AN ABSTRACT OF THE DISSERTATION OF

Erich Merrill III for the degree of Doctor of Philosophy in Computer Science presented on June 8th, 2022.

Title: Stake-Free Evaluations of Black-Box Optimization and Spatio-Temporal Graph Network Algorithms

Abstract approved: ____

Alan Fern

Papers proposing novel machine learning algorithms tend to present the algorithm or technique in question in the best possible light. The standard practice is generally for authors to emphasize their proposed algorithms' performance in the precise setting where it is maximally impressive, often by only fully evaluating their best known hyperparameter configuration or by only considering the problem domain that the algorithm was designed to solve. While this is an effective approach for demonstrating that their contribution is relevant and competitive, it does not allow practitioners to easily understand the presented algorithm's overall behavior and capabilities. This lack of crucial context to the presented results can especially make practical transitive comparisons of multiple algorithms across multiple papers difficult to understand or simply misleading. This issue demonstrates the need for 'stake-free' empirical evaluations of families of algorithms, in which the goal is not to demonstrate that one algorithm dominates the rest but rather to gain a more complete understanding of the overall strengths and weaknesses of each approach. In this thesis, we conduct such 'stake-free' evaluations and analyze their results for two significantly different machine learning domains: acquisition-based and partition-based black box optimization algorithms, and graph neural networks applied to spatio-temporal prediction problems. The results of this study reveal surprising facts about the optimization algorithms' relative performance, and demonstrate meaningful differences in the interpretability and capabilities of graph networks that suggest avenues for future development.

©Copyright by Erich Merrill III
June 8th, 2022
All Rights Reserved

Stake-Free Evaluations of Black-Box Optimization and Spatio-Temporal Graph Network Algorithms

by

Erich Merrill III

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 8th, 2022
Commencement June 2023

Doctor of Philosophy dissertation of Erich Merrill III presented on June 8th, 2022.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Erich Merrill III, Author

ACKNOWLEDGEMENTS

I would like to express my sincere apprecaiton of:
Alan Fern, my advisor, who always enabled me to pursue interesting and creative research I was personally interested in, and has been invaluable in guiding me through the challenging process of completing this degree
Erich and Laura Merrill, my parents, who have selflessly provided for me and encouraged me throughout my lengthy academic career
Iggy, my dog, who is an awesome friend and companion who has endlessly enthusiastically supported me in more ways than I can express

CONTRIBUTION OF AUTHORS

Xiaoli Fern provided feedback and guidance throghout the development of the work presented in Chapter 2.
Nima Dolatnia wrote the 'Problem Setup and Background' section of Chapter 2.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Publishing Incentives 1

1.2 Hyperparameter Optimization 3

1.3 Reproducability 3

2 An Empirical Study of Bayesian Optimization: Acquisition Versus Partition 6

2.1 Abstract 6

2.2 Introduction 6

2.3 Problem Setup and Background 8

2.4 Description of Algorithms 9

2.4.1 Acquisition-Based BO 10

2.4.2 Partitioning-Based Optimization 11

2.4.3 Partition-Based Bayesian Optimization 16

2.5 Experimental Setup 20

2.5.1 Hyperparameter Selection 20

2.5.2 Software Platform 20

2.5.3 Black Box Functions 21

2.5.4 Evaluation Process 22

2.6 Empirical Results 24

2.6.1 General Trends of Regret Curves 24

2.6.2 Pairwise Comparisons 27

2.6.3 Other Results and Discussion 30

2.7 Summary 36

3 Stake-Free Evaluation of Graph Networks for Spatio-Temporal Processes 39

3.1 Abstract 39

3.2 Introduction 39

3.3 Spatio-Temporal Processes 43

3.4 Benchmark Domains 44

3.4.1 Starcraft II 45

3.4.2 Weather Nowcasting 47

3.4.3 Traffic Prediction 49

3.5 Models 50

TABLE OF CONTENTS (Continued)

Page

3.5.1 Graph Encoding 51
3.5.2 GraphConv 53
3.5.3 Interaction Networks 54
3.5.4 PointConv 56
3.5.5 PointConv with Attention 57
3.6 Hyperparameter Selection and Training 58
3.6.1 Checkpoint Learning Rate Test 59
3.7 Results 62
3.7.1 Overall Performance 62
3.7.2 Starcraft II 65
3.7.3 Traffic Prediction 71
3.7.4 Weather Nowcasting 73
3.8 Evaluation Platform 76
3.9 Conclusion 77

4 Conclusion 79

5 Tutorial 82
5.1 Problem Class 82
5.2 Network Class 84
5.3 Experiment Definition File 86
5.4 Training Engine 88
5.5 Evaluation Scripts 88
5.6 Performing Experiments 88

Bibliography 90

LIST OF FIGURES

Figure		Page
2.1	Regret curves for of each algoritm on many functions	25
2.2	Worst-case regret curves	26
2.3	Regret curves across differing β values	28
2.4	Regret in terms of wall clock time	32
2.5	Sparse ABO demonstration	33
3.1	Example SC2 scene	46
3.2	The results of three styles of learning rate test applied to the same network architecture. Old is the classic learning rate test, Random is the random checkpoint test, and Demo is the 'poorly trained' checkpoint test. Note the clear area of improvement for the Old and Random tests, while Demo has a much less clear area of improvement and 'explodes' at a much lower learning rate.	62
3.3	Average training loss plots for all instances of each model trained on the SC2 dataset.	66
3.4	Filters learned by a PointConv model trained on the Starcraft 2 prediction problem.	68
3.5	PointConv Attention visualization. The 'main' unit is circled in pink. Each unit in the neighborhood is highlighted with a solid circle colored according to its attention weights. Each unit icon's shape indicates its unit type, while its color indicates its team.	69
3.6	Plots showing the average prediction loss for each model type for each timestep offset between T+1 and T+12.	71
3.7	Average training loss plots for all instances of each model trained on the traffic prediction dataset.	72
3.8	Each model's predicted traffic speed throughout an entire day for individual sensors. Black line is the target signal.	74
3.9	Average training loss plots for all instances of each model trained on the weather dataset.	74

LIST OF TABLES

Table Page

2.1 Algorithm properties 19

2.2 Objective function properties 22

2.3 Pairwise comparison between algorithms 27

2.4 Pairwise comparison, maximum samples 33

2.5 Pairwise comparison of 2D objective functions 34

2.6 Pairwise comparison of >2D objective functions 35

3.1 Spatio-Temporal Process Notation 44

3.2 Description of feature vector representation of each unit in the Starcraft domain. The variable is used as a prediction query target if the Target? column is checked. 47

3.3 Description of feature vector representation of each station in the weather domain. The variable is used as a prediction query target if the Target? column is checked. Metadata? is checked if the value is static and associated with the weather station in question. 48

3.4 Graph Encoding Notation 51

3.5 GraphConv notation 54

3.6 Interaction network notation 55

3.7 Loss Table 66

3.8 Table showing the 25%, 50%, 75% percentile, and mean prediction error for each model in MPH. Bolded entries indicate the model beat the baseline's performance. 73

3.9 Table demonstrating the change in each model's test loss as station dropout is increased. Columns show the median and mean loss as the test-time station dropout is raised from 0% to 50%. Models with Drop20 appended to their name were trained with 20% station dropout. 75

LIST OF ALGORITHMS

Algorithm — Page

1 Bayesian optimization Process 12
2 Simultaneous Optimistic Optimization 14
3 Simultaneous Optimistic Optimization (simplified) 15
4 Locally Oriented Global Optimization 16
5 Bayesian Multi-Scale Optimistic Optimization 18
6 Infinite-Metric GP Optimization 19

7 53
8 53
9 Checkpoint LR Test 63
10 Calculate LR Ranges 64
11 Learing Rate Calculation 64

Chapter 1: Introduction

One of the primary purposes of publishing academic research is to objectively communicate the results of research efforts to other practitioners. This allows them understand the properties of the works being presented without having to perform that research effort themselves. A collection of publications from different authors proposing novel advancements in a field should ideally present a coherent, consistent view of the current state of the field when read as a whole. However, this ideal vision of academic publishing does not always play out in practice. The incentives and pressures for authors surrounding the publishing process can result in research being developed and presented in such a way that maximizes its publishability rather than maximizing the work's utility to readers. This persistent issue shows that there a need for 'stake-free' evaluations of families of algorithms. A 'stake-free' evaluation is one in which the goal is to fairly examine the relevant algorithms' relative performance and characteristics across many different settings, rather than demonstrate that one specific algorithm dominates the others. In this thesis, we discuss our encounters with misunderstandings which may have been caused by these research publishing incentive issues and perform thorough 'stake-free' evaluations of two different classes of algorithms. Our evaluations revealed interesting and sometimes surprising results that could not have been inferred from the publications presenting the original algorithms. To enable others to build on our work in this direction, we release the software platform we developed for our most recent evaluation. This platform is explicitly meant to enable designing, running, and evaluating repeatable experiments, including training every model and reproducing every plot included in Chapter 3 with minimal user effort.

1.1 Publishing Incentives

The competitive nature of the modern publication process imposes a strong incentive for authors to present their work in the best possible light. Most high-profile venues have very limited space to present publications compared to the number of viable sub-

missions, requiring that they be very selective when deciding on which submitted works to accept. As a result, most works that are accepted are ones that appear to be 'high impact', usually meaning it clearly demonstrates a significant advance in methodology or raw performance over other state-of-the-art or well-regarded approaches in the relevant field. Authors also have limited space with which to present their work. As a result, they tend to focus on demonstrating their proposed ideas in the specific settings where those proposals appear the most impressive compared to the state of the art. This is an appropriate and effective way to show the strengths of a given approach or algorithm, which is necessary considering the constraints and incentives described above. Regardless, works that exclusively focus on the strengths of the proposed algorithm or technique can hinder readers from gaining an understanding of for which settings the proposed technique is appropriate (or inappropriate) to apply. This potential for misunderstanding may be amplified if readers look at multiple works proposing new algorithms within a given area, and transitively compare their performance to each other based on their reported results relative to well-established baseline algorithms.

We encountered this issue when attempting to learn about the space of recently-proposed black-box optimization algorithms. A recently proposed family of algorithms based on space partitioning techniques reported promising results when compared to much more computationally expensive Bayesian optimization algorithms. Specifically, the partitioning-based optimization algorithms were demonstrated to be competitive with the state of the art Bayesian optimization algorithms despite running with orders of magnitude less resources. From reading the papers, one would assume that this line of work demonstrates that Bayesian approaches must be 'wasting effort' since the partitioning approaches produce better results in less time and space. We investigated this topic by performing an evaluation of each type of algorithm in question across a wide, varied range of objective functions, as described in Chapter 2. Our results conflicted with the intuition we had gained by reading the works proposing partitioning algorithms: we found that the partitioning approaches were dominated by the best-performing Bayesian approach in almost every case. This discrepancy between the findings of our stake-free evaluation and those inferred from reading the papers present the partitioning methods in the best possible light shows the confusion that this issue can cause to those attempting to follow the literature.

1.2 Hyperparameter Optimization

Many machine learning algorithms' (especially deep learning models') performance is influenced by the hyperparameters used to train the model or otherwise define its behavior. In some cases, the effect of selecting a good or bad hyperparameter configuration on the model's observed performance can dominate the performance characteristics that are due to the proposed algorithm's structure and operation. However, hyperparameter selection processes are rarely explicitly recorded in publications which evaluate these models. Commonly, the authors will describe the method used to search for a performant hyperparameter setting for their model (e.g. 'hyperparemeter optimization was performed via grid search') and list the final values each hyperparameter was set to. This omits the crucial information of how much effort was put into the grid search for each model. If the authors' proposed model received hundreds of hours of hyperparameter search time over months of development, while the models they compare against were only given a brief hyperparameter search shortly before the submission deadline, this could have a significant impact on the results they report.

To address this issue, in the evaluation in Chapter 3 we explicitly describe a repeatable hyperparameter search procedure which is used to determine the training hyperparameters for each instance of each model on each problem type. This procedure quickly experimentally determines a range of effective values for the hyperparameter in question. While the approach is less thorough than a full grid search, it is orders of magnitude faster which enables us to tune the hyperparameters for each model on each problem automatically with little effort. Most importantly, this procedure ensures that all models included in the evaluation receive approximately equal amounts of hyperparameter tuning effort. This ensures that observed differences in the models' performance is due to the structural differences between the models rather than the amount of tuning effort they received.

1.3 Reproducability

Both of these potential pitfalls could be worked around if the publications provided usable code that allowed others to easily reproduce their reported results. This would allow interested parties to observe details about the training or evaluation process that

may not have been fully included in the publication, such as exact hyperparameter search details. Additionally, providing a working evaluation platform allows interested parties to build on and extend it, potentially to evaluate the proposed algorithm on a different set of domains than was emphasized in its paper.

Unfortunately, many publications do not provide any usable code at all, making it nearly impossible to independently validate their claims. Works that do provide code frequently only provide a handful of python scripts which define the models being evaluated and the dataset implementation. While this is helpful to those wishing to build on the core aspect of the work in some other way, a handful of python scripts is usually not sufficient to validate all the results presented in the paper the code is associated with.

We intentionally take the opposite approach to make our work as reproducible and approachable as possible. To perform the evaluation in Chapter 3, we developed a software platform which explicitly defines and controls every aspect of the evaluation. The software platform includes: generic implementations of all models and datasets used in the evaluation; experiment definition files which configure the datasets and models and define how they should be trained; scripts to recreate the automated hyperparameter search procedure; scripts to produce the plots and tables included in Chapter 3; and documentation to explicitly instruct users on how to use the provided tools to recreate the results shown in the evaluation. This allows practitioners to run the same experiments we did, and hopefully fully reproduce our results with minimal effort. Additionally, we hope that the software platform developed for this evaluation will be extended to further develop this line of fully-reproducible research.

An Empirical Study of Bayesian Optimization: Acquisition Versus Partition

Erich Merrill, Alan Fern, Xiaoli Fern, Nima Dolatnia

Journal of Machine Learning Research
Vol. 22(4), 1–25

Chapter 2: An Empirical Study of Bayesian Optimization: Acquisition Versus Partition

2.1 Abstract

Bayesian optimization (BO) is a popular framework for black-box optimization. Two classes of BO approaches have shown promising empirical performance while providing strong theoretical guarantees. The first class optimizes an acquisition function to select points, which is typically computationally expensive and can only be done approximately. The second class of algorithms use systematic space partitioning, which is much cheaper computationally but the selection is typically less informed. This points to a potential trade-off between the computational complexity and empirical performance of these algorithms. The current literature, however, only provides a sparse sampling of empirical comparison points, giving little insight into this trade-off. The primary contribution of this work is to conduct a comprehensive, repeatable evaluation within a common software framework, which we provide as an open-source package. Our results give strong evidence about the relative performance of these methods and reveal a consistent top performer, even when accounting for overall computation time.

2.2 Introduction

We consider the problem of optimizing an unknown function f by selecting experiments that each specify an input x and return a response $f(x)$. Given an experimental budget, the goal is to select a sequence of experiments in order to find an input that approximately maximizes f. An effective approach to this problem is Bayesian optimization (BO) [Brochu et al., 2010], which assumes a Bayesian prior in order to quantify the uncertainty over f via posterior inference. This posterior can then be used to bias the experiment selection in a variety of ways.

Perhaps the most traditional and widely used BO approach is acquisition-based BO (ABO) [Kushner, 1964, Jones, 2001]. The key idea is to define an acquisition func-

tion in terms of the posterior, which is then optimized at each iteration to select the next experiment. Two commonly used acquisition functions are Expected Improvement (EI) [Mockus, 1994] and Upper Confidence Bound (UCB) [Srinivas et al., 2010], which have both been shown to be practically effective. In addition, UCB has been shown to have probabilistic guarantees on performance under the assumption that the acquisition function can be perfectly optimized at each iteration. However, both UCB and EI are non-convex, making optimization costly and inexact for higher dimensional functions. Thus, in practice, the theoretical results for ABO do not hold and the selection of each experiment can be computationally expensive.

A recent alternative approach to ABO completely avoids the optimization of acquisition functions. These approaches are inspired by the simultaneous optimistic optimization [Munos et al., 2014] (SOO) algorithm, which is a non-Bayesian approach that intelligently partitions the space based on observed experiments to effectively balance exploration and exploitation of the objective. We will refer to SOO and algorithms derived from it as partition-based global optimization (PGO) algorithms. A key feature of SOO is that it provides finite time performance guarantees with minimal assumptions about the objective function's properties. SOO does not, however, exploit the potential benefits of posterior inference. This has led to variants of SOO, such as BaMSOO [Wang et al., 2014] and IMGPO [Kawaguchi et al., 2015], that integrate posterior inference to better direct SOO's exploration. We will refer to these alternative approaches that integrate posterior inference with PGO approaches as partitioning-based BO (PBO) algorithms. Importantly, the PBO algorithms are able to select each experiments using significantly less computation compared to ABO. At the same time, they maintain probabilistic variants of SOO's performance guarantees.

ABO uses more computation per experiment selection than the partition-based approaches, so one might expect ABO to make higher quality decisions and outperform partition-based methods given the same number of experiments. Thus, ABO may have an advantage for application where the number of experiments is fixed and not limited by the runtime of experiment selection. For example, when experiments involve lengthy wet lab trials or expensive simulations, the computation time required for selecting experiments may be negligible. Alternatively, there are applications where the expense of ABO can limit the number of experiments compared to partition-based methods. For example, if experiments correspond to running fast physics simulations, then the higher

computational cost of ABO may be a bottleneck that will lead to running fewer experiments in a fixed time horizon and potentially perform worse than partitioning methods.

The above intuitions suggest a potential trade-off between acquisition-based and partition-based methods, however, the literature provides little guidance regarding this trade-off. Most comparisons between ABO, PBO, and PGO have been either indirect or on a sparse set of problems. For example, the PBO algorithm BaMSOO was shown [Wang et al., 2014] to outperform both SOO and selected ABO algorithms. However, this was on a modest number of problems and for Bayesian hyper-parameters that were selected on a per problem basis, which is not always a realistic real-world use scenario. Later, a non-Bayesian PGO algorithm, LOGO [Kawaguchi et al., 2016], was shown to outperform SOO and BaMSOO, which combined with the prior BaMSOO results was taken as evidence for preferring LOGO over ABO approaches. More recent work on IMGPO [Kawaguchi et al., 2015] shows further benefits of PBO over ABO on a small set of problems. However, the ABO implementations used in those evaluations appear to yield inferior performance to the experience of others. If taken at face value, these prior results suggest that PBO approaches dominate ABO approaches despite their much lower computational cost. Yet most applications of BO are still using ABO approaches.

The primary contribution of this paper is to conduct a more thorough evaluation of these approaches with the aim of understanding when and if one should be preferred. We conduct this investigation using a common software framework, which will be publicly available and allow for complete reproducibility. Importantly, we do not introduce new algorithms or variants of existing algorithms in order to avoid the potential appearance of bias in our evaluation. Our results yield fairly consistent observations across a variety of test problems, shedding light on the relative performance of some of key representative acquisition-based and partition-based algorithms.

2.3 Problem Setup and Background

We consider optimizing an unknown function $f : \mathcal{X} \rightarrow \mathbb{R}$ where $\mathcal{X}$ is a compact d-dimensional subset of $\mathbb{R}^d$. The black box function f does not necessarily have a closed form but can be evaluated at any point in the domain. Running an experiment x allows us to observe the outcome $y = f(x)$ at some cost. Given a budget that constrains the number of experiments, the goal is to find a input x that approximately maximizes f.

Without any constraints on f, there is no way to guarantee a near optimal value will be found. Thus, prior theoretical and practical work on this black-box optimization problem typically makes some form of smoothness assumption on f [Munos et al., 2014]. Under such assumptions it is possible to design more intelligent optimization procedures that prune away and prioritize parts of the input space based on previously observed experiments.

Bayesian optimization (BO) formalizes smoothness via a Bayesian prior over f, which is often represented by a Gaussian Process (GP) [Williams and Rasmussen, 2006]. In this work, we will focus on BO using GP priors. A GP is a collection of possibly infinite random variables where any subset is multivariate Gaussian distributed. One advantage of using a GP prior over f is that for any experiment x there is a closed form for the mean and standard deviation of its response $f(x)$, conditioned on the previously observed experiments. A GP is completely specified by its mean function, $m(x)$ and its covariance function, $k(x_1, x_2)$. A common choice in the BO literature, which we follow in our experiments, is to select the prior mean to be zero; that is, $m(x) = 0$ for all $x \in \mathcal{X}$. In addition, we will use the squared-exponential kernel with a width hyper-parameter θ_i for each input dimension, which is a widely used kernel in the BO and more generally GP literature. Given a set of observed experiments, we select the hyper-parameters via automatic relevance determination (SE-ARD) [MacKay, 1998], which optimizes the marginal likelihood with respect to the hyper-parameters.

To evaluate the performance of each algorithm on each objective function, we calculate its regret after t objective observations $r_t = f(x^*) - f(x_t^+)$, where x^* is the optimum point and x_t^+ is the best point the algorithm has observed so far after making t observations. Note that regret for ABO methods is frequently calculated by setting x_t^+ to the point that maximizes its GP's mean function after making t observations. Since partitioning-based methods cannot provide a similar prediction, to ensure a fair comparison between the different results we exclusively consider points that the algorithms have observed directly when calculating their regret.

