisation à deux niveaux

Abstract

The primary goal of this work is to propose a methodology for discovering hyperparameters. Hyperparameters aid systems in convergence when well-tuned and handcrafted. However, to this end, poorly chosen hyperparameters leave practitioners in limbo, between concerns with implementation or improper choice in hyperparameter and system configuration. We specifically analyze the choice of learning rate in stochastic gradient descent (SGD), a popular algorithm. As a secondary goal, we attempt the discovery of fixed points using smoothing of the loss landscape by exploiting assumptions about its distribution to improve the update rule in SGD. Smoothing of the loss landscape has been shown to make convergence possible in large-scale systems and difficult black-box optimization problems. However, we use stochastic value gradients (SVG) to smooth the loss landscape by learning a surrogate model and then backpropagate through this model to discover fixed points on the real task SGD is trying to solve. Additionally, we construct a gym environment for testing model-free algorithms, such as Proximal Policy Optimization (PPO) as a hyperparameter optimizer for SGD. For tasks, we focus on a toy problem and analyze the convergence of SGD on MNIST using model-free and model-based reinforcement learning methods for control. The model is learned from the parameters of the true optimizer and used specifically for learning rates rather than for prediction. In experiments, we perform in an online and offline setting. In the online setting, we learn a surrogate model alongside the true optimizer, where hyperparameters are tuned in real-time for the true optimizer. In the offline setting, we show that there is more potential in the model-based learning methodology than in the model-free configuration due to this surrogate model that smooths out the loss landscape and makes for more helpful gradients during backpropagation.

Keywords: reinforcement learning, hyperparameter optimization, optimal control, deep learning, meta-learning, model-based reinforcement learning, bilevel optimization

Contents

Résumé 5

Abstract 7

List of tables 13

List of figures 15

List of abbreviations 17

Thanks 19

Introduction 21

Chapter 1. Background 23

1.1. Stochastic approximation 23
1.1.1. Empirical Risk Minimization 24

1.2. Optimization 25
1.2.1. The Heavy Ball Method 25
1.2.2. Errors with momentum 26
1.2.3. Nonlinear models and implicit regularization in deep networks 26
1.2.4. Unrolling 27
1.2.5. Forward mode and unrolling 28
1.2.6. Backprop in neural networks 31
1.2.7. TBPTT and Unrolling SGD with an optimizer 32

1.3. Optimal Control 33
1.3.1. Optimal Control Problems 33
1.3.2. Dynamic Programming and Optimal Control 34
1.3.3. Analytic Policy Gradient in Optimal Control 34
1.3.4. Forward propagation in Optimal Control 35
1.3.5. Lagrangian Method for Adjoint Equations 35

1.4. Reinforcement learning 37
1.4.1. Policy Optimization or Dynamic Programming 37
1.4.2. Stochastic optimization and Policy Search 37
1.4.3. Gradient Estimation 38
1.4.4. Concluding thoughts 40

1.5. Hyperparameter optimization 40
1.5.1. Hyperparameter optimization in a model-free setting 41
1.5.2. Model-based hyperparameter optimization 41
1.5.3. Differentiating through a model 42

Chapter 2. Model-free RL hyperparameter optimization 43

2.1. Learning to optimize with RL 43

2.2. Hyperparameter optimization with PPO 44
2.2.1. Gym Environment with optimizer 45
2.2.2. PPO method as a hyperparameter optimizer 45

Chapter 3. Model-based RL hyperparameter optimization 47

3.1. Model-Based Policy Search 48

3.2. Differentiating through the learned model 48
3.2.1. System Identification (SysID) 49
3.2.2. Model learning 51

3.3. Offline Dyna-style Hyperparameter Optimization 52

3.4. Online Hyperparameter Optimization SVG 52
3.4.1. Bandit approach 52
3.4.2. Adaptive 52

Chapter 4. Results and comparisons 53

4.1. Initial experiments 53

4.2. Model-free hyperparameter optimization with PPO 54
4.2.1. MNIST Example 54

4.3. Model-based hyperparameter optimization with SVG 56
4.3.1. SysID Environment 57
4.3.2. Online HyperSVG 58

4.3.3. Bayesian Optimization Baseline 59
4.3.4. Comparisons 59

Chapter 5. Conclusion 67

References 68

List of tables

4.1 Episodic comparison with methods 60

4.2 Online best-chosen comparison of last accuracy and last hyperparameter with methods (with good learning rate ranges) 60

4.3 Environment settings 60

4.4 The tabular results corresponding with figure 4.13 and 4.14, where each line in the plot is represented. 64

List of figures

1.1 Standard loop: 28

1.2 Fully unrolled(flattened) loop: 28

1.3 Loop unrolled with K = 2: 28

4.1 Example trajectories with imitation optimizer, showing rollouts from different initializations on the toy task. These trajectories indicate that it is possible to use a learned optimizer to learn the dynamics for a given task, where the learned optimizer is trained in the model-based reinforcement learning fashion for hyperparameter optimization. This diverges from previous literature, focused on model-free reinforcement learning for hyperparameter optimization. 55

4.2 Sparse vs dense reward experiment with PPO 56

4.3 200 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task, showing classification validation accuracy (highest is 1.0) 57

4.4 200 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task (learning rates) 57

4.5 100 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task, showing classification validation accuracy (highest is 1.0) 58

4.6 100 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task (learning rates) 58

4.7 100 iterations with a length of 20 trajectories with Bayesian Optimization (BO) baseline with found learning rate on the MNIST task, showing classification validation accuracy (highest is 1.0) 59

4.8 100 iterations with length of 10, 20, unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0) 61

4.9 100 iterations with length of 50 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0) 62

4.10 100 iterations with length of adaptive SVG with MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0) 62

4.11 Different runtimes (100 iterations) of MNIST online task with different algorithms (in seconds). Our methods are shown in red, blue, purple, gray 63

4.12 100 iterations with length of 10, 20 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the bandit version can adapt the badly chosen parameters in grid search into more competitive performance. 65

4.13 100 iterations with length of 50 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the bandit version can adapt the badly chosen parameters in grid search into more competitive performance. 66

4.14 100 iterations with length of adaptive SVG with MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the online version can adapt some of the badly chosen parameters in grid search into more competitive performance. 66

List of abbreviations

SGD	Stochastic gradient descent
MC	Monte Carlo
MSE	Mean Squared Error
RL	Reinforcement learning
LQR	Linear Quadratic Regulator (linear dynamics, quadratic cost)
OCP	Optimal Control Problem
ECP	Equality Constrained Problem
PPO	Proximal Policy Optimization
TRPO	Trust Region Policy Optimization
CEM	Cross-Entropy Method

CMA-ES	Covariance Matrix Adaptation- Evolutionary Strategies
BPTT	Back Propagation Through Time
TBPTT	Truncated Backpropagation Through Time
SVG	Stochastic Value Gradients

Thanks

I would like to thank Pierre-Luc Bacon, for his advising and additionally his family for inviting us over for great food and discussion. It has been a great journey to learn and grow as a researcher under your leadership. To Michel, Evgenii, David K. David Yu-Tung, Aneri, Padideh, Tianwei, Anushree, and Niki, Mahan, Ryan, Pierluca, and Simon thanks for all of the help, it could not have been possible without your support. Last, but certainly not least, I thank my partner, Elizabeth for her support and encouragement. To anyone I left out, I appreciated our time working on this together.

Introduction

In the past, machine learning research has focused on the handcrafted design of features, such as Scale Invariant Feature Transforms (SIFT) [Geng and Jiang, 2009] and Histogram of Oriented Gradients (HOG) [Dalal and Triggs, 2005] as feature descriptors. Engineering these features was difficult, not only in obtaining good accuracy on tasks such as recognition, but also the engineering process itself, for example, handcrafting of new of features for different objects. To remedy this, the focus naturally shifted to learning features within standard MLPs such as AlexNet [Krizhevsky et al., 2017] and LeNet [LeCun et al., 1998] that, when given enough data, easily outperformed well-engineered features. The refocus to learning features also brought new challenges, such as a further emphasis on hyperparameter tuning. In essence, this trades off a fraction of the difficulty of designing features on a specific subtask, such as learning the edges of a cat or dog, for the engineering of an architecture and a process, such as determining how quickly to update your neural network parameters for object recognition.

Another explanation for the shift from engineered features to learned features is that the performance can be more reliably determined from an engineered architecture instead of engineered features. An engineered architecture as a model has the capacity to represent the many functions that are meant to approximate the decision boundaries of a particular subtask, better than the heuristics applied to determine these decision boundaries directly as a model.

However, it is still difficult to tune these architectures, leading to underperformance. Usually, this underperformance is not due to a lack of capacity, but due to sensitivity to certain small architectural and heuristic changes in hyperparameters. Another confounder is inadequately judging the performance of the model or architecture on the wrong distribution of the data by training on the wrong dataset.

To this end, the focus has once again shifted from developing with primarily architectural and heuristic considerations to learning how to learn engineering architectures. Learning to learn differentiates itself from engineering to learn by introducing a context over distributions in data, or by using multiple datasets in the optimization and training of the architecture's parameters.

With gradient-based hyperparameter optimization approaches, there have been advances utilizing automatic differentiation, such as forward mode, reverse mode, and implicit differentiation. The advances help deliver reasonable hyperparameters, but an underlying problem still exists with the gradient descent and approaches that nest gradients [Goodfellow et al., 2016]. Issues with scalability due to long-term dependencies also persist [Pascanu et al., 2013], and are compounded when using forward and backward propagation through a neural network. Erratic dynamics result from large learning rates [Maclaurin et al., 2015a] and give uninformative gradient information or exploding gradients, whereas small learning rates can fail to converge reasonably quickly. This can be classified as unrolled optimization or truncated unrolled optimization like truncated backpropagation through time(BPTT), TBPTT, which, while improving in efficiency over unrolled optimization, suffers from truncation bias.

Some methods, such as Self-Tuning Networks [MacKay et al., 2019] and implicit hyperparameter optimization [Lorraine et al., 2020]seek to bypass some of the issues with gradients with long term dependencies by trying to approximate the Jacobian through various means such as utilizing the "best-response" Jacobian local approximation in STN, and by utilizing the implicit function theorem(finding a fixed point where the gradient zero through some root finding method).

Recent methods have sought to treat the hyperparameter optimization problem as a black-box optimization problem, such as Persistent Evolutionary Strategies (PES) [Vicol et al., 2021], doing a form of unrolled optimization but using gradient estimates. These methods have seen a lot of success and is a piece of the framework this thesis follows, using unrolled optimization with gradient estimates.

Another axis for modification to improve performance while reducing the engineering difficulty is to investigate the architecture itself and the methodology in which it is learned. Borrowing from the model-based reinforcement learning framework, the model can be used to predict the next states of the world with respect to a current state by unrolling, which in the context of learning to learn is predicting what the architecture could look and function like. If this model is untrustworthy, the next state architecture should also be untrustworthy and should not be used to represent the future performance of a distribution of data. Moreover, if a model is trustworthy, it can then be used to represent future performance on a distribution of data, alleviating some of the difficulty in tuning.

The work in this thesis directly utilizes the model to obtain hyperparameters on a distribution of data, where the model that is learned is a transition model as a surrogate optimizer of the true optimizer's dynamics from a distribution of data over different contexts.

Chapter 1

Background

1.1. Stochastic approximation

In machine learning, since [Robbins and Monro, 1951] many have sought to understand convergence with stochastic gradient descent (SGD). Imagine one would like to find the root of some function f, with samples from some distribution. In [Robbins and Monro, 1951], it would be modeled as a *stochastic approximation* with some noise as a random variable ξ, represented as $g(\mathbf{x}, \xi)$, which one can call the *stochastic gradient.* We would like to compute the true gradient $\nabla f(\mathbf{x})$, assuming we have information about f, we assume $g(\mathbf{x}, \xi)$ is an unbiased estimate of $\nabla f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = \mathbb{E}_{\xi}[g(\mathbf{x}, \xi)] \tag{1.1.1}$$

Therefore, preceding this definition with an expectation over ξ with probability P, one can substitute this estimate for the true gradient $\nabla f(x)$ within an update rule, with $x_t \sim D$ and $\epsilon \sim P$ and α_t is a step size, where there is an assumption on the smoothness (continuous gradients) of f, but it is non-convex:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t g(\mathbf{x}_t, \xi_t) \tag{1.1.2}$$

One could also incrementally compute this gradient using an incremental gradient method (Wright), where f is a finite-sum $f(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{x})$. Knowing that N is large, computing the full gradient $\nabla f(\mathbf{x})$ is expensive, so a procedure is performed:

$$\mathbb{E}_{\xi}(g(\mathbf{x}, \xi)) = \nabla\left(\frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{x})\right) = \frac{1}{N}\sum_{i=1}^{N}\nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}) \tag{1.1.3}$$

In addition to this, one can apply the stochastic gradient to the finite-sum objective 1.1.3, and collect a *minibatch* of samples, for example, of size m such that there is a subset of samples at every iteration S_t, chosen uniform random:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \frac{1}{m} \sum_{i \in S_t} \nabla f_i(\mathbf{x}_t) \tag{1.1.4}$$

This leads to having a lower variance estimate of $\nabla f(\mathbf{x}_t)$ compared to $\nabla f_i(\mathbf{x}_t)$ in 1.1.3 overall, but trades this off for being m times more expensive in computation. In the next section, the focus shifts to empirical risk minimization, and then a discussion on dynamics in SGD with step size and minibatch size.

1.1.1. Empirical Risk Minimization

In the optimization of machine learning, tasks can be evaluated as expected values of error with data drawn from some distribution or simulator. Given some data drawn from some probability distribution $\mathbf{x} \sim p(D)$ and a loss function $l(\mathbf{x},\mathbf{y})$, the *true risk* R is defined as:

$$R(f) = \mathbb{E}_{(\mathbf{x},\mathbf{y}) \sim p(D)} \left[L(f(\mathbf{x}), \mathbf{y}) \right]$$

Given that it is intractable to minimize the risk function, instead, we opt for data drawn independently and identically distributed (i.i.d.) from a joint distribution $(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_N, \mathbf{y}_N) \sim p(\mathbf{x}, \mathbf{y})$ such now it becomes the empirical risk:

$$R_{emp}(f) = \frac{1}{N} \sum_{i=1}^{N} l(f(\mathbf{x}_i), \mathbf{y}_i)$$

Such that, in expectation with respect to the sample set $S = (\mathbf{x}_{1:N}, \mathbf{y}_{1:N})$ drawn from data D, the empirical risk is close to the true risk given that the empirical risk is a random variable. This is what supervised learning intends to maximize, as the performance on given data given an adjustment of hypothesis f. These two risks subtracted from each other constitute the generalization gap, $(R[f_S] - R_{emp}[f_S])$, where f_S where f is a hypothesis learned on samples S with a loss l. Finally, a critical observation would be that given we want to minimize the true risk $R[f]$, can look at the following [Hardt and Recht, 2022]:

$$R[f] = R_{emp}[f] + (R[f] - R_{emp}[f])$$

This relation between true risk and empirical risk shows the generalization gap $(R[f] - R_{emp}[f])$ because in order to minimize the true risk, one must also minimize this generalization gap. In the following section, we illustrate methods to do optimization with respect to the objective $f(\mathbf{x})$, in a nonlinear equality-constrained optimal control setting. In future sections, there will be a discussion on stochastic optimization and policy search, which will revisit some of these topics. To motivate the outcomes of this work, the discussion shifts to gradient methods and momentum as a way to accelerate toward a solution using the memory of gradients from the past.

1.2. Optimization

1.2.1. The Heavy Ball Method

In gradient methods, a previous iteration has information that goes unused, such that it is basically a "memoryless" method. However, one could consider a multistep method, which in Polyak's words, takes into account the "prehistory" [Rivet and Souloumiac, 1987] [Polyak, 1983] of the process, or some sort of memory to be able to aid convergence. In this case, he describes a method that makes an approximation by taking into consideration k previous iterations:

$$\mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k+1}) \tag{1.2.1}$$

Called a k-step method, where Newton's method and the gradient descent method were single-step methods. The heavy ball method, described in [Polyak, 1964], or Polyak's momentum, introduces an inertia term $\beta(x_t - x_{t-1})$:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t) + \beta(\mathbf{x}_t - \mathbf{x}_{t-1}) \tag{1.2.2}$$

Which shows clearly that if $\beta = 0$, it is just the gradient method. Where starting with dynamics $\frac{d\mathbf{x}}{dt} = -\nabla f(\mathbf{x})$, the first fixed point occurs at $\nabla f(x) = 0$. To further illustrate, describe the motion of a heavy ball in a potential field under viscosity using a second-order ODE, where $\mu \geq 0$ is the particle mass and $p \geq 0$ is the friction during the system evolution:

$$\mu \frac{d^2\mathbf{x}(t)}{dt^2} = -\nabla f(\mathbf{x}(t)) - p\frac{d\mathbf{x}(t)}{dt}$$

Using a finite-difference approximation, such as the following:

$$\mu \frac{\mathbf{x}(t+\Delta t) - 2\mathbf{x}(t) + \mathbf{x}(t-\Delta t)}{(\Delta t)^2} \approx -\nabla f(\mathbf{x}(t)) - p\frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t)}{\Delta t}$$

Then, after some term rearrangement and using α and β:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) - \alpha\nabla f(\mathbf{x}(t)) + \beta(\mathbf{x}(t) - \mathbf{x}(t-\Delta t))$$

Revealing Polyak's momentum in dynamics form. Polyak's momentum uniquely balances the curvature dynamics h with the update rule for convergence with step size α. Importantly, it can be shown that for the values α_t, β_t, are set based on the bounds of the condition number of the Hessian ($\mathbf{H}(x) = \nabla^2 f(\mathbf{x})$) of f, which bounds the entire landscape of f. Finding the condition number κ for a symmetric matrix $\mathbf{A}$:

$$\kappa = \frac{|\lambda_{max}(\mathbf{A})|}{|\lambda_{min}(\mathbf{A})|} \tag{1.2.3}$$

Nesterov's method [Nesterov, 1983] (also known as Nesterov's accelerated gradient method) can be shown to be the following update rule where $(\mathbf{x}_t - \mathbf{x}_{t-1})$ is the momentum term:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t + \beta(\mathbf{x}_t - \mathbf{x}_{t-1})) + \beta(\mathbf{x}_t - \mathbf{x}_{t-1}) \tag{1.2.4}$$

Nesterov's accelerated gradient helps the ball slow before the hill slopes back up again, giving an approximation of the next set of parameters, approximating the future of the position of parameters. It has been shown that compared to vanilla gradient descent, which has a convergence of $O(1/t)$ [Rockafellar, 1970], Nesterov's has a convergence of $O(1/t^2)$ [Ghadimi et al., 2015]. Importantly, this shows that accelerated methods help speed up convergence. However, this convergence benefit trades off with the increases in stochastic errors [Hardt et al., 2016, Ramezani-Kebrya et al., 2018].

