Learning to Switch Optimizers for Quadratic Programming

Grant Getzelman GRANTGET@GMAIL.COM
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439, USA

Prasanna Balaprakash PBALAPRA@ANL.GOV
Mathematics and Computer Science Division &
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL 60439, USA

Editors: Vineeth N Balasubramanian and Ivor Tsang

Abstract

Quadratic programming (QP) seeks to solve optimization problems involving quadratic functions that can include complex boundary constraints. QP in the unrestricted form is $\mathcal{NP}$-hard; but when restricted to the convex case, it becomes tractable. Active set and interior point methods are used to solve convex problems, and in the nonconvex case various heuristics or relaxations are used to produce high-quality solutions in finite time. Learning to optimize (L2O) is an emerging approach to design solvers for optimization problems. We develop an L2O approach that uses reinforcement learning to learn a stochastic policy to switch between pre-existing optimization algorithms to solve QP problem instances. In particular, our agent switches between three simple optimizers: Adam, gradient descent, and random search. Our experiments show that the learned optimizer minimizes quadratic functions faster and finds better-quality solutions in the long term than do any of the possible optimizers switched between. We also compare our solver with the standard QP algorithms in MATLAB and find better performance in fewer function evaluations.

Keywords: Reinforcement Learning, Learning to Optimize, Quadratic Programming

1. Introduction

Modern optimization solver design is a challenging process. One needs to combine various mathematical intuitions with an extensive experimental process to produce novel algorithms. Learning to optimize (L2O) (Li and Malik, 2016) is an emerging subfield of machine learning where one attempts to learn a new optimization algorithm automatically. This approach has many positive elements, such as reducing the need for new mathematical ideas to produce improved solvers and providing a natural increase in quality that scales with data and compute resources. A significant advantage of learned optimizers is that if one has a known distribution of problems that need to be solved repeatedly, then learned optimizers can be adapted to that distribution and improve the solution speed and quality.

Quadratic programming (QP) is the task of solving optimization problems for quadratic functions with boundary conditions. Quadratic functions naturally arise in many settings, from agriculture to finance. QP is also a hidden workhorse of optimization because many

© 2021 G. Getzelman & P. Balaprakash.

trust-region methods use quadratic functions as models for more complex functions (Conn et al., 2009). In the trust-region case, the boundary conditions are that of an n-dimensional sphere. In our case, we focus on simple box constraints. Moreover, because of the complexity issues of nonconvex QP, it can be used as a model for hard optimization. Nonconvex QP hardness comes from the fact that the number of local minima can grow at an exponential rate, and there is no efficient way to guarantee that a local solution is a global one.

In this paper we develop an L2O approach that uses reinforcement learning to learn a stochastic policy to switch between pre-existing optimization algorithms in order to solve QP problem instances. In particular, our policy switches between three simple optimizers: Adam, gradient descent (GD), and random search (RS). Because QP is a classic and broadly applicable problem type, we use this as a conceptual framework to investigate the challenges of learning optimizers that scale across dimensions and problem complexity while having access to an endless amount of easily generated problem instances. We show that our learned optimizer outperforms each classical optimizer in our training set and generalizes well on the test problem instances.

L2O methods for solver design have attracted increasing attention within the artificial intelligence/machine learning (AI/ML) community. Some studies have focused on automating the design of methods for continuous variables in an attempt to outperform widely used approaches in deep learning such as stochastic gradient descent (Andrychowicz et al., 2016; Li and Malik, 2017, 2016). Other work has focused on learning combinatorial optimization heuristics that can outperform existing search methods (Chen and Tian, 2019; Khalil et al., 2017; Barrett et al., 2019). Although these studies have shown promising results on some tasks, they ignore centuries-long understanding of optimization theory. Our proposed approach is therefore orthogonal to current research trends in the sense that, instead of automating the optimizer design, we aim to improve the efficacy of existing solvers, which have been carefully designed based on rigorous mathematical foundations.