2.4 Description of Algorithms

This section describes the algorithms that are included in our empirical study, which covers three algorithmic classes. First, we describe two widely used algorithms from the

class of acquisition-based BO (ABO) algorithms. Second, we describe two partitioning-based global optimization (PGO) algorithms, which do not use a Bayesian prior, but have strong theoretical guarantees on regret. Third, we describe two partitioning-based Bayesian optimization (PBO) algorithms, which incorporate a Bayesian prior into the PGO approaches. A comparison of the most relevant properties of all the algorithms is provided in Table 2.1.

2.4.1 Acquisition-Based BO

Assume we have the observations $D_t = \{x_{1:t}, y_{1:t}\}$ and we are interested in the distribution of the output y_* of a test point (another experiment) x_*. Since we have a GP with prior mean zero, the joint distribution of $y_{1:t}$ and y_* can be written as:

$$\begin{bmatrix} y_{1:t} \\ y_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(x_{1:t}, x_{1:t}) & K(x_{1:t}, x_*) \\ K(x_*, x_{1:t}) & K(x_*, x_*) \end{bmatrix}\right)$$

where $K(\cdot, \cdot)$ is the corresponding covariance matrix using the covariance function element-wise. Having the joint distribution, the conditional distribution of y_* given the observed data can be derived as:

$$\begin{aligned} y_* | D_t, x_* &\sim \mathcal{N}\left(\mu(x_*), \sigma^2(x_*)\right) \\ \mu(x_*) &= K(x_*, x_{1:t})[K(x_{1:t}, x_{1:t})]^{-1} y_{1:t} \\ \sigma^2(x_*) &= K(x_*, x_*) - K(x_*, x_{1:t})[K(x_{1:t}, x_{1:t})]^{-1} K(x_{1:t}, x_*) \end{aligned}$$

ABO uses an acquisition function as a selection heuristic that evaluates each candidate point based on its mean and variance. Acquisition functions are generally designed so that their high values correspond to potentially high values of the objective function. Starting with some initial observed data points, the covariance matrix of the GP is calculated. Using the posterior mean and variance of each candidate data point, the value of the acquisition function can be obtained. The point that maximizes this value is then selected for observation. The experiment and its observed output get added to the data set and this process can be repeated until a specified horizon or budget is exhausted. Since optimizing the hyperparameters of the kernel function helps fit a more accurate GP model to the data, the kernel parameters are tuned periodically during the iterative process. Pseudocode for the ABO algorithm is provided in Algorithm 1.

In this paper, we consider two of the most popular acquisition functions, namely Expected Improvement (EI) [Mockus et al., 1978] and Gaussian Process Upper Confidence Bound (GP-UCB) [Srinivas et al., 2010]. At any time step $t+1$ with prior experiments $x_{1:t}$, the acquisition function EI measures the expected improvement with respect to the current best previously observed objective value $f(x_t^+)$, where $x_t^+ = \arg\max_{x_i \in x_{1:t}} f(x_i)$. Under Gaussian Processes, EI can be analytically computed as follows:

$$A_{EI}(x) = (\mu(x) - f(x^+))\Phi(\tfrac{\mu(x)-f(x^+)}{\sigma(x)}) + \sigma(x)\phi(\tfrac{\mu(x)-f(x^+)}{\sigma(x)})$$

where $\mu(x)$ and $\sigma^2(x)$ are the predicted mean and variance of point x. And, Φ and ϕ are the CDF and PDF of the standard normal distribution, respectively.

More recently, GP-UCB with provable cumulative regret bounds with high probability was proposed as a BO algorithm [Srinivas et al., 2010]. Formally, the algorithm defines the following acquisition function:

$$A_{UCB}(x) = \mu(x) + \beta_t^{1/2}\sigma(x)$$

where β_t are appropriate coefficients that balance exploitation against exploration.

The regret bounds provided by Srinivas et al. [2010] depend on the assumptions on the kernel spectrum and their correspondingly defined β_t. In practice, the researchers choose the form of β_t based on their domain knowledge. For instance, although different from β_t used in their theoretical results, Kandasamy et al. [2015b] use $\beta_t = 0.2d \log 2t$ in their experiments where d is the number of dimensions. Since we wish to evaluate the performance of these algorithms in a setting where we have no pre-existing domain knowledge that can be used to effectively set these values, we chose to instead select a value for β_t that others had reported experimental success with. In this work, we use $\beta_t = 2\log(|\mathcal{X}|t^2\pi^2/6\delta)$ which was found to be effective by Srinivas et al. [2010] with $\delta = 0.1$, where the smaller value of δ provides a higher probability of achieving the regret bound. The size of $\mathcal{X}$ is obtained assuming each dimension is discretized into 1000 points.

2.4.2 Partitioning-Based Optimization

Partitioning-based approaches attempt to use an implicit upper bound on the values contained within cells of varying sizes of the objective function's space. Once a procedure has identified the cells with the most promising upper bound, the algorithms can

Algorithm 1 Bayesian optimization Process

Input: D_0, N_{up} ▷ D_0 is the initial data; the hyperparameters get updated every N_{up} iterations

1: **for** t = 1,2, ... **do**
2: **if** N_{up} is a divisor of t **then**
3: Update the kernel hyperparameters
4: **end if**
5: Given D_{t-1}, specify the GP
6: Select x_t by optimizing the acquisition function $x_t = \arg\max_{x \in \mathcal{X}} A(x|D_{t-1})$
7: Observe y_t, the output of x_t
8: Augment data $D_t = \{D_{t-1}, (x_t, y_t)\}$
9: **end for**

then direct their search focusing on those promising ones by refining them into smaller, hopefully more informative cells. These newly created cells require additional objective observations so that they can be assigned a value representative of the space each cell encloses.

This loop of selecting promising cells and refining them with additional function evaluations is what drives the partitioning algorithms' search, and the predictable behavior of the growth of the partitioning tree is what permits the algorithms their theoretical guarantees. In this section, we consider two PGO algorithms with strong theoretical guarantees, one of which (LOGO) has claimed in prior work [Kawaguchi et al., 2016] to be competitive with ABO.

2.4.2.1 Simultaneous Optimistic Optimization (SOO)

The simultaneous optimistic optimization algorithm is a partitioning-based global optimization (PGO) algorithm introduced by Munos et al. [2014]. Two advantages of SOO over other algorithms are its weak assumptions about the function being optimized and its relative speed. To meet its preconditions SOO only requires that there exists some semi-metric ℓ over the objective such that $f(x^*) - f(x) \leq \ell(x^*, x) \forall x$ (where x^* is the optimum point of the objective function) and that the space can be partitioned into cells with monotonically decreasing 'size' according to ℓ. However, it is not required to know or estimate this metric for the algorithm. Additionally, since SOO is computa-

tionally very simple and does not require the optimization of any auxiliary functions to operate, it can consistently select points for observation in a near-zero amount of time when executed on modern computer hardware. A high-level pseudocode description of the algorithm is provided at Algorithm 2.

SOO works by partitioning its objective function's domain using a tree τ. Each node in the tree represents a hyper-rectangle (cell) in the space and is assigned the observed value of the objective function at the center point of the hyper-rectangle. This single objective sample and the size of the cell is used to reason about the potential function values that could be contained within the cell's region. We use $x_{h,i}$ to represent the center point of the ith node at depth h in τ, and $g(x_{h,i})$ for its associated observed value from the objective function f. In each iteration, SOO selects a set of promising cells for refinement, where each selected cell is partitioned into three equal-sized sub-cells as its children[1]. If a cell has not been refined, it is referred to as a leaf.

Initially, the partition tree contains only a single leaf node that covers the entire input space. The center of the input space is sampled and the resultant function value is assigned to this node. From that point onward, SOO iteratively selects the leaf node at each depth in the tree with the highest upper bound according to the implied semi-metric ℓ for further partitioning (refinement), as long as the selected leaf's upper bound is greater than the upper bound of any leaf of larger cell sizes in τ during that iteration. Importantly, we do not need to explicitly compute the upper bounds in order to select the leaf with the highest upper bound.

Specifically, since all leaves at a given depth of the tree have cells of the same size, finding the leaf with the highest upper bound at a fixed depth h according to ℓ is simply a matter of finding the leaf at that depth with the highest center value $g(x_{h,i})$. Furthermore, we know that if any larger-sized leaf has a greater center value than a smaller-sized leaf, the larger one's upper bound according to ℓ must be greater due to its larger size and higher known value at its center. Therefore, we can safely ignore the smaller-sized leaf. As such, as we iterate through each depth level of the partitioning tree and select the leaf with the highest center value at each depth, we only need to consider those leaves whose center value is higher than that of any larger-sized leaves. This will guarantee that the leaf with the maximum upper bound according to ℓ is always selected for expansion regardless of the true definition of ℓ.

[1]Tie breaking is discussed in Section 2.5.4.

SOO takes as parameter a depth-limiting function $h_{\max}(n)$ that defines the maximum depth to which the partitioning tree can be expanded after n cell refinements. In previous work [Munos et al., 2014] [Kawaguchi et al., 2016], this function has always been defined as $h_{\max}(n) = \sqrt{n}$ so we ignore this parameter and assume that this common setting of $h_{\max}$ is a part of SOO itself.

At a high level, the SOO algorithm can be viewed as consisting of two parts: the cell selection process (lines 8-15), and the cell expansion process (lines 16-23). Since the remaining partitioning-based algorithms involve modifying one or both of these processes, we define a simplified version of the algorithm at Algorithm 3, which is used to enable a more clear definition of the derivative algorithms.

Algorithm 2 Simultaneous Optimistic Optimization

```
1: Initialize τ_h = ∅ ∀h ≥ 0                  ▷ τ_h contains the set of leaves at depth h
2: τ_0 = {x_{0,0}}                  ▷ x_{0,0} is the center of the objective function f's domain
3: g(x_{0,0}) = f(x_{0,0})
4: n = 1
5: loop
6:     E = ∅
7:     v_max = −∞
8:     for h = 0 ... min(depth(τ), h_max(n)) do
9:         (h, i) = arg max_{x_{h,j} ∈ L_h} g(x_{h,j}), where L_h contains all leaves of τ at depth h
10:        if g(x_{h,i}) ≥ v_max then
11:            E = E ∪ {x_{h,i}}
12:            v_max = g(x_{h,i})
13:            n = n + 1
14:        end if
15:    end for
16:    for each cell x_{h,i} in E do
17:        Subdivide x_{h,i} into its three resultant children cells x_{h+1,j_1} ... x_{h+1,j_3}
18:        for each child cell x_{h+1,j} do
19:            g(x_{h+1,j}) = f(x_{h+1,j})
20:            τ_{h+1} = τ_{h+1} ∪ x_{h+1,j}
21:        end for
22:    end for
23: end loop
24: return arg max_{x_{h,i} ∈ τ} g(x_{h,i})
```

Algorithm 3 Simultaneous Optimistic Optimization (simplified)

1: Initialize the partitioning tree τ with an observation from the center
2: **loop**
3: $\quad E = \text{SELECTCELLS}_{\text{SOO}}$
4: $\quad \text{EXPANDCELLS}_{\text{SOO}}(E)$
5: **end loop**
6: **return** $\arg\max_{x_{h,i} \in \tau} g(x_{h,i})$

2.4.2.2 Locally Oriented Global Optimization (LOGO)

Locally Oriented Global Optimization (LOGO) [Kawaguchi et al., 2016], as described in Algorithm 4, is a modification to SOO that introduces a local bias parameter w to achieve a finer control of the exploration-exploitation behavior of the algorithm. In particular, instead of selecting the best leaf in the partitioning tree that meets the refinement criteria at *each depth level* for expansion, LOGO uses the same selection process across disjoint sets of w adjacent depth levels. With this approach, when w is set to a high value LOGO will be more inclined to spend its observation budget exploiting a smaller number of the most attractive cells rather than exploring those in nearby depths that are not as attractive, but would have been expanded by SOO. Note that when $w = 1$, the behavior of LOGO and SOO are identical.

LOGO sets the value of w according to the local bias schedule hyperparameter W, which must be a list of positive integers and should be monotonically increasing. At the end of each iteration of the algorithm, if the value of the best cell observed during that step is greater than that of the previous step w is set to the next value in W. Otherwise, w is set to the previous value in the schedule. The intuition behind this design is that when the algorithm is succeeding (that is, successively observing higher and higher objective values) it should continue exploiting, which is enabled by increasing w. When the opposite situation occurs (that is, the algorithm repeatedly fails to find values that improve on the previous step's observations), the algorithm should instead fall back to more exploration-focused behavior to attempt to more quickly locate the next area that may offer further improvement. A fixed-size schedule is used to avoid edge cases where the algorithm would frequently 'get lucky' early on, causing the value of w to consistently increase to the point where the algorithm is then 'stuck' exhibiting exploitative behavior for the remainder of the optimization.

On some objective functions, this adaptive behavior leads to significantly better performance than SOO's static approach without violating any of the original algorithm's performance guarantees.

Algorithm 4 Locally Oriented Global Optimization

1: **function** SELECTCELLS$_{\text{LOGO}}$
2: $E = \emptyset$
3: $v_{\max} = -\infty$
4: **for** $k = 0 \ldots \lfloor \min(\text{depth}(\tau), h_{\max}(n))/w \rfloor$ **do**
5: $D = \tau_{kw} \cup \tau_{kw+1} \cup \ldots \tau_{kw+w-1}$
6: $i = \arg\max_{i: x_{h,i} \in L_D} g(x_{h,i})$, where L_D contains all the leaves in D
7: **if** $g(x_{h,i}) \geq v_{\max}$ **then**
8: $E = E \cup \{x_{h,i}\}$
9: $v_{\max} = g(x_{h,i})$
10: $n = n + 1$
11: **end if**
12: **end for**
13: **return** E
14: **end function**

2.4.2.3 Dividing Rectangles (DIRECT)

DIRECT [Jones et al., 1993] is a popular partitioning-based optimization algorithm which, while very similar in structure to SOO, is meant to improve on Lipschitzian optimization rather than achieve certain theoretical regret bounds. Its primary differences from SOO are modified cell selection conditions, the lack of the concept of a 'depth limit', and a more thorough expansion procedure when expanding cells which are hyper-rectangles for which all edges are of the same size.

We are considering DIRECT in this evaluation primarily as a benchmark for other partitioning-based optimization algorithms due to its ubiquity and efficacy in practice, so we will not provide a more detailed description of the DIRECT algorithm itself.

2.4.3 Partition-Based Bayesian Optimization

While the PGO methods exhibit predictable average-case performance, they clearly have room for improvement. Their treatment of the objective as a true black box results in the PGO algorithms frequently observing regions of the objective that should seem obviously unpromising. However, without any ability to predict the behavior of the objective at a given point, the algorithm's only choice is to spend a function evaluation to prove to itself that the region in question is 'bad' and can be ignored. Even when a poor region is ignored, though, it is still guaranteed to be expanded once it is the only remaining leaf at its depth level, resulting in even more 'unnecessary' function observations in potentially irrelevant areas of the objective.

Here we examine two methods that attempt to improve PGO optimization by incorporating a GP prior into the procedure to use past observations to better direct the expansion and selection of cells.

2.4.3.1 Bayesian Multi-Scale Optimistic Optimization (BaMSOO)

The Bayesian Multi-Scale Optimistic Optimization [Wang et al., 2014] algorithm (Algorithm 5) is a SOO-based algorithm that uses the same selection strategy, but avoids evaluating a point during expansion if the point is deemed unpromising according to the posterior. Specifically, this is done by comparing the upper confidence bound (UCB) derived from the prior at the locations which are about to be observed to the best function value observed in the tree. If the UCB of a cell's center is smaller than the best value observed, we know that the cell is unlikely to contain the optimum. Therefore, Instead of using up a function observation assigning a value to a cell that is unlikely to be further expanded, the cell is assigned the value of the lower confidence bound of its center point according to the prior. This allows the algorithm to continue as expected by effectively 'ignoring' the unpromising cell instead of spending an objective evaluation to assign a value to the probably-unimportant cell.

2.4.3.2 Infinite-Metric GP Optimization (IMGPO)

IMGPO (Algorithm 6) builds on BaMSOO by further taking advantage of the information encoded in the prior by also using it to guide the cell selection process. In addition

Algorithm 5 Bayesian Multi-Scale Optimistic Optimization

```
1: function ExpandCells_BAMSOO(E)
2:     D = the observations in τ not marked as GP-based
3:     for each cell x_{h,i} in E do
4:         Refine x_{h,i} into its three resultant children cells x_{h+1,j_1} ... x_{h+1,j_3}
5:         for each child cell x_{h+1,j} do
6:             if U(x_{h+1,j}|D) ≥ f* then
7:                 g(x_{h+1,j}) = f(x_{h+1,j})
8:             else
9:                 g(x_{h+1,j}) = L(x_{h+1,j}|D)
10:                Mark g(x_{h+1,j}) as GP-based
11:            end if
12:            τ_{h+1} = τ_{h+1} ∪ x_{h+1,j}
13:        end for
14:    end for
15: end function
```

to the requirement that any cell suitable for refinement must have a value greater than the value of any larger cell, IMGPO also requires that the cell being considered must contain a UCB greater than the value of any smaller (that is, deeper in the tree) cell.

To determine the UCBs that a cell contains, IMGPO builds a subtree within the cell down to a fixed depth using the same node refinement rules. Instead of observing the value of the function at the center of each cell in the subtree, the UCB of each cell's center point is calculated instead. The highest value in this temporary UCB subtree is then considered to be the best UCB contained within the cell in question. With this approach, IMGPO determines whether or not a cell is worth expanding based on the prior information about the upper bound on the value of its potential children, further allowing the algorithm to ignore areas of the objective that seem unpromising.

This change is also the most significant departure from SOO or any of the partitioning-based algorithms. Every other partitioning algorithm's expansion criteria for a leaf is solely based on the properties of the cells above it in the tree. It follows that, for these algorithms, the top-most leaf of the tree *must* be expanded regardless of its value, leading to predictable grid-search-like behavior as the minimum depth of any leaf in the tree grows. Since IMGPO also evaluates cells for refinement based on how they compare to smaller cells, a cell can remain unexpanded while being the only leaf at its depth

Algorithm	Sample Selection	Fits GP	Optimizes Acquisition Function	Sample 'depth limit'
SOO	Grid-aligned	No	No	Yes
LOGO	Grid-aligned	No	No	Yes
DIRECT	Grid-aligned	No	No	No
BaMSOO	Grid-aligned	Yes	No	Yes
IMGPO	Grid-aligned	Yes	No	No
BO	Unrestricted	Yes	Yes	N/A

Table 2.1: Comparison of several properties shared by the algorithms being compared.

level. This seemingly minor change prevents IMGPO from exhibiting the grid-search-like behavior that can be seen in the PGO algorithms' results. Accordingly, the depth limit function $h_{\max}(n)$ that is used in every other SOO-derived algorithm is no longer necessary for IMGPO.

Algorithm 6 Infinite-Metric GP Optimization

1: **function** $\text{SELECTCELLS}_{\text{IMGPO}}$
2: $E = \emptyset$
3: $D =$ the observations in τ not marked as GP-based
4: $v_{\max} = -\infty$
5: **for** $h = 0 \ldots \text{depth}(\tau)$ **do**
6: $(h, i) = \arg\max_{x_{h,j} \in L_h} g(x_{h,j})$, where L_h contains all leaves of τ at depth h
7: **if** $x_{h,i}$ would be selected by $\text{SelectCells}_{\text{SOO}}$ **then**
8: Refine $x_{h,i}$ into a subtree S
9: $U^* = \max_{s_{h',j} \in S} U(s_{h',j}|D)$
10: $f^* =$ the greatest value among all non-GP-based cells at depths $h' > h$
11: **if** $U^* \geq f^*$ **then**
12: $E = E \cup \{x_{h,i}\}$
13: $n = n + 1$
14: **end if**
15: **end if**
16: **end for**
17: **end function**

2.5 Experimental Setup

When reviewing the reported results of the algorithms evaluated in this paper, we found that they were rarely evaluated in a true black-box setting. Instead, the algorithms'

hyperparameters were tuned after the fact or alternatively the results were presented with non-standard metrics to best demonstrate the strengths of the proposed approach.

These differences in procedure led to confusing and seemingly contradictory results being presented, which spurred the development of this work. Our goal is to directly compare the performance of each algorithm in a setting where they have minimal knowledge of the objective being optimized with the hopes of resolving some of the confusion one could suffer from trying to collectively reason about the previous works' results.

2.5.1 Hyperparameter Selection

To avoid the issue of ex post facto algorithm tuning, we attempted to separate the process of selecting values and settings for any algorithm's hyperparameters from the evaluation itself. To accomplish this, we looked to previous works that had used the algorithms or the papers that had introduced the algorithms themselves and duplicated settings that were suggested or reported as having good empirical performance. To accomplish this, we selected all the hyperparameters for each algorithm before executing them on any objective functions. The hyperparameter values were selected by duplicated them from the paper that had introduced the algorithm or other works that reported good empirical performance with certain settings. The specific values considered for this evaluation are described in the algorithms descriptions in Section 2.4.

2.5.2 Software Platform

To perform these experiments, we built a custom, extendable C++ framework that executes easily-repeatable evaluations of arbitrary black-box optimization algorithms. The source code is publicly available at `https://github.com/Eiii/opt_cmp`.

Our framework includes C++ implementations of all partitioning-based algorithms (SOO, LOGO, BaMSOO, IMGPO), while a modified[2] version of the bayesopt library [Martinez-Cantin, 2014] provides the implementation of BO that we use in our evaluation. Although the authors of some partitioning-based algorithms provide implementations of their algorithm, we chose to use custom implementations of the algorithms since each

[2] Modifications were made to expose previously purely internal functions and data structures as required for the BO-aware algorithms to function. The optimization behavior of the library itself is unchanged.

derivative PGO and PBO algorithm can be trivially implemented as a minor extension of a simpler partitioning-based algorithm. This choice to define all the partitioning-based algorithms from the same base behavior also prevents one algorithm from incorrectly appearing more or less effective due to differing design decisions or assumptions made by each author of an implementation.