1.2.2. Errors with momentum

Underdamping, when β is too large, leads to oscillation, which can be seen in the mass/spring example. Overdamping, such as when β is too small, which almost reduces this to the gradient descent case, means the process takes too long to converge. Critical damping results when the right β is picked can be chosen with respect to the step size α.

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t g(\mathbf{x}_t, \xi_t) + \beta_t(\mathbf{x}_t - \mathbf{x}_{t-1}) \tag{1.2.5}$$

Additionally, instability arises in long-term dependencies [Bengio et al., 1994] and the architectural choice itself [Doya, 1993] [Bengio et al., 1993] [Pascanu et al., 2013]. In the following section, we examine these long-term dependencies and architectural changes.

1.2.3. Nonlinear models and implicit regularization in deep networks

The empirical risk, which was described in 1.1.1 [Hardt and Recht, 2022, p.35], can be used for nonlinear models with layers k as linear relationship structure mappings known as features, $\mathbf{x}_{k+1} = \phi(\mathbf{W}_l \mathbf{x}_k + \mathbf{b}_k)$ or as a recurrent structure with $\mathbf{x}_{k+1} = f(\mathbf{x}_k)$ such as in a deep neural network with a composition of functions $f \equiv f_0 \circ f_1 \circ f_2 \ldots$ and $\boldsymbol{\theta} = \{\mathbf{W}_0, \mathbf{b}_0, \ldots \mathbf{W}_K, \mathbf{b}_K\}$ the empirical risk is the following:

$$R_{emp}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} l(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i) \tag{1.2.6}$$

Can use a gradient method, such as gradient descent, to optimize, where $\mathbf{y}_t$ are predictions from the model $f_{\boldsymbol{\theta}}$, and then $\nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)$ is a $|\boldsymbol{\theta}| \times N$ Jacobian, and $\mathbf{y}$ targets such that the update rule becomes:

$$\mathbf{y}_t = \begin{bmatrix} f(\mathbf{x}_0, \boldsymbol{\theta}_t) \\ \vdots \\ f(\mathbf{x}_N; \boldsymbol{\theta}_t)) \end{bmatrix} \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)(\mathbf{y}_t - \mathbf{y})$$

Theorem 1.2.1 (Taylor's Theorem)**.** *If f is a continuously differentiable function, then for some $t \in [0,1]$,*

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(t\mathbf{x} + (1-t)\mathbf{x}_0)^\top (x - \mathbf{x}_0)$$

If f is twice continuously differentiable, then,

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_0) + \int_0^1 \nabla^2 f(t\mathbf{x} + (1-t)\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) dt$$

and for some $t \in [0,1]$

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0) \nabla^2 f(t\mathbf{x} + (1-t)\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) dt$$

Such that it can be written for a function f in the form as an approximation:

$$f(\mathbf{x} + \epsilon \boldsymbol{\nu}) = f(\mathbf{x}) + \epsilon \nabla f(\mathbf{x})^\top \boldsymbol{\nu} + \frac{\epsilon^2}{2} \boldsymbol{\nu}^\top \nabla^2 f(\mathbf{x} + \delta \nu)^\top \boldsymbol{\nu} \tag{1.2.7}$$

Then can rewrite with respect to a prediction $\mathbf{y}_t$, by using Taylor's theorem:

$$\begin{aligned} \mathbf{y}_{t+1} &= f(\mathbf{x}; \boldsymbol{\theta}_{t+1}) \\ &= f(\mathbf{x}; \boldsymbol{\theta}_t) + \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)^\top (\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t) + \int_0^1 \mathbf{H}(\boldsymbol{\theta}_t + s(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t))(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t) ds \\ &= \mathbf{y}_t - \alpha \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} f(x; \boldsymbol{\theta}_t)(\mathbf{y}_t - \mathbf{y}) + \alpha \boldsymbol{\epsilon}_t \end{aligned}$$

Note that $\boldsymbol{\epsilon}_t = \alpha \int_0^1 \mathbf{H}(\boldsymbol{\theta}_t + s(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t))(\nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)(\mathbf{y}_t - \mathbf{y}), \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)(\mathbf{y}_t - \mathbf{y})) ds$, and if $\mathbf{y}$ labels are removed from both sides:

$$\mathbf{y}_{t+1} - \mathbf{y} = \left(\mathbf{I} - \alpha(\nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} f(\mathbf{x}; \boldsymbol{\theta}_t)\right)(\mathbf{y}_t - \mathbf{y}) + \alpha \boldsymbol{\epsilon}_t$$

In practice, with deep learning, it can be shown that $\boldsymbol{\epsilon}_t = \frac{\alpha}{2} \mathbf{C}(\mathbf{y}_t - \mathbf{y})^\top (\mathbf{y}_t - \mathbf{y})$ [Hardt and Recht, 2022, p. 131], where $\mathbf{C}$ is the curvature, which implicitly induces regularization in this *unrolled* SGD process.

1.2.4. Unrolling

Since we are dealing with an iterative process, it is essential to acknowledge how unrolling benefits optimization with memory and run time trade-off [Hellman, 1980]. With learned optimization, the process of unrolling gets unrolled through time as the equations 1.2.13. We take a short detour into software and compiler design to review unrolling as an optimization.

In software, a common approach is an optimization known as *loop unrolling* [Aho et al., 1977] or *loop unwinding*, which is a process of, at a high level, trading off memory for time (space vs time complexity) using the same mathematical definition. One can reduce the cost of loop overhead by adding the same function within the loop in multiple instances to reduce the overall outer loop length in iterations. This trades the size of memory being used directly for speed, but the effectiveness of loop unrolling can be shown using an unrolling factor. The unrolling factor is the number of times the loop is unrolled. If this factor is increased, more memory is used; decreasing this factor results in a method much closer to a standard loop. Using an unroll factor of K, the loop body can be repeated K times, resulting in a reduced number of overall loop iterations. Using the following example, showing an unrolled vs a standard loop as shown in and modified from [Huang and Leng, 1999]:

Fig. 1.1. Standard loop:

```
for i in range(0, 4):
  g[i] = a[i] + b * c
```

Fig. 1.2. Fully unrolled(flattened) loop:

```
  g[0] = a[0] + b * c
  g[1] = a[1] + b * c
  g[2] = a[2] + b * c
  g[3] = a[3] + b * c
```

Fig. 1.3. Loop unrolled with K = 2:

```
for i in range(0, 4, 2):
  g[i] = a[i] + b * c
  g[i + 1] = a[i + 1] + b * c
```

Loop unrolling with optimization, similarly, amounts to trade-offs in overall loop iterations for storage and memory used. In the shown examples, using an unroll factor of K reduced the number of overall iterations by repeating the loop body K times. In a similar way, at a high level, note the trade-off of increased memory usage in *forward mode*, and additional loop iterations in *reverse mode* with *automatic differentiation.* We derive reverse mode automatic differentiation or backpropagation after a discussion on the forward mode in the following section.

1.2.5. Forward mode and unrolling

In the following, the explanation of the unrolling process is described using a recurrent neural network architecture. Adopting the dynamical system framework in the form of a recurrent network [Goodfellow et al., 2016], we start with a given N-length sequence x_N, to illustrate the architecture with the following dynamical system:

$$s_{t+1} = f(s_t, \theta)$$

Note, that there is a recurrence around the state s, such that at the next state timestep s_{t+1} comes from another call of the function f evaluated at previous state s_{t-1} with (shared) parameters θ.

This recursion can continue to be unrolled, similar to figure 1.2.4 as a function composition notation. Shown below with given steps N, this graph can be unrolled $N - 1$ times, and explicitly written:

$$\begin{aligned} s_1 &= f(s_0; \theta) \\ s_2 &= f(s_1; \theta) \\ s_3 &= f(s_2; \theta) \\ s_4 &= f(s_3; \theta) \end{aligned}$$

Which, when unrolled, reveals the structure in a nested form, starting with s_0

$$\begin{aligned} s_4 &= f(f(s_3, \theta); \theta) \\ &= f(f(f(s_2, \theta); \theta); \theta) \\ &= f(f(f(f(s_1, \theta); \theta); \theta); \theta) \\ &= f(f(f(f(f(s_0, \theta); \theta); \theta); \theta); \theta) \end{aligned}$$

Using an external input x_t, then the update for a recurrent model is revealed:

$$s_t = f(s_{t-1}, x_t; \theta)$$

Where the state s is defined as the carry or hidden state h of the recurrent network, such that the update becomes:

$$h_t = f(h_{t-1}, x_t; \theta)$$

This can map from an input sequence of length N with the following form $x = (x_N, x_{N-1}, \dots x_2, x_1)$, by representing the full sequence as an input to g, which can be any length as a hyperparameter, finally getting for the full sequence:

$$h_N = g_N(x_N, x_{N-1}, \dots x_2, x_1)$$
$$h_N = f(h_{N-1}, x; \theta)$$

Real-time recurrent learning [Williams and Zipser, 1989] doesn't need to store past network states, and might theoretically be used to learn dependencies of any length and online, learning at every step. Forward mode automatic differentiation and thus RTRL have large storage requirements, however - requiring $O(k \cdot |\theta|)$ where k is the state size, and $|\theta|$ is the size of the shared parameters. Using the same template gradient of the loss from above:

$$\begin{aligned}\nabla_\theta L &= \sum_t^T \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial \theta_t} \\ &= \sum_t^T \frac{\partial L}{\partial h_t}\left(\frac{\partial h_t}{\partial \theta_t} + \frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial \theta}\right)\end{aligned}$$

Note how no loss terms appear in the unrolling, as it is happening purely with respect to the hidden state parameters, and not unrolling through the loss - and it is updating with respect to the loss in an iterative fashion. However, the memory requirements increase dramatically, requiring a $|\theta|$ times more computation than the original forward pass. This is important because in forward mode, instead of propagating tangent vectors as many times as there are parameters, you can form the full Jacobian at each step. Basically, we would need as many propagations as there are parameters, and a vector in each position is called a *seed vector*. Another way to consider this approach is to think about the Jacobian-vector-product (JVP), where $\mathbf{x} = \mathbf{J}^\top \mathbf{v}$ for forward mode. In reverse mode, however, you only propagate a vector backward in time, so only one pass is needed. This can also be viewed as a Vector-Jacobian-product (VJP), where $\mathbf{x} = \mathbf{v}^\top \mathbf{J}$. To view this in the context of deep learning and then control, we continue the discussion to get the update in reverse mode with respect to the network parameters θ:

$$\nabla_\theta L = \sum_t^T \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial \theta_t}$$

Then, expanding $\frac{\partial L}{\partial h_t}$ by performing a unrolling through time T steps using the recursion $\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L_t}{\partial h_t}$, which is just reverse mode where θ_t is just a copy of the weights at time t:

$$\begin{aligned}\nabla_\theta L &= \sum_t^T \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial \theta_t} \\ &= \sum_t^T \left(\frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t} + \frac{\partial L_t}{\partial h_t}\right)\frac{\partial h_t}{\partial \theta_t}\end{aligned}$$

A particular problem emerges with the Jacobian $\frac{\partial h_{t+1}}{\partial h_t}$, namely exploding gradients, due to the eigenvalues of the ratio between ∂h_{t+1} and ∂h_t. If it is greater than 1, the gradients explode. In early work, there were clipping [Pascanu et al., 2013] techniques applied to avoid this issue. However, it continues to be a foundational problem with backpropagation and recurrent networks.

1.2.6. Backprop in neural networks

There are many ways to evaluate the gradient $\nabla_{\boldsymbol{\theta}} f$, or its approximation to get the full performance $J(\boldsymbol{\theta})$. However, alternatives give way to many trade-offs in implementation difficulty and time/space complexity. Consider a function f such as a deep network as a nested composition of applications of layers l within $f(\mathbf{x})$, and $\mathbf{x} \in \mathbb{R}^n$:

$$f(\mathbf{x}) = (\phi \circ \phi_l \circ \phi_{l-1} \circ \ldots \circ \phi_2 \circ \phi_1)(\mathbf{x}) = \phi(\phi_l(\phi_{l-1}(\ldots(\phi_2(\phi_1(\mathbf{x})))\ldots)) \tag{1.2.8}$$

From [Wright and Recht, 2022], note the structure within the nested composition, such as the following where $\phi_1 : \mathbb{R}^n \to \mathbb{R}^{m_1}$ and $\phi_i : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$ and $\phi : \mathbb{R}^{m_l} \to \mathbb{R}$, then if using the chain rule $\nabla f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = (\nabla_x \phi_1)(\nabla_{\phi_1}\phi_2)(\nabla_{\phi_2}\phi_3)\ldots(\nabla_{\phi_{l-1}}\phi_l)(\nabla_{\phi_l}\phi) \tag{1.2.9}$$

Such that these partial derivatives are evaluated at current point $\nabla_x \phi_1 : \mathbb{R}^{n\times m_1}$, and $\nabla_{\phi_i}\phi_{i+1} : \mathbb{R}^{m_i \times m_{i+1}}$, $\nabla_{\phi_l}\phi : \mathbb{R}^{m_l}$: To properly connect this with the previous section 1.3.5 on adjoints method, adapt these equations to yield structure among $\nabla_\theta f_i(\mathbf{x})$ at i stages in the chain, where vector $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_l)$ where $\mathbf{x}_i \in \mathbb{R}^{n_i}$ is required along with a value of the previous function application ϕ_{i-1}:

$$f(\mathbf{x}) = (\phi \circ \phi_l \circ \phi_{l-1} \circ \ldots \circ \phi_2 \circ \phi_1)(\mathbf{x}) \tag{1.2.10}$$

$$= \phi(\phi_1(\mathbf{x}_{l-1}, \phi_{l-2}(\mathbf{x}_{l-2}(\mathbf{x}_l, \phi_{l-1}(\ldots(\mathbf{x}_3, \phi_2(\mathbf{x}_2, \phi_1(\mathbf{x}_1))))\ldots)) \tag{1.2.11}$$

It can be shown that this method of adjoints, backpropagation, which applies progressive functions $\phi_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{m_1}$, $\phi_i \in \mathbb{R}^{n_i} \to \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}$ where $(i = 2, 3, \ldots, l)$ and finally

$\phi : \mathbb{R}^{m_l}$ to show the final composition with any data $\mathbf{x}_i$:

$$\nabla_{\mathbf{x}_i} f(\mathbf{x}) = (\nabla_{\mathbf{x}_i}\phi_i)(\nabla_{\phi_i}\phi_{i+1})(\nabla_{\phi_{i+1}}\phi_{i+2})\dots(\nabla_{\phi_{l-1}}\phi_1)(\nabla_{\phi_1}\phi) \tag{1.2.12}$$

This exactly corresponds to the backward pass shown later in 1.3.24. In fact, one can observe the following orders of convergence in the VJP vs JVP section 1.2.5. Next, we discuss TBPTT and then work out an unrolled example at the extremes.

1.2.7. TBPTT and Unrolling SGD with an optimizer

In addition to backpropagation through time, one might consider cutting the length of an optimization loop unrolling by, for example, setting the length in steps from N to τ, which is helpful in backpropagation/reverse mode because it scales linearly with the number of iterations, by shortening the length $\tau < N$, the unrolls become shorter. This is shown in [Williams and Zipser, 1995, Aicher et al., 2020]. The trade-off for this technique is that it adds bias to the gradients. In an example to illustrate, consider learning a learning rate using the gradient descent rule, such as in [Metz et al., 2019], where m is an optimizer with a loss l defined in the following way, similarly to [Andrychowicz et al., 2016]:

$$\mathbf{w}_{t+1} = m(\mathbf{w}_t; \alpha) = \mathbf{w}_t - \alpha\nabla l(\mathbf{w}_t)$$

Then, the unrolling with l as an inner loss can be written using the following total derivative and chain rule applied to the update, to result after T steps of gradient descent with respect to the learning rate α (a scalar) as a single (hyper)parameter with a simplified update $\mathbf{w} - \alpha\nabla l(\mathbf{w})$, then if $\nabla^2_{\mathbf{w}} l(\mathbf{w}_t) = \mathbf{H}_t$ and $\nabla_{\mathbf{w}} l(\mathbf{w}_t) = \mathbf{g}_t$, noting the recursion around the $\frac{d\mathbf{w}_{T-1}}{d\alpha}$ term:

$$\frac{d\mathbf{w}_T}{d\alpha} = \frac{\partial \mathbf{w}_T}{\partial \mathbf{w}_{T-1}}\frac{\partial \mathbf{w}_{T-1}}{\partial\alpha} - \frac{\partial \mathbf{w}_T}{\partial \alpha} \tag{1.2.13}$$

$$= \left(\mathbf{I} - \nabla^2_{\mathbf{w}} l(\mathbf{w}_{T-1})\right)\frac{d\mathbf{w}_{T-1}}{d\alpha} - \nabla_{\mathbf{w}} l(\mathbf{w}_{T-1}) \tag{1.2.14}$$

$$= (\mathbf{I} - \mathbf{H}_{T-1})\frac{d\mathbf{w}_{T-1}}{d\alpha} - \mathbf{g}_{T-1} \tag{1.2.15}$$

$$\tag{1.2.16}$$

Then finally, expanding in the same way as in [Metz et al., 2019, section A] from $t = 1$ to $T = 1$ and looking at the form of $\frac{\partial dl}{\partial \alpha} = \langle \mathbf{g}_T, \frac{d\mathbf{w}_T}{d\alpha}\rangle = \mathbf{g}_T^\top \frac{d\mathbf{w}_T}{d\alpha}$ as a full unroll.

$$\frac{dl(\mathbf{w}_T)}{d\alpha} = \langle \mathbf{g}_T, -\sum_{i=0}^{T-1}\left(\prod_{j=i+1}^{T-1}(\mathbf{I} - \alpha\mathbf{H}_j)\right)\mathbf{g}_i\rangle \tag{1.2.17}$$

$$= -\mathbf{g}_T^\top \sum_{i=0}^{T-1}\left(\prod_{j=i+1}^{T-1}(\mathbf{I} - \alpha\mathbf{H}_j)\right)\mathbf{g}_i \tag{1.2.18}$$

In the section, it can also be shown that this unrolled gradient (such as in figure 1.3) can be shown as just the negative inner product of the current and previous gradients:

$$
\begin{aligned}
\frac{dl(\mathbf{w}_T)}{d\alpha} &= -\langle \mathbf{g}_T, \mathbf{g}_{T-1} \rangle && (1.2.19) \\
&= -\mathbf{g}_T^\top \mathbf{g}_{T-1} && (1.2.20) \\
&= -\nabla l(\mathbf{w}_T)^\top \nabla l(\mathbf{w}_{T-1}) && (1.2.21)
\end{aligned}
$$

This reveals reminiscent discussion 1.2.1 in prehistory and 1.2.2 momentum error. We would like tools to analyze the stability and convergence of these systems given perturbations in the input to observe how they behave and correct divergences.