2. Problem Setting

We focus on solving QP problems defined by the following distribution and varying dimension.

$$\begin{aligned} \underset{x}{\text{minimize}}\, f(x) =& \frac{1}{2} x^T M x + b^T x \\ \text{where } M_{i,j}, b_i \sim & \quad uniform(-1, 1) \\ M^T =& M \\ -10 \leq& x_i \leq 10 \end{aligned} \tag{1}$$

Thus, $f : \mathbb{R}^N \to \mathbb{R}$ is an instance of a scalar value function we want to minimize, defined by the matrix M and vector b. Each of the values of M, b follows the uniform distribution above and thus defines the problem distribution. M is required to be symmetric, and each entry x is bounded in magnitude by 10. These bounds were the first and only ones we tested. We chose to exclusively use them under the principle that experimenter degrees of freedom introduce bias. We can think of our distribution as having two problem classes. In one problem class the eigenvalues of M are all either positive or negative, and in the second problem class the eigenvalues have mixed signs. We conjecture that as the dimension of M

increases, the probability that M is indefinite increases. We have tested this experimentally, and such a conjecture is supported by Wigner's semicircle law (Tao, 2012). Moreover, if we instead use the normal distribution in Eq. 1 for M, b, then the result is known (Dean and Majumdar, 2008). Thus, the percentage of clearly convex quadratic problems decreases, and the percentage of problems that are in the $\mathcal{NP}$-hard class increases as the dimension of M increases. This give us access to $\mathcal{NP}$-hard problems without explicitly excluding convex ones; but even though we are able to change the ratio of our problem classes, we are unable to truly control the average hardness of our problem. The downside of this problem distribution is that its dependence on dimension could add to the difficulty of generalization. On the other hand, for a classical optimizer it could introduce difficulties in selecting an effective budget for the search for high-quality local solutions in a fixed time horizon.

3. Learning to Switch Optimizers with Reinforcement Learning

We formulate the problem of switching optimizers as a Markov decision process (MDP), where an autonomous agent seeks to maximize its reward by repeated interaction with its environment (Sutton and Barto, 2018). Formally, an infinite-horizon MDP with discounted returns is defined as a tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{P}_0, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state-action-state transition probability matrix, $\mathcal{P}_0 : \mathcal{S} \to [0, 1]$ is the distribution over initial states, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, is the immediate reward function, and γ is a discount factor to bound the cumulative rewards and trade off how far- or short-sighted an agent is in its decision-making. The autonomous agent learns how to map the current state $s_k \in \mathcal{S}$ to an action $a_k \in \mathcal{A}$, by repeated interaction with the environment to minimize a performance measure, which is a function of the cumulative discounted rewards. In the context of switching optimizer, given a training problem instance $i \in I$, the current state s_k is the state of the solver at iteration k; the action a_k is the optimization update step at iteration k, namely, θ_k; and the immediate reward is the change in the objective function value of instance i, $f_i(x)$, which is to be minimized by the solver, relative to the best objective value found so far, $\mathcal{R}(s_k, a_k = \theta_k) = \min_{l<k} f_i(x_l; \theta_l) - f_i(x_k; \theta_k)$. The interaction with the environment of instance i consists of running the solver with action $a_k = \theta_k$, transitioning to a new solver state s_{k+1}, and receiving a reward $\mathcal{R}(s_k, a_k)$. The task is therefore to learn a stochastic policy $\pi(a_k|s_k)$, which is optimal with respect to the performance measure. That is,

$$\pi^* \in \arg\min_{\pi} \mathcal{C}(\pi), \tag{2}$$

where $\pi \in \Pi$ is a stochastic policy, Π is a set of all possible policies for a given solver, and $\mathcal{C}(\pi)$ is the overall selection criterion (empirical risk measure) that defines the best policy. This is given by

$$\mathcal{C}(\pi) = \mathbb{E}_{I,\mathfrak{C}}[c(\pi, i)] = \int_I \int_{\mathfrak{C}} c(\pi, i) \, \mathrm{d}P_{\mathfrak{C}}(c|\pi, i) \, \mathrm{d}P_I(i), \tag{3}$$

where I is a training set of optimization problem instances, $c(\pi, i)$ is the cost of the best solution found by running the solver configuration π on an optimization problem instance $i \in I$, $\mathfrak{C}$ denotes the possible values of c, P_I is a probability measure over the set I, and $P_{\mathfrak{C}}$ is a probability measure over $\mathfrak{C}$. Thus, $\mathcal{C}(\pi)$ is an expected value, where the expectation

is considered with respect to both P_I and $P_{\mathfrak{C}}$ and the integration is taken in the Lebesgue sense. We assume that the probability measures P_I and $P_{\mathfrak{C}}$ are not explicitly available and the analytical solution of the integrals is not tractable. Therefore, the integrals are estimated in a Monte Carlo fashion on the basis of a training set of optimization problem instances. Here, $c(\pi, i)$ is

$$c(\pi, i) = \mathbb{E}_{\pi,i}[\sum_{t=0}^{\infty} -\gamma^t \mathcal{R}(s_t, a_t^{\pi})]. \tag{4}$$

Note that by minimizing $\mathcal{C}(\pi)$, the agent learns to maximize the total expected discounted rewards, thus encouraging the agent to dynamically switch the optimizers to achieve the highest decrease in $f_i(x_k; \theta_k)$.