To ensure a fair comparison between the ABO and PBO methods, the PBO implementations repurpose bayesopt's internal GP functionality when they require the use of a GP.

2.5.3 Black Box Functions

We chose to exclusively use synthetic benchmark functions as objectives for this evaluation to ensure that each benchmark could sufficiently evaluate each objective with the time and resources available. The benchmark functions were selected with the goals of including functions with a wide range of quantitative and qualitative properties and including functions that the authors of the partitioning-based methods used to evaluate their algorithms.

The LOGO paper evaluates its algorithm on the sin_2, branin, Rosenbrock, Hartmann3, Hartmann6, and Shekel (with $m = 5, 7, 10$) functions [Kawaguchi et al., 2016] with varying dimensionality and parameterizations when possible. IMGPO and BaMSOO are also both evaluated on a subset of these objectives [Kawaguchi et al., 2015] [Wang et al., 2014], so we chose to include all of them in our experiments to allow for a more direct comparison between results. Additionally, we chose to include the Rastrigin, Schwefel, and Ackley test functions [Molga and Smutnicki, 2005] in our evaluation to provide more variety among the objectives.

The Rastrigin, Schwefel, Ackley, and Rosenbrock objective functions [Molga and Smutnicki, 2005] can be defined with arbitrary dimensionality D. Functions with this property are useful in allowing us to better determine the effect that the dimensionality of the objective has on each algorithm's performance. For these objectives, we evaluated each algorithm on each function with D =2, 4, 6, and 10.

A summary of the properties of each objective function can be found in Table 2.2.

Function Name	Dimensionality	Description/Category	Evaluated in Previous Works:
Sin	2	Simple	SOO, LOGO, IMGPO
Branin	2	Simple	LOGO, BaMSOO, IMGPO
Rastrigin	2, 4, 6, 10	Many local maxima	
Schwefel	2, 4, 6, 10	Many local maxima	
Ackley	2, 4, 6, 10	Many local maxima	
Rosenbrock	2, 4, 6, 10	Valley shaped	LOGO, BaMSOO, IMGPO
Hartmann	3, 6		LOGO, BaMSOO, IMGPO
Shekel ($m = 5, 7, 10$)	4		LOGO, BaMSOO, IMGPO

Table 2.2: Summary of the properties of each objective function used.

2.5.4 Evaluation Process

To evaluate the set of algorithms we ran 70 iterations consisting of executing each optimization algorithm on each function to a horizon of at least 500 samples. For further comparisons, we extended the horizon for SOO, LOGO, DIRECT, and Random to 10,000 samples and configured BaMSOO and IMGPO to run for the average amount of wall-clock time the ABO algorithms expended across all their runs on each objective. These extended horizons were chosen considering the cost to run experiments and the runtime of each algorithm. Each individual evaluation of one algorithm on one objective function was run on one core of a c4.4xlarge Amazon EC2 instance. The decision to run 70 iterations in total was made beforehand by estimating what was feasible within the compute time available for these experiments.

2.5.4.1 Objective Function Randomization

Since SOO and LOGO are deterministic algorithms, it is possible that the algorithm could get unusually lucky or unlucky on some objective functions. This luck could manifest as making a series of observations and resultant partitioning decisions that happen to result in exceptional behavior that is not representative of the algorithm's average-case performance on objective functions with similar properties. Additionally, since any SOO-derived algorithm must first observe the exact center of the hyper-rectangle that defines the objective function's domain, they are all guaranteed to trivially and immediately achieve zero regret on any objective function whose maximum is at its center. Since neither of these difficulties of evaluating SOO-derived algorithms is relevant to the

algorithms' performance on real-world problems, we chose to avoid them by randomizing certain properties of the objective functions used in our evaluation.

The cell refinement procedure shared by SOO and its derived algorithms simply splits evenly along the dimension along which the cell has the longest edge. This approach frequently results in ties between dimensions which must be resolved using a tie breaking procedure. The tie breaking procedure used by the partitioning algorithms is simply to assign a fixed priority order to the dimensions and split the higher-priority dimensions first in the case of a tie. It is plausible that this approach could lead to situations where the assignment of this arbitrary ordering could have a significant impact on the algorithms' performance on some objective functions. To mitigate the possibility that this behavior causes certain algorithms to exhibit non-representative behavior, we randomize the tie break dimension ordering at the beginning of each run of each algorithm.

To allow us to evaluate the performance of the partitioning algorithms on objective functions in which the optimum point is in its center, we randomly shrink the bounds of the hyper-rectangle that define the objective's domain such that the optimum point is still guaranteed to be contained within the new bounds.

These methods of randomizing objective functions are applied to every algorithm in a predictable and repeatable way identical to all algorithms so that, for example, one algorithm will not be advantaged by getting 'easier' domains on certain functions than another algorithm.

2.5.4.2 Hyperparameter Settings

To evaluate the algorithms in a true black-box setting, we use one set of hyperparameters for each algorithm across every objective function. To avoid manually tuning the algorithms with the goal of maximizing their performance on our specific problem set, when possible we use the same hyperparameter settings the authors of each method used during their evaluation process.

We use the implementation of BO from the bayesopt [Martinez-Cantin, 2014] library. For our evaluations, we use the squared exponential kernel with automatic relevance detection in which the parameters are estimated using maximum total likelihood. Each ABO run is started with three randomly chosen initial points (which count against the algorithm's budget), and the GP's parameters are re-estimated from the observed data

every second ABO iteration.

The EI criteria is parameter-free, but UCB requires us to define a value for β. This value is frequently tuned per-objective, but we instead set it to $\beta_t = 2\log\left(|D|t^2\pi^2/6\delta\right)$, which Srinivas et al. [2010] found to be effective. In this case, t is the number of function observations made so far and δ is a constant set to 0.5.

SOO has no parameters to set. LOGO only requires a list of integers to use as the adaptive schedule W for its local bias parameter. In this work, we set $W = 3, 4, 5, 6, 8, 30$ to duplicate the value chosen by Kawaguchi et al. [2016] in their work that introduced the LOGO algorithm.

BaMSOO and IMGPO both require the use of a GP prior to estimate the upper bound on the objective function's value at certain nodes' locations. Using the GP's estimated mean function μ and standard deviation function σ, we define the LCB and UCB as $\mu(x) \pm B_N\sigma(x)$. For BaMSOO we define $B_N = \sqrt{2\log\left(\pi^2N^2/6\eta\right)}$ as suggested by Wang et al. [2014], and for IMGPO we define $B_N = \sqrt{2\log\left(\pi^2N^2/12\eta\right)}$ as suggested by Kawaguchi et al. [2015]. In both cases, N is the total number of times the GP was used to evaluate a node, and the constant η is set to 0.05 to duplicate the value used in the experimental results included by Kawaguchi et al. [2015] and Wang et al. [2014].

2.6 Empirical Results

In this section we present the results of our experiments using simple regret as our primary performance metric. In particular, after each run of an optimization algorithm, the simple regret is the difference between the best observed outcome and the optimal value. The reported regrets are averages over 70 independent trials.

2.6.1 General Trends of Regret Curves

We first consider the performance of the algorithms when given the same experimental budget. Figure 2.1 shows the average regret curves for each algorithm on the Ackley, Rastrigin, Rosenbrock, and Schwefel functions with dimensionality $D = \{4, 6, 10\}$, as well as the 2D functions Sin2, Branin, and Rosenbrock2. For all but the two-dimensional functions, there appears to be a clear ordering of the different classes of algorithms: ABO-EI overwhelmingly dominates the remaining algorithms at most sample counts

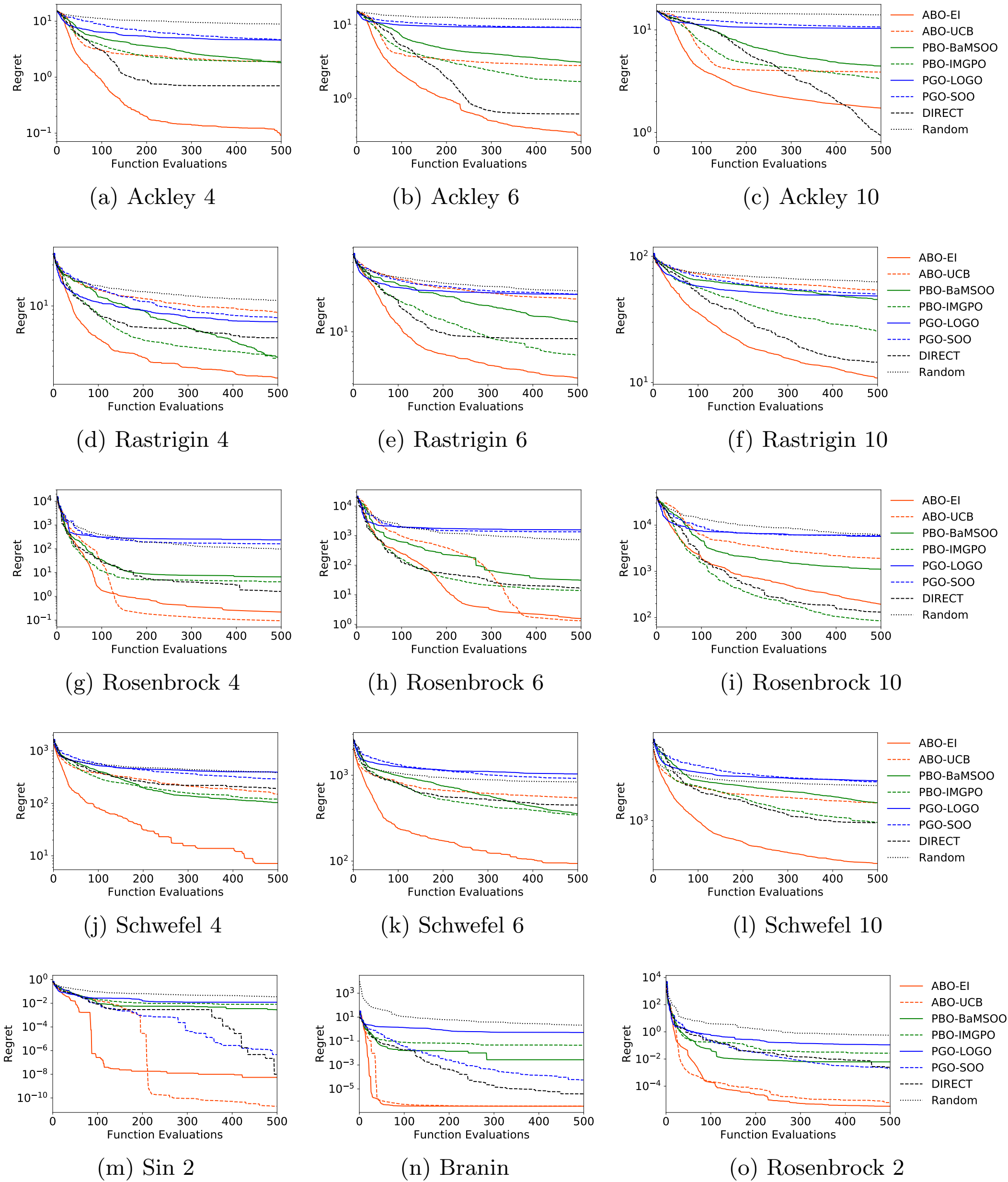
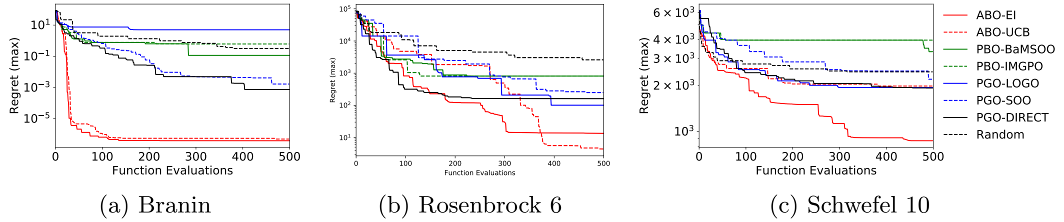


Figure 2.1: Regret curves for each algorithm on a variety of functions. Each curve shows the regret at each time step averaged across 70 randomized runs. Error bars have been omitted for readability. Statistical significance is considered in Section 2.6.2.



(a) Branin (b) Rosenbrock 6 (c) Schwefel 10

Figure 2.2: Worst-case regret curves for each algorithm. Each curve shows the maximum regret at each time step across 70 randomized runs.

with only a few exceptions. The PBO methods are consistently the next closest algorithm class to ABO-EI's performance but are rarely able to achieve the same regret. The PGO algorithms keep up with the best-performing algorithms initially, but their performance quickly seems to 'flat line' and they have difficulty significantly improving further throughout the remainder of the sample budget.

In practice, it is important to understand the worst case performance that we may expect to encounter. Figure 2.2 shows the worst-case regret for each algorithm observed across the 70 trials at each sample size. Results are shown for three objective functions that are representative of the overall results. Overall, we found that the ordering of the worst-case performance of the algorithms was approximately similar to their relative average-case performances. Most notably, the PGO methods consistently have an inferior worst-case performance than Random on higher dimensional functions.

One difference compared to the average case results is that PBO methods occasionally have significantly poorer worst-case performance than the PGO methods. This is likely due to behavior we have observed in which the GP prior used in PBO methods can be 'tricked' by a few unlucky unappealing function samples near the optimum value that causes the algorithm to ignore what should ideally be seen as a promising region of the function. Since the PGO methods are 'dumber' in that they do not take advantage of a prior and are more likely to fall back to grid-search-like behavior, they do not suffer from this failure case.

	ABO-EI	ABO-UCB	PBO-BaMSOO	PBO-IMGPO	PGO-LOGO	PGO-SOO	DIRECT	Random
ABO-EI		12 - 3 - 8	12 - 0 - 11	10 - 1 - 12	18 - 1 - 4	18 - 1 - 4	3 - 1 - 19	23 - 0 - 0
ABO-UCB	3 - 12 - 8		0 - 2 - 21	0 - 4 - 19	12 - 1 - 10	11 - 1 - 11	0 - 6 - 17	20 - 0 - 3
PBO-BaMSOO	0 - 12 - 11	2 - 0 - 21		0 - 2 - 21	9 - 1 - 13	10 - 1 - 12	0 - 4 - 19	23 - 0 - 0
PBO-IMGPO	1 - 10 - 12	4 - 0 - 19	2 - 0 - 21		10 - 1 - 12	10 - 1 - 12	0 - 2 - 21	22 - 0 - 1
PGO-LOGO	1 - 18 - 4	1 - 12 - 10	1 - 9 - 13	1 - 10 - 12		1 - 2 - 20	0 - 13 - 10	12 - 1 - 10
PGO-SOO	1 - 18 - 4	1 - 11 - 11	1 - 10 - 12	1 - 10 - 12	2 - 1 - 20		0 - 13 - 10	14 - 0 - 9
DIRECT	1 - 3 - 19	6 - 0 - 17	4 - 0 - 19	2 - 0 - 21	13 - 0 - 10	13 - 0 - 10		22 - 0 - 1
Random	0 - 23 - 0	0 - 20 - 3	0 - 23 - 0	0 - 22 - 1	1 - 12 - 10	0 - 14 - 9	0 - 22 - 1	

Table 2.3: Pairwise comparison of each algorithm across all objective functions at $t = 500$ samples. Each cell displays the number of 'wins', 'losses', and 'ties' between the algorithm in the row and the algorithm algorithm in the column (for example, the bottom-left-most cell shows that Random beat ABO-EI 0 times, lost 23 times, and tied 0 times).

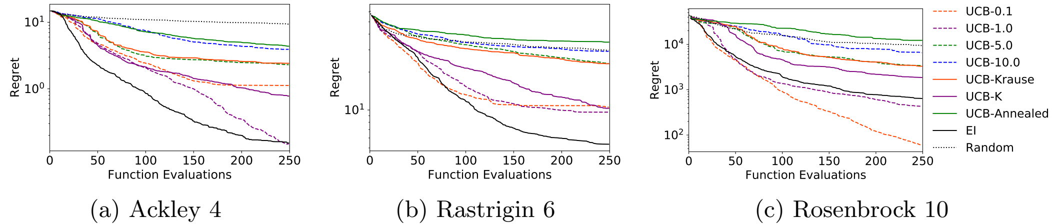
2.6.2 Pairwise Comparisons

To gain a better understanding of how the algorithms compare to one another across all the objective functions, we compiled Table 2.3 which shows a pairwise comparison of all the algorithms across every objective after 500 samples. To generate the table, we calculated the 95% confidence interval of the mean regret of each algorithm on each objective function at the specified sample size. We considered one algorithm to beat another on a given function if the upper bound of the confidence interval of the mean of its regret was less than the lower bound of the 'challenger' algorithm. If the confidence intervals of the two algorithms' performance overlapped, then they were considered to tie. We use this information to examine differences between different classes of algorithms, across different types of objective functions, and different sample budgets.

ABO-UCB versus ABO-EI. ABO-EI and ABO-UCB, despite being similar algorithms, performed very differently from one another compared to other algorithm pairs in the same class. EI wins over UCB over half of the time, and loses only 3 times. While EI consistently beats every other algorithm, UCB seems to be able to at best tie with the PBO algorithms, and could only beat the PGO algorithms approximately half of the time.

Note that this result is based on our selected method for updating the UCB hyperparameter β. It is important to recall that this choice was based on our best effort to select from the variety of β selection methods considered in previous work during a preliminary validation period—we know of no better overall method for β selection.

It is very likely that one could tune β on a per problem basis to outperform EI, however, this tuning methodology would not result in algorithm behavior that is repre-



(a) Ackley 4 (b) Rastrigin 6 (c) Rosenbrock 10

Figure 2.3: Regret curves for a variety of different β values for the UCB algorithm. ABO-EI and Random are provided as benchmarks.

sentative of its efficacy in many real-world scenarios. To test this possibility, we selected a number of additional settings for the β parameter and compared UCB's performance using these settings to our previous results and ABO-EI's. Selected results from these experiments are shown in Figure 2.3.

We compared a variety of constant values for β (labeled as UCB-x, where x is the constant value), another suggested schedule that was used by Kandasamy et al. [2015a] (labeled as UCB-K), an annealing schedule provided by the Bayesopt library [Martinez-Cantin, 2014] (labeled as UCB-Annealed), the original schedule we used in the previous results (labeled as UCB-Krause), and ABO-EI.

The results show that no one β setting appears to be able to match EI's performance more than occasionally. The UCB-K setting does appear to generally do better than the UCB-Krause setting we evaluated, but not to the extent that it would significantly change our results had we selected that schedule instead.

Simple constant values of β perform surprisingly well on some objectives, with 0.1 and 1.0 occasionally beating EI. However, the lack of a consistent winner suggests that the β value needs to be manually tuned to perform well on each objective function. It's unclear what knowledge about the objective's properties is required to determine which β values will be effective or if it's feasible to make that determination during the optimization of an unknown objective function.

On the other hand, EI is parameter free and therefore does not require such a selection process. Thus, these results suggest that ABO-EI is more appropriate to use in a true black-box setting where the objective's properties are unknown, or that further work should be done on determining how to adaptively set ABO-UCB's β parameter to

improve its black-box performance.

Acquisition Functions versus Partitioning. If we consider ABO-EI as the representative of the ABO class of algorithms, it appears that ABO approaches dominate partitioning-based approaches across the board. PBO methods frequently seem to tie with EI's performance, but the fact that they are almost never able to actually outperform EI when given an equal number of objective function observations suggests that EI is the more effective approach in this study.

PBO versus PGO. As should be expected, augmenting SOO to enable it to take advantage of Bayesian inference significantly improves the PBO methods' performance over that of the PGO algorithms. While PBO and PGO methods frequently tie with each other in our comparison, both PBO methods only lose once to SOO and LOGO.

Comparison to Random. The ABO and PBO algorithms are consistently able to beat Random, with only UCB and IMGPO ever tying with it. PGO methods are not as dominant versus Random, with both SOO and LOGO winning over Random only slightly more often than they're able to tie, and LOGO losing on 1 function.

Comparison to DIRECT. DIRECT clearly performs significantly better than either SOO or LOGO in our results. It ties with ABO-EI approximately as often as the other PGO methods lose to ABO-EI, and most notably never loses to either SOO or LOGO while consistently outperforming them.

This is surprising, considering how similar DIRECT is to SOO in its structure and operation. The most notable differences are DIRECT's lack of a 'depth limit' when refining its partitioning tree over the objective's domain, and its lack of a uniform selection of cells to refine among all depths of its partitioning tree.

While these properties are what give SOO its theoretical guarantees, the former may prevent SOO from quickly exploiting an area of the objective that's known to be good while the latter forces it to 'waste' observations exploring areas that might otherwise seem unappealing.

In practice, this appears to suggest that DIRECT's lack of these potential limitations allows it to significantly outperform the PGO algorithms. Since DIRECT's runtime is comparable to SOO and LOGO's, our results do not suggest a potential use case for which SOO or LOGO would be more appropriate to select as an optimization algorithm over DIRECT.