1.3. Optimal Control

1.3.1. Optimal Control Problems

To lay the foundation for further discussion, we consider a few optimal control problems (OCP) in the discrete-time framework, more specifically, an equality-constrained problem. To illustrate, start with a simple linear program of the following form:

$$
\begin{array}{lll}
\text{minimize} & c(\mathbf{x}) & \text{minimize some function cost} \\
\text{subject to} & g(\mathbf{x}) = 0, & \text{subject to some (equality) constraints} \\
\text{given} & 0 \leq \mathbf{x} \leq 1 & \text{at endpoint constraints between 0 and 1}
\end{array}
\qquad (1.3.1)
$$

To start, it can be solved using a *Lagrangian method* formulation, given that there is an equality constraint such as $g(\mathbf{x}) = 0$ and an objective function $c(\mathbf{x})$ and Lagrange multiplier $\boldsymbol{\lambda}$:

$$
\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = c(\mathbf{x}) + \boldsymbol{\lambda}^\top g(\mathbf{x}) \qquad (1.3.2)
$$

Moreover, in an OCP, consider the following nonlinear program with terminal cost $c_T(\mathbf{x}_T)$, controls $\mathbf{U} = (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{T-1})$, a sum of costs $\sum_t c_t(\mathbf{x}_t, \mathbf{u}_t)$ some dynamics f and an initial state $\mathbf{x}_0$ to get the Bolza problem (named in continuous time):

$$
\begin{array}{lll}
\text{minimize} & c_T(\mathbf{x}_T) + \sum_{t=0}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t) & \\
\text{subject to} & \mathbf{x}_{t+1} = f_t(\mathbf{x}_t, \mathbf{u}_t), & t = 0, 1, \ldots, T-1 \\
\text{given} & 0 \leq \mathbf{x} \leq 1 &
\end{array}
\qquad (1.3.3)
$$

At a high level, if given *open-loop* problem formulation, where controls $\mathbf{U}*$ are only optimal for an initial state $\mathbf{x}_0$, one can use *direct methods* or *indirect methods*. Solving with an *indirect method* can use necessary conditions (like the Pontryagin maximum principle), whereas *direct methods* are solved through simulation of parameterized controls or controls and states. Lastly, one could use a *closed-loop* formulation, or with optimal control law or policy for $\mathbf{x}_t$ with a differential/dynamic programming (DDP/DP) approach, or linear quadratic regulator (LQR). In the following section, we discuss the differentiation of these costs and dynamics to attempt the discovery of solutions in the framework of direct methods. This is relevant in our work because optimal control gives us an important framework to understand and utilize the differentiation with a model, while also giving us tools for assumptions like stochastic dynamics. In later discussions, such as the section on stochastic optimization and gradient estimation, we exploit this knowledge. However, for now, we continue the discussion in the background to build intuition on forward and reverse mode in control.

1.3.2. Dynamic Programming and Optimal Control

Bellman equation, solving 1.3.3 with the same cost, but new described as a *payoff* with finite horizon $\max_{a_t} \sum_t^T \beta^t F(\mathbf{x}_t, \mathbf{a}_t)$ to be *maximized* instead of a cost to be minimized, where β is a discount factor $0 < \beta < 1$ and $T(\mathbf{x}, \mathbf{a})$ is a transition model for next states.

$$V(\mathbf{x}) = \max_{\mathbf{a}} \{F(\mathbf{x}, \mathbf{a}) + \beta V(T(\mathbf{x}, \mathbf{a}))\} \tag{1.3.4}$$

In future work, we intend to use what will be called SVG(1), in which you construct a value function to use with SVG beyond just a method of collecting gradients. The Bellman equation connects control to reinforcement learning. In the following section, we view the analytic form of the policy gradient, which is differentiated directly.

1.3.3. Analytic Policy Gradient in Optimal Control

Consider the policy now parameterized by $\boldsymbol{\theta}$, which could be a neural network, such that $\pi_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{u})$ is now constrained by this class of functions. Also, envision that one knows the exact environment dynamics, such that this policy gradient can be computed *analytically* Then, consider some cost $J(\boldsymbol{\theta})$. Taking the total derivative $\frac{d}{d\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_t^T c(\mathbf{x}_t, \mathbf{u}_t)$ (just the sum of costs for simplicity) as shown in [Deisenroth et al., 2013] (section 3.3) and [Tedrake, 2004] (section 2.3):

$$\begin{aligned}
\frac{dJ(\boldsymbol{\theta})}{d\boldsymbol{\theta}} &= \sum_t^T \frac{dc(\mathbf{x}_t, \mathbf{u}_t)}{d\boldsymbol{\theta}} && (1.3.5)\\
&= \sum_t^T \frac{\partial c(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}_t} \frac{d\mathbf{x}_t}{\partial \boldsymbol{\theta}} && (1.3.6)\\
&= \sum_t^T \frac{\partial c(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}_t} \left[\left(\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}} \right) \frac{d\mathbf{x}_{t-1}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{x}_t}{\partial \mathbf{u}_{t-1}} \frac{d\mathbf{u}_{t-1}}{\partial \boldsymbol{\theta}} \right] && (1.3.7)\\
& && (1.3.8)
\end{aligned}$$

1.3.4. Forward propagation in Optimal Control

Using the OCP formulation in 1.3.3, consider the approach where one differentiates the cost $J(\mathbf{x}, \mathbf{u}) = c_T(\mathbf{x}_T) + \sum_{t=0}^{T-1} c_t(\mathbf{x}_t, \mathbf{u}_t)$ and does what then use the forward dynamics $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t)$ to do what is called *forward simulation* (differentiation of the dynamics). The following emerges for decision variable $\mathbf{u}_k$:

$$\frac{\partial J}{\partial \mathbf{u}_k} = \frac{\partial c_T(\mathbf{x}_T)}{\partial \mathbf{u}_k} + \sum_{t=0}^{T-1} \left[\frac{\partial c_t(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{u}_k} + \frac{\partial c_t(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{u}_k} \right] \tag{1.3.9}$$

$$\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_k} = \frac{\partial f(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{u}_k} + \frac{\partial f(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{u}_k} \tag{1.3.10}$$

Which corresponds to forward propagation [Williams and Zipser, 1989]. If $\mathbf{x} \in \mathbb{R}^D, \mathbf{u} \in \mathbb{R}^M$ for T time steps accumulate on the order of O(TDM).

It can be shown that, with the total derivative of the forward dynamics model in 1.3.10 followed by the chain rule:

$$\frac{d\mathbf{x}_{t+1}}{d\mathbf{u}_k} = \frac{df(\mathbf{x}_t, \mathbf{u}_t)}{d\mathbf{u}_k} \tag{1.3.11}$$

$$= \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{u}_k} + \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_k} \tag{1.3.12}$$

$$\tag{1.3.13}$$

Here it is easy to see the recurrent Jacobian $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t}$, leading to a product of recurrent Jacobians as it is simulated all the way to T. This recurrence leads to instability and either exponentially exploding gradients, and when small, leads to vanishing gradients in the sum. For vanishing gradients, adding residual connections in the architecture helps [Li et al., 2019], and exploding gradients can be clipped [Pascanu et al., 2013], although this solution does not guarantee convergence [Metz et al., 2021].

1.3.5. Lagrangian Method for Adjoint Equations

In this section, we develop the *adjoint equations* via the Lagrangian in 1.3.2 and using the OCP defined with the cost and dynamics in 1.3.3. Setting $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) = 0 \to f(\mathbf{x}_t, \mathbf{u}_t) - \mathbf{x}_{t+1}$ gives the final term. Applying the method:

$$\mathcal{L}(\mathbf{x},\mathbf{u},\boldsymbol{\lambda}) = c_T(\mathbf{x}_T) + \sum_{t=0}^{T-1} c_t(\mathbf{x}_t,\mathbf{u}_t) + \boldsymbol{\lambda}_t^\top \sum_{t=0}^{T-1} (f(\mathbf{x}_t,\mathbf{u}_t) - \mathbf{x}_{t+1}) \tag{1.3.14}$$

$$= c_T(\mathbf{x}_T) + \sum_{t=0}^{T-1} c_t(\mathbf{x}_t,\mathbf{u}_t) + \sum_{t=0}^{T-1} \boldsymbol{\lambda}_t^\top (f(\mathbf{x}_t,\mathbf{u}_t) - \mathbf{x}_{t+1}) \tag{1.3.15}$$

This then gives rise to actually revealing the adjoints, by taking the following partial derivatives with respect to each argument:

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{u},\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}_t} = f(\mathbf{x}_t,\mathbf{u}_t) - \mathbf{x}_{t+1} = 0 \tag{1.3.16}$$

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t,\mathbf{u}_t) \tag{1.3.17}$$

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{u},\boldsymbol{\lambda})}{\partial \mathbf{x}_t} = \frac{\partial c_t(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} + \boldsymbol{\lambda}_t^\top \frac{\partial f(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} - \boldsymbol{\lambda}_{t-1}^\top = 0 \tag{1.3.18}$$

$$\boldsymbol{\lambda}_{t-1}^\top = \frac{\partial c_t(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} + \boldsymbol{\lambda}_t^\top \frac{\partial f(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} \tag{1.3.19}$$

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{u},\boldsymbol{\lambda})}{\partial \mathbf{x}_T} = \frac{\partial c_T(\mathbf{x}_T)}{\partial \mathbf{x}} - \boldsymbol{\lambda}_{T-1}^\top = 0 \tag{1.3.20}$$

$$\boldsymbol{\lambda}_{T-1}^\top = \frac{\partial c_T(\mathbf{x}_T)}{\partial \mathbf{x}} \tag{1.3.21}$$

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{u},\boldsymbol{\lambda})}{\partial \mathbf{u}_k} = \frac{\partial c_T(\mathbf{x}_T)}{\partial \mathbf{x}} - \boldsymbol{\lambda}_{T-1}^\top = 0 \tag{1.3.22}$$

$$= \frac{\partial c_t(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{u}} + \boldsymbol{\lambda}_t^\top \frac{\partial f(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{u}} \tag{1.3.23}$$

Using these adjoint equations, one can write out *exactly* backpropagation, shown in the following: Doing forward simulation 1.3.4, or the *forward pass*, start with $\mathbf{x}_0$ from 0 to T, using equation 1.3.18:

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t,\mathbf{u}_t)$$

Then, after the forward pass, calculate backward in time starting from $T-1$ with adjoint equation 1.3.22 to $t-1$ with adjoint equation 1.3.20:

$$\boldsymbol{\lambda}_{T-1}^\top = \frac{\partial c_T(\mathbf{x}_T)}{\partial \mathbf{x}} \tag{1.3.24}$$

$$\vdots \qquad \vdots \qquad \vdots \tag{1.3.25}$$

$$\boldsymbol{\lambda}_{t-1}^\top = \frac{\partial c_t(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} + \boldsymbol{\lambda}_t^\top \frac{\partial f(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{x}} \tag{1.3.26}$$

$$\tag{1.3.27}$$

Then finally get the gradients with respect to the decision variable $\mathbf{u}_t$ with adjoint equation 1.3.23, and then sum for all decision variables:

$$\frac{\partial J(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}_t} = \frac{\partial c_t(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{u}} + \boldsymbol{\lambda}_t^\top \frac{\partial f(\mathbf{x}_t,\mathbf{u}_t)}{\partial \mathbf{u}}$$

$$\frac{\partial J(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}_k} = \sum_t \frac{\partial J(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}_t} \frac{\partial \mathbf{u}_t}{\mathbf{u}_k}$$

This is a control-oriented perspective of backpropagation, which is shown from a process-oriented point of view, outlined in [Tedrake, 2004, Ch. 10]. However, to apply it to neural networks and eventually hyperparameter optimization, one will need to make the necessary modifications of controls and states. Additionally, we discover a stochastic problem, in which we will need additional tools. We move to a reinforcement learning perspective, where we develop gradient estimation techniques for dealing with stochastic optimization problems.

1.4. Reinforcement learning

In the introduction section of [Sutton and Barto, 2018], Sutton describes reinforcement learning as learning what to do and how to map situations (states s) to actions a to maximize a reward signal $r(s,a)$. This is in light contrast in notation to optimal control literature, which minimizes a cost $c(\mathbf{x},\mathbf{u})$ with controls $\mathbf{u}$, but operates in a similar framework. From an optimal control perspective, one might call reinforcement learning *adaptive optimal control.*

1.4.1. Policy Optimization or Dynamic Programming

There are many structural and empirical reasons to consider when choosing a reinforcement learning algorithm and direction. A few reasons to consider policy optimization (PO) over dynamic programming (DP) approaches are that with PO one can optimize directly what one cares about, whereas, with DP, one might have to consider how to exploit the problem structure and think more about exploration and off-policy learning. In the particular setting of hyperparameter optimization, the success of methods that optimize the parameters directly, such as CMA-ES performs quite well for a large range of black-box optimization problems. Following this perspective, we discuss stochastic optimization problems and gradient estimation, which underpins many policy gradient methods.

1.4.2. Stochastic optimization and Policy Search

Recall the stochastic approximation examples in 1.1.1 and the minibatch example 1.1.4. One could imagine these as pieces to the following equality constrained *stochastic optimization* problem as described by [Na et al., 2022]:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = \mathbb{E}\left[g(\mathbf{x};\boldsymbol{\epsilon})\right] \\ \text{subject to} \quad & h(\mathbf{x}) = 0 \end{aligned} \tag{1.4.1}$$

In this setting, objective $f(\mathbf{x})$, gradient $\nabla f(\mathbf{x})$, or the Hessian $\nabla^2 f(\mathbf{x}) = \mathbf{H}$ might be too expensive to compute or evaluate, due to an expectation over $\boldsymbol{\epsilon}$, and samples can only be generated $\boldsymbol{\epsilon} \sim P$. In machine learning, the objective might be of the form:

$$l(\boldsymbol{\theta}) = \int l(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) dP(\mathbf{x},\mathbf{y}) \tag{1.4.2}$$

Such that $\boldsymbol{\epsilon}_i = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^m \sim P(\mathbf{x},\mathbf{y})$ might only be available, in this case as a minibatch. We then approximate $P(\mathbf{x},\mathbf{y})$ by an empirical distribution to get the following:

$$l(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} l(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i) = \mathbb{E}\left[l(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i)\right]$$

This reveals $l(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i)$ to be a negative log-likelihood such as in maximum likelihood estimation (MLE) given constraints $h(\mathbf{x})$. In stochastic optimization, the problem posed in 1.4.1 can be viewed as a connection between constrained problems with solutions that utilize *augmented* Lagrangian methods and sequential quadratic programming (SQP), while solutions to stochastic problems in the unconstrained setting have been approached by first-order and second-order methods, gradient estimation, and trust-region methods. In reinforcement learning, there are solutions proposed to deal with this unique problem, such as policy and value-based methods. In this section, the focus is on gradient estimation, policy search, and policy optimization, with reasons outlined in the proceeding, elaborated discussion.