Given the set of training instances I, we seek to find a stochastic policy π^* that is expected to have the best performance over a set of possible instances. The key assumption is that training instances are representative samples from the whole set. We seek to learn from a moderately sized set of training instances and generalize to a possibly infinite set of unseen test instances. This generalization is justified by the assumption that the same probability measure P_I governs the selection of all the instances (Birattari et al., 2002): those used for selection and those that will be solved afterwards. The training instances are representative examples of the whole set of instances.

We specify the details of our implementation as follows. We use a discrete action space of three choices for $\mathcal{A}$, and we use a continuous observation space of scalars, in contrast to previous methods that focused on tensor inputs. Our observed values are the norm of the gradient, function value, step count, dimension, norm of M, norm of b, norm of x, and average norm of all the gradients up to and including the current step. We run the optimizer for a fixed 100 steps. We use the stable baselines framework (Hill et al., 2018) to implement our agent with OpenAI Gym (Brockman et al., 2016). We use the proximal policy optimization algorithm (Schulman et al., 2017) PPO2 from a stable baseline and the default multilayer perceptron agent, since it has good performance with minimal hyperparameter tuning. Our experiments with recurrent networks did not provide meaningful advantage to the collection of hyperparameters we tested, and again maximizing the performance of the optimizer by extensive hyperparameter tuning was outside our set of goals.

The last issue is how to combine Adam with the other optimization methods, because Adam uses the gradient variance and mean as inputs. Clearly, averaging the gradients after an RS step with those produced by Adam would provide little value. Thus, we reset the mean and variance computation RS step is called. In the projected gradient case, we compute the rolling gradient mean and variance and store those values in Adam. For the reinforcement learning reward, we pass the improvement over the best-known value of our objective function. This proved to be unstable in practice. We needed to normalize relative to each problem in order to get reasonable sample efficiency. We also noticed that passing a negative reward would result in early stopping. Thus we modified the reward in two ways. First, if the reward value is below zero, we pass zero. If we let x_{Adam} denote our best x value produced by Adam after 100 steps, then our reward at time step t can be expressed

as follows:

$$\mathcal{R}(s_k, a_k = \theta_k) = \begin{cases} \frac{\max\{\min_{l<k} f_i(x_l;\theta_l) - f_i(x_k;\theta_k), 0\}}{f(x_0) - f(x_{Adam})}, & \text{if} \quad f(x_0) - f(x_{Adam}) \geq 1 \\ \frac{\max\{\min_{l<k} f_i(x_l;\theta_l) - f_i(x_k;\theta_k), 0\}}{Dim*100*10}, & \text{if} \quad f(x_0) - f(x_{Adam}) < 1. \end{cases} \tag{5}$$

The motivation to provide this sort of reward is to encourage exploration; otherwise, the agent could have a wrong incentive to focus on getting as close as possible to one local minimum. The normalization process is as follows. First, we run Adam for 100 steps with the same starting point and divide each reward by the difference between the starting point value and the best value found by Adam if that is larger than one. If the value is less than one, we divide by the dimension of the problem times one hundred times the upper bound on x. We find experimentally that this type of normalization reduces the sample requirements for reinforcement learning by reducing the variance in rewards.

4. Experiments

Our goal in the experimental evaluation is to evaluate the efficacy of the RL-based switching optimizer on the QP problems and show that the switching optimizer outperforms each of the optimizers when considered in isolation and the widely used MATLAB QP solver quadprog. The code for all our experiments can be found at github.com/Grant-E-G/switching_opt.