2.6.3 Other Results and Discussion

Large Numbers of Experiments. Since PGO algorithms are orders of magnitude faster than ABO and PBO algorithms, and PBO algorithms can be at least an order of magnitude faster than ABO algorithms, it is interesting to consider the performance of each algorithm when the limiting factor is time rather than a sample budget. We present the performance of the algorithms when run for the maximum number of samples considered in this study in Table 2.4. This simulates the extra objective observations that the faster methods would be able to collect within a fixed time budget, assuming the evaluation of the objective is fast and cheap. These sample sizes were selected based on computational feasibility of conducting the 70 trials for each algorithm on each function.

We allowed the PGO algorithms 10K samples, the ABO algorithms 500 samples, and the PBO algorithms were allowed to run for the same average wall clock run time as the ABO algorithms. This approach resulted in approximately 25k samples on average per problem for BaMSOO, and 4k samples for IMGPO. We empirically observe that both SOO and LOGO improve in comparison to the lower-sample-size ABO and PBO approaches, contrasting the pairwise comparisons in Table 2.3 and Table 2.4 reveals that the difference is less significant than would be expected considering the PGO methods are allowed ten to twenty times as many samples as the others. On the other hand, DIRECT seems to be able to take advantage of the extra samples—whereas at 500 samples it was beaten handily by ABO-EI, after 10,000 samples it never loses to ABO-EI and wins against it more than any other algorithm.

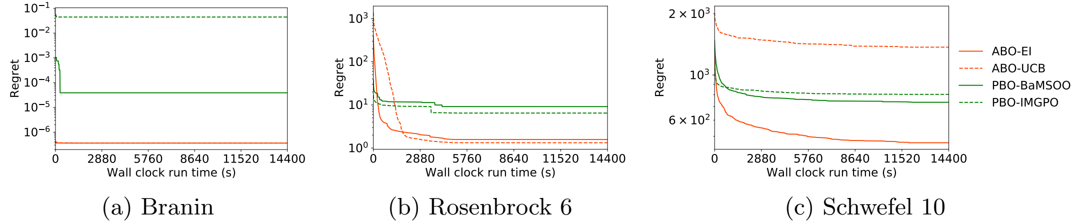
This suggests that the depth-limiting behavior, which is necessary to maintain SOO and LOGO's exploration behavior in earlier stages of the optimization, may be hurting their ability to take advantage of large numbers of objective samples. The resulting coarse grid new samples are restrained to by the limited depth of the refinement tree may be limiting the extent to which the partitioning algorithms are able to quickly 'hone in' on promising regions of the function once they've been identified. Modifying the algorithms' depth limiting $h_{\max}$ function to allow for greater tree expansion at very large number of samples could prevent this behavior, although it may also reduce performance by allowing PGO to spend too many samples exploiting attractive-looking regions earlier on. Investigations during our validation period did not reveal a superior choice for $h_{\max}$ overall.

Comparing PBO versus ABO, we see that both PBO algorithms reduce the number of losses to ABO-EI at the larger sample size, they do not improve the number of wins over ABO-EI. The performance of the PBO algorithms improves slightly against ABO-UCB, however, ABO-UCB was already not very competitive when PBO was allowed only 500 samples.

To more directly compare the performance of PBO and ABO algorithms, we removed the sample budget and applied a time horizon to the PBO methods that allows them to execute for the same average runtime as the ABO method would on the same objective. Due to ABO's need to optimize its increasingly expensive to evaluate acquisition function to select each point, at large numbers of observations ABO runs slowly enough that the PBO methods which are not burdened by this responsibility are able to achieve up to thousands of extra samples of the objective in the same amount of time. We show a sample of representative results obtained with varying time horizon in Figure 2.4. As our previous results indicate, despite being allowed up to thousands of extra samples of the objective over ABO, the PBO methods are unable to translate those extra samples into a better final regret.

We also evaluated the algorithms' relative performance when the objective function takes more or less time to execute. Our data set contains the wall clock execution time (e) and sample count (t) for each step in each optimization. Since we know the benchmark functions take near-zero time to evaluate, we assume that the wall clock time represents the amount of time the algorithm itself has consumed up until that point. We can then derive a data set simulating the algorithms' performance on objective functions that takes time o to evaluate by replacing each wall clock time/sample count pair (e, t) in the data set with an updated pair $(e + ot, t)$. We hoped to discover that by adjusting the 'objective complexity' we would could find a trade-off where the rapid sampling pace of the partitioning algorithms would outperform the slower optimization-bound approach of the acquisition algorithms. However, we did not follow through on this analysis once we observed that the partitioning methods were unable to reliably outperform the acquisition methods even when given orders of magnitude more samples.

'Flexibility' of ABO. Although we're using one set of hyperparameters on one implementation of ABO for our evaluation, it's important to consider that ABO has many implicit and explicit parameters that can be modified to achieve a desired performance/computation trade off. For example, by tweaking the time allowed by the



(a) Branin (b) Rosenbrock 6 (c) Schwefel 10

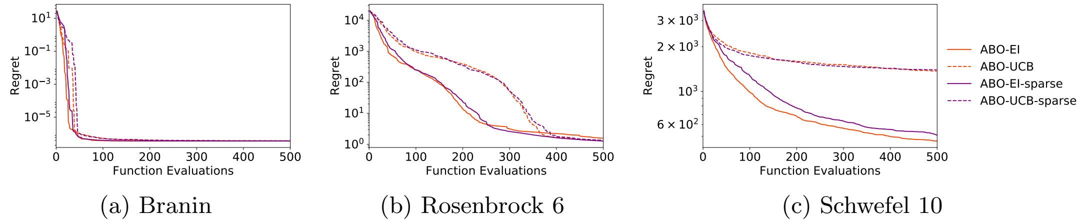
Figure 2.4: Regret curves for PBO and ABO methods in terms of *wall clock time* rather than number of function evaluations executed. Each curve shows the regret at each time step averaged across 70 randomized runs.

inner optimization algorithm that optimizes the GP's parameters or the algorithm that optimizes the acquisition function, ABO can be made to run much faster with some unknown penalty to its performance that depends on the objective's sensitivity to the underlying GP's accuracy or the accuracy of the acquisition function optimization. We do not explicitly explore the details of those minor tweaks to ABO in this work. Instead, to examine the impact on ABO's performance when it is not allowed the same amount of wall clock execution time, we adjust the number of samples in-between GP parameter optimizations.

Figure 2.5 compares the performance of the instance of ABO we present in this paper to a 'sparse' instance that is identical to the implementation of ABO we used other than that it re-learns GP parameters from the observed data one-eighth as frequently. That is, it performs the GP optimization every sixteen observations instead of at every second observation.

Despite an expected decrease in execution time by a factor of eight, the performance of this 'lighter' instance of ABO does not appear to perform significantly differently than our 'standard' ABO. If ABO-EI did not already appear to be the dominant algorithm in this evaluation, instead presenting a version of it that achieves similar results in one-eighth the time would make it appear even more attractive when compared to the other methods. However, since making this change would not affect our conclusions, we do not consider this 'lighter' ABO in our results other than to demonstrate how flexible ABO approaches can be made to be.

Dependence on Dimensionality. Because ABO is known to be effective in problems with low dimensionality, we intentionally compared the algorithms on objective



(a) Branin (b) Rosenbrock 6 (c) Schwefel 10

Figure 2.5: Regret curves for each instance of ABO on a representative selection of functions. Each curve shows the regret at each time step averaged across 70 randomized runs. Error bars have been omitted for readability.

	ABO-EI	ABO-UCB	PBO-BaMSOO	PBO-IMGPO	PGO-LOGO	PGO-SOO	DIRECT	Random
ABO-EI		12 - 3 - 8	15 - 0 - 8	6 - 1 - 16	13 - 3 - 7	13 - 4 - 6	0 - 9 - 14	23 - 0 - 0
ABO-UCB	3 - 12 - 8		9 - 0 - 14	1 - 6 - 16	4 - 2 - 17	4 - 2 - 17	0 - 12 - 11	14 - 2 - 7
PBO-BaMSOO	0 - 15 - 8	0 - 9 - 14		0 - 6 - 17	1 - 6 - 16	0 - 6 - 17	0 - 13 - 10	3 - 2 - 18
PBO-IMGPO	1 - 6 - 16	6 - 1 - 16	6 - 0 - 17		7 - 0 - 16	7 - 0 - 16	0 - 2 - 21	13 - 0 - 10
PGO-LOGO	3 - 13 - 7	2 - 4 - 17	6 - 1 - 16	0 - 7 - 16		1 - 0 - 22	1 - 10 - 12	11 - 3 - 9
PGO-SOO	4 - 13 - 6	2 - 4 - 17	6 - 0 - 17	0 - 7 - 16	0 - 1 - 22		1 - 10 - 12	12 - 3 - 8
DIRECT	9 - 0 - 14	12 - 0 - 11	13 - 0 - 10	2 - 0 - 21	10 - 1 - 12	10 - 1 - 12		23 - 0 - 0
Random	0 - 23 - 0	2 - 14 - 7	2 - 3 - 18	0 - 13 - 10	3 - 11 - 9	3 - 12 - 8	0 - 23 - 0	

Table 2.4: Pairwise comparison of each algorithm across all objective functions when algorithms are run for the maximum number of samples considered in this paper. PGO, DIRECT, and Random are allowed 10,000 samples, ABO is allowed 500 samples, and PBO is allowed the same wall-clock run time as the ABO algorithms.

	ABO-EI	ABO-UCB	PBO-BaMSOO	PBO-IMGPO	PGO-LOGO	PGO-SOO	DIRECT	Random
ABO-EI		1 - 1 - 4	1 - 0 - 5	2 - 0 - 4	3 - 1 - 2	3 - 1 - 2	0 - 1 - 5	6 - 0 - 0
ABO-UCB	1 - 1 - 4		0 - 0 - 6	0 - 0 - 6	2 - 1 - 3	3 - 1 - 2	0 - 1 - 5	6 - 0 - 0
PBO-BaMSOO	0 - 1 - 5	0 - 0 - 6		0 - 0 - 6	1 - 1 - 4	0 - 1 - 5	0 - 1 - 5	6 - 0 - 0
PBO-IMGPO	0 - 2 - 4	0 - 0 - 6	0 - 0 - 6		1 - 1 - 4	0 - 1 - 5	0 - 1 - 5	5 - 0 - 1
PGO-LOGO	1 - 3 - 2	1 - 2 - 3	1 - 1 - 4	1 - 1 - 4		1 - 2 - 3	0 - 3 - 3	4 - 1 - 1
PGO-SOO	1 - 3 - 2	1 - 3 - 2	1 - 0 - 5	1 - 0 - 5	2 - 1 - 3		0 - 3 - 3	6 - 0 - 0
DIRECT	1 - 0 - 5	1 - 0 - 5	1 - 0 - 5	1 - 0 - 5	3 - 0 - 3	3 - 0 - 3		6 - 0 - 0
Random	0 - 6 - 0	0 - 6 - 0	0 - 6 - 0	0 - 5 - 1	1 - 4 - 1	0 - 6 - 0	0 - 6 - 0	

Table 2.5: Pairwise comparison of each algorithm across all objective functions with dimension $D = 2$ at $t = 500$ samples.

functions with a wide range of dimensionality to better understand where and why the partitioning methods' performance differs from ABO's. Table 2.5 shows the same comparison restricted to only two-dimensional objective functions, while Table 2.6 only shows the results for the objectives with dimension greater than two.

For the two-dimensional functions we evaluated, it appears that the difference between the algorithms' performance is not as pronounced. While ABO-EI still appears to be the most effective, the PGO approaches frequently tie with its results and even beat it on one occasion each.

For objectives with dimensions greater than two, ABO is much more dominant. ABO-EI only loses to a partitioning method once, and wins against the other algorithms much more frequently than it ties with them. Although we expect ABO's performance to degrade at higher dimensions, it seems that PGO's performance is hit much harder by the increase in objective dimensions. This may be because having more dimensions along which to split the cells means the partitioning tree over the space would be much deeper before PGO can start to refine its search towards promising areas (since each cell must be split along each dimension in some order, regardless of the values observed). This 'refinement' shortcoming, combined with the depth-limiting behavior of PGO algorithms, is likely what causes the performance of PGO algorithms to degrade so consistently on high-dimensional functions.

Comparison to Previous Results. Several aspects of our results appear to differ from some of those reported in previous works. The experimental results presented by Wang et al. [2014] suggest that BaMSOO performs better than both SOO and ABO-UCB while being significantly faster than ABO-UCB. Those presented by Kawaguchi et al. [2015] suggest that IMGPO performs better than BaMSOO, SOO, and UCB-EI while being significantly faster than BaMSOO. Later, Kawaguchi et al. [2016] report

	ABO-EI	ABO-UCB	PBO-BaMSOO	PBO-IMGPO	PGO-LOGO	PGO-SOO	DIRECT	Random
ABO-EI		11 - 2 - 4	11 - 0 - 6	8 - 1 - 8	15 - 0 - 2	15 - 0 - 2	3 - 0 - 14	17 - 0 - 0
ABO-UCB	2 - 11 - 4		0 - 2 - 15	0 - 4 - 13	10 - 0 - 7	8 - 0 - 9	0 - 5 - 12	14 - 0 - 3
PBO-BaMSOO	0 - 11 - 6	2 - 0 - 15		0 - 2 - 15	8 - 0 - 9	10 - 0 - 7	0 - 3 - 14	17 - 0 - 0
PBO-IMGPO	1 - 8 - 8	4 - 0 - 13	2 - 0 - 15		9 - 0 - 8	10 - 0 - 7	0 - 1 - 16	17 - 0 - 0
PGO-LOGO	0 - 15 - 2	0 - 10 - 7	0 - 8 - 9	0 - 9 - 8		0 - 0 - 17	0 - 10 - 7	8 - 0 - 9
PGO-SOO	0 - 15 - 2	0 - 8 - 9	0 - 10 - 7	0 - 10 - 7	0 - 0 - 17		0 - 10 - 7	8 - 0 - 9
DIRECT	0 - 3 - 14	5 - 0 - 12	3 - 0 - 14	1 - 0 - 16	10 - 0 - 7	10 - 0 - 7		16 - 0 - 1
Random	0 - 17 - 0	0 - 14 - 3	0 - 17 - 0	0 - 17 - 0	0 - 8 - 9	0 - 8 - 9	0 - 16 - 1	

Table 2.6: Pairwise comparison of each algorithm across all objective functions with dimension $D > 2$ at $t = 500$ samples.

results that suggests that LOGO is more effective than BaMSOO and significantly more so than SOO.

Taken as a whole, these results seem to suggest that LOGO and IMGPO are the dominant optimization algorithms among those considered, both handily beating the current state-of-the-art ABO methods. Our results contradict this implied ranking. Most notably, we found that ABO-EI was a dominant algorithm and LOGO rarely performs significantly better than SOO.

Of course, we need to be cautious about reasoning about prior results in a transitive fashion. Each evaluation was performed on a different set of black-box functions, sometimes according to different metrics, and likely using different implementations of each algorithm. Still, it's surprising that there is such a discrepancy between our observed relatively performances and those derived from previous work.

For LOGO, the code used to generate prior results was not available due to contractual issues and we have not been able to replicate those results. However, we have validated our implementation with the authors of LOGO. One potential source for the discrepancy is that our evaluation employs randomization of the objective function, which we found was important to avoid observing performance differences due to lucky initializations or grid alignment.

ABO algorithms are necessarily 'tuned' by the authors of papers that use them in comparisons—however, the parameters selected are rarely reported since they're not considered relevant to the paper itself. Although EI is parameter-free, the Gaussian processes used in ABO methods have many parameters that can significantly effect ABO's performance. This may explain the unusually poor performance by ABO-EI in previous work. Without any motivation to maximize the ABO methods' performance in the comparison, it's unclear whether or not the parameters selected for the evaluation

result in representative behavior from the algorithm.

2.7 Summary

We presented experimental results comparing PGO, PBO, and ABO methods within a common open-source evaluation framework. The results demonstrate that acquisition-based optimization approaches, specifically ABO-EI, outperform partitioning-based optimization methods when evaluated by the average regret achieved after a given number of function observations in a strict black-box setting. Even when partitioning methods are given significantly more samples of the objective function, they are frequently unable to match the results that ABO-EI can achieve with much fewer samples.

The utility of an extremely computationally cheap black box optimization algorithm is already questionable since the limiting factor in problems that apply these algorithms is usually evaluating the black box function itself rather than the optimization algorithm's runtime. We demonstrate that fast partitioning-based methods tend to 'flat-line' on difficult problems even when given significantly more observations, or at least equal runtime, than competing expensive algorithms. This suggests that there are fewer situations in which PGO optimization should be chosen over BO-enabled methods than one might otherwise assume.

This apparent weakness is described in our results, but not well explained. Follow-up investigations into why or how the partitioning methods fail to improve as consistently as ABO methods can after a large number of samples are warranted. If the algorithm's shortcomings are in fact due to the refinement issue we discuss above, it's possible that modifying the depth limiting function or making other small changes to the algorithm could significantly improve its long-term performance compared to otherwise slower algorithms. Still, the partitioning methods' relative speed and efficiency make them a promising target for future work.

Although UCB is commonly shown to perform well when evaluated on synthetic objective functions, we found that without manually tuning its β parameter to maximize its performance it regularly failed to outperform both PBO and PGO methods.

Acknowledgements

This work was supported by NSF grant IIS-1619433 and DARPA contract N66001-19-2-4035.

Stake-Free Evaluation of Graph Networks for Spatio-Temporal Processes

Erich Merrill, Alan Fern

Submission to Transactions on Machine Learning Research

Chapter 3: Stake-Free Evaluation of Graph Networks for Spatio-Temporal Processes

3.1 Abstract

Spatio-temporal processes are a class of prediction problem that are poorly served by traditional deep learning architectures owing to the problems' frequently irregular structures. We propose a method for encoding such prediction problems as a graph which describes both the relationship between input samples from the process and the desired prediction targets requested of the model. However, it is difficult to determine which graph neural network (GNN) architecture is appropriate to apply to these problem domains, as works proposing novel GNNs generally only evaluate them in settings where they are expected to perform optimally rather than investigating their behavior and properties in general. We aim to address this gap in knowledge by performing a fair, stake-free evaluation of three different GNN architectures on three distinct spatio-temporal problems. The goal of this stake-free evaluation is to examine the behavior of each model on each problem type rather than demonstrate that one dominates the other. We find that GNNs which cannot exploit edge features perform poorly in this setting, and that GNNs which learn explicit interpretable weight functions are slightly outperformed by their counterparts that employ black-box function for the same purpose. Finally, we release the software platform used to perform this evaluation, which is designed to enable practitioners to easily reproduce or extend our experiments.

3.2 Introduction

Many interesting real-world problems can be described as spatio-temporal processes. A spatio-temporal process problem consists of a set of data samples for which each sample has an associated timestamp and associated position in some shared space. Frequently, these samples represent discrete observations of some underlying continuous process of interest. Examples include a citizen science setting, in which a collection of data is gath-

ered from a set of sensors distributed throughout some area that record environmental conditions over time; and a military engagement, in which a changing set of allied units periodically report their status and location, as well as any information available about the enemy's status and location. In both settings, the information of interest changes continuously over time, but generally its associated data would only be recorded at a set of instantaneous time points. In the environmental citizen science setting, the environmental conditions of interest likely change continuously throughout space as well, but we are only able to observe its state at the provided sensors' locations which we may not be able to control. Making useful inferences about the state of such processes requires jointly reasoning about both the spatial and temporal relationships between its samples.

There are well-known deep learning network architectures designed to process spatial and time series data: convolutional neural networks (CNNs) and recurrent neural networks (RNNs) respectively. These building blocks have previously been combined in various ways to construct models which are capable of reasoning about spatio-temporal data, such as by Shi et al. [2015] or Li et al. [2017] However, these architectures are challenging to effectively apply to real-world spatio-temporal processes in general. These architectures require that their input consists of a regular, discrete structure such as a grid of pixels for CNNs, and a sequence of predictably-spaced observations for RNNs. However, real-world spatio-temporal process data are unlikely to conform to this 'grid world' paradigm. For example, in the citizen science setting the sensors' positions cannot be controlled and generally cannot be approximated with a regular grid structure, making it difficult to effectively apply CNNs to reason about the data's spatial relationships. Additionally these architectures require their input structure to be densely populated with data, meaning they are unable to natively handle sparse structures with 'missing' data entries. For example, in the military setting information about enemy activity may be extremely sparsely and irregularly distributed throughout the duration of the encounter, making it difficult to apply RNNs to infer the enemy units' complete state at any given point in time.

One solution to these issues is to re-sample the data such that it is forced to conform to the required grid structure. For example, one could re-sample a set of points in 2D space by imposing a grid structure over the same space, and assigning each cell in the grid a value based on the samples contained within its bounds. However, this approach has significant drawbacks. Performing this resampling by discretizing continuously-valued

locations as described requires the grid resolution to be specifically tuned for each problem. If the grid resolution is too small data samples with close neighbors may be 'lost' since each grid cell only represents one sample, likely making it impossible for the model to have an accurate understanding of the state of the process. If the grid resolution is too large the majority of the cells will not represent any data sample in the problem, making it extremely difficult for the network to effectively reason about the relationships between the relevant data samples.

Another possible solution for some domains is to use an imputation model to fill in any data that is 'missing' from the required regular structure. However, in complex processes where the data samples are very sparsely distributed, the imputed samples far from any actual observations are unlikely to be informative. As before, the model will struggle to make useful inferences about the process if the vast majority of the samples in the sequence are 'fake' imputed samples. Additionally, generating and learning over these imputed samples may itself be significantly expensive. In these complex processes, computing a large number of imputed samples may impose an unacceptable performance overhead just to determine their values. On top of that, the model itself may require significantly more memory or other resources to process such large input data. Since the only purpose of the imputation is to allow the data to conform to the model's prescribed regular structure, a model that does not require such a regular structure would be more efficient since it is not required to generate this extra data, and presumably more performant since it can exclusively reason about the most informative samples rather than mostly working with the output of the imputation model.