1.4.3. Gradient Estimation

In this section, the discussion centers on derivative estimation [Mohamed et al., 2020] of functions that are difficult to optimize due to stochasticity, or when the function of interest can only be accessed through noisy observations. This is typically the case when we do stochastic optimization, where as shown in the stochastic optimization section 1.4.1:

$$\min \mathbb{E}\left[g(\mathbf{x}; \boldsymbol{\xi})\right] \tag{1.4.3}$$

To look more closely, it helps to define *distributional* parameters, where $\boldsymbol{\theta}$ only appears in $p_{\boldsymbol{\theta}}$, and *structural* parameters, where $\boldsymbol{\theta}$ appears inside of the function f. This is shown below as in [L'ecuyer, 1990], also take note that in the structural case $p(\cdot)$ is could be some other distribution, as opposed to in the distributional case:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}}\left[f(\mathbf{x})\right] = \int f(\mathbf{x}) p_{\boldsymbol{\theta}}(\mathbf{x}) d\mathbf{x} \qquad \boldsymbol{\theta} \text{ is distributional} \tag{1.4.4}$$

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\cdot)}\left[f(\mathbf{x}, \boldsymbol{\theta})\right] = \int f(\mathbf{x}, \boldsymbol{\theta}) p_{(\cdot)}(\mathbf{x}) d\mathbf{x} \qquad \boldsymbol{\theta} \text{ is structural} \tag{1.4.5}$$

$$\tag{1.4.6}$$

To take the derivative of this expectation, considering the distributional and structural cases, one can derive the *score-function gradient* from the distributional case, in which the derivative pushes past the function which is our cost or loss in the integral, then we apply the log-derivative trick $\nabla_\theta \log p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{\nabla_{\boldsymbol{\theta}} p(\mathbf{x})_{\boldsymbol{\theta}}}{p(\mathbf{x})_{\boldsymbol{\theta}}} \rightarrow \nabla_{\boldsymbol{\theta}} p(\mathbf{x})_{\boldsymbol{\theta}} = p(\mathbf{x})_{\boldsymbol{\theta}} \nabla_\theta \log p_{\boldsymbol{\theta}}(\mathbf{x})$ in equation 1.4.9 and rearrange to get REINFORCE [Williams, 1992]:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{\theta}}\left[f(\mathbf{x})\right] &= \nabla_{\boldsymbol{\theta}} \int f(\mathbf{x}) p_{\boldsymbol{\theta}}(\mathbf{x}) d\mathbf{x} && (1.4.7)\\
&= \int f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathbf{x}) d\mathbf{x} && (1.4.8)\\
&= \int f(\mathbf{x}) p(\mathbf{x})_{\boldsymbol{\theta}} \nabla_\theta \log p_{\boldsymbol{\theta}}(\mathbf{x}) d\mathbf{x} && (1.4.9)\\
&= \mathbb{E}_{\boldsymbol{\theta}}\left[f(\mathbf{x}) \nabla_\theta \log p_{\boldsymbol{\theta}}(\mathbf{x})\right] && (1.4.10)\\
&= \mathbb{E}_{\boldsymbol{\theta}}\left[\nabla_\theta \log p_{\boldsymbol{\theta}}(\mathbf{x}) f(\mathbf{x})\right] && (1.4.11)
\end{aligned}$$

Now, looking at the analogous in structural case, we can see the difficulty with getting the derivative of $\nabla_{(\cdot)} p_{(\cdot)}(\mathbf{x})$ without knowing its distribution. However, in machine learning, we can make an assumption to make our lives easier. In fact, an assumption was already made with the score-function derivation in the distributional case, notably that $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ such that $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$, or in another way, that $\boldsymbol{\theta}$ are parameters of $f(\cdot)$ as a system, or world model, or environment from which we know the distribution. Using this information, let's redefine the problem through *reparameterization* or in the simulation community, known as *infinitesimal perturbation analysis* (IPA). Define a new variable $\mathbf{z} \sim \mathcal{N}(0, \mathbf{1})$ such that we can now solve for $\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\sigma}\mathbf{z}$. However, what this does is quite subtle, where we need samples of $\mathbf{z}$ to get $\mathbf{x}$. Note, $\mathbf{z}$ could also come from other distributions, as long as we can apply a *change of variable* to get back to $\mathbf{x}$. In other words, $\mathbf{x} = F^{-1}(\mathbf{z})$, also known as an *inverse transform.* Also as an important note, to do the inverse transform method, X is a random variable, such that $X \sim \mathcal{N}(X \mid 0, 1)$. Another tool we can use comes from stochastic graphs [Schulman et al., 2015a], where we want to unblock the derivative path on an directed acyclic graph (DAG) by setting a stochastic node as a constant, thereby differentiating through it, like the *straight-through estimator* (STE) [Bengio et al., 2013, Yin et al., 2019]. Armed with these tools, we can now tackle the structural dependency case:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})}\left[f(\mathbf{x}, \boldsymbol{\theta})\right] && (1.4.12)\\
&= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(0, \mathbf{1})}\left[f(\mathbf{x}(\mathbf{z}, \boldsymbol{\theta}))\right] && (1.4.13)\\
&= \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(0, \mathbf{1})}\left[\nabla_{\boldsymbol{\theta}} f(\mathbf{x}(\mathbf{z}, \boldsymbol{\theta}))\right] && (1.4.14)\\
& && (1.4.15)
\end{aligned}$$

This completes the *pathwise gradient* or *reparameterization gradient*, and to take Monte Carlo estimators [Metropolis and Ulam, 1949] of the two approaches in 1.4.7 and 1.4.12, we get the following:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathcal{J}_{score-function} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\theta} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i) f(\mathbf{x}_i) \qquad \text{where } \mathbf{x}_i \sim \mathcal{N}(0, \mathbf{1}) \tag{1.4.16}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathcal{J}_{pathwise} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} f(\mathbf{x}(\mathbf{z}_i, \boldsymbol{\theta})) \qquad \text{where } \mathbf{z}_i \sim \mathcal{N}(0, \mathbf{1}) \tag{1.4.17}$$

However, the probability densities $p(\cdot)$ could be characterized by a different distribution.

1.4.4. Concluding thoughts

In conclusion, to compute $\nabla_\theta J(\theta)$, which is the gradient of the performance $J(\theta)$ with respect to parameters θ, it is important to illuminate differences between approaches and trade-offs. These gradient estimation tools give us an extra degree of freedom for solving stochastic optimization problems.

1.5. Hyperparameter optimization

Consider the problem of evaluating a system's performance with a stationary configuration (or context), c. Given a budget of time as a resource, T, one could run the system for the entire budget and check the performance given the current configuration. This could be any sort of model of which a practitioner wishes to evaluate the performance given a budget T and configuration c with inputs from some task distribution D that gives inputs $x, x_{valid} \sim D$. At the end of this training budget, the practitioner evaluates this system using a metric of their choice with x_{valid} to characterize performance J. Unfortunately, what is usually found, is that the system, run to completion, results in lower performance than desired and requires tuning. This is because the configuration c is not guaranteed to be a sufficient configuration of the system, outside of the particular domain knowledge of the practitioner. In particular, one way to view the optimal system configuration for hyperparameters is by using the framework from [Hutter et al., 2019], where within a configuration space Λ the selection of the optimal hyperparameter $\lambda \in \Lambda$ can be shown as the following, where V is a way to evaluate the loss L on the algorithm or system A with a validation protocol, such as the following:

$$\lambda^* = \arg\min_{\lambda} \mathbb{E}_{D_{tr}, D_{val} \sim D} V(L, A_\lambda, D_{tr}, D_{val})$$

The simplest approach to a choice in hyperparameters for a given system would be to consider a choice of a uniform range of choices over a reasonable domain. In effect, what the practitioner intends is to provide *coverage* over a possible number of solutions to ensure the optimum is among them. Grid search illustrates this approach, where the pre-determined domain has the hyperparameters equally spaced, giving the illusion of coverage. In fact, it is shown with simple examples that a learning system with high sensitivity to hyperparameters suffers in performance in training and validation. For example, with an algorithm $A = \text{SVM}$, where $\Lambda = \{C = \{5, 10, 20\}\}$.

To alleviate this issue, random search [Bergstra and Bengio, 2012] proposes more coverage by sampling hyperparameters in the domain of interest, making it more likely that, in validation, better settings are discovered. Random search can be further described as searching for random configurations of the system, in the system identification [Eykhoff, 1974] context. Using the algorithm $A = \text{SVM}$ above, the practitioner can choose some number of m hyperparameters in configurations to use for validation $\lambda_{0:m} \sim U(\Lambda_{min}, \Lambda_{max})$, use the same in equation 2.2.1.

1.5.1. Hyperparameter optimization in a model-free setting

In addition to these approaches, the discussion can be extended to other model-free methods. The label *model-free* will be overloaded into a reinforcement learning context later in this work. However, this section refers to model-free approaches in hyperparameter optimization. Keeping in theme with traditional hyperparameter approaches, gradient-free methods are natural to apply to black-box optimization, and by extension, hyperparameter optimization. One example is a gradient-free method called CMA-ES (Covariance Matrix Adaptation - Evolutionary Strategies) [Hansen et al., 2003] for hyperparameter optimization of an SVM (Support Vector Machines) algorithm, or for neural networks [Friedrichs and Igel, 2005, Loshchilov and Hutter, 2016]. Similarly, there are other population-based methods, such as Population Based Training (PBT) [Jaderberg et al., 2017] for tuning neural networks, that be used in a model-free context. In addition, one can use early stopping to better utilize the budget.

In the context of hyperparameter optimization, model-free approaches are consistent with applying black-box optimization to SysID in reinforcement learning, being where the model is the environment, and the configuration is an action from a parameterized or greedy policy. Evolutionary strategies have also been used for reinforcement learning [Salimans et al., 2017], and more recently, for unrolled optimization [Vicol et al., 2021], which will be discussed in more detail in later sections, with reinforcement learning. Given a model f for the update rule $\boldsymbol{\theta} = \boldsymbol{\theta} - \nabla f(\mathbf{x})$, the following section outlines model-based methods for hyperparameter optimization.

1.5.2. Model-based hyperparameter optimization

Given a model f and data $x \sim D$, can use Bayesian optimization (Sequential Model-Based Global Optimization, or SMBO) Bergstra et al. [2011], develop what is called an acquisition function a to decide what the next point to query will be, such as expected improvement.

$$a(\lambda) = \mathbb{E}[max(f_{min} - y, 0)]$$

1.5.3. Differentiating through a model

To caution the adoption of a gradient-based approach, [Bengio et al., 1994] notes that "learning long-term dependencies with gradient descent is difficult.", and this problem is particularly difficult given the gradients come from a learned process. It is shown that with the training loss, the gradient becomes difficult and uninformative when the learning rate is large for the inner process. Overfitting is also a common issue with these approaches, where memorization can occur. There have been approaches that intend to differentiate through a model, rather than just using one such as in Bayesian optimization. In [Maclaurin et al., 2015a], the authors suggest that one could compute exact gradients of the validation performance with respect to the hyperparameters by chaining derivatives backwards through training with reverse mode. This work showed that these hypergradients allow for the optimization of multiple hyperparameters, such as step size, mometum in a layer-wise architectural way, as shown in the example of reversing dynamics of SGD with momentum. In addition, in later work given a model f [Franceschi et al., 2017] develop and forward and reverse mode model for hypergradients. For the reverse mode hypergradient (Reverse-HG) approach, where λ are the hyperparameters, and θ is an optimizer state It is also possible to use implicit differentiation as a way to do gradient-based hyperparameter optimization. One popular approach [Lorraine et al., 2020] shows that IFT can be used to approximate what is called the "best response" Jacobian using an approximate inversion algorithm for the inverse Hessian. They show hypergradients broken into multiple parts as a nested optimization procedure, by getting an implicit function:

$$\lambda^* = \arg\min_{\lambda} L(\lambda, \theta^*(\lambda)) \qquad \theta^*(\lambda) = \arg\min_{\theta} L_T(\lambda, \theta)$$

Where the implicit function is $\theta^*(\lambda)$, which is called the "best response" of weights θ to hyperparameters θ. Showing in the hypergradient, a hyperparameter direct gradient term $\frac{\partial L(\lambda,\theta^*(\lambda))}{\partial \lambda}$, and a hyperparemeter indirect gradient term, which, the second part, is composed of a parameter direct gradient $\frac{\partial L(\lambda,\theta^*(\lambda))}{\partial \theta^*(\lambda)}$ and then finally, the best response Jacobian $\frac{\partial \theta^*(\lambda)}{\partial \lambda}$. The previous section lay the foundation for understanding hyperparameter optimization and its challenges, in particular with utilizing or computing the Jacobian, and its approximations. In a similar way, we approximate gradients by using reparameterization with a learned model, but before doing this, we show that it is possible to use reinforcement learning for hyperparameter optimization in a model-free context.

Chapter 2

Model-free RL hyperparameter optimization

The first example for model-free optimization can be considered as [Andrychowicz et al., 2016], where the authors use an LSTM recurrent model to learn an update rule. Building on this, Learning to Optimize (L2O) [Li and Malik, 2016] proposes using reinforcement learning to optimize by framing automating algorithm design as a method to learn an optimization algorithm. RL^2 [Duan et al., 2016] proposes the task of learning algorithms using off-the-shelf methods (such as Trust Region Policy Optimization, TRPO [Schulman et al., 2015b]) in their case to optimize the policy, which is an RNN policy, on a range of bandit-style tasks. In their work, they learn a GRU policy to solve bandit problems, with the expectation that if the policy is learned, it should be competitive with theoretically optimal algorithms. Interestingly, RL^2 authors suggest an end-to-end approach for RL, aiming for a goal analogous to automatic machine learning via model generation. Rather than hand crafting RL algorithms, one should prefer to learn an algorithm. In a follow up to L2O, the authors then apply this method to learning neural nets [Li and Malik, 2017], with a method called predicted step descent, which uses guided policy search (GPS) [Levine and Koltun, 2013] to learn using iterative LQR estimates of a difficult-to-learn nonlinear policy and an easier-to-learn linear policy to update the optimization process of a neural network. Benefiting in similarity from these approaches, the model-free approach in this section is applied to optimize an algorithm. Moreover, in this case, the algorithm is an optimizer (eg. SGD, Adam) on a toy example and MNIST task, where the policy optimization is performed by Proximal Policy Optimization (PPO)[Schulman et al., 2017] (an extension of TRPO) similar to RL^2, and is a SOTA algorithm for continuous control tasks.

2.1. Learning to optimize with RL

Adopting the L2O approach takes an objective function $f \in \mathcal{F}$ and an optimization algorithm A and an initial iterate $\mathbf{x}_0 \sim D$, which is a neural network initialization, and produces a sequence of iterates $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ where $\mathbf{x}_T$ is the solution found by the optimizer.

$\mathcal{L}$ is some meta-loss that measures the quality of the iterates, such as $\mathcal{L}(\cdot,\cdot) = \sum_{i=1}^{T} f(\mathbf{x}_i)$. Finding the best algorithm for optimization amounts to:

$$A^* = \arg\min_{A\in\mathcal{A}} \mathbb{E}_{f\in\mathcal{F},x_0\sim D}\left[\mathcal{L}(f, A^*(f, x_0))\right]$$

This can be minimized via policy search where the algorithm is the optimal policy $A^* = \pi^*$ and the meta-loss is the cost to be minimized after state s_t following the policy π such that $\mathcal{L}(f, A^*(f, x_0)) = c(s_t, \pi^*(f, x_0))$, and the action $a = \pi(f, x_0)$ is the learning rate:

$$\pi^* = \arg\min_{\pi\in\diamond} \mathbb{E}_{s_0,a_0,\ldots,s_T}\left[\sum_{t=0}^{T} c(s_t, \pi^*(f, x_0))\right]$$

Recall the performance $J(\theta)$ policy search directly optimizes:

$$J(\theta) = \mathbb{E}\left[R(\tau) \mid \theta\right] = \int R(\tau)p_\theta(\tau)d\tau$$

Where trajectory $\tau = (s_0, a_0, \ldots, s_T)$ and $p_\theta(\tau) = p_\theta(s_0)\prod_{t=0}^{T-1} p_\theta(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t)$ for a stochastic policy or $p_\theta(s_{t+1} \mid s_t, \pi(s_t))\pi(a_t \mid s_t)$ for a deterministic policy. We consider a hybrid of the episode-based setting and the step-based setting with the policy gradient given by $\nabla_\theta J(\theta) = \int_\tau \nabla_\theta p_\theta(\tau)R(\tau)d\tau$ where the order is flipped and structure/dependencies are shown in 1.4.12. The gradient ascent to maximize the return $R(\tau)$ (as opposed to descent, minimizing the cost $c(s_t, a_t)$) is then $\theta_{t+1} = \theta_t + \alpha\nabla_\theta J(\theta)$ for the policy parameters θ with a step size α. Model-free policy search is shown in 1 in the most general setting. Although take note it is not possible to optimize the $\nabla_\theta J(\theta)$ directly, with the true risk. It needs to be evaluated under the empirical risk.

Algorithm 1 Model-Free Policy Search

```
Require: budget = $N$
  while $t < N$ do
      $\tau \sim \pi_{\theta_t}$                                  ▷ explore with generated trajectories/state-action pairs
      $\nabla_\theta J(\theta) = \int_\tau \nabla_\theta p_\theta(\tau)R(\tau)d\tau$      ▷ evaluate quality of trajectories/state-action pairs
      $\theta_{t+1} \leftarrow \theta_t + \alpha\nabla_\theta J(\theta)$              ▷ update strategy (gradient ascent)
      $t \leftarrow t + 1$
  end while
```

2.2. Hyperparameter optimization with PPO

With this template for model-free policy search, we apply the changes to convert this to a hyperparameter optimization platform using an off-the-shelf model-free policy optimization algorithm, such as Proximal Policy Optimization (PPO) [Schulman et al., 2017]. Consider the vanilla policy gradient using the likelihood ratio trick shown in section 1.4.12 for stochastic policies $\nabla_\theta J(\theta) = \int_\tau p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)R(\tau)d\tau = \mathbb{E}_{p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)R(\tau)\right]$, and an advantage function $A_{\pi_\theta}(s_t, a_t)$. One can derive PPO by optimizing a surrogate objective from the importance sampled perspective where to avoid the issue of larger ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, a clipped objective $L^{CLIP}(\theta)$ where $\hat{A}_t$ is an advantage $A(s,a) = Q(s,a) - V(s)$ estimator:

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right]$$

The algorithm is shown below in 2 which shows an actor-critic style update for the PPO algorithm on the hyperparameter optimization task. Changes from 1 are the policy network and the value network.

Algorithm 2 HyperPPO$(\cdot)$ Model-Free Hyperparameter Optimization with PPO

```
Require: budget = $N$
Require: initial optimizer policy parameters $\theta_0$, initial value function parameters $\phi_0$
  while $k < N$ do
      $\tau \sim \pi_{\theta_{old}}$   ▷ run optimizer policy in environment for $T$ steps to generate trajectories
      $\hat{A}_1, \ldots, \hat{A}_T$   ▷ compute advantage estimates from rewards-to-go $\hat{R}_t$ and $V_{\phi_k}(s_t)$
      $\theta_k = \arg\max_\theta \mathbb{E}_\tau\left[\mathbb{E}_t\left[L^{CLIP}_{\theta_k}(\theta)\right]\right]$   ▷ evaluate quality of trajectories
      $\phi_k = \arg\min_\phi \mathbb{E}_\tau\left[\mathbb{E}_t\left[\left(V_{\phi_k}(s_t) - \hat{R}_t\right)^2\right]\right]$   ▷ evaluate quality of trajectories
      $\theta_{k+1} \leftarrow \theta_k$   ▷ update strategy for policy parameters (gradient ascent)
      $\phi_{k+1} \leftarrow \phi_k$   ▷ update strategy for fitting values (gradient descent)
      $k \leftarrow k + 1$
  end while
  $\lambda \leftarrow \pi_N$   ▷ policy that gives a learning rate for task
  return optimizer policy $\lambda$
```

2.2.1. Gym Environment with optimizer

The gym environment 3 for these experiments sets the learning rate for an Adam optimizer, which runs on the meta-objective for a number of meta epochs until it reaches its goal. The PPO policy π is responsible for outputting a learning rate α, a continuous scalar, and the PPO algorithm is responsible for learning the best parameter for the optimizer.