4.1. Setup and description

We generate each problem sampled from the appropriate n-dimensional distribution as stated above. For training, we mix problems of different dimensions as follows. First, we select uniformly from the set of possible problem dimensions $\{5, 10, 15\}$, and then we produce a new problem based on the chosen dimension. We run our trainer for 8e7 steps on 800,000 unique quadratic problems. We then generate 10,000 new test problems for each test and evaluate on those. We run two types of tests: one in which all problems share the same dimension and the second a mixed-dimensional case where the problems are uniformly generated from a set of dimensions just as in training. In each test, we use this fixed number of problems, so in the $\{5, 10, 15\}$ mixed test we generate only 10,000 test problems, not 30,000. We give greater experimental details in our supplement.

Evaluating the performance of our trained agent presents some challenges, because problems of the same dimension can have wildly different local and global minima. Another problem is that most random starting points have different closest local minima. Moreover, we want to evaluate our agent's performance with respect to our distribution; we want to see that our agent performs well on most problems, not just on a few outliers. We address these issues in multiple ways. First, when we test our optimizers, we generate unique problems and starting-point pairs both sampled from a uniform distribution. We then feed each pair into our optimizers. If we instead used independently randomly generated starting points for each solver, we would need to increase our test cases dramatically as we have an unknown extra variance due to initialization. Since we are pairing problems and starting points, any unfairness needs to be seen at a distributional level, because we are looking at distributional performance measures. When we began training RLSO using uniformly random initialization, we decided to fix that hyperparameter instead of having another possible

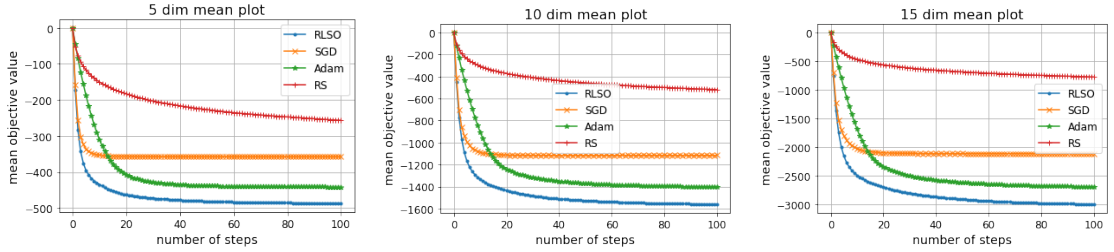
experimenter degree of freedom with the issues it would cause. Second, we use the idea of matched differences. Given the same problem and starting point, we subtract one solver from our learned solver iteration-wise. Concretely, this means that if our learned solver is better at a given timestep, then the reported value should be negative. Smaller negative values mean a larger performance gap in favor of our learned optimizer, and larger positive values mean a performance gap in favor of the alternative optimizer.

We also fix the step size of both Adam and gradient descent, since these values could both be considered hyperparameters controllable by the machine learning agent. We follow our general goal of reducing complexity by fixing them.

4.2. Comparison between RL-based switch and individual optimizers

Here, we show that our RL-based switch optimizer (RLSO) obtains better solution quality in shorter computation time compared with RS, GD, and Adam on our training set of dimensions.

We trained our RLSO on a uniform mixture of $\{5, 10, 15\}$-dimensional problem instances, and we then evaluated on the same dimensions but with new testing data. Unlike in training, however, we break testing into one testing set for each dimension. Figure 1 shows the mean performance for our optimizers attempting to minimize the quadratic problem instances. We compute the average performance with raw scores, not normalized ones. Thus, we average the value of the quadratic problem at each time step. Explicitly, if we index our quadratic problem by i, then at time step j we have the following formula: $\frac{1}{10000}\sum_i f_i(x_{i,j})$, where $x_{i,j}$ is our x value for the ith problem at time step j. We note that GD on average rapidly converges to a local minimum whereas Adam on average has a much slower convergence rate but finds a better local solution. RS is both slower and significantly worse than all methods tested, and we note that RS scales poorly with dimensions such that the performance gap increases. RLSO, on the other hand, finds higher-quality solutions with the speed of GD and outperforms all other optimizers on average. In our training set of dimension $\{5, 10, 15\}$, as dimensions of the problem increase, the average performance gap between GD, Adam, and RLSO stays at the same relative scale. However, the absolute value of the performance difference increases with dimension in favor of RLSO.



(a) (b) (c)

Figure 1: Average performance of the (a) 5-, (b) 10-, and (c) 15-dimensional problem instances up to 100 iterations.