These potential issues suggest that it would be preferable to employ a deep learning architecture that can directly consume irregularly-distributed, continuously-valued data such as that found in spatio-temporal processes. Graph neural networks (GNNs) are one such suitable architecture. Rather than operating on input of a fixed, static structure, graph networks directly exploit the relationships explicitly described by an arbitrary graph provided as input. Their use of parameter-sharing schemes and set functions allows them to consume graphs with arbitrary numbers of nodes, edges, and those with nodes of differing numbers of attached edges. This is much more flexible than the traditional CNN and RNN architectures, which are beholden to their discrete, fixed underlying structure. As a result, representing spatio-temporal problem instances with a graph structure and then processing them with a GNN model allows us to avoid the pitfalls describe above

associated with applying CNN- and RNN-style models to real-world spatio-temporal processes.

Given the potential for graphs to represent sparse spatio-temporal data, there is a question of how effectively different GNN architectures will perform on graph representations of such problems. As graphs can represent extremely diverse types of data, different GNN architectures employ drastically different approaches to processing their input, with drastically different corresponding inductive biases meant to exploit different features of the provided graph structure. Each type of graph network is generally inspired by certain types of problems and often evaluations are limited to just those problem types. While such evaluations highlight strengths each model, they fail to provide insight into performance on other problem types. This makes it difficult for practitioners to infer which GNN architecture may be most appropriate for a problem at hand. This, for example, is the case for sparsely sampled spatio-temporal process. This lack of a useful, neutral examination of GNN models' capabilities suggests the need for a 'stake-free' evaluation of these models, in which the goal is not to prove dominance of one model over the others. Rather, a 'stake-free' evaluation aims to examine the differing behavior and capabilities of each type of model on different problems.

In this work, we aim to address this gap in knowledge by fairly evaluating three meaningfully distinct GNN architectures on three different types of spatio-temporal problems. Each architecture is evaluated by defining three instantiations of each network type of varying sizes (i.e. Small, Medium, Large) such that every model of a given size has similar parameter count. To ensure that all models are trained and evaluated fairly we also propose a procedure for determining appropriate training hyperparameters for any given model instantiation, modified from the learning rate test proposed by Smith [2017].

We find that the edge-feature-aware graph networks significantly outperform the graph network architecture which exclusively relies on graph adjacency information to reason about the relationship between nodes. Of the edge-aware networks, some architecture learn explicit weight functions to filter the values of neighboring nodes. These weight functions can be visualized to demonstrate the model's understanding of the problem domain. However, the architectures that instead reason about each node's neighbors using a black-box function seem to generally perform better and exhibit better generalization performance despite their lack of explicit interpretability. Finally, we publicly release the software platform developed to perform all the experiments in this evaluation

to allow others to reproduce or extend our work.

3.3 Spatio-Temporal Processes

Each instance of a spatio-temporal problem consists of a set of observations (or samples) from distinct entities that change over time while occupying some shared space, and a set of domain-specific queries which describe the desired predictions within that same space. It is important to note that there are no constraints on how an observation's associated position and time are described. Whereas most deep learning architectures require spatial and temporal data to conform to some regular structure (such as a 2D grid representing an image, or an ordered sequence representing regularly-spaced observations throughout time), in our setting all observations' spatial and temporal locations are represented by unconstrained continuous values. Consequently, the number of input observations and target queries in a problem instance is not fixed. Any problem instance may include any number of entities, each of which may have any number of representative samples included in the set of input observations.

Just as instances of such spatio-temporal problems consist of a dynamic number of samples in the input set, the number of queries associated with each problem instance is also not fixed. For example, the problem's goal may be to predict the state of each entity at some future time, or it may be to predict the state of some unobserved location in space and time from the provided input samples. In either case, the number of desired predictions and their relationship to the input samples is not fixed. This is in contrast to most deep models, which generally produce a constant-size output or produce outputs of varying length but over a fixed structure, such as a sequence.

We observe a problem instance via a set of input samples X which describes the state of the process' entities at certain locations and time points. Specifically, each sample $x \in X$ describes an entity uniquely identified by $Ent(x)$ located at position $Pos(x)$ at time $Time(x)$. The domain-specific information for each sample needed for inference is represented by the sample's feature vector $Feat(x)$. To represent the semantic meaning of the desired predictions, we also define a set of 'query targets' Q. Each domain-specific query $q \in Q$ describes the relationship between the desired prediction target and the samples represented in X. For example, in a domain in which the goal is to predict the future state of an entity in the process, each query q would specify the entity in question,

Expression	Meaning
X	Set of all input samples that make up a problem instance
x	An individual sample within the process
Q	Set of all query targets that describe the problem instance's requested predictions
Q^*	Ground truth for queries Q
$Pos(x)$	The spatial location of the sample
$Time(x)$	The temporal location of the sample
$Feat(x)$	The sample's feature values, excluding positional information
$Ent(x)$	The sample's associated entity ID
$Dist_P(x, x')$	The difference vector between two samples' spatial locations
$Dist_T(x, x')$	The difference between samples' locations in time
$GNN(G, X)$	Calling a GNN to calculate latent encodings for the set of nodes X in graph G

Table 3.1: Spatio-Temporal Process Notation

$Ent(q)$, and the desired time point of the prediction, $Time(q)$. In a domain in which the goal is to predict the state of the process at some location rather than the prediction being associated with a specific entity, such as predicting the current weather conditions at an unobserved location, q would instead be defined by $Pos(q)$ and $Time(q)$.

An individual spatio-temporal problem instance is then defined by the tuple (X, Q) describing the input samples and desired queries. Models are trained on a labeled dataset $\mathcal{D} = \{\langle (X_i, Q_i), Q_i^* \rangle | i \in \{1, \dots, N\}\}$, where each Q_i^* is the ground truths for the set of queries Q_i on the observed input data X_i. The models are evaluated on their ability to accurately predict the correct Q^* when provided with a problem instance (X, Q). This problem structure allows us to train models on spatio-temporal problems consisting of a dynamic number of input samples and a dynamic number of domain-defined prediction targets. Such models are expected to reason about the spatial and temporal relationships within the input samples X, as well as the domain-specific relationships between those encoded input samples and the desired query targets Q.

3.4 Benchmark Domains

We consider three spatio-temporal problem domains to support our evaluation: Starcraft II battle unit state prediction, weather nowcasting, and traffic forecasting. These problem domains demonstrate the models' ability to understand qualitatively different

types of processes and queries. For example, observations from the Starcraft II domain describe the state of each individual unit present in the scene, whereas observations from the weather nowcasting and traffic forecasting domains represent point samples of the underlying process in question from fixed observation stations (that is, the atmospheric conditions recorded by weather stations and the traffic information recorded by road sensors). Queries in the Starcraft II and traffic forecasting domains task the model with predicting the future state of a specific entity described in the input observations, whereas the goal of the weather nowcasting problem is to use recent samples of nearby weather conditions to predict the current weather conditions at some unobserved location. Despite the significant differences in the underlying semantics of the input data and the structure of the queries, all problem instances are described as a spatio-temporal process as defined above. Specific information about each problem domain and how they are translated into a spatio-temporal problem instance is described below.

3.4.1 Starcraft II

The video game Starcraft II is a popular, challenging domain for evaluating machine learning models, most notably used by [Vinyals et al., 2019] as a challenge problem for reinforcement learning. It is interesting for our purposes primarily due to the dynamic nature of the military encounters represented by the game. These encounters may consist of just a couple of opposing units fighting one-on-one, or an entire battle with dozens of units on each side interacting with the others, making it a good platform to demonstrate how models can handle problems consisting of differing numbers of entities. Additionally, since the game engine reports the units' state at a high frequency, we can sample a subset of the recorded timesteps as input to the models to examine their ability to handle samples which are irregularly spaced in time.

The dataset is generated from a custom Starcraft II scenario in which two opposing armies fight each other on a featureless play field. Each scenario begins with a random number of three distinct unit types placed in random locations on the play field for both teams. The game then runs without any input, so the units perform their 'default' behavior of attacking any enemy units until they die or there are no enemies nearby. The scenario ends once all the units for one team have been defeated. Figure 3.1 shows an example of what a small encounter in this scenario looks like in-game.

Figure 3.1: Example SC2 scene

We use the PySC2 API interface provided by DeepMind [Vinyals et al., 2017] to record the state of each unit at each timestep across 1000 runs of the scenario. See Table 3.2 for a complete description of the feature vector representation for each unit. Individual problem instances are derived from this dataset by first selecting one individual timestep from one of the scenarios. We then choose a subset of the recorded game states before that timestep to use as 'input frames', and choose a subset of the timesteps after the selected timestep to use as 'query times'. The models are then evaluated based on their ability to use the unit information provided from the input frames to correctly predict the state of each unit at each query time.

Specifically, when evaluating models on this domain we select the 'input frames' by randomly selecting up to five timestep frames within the previous ten before the timestep in question. All entities in each selected frame are included in the set of input samples X. We fixed the set of target 'query times' when training and evaluating models on this domain unless otherwise specified. Specifically, for problem instance at time t we task the model with predicting each unit's state at all of the timesteps $t+1$, $t+2$, $t+4$, and $t+7$ that exist. Each timestep is approximately half a second of in-game time, so the models are asked to predict the future state of each unit in the scene at 0.5, 1, 2, and

Name	Type	Size	Target?	Description
Owner	Onehot	1		Binary representation of the team the unit belongs to
Type	Onehot	3		Onehot encoding of the unit type (marine, zergling, zealot)
Heath	Real	1	✓	Current health value of the unit
Shields	Real	1	✓	Current shield value of the unit
Orientation	Onehot	7	✓	Direction the unit is facing
Position	Real	2	✓	Cartesian position of the unit on the game field

Table 3.2: Description of feature vector representation of each unit in the Starcraft domain. The variable is used as a prediction query target if the Target? column is checked.

3.5 seconds into the future.

3.4.2 Weather Nowcasting

Predicting the current atmospheric conditions at a target location given a set of current or prior weather observations from nearby areas ('nowcasting') requires jointly reasoning about the spatial and temporal relationships between the target location of interest and the provided historic weather observations. We use a dataset of weather conditions recorded by a group of weather stations distributed throughout Oklahoma to derive such a weather nowcasting problem to evaluate the models.

The dataset consists of the atmospheric conditions and associated quality metrics recorded by each weather station in five-minute intervals throughout the entirety of 2008, as well as metadata describing each station's static properties (e.g. location, elevation, soil type). See Table 3.3 for a complete description of the feature vector representation for each station reading. Any samples with quality metrics that report their readings may not be accurate are removed from the dataset entirely. No imputation is performed to 'fill in' these missing data points, since our models are expected to be able to process such sparse data on their own. 90% of the stations are selected to be used as training data, while the remaining 10% are used for testing.

To derive a training problem instance from this dataset, we select a timestep t from the dataset and a subset of the training stations to use as 'target' stations. The set of input samples X consists of all weather station observations from the hour prior to the selected timestep t which do not come from a selected 'target' station. The set of

Name	Metadata?	Target?	Description
RELH		✓	Relative Humidity
TAIR		✓	Air Temperature
WSPD		✓	Average Wind Speed
WVEC			Vector Average Wind Speed
WDIR			Wind Direction (heading)
WDSD			Standard Dev. of Wind Direction
WSSD			Standard Dev. of Wind Speed
WMAX			Maximum Wind Speed
RAIN			Liquid precipitation since 00 UTC
PRES		✓	Station Pressure
SRAD			Solar Radiation
TA9M			Air Temperature at 9m
WS2M			Wind Speed at 2m
TS10			Sod Soil Temp at 10cm
TB10			Bare Soil Temp at 10cm
TS05			Sod Soil Temp at 5cm
TB05			Bare Soil Temp at 5cm
ELEV	✓		Elevation
LAT	✓		Latitude of station
LON	✓		Longitude of station
Soil Info	✓		A vector of length 18 describing soil properties

Table 3.3: Description of feature vector representation of each station in the weather domain. The variable is used as a prediction query target if the Target? column is checked. Metadata? is checked if the value is static and associated with the weather station in question.

queries Q describes the locations of each desired nowcasting prediction, which are the locations of all of the selected 'target' stations. The ground truth Q^* is then set to be the recorded observation by each target station at time t. Test problems are generated similarly, except that all the training stations' samples are included in the input to X, and the model is tasked with predicting the state of the held out test stations at the selected timestep t.

3.4.3 Traffic Prediction

Predicting traffic flow is a common problem to demonstrate the performance of spatio-temporal GNN models (such as by Yu et al. [2017] and Zhang et al. [2020]), as the signals of interest are highly periodic in time and exhibit significant spatial locality throughout the road network. We use the publicly available METR-LA dataset [Li et al., 2017], which consists of 206 sensors distributed through the LA highway system. The sensors report the average speed of the highway traffic at their location every five minutes. The dataset consists of all readings between between March 2012 and June 2012.

An individual problem instance is derived from this dataset by first selecting a timestep t in the dataset. The traffic network's sensor readings from the previous hour (that is, timesteps $t - 11$ through t) are collected to use as the set of input samples X. The set of queries Q is defined to request the state of each sensor in the traffic network one hour in the future (that is, timestep $t + 12$), and Q^* is set to be the observed traffic conditions at each sensor at that time. The models are evaluated on their ability to use the provided recent traffic data to predict the speed of traffic flow at each sensor's position in the road network at the selected target timestep. Note that the are no held out 'target sensors' that are hidden from the input data, unlike in the weather nowcasting domain. The goal of each problem instance is to use the historic data from all sensors to predict the future state of all sensors. Instead, we select the data from every tenth week from the dataset to use as test data, while the remaining 90% of the data is used for training.

Note that the input and query structures for this domain are fixed, unlike in the other domains. For Starcraft, the relative position of the units is always changing, and units may disappear at any time. For weather nowcasting, some stations' observations may not be present in each input timestep and the requested queries may change with

each problem instance. In comparison, the input to each traffic prediction problem is a fixed size, as each traffic sensor is present at every timestep. Additionally, the queries always represent the same relationship– requesting the state of each sensor one hour in the future. This enables this specific problem formulation to be fully compatible with classic temporal models, such as RNNs. However, the spatial component of the problem is still incompatible with classic spatial models, as spatial gridding would still be necessary and CNNs would be unable to exploit the explicitly-defined graph structure of the road network's spatial layout.

3.5 Models

Graph networks are a class of artificial neural networks which operate on graph-structured data. These graphs consist of a set of nodes with associated feature vectors, and at least one set of edges describing the connectivity of the nodes. In our settings, edges also have associated feature vectors describing the relative information between the two nodes they connect.

Message-passing GNNs are a class of GNN that generally operate by using their input graph's adjacency information described by its edges and the attached nodes' feature values to calculate a latent 'neighborhood representation' for some (usually all) nodes in the graph. Note that the structure of the input graph in not constrained, so a node may have any number of neighbors and associated edges which contribute to its latent neighborhood representation. The resulting fixed-sized neighborhood representation is then combined with the node's feature vector to determine a new latent representation for each node which is meant to describe the 'context' from its immediate neighbors as it relates to the node in question itself. This process can be performed for any subset of nodes in the graph to calculate latent feature vector for those specific nodes. When performed on all nodes X present in the graph, the output of this encoding process is a set of latent feature vectors which correspond to all of the provided input nodes. By setting the graph's node features to be the corresponding latent feature vectors calculated by this application of the GNN, we get an updated latent graph with identical structure as the input but with latent feature vectors which hopefully represent useful information about each node's neighbors. This encoding process can then be repeated with any number of layers, similar to other DNN architectures, which effectively increases the 'range' at

Expression	Meaning
X	Set of nodes
E	Set of edges
$G(X,E)$	Graph consisting of nodes X and edges E
e_{xy}	Edge connecting nodes x and y
$EdgeFeat(x,y)$	Feature value of e_{xy}

Table 3.4: Graph Encoding Notation

which information can propagate throughout the graph. We represent this process of using a GNN to calculate feature representations for a subset of nodes X in the graph G with $GNN(G,X)$.

In contrast, calculating the latent feature vectors for the query nodes Q is done with $GNN(G,Q)$. Note that the output of this GNN does not directly correspond with the nodes in the input graph G, since Q is a strict subset of the input nodes X. As a result, this process is not repeatable or layerable as the input encoding process is. Since the output of $GNN(G,Q)$ correspond to the queries Q, we finally pass each latent encoding of a query node through an MLP to calculate the final prediction for each query based on the input graph.

3.5.1 Graph Encoding

Since the models we examine operate exclusively on graphs, the process of deriving a graph representation of each problem instance is crucially important in enabling the models to effectively reason about the problem. We propose a simple technique for converting such spatio-temporal prediction problems into a multi-relational graph which captures the spatial and temporal relationships between relevant samples in the problem instance. See Table 3.4 for a description of the notation used in describing this process.

The core idea is to define three different sets of edges between the samples X described in the input spatio-temporal process instance:

- E_s to represent the spatial relationships between samples on each individual timestep, such as the distance vector between two interacting units in a Starcraft battle; and
- E_t to represent the temporal relationships between samples of each individual entity

across all timesteps in which it is present, usually the difference in timestamps between ; and

- E_q to represent the domain-appropriate relationship between the problem's input samples and the specified query targets. For example, in the Starcraft domain in which the goal is to predict the future state of a specific unit, E_q is defined identically to E_t. In the weather domain, where our goal is to predict the weather conditions at a unobserved location at an observed timestep, E_q is defined identically to E_s.

Specifically, the set of temporal edges E_t and their values is defined as follows:

$$EdgeFeat_t(x, y) = Time(y) - Time(x) \tag{3.1}$$

$$E_t = \{\forall_{x \in X} \forall_{y \in X, x \neq y}\ e_{xy} \text{ if } Ent(x) = Ent(y)\} \tag{3.2}$$

The spatial edges are defined similarly. Due to the potentially large number of entities within a given timestep, spatial edges are only created between samples within some specified maximum interaction distance. Specifically, the set of spatial edges E_s and their values is defined as follows:

$$EdgeFeat_s(x, y) = Pos(y) - Pos(x) \tag{3.3}$$

$$E_s = \{\forall_{x \in X} \forall_{y \in X, x \neq y}\ e_{xy} \text{ if } |EdgeFeat_s(x, y)| < \Delta_{\max}\} \tag{3.4}$$

With these two disjoint sets of edges, we can realize two different graphs that share the same set of nodes but use the two sets of edges to represent both types of relationships: the spatial relationship graph $G_s = G(X, E_s)$, and the temporal relationship graph $G_t = G(X, E_t)$.

Encoding an input graph G by calculating a latent feature vector for each node X in the graph requires employing the spatial and temporal GNN on their respective graph representations, as well as MLP applied in parallel to each latent node representation to combine the output from each GNN into a latent representation for each node that combines information from its spatial and temporal neighborhoods.

Algorithm 7

function SpatioTemporalEncoder(X, E_s, E_t, GNN_s, GNN_t, $NodeMLP$)
  $G_s \leftarrow G(X, E_s)$
  $Y \leftarrow GNN_s(G_s, X)$
  $G_t \leftarrow G(X, E_t)$
  $Z \leftarrow GNN_t(G_t, X)$
  $X' \leftarrow NodeMLP(X||Y||Z)$
  return X'
end function

Algorithm 8

function SpatioTemporalQuery(X, E_q, GNN_q, $QueryMLP$)
  $G_s \leftarrow G(X, E_q)$
  $Y \leftarrow GNN_q(G_s, X)$
  $P \leftarrow QueryMLP(Y)$
  return P
end function

The resultant latent encoding of the samples X' can then be fed into the next layer of spatial/temporal GNN pairs. See Algorithm **??** for a description of how one such layer is executed. Note that E_s and E_t are constant across all layers, and therefore only need to be determined once per problem.

Finally, we must use the resulting latent encoding of each sample to derive a latent encoding of each desired query target $q \in Q$. As before, we define a set of edges E_q that describes the relationship between the encoded samples X and the desired queries Q. The specific procedure for defining E_q is domain specific, as it depends on the type of relationship between the samples and the queries. Note that E_q must be a bijection between X and Q, representing the fact that the result of each query q is exclusively dependent on the latent encoding of the input samples, and not dependent on the other contents of Q itself. See Algorithm **??** for a description of how the query predictions P are derived from the graph from the encoding layers.

Expression	Meaning
X^ℓ	Input node features at layer ℓ
A	Adjacency matrix
D	Degree matrix
W^ℓ	Per-layer trainable parameter matrix for layer ℓ

Table 3.5: GraphConv notation

3.5.2 GraphConv

GraphConv [Kipf and Welling, 2016] is a simple message-passing GNN model designed to approximate a full spectral convolution of the input graph. This approach exclusively uses adjacency information from the graph's set of edges to determine how nearby nodes' features should be combined.

The GraphConv update rule is defined as follows:

$$X^{\ell+1} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}X^\ell W^\ell\right) \tag{3.5}$$

Where X is a $n \times f_1$ matrix of the nodes' feature values. $\tilde{A}$ is the $n \times n$ adjacency matrix A with added self-connections, that is $\tilde{A} = A + I_N$. $\tilde{D}$ is the $n \times n$ degree matrix, defined as $D_{ii} = \sum_j A_{ij}$. W^ℓ is the $n \times f_2$ weight matrix for layer ℓ.