2.2.2. PPO method as a hyperparameter optimizer

Hyperparameter optimization with PPO can be seen as a bilevel optimization problem [Colson et al., 2007] of the following form:

Algorithm 3 HyperOpt-Gym(OptTask($\cdot,\cdot$), HyperPPO($\cdot$))

```
Require: OptTask(s=flattened(params), a=learning rate) (e.g. MNIST with Adam)
Require: HyperPPO (·)                    ▷ hyperparameter tuning algorithm with policy
  state ← initialize(params)            ▷ env.reset(), init network using strategy (e.g. Glorot)
  for $0 \le episodes \le M$ do
      rewards = []
      for $0 \le steps \le N$ do
          action ← HyperPPO(state)  ▷ State is the network/task params, action is hyper
          nextstate, reward, done ← OptTask(state, action)              ▷ env.step(action)
          if sparse then                                       ▷ If in the "sparse" setting
              if steps == N then
                  rewards.append(reward)                                ▷ Keep last reward
              else
                  rewards.append(0.0)                  ▷ Observe 0 all of the other timesteps
              end if
          end if
          if dense then                                         ▷ If in the "dense" setting
              rewards.append(reward)                           ▷ Keep all observed rewards
          end if
          state ← nextstate
          if done then
              stare ← initialize(params)                                       ▷ env.reset()
          end if
      end for
      rewards.backward()
  end for
```

$$\boldsymbol{\theta}_{new} = \arg\max_{\boldsymbol{\theta}} L^{CLIP}(\boldsymbol{\theta}; \hat{A}_{\pi_{old}}(s = \mathbf{w}^*, a = \lambda_{lr})) \tag{2.2.1}$$

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \mathcal{L}_T^{task}(\mathbf{x}; \mathbf{w}, \lambda_{lr}) \tag{2.2.2}$$

Here, the inner task loss is run to a terminal state and either the last loss $\mathcal{L}_T^{task}$ is returned or the mean of the task losses. The outer loss is the PPO surrogate objective L^{CLIP}, with the state s being the parameters $\mathbf{w}$ of the inner problem optimizer, and the hyperparameter λ_{lr} for that optimizer is the action a, only allowed to be changed at the beginning of the inner optimization by the PPO policy π_{old} while running on task data $\mathbf{x}$. These are then used with the advantage estimator $\hat{A}$. To this end, HyperPPO and the HyperGym were developed to study hyperparameter optimization in an episodic setting mainly for experimentation and a proof of concept. To apply this to hyperparameter tuning in a more natural setting, one needs to consider using a model optimizer alongside the true optimizer. This leads to the following discussion on model-based reinforcement learning hyperparameter optimization.

Chapter 3

Model-based RL hyperparameter optimization

Model-based learned optimization has been richly studied as a form of semi-parametric modeling, where smoothing and kernels are often used. This statistical learning that blends together learning from data and learning from structure [Chen et al., 2021, Section 3.2] (e.g. priors). Of the different types of model-based learned optimizers, two popular groups persist, namely plug-and-play optimization and unrolled algorithm optimization [Chen et al., 2021]. Plug-and-play (PnP) learned optimizers are from a non-convex framework which combines denoising priors into methods such as proximal methods (e.g. ADMM). These optimizers show promise in that they can use these priors even when there is not enough data for proper denoising even for end-to-end training [Ryu et al., 2019]. The other method known as algorithm unrolling either focuses on learning a particular objective or to retain reconstruction accuracy of a base signal. Gradient descent can be viewed from the perspective of what is called a forward-backward splitting method [Singer and Duchi, 2009], where the gradient application is the forward method applied, and the backward is just the identity [Chen et al., 2021]. Unrolled algorithms with sparsity constraints, which are popular with the sparse coding community consist of iterative shrinkage thresholding algorithm (ISTA) [Daubechies et al., 2004], and its variants such as Fast ISTA (FISTA) [Beck and Teboulle, 2009], and Learned ISTA (LISTA) [Gregor and LeCun, 2010] for reconstruction given noise to retain a base signal. One can consider algorithm unrolling in sequence models as a backpropagation algorithm, or as unrolled optimization to solve a full-length problem by unrolling intermediate iterations. Rather than the 2nd order derivative which can be seen in methods like Model-Agnostic Meta-Learning (MAML) [Finn et al., 2017], or a K-step model which requires the storage of a Hessian at every iteration step, First-Order MAML (FOMAML) [Nichol et al., 2018] approximates the derivative as the identity, relying on a notion that 2nd order derivative terms carry little information because neural nets are generally locally linear [Goodfellow et al., 2014b] [Goodfellow et al., 2014a] [Nagarajan and

Kolter, 2017]. Unrolled GANs [Metz et al., 2016] utilize this same assumption, while also using unrolled optimization in the GAN learning problem. This paper differentiates itself, particularly from [Goodfellow et al., 2014a] [Goodfellow et al., 2020] in that instead of keeping the generator fixed during a discriminator update and the discriminator fixed during the generator update, unrolled GANs unrolls the discriminator K-steps between each generator update, using a surrogate loss $f_k(\boldsymbol{\theta}_G, \boldsymbol{\theta}_D)$. [Maclaurin et al., 2015a] uses unrolled optimization in particular for hyperparameter optimization specifically using automatic differentiation to evaluate gradients as opposed to writing them out explicitly shown in algorithm 2 of [Maclaurin et al., 2015b]. Additionally, work has been done on warm-starts [Sambharya et al., 2022], but this chapter focuses on a model-based reinforcement learning approach, combining the learning of algorithm steps of SGD similarly to [Li and Malik, 2016], but also learning a surrogate model. Most importantly, in this work, the surrogate model is used to learn the actual algorithm steps in a "throw-away" format.

3.1. Model-Based Policy Search

Recall from the previous section with 1 and the discussion on a model-free policy search method a framework for updating a policy's parameters given a trajectory. Interestingly, Guided Policy Search (GPS) is a model-based policy search method using guiding distribution for dynamics. The work repeatedly computes a policy that iteratively solves a linear dynamics and quadratic reward function framework and then connects it to MDPs and maximum entropy to get an approximate reward projection. In this work, an assumption is later made on the environment distribution, where we sample trajectories and perform what we will later call noise inference to get gradients.

3.2. Differentiating through the learned model

In the previous chapter, the model-free approach to hyperparameter optimization does not take advantage of the environment as a model, taking $\mathbf{s}_{t+1}, \mathbf{r}_t \sim \text{environment\_step}(\mathbf{a}_t)$. As a first consideration, one could take the approach of modeling this in a stochastic optimization framework, such as in section 1.4.2. Writing this out, $\xi \sim P(\text{env})$, to model the environment as a normal distribution. Consider the following normally distributed variable:

$$p(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y} \mid \mu(\mathbf{x}), \sigma(\mathbf{x})^2) \tag{3.2.1}$$

Where the $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})^2$ are outputs of a neural network, similar to a Gaussian parameterized policy network (cite). However, reparameterizing $\mathbf{y}$, we can do noise inference on ξ by the following:

$$\mathbf{y} = \mu(\mathbf{x}) + \sigma(\mathbf{x})\xi \qquad \xi \sim \mathcal{N}(0, \mathbf{1}) \tag{3.2.2}$$

Such that we sample ξ to generate $\mathbf{y}$, such that:

$$\mathbf{y} = \mu(\mathbf{x}) + \sigma(\mathbf{x})\xi = f(\mathbf{x}, \xi) \tag{3.2.3}$$

Going back to the expectation as the stochastic optimization objective, we get the following expectation under $p(\mathbf{y} \mid \mathbf{x})$:

$$\mathbb{E}_{p(\mathbf{y}|\mathbf{x})}\left[g(\mathbf{y})\right] = \int g(f(\mathbf{x}, \xi)\rho(\xi)d\xi \tag{3.2.4}$$

Such that its gradient can be estimated with the pathwise gradient estimator by using the stochastic graphs surrogate for the stochastic node $g(\cdot)$ by sampling ξ_i to compute the expectation:

$$\nabla_{\mathbf{x}}\mathbb{E}_{p(\mathbf{y}|\mathbf{x})}\left[g(\mathbf{y})\right] = \mathbb{E}_{p(\xi)}\frac{\partial g(f(\mathbf{x},\xi))}{\partial \mathbf{y}}\frac{\partial f(\mathbf{x},\xi)}{\partial \mathbf{x}} \approx \frac{1}{m}\sum_{i=1}^{m}\frac{\partial g(f(\mathbf{x},\xi))}{\partial \mathbf{y}}\frac{\partial f(\mathbf{x},\xi)}{\partial \mathbf{x}} \qquad \xi_i \sim p(\text{env}) \tag{3.2.5}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\mathbf{g}_{\mathbf{y}}\mathbf{f}_{\mathbf{x}} \qquad \xi_i \sim p(\text{env}) \tag{3.2.6}$$

This then leaves a neural network as a "mean predictor" for $g(\cdot)$ and allows us to backpropagate through the now *unblocked* deterministic node, assuming we know g. Where we learn a dynamics model $g(\mathbf{s}_t, \mathbf{a}_t, \xi) = \hat{f}(\mathbf{s}_t, \mathbf{a}_t) + \xi = \mathbf{s}_{t+1} = \hat{\mu}(\mathbf{x}) + \hat{\sigma}(\mathbf{x})\xi$ or a policy model with noise η in the following way $\hat{\pi}(\mathbf{s}; \eta; \boldsymbol{\theta}) = \mathbf{a} = \mu(\mathbf{x}) + \sigma(\mathbf{x})\eta$ is the dynamics (or policy) model. These equations underpin the allowing the policy and dynamics model to be differentiated through with backpropagation, using samples of noise inferred from the environment. This was first explored by [Heess et al., 2015] in reinforcement learning for stochastic value gradients (SVG), an extension to the deterministic value gradient counterparts in [Fairbank, 2014]. There are a few salient ways to use these questions, namely SVG(∞), SVG(1), or SVG(0). In the results section, we only performed hyperparameter optimization using the SVG(∞), leaving the others for later work. The algorithm for SVG(∞) is shown in algorithm 4, where the losses are found by looking at credit assignment on computation graphs [Weber et al., 2019]:

In reinforcement learning, pathwise gradients have recently been applied with success in Dreamer [Hafner et al., 2019] and SAC-SVG [Amos et al., 2021]. The reparameterization trick [Kingma and Welling, 2013] has also been used in variational inference [Miller et al., 2017], as well as extended to other distributional families [Ruiz et al., 2016].

3.2.1. System Identification (SysID)

In [Ross et al., 2011] develop DAgger (Dataset Aggregation) and a bound for what they call no-regret online learning in the context of imitation learning. In a similar vein, we

Algorithm 4 SVG(∞)

```
$\tau \sim p(\text{envirnoment})$
**for** $t \leftarrow T$ to 0 **do**:
    $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \leftarrow \tau_t$
    $\boldsymbol{\xi} \leftarrow (\mathbf{s}_{t+1} - f(\mathbf{s}_t, \mathbf{a}_t))/\sigma_{model}$
    $\boldsymbol{\eta} \leftarrow (\mathbf{a}_t - \pi(\mathbf{s}_t))/\sigma_{policy}$
    $\boldsymbol{\xi} \leftarrow \text{stopgrad}(\xi)$
    $\boldsymbol{\eta} \leftarrow \text{stopgrad}(\eta)$
    $l_{model} = \frac{1}{2}(\mathbf{s}_{t+1} - f(\mathbf{s}_t, \mathbf{a}_t))^2$
    $l_{policy} = \frac{1}{2}(\mathbf{a}_t - \pi(\mathbf{s}_t))^2$
```

Algorithm 5 HyperSVG($g_\theta(\cdot,\cdot)$,$\pi_\phi(\cdot)$, D_{batch_tr},unflatten)

```
**Require:** data batches $D_{traj}, D_{task}$, policy $\pi_\phi(\cdot)$, unflatten shape transformation
    unflatten($\cdot$), transition model $g(\cdot,\cdot)$, task loss $l(\cdot,\cdot,\cdot)$
**Ensure:** dynamics std $\sigma = 0.1$
**Ensure:** $\mathbf{v} \leftarrow 0$
**Ensure:** $\mathbf{s}, \_, \mathbf{s}' \sim D$
**Ensure:** $\mathbf{a} \leftarrow \pi_\phi(s)$
  **for** $t = 1 \ldots T$ **do**
      $\hat{\mathbf{f}}_t \leftarrow \frac{g_\theta(\mathbf{s}_t, \mathbf{a}) - \mathbf{s}'_t}{\sigma}$
      $\hat{\mathbf{f}}_t \leftarrow \text{stop\_gradient}(\hat{\mathbf{f}})$
      $\hat{\mathbf{s}'_t} \leftarrow g(\mathbf{s}_t, \mathbf{a}) + \sigma \hat{\mathbf{f}}_t$
      $\boldsymbol{\theta}_t \leftarrow \text{unflatten}(\mathbf{s}'_t)$
      $X_{task}, Y_{task} \sim D_{task}$
      $\mathbf{v}_t \leftarrow l(\boldsymbol{\theta}_t, X_{task}, Y_{task})$
  **end for**
  **return** $\mathbf{v}_T, \mathbf{a}$ =0
```

develop a model-based approach for system identification from observations for *controller synthesis*. [Ross et al., 2011] call system identification and controller synthesis model-based reinforcement learning. In essence, a critical issue in past approaches such as [Ljung, 1998] [Abbeel and Ng, 2004] utilizes an open-loop approach that assumes the true model exists in the classes considered, ensuring a correct model is learned. In a hyperparameter approach context, this shows similarities to a grid or random search approach which are non-adaptive. It is possible the best hyperparameter $\lambda*$ is between models selected and validated with $\lambda_{left}, \lambda_{right}$. In *System Identification* [Ljung, 1998] a discussion on learning from a generative model, an open-loop approach, or by watching an expert is outlined. The *Batch* algorithm [Ross et al., 2011] takes $\mathcal{T}$, a class of transition models consider, ν a state/action distribution exploration to sample the system from. Batch executes real system m, $(s, a) \sim \nu$ sampled iid to get m sampled transitions, then it finds the best model $\hat{\mathcal{T}} \in \mathcal{T}$, and solves the optimal control problem (even approximately) with $\hat{\mathcal{T}}$ and known cost C to return policy $\pi*$ for execution.

Algorithm 6 Offline Dyna-Style training

```
Ensure: trajectory data $D_{traj}$, task data (e.g. MNIST) $D_{task}$
Ensure: policy network $\pi_\phi(\cdot)$, policy step size $\lambda_\pi$, policy parameters $\phi$
Ensure: synthetic optimizer $g_\theta(\cdot, \cdot)$, synthetic parameters $\theta$
Ensure: model optimizer $Adam$, model optimizer step size $\lambda_m$
Ensure: task loss (e.g. MNIST) $l(\cdot, \cdot, \cdot)$, imitation loss (e.g. MNIST) $l_{MSE}(\cdot, \cdot, \cdot)$, epochs $N$
  $D_{tr}, D_{val}$ = train_test_split($D_{traj}$)
  $a \leftarrow 0.001$                                         ▷ initial learning rate
  $v \leftarrow 0$
  for $k = 0 \dots N - 1$ do
      $D_{batch\_tr} \sim D_{tr}$                               ▷ batch of trajectory dynamics
      $D_{batch\_val} \sim D_{val}$
      $\hat{\theta} \leftarrow g_\theta$(current_states($D_{batch\_tr}$))
      $v, a \leftarrow$ HyperSVG($g_\theta(\cdot, \cdot)$,$\pi_\phi(\cdot)$, $D_{batch\_tr}$,unflatten)     ▷ Algorithm referenced in 5
      $g_{pol} \leftarrow v$.backward      ▷ backprop through determinstic node from reparameterization
      $\phi_{k+1} \leftarrow \phi_k - \text{Adam}(\lambda_\pi, g_{pol})$
      $loss \leftarrow l_{MSE}(\hat{\theta}$, next_states($D_{batch\_tr}$))
      $g_{model} \leftarrow loss$.backward                      ▷ backprop from model loss
      $\theta_{k+1} \leftarrow \theta_k - \text{Adam}(\lambda_{model}, g_{model})$
```

3.2.2. Model learning

We motivate this work by learning a *synthetic optimizer* as a model of the true environment, which can be thrown away but is used specifically for hyperparameter optimization. In this way, the learned synthetic optimizer learns a representation that learns by doing the following:

- We assume the environment can be modeled by a dynamics model with a normal distribution, utilizing reparameterization gradients to differentiate through the stochastic node through unblocking
- Learn and use a model of the optimizer as a surrogate model [Schulman et al., 2015a] [Maheswaranathan et al., 2019] with a simple architecture and surrogate losses [Weber et al., 2019] rather than learning with the true optimizer
- The normal assumption on the environment smooths out the nonconvex meta-loss landscape similar to what is shown in [Vicol et al., 2021]
- We use neural network representation, assuming a system is identifiable [Roeder et al., 2021] "checkpoints" as a trajectory similar to [Peebles et al., 2022]
- Another departure from previous work is using not using the gradients of the true optimizer but learning a "flattened" representation of the architecture itself. This avoids the issues of gradient pre-processing [Hochreiter et al., 2001] [Younger et al., 2001] [Andrychowicz et al., 2016]

In the *offline* hyperparameter optimization setting, the first focus is on learning a synthetic optimizer that is representative of the true optimizer. Secondly, we see if the synthetic optimizer learned with a policy does well on task data. In the online optimizer, we use a similar architecture to show results on a task where the synthetic optimizer "steers" the true one with the hyperparameters (actions) from its policy.

3.3. Offline Dyna-style Hyperparameter Optimization

HyperSVG shown in algorithm 6 is Dyna-style [Sutton, 1990, 1991] in that it learns domain knowledge of situation, or (state) and constructs a world model of the true optimizer. This environment was developed to test if SVG(∞) gets good gradients and can construct a useful and simple world model of the true optimizer given trajectory data.

3.4. Online Hyperparameter Optimization SVG

In the online experiments, we show the learned optimizer learning alongside the true optimizer, which is SGD in this case. In these experiments, the synthetic optimizer runs in different configurations to test performance. In particular, two of the most salient settings are outlined in the following sections. For brevity, the offline and online are only different in that instead of a batch of static data being fed to HyperSVG, it is a real optimizer state.

3.4.1. Bandit approach

We formulate a bandit-style [Bergemann and Valimaki, 2006] approach, where the learned optimizer has a warm-up period where it is trained along side the true optimizer to get gradient updates, and then once it reaches a set period of time for warm-up, outlined in the experiment labels, it stops being trained and only outputs a single action (hyperparameter) for the true optimizer.