We also explore the distributional properties of our RLSO by generating box plots. Because of space constraints, we show plots of relative performance only to Adam, since it was the second-best solver experimentally in our particular problem setting with our chosen hyperparameters. We use the matched difference and then look at the the distributional properties. At time step j, we compute $f_i(x_{i,j,RLSO}) - f_i(x_{i,j,Adam})$ and compute the quartiles Q_1, Q_2, Q_3, mean over the set of all f_i., where we have subindexed x by the problem, time step, and solver, respectively.

We see that RLSO in 5, 10, and 15 dimensions outperforms not only on average but in 75% of all test problems at each iteration. From this information we conclude that RLSO is not just producing high-quality outliers but is outperforming generally. We also note that as the dimension of our problem increases, the range between Q1 and Q3 increases in absolute terms. Thus, in higher dimensions, RLSO more frequently produces significantly better solutions at each iteration.

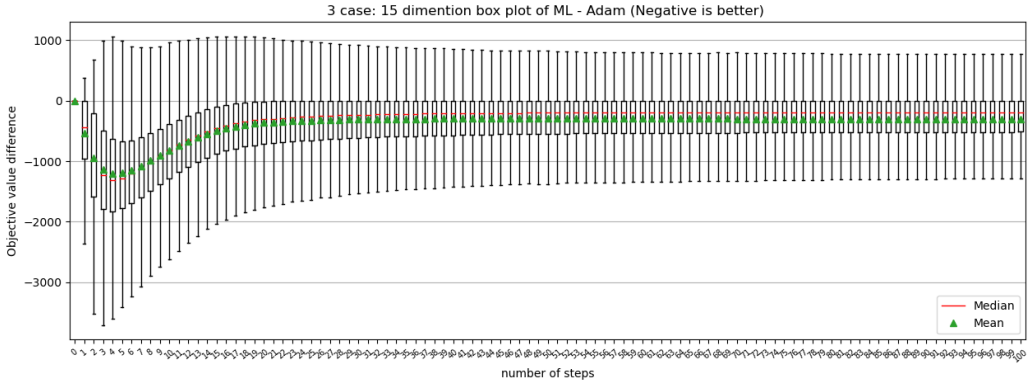


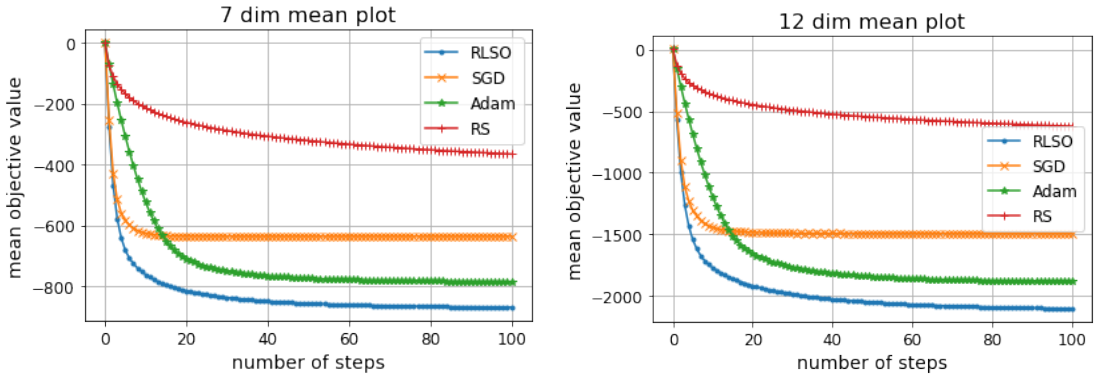
Figure 2: Box plot of the 15-dimensional test. RLSO outperforms Adam on average at each iteration, as demonstrated by the negative mean relative performance (green triangles). Furthermore, the third quartile is always below zero, indicating that RLSO outperforms Adam 75% of the time.

4.3. Interpolation

Here we show that our RLSO generalizes to new unseen problem instances with dimensions that are in between the training problems' dimensions. RLSO finds better solutions and in a shorter number of iterations than the other solvers do.