Notably this approach completely ignores any features associated with the graph's edges. In our setting, these edges are assigned feature values that explicitly describe the spatial or temporal relationship between the two samples. Therefore, GraphConv cannot take advantage of this information directly. It must be inferred via some implicit pairwise comparison between a node and its neighbors' features. However, since the neighbors' features are collected purely via the operation multiplying the normalized adjacency matrix $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ by the nodes' feature values, there is no mechanism by which individual neighbors' features can be processed differently based on their relation to the 'central' node. Any such logic must therefore be expressed by the effects of the weight matrices W on the weighted combination of neighborhood and node features, which is unlikely to be an efficient or effective way to express those functions.

Expression	Meaning
X^ℓ	Node features at layer ℓ
a	Aggregation function – combines effects with their associated objects
ϕ_O	Object model – determines the future state of each object and its associated interactions
ϕ_R	Relational model – determines the effect of each interaction
m	Neighborhood/marshalling function – determines interactions and relative distances

Table 3.6: Interaction network notation

3.5.3 Interaction Networks

Interaction networks [Battaglia et al., 2016] are a general class of GNN designed to operate on problems involving many individual interacting entities. A single Interaction Network layer is described as follows (see Table 3.6 for notation):

$$X^{\ell+1} = \phi_O\left(a\left(G^\ell, \phi_R\left(m\left(G^\ell\right)\right)\right)\right) \tag{3.6}$$

In our setting, each function is defined as follows:

- m, the marshalling/neighborhood function, which operates on the provided graph $G(X, E)$. E is one of E_s, E_t, or E_q depending on the context in which the layer is being applied. The output of m represents each edge (or relation) in the graph with a tuple consisting of the source node's feature vector, the destination node's feature vector, and the edge feature itself describing the relative information between those two nodes. Specifically, $m(G(X, E)) = \{[x, y, EdgeFeat(x, y)] \mid \forall_{x \in X} \forall_{y \in X, y \neq x}, s.t. e_{xy} \in E\}$.

- ϕ_R, the relational model, which transforms each relation tuple from m into a fixed-length latent representation of that relation's 'effects'. Specifically, ϕ_R is implemented with a MLP that is applied in parallel to each relation tuple.

- $a(G(X, E), R)$, the aggregation function, performs two tasks: combining the variable number of latent relational effect vectors associated with each node into a single fixed-size 'neighborhood effect' vector, and combining this 'neighborhood

effect' vector with the node's own latent feature representation. Specifically, for each node $x \in X$, $N_x = \sum_{r \in R_x} r$, where R_x is the set of relations in R which are associated with edges in E directed towards node x. The output of the aggregation function is then $a(G(X,E),R) = \{[x, N_x] \mid \forall_{x \in X}\}$, a set of pairs consisting of each node's feature representation and its aggregated relation latent vectors describing all its neighbors.

- ϕ_O, the object function, which transforms the pairs of node features and associated aggregated neighborhood features into a final latent representation for that node to be provided to the next layer. Specifically, ϕ_O is implemented with an MLP that is applied in parallel to each object-neighborhood pair.

At a high level, applying one layer of an interaction network performs the following steps:

- use the edge information provided by the input graph to determine all the neighboring nodes for each node in the graph;
- transform each neighbor relation into a latent feature vector based on the main node's value, the neighboring node's value, and the connecting edge's value;
- sum together all latent neighbor relations associated with each node into a single fixed-size neighborhood representation;
- finally, combine each node's latent feature representation with its neighborhood representation to produce an encoded latent feature representation for each node.

3.5.4 PointConv

PointConv [Wu et al., 2019] is a convolutional neural network designed to operate on point clouds rather than the image or voxel data that would be consumed by a traditional CNN. Whereas CNNs learn a set of discrete filters to apply over regularly-structured hyper-rectangles (usually 2D/3D images), PointConv instead learns a filter function represented by an MLP which transforms a distance vector between two points into the filter value for that location. The authors prove that PointConv is equivalent to traditional pixel-based image convolution

At a high level, PointConv performs the following steps:

- determine the 'neighborhood' of each point in the cloud by collecting all other points within a certain distance;
- calculate the relative distance between each point and each point in its neighborhood;
- transform each relative distance vector into a filter vector using a MLP;
- perform a matrix multiplication between the filter vector and its associated neighbor's feature vector;
- sum all resulting filtered neighbor vectors into a single fixed-size neighborhood representation;
- finally, combine each point's latent feature representation with its neighborhood representation to produce an encoded latent feature representation of each point.

Although it isn't presented as such in the paper, PointConv can also be described in terms of being a graph network. The only significant difference between the PointConv and Interaction Network algorithms is how the neighborhood feature vector is calculated by the relational model ϕ_R. While the interaction network simply calculates each neighbor representation with a MLP such that $r = MLP(x||y||EdgeFeat(x, y))$, PointConv calculates each neighbor representation as $r = F(y^{\mathsf{T}}W(EdgeFeat(x, y)))$, where W is a MLP that transforms an edge feature vector into a fixed-size filter vector, and F is a 'flattening' function that reshapes an $n \times m$ matrix into a $1 \times (n \times m)$ vector.

Notably, this filtering-based approach removes the neighbor vector's dependency on the feature value of the point (or node) for which the neighborhood is being calculated. As the filter value is only dependent on the relative distance between two samples, any pair of samples with the same relative distance will also have the same filter value returned by W.

3.5.5 PointConv with Attention

As described above, the filters calculated by PointConv do not depend on the feature value of the node x or its neighbor x'. While this allows us to explicitly calculate filter

weights for each neighbor, it seems reasonable to expect that in some domains these neighbors should be filtered differently depending on both their feature descriptions.

We can extend PointConv to have such capabilities by applying a simple attention mechanism to the weight calculation, as described by Horn et al. [2019]. Instead of calculating the weights exclusively with $W(EdgeFeat(x, y))$, we first calculate an attention vector $a = softmax(A(Feat(x)||Feat(y)))$, which uses an MLP A to calculate a vector of a small number of normalized scalar weights depending on the comparison between the node x and its neighbor y. We then multiply the filters by each value in the attention weights as follows: $r = F(x'^{\intercal} F(a^{\intercal} W(EdgeFeat(x, y))))$.

Note that now our neighbor representation r for a given node x and neighbor y is now dependent on the values of x, y, and the relative distance between them, $EdgeFeat(x, y)$. However, instead of accomplishing this by simply concatenating these values together and applying an MLP as Interaction Networks do, we explicitly learn a set of spatial filters that depend exclusively on the relative distance between the samples, and a set of weights that depend exclusively on the two samples' feature values.

3.6 Hyperparameter Selection and Training

The performance of deep learning models, is highly dependent on the hyperparameter configuration used for training. In some cases, the effect of thorough hyperparameter optimization on a model's observed performance may dominate the performance impact of that model's structural or procedural differences from the other models it is being compared to. As a result, when comparing different models' performance it is necessary to ensure that each model receives similar amounts of hyperparameter tuning effort to ensure that the comparison's results represent the differences in the models' inherent properties rather than the differences in their high-level training procedures. This can be problematic since many works presenting novel models do not make an attempt to quantify how much time or effort was put into determining the hyperparameters for training the model being evaluated. Future works that then use such reported results as comparison points cannot guarantee that the models being evaluated in the comparison are 'evenly matched' in terms of hyperparameter optimization effort. The widespread assumption is then that any presented results are from a hyperparameter configuration which 'maximizes' the performance of the model in question. This assumption incen-

tivizes authors to commit significant time and resources to optimizing hyperparameters to improve their proposed model's performance without necessarily recording or describing that optimization process, since it is generally assumed that the comparison models (whose performance frequently must be matched for the work to appear relevant) also had significant time and resources committed to maximizing their performance.

We intentionally take a different approach for our 'stake-free' setting by defining a single hyperparameter selection procedure to be applied to each instantiation of each model in the evaluation. Explicitly making the hyperparameter selection process part of the evaluation procedure allows us to guarantee that each model received a fair amount of hyperparameter tuning 'effort' since we can demonstrate that the selection process and resources used were the same for all models. The simplest common approach for hyperparameter optimization is grid search or random search [Bergstra and Bengio, 2012], in which every hyperparameter is enumerated and regular or random samples drawn from the resulting hyperparameter configuration space are evaluated with an auxiliary training procedure. The hyperparameter configuration with the best empirical performance demonstrated by the auxiliary training procedure is then selected to be the representative hyperparameter setting for the model in question. This approach can be very expensive due to the massive number of training runs that must be executed to ensure good coverage of the hyperparameter configuration space. Since we need to perform the hyperparameter selection for each size, for each model, for each domain, performing a complete grid search is infeasible.

To determine a reasonable hyperparameter settings within a reasonable amount of time, we fix most training hyperparameter values to commonly-used defaults, and focus exclusively on determining an appropriate learning rate schedule for each model instantiation, as the learning rate has been shown to generally be the most impactful hyperparameter in determining model performance [Smith, 2017]. Specifically, for all experiments we use the ADAM optimizer [Kingma and Ba, 2014] as implemented by PyTorch [Paszke et al., 2019], using its default parameters of a weight decay of 0, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We use a cyclic cosine annealing learning rate schedule as described by Loshchilov and Hutter [2016], with T_{mult} set to 2 and T_0 set to $\frac{1}{7}$th of the total number of epochs to ensure that the schedule can complete three periods during each training run. The cosine annealing schedule's minimum and maximum learning rates are determined experimentally for each model instantiation using a modified learning rate test described

below.

3.6.1 Checkpoint Learning Rate Test

Smith [2017] describes a procedure for quickly determining a range of usable learning rates for training any given network. The key idea is to perform an auxiliary training run on a randomly initialized network using the intended optimizer, while exponentially interpolating the optimizer's learning rate from a low value known to have no meaningful impact on the network's performance to a high value known to cause the network to 'explode' and lose performance. This is in opposition to the usual approach for training a deep network, in which we would start with an appropriately large learning rate to allow the optimizer to find a promising parameter region, then gradually lower the learning rate to 'refine' its behavior with smaller parameter updates. Instead, the goal of a learning rate test is to quickly 'scan' through a wide range of plausible learning rates and examine the resulting effect on the performance behavior on the model being tested. The expectation is that there will be three regions which can be identified from these results:

- a 'too cold' region in which the learning rate is too low to impact the model's parameters, and the observed training loss 'flatlines';
- a 'just right' region in which the learning rate is effective and enables the model to improve, causing the observed training loss to fall;
- and a 'too hot' region in which the high learning rate causes the parameter updates to 'explode' the model, causing the observed training loss to quickly increase.

One can then assume that any learning rate from the 'just right' region is appropriate to use when training the final model.

This simple learning rate test procedure ignores some crucial facts about training deep networks. Specifically, the concept of a learning rate schedule is validated by the observed evidence that the most effective learning rate for an optimizer can change significantly over time as the model moves into effectively different regions of its parameter space. However, the learning rate test seems to make the opposite assumption– that the most effective learning rate does *not* change even as we update the model's parameters. By

constantly updating the model's parameters as the learning rate is varied throughout the test, the test is essentially modifying its underlying problem domain while simultaneously searching for a solution within it. This calls into question the general utility of the results of such a test.

Fortunately this shortcoming is easy to address. We propose a simple modification to the learning rate test, the 'checkpoint learning rate test', in which the network's parameters are reset to a fixed state after each reported training loss. This change significantly reduces the 'domain shift' effect imparted by constantly updating a single model throughout the test. Instead, by constantly resetting to a checkpoint the observed model's parameters are never able to get too 'far' from the checkpoint model's parameters, increasing the chance that the test's results represent the expected behavior of the checkpoint model. Algorithm 9 describes our proposed algorithm for performing the checkpoint learning rate test, while Algorithm 11 describes the procedure used to determine the appropriate minimum and maximum learning rate from the output of a learning rate test.

Adding a checkpoint addresses the learning rate test's 'domain shift' problem, but still does not change the fact that effective learning rates generally change over time as a deep model is trained. If the checkpointing learning test is run with a randomly initialized parameter configuration as the checkpoint, then its results will represent the training characteristics of a randomly initialized network, such as one at the very beginning of a training run. However, the learning rate test is intended to determine an appropriate learning rate range to use throughout the entire training procedure, not just the very beginning. In fact, since networks tend to become more sensitive to the learning rate being used as they are trained, exclusively using the network's random initial conditions to determine acceptable learning rates for the entire training procedure may be of limited value.

To ensure the results of the learning rate test are useful for training across more of the training phase, we first apply a common auxiliary training procedure to produce a 'poorly-trained' network. Using this 'poorly-trained' network as the checkpoint results in an estimated learning rate range which is much more appropriate for the entire training process, since its parameter configuration is much closer to the state the network will be in during training compared to the initial random state. We found that networks trained with the learning rate range derived from this 'poorly trained' network were much more

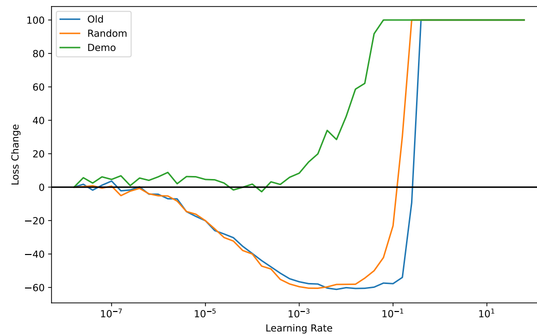


Figure 3.2: The results of three styles of learning rate test applied to the same network architecture. Old is the classic learning rate test, Random is the random checkpoint test, and Demo is the 'poorly trained' checkpoint test. Note the clear area of improvement for the Old and Random tests, while Demo has a much less clear area of improvement and 'explodes' at a much lower learning rate.

likely to converge in some domains than networks trained with the learning rate range derived from the random checkpoint or classic learning rate test. Figure 3.2 demonstrates this by showing the results of a learning rate test with three different approaches: the classic learning rate test, the test with random checkpointing, and the test with 'poorly trained' checkpointing.

This demonstrates that the effective learning rate range for a model changes as the model's parameters are updated during training. From this evidence, seems reasonable to then conclude that a single learning rate test is unlikely to be able to determine an appropriate learning rate for the entirety of the training procedure. However, since our modified learning rate test does not suffer from the same failures as the original learning rate test, we employ it to determine the learning rate ranges for every model instantiation across all domains. Further examining and developing this style of hyperparameter test is left to future works.

3.7 Results

In this section we present the results of our empirical investigation. We first, present an overall summary of the main observations. Next we provide additional details for each of the benchmark problems.

Algorithm 9 Checkpoint LR Test

```
function CheckpointLRTest(net, θ, dataset)
    SetNetworkParams(net , θ)
    batchCount ← 0
    losses ← empty list
    nextReport ← reportEvery
    for batch in dataset do
        batchCount ← batchCount + 1
        pred ← Predict(net, batch)
        loss ← CalculateLoss(pred, batch)
        lr ← LRSchedule(batchCount)
        UpdateParams(net, loss, lr)
        Append loss to losses
        if batchCount ≥ nextReport then
            nextReport ← nextReport + reportEvery
            SetNetworkParams(net, θ)
            RecordLoss(lr, mean(losses))
            losses ← empty list
        end if
    end for
end function
```

Algorithm 11 Learing Rate Calculation

function CalculateLRRange(lrs, losses, margin)
  baseline ← losses[0]
  highThresh ← baseline × margin
  highIdx ← min i such that losses[i] ¿ highThresh
  lrs ← lrs[0:highIdx]
  losses ← losses[0:highIdx]
  lowScores ← [Score(i, lrs, losses, baseline) for i in 0..length(losses)]
  lowIdx ← argmax_i lowScores[i]
  lowLR ← lrs[lowIdx]
  highLR ← lrs[highIdx]
  return (lowLR, highLR)
end function
function Score(idx, lrs, losses, threshold)
  target ← [loss ¡ threshold for loss in losses]
  pred ← [i ≥ idx for i in 0..length(losses)]
  score ← F1Score(pred, target)
  return score
end function

3.7.1 Overall Performance

Table 3.7 shows the loss statistics for all network architectures and sizes on all domains. In it, and all other figures in this paper, the PointConv-based network is abbreviated as 'PC', PointConv with attention is abbreviated as 'PCA', GraphConv is abbreviated as 'GC', and Interaction Network is abbreviated is 'Int'. Rather than report mean test loss, to avoid sensitivity to outlier predictions, we report the 25%, 50%, and 75% percentiles of loss values among all individual predictions each model made on each domain, in columns labeled P_{25}, P_{50}, and P_{75}. We then identify 'bad' predictions by identifying all predictions with a loss 100x higher than the 75% percentile of the loss values. These predictions are labeled as failures, and the column labeled %Fail shows the percentage of all predictions for that model that were identified as bad. Finally, we report the mean of all losses not identified as bad in the column labeled Mean.

GraphConv is clearly not competitive with the edge-aware GNN models overall. Its performance is especially poor on the Starcraft 2 domain, in which being able to differentiate near and far units is critical to predicting what action each unit will take. This

supports the idea that edge-aware GNN models should be applied to graph problems with highly dynamic structures, whereas GraphConv may be better suited for graphs which always have the same or similar structures.

Accordingly, Interaction Networks dominate the Starcraft 2 domain, suggesting that their black box MLP relational model is significantly more performant that the decomposed versions used by PointConv. The Interaction network based models seemed to perform best at their largest size, dominating both the Starcraft 2 and Weather Nowcasting domains among large models. This may suggest that the black box relational model approach is more efficient at higher parameter counts than the biased convolution-style approach employed by PointConv, but more research would be necessary to investigate this effect in detail.

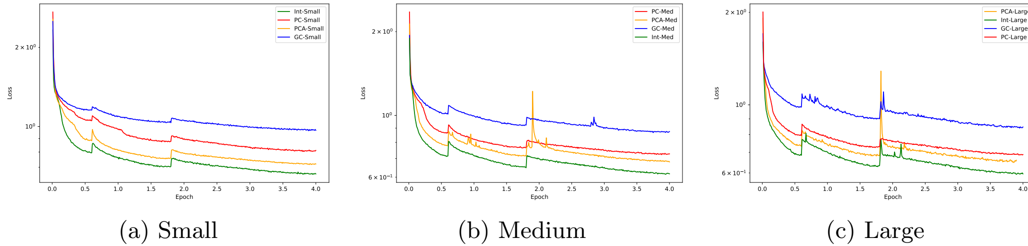
PointConv with Attention significantly outperforms PointConv in the Starcraft 2 domain, but appears to have completely failed on the weather problem. With a 25th percentile loss more than 10x higher than the other models for the small and medium sizes without any extreme prediction errors resulting in failures, it's clear that almost all of the PointConvAttention's predictions in this domain are useless. It's unclear why the addition of an attention mechanism may have caused this, considering that the extremely similar PointConv model performs well on the same domain.

The PointConv models seem to perform their best in the Traffic domain and the Weather domain (other than the large size). This may be related to the fact that these domains both have static graph structures, which may favor PointConv's approach of explicitly learning spatial filters, whereas in the Starcraft 2 domain which has extremely dynamic graph structures and entity interactions the Interaction Networks, with their explicit pairwise comparisons between samples, are much more performant.

Overall the results confirm our suspicion that when trained with equal amounts of hyperparameter tuning and training effort, no one GNN model dominates the others across all domains. Instead, the most important thing is to select the model with the inductive bias that best matches the properties of the problem instances that the model will be tasked with. The evaluation results suggest that PointConv-style, convolution-inspried architectures may be preferable when the graph structure is fixed, while the more entity-focused Interaction Networks may be a better fit for problems with highly dynamic graph structures.

Domain	SC2					Weather					Traffic				
Network	P_{25}	P_{50}	P_{75}	Mean	%Fail	P_{25}	P_{50}	P_{75}	Mean	%Fail	P_{25}	P_{50}	P_{75}	Mean	%Fail
GC-Small	1.37	5.85	14.76	9.48	0.0	**2.38**	5.85	13.05	10.07	0.2	0.77	3.07	14.06	45.92	**0.0**
Int-Small	**0.20**	**2.43**	**9.61**	**6.50**	0.0	4.89	9.64	18.31	14.09	0.4	0.70	2.73	11.03	43.95	¡0.1
PC-Small	0.63	4.18	12.15	7.97	0.0	2.77	**5.69**	**11.22**	**9.00**	**¡0.1**	0.77	2.81	11.24	45.17	¡0.1
PCA-Small	0.33	3.04	10.64	7.10	0.0	21.48	34.05	53.13	41.76	0.4	**0.51**	**2.17**	**10.69**	**42.26**	0.2
GC-Med	0.84	4.81	13.15	8.51	0.0	1.26	2.63	5.49	4.72	¡0.1	0.83	3.22	13.78	45.79	¡0.1
Int-Med	**0.14**	**2.01**	**8.79**	**6.06**	0.0	0.61	1.42	3.40	3.16	**0.0**	0.68	2.86	13.69	45.39	**0.0**
PC-Med	0.36	3.26	10.78	7.17	0.0	**0.43**	**0.89**	**1.98**	**1.94**	**0.0**	**0.40**	**1.88**	**9.68**	**36.44**	0.9
PCA-Med	0.25	2.68	10.02	6.73	0.0	20.67	34.40	53.49	41.74	0.3	0.41	1.96	9.89	37.57	0.8
GC-Large	0.74	4.52	12.71	8.26	¡0.1	5.43	9.11	17.06	20.98	8.7	**0.39**	2.08	13.55	46.75	**0.0**
Int-Large	**0.14**	**1.85**	**8.40**	**5.85**	**0.0**	**0.91**	**1.91**	**3.95**	**3.54**	**0.0**	1.76	5.63	19.08	47.16	**0.0**
PC-Large	0.27	2.81	10.10	6.79	**0.0**	1.32	2.79	5.85	4.84	0.8	0.42	**1.97**	**9.86**	**38.44**	0.8
PCA-Large	0.21	2.45	9.48	6.44	**0.0**	7.58	14.06	24.30	18.11	0.2	0.54	2.38	10.45	43.05	0.4

Table 3.7: Loss Table



(a) Small (b) Medium (c) Large

Figure 3.3: Average training loss plots for all instances of each model trained on the SC2 dataset.