3.4.2. Adaptive

The final discussed experiments are where the synthetic optimizer steers the true optimizer at every timestep. It only uses batch data and a similar loss as the true optimizer, but it learns a representation of the true optimizer in this online setting for learning rates.

Chapter 4

Results and comparisons

We engage with a few questions to motivate the discussion and experiments for the described methods in chapter 2 and 3. In this discussion, the goal is to answer the following questions, such as:

(1) Can an agent learn a policy that discovers decent hyperparameters with model-free reinforcement learning?
(2) Is it possible to learn a model of the optimizer?
(3) Does the model help with the optimization of hyperparameters?
(4) Can we use the model to do online hyperparameter optimization?

To answer the first question, we consider the method from chapter 2 and experiments that look at a sparse and dense reward setting, where the agent either observes rewards at the last timestep, or every timestep. Answering the second question, we run experiments learning a model of the dynamics in section 4.2, where we show different trajectory lengths and the performance of the learned model on example data. Answering the third question, we refer to experiments in section 4.3 and section 4.3.1, where the model is used for the episodic (HyperPPO) and offline setting (OfflineSVG). Lastly, we answer the fourth question with section 4.3.2 on online hyperparameter optimization using a bandit-style setting (Bandit10-SVG, Bandit20-SVG, Bandit50-SVG) and a truly adaptive setting (HyperSVG, Adaptive). The next section outlines initial experiments, and the possibility to use model-based reinforcement learning to learn a surrogate optimizer model.

4.1. Initial experiments

For initial experiments, we developed an imitating learned optimizer, that learned from generated trajectory data on a toy problem illustrated in [Metz, 2021]. The goal is to fit a model of the dynamics given the collected trajectories. In particular, we use a Gaussian model where the mean is parameterized by an MLP defined in the configuration table in 4.3 for the toy problem. It takes as an input the current iterate $\boldsymbol{\theta}_t$ and hyperparameter $\boldsymbol{\lambda}$

and outputs the corresponding mean from the network. This is similar to 3.2.6, but without noise inference or unrolling.

$$\boldsymbol{\theta}_{t+1} = \hat{f}(\boldsymbol{\theta}_t, \boldsymbol{\alpha}, \boldsymbol{\xi}; \mathbf{w}) = \mu(\boldsymbol{\theta}_t, \boldsymbol{\lambda}; \mathbf{w}) + \sigma\boldsymbol{\xi} \ \ .$$

In this notation, μ is a function represented by an MLP with parameters $\mathbf{w}$. We will train this MLP with maximum likelihood via the following L2 loss:

$$L(\mathbf{w}) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}\|\boldsymbol{\theta}_t^{(i+1)} - \hat{f}(\boldsymbol{\theta}_t^{(i)}, \boldsymbol{\lambda}^{(i)}, \boldsymbol{\xi}_t^{(i)}; \mathbf{w})\|^2 \tag{4.1.1}$$

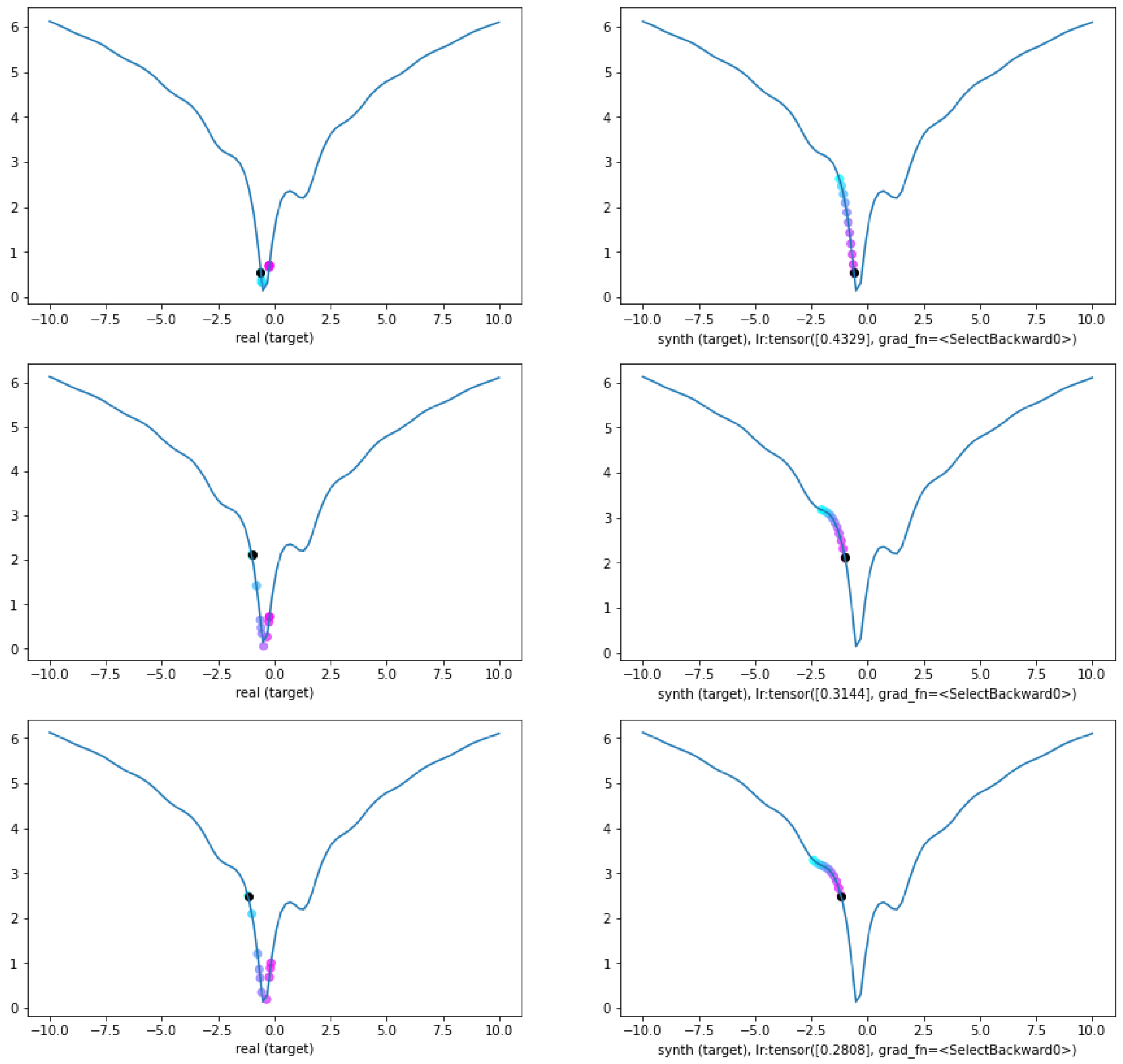
Where the superscript represents the trajectory number and the subscript is the time index within that trajectory. Observing some of the outputs of these trajectories reveals interesting behavior, as it learns to follow some of the trajectories in the generated dataset, with behavior shown of a small sample of learned optimizer outputs on the right 4.1. This serves as a proof of concept to build upon with the following sections.

4.2. Model-free hyperparameter optimization with PPO

4.2.1. MNIST Example

In this section, we explore learning trajectories of an optimizer training on the MNIST [LeCun et al., 2010] dataset. To do this, there is another gym environment that was built similar to 3, with a few modifications for evaluation, logging, and support for larger-scale experimentation. Firstly, the PPO agent that was developed diverges from the original PPO [Schulman et al., 2017] and the baselines version in that instead of using 64 hidden units, a few tests were run with a 512 hidden unit layer for both the policy and value networks. Experiments were also run with the clipping parameter, but due to the results being very similar in all approaches, the results are left out for additional brevity, however, a few plots are included in the appendix. The first set of shown experiments using the Coax [coa, 2020] library with heavy modification consists of a discussion on two large experiments run, between a *sparse* reward setting, where the PPO agent only gets to take an action at the beginning and observe a reward at the end of the episode. The *dense* setting gives the agent access to reward at all times and an action can be taken at any time looking at 4.2, it is clear to see that sparse is the preferred setting, which give incentives for the agent to pick a good learning rate that will give it good overall performance through the entire run. The trajectory length was the same as the episode, in this case, 20 steps. The interpretation here is that note the scales of the temporal difference errors, which are near zero for the sparse setting, and the policy loss and value loss plateau around zero when learned. Looking at

Fig. 4.1. Example trajectories with imitation optimizer, showing rollouts from different initializations on the toy task. These trajectories indicate that it is possible to use a learned optimizer to learn the dynamics for a given task, where the learned optimizer is trained in the model-based reinforcement learning fashion for hyperparameter optimization. This diverges from previous literature, focused on model-free reinforcement learning for hyperparameter optimization.



the evaluation average rewards, it shows the agent exploits a policy learned to get around 50 percent accuracy on the MNIST task (validation classification accuracy with a defined in configurations 4.3 network that resets). One indication is that when using a fixed policy that was a decent learning rate (0.01), the performance was still somewhere around 60 percent. This indicates the difficulty of the benchmark, and also probably some additional fixes to the gym environment that might benefit the agent. It could be that too many resets were occurring, crippling the capability of the agent to get higher accuracy. The average learning rate is also shown which is the output of the MLP policy.

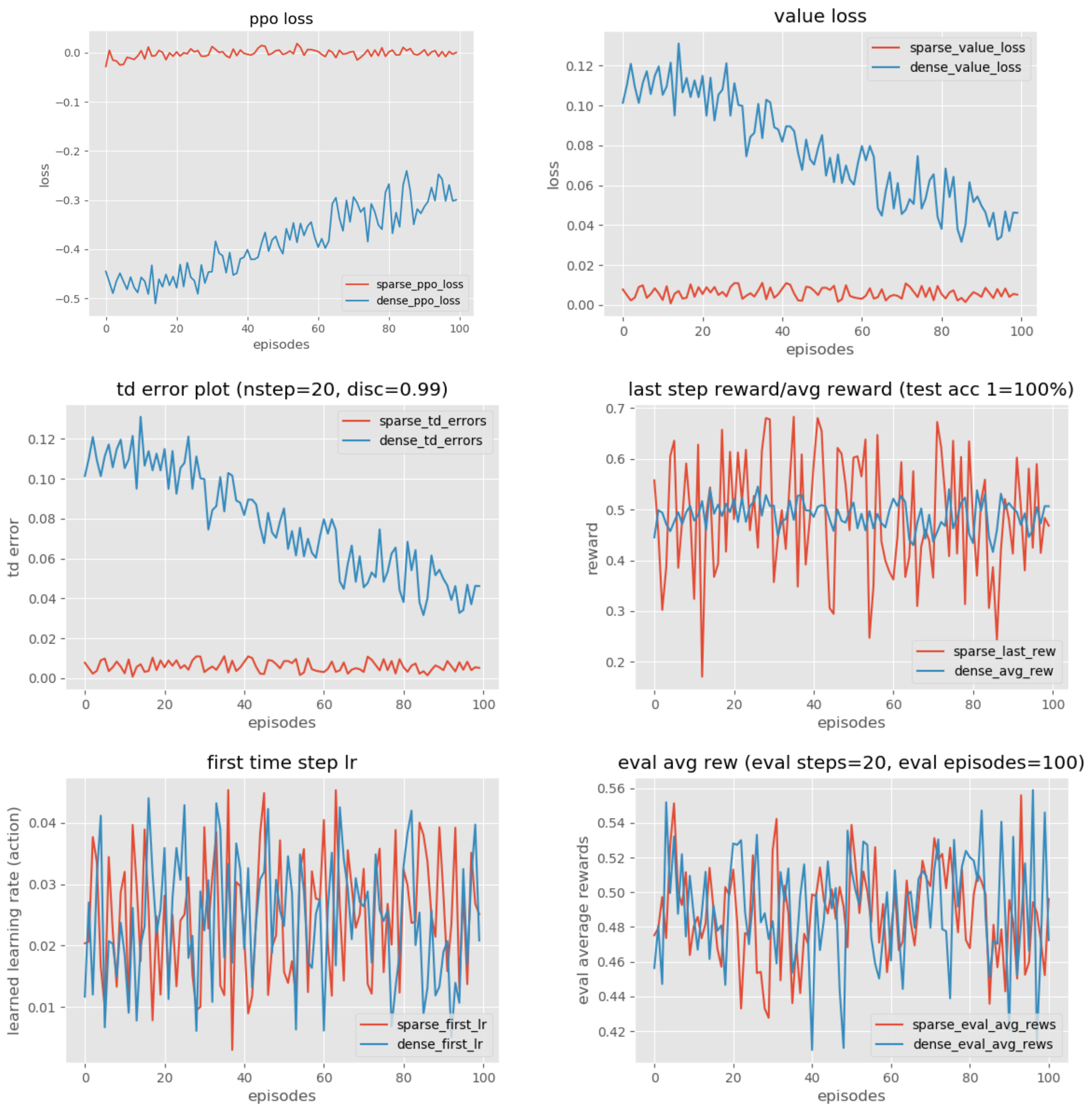


Fig. 4.2. Sparse vs dense reward experiment with PPO

4.3. Model-based hyperparameter optimization with SVG

In this section, the focus will be strictly on HyperSVG with the MNIST environment in the offline and online settings. First, beginning in the offline setting, there are a lot of interesting interpretations of results, and then finally, in the online setting, there is an additional discussion that reveals interesting results for interpretation.

4.3.1. SysID Environment

In this environment the focus is imitating trajectories through Batch [Ross et al., 2011] and [Sutton, 1991] style training, using a configuration outlined in the 4.3 The implementation of SVG(∞) is completed in PyTorch [Paszke et al., 2019]. In the loss figures in both 4.3 and 4.5, the same loss shown in 4.1.1 is minimized, but the loss is backpropagated through the unrolled synthetic optimizer. It is clear to see that all length trajectories (5, 10, 20 length unrolls) learn a representation of the true optimizer, as the loss gets very close to zero. The values are the synthetic optimizer trained by learning the representation of the true optimizer on the MNIST task. The values are MNIST classifications, where it is clear to see it approaches the blue line which is the top 20 percent of trajectories in the dataset. The average learning rates are shown in the 4.4 and 4.6.

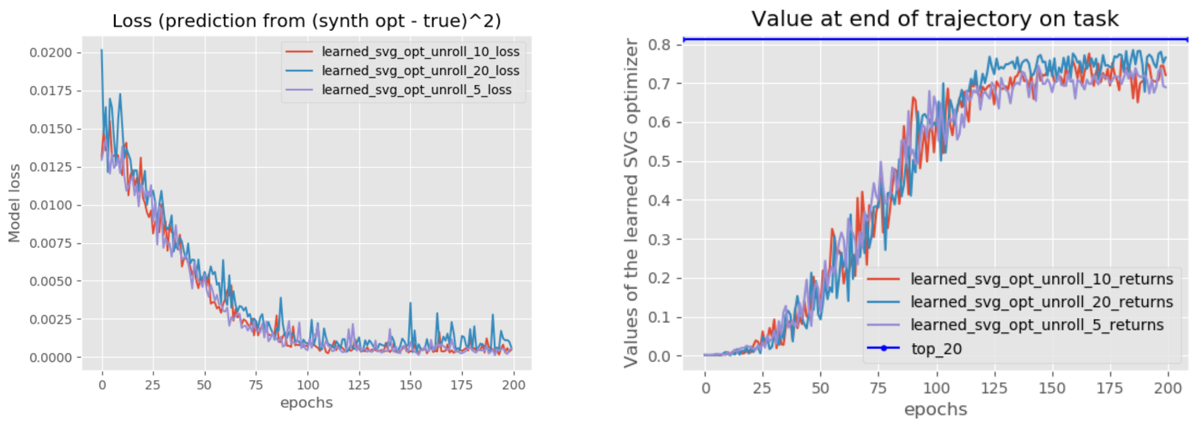


Fig. 4.3. 200 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task, showing classification validation accuracy (highest is 1.0)

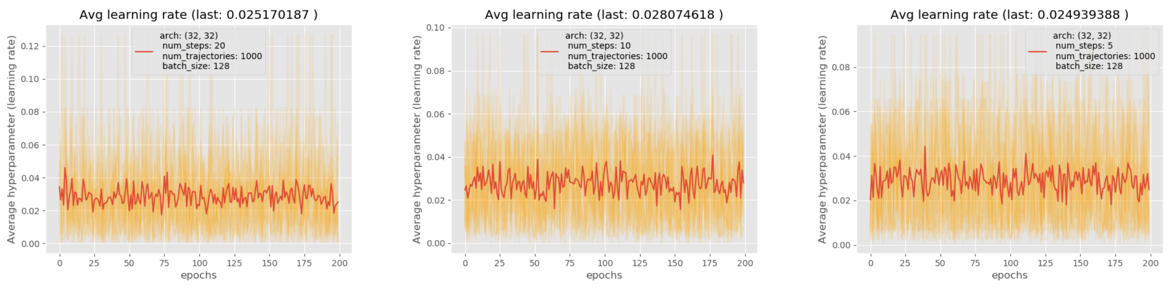


Fig. 4.4. 200 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task (learning rates)

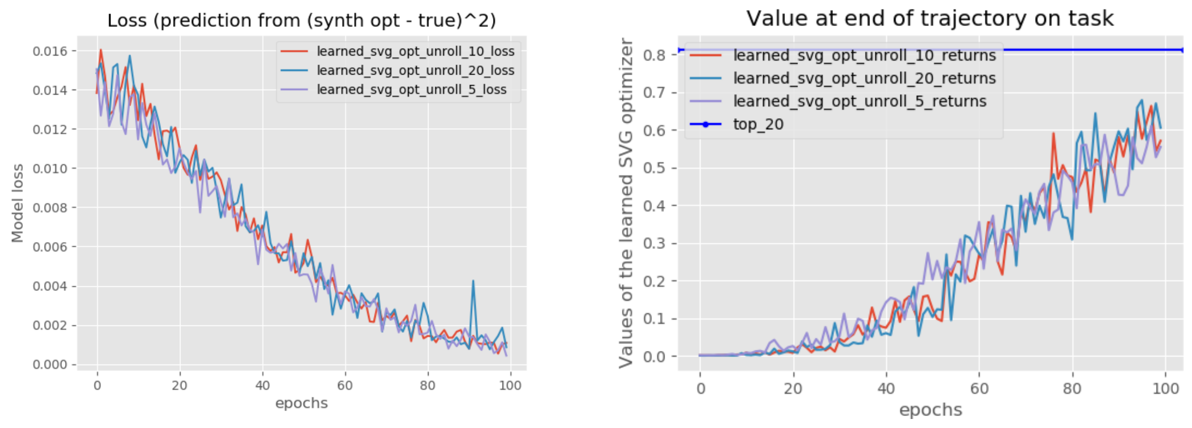


Fig. 4.5. 100 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task, showing classification validation accuracy (highest is 1.0)