We examine how our previously trained RLSO (see Sec 4.2) performs on 7- and 12-dimensional problems. The experimental design is the same as in Sec 4.2, and all of the metrics are evaluated in the same way. Figures 3 and 4 show our mean performance and distributional performance over time, respectively. We see nearly identical relative performance between solvers as in the $5, 10, 15$-dimensional test problems, but at a different scale due to changes in the underlying dimensions. RLSO minimizes faster and to an overall better solution. RS performs the worst, and its relative performance becomes much worse as the dimension increases. The distributional results are similar to those of Sec. 4.2 in the relative performance but at a different scale due to the change in dimensions. We again

see that RLSO outperforms Adam on 75% of the test problems at each iteration for both 7 and 12 dimensions. Overall, for this type of generalization, RLSO performs as well as in our training set of dimensions.

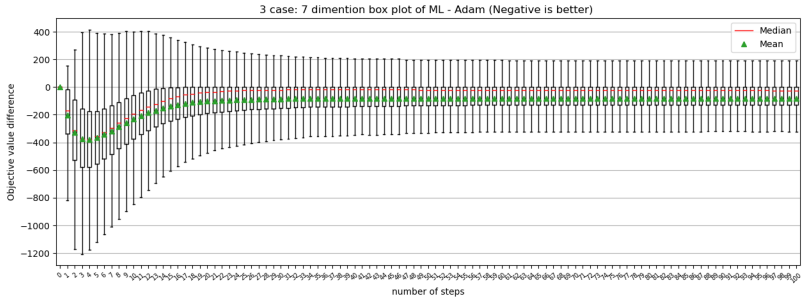


(a) (b)

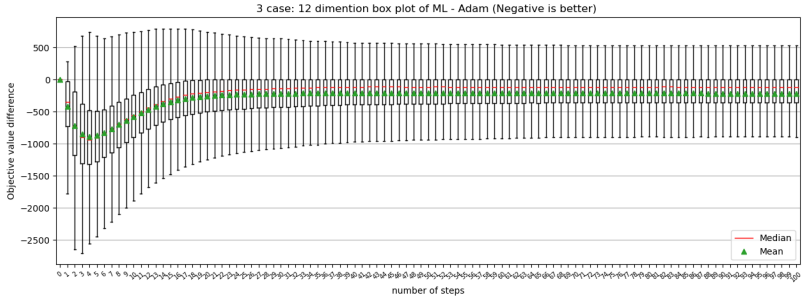
Figure 3: Mean performance of RLSO, SGF, Adam, and RS over 100 iterations for the (a) 7- and (b) 12-dimensional test cases. A more negative "mean objective value" indicates a better performance.

Dimension	Score - Adam	Score - GD	Score - RS
5, 10, 15 mixed	-211.67 σ = 278.16	-390.74 σ = 835.64	-1124.73 σ = 828.92
5	-59.76 σ = 76.6	-118.05 σ = 244.84	-249.11 σ = 114.98
7	-109.93 σ = 123.09	-198.24 σ = 389.66	-513.63 σ = 177.74
10	-203.7 σ = 204.42	-362.75 σ = 676.09	-1030.89 σ = 271.53
12	-273.63 σ = 273.02	-487.49 σ = 881.58	-1437.72 σ = 345.27
15	-387.85 σ = 367.2	-690.09 σ = 1205.36	-2138.52 σ = 440.8
18	-490.57 σ = 473.99	-926.75 σ = 1563.3	-2908.15 σ = 554.38
20	-571.96 σ = 549.95	-1087.58 σ = 1805.58	-3467.52 σ = 633.57
25	-739.99 σ = 779.21	-1579.81 σ = 2525.38	-4971.17 σ = 858.13
30	-876.39 σ = 1030.1	-2093.5 σ = 3295.68	-6607.67 σ = 1111.48
40	-1036.85 σ = 1621.53	-3184.2 σ = 4865.87	-10231.74 σ = 1731.03
50	-1042.84 σ = 2286.78	-4640.76 σ = 6756.92	-14186.0 σ = 2407.18
60	-712.25 σ = 3040.49	-6020.77 σ = 8691.89	-18340.04 σ = 3169.99
70	-182.36 σ = 3904.73	-7620.58 σ = 10878.37	-22686.6 σ = 4062.02
80	704.14 σ = 4897.9	-9077.15 σ = 12842.79	-27069.39 σ = 5035.23
90	1906.22 σ = 5857.77	-10731.48 σ = 15027.85	-31566.18 σ = 6036.55
100	3341.96 σ = 6933.57	-12368.93 σ = 17453.67	-36111.59 σ = 7074.06

Table 1: Full switching agent's mean score over time (computed using Eq. 6) with standard deviation σ. Negative scores are better. For larger dimensions, the same relative performance requires larger values in magnitude. Thus, having a similar score in 25 dimensions and 60 dimensions represents a much worse performance gap. Note that performance remains strong against both GD and RS



(a)



(b)

Figure 4: Box plot of the (a) 7-dimensional and (b) 12-dimensional test cases. At each iteration, RLSO outperforms Adam on average, as demonstrated by the negative mean relative performance (green triangles). Moreover, the third quartile is always below zero, indicating that RLSO outperforms Adam 75% of the time

.