3.7.2 Starcraft II

Training Curves. Figure 3.3 shows the average training loss across all training runs for each architecture instantiation on the Starcraft 2 dataset. The models' ordering is identical to that observed when evaluating them on the test data, suggesting no overfitting is occurring. PointConv with Attention is noticeably unstable during training, however. This may be due to the learning rate test failing for this architecture and calculating too high of an initial learning rate, since the loss spikes seem to occur shortly after the cyclic learning rate schedule returns to its highest learning rates.

3.7.2.1 Spatial Filter Visualization

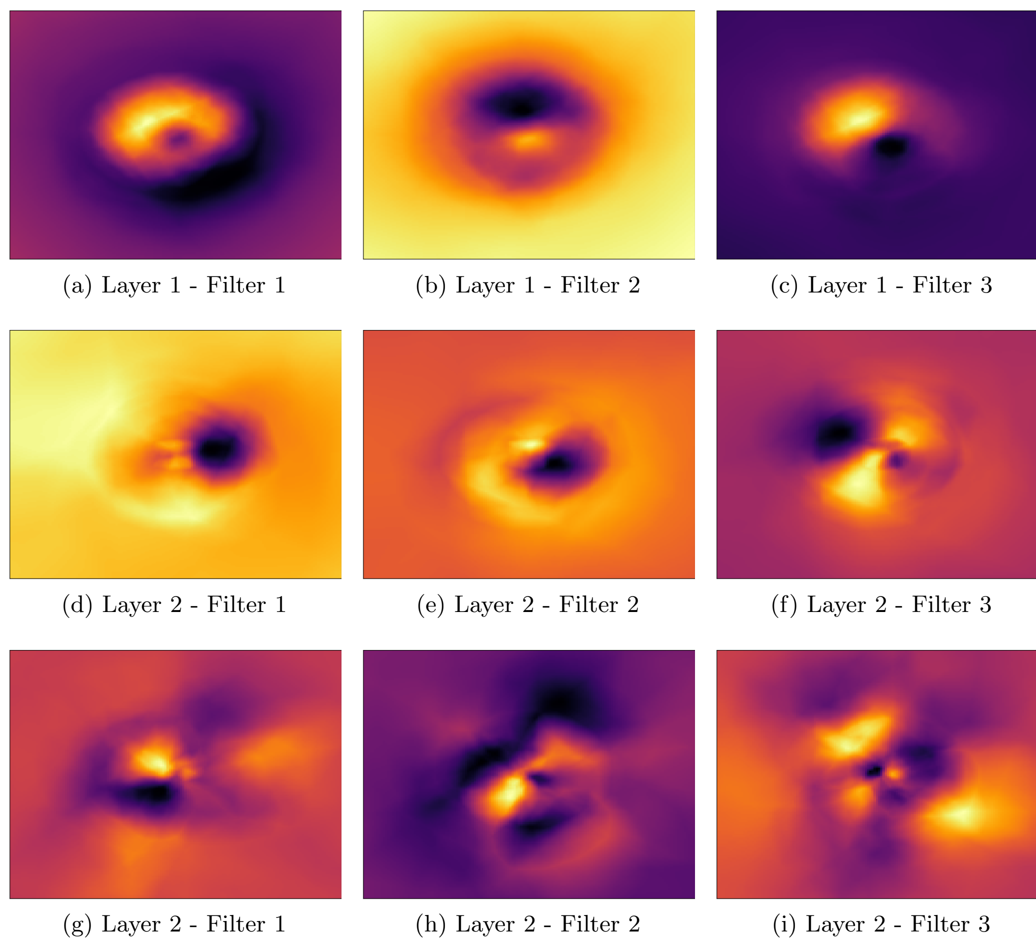
Figure 3.4 shows several of the continuous spatial filters learned by a PointConv model trained on the Starcraft 2 dataset. The filters seem to focus on the area in a tight radius around the unit in question with the further positions generally being constant, especially in the early layers where sufficiently far units will not have any impact on the unit in question's future state. This shows that the network is learning meaningful filters to gather information about the nearby units. If the learned filters were uninformative, the filters would appear to be a random projection– instead, we can clearly identify shapes that we know are directly relevant to the underlying spatio-temporal process' behavior. Note that the PointConv architectures are the only ones capable of producing such visualizations, since GraphConv and Interaction Networks do not learn a explicit weight function.

Attention. As with PointConv's learned weights, PointConv with Attention's attention mechanism can also be demonstrated by visualizing the attention weights assigned to each unit in a neighborhood. We select a random timestep and three random units, then highlight all the other units in its neighborhood with a color corresponding to their attention weights. As our attention mechanism learns three different weights, each of which ranges between 0 and 1, we just display each triple of attention weights as its corresponding RGB color.

Figure 3.5 demonstrates that PointConv with Attention learns meaningful domain-specific attention weights. Specifically, the nearby enemy units clearly receive a lot of attention, while more distant units and friendly units tend to be less active. This lines up perfectly with what we expect, as the default behavior of all units in Starcraft 2 is to run towards and attack any enemy unit within close range. The significant performance gain over vanilla PointConv as well as the attention mechanism's clear understanding of the dynamics of the domain clearly demonstrate that this architecture is a good fit for the Starcraft 2 domain.

3.7.2.2 Query Timestep Distribution

One of the most notable features about our spatio-temporal problem setting is that the input and query target timesteps are not fixed. Rather, the model's capabilities and



(a) Layer 1 - Filter 1 (b) Layer 1 - Filter 2 (c) Layer 1 - Filter 3

(d) Layer 2 - Filter 1 (e) Layer 2 - Filter 2 (f) Layer 2 - Filter 3

(g) Layer 2 - Filter 1 (h) Layer 2 - Filter 2 (i) Layer 2 - Filter 3

Figure 3.4: Filters learned by a PointConv model trained on the Starcraft 2 prediction problem.

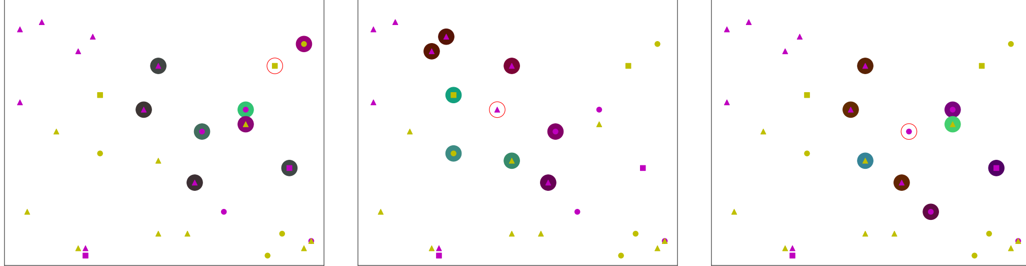


Figure 3.5: PointConv Attention visualization. The 'main' unit is circled in pink. Each unit in the neighborhood is highlighted with a solid circle colored according to its attention weights. Each unit icon's shape indicates its unit type, while its color indicates its team.

performance depend on the distribution of inputs and queries it was trained on.

We examine the 'out-of-bounds' behavior of these models by evaluating them on a modified Starcraft II dataset in which they must predict the future state of timestep offsets that were not present in the training dataset. Specifically, we show their average prediction loss for each query target timestep from $T+1$ to $T+12$, despite the fact that the networks were only trained on targets $T+[1,2,4,7]$.

Additionally, we train these models in two addition settings: 'TwoPred', in which they are trained on a modified dataset where the query target offsets are set to $T+[2,7]$; and 'OnePred', in which the query target offset is only $T+7$. Figure 3.6 shows a bar chart demonstrating the loss for each model trained on each query distribution on these out-of-bounds query target timesteps.

GraphConv. The GraphConv model appears to be least affected by being trained or evaluated on differing query timesteps. Despite no GraphConv having never observed any query timesteps above $T+7$, they do not exhibit any extreme prediction errors that are indicative of overfitting. However, the performance of GraphConv models seems to increase when trained on smaller number of query target timestep offsets. This is not desirable behavior, since we would hope that giving the models more information on how the units change over time would lead to a better model of the units' behavior. Instead, the 'OnePred' model has clearly specialized to only focus on the specific timestep it

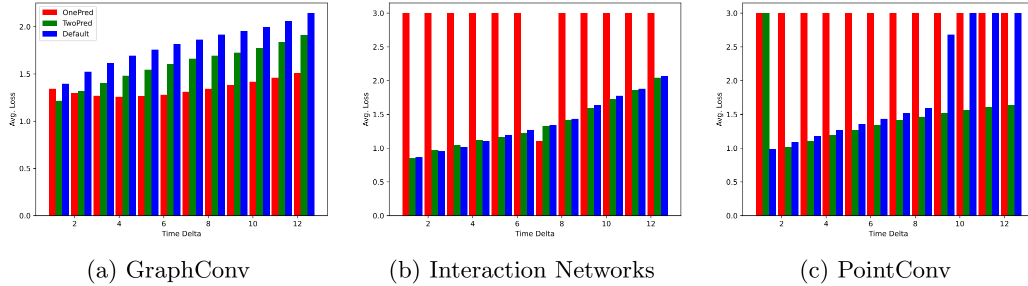
was trained on $(T + 7)$, and performs slightly worse on any other timestep. One would expect that 'TwoPred' should out-perform 'OnePred' on timestep $T + 2$, as 'OnePred' never been trained on targets at $T + 2$. Instead, 'TwoPred' preforms worse at almost all timesteps. Similarly, the 'default' setting (training on $T + [1, 2, 4, 7]$) performs worse than its more restricted counterparts despite effectively receiving more training data.

This suggests that GraphConv is not able to effectively distinguish between the different query targets it is trying to predict. This aligns with our intuition that GraphConv is significantly hampered by being unable to explicitly exploit edge features in a graph. Instead, the GraphConv model seems to have fallen back to underfitting behavior, in which it is unable to distinguish between the different query timestep offsets it must predict. In this mode of operation the loss would be expected to decrease as the diversity between the training targets decreases, which is exactly what we observe. This result shows that GraphConv is not an effective model to employ in this setting, and likely is ineffective on most graph problems with meaningful edge features.

Interaction Networks. The interaction networks' performance is nearly indistinguishable across all timesteps when trained on the default and 'TwoPred' settings. This suggests that both networks are able to learn similar, meaningful unit state transition models despite being trained on different query target timesteps.

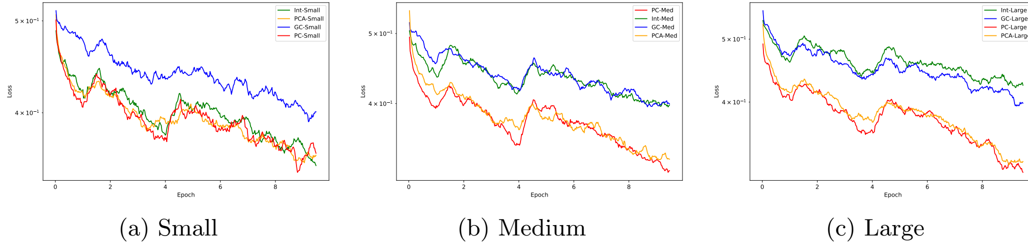
However, the interaction network trained on the 'OnePred' setting has clearly overfit. It is effectively useless at predicting unit states at any timestep other than $T + 7$, and only performs slightly better than the other models at its one training target $T + 7$ itself. Note that adding just one additional query target during training (that is, query target offsets $T + [2, 7]$ instead of just $T + 7$) causes the model to go from overfitting on a single timestep to learning a general transition model which performs reasonably well across all timesteps. This seems to validate that applying graph models to this kind of graph realization of a spatio-temporal process is an effective way to enable the models to understand the process' dynamics in general.

PointConv. The PointConv networks' behavior is similar to the interaction networks' behavior, but PointConv appears to be more prone to overfitting behavior on unseen timesteps. The model trained on the default setting performs well up until $T + 8$, the first timestep it has never encountered during training, after which the prediction error rapidly increases. Alternatively, the model trained on the 'TwoPred' setting (that is, only on $T + [2, 7]$) exhibits overfitting behavior at the very earliest timestep it's never



(a) GraphConv (b) Interaction Networks (c) PointConv

Figure 3.6: Plots showing the average prediction loss for each model type for each timestep offset between T+1 and T+12.



(a) Small (b) Medium (c) Large

Figure 3.7: Average training loss plots for all instances of each model trained on the traffic prediction dataset.

seen before ($T+1$), but appears to do a better job at making coherent predictions for more distant targets compared to the default setting. Finally, the model trained on the 'OnePred' setting has completely failed to train and produces erroneous predictions at all timesteps on the test problem instances despite achieving reasonable performance during training.

Both these behaviors – a discrepancy between training and test performance, and training on more data resulting in worse general-case performance – are indicators of overfitting. This suggests that PointConv's learned filters, that it must rely on to reason about relationship between queries, is not as effective at generalization as the interaction network approach of replacing the filtering mechanism with a black-box MLP.

Network	P_{25}	P_{50}	P_{75}	Mean
Nearest	0.75	2.10	5.88	7.59
GC-Small	1.78	3.55	7.60	**7.54**
Int-Small	1.69	3.35	6.73	**7.15**
PC-Small	1.77	3.40	6.79	**7.26**
PCA-Small	1.44	2.99	6.62	**7.02**
GC-Med	1.85	3.64	7.52	**7.57**
Int-Med	1.67	3.43	7.50	**7.43**
PC-Med	1.28	2.77	6.30	**6.90**
PCA-Med	1.30	2.83	6.37	**6.92**
GC-Large	1.26	2.92	7.46	**7.31**
Int-Large	2.69	4.81	8.85	8.45
PC-Large	1.31	2.84	6.36	**6.99**
PCA-Large	1.50	3.13	6.55	**7.14**

Table 3.8: Table showing the 25%, 50%, 75% percentile, and mean prediction error for each model in MPH. Bolded entries indicate the model beat the baseline's performance.

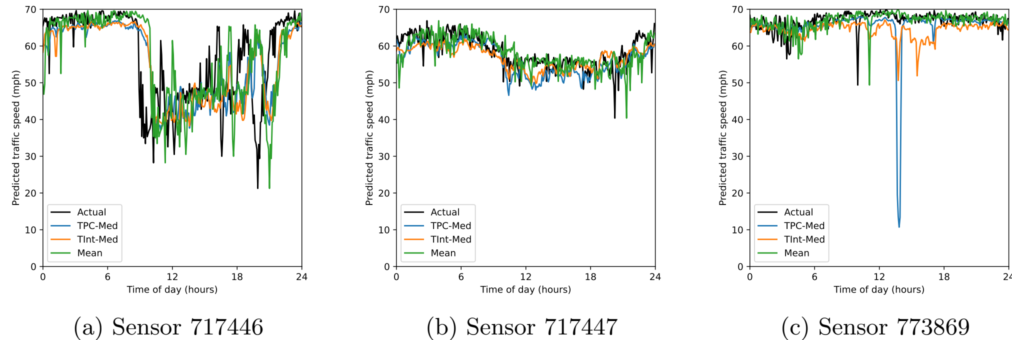
3.7.3 Traffic Prediction

Training Curves. Figure 3.7 shows the average training loss across all training runs for each architecture instantiation on the traffic prediction dataset. Based on the lack of clear convergence among any of the models trained, it seems likely that the models could be significantly further improved with additional training time.

Baseline Comparison. To evaluate the models' usefulness, we compare its average performance to a simple baseline. Specifically, we use the baseline which simply predicts that the traffic speed at any given sensor in one hour will be the same as the sensor's current recorded speed. Interestingly, while this baseline model significantly outperforms all deep models in its first, second, and third quantile performance, the deep models significantly outperform the dumb baseline on average.

Individual Sensor Predictions. To gain a better understanding of the deep models' behavior and performance, we plot the predictions of each medium-sized network for a single sensor throughout one entire day. By comparing the predicted traffic speed to the actual signal, we may be able to gain some insight into how or why each model is failing.

Figure 3.8 shows an entire day's worth of predictions from each model for three



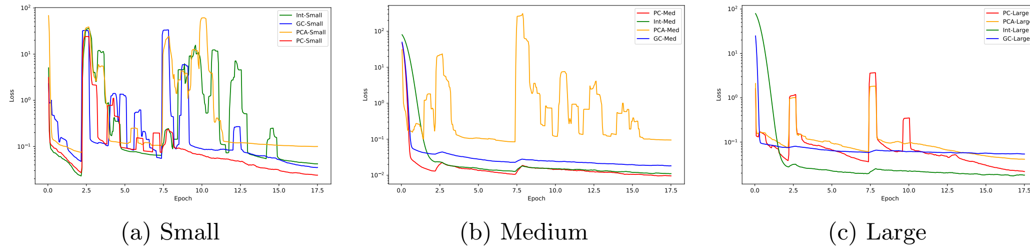
(a) Sensor 717446 (b) Sensor 717447 (c) Sensor 773869

Figure 3.8: Each model's predicted traffic speed throughout an entire day for individual sensors. Black line is the target signal.

different sensors. The one-hour delay from the baseline model is clearly visible, causing it to always miss quick changes in traffic speed by an hour. However, the deep models aren't much better at this. For example, in Figure 3.8a, while they seem to start predicting a declining speed well before the baseline is able to, indicating they're using information from surrounding sensors to detect the oncoming traffic jam before it can be observed at the sensor, the prediction is still far off from the actual speed at that time and the model clearly 'follows' the baseline model's plunge in predicted speed as soon as it has access to sensor readings showing that traffic is stopped now. Additionally, the deep models seem to constantly predict slightly too low of a speed, as if they are hedging their bets expecting a traffic jam to materialize. These two observations may be a demonstration of the commonly observed phenomenon in which graph networks tend to 'underfit' or produce decent but clearly biased results in this style of prediction problem. It's unclear whether this issue may disappear with more training or if it is inherent to the GNN architecture itself.

3.7.4 Weather Nowcasting

Training Curves. Figure 3.9 shows the average training loss across all training runs for each architecture instantiation on the weather prediction dataset. However, these training plots clearly indicate there's significant training instability. This suggests a failure



(a) Small (b) Medium (c) Large

Figure 3.9: Average training loss plots for all instances of each model trained on the weather dataset.

in the learning rate test's ability to consistently determine appropriate hyperparameters for training. Clearly the learning rate is significantly too high for much of the training– but once it lowers towards the end, the networks' behavior seem to recover.

Station Dropout. In domains such as Starcraft II each observation directly represents an individual unit's state. Adding or removing a unit and its corresponding observations from a Starcraft II prediction problem will significantly change the expected dynamics of the process, since the units' behavior is highly dependent on the presence of other units in the scene. In contrast, in the weather problem domain each observation from a station is a point sample of the underlying continuous process of interest, the atmospheric conditions within the area. Since we cannot directly observe the entire the process, models must instead infer its overall state from these individual point samples.

Ensembling is a common technique to increase prediction performance by aggregating several predictions from multiple models instead of relying on a single model's prediction. Usually this is accomplished by training multiple models and averaging their predictions together. The graph structure of our problem and our understanding of the semantics of the domain suggest a different approach. We can instead provide a single model with several augmented problem instances whose input data has been modified while the queries are fixed. This single model's predictions on different realizations of the original input data can then be averaged together to perform 'self-ensembling'.

In this domain, we can augment a problem instance by randomly removing a percentage of the input samples while keeping queries unmodified. We demonstrate this self-ensembling approach by having each model produce prediction for five augmented problems derived from the original problem instance, and average the model's predic-

Test-time Dropout	0%		10%		20%		50%	
Network	P_{50}	Mean	P_{50}	Mean	P_{50}	Mean	P_{50}	Mean
GC-Med	2.57	4.64	2.55	4.59	2.72	4.87	3.47	7.54
Int-Med	1.46	3.23	1.85	3.76	2.30	4.45	5.61	9.13
PC-Med	0.78	1.89	1.21	2.56	1.99	3.62	11.02	24.82
GC-Med-Drop20	2.93	5.18	2.72	4.92	2.68	5.06	3.30	8.51
Int-Med-Drop20	1.22	2.90	1.40	3.09	1.61	3.37	2.40	4.61
PC-Med-Drop20	0.90	1.97	1.03	2.21	1.21	2.54	2.17	4.10

Table 3.9: Table demonstrating the change in each model's test loss as station dropout is increased. Columns show the median and mean loss as the test-time station dropout is raised from 0% to 50%. Models with Drop20 appended to their name were trained with 20% station dropout.

tions together to produce the final prediction. Table 3.9 shows the mean and median performance of each model architecture as the percentage of dropped input samples is set to values between 0% (no augmentation) and 50%. We also show the performance of each architecture when trained with a 20% drop rate (models in the table postfixed with Drop20).

Among all models, increasing the amount of test-time station dropout generally decreases the model's prediction performance. However, the interaction network models trained in the 20% dropout setting significantly outperform the interaction network model trained on the un-augmented dataset regardless of the amount of test-time dropout. PointConv trained with dropout performs slightly worse than its default setting at 0% test-time dropout, but as the amount of test-time dropout increases it demonstrates a significant advantage over the model trained in the default setting. This is most obvious at 50% dropout where the default PointConv appears to be exhibiting overfitting behavior and producing erroneous predictions, while the model trained with dropout is able to outperform every other model.