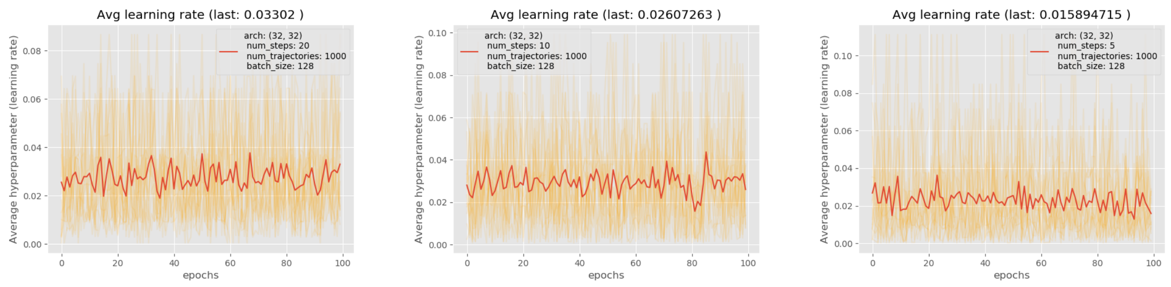


Fig. 4.6. 100 iterations with a length of 5, 10, 20 unrolls for SVG with loss and internal values on MNIST offline task (learning rates)

4.3.2. Online HyperSVG

In the online experiments, there are bandit and adaptive approaches to learn a learning rate for the true optimizer. These are shown in 4.8, 4.9 and, 4.10 with a comparison of best results in 4.2, where the synthetic optimizer adapts good and bad learning rates from grid search. To read the results, it can be seen that an optimizer starting at the hyperparameter shown in the left accuracy plot gets adapted, whose final hyperparameter value is color coordinated within the range of the colormap on the right plot (where the learned optimizer policy's last learning rate recommendation was). This shows that grid and random search maintains a fixed learning rate, adaptive SVG tunes the learning rate online with the true optimizer, and bandit SVG tunes it with the learned optimizer for a warmup period, outlined by the number $(10, 20, 50)$ and then only uses the last action from the policy, which remains fixed afterward. The bandit experiment also takes the true optimizer and resets it completely using the learned optimizer to initialize it after the warm-up (noting the spike in performance in bandit 50 experiments). Another comparison is in the running time for each method, shown in 4.11, which to be noted adds the additional runtime for the warm-up period for the bandit versions and little overhead for the online version. However, there is still work to do in modifying this method to find fixed-point solutions in the forward pass, such as with implicit methods.

4.3.3. Bayesian Optimization Baseline

As a baseline, we used a black-box optimization approach for the same experiment, using [Nogueira, 2014–] and 100 iterations with 100 initial points, and bounds set to $1e-1$ and $1e-5$ for a black-box optimizer. We compare this to Bayesian Opt with 100 iterations, which received 90.37 percent accuracy on the same style test, with a result of $lr = 0.01581$. This result is found in figure 4.7, with each epoch being a full "meta-epoch" over the task data and task model, with 20 steps of optimization happening at the inner level.

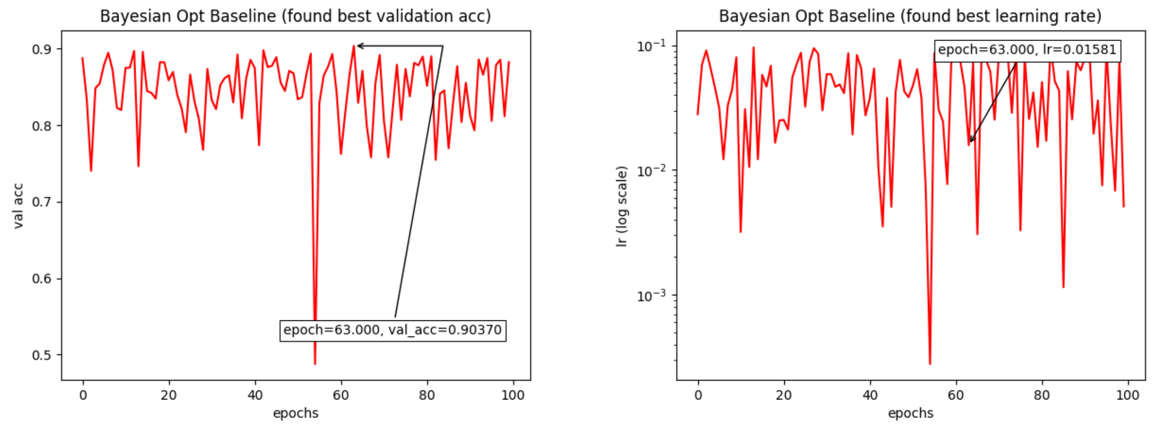


Fig. 4.7. 100 iterations with a length of 20 trajectories with Bayesian Optimization (BO) baseline with found learning rate on the MNIST task, showing classification validation accuracy (highest is 1.0)

4.3.4. Comparisons

It makes sense, in terms of the results to combine them based on an *episodic* type benchmark setting and an *online* setting. The HyperPPO, Bayesian Opt, and Offline SVG settings can be bucketed into the episodic regime. It is natural due to the gym and gym-style frameworks used to train and evaluate the optimizers. However, the online results and grid search results, are meant to directly influence the true optimizer and get a measure of true performance alongside it. Of the online results, they are also split into *good hyerparameter* and *bad hyerparameter* experiments, where the learning rate is chosen from a good healthy range of choices, or too wide a range and incorporates bad choices. This is to show that HyperSVG can adapt badly chosen learning rates and make competitive recommendations for good hyperparameters (in the bandit case).

Algorithm	**Validation Acc. (%)**	**Length of Experiment**	**Chosen lr**
Bayesian Opt Baseline	90.37	100	0.01581
(Ours) OfflineSVG	61	100	0.01589
(Ours) HyperPPO	48	100	0.021

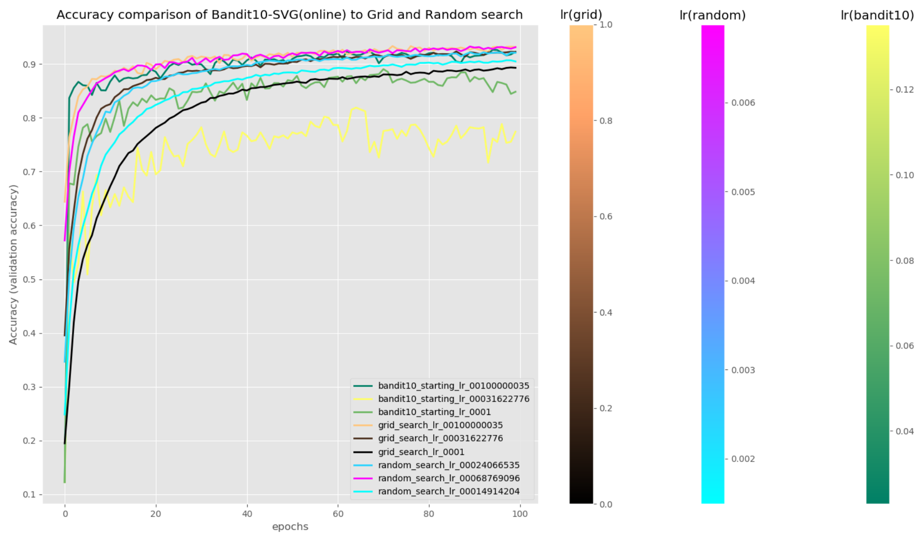
Table 4.1. Episodic comparison with methods

Algorithm	**Validation Acc. (%)**	**Length of Experiment**	**Chosen lr**
Grid Search	93.13	100	0.010
Random Search	93.10	100	0.00687
(Ours) HyperSVG (Adaptive)	83.1	100	0.0912
(Ours) Bandit10-SVG	92.23	100+10	0.02288568
(Ours) Bandit20-SVG	87.78	100+20	0.0732
(Ours) Bandit50-SVG	93.72	100+50	0.0090

Table 4.2. Online best-chosen comparison of last accuracy and last hyperparameter with methods (with good learning rate ranges)

Config	**Toy**	**MNIST (PPO)**	**MNIST (SVG)**
synthetic optimizer (hidden, output)	(32, 32, 1)	(512, 64, 64,1)	(3072,1), but varies
true optimizer (hidden, output)	(256, 256, 1)	(784, 32, 32, 10)	(784, 32, 32, 10)
activations (at hiddens)	ReLU	Tanh	ELU
number of trajectory steps	varies	varies	varies
nange of initializations	$x_0 = [-10, 10]$	Glorot	He Uniform
num epochs	100	100-200	100-200
evaluation init	$x_0 = 7$	Glorot	He Uniform
outer SGD learning rate	varies	varies	varies

Table 4.3. Environment settings



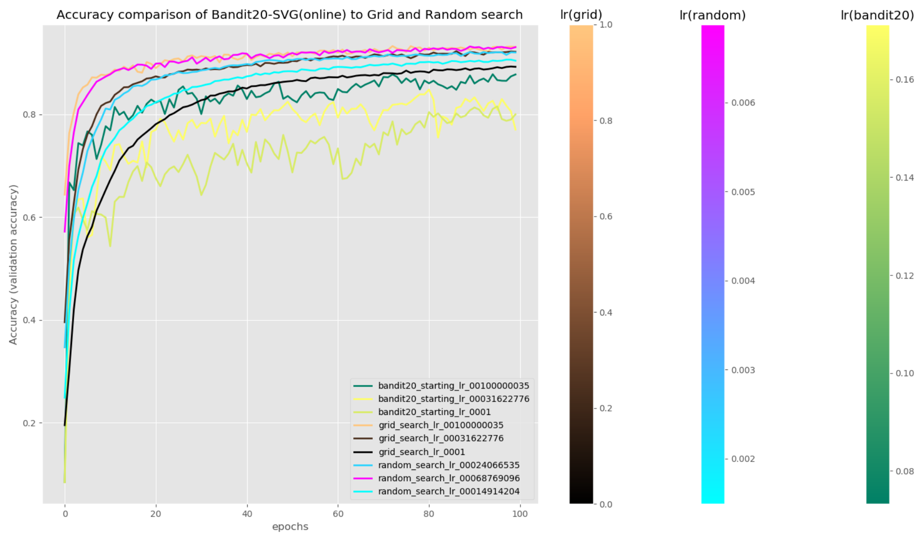


Fig. 4.8. 100 iterations with length of 10, 20, unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0)

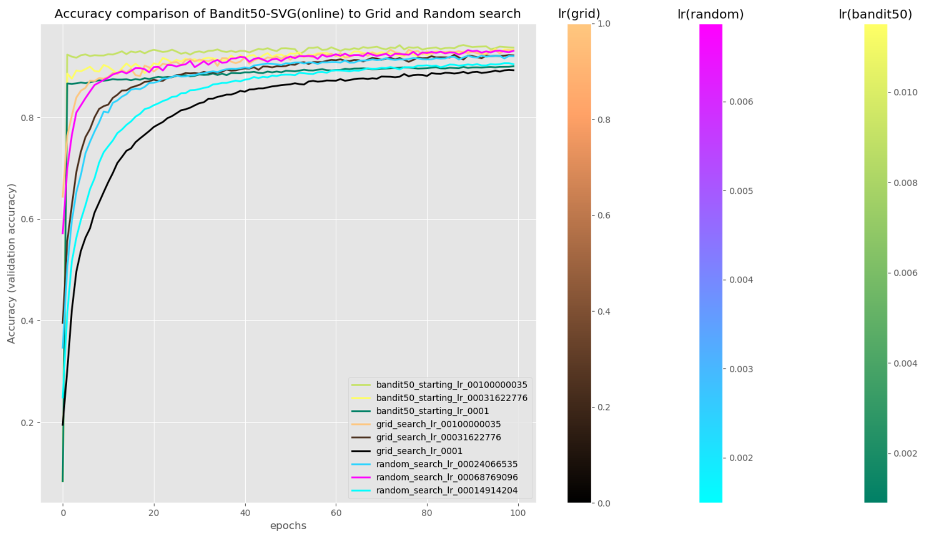


Fig. 4.9. 100 iterations with length of 50 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0)

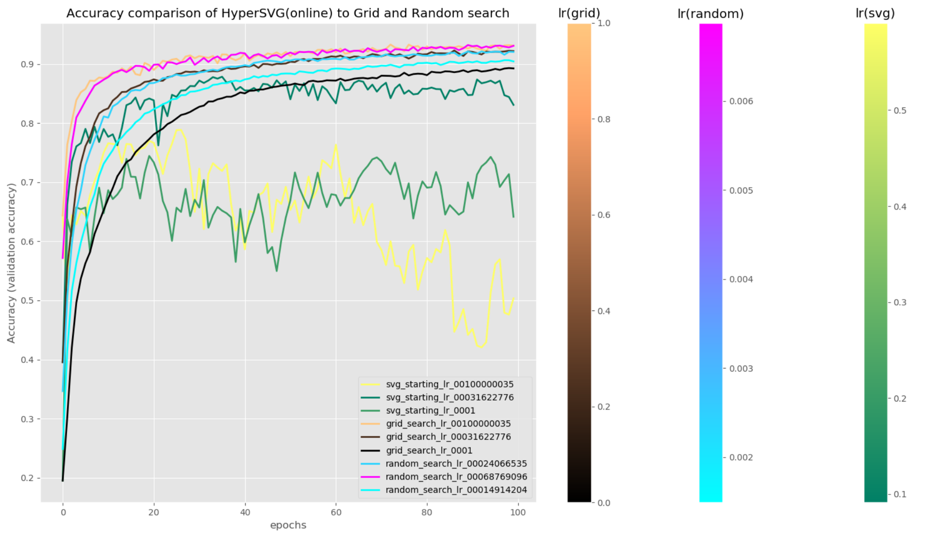


Fig. 4.10. 100 iterations with length of adaptive SVG with MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0)

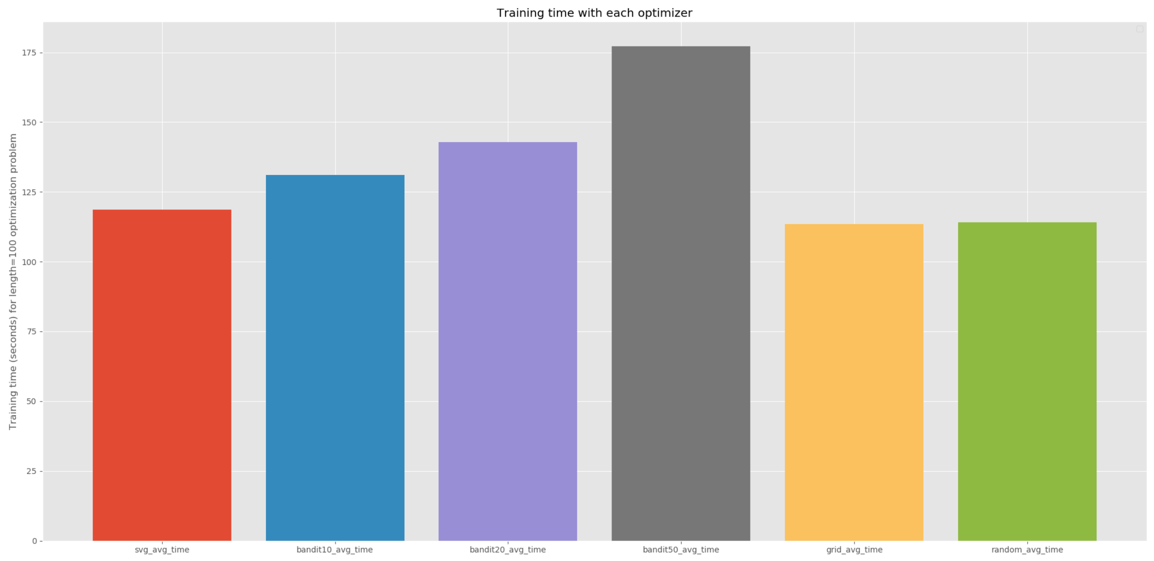
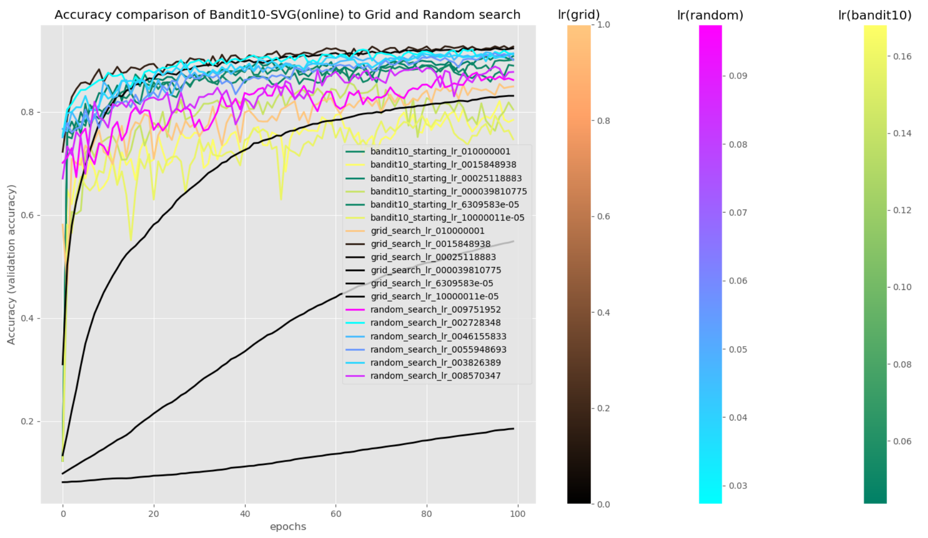