Dimension	Score - Adam	Score - GD	Score - RS
5, 10, 15, 25 mixed	-352.04 σ = 513.53	-689.1 σ = 1531.57	-2094.48 σ = 1874.89
5	-60.0 σ = 77.81	-112.85 σ = 236.03	-250.96 σ = 115.89
7	-108.25 σ = 121.2	-203.47 σ = 400.28	-513.67 σ = 179.33
10	-204.54 σ = 207.73	-363.62 σ = 668.03	-1031.7 σ = 277.8
12	-278.43 σ = 267.13	-496.46 σ = 867.33	-1442.15 σ = 339.49
15	-390.65 σ = 369.16	-715.58 σ = 1222.02	-2133.8 σ = 438.94
18	-509.01 σ = 471.85	-925.36 σ = 1543.87	-2924.54 σ = 545.41
20	-584.88 σ = 544.76	-1127.59 σ = 1839.29	-3481.33 σ = 618.5
25	-772.94 σ = 759.98	-1610.89 σ = 2550.82	-4994.5 σ = 827.0
30	-962.49 σ = 976.35	-2121.95 σ = 3309.67	-6698.99 σ = 1043.53
40	-1217.75 σ = 1551.69	-3480.17 σ = 5011.33	-10392.43 σ = 1633.4
50	-1339.72 σ = 2229.7	-4997.26 σ = 6817.12	-14495.64 σ = 2346.82
60	-1298.46 σ = 2988.18	-6521.53 σ = 8661.62	-18889.76 σ = 3136.89
70	-984.16 σ = 3889.82	-8353.74 σ = 10736.68	-23485.81 σ = 4055.27
80	-427.65 σ = 4845.61	-10113.63 σ = 12758.81	-28198.62 σ = 5070.45
90	496.99 σ = 5905.6	-12341.07 σ = 15140.01	-33005.72 σ = 6057.65
100	1650.91 σ = 7032.15	-13989.36 σ = 17142.8	-37789.73 σ = 7209.86

Table 2: Retrained RLSO's mean score over time (computed using Eq. 6) with standard deviation σ. Negative scores are better. For larger dimensions, the same relative performance requires larger values in magnitude. RLSO-Adam's score improves significantly after retraining in the 40–100 dimension range. Note that performance also improves compared with that of both GD and RS.

4.4. Extrapolation

Here we test on {18, 20, 25, 30, 40, 50, 60, 70, 80, 90, 100}-dimensional problems and show that our RLSO performs well on unseen dimensions larger than our training data up to 80-dimensional problems (see Table 1). RLSO initially is faster and finds better solutions; but in dimensions higher than 80, RLSO's relative performance decays. Since we are dealing with $\mathcal{NP}$-hard problems, we expect all methods to have worse performance in solution quality as our problem dimension increases.

Our experimental setup is the same as in Sec. 4.2. Because of space constraints, we show only summary statistics of our experiments in a tabular format and describe some distributional properties. We use the following formula to compute our scores for the table:

$$\frac{1}{10000}\frac{1}{100}\sum_{i}\sum_{j} f_i(x_{i,j,RLSO}) - f_i(x_{i,j,solver}), \tag{6}$$

Where f_i is our collection of functions in our test set and j indexes our iteration number. We then compute the standard deviation across the inner sum.

From Table 1 we see that RLSO consistently outperforms both GD and RS. RLSO also outperforms Adam in the 18–60-dimensional range. We note that the box plots of our experiments show that RLSO outperforms Adam at all iterations up to 50 dimensions in the majority of test problems, but again we have been unable to include all of those plots. We begin to see our learned optimizer perform worse on average at 60-dimensional problems. Even then, RLSO is outperformed by Adam by a small amount in the step range of approximately 17–40 when we look at our box plots for that experiment. For 80-dimensional problems or higher, Adam is clearly outperforming on average in the medium and long term. Our optimizer consistently outperforms GD and RS for all dimensions we tested.