These results show how applying appropriate graph data augmentation during training and evaluation effects the performance of these graph models. The models' overall performance did not significantly improve when only test-time data augmentation was applied. However, training with this data augmentation does not significantly negatively impact the models' best-case performance while greatly increasing their ability to make effective predictions as the characteristics of the input data changes.

3.8 Evaluation Platform

One of the main goals of this work is to provide an approachable software platform to allow others to fully reproduce the experiments run for this paper, or modify and extend them if desired. In contrast to some other research codebases, we define general implementations of each model type and problem domain. Our training engine loads human-readable configuration files which describe the desired configurations for the model, problem, and training hyperparameters for the experiment it represents. This approach of separating the model and problem implementations from the specification of each experiment or training run reduces the barriers to running large sets of diverse experiments, such as those examined in this paper. Specifically, we have published a codebase which includes:

- Scripts to fetch each dataset used in the evaluation;
- Implementations of dataset loaders which derive graph representations of spatio-temporal problems from the raw datasets;
- Implementations of GraphConv, PointConv, Interaction Networks, and all of their components;
- Experiment definition files which configure the datasets and networks to train all the models used in this evaluation;
- Scripts to collect the results from training and produce all the plots and tables included in this paper;
- Instructive documentation on how to set up and run the code.

The repository for this project can be found at .

3.9 Conclusion

We proposed a simple procedure to encode a spatio-temporal problem using a graph structure, including describing dynamic domain-specific queries. This graph approach enables us to describe a variety of problem types in one common format. Additionally, the graph structure allows for trivial data augmentation (when supported by the domain's

semantics) and the query structure allows one to define multiple simultaneous prediction targets which force the models to learn the underlying process dynamics rather than a single relationship between the input data and the desired prediction.

We extended the 'learning rate test' concept, showing how it can be modified to more robustly identify an effective learning rate region for each individual model instantiation. This effort allowed us to determine an effective training hyperparameter region for each individual model instantiation on each problem type, which is a process that would have otherwise been prohibitively expensive. However, we find that while this approach was reasonably effective in practice it still has significant shortcomings that caused undesirable training behavior in some cases.

Of the graph models we evaluated, we found that GraphConv consistently performed the worst, seemingly underfitting to most problems as it was unable to demonstrate it had learned the dynamics of the problems it was trained on. This is almost certainly owing to the fact that our graph realization of spatio-temporal problems stores relative information about neighboring nodes in edge features connecting them. Since GraphConv cannot exploit these edge features, it cannot take advantage of this useful bias.

Interaction Networks and PointConv seem to make meaningful predictions on all domains. This demonstrates that they are able to learn the dynamics of the problem they are trained on in some useful way. As PointConv learns an explicit weight function to filter neighboring nodes in the graph, we can visualize and interpret this weight function to validate that it is appropriate for the domain. Interaction networks are largely composed of black-box functions, and lack this interpretability. However, they perform slightly better than PointConv on most tasks and seem to generalize better to previously unseen queries. We show how PointConv can be augmented with an attention mechanism to bring its performance closer to interaction networks' performance without sacrificing intrepretability. However, it seems that in general if this interpretability is not needed, interaction networks are generally the most appropriate model architecture to apply to the spatio-temporal problems we evaluated.

Finally, we provide the codebase used to implement, define, perform, and evaluate every experiment presented in this work. The codebase is designed to be approachable and extensible, to allow and encourage interested parties to validate our results or modify and extend our experiments for further investigation.

Chapter 4: Conclusion

In the work presented in Chapter 2, we evaluated several black box optimization algorithms belonging to two different classes of approaches. Our goal was to validate or examine the claims being made that the faster and yet ostensibly more performant partitioning-based optimization algorithms could consistently out-perform the state of the art Bayesian optimization approaches. We were surprised that our experimental findings demonstrated that essentially the opposite was true. Our stake-free evaluation instead demonstrated that despite being much faster, the partitioning based methods were consistently unable to approach the performance of the state of the art approaches even when given significant computational advantages.

This experience underlines how easy it can be to let this type of bias sneak into our publications, especially when the incentive structures surrounding the publication process generally do not reward researchers for thoroughly optimizing the performance of the 'baseline' algorithms they are comparing against and hoping to beat. Most interesting were the claims that the partitioning algorithms were able to beat the BO-UCB algorithm, was widely presented as the most effective flavor of Bayesian optimization. In our investigation, we found that BO-UCB's performance was almost entirely dependent on setting the value of its β hyperparameter correctly. 'Good' values would allow it to be competitive with all other algorithms while 'bad' values caused it to fall behind almost all other algorithms. However, the 'good' values seemed to significantly change as the dynamics of the objective function being optimized changed, making it exceptionally difficult to choose an effective value for β for an unknown objective function in a true black-box setting.

The papers proposing partitioning-based optimization algorithms which used BO-UCB as a comparison seemed to just use a value for β that had been suggested by other works. They did not demonstrate that this selection was optimal or acknowledge the potential impact that different β values could have on the performance of the algorithm. This may be because they simply were not incentivized to devote space in their paper describing this issue, or because they lacked the time or resources to perform such in-

vestigations themselves. Regardless, it's not clear how close to 'state of the art' their instantiation of the BO-UCB algorithm really was. As these works explicitly did not provide any implementation of their proposed algorithm, the only way we were able to resolve this confusion was by re-implementing their algorithm ourselves and evaluating it alongside other similar algorithms across a wider variety of settings. This experience was a major influence on how we designed and implemented our research and evaluation platforms for future works.

The software platform we used in the work presented in Chapter 3 is intended to avoid these challenges to the greatest extent feasible. This is accomplished by ensuring that every experiment we run is explicitly clearly defined in advance and repeatable with minimal human intervention, making them cheap and easy to execute. This allowed us to define an objective procedure for determining appropriate training hyperparameter values for each model instance applied to each domain. As a result, rather than just stating the hyperparameter values we used we are able to justify their selection and allow others to easily perform the same procedure themselves to validate our findings. Using our learning rate test approach was crucial in enabling us to train a large number of network architectures on several significantly different domains. Other common approaches to optimizing hyperparameters, such as grid search, would have been computationally infeasible for us to perform considering the large number of unique network/problem pairs included in our work.

However, our modified learning rate test still had clear shortcomings. For example, it did not seem to select an appropriate learning rate range for the weather datasets especially. Figure 3.9 shows extremely unstable training behavior, suggesting that we have not found an appropriate hyperparameter configuration for training these networks. This shows that the learning rate test we employed cannot be a 'one size fits all' approach. While most of the domains and networks trained well under their experimentally derived learning rate ranges, the fact that several networks' prediction loss exploded during training suggests that this approach is not stable in general. Specifically, it seems that a procedure for determining training hyperparameters cannot take place entirely independently from the training procedure itself. As the network is trained, its dynamics change significantly, and therefore its effective ranges of hyperparameter values may significantly change as well. This suggests that future approaches to automatically determining training hyperparameters should be run concurrently with the training itself

rather than exclusively beforehand, but we leave further investigation of these challenges and ideas to future works.

After performing the evaluation of the graph network algorithms on our spatio-temporal problems, we found that there were significant differences between the performance and behavior of the algorithms we evaluated. GraphConv performed worst across the board. This is not an extremely surprising result, since we expect that the edge features explicitly describing the relationship between nodes in the graph are crucial to effectively reasoning about the effects the nodes may have on each other.

Among the edge-aware graph architectures, interaction networks and PointConv seem to exhibit similar median-case performance, with interaction networks slightly coming out ahead. One key difference between them is their interpretability. While interaction networks reason about nodes' neighbors with black box functions, PointConv learns an explicit weight function which is used to filter the values of the nearby nodes in the graph. This weight function can generally be visualized to validate that the model is learning a domain-appropriate way of reasoning about the relationship between neighbors.

However, PointConv is much more prone to overfitting behavior than interaction networks seem to be. Across multiple domains, we can observe PointConv making extremely erroneous predictions, especially when reasoning about relationships that it had not seen during training. In contrast, interaction networks generally did not exhibit overfitting behavior. Interaction networks seem to perform better on average, and generalize better without being more expensive computationally than PointConv. As a result, unless it's necessary to visualize on apply constraints to PointConv's learned filter function, it seems likely that interaction networks are generally the most correct choice for reasoning about this kind of graph problem.

Our final key contribution from this work is to release the software platform used to define the models, domains, and experiments presented in Chapter 3. Chapter 5 contains a brief tutorial describing the steps to take to extend the software platform to support additional problem types and additional models. Our hope is that interested practitioners are able to use this platform to validate our results or use it as a base for their own experimental research can be similarly reproducible by taking advantage of the tools we provide.

Chapter 5: Tutorial

To facilitate the use of the software platform developed for the evaluation in Chapter 3, this chapter is a brief tutorial on how to extend the platform by adding additional problem types and network types, creating experiment definition files to train a network, and using the evaluation scripts plot their training performance.

There are x key modules to understand:

- Problem class
- Network class
- Experiment definition file
- Training engine
- Evaluation scripts

This chapter will step through implementing or using each one to allow the software platform to train networks on the CIFAR10 and CIFAR100 datasets.

5.1 Problem Class

The first step is to define a Problem class for the prediction problem we want to train the networks on. The problem class defines the training and validation datasets to use during training, the loss function to use to evaluate networks' predictions, and (for more complex datasets) a collation function to gather individual training instances into a batch.

Every class defining a problem must be derived from the base Problem class defined at problem/base.py:Problem. This superclass registers the name of all its child Problem classes so that they can be looked up via a string when defining an experiment later.

Each problem class must have these five functions/variables:

- An init function, which will be provided arguments by the experiment definition file.
- self.train_dataset, an instance of torch.Dataset to use to train the network.
- self.valid_dataset, an instance of torch.Dataset to use as validation data.
- self.collate_fn, an optional function that define how to gather individual Dataset elements into a batch. Usually set to None if the default collation logic is sufficient.
- loss(), a function which calculates the loss of the network's predictions using the ground truth from the dataset.

An implementation of a problem class for the CIFAR datasets is as follows:

```
# problem/cifar.py
from .base import Problem

import torch.nn.functional as F
from torchvision.transforms import ToTensor
from torchvision.datasets import CIFAR10, CIFAR100

class CIFAR(Problem):
    def __init__(self, data_path, classes):
        assert classes in (10, 100)
        cifar = CIFAR10 if classes == 10 else CIFAR100
        cifar_args = {'root': data_path, 'download': True,
                      'transform': ToTensor()}
        self.train_dataset = cifar(**cifar_args, train=True)
        self.valid_dataset = cifar(**cifar_args, train=False)
        self.collate_fn = None

    def loss(self, item, pred):
        labels = item[1]
        loss = F.cross_entropy(pred, labels)
        return loss
```

This problem class has two parameters that will be specified by the experiment configuration: data, defining the path where the CIFAR data is stored; and classes which allows the user to choose between loading data from the CIFAR10 or CIFAR100

datasets. More complex problems are likely to have more parameters that have a greater impact on how data is loaded or presented.

Finally, the problem class has to be imported in problem/__init__.py so that is it loaded and available for lookup:

```
# problem/__init__.py
from . import base
from .cifar import CIFAR # <- Add this to enable the newly defined
                                            problem!


def make_problem(problem_args):
    problem_name = problem_args['problem_type']
    kwargs = {k: v for k, v in problem_args.items() if k != 'problem_type
                                                 '}
    return base._problem_map[problem_name](**kwargs)
```

5.2 Network Class

Network classes must be derived from the Network class in nets/base.py. A network class must have these three functions:

- an init function, which will be provided arguments by the experiment definition file.
- get_args(), a function which extracts the appropriate input data from a dataset instance and returns them as a list of arguments.
- forward(), which is called with the list of arguments from get_args() and should return a prediction for the provided input data.

The network we implement will consist of some number of 2D convolution layers, each with different kernel sizes and number of output channels. The encoded image will then be flattened and passed to an MLP with some number of layers which will produce the final prediction scores for each class. Since we want the exact size of each of these components to be user-configurable, the parameters conv_sizes, kernel_sizes, mlp_hidden,

and pred_size should all be parameters of the network class. The implementation then would look as follows:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from .base import Network

class CIFARConv(Network):
    def __init__(self, conv_sizes, kernel_sizes, mlp_hidden, pred_size):
        super().__init__()
        img_dim = 32
        in_size = 3
        layers = []
        assert len(conv_sizes) == len(kernel_sizes)
        for out_size, kernel_size in zip(conv_sizes, kernel_sizes):
            layers.append(nn.Conv2d(in_size, out_size, kernel_size))
            layers.append(nn.ReLU())
            in_size = out_size
            img_dim = img_dim - kernel_size + 1
        self.conv = nn.Sequential(*layers)

        in_size = in_size * img_dim ** 2
        layers = []
        for out_size in mlp_hidden:
            layers.append(nn.Linear(in_size, out_size))
            layers.append(nn.ReLU())
            in_size = out_size
        layers.append(nn.Linear(in_size, pred_size))
        self.pred = nn.Sequential(*layers)

    def get_args(self, item):
        imgs = item[0]
        return [imgs]

    def forward(self, imgs):
        latent_img = self.conv(imgs)
        latent = latent_img.reshape(latent_img.size(0), -1)
        preds = self.pred(latent)
        return preds
```

At a high level, the init function uses the user provided definition of the desired shape of the network to construct individual layers which fit together. The get_args function gets the input data, and passes it along to forward() which finally executes the network and returns a prediction.

5.3 Experiment Definition File

With the problem and networks implemented, we now need to define an experiment which instantiates them with the desired arguments and then trains the network on the defined problem. Experiment files are defined using JSON, as it is human-readable and easy to load directly into python.

Here is an example of such an experiment definition file experiments/cifar10.json:

```
[experiments/cifar10.json]
{
  "output_path": "./output/cifar",
  "problem_args": {
    "problem_type": "CIFAR",
    "data_path": "./data/cifar10",
    "classes": 10
  },
  "entries": [
    {
      "name": "Demo",
      "train_args": {
        "epochs": 1,
        "report_every": 0.1,
        "valid_every": 1,
        "batch_size": 8
      },
      "net_args": {
        "net_type": "CIFARConv",
        "conv_sizes": [16, 32, 16],
        "kernel_sizes": [7, 5, 3],
        "mlp_hidden": [256, 64],
        "pred_size": 10
      }
    }
  ]
}
```

The experiment definition is a JSON document consisting of a single dictionary with the following entries:

- output_path, a string which describes where to output the trained network.
- probem_args, a dictionary describes the problem to train the networks on. The value of the problem_type key is the name of the problem class to instantiate. All other entries are provided directly to the problem class' init function as a set of keyword arguments.
- entries, a list of dictionaries where each entry defines a separate network to train. The networks defined in the list are trained sequentially. Usually there will only be one network in the list, unless you need to use the output of one training run as an input to the next training run. Each dictionary in the entries list has the following structure:
  - name, a string to identify the network being trained
  - train_args, a dictionary which specifies how this network should be trained. There are many different arguments that could be specified here, see section 5.4 for more information. However, the most common and important arguments are:
    - epochs, the number of epochs to train for before terminating.
    - batch_size, the batch size to use during training.
    - report_every, the period at which the training engine should calculate and report recent cumulative training loss.
    - valid_every, the period at which the training engine should run the network on the validation dataset and report the validation loss.
  - net_args, a dictionary that defines the network to instantiate. The net_type key describes which network class you want to instantiate, all other entries are provided directly to the network class' init function as a set of keyword arguments.

5.4 Training Engine

Once the problem and network are defined, they are passed to the training engine to start the training process. As would be expected, there are a large number of parameters to modify here. Parameters such as the learning rate, learning rate schedule, reporting frequency, gradient clipping, weight decay, and more can be set by adding their value to the train_args dictionary in the experiment file. This part of the platform especially is intended to be modified and updated to meet the training needs of the project.

Finally, we have everything needed to perform a training run: the problem implementation, the network implementation, and an experiment file defining the desired instances of each. To execute the experiment file and train the network as described, we simply call the software platform's entry point with the experiment file as its argument:

This will execute a training run as requested, and once it terminates the trained network and statistics about the training process will be written to an output python pickle (.pkl) file at the requested location.

5.5 Evaluation Scripts

Once the desired training runs are complete, each training run will have a corresponding output file in the output folder specified by the experiment definition. These output files contain the trained network's final weights, the arguments used to instantiate the network, and information about the network's performance during training. These data can then be loaded and processed by evaluation scripts to examine the networks' performance, their behavior during training, or other investigations into interesting properties they may exhibit. For example, the eval.training script collects the training information from all output files in the specified folder, groups them by network type, and plots the average training and validation loss for each model type across their training runs. This lets you easily compare the training behavior of several models with very little effort.

5.6 Performing Experiments

With all the pieces in place, we can now go through the steps of defining, running, and evaluating experiments to test the properties of the problem domains and networks we've defined. For example, we may be interested in how the networks' behavior changes as we

increase the depth of the convolution layers or the MLP prediction layers. We can easily define these networks by modifying the experiment definition file we created earlier to represent these different network instantiations:

```
[experiments/cifar10-conv.json]
{
  "output_path": "./output/cifar",
  "problem_args": {
    "problem_type": "CIFAR",
    "data_path": "./data/cifar10",
    "classes": 10
  },
  "entries": [
    {
      "name": "Demo-BigConv",
      "train_args": {
        "epochs": 3,
        "report_every": 0.1,
        "valid_every": 0.5,
        "batch_size": 256
      },
      "net_args": {
        "net_type": "CIFARConv",
        "conv_sizes": [16, 16, 32, 32, 16, 16],
        "kernel_sizes": [7, 7, 5, 5, 3, 3],
        "mlp_hidden": [256, 64],
        "pred_size": 10
      }
    }
  ]
}

[experiments/cifar10-mlp.json]
{
  "output_path": "./output/cifar",
  "problem_args": {
    "problem_type": "CIFAR",
    "data_path": "./data/cifar10",
    "classes": 10
  },
  "entries": [
    {
      "name": "Demo-BigMLP",
      "train_args": {
        "epochs": 3,
        "report_every": 0.1,
```

```
        "valid_every": 0.5,
        "batch_size": 256
      },
      "net_args": {
        "net_type": "CIFARConv",
        "conv_sizes": [16, 32, 16],
        "kernel_sizes": [7, 5, 3],
        "mlp_hidden": [256, 256, 128, 128, 64, 64],
        "pred_size": 10
      }
    }
  ]
}
```

Then train the defined networks:

```
for f in $(seq 2); do
python -m pointnn experiments/cifar10.json
python -m pointnn experiments/cifar10-conv.json
python -m pointnn experiments/cifar10-mlp.json
done
```

Then visualize and evaluate the results of their training runs:

```
python -m pointnn.eval.training --out cifar-demo.png
```

This approach of explicitly defining and automatically instantiating networks and the problems to train them on makes it exceptionally easy to execute large sets of experiments that investigate the performance of different model structures, different model types, different problem domain configurations, et cetera. This flexibility and ease of use was a major factor in our ability to efficiently run a very large number of experiments for the evaluation in Chapter 3. We provide this software platform so that any interested parties have the opportunity to continue to build future research work within its framework. The code repository can be found at `https://github.com/Eiii/NNPlatform`.

Bibliography

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016.

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

Eric Brochu, Vlad M. Cora, and Nando de Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv e-prints*, art. arXiv:1012.2599, Dec 2010.

Max Horn, Michael Moor, Christian Bock, Bastian Rieck, and Karsten Borgwardt. Set functions for time series, 2019. URL `https://arxiv.org/abs/1909.12064`.

Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79 (1):157–181, 1993.

Kirthevasan Kandasamy, Jeff Schneider, and Barnabás Póczos. High dimensional bayesian optimisation and bandits via additive models. In *International Conference on Machine Learning*, pages 295–304, 2015a.

Kirthevasan Kandasamy, Jeff G Schneider, and Barnabás Póczos. High dimensional bayesian optimisation and bandits via additive models. In *ICML*, pages 295–304, 2015b.

Kenji Kawaguchi, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Bayesian optimization with exponential convergence. In *Advances in Neural Information Processing Systems*, pages 2809–2817, 2015.

Kenji Kawaguchi, Yu Maruyama, and Xiaoyu Zheng. Global continuous optimization with error bound and fast convergence. *Journal of Artificial Intelligence Research*, 56 (1):153–195, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106, 1964.

Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

David JC MacKay. Introduction to gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 168:133–166, 1998.

Ruben Martinez-Cantin. Bayesopt: a bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 15(1):3735–3739, 2014.

J. Mockus, V. Tiesis, and A. Zilinskas. The application of bayesian methods for seeking the extremum. In L.C.W.Dixon and G.P. Szego, editors, *Towards Global Optimisation 2*, pages 117–129. 1978.

Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.

Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 101, 2005.

Rémi Munos et al. From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning. *Foundations and Trends® in Machine Learning*, 7(1):1–129, 2014.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and

R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.

Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: no regret and experimental design. In *Proceedings of the 27th International Conference Machine Learning*, pages 1015–1022. Omnipress, 2010.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Ziyu Wang, Babak Shakibi, Lin Jin, and Nando de Freitas. Bayesian multi-scale optimistic optimization. In *AISTATS*, pages 1005–1014, 2014.

Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT Press Cambridge, MA, 2006.

Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9621–9630, 2019.

Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

Qi Zhang, Jianlong Chang, Gaofeng Meng, Shiming Xiang, and Chunhong Pan. Spatio-temporal graph structure learning for traffic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1177–1185, 2020.