Fig. 4.11. Different runtimes (100 iterations) of MNIST online task with different algorithms (in seconds). Our methods are shown in red, blue, purple, gray

Algorithm	**Acc 1.0=100%**	**Last/learned LR**
(Ours) Adaptive-SVG (starting lr = 0.10000001)	0.8127	0.16248728
(Ours) Adaptive-SVG (starting lr = 0.015848938)	0.9037	0.0
(Ours) Adaptive-SVG (starting lr = 0.0025118883)	0.91179997	0.020831108
(Ours) Adaptive-SVG (starting lr = 0.00039810775)	0.8962	0.0006833896
(Ours) Adaptive-SVG (starting lr = 6.309583e-05)	0.6899	0.21076679
(Ours) Adaptive-SVG (starting lr = 1.0000011e-05)	0.9127	0.00031906366
Grid Search (lr = 0.10000001)	0.84919995	NA
Grid Search (lr = 0.015848938)	0.92649996	NA
Grid Search (lr = 0.0025118883)	0.92289996	NA
Grid Search (lr = 0.00039810775)	0.831	NA
Grid Search (lr = 6.309583e-05)	0.5487	NA
Grid Search (lr = 1.0000011e-05)	0.18519999	NA
(Ours) Bandit10-SVG (starting lr = 0.10000001)	0.88949996	0.050463855
(Ours) Bandit10-SVG (starting lr = 0.015848938)	0.7845	0.16824883
(Ours) Bandit10-SVG (starting lr = 0.0025118883)	0.908	0.04358094
(Ours) Bandit10-SVG (starting lr = 0.00039810775)	0.8045	0.14111823
(Ours) Bandit10-SVG (starting lr = 6.309583e-05)	0.9016	0.05018574
(Ours) Bandit10-SVG (starting lr = 1.0000011e-05)	0.7468	0.15843117
(Ours) Bandit20-SVG (starting lr = 0.10000001)	0.7741	0.2983405
(Ours) Bandit20-SVG (starting lr = 0.015848938)	0.8552	0.10251433
(Ours) Bandit20-SVG (starting lr = 0.0025118883)	0.76	0.18449491
(Ours) Bandit20-SVG (starting lr = 0.00039810775)	0.691	0.17127055
(Ours) Bandit20-SVG (starting lr = 6.309583e-05)	0.7425	0.19566791
(Ours) Bandit20-SVG (starting lr = 1.0000011e-05)	0.8398	0.07704735
(Ours) Bandit50-SVG (starting lr = 0.10000001)	0.8544	0.057865024
(Ours) Bandit50-SVG (starting lr = 0.015848938)	0.765	0.18487746
(Ours) Bandit50-SVG (starting lr = 0.0025118883)	0.83769995	0.13448596
(Ours) Bandit50-SVG (starting lr = 0.00039810775)	0.8556	0.1130591
(Ours) Bandit50-SVG (starting lr = 6.309583e-05)	0.9368	0.0060367584
(Ours) Bandit50-SVG (starting lr = 1.0000011e-05)	0.7687	0.121899426
Random Search (lr=0.09751952)	0.8617	NA
Random Search (lr=002728348)	0.91279995	NA
Random Search (lr=0046155833)	0.91349995	NA
Random Search (lr=0055948693)	0.9004	NA
Random Search (lr=003826389)	0.9111	NA
Random Search (lr=008570347)	0.87729996	NA

Table 4.4. The tabular results corresponding with figure 4.13 and 4.14, where each line in the plot is represented.



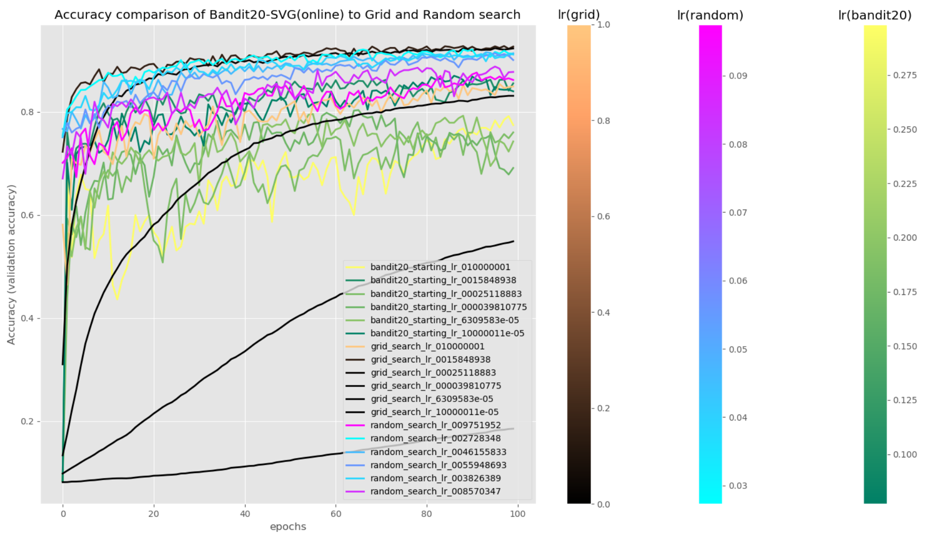


Fig. 4.12. 100 iterations with length of 10, 20 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the bandit version can adapt the badly chosen parameters in grid search into more competitive performance.

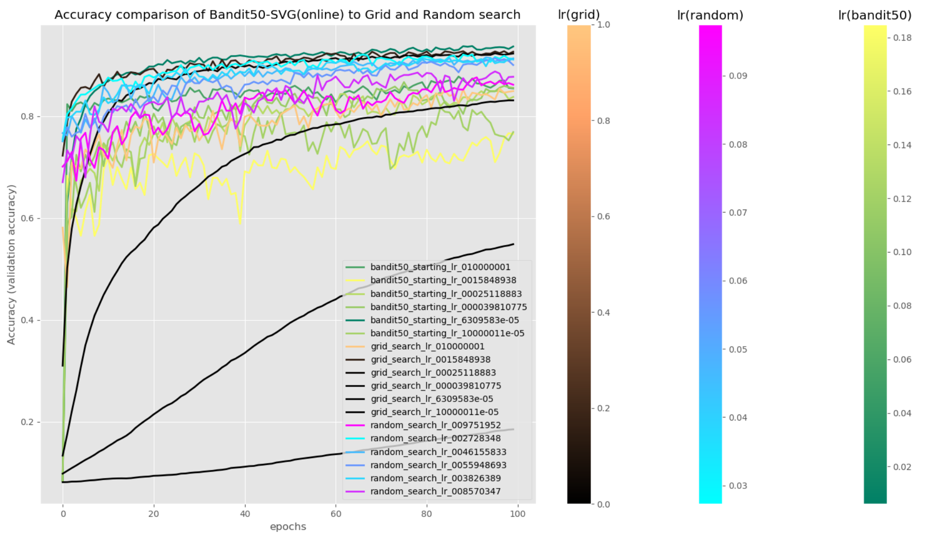


Fig. 4.13. 100 iterations with length of 50 unrolls for SVG MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the bandit version can adapt the badly chosen parameters in grid search into more competitive performance.

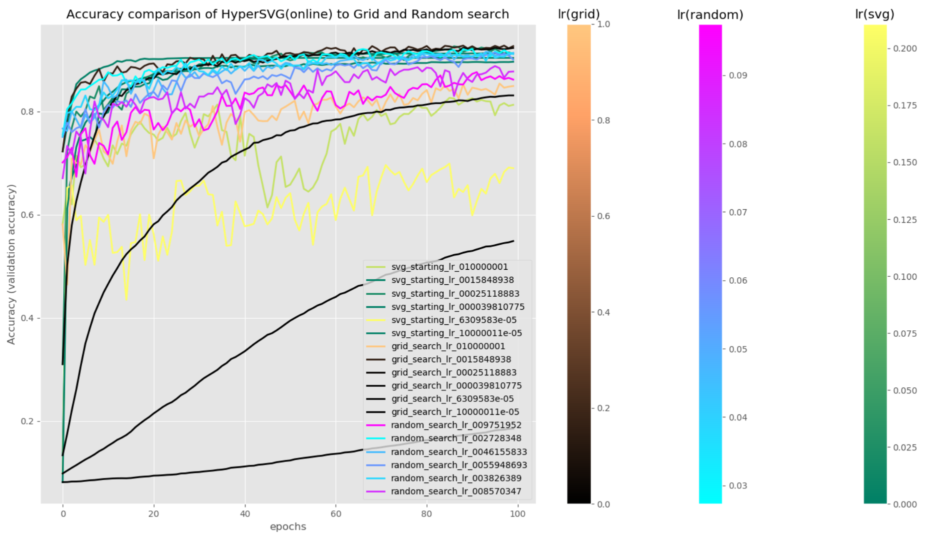


Fig. 4.14. 100 iterations with length of adaptive SVG with MNIST online task, our method is shown in yellow/green (classification validation accuracy, highest=1.0), with bad learning rates also used from grid search + SVG initialization. This shows that with badly chosen hyperparameters, the online version can adapt some of the badly chosen parameters in grid search into more competitive performance.

Chapter 5

Conclusion

In conclusion, the methods described show some promise for tuning bad learning rates, and competitive solutions (Bandit50-SVG getting 93.72% with 100+50 epochs, 100 online epochs + 50 warmup period) compared to good learning rates with BO (90.37% accuracy with 100 meta-epochs) and random search (93.1% accuracy with 100 epochs). It can also be shown that in the adaptive setting, there is much more to discuss and uncover. In particular, it will be interesting to extend these experiments to SGD with momentum for finding more than one hyperparameter, as well as other types of architectures beyond fully-connected networks, and then the layer-wise discovery of hyperparameters. The synthetic optimizer could be used to learn alongside another real optimizer on different tasks, such as robotics and reinforcement learning rather than supervised learning. It would be interesting to see whether it is still able to learn a representation that is identifiable. There is much to be said about what the output of the synthetic optimizer means mathematically. It is possible the optimizer is doing a sort of trapezoidal rule-style technique to get a learning rate approximation. For example, looking at Heun's method [Süli and Mayers, 2003]:

$$y_{n+1} = y_n + \frac{h}{2}(f_{n+1} + f_n)$$

Which, after some algebra, gets the following for the optimal step size h:

$$h = 2(f_{n+1} + f_n)^{-1}(y_{n+1} - y_n)$$

This could inductively be shown using a stochastic finite difference approximation using noise $\xi = \mathbf{s} - f(\mathbf{s}, \mathbf{a})$ that is used for noise inference in SVG to build a matrix of perturbations with respect to the current state and next state. Additionally, given the intended smoothing in the loss landscape, it would be interesting to explain the synthetic optimizer from the perspective of iterative LQR/LQG. Given that there is an interpretation of iLQR as step size and line search, it may be possible to use this to explain the backpropagation with reparameterization of the synthetic model.

References

Coax : A modular rl package, 2020. URL `https://www.microsoft.com/en-us/research/project/coax-rl/`.

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

Alfred V Aho, Jeffrey D Ullman, et al. *Principles of compiler design.* Addision-Wesley Pub. Co., 1977.

Christopher Aicher, Nicholas J Foti, and Emily B Fox. Adaptively truncating backpropagation through time to control gradient bias. In *Uncertainty in Artificial Intelligence*, pages 799–808. PMLR, 2020.

Brandon Amos, Samuel Stanton, Denis Yarats, and Andrew Gordon Wilson. On the model-based stochastic value gradient for continuous reinforcement learning. In *Learning for Dynamics and Control*, pages 6–20. PMLR, 2021.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

Yoshua Bengio, Paolo Frasconi, and Patrice Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE international conference on neural networks*, pages 1183–1188. IEEE, 1993.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Dirk Bergemann and Juuso Valimaki. Bandit problems. 2006.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *arXiv preprint arXiv:2103.12828*, 2021.

Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153:235–256, 2007.

Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

Ingrid Daubechies, Michel Defrise, and Christine De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 57(11):1413–1457, 2004.

Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

Kenji Doya. Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on neural networks*, 1(75):218, 1993.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl Θ2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Pieter Eykhoff. *System identification*, volume 14. Wiley London, 1974.

Michael Fairbank. *Value-gradient learning*. PhD thesis, City University London, 2014.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *International Conference on Machine Learning*, pages 1165–1173. PMLR, 2017.

Frauke Friedrichs and Christian Igel. Evolutionary tuning of multiple svm parameters. *Neurocomputing*, 64:107–117, 2005.

Cong Geng and Xudong Jiang. Face recognition using sift features. In *2009 16th IEEE international conference on image processing (ICIP)*, pages 3313–3316. IEEE, 2009.

Euhanna Ghadimi, Hamid Reza Feyzmahdavian, and Mikael Johansson. Global convergence of the heavy-ball method for convex optimization. In *2015 European control conference (ECC)*, pages 310–315. IEEE, 2015.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014a.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014b.

Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *Proceedings of the 27th international conference on international conference on machine learning*, pages 399–406, 2010.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

Moritz Hardt and Benjamin Recht. *Patterns, predictions, and actions: Foundations of machine learning*. Princeton University Press, 2022.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International conference on machine learning*, pages 1225–1234. PMLR, 2016.

Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International conference on artificial neural networks*, pages 87–94. Springer, 2001.

Jung-Chang Huang and Tau Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*, pages 244–248. IEEE, 1999.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Corinna Cortes, and Chris Burges. Mnist handwritten digit database, 2010.

Pierre L'ecuyer. A unified view of the ipa, sf, and lr gradient estimation techniques. *Management Science*, 36(11):1364–1383, 1990.

Sergey Levine and Vladlen Koltun. Guided policy search. In *International conference on machine learning*, pages 1–9. PMLR, 2013.

Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Ke Li and Jitendra Malik. Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*, 2017.

Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1205–1211. IEEE, 2019.

Lennart Ljung. System identification. In *Signal analysis and prediction*, pages 163–173. Springer, 1998.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020.

Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

Matthew MacKay, Paul Vicol, Jon Lorraine, David Duvenaud, and Roger Grosse. Self-tuning networks: Bilevel optimization of hyperparameters using structured best-response functions. *arXiv preprint arXiv:1903.03088*, 2019.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR, 2015a.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, volume 238, 2015b.

Niru Maheswaranathan, Luke Metz, George Tucker, Dami Choi, and Jascha Sohl-Dickstein. Guided evolutionary strategies: Augmenting random search with surrogate gradients. In *International Conference on Machine Learning*, pages 4264–4273. PMLR, 2019.

Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

Luke Metz. Exploring hyperparameter meta-loss landscapes with jax. `http://lukemetz.com/exploring-hyperparameter-meta-loss-landscapes-with-jax/`, 2021. Accessed: 2021-06-15.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*, 2016.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565. PMLR, 2019.

Luke Metz, C Daniel Freeman, Samuel S Schoenholz, and Tal Kachman. Gradients are not all you need. *arXiv preprint arXiv:2111.05803*, 2021.

Andrew Miller, Nick Foti, Alexander D'Amour, and Ryan P Adams. Reducing reparameterization gradient variance. *Advances in Neural Information Processing Systems*, 30, 2017.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *J. Mach. Learn. Res.*, 21(132):1–62, 2020.

Sen Na, Mihai Anitescu, and Mladen Kolar. An adaptive stochastic sequential quadratic programming with differentiable exact augmented lagrangians. *Mathematical Programming*, pages 1–71, 2022.

Vaishnavh Nagarajan and J Zico Kolter. Gradient descent gan optimization is locally stable. *Advances in neural information processing systems*, 30, 2017.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady an ussr*, volume 269, pages 543–547, 1983.

Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–. URL `https://github.com/fmfn/BayesianOptimization`.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

William Peebles, Ilija Radosavovic, Tim Brooks, Alexei A Efros, and Jitendra Malik. Learning to learn with generative models of neural network checkpoints. *arXiv preprint arXiv:2209.12892*, 2022.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Boris Teodorovich Polyak. *Introduction to optimization.* Science. Ch. ed. Phys.-Math. lit., 1983.

Ali Ramezani-Kebrya, Ashish Khisti, and Ben Liang. On the generalization of stochastic gradient descent with momentum. *arXiv preprint arXiv:1809.04564*, 2018.

A Rivet and Antoine Souloumiac. Introduction to optimization. In *Optimization Software, Publications Division.* Citeseer, 1987.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

R Tyrrell Rockafellar. *Convex analysis*, volume 18. Princeton university press, 1970.

Geoffrey Roeder, Luke Metz, and Durk Kingma. On linear identifiability of learned representations. In *International Conference on Machine Learning*, pages 9030–9039. PMLR, 2021.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

Francisco R Ruiz, Titsias RC AUEB, David Blei, et al. The generalized reparameterization gradient. *Advances in neural information processing systems*, 29, 2016.

Ernest Ryu, Jialin Liu, Sicheng Wang, Xiaohan Chen, Zhangyang Wang, and Wotao Yin. Plug-and-play methods provably converge with properly trained denoisers. In *International Conference on Machine Learning*, pages 5546–5557. PMLR, 2019.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Rajiv Sambharya, Georgina Hall, Brandon Amos, and Bartolomeo Stellato. End-to-end learning to warm-start for real-time quadratic optimization. *arXiv preprint arXiv:2212.08260*, 2022.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *Advances in Neural Information Processing Systems*, 28, 2015a.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015b.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Yoram Singer and John C Duchi. Efficient learning using forward-backward splitting. *Advances in Neural Information Processing Systems*, 22, 2009.

Endre Süli and David F Mayers. *An introduction to numerical analysis*. Cambridge university press, 2003.

Richard S Sutton. First results with dyna. In *Proceedings of the AAAI Spring Symposium*, 1990.

Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

Russell L Tedrake. *Applied optimal control for dynamically stable legged locomotion*. PhD thesis, Massachusetts Institute of Technology, 2004.

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*, pages 10553–10563. PMLR, 2021.

Théophane Weber, Nicolas Heess, Lars Buesing, and David Silver. Credit assignment techniques in stochastic computation graphs. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2650–2660. PMLR, 2019.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Ronald J Williams and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection science*, 1(1):87–111, 1989.

Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, 433:17, 1995.

Stephen J Wright and Benjamin Recht. *Optimization for data analysis*. Cambridge University Press, 2022.

Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. *arXiv preprint arXiv:1903.05662*, 2019.

A Steven Younger, Sepp Hochreiter, and Peter R Conwell. Meta-learning with backpropagation. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 3. IEEE, 2001.