We next test whether our performance is due to a problem of generalizing to large-dimensional problems from small training examples or whether there is an intrinsic issue with our optimizer. We address this concern by simply increasing the collection of dimensions of our training set from $\{5, 10, 15\}$-dimensional to $\{5, 10, 15, 25\}$-dimensional problems and retraining our previously trained agent. For our retraining experiment, we reload our trained model and train it on a new 800,000 quadratic problem uniformly generated from $\{5, 10, 15, 25\}$. We show the summary statistics of this experiment in Table 2. We observe next to no difference in performance in the 5- to 25-dimensional range between the retrained RLSO and the original agent. Minor improvements are noticeable in the 30- to 50-dimensional test problems regarding quartile behavior; after that, the retrained optimizer is better on average in the 60-dimensional test cases. Likewise, our long-term behavior is better on average or equal up to 90-dimensional test problems. Thus, our extrapolation results before retraining are likely a product of our training data size and not some issues with our approach.

4.5. Comparison with two switching cases

Next we investigate the value added by each static optimizer. An initial conjecture after some exploratory experiments was that RS functioned as an option to restart at a new point if the current search did not seem promising enough. Our second hypothesis was

that Adam and GD with the specific hyperparameters we set to static values would provide access to different convergence behaviors to local minima. We believed that GD was faster but less accurate and Adam had better long-term behavior (slow convergence to better local solutions), thus motivating our desire to switch between the two. Given those conjectures, we trained three other agents, but this time with only two options to choose from: Adam & GD, Adam & Random and GD & Random. Next, we performed the same experiments with interpolation, retraining, and extrapolation as above.

Switching between Adam and GD We show the summary statistics in the leftmost subtable of Table 3. The overall performance was significantly worse, even after retraining, than that of our initial three switching optimizers in extrapolation. From a purely theoretical point of view, with perfect optimization our agent should never have an issue doing as well as any of the static optimizers in expectation. However, it seems clear from this experiment and others that the agents are learning behavior in low dimensions that do not generalize well without retraining. On the other hand, the only way our switching optimizer could outperform our best static optimizer, Adam, with access only to Adam and GD, is by correctly exploiting the advantages of one static solver over the other in some step-dependent fashion. Furthermore, by investigating the action distribution of our various agents, we do see a change in behavior across dimensions and step numbers. More important, none of the learned optimizers completely ignores one of the possible sets of available actions.

Table 3: Summary statistics for our restricted switching optimizers. Left: data for Adam and GD; right: data for Adam and RS. We can see that Adam and RS perform unexpectedly poorly in high dimensions. However, both of these restricted RLSOs still outperform GD and RS.

	RLSO: Adam and GD only			RLSO: Adam and RS only		
Dimension	Score - Adam	Score - GD	Score - RS	Score - Adam	Score - GD	Score - RS
5, 10, 15, 25 mixed	-145.42 $\sigma = 446.22$	-480.75 $\sigma = 1356.72$	-1899.46 $\sigma = 1761.39$	-32.15 $\sigma = 398.22$	-383.29 $\sigma = 1590.65$	-1762.79 $\sigma = 1576.39$
5	-19.18 $\sigma = 60.71$	-73.69 $\sigma = 227.51$	-210.19 $\sigma = 129.25$	-31.41 $\sigma = 75.18$	-86.85 $\sigma = 245.34$	-224.43 $\sigma = 115.38$
7	-36.36 $\sigma = 101.75$	-130.28 $\sigma = 368.19$	-440.93 $\sigma = 210.6$	-41.56 $\sigma = 125.51$	-129.49 $\sigma = 407.05$	-445.82 $\sigma = 177.78$
10	-77.21 $\sigma = 178.64$	-232.75 $\sigma = 618.61$	-901.39 $\sigma = 326.16$	-46.72 $\sigma = 208.59$	-207.73 $\sigma = 705.78$	-876.47 $\sigma = 268.69$
12	-110.31 $\sigma = 232.34$	-324.82 $\sigma = 801.61$	-1279.52 $\sigma = 410.55$	-47.34 $\sigma = 271.82$	-264.06 $\sigma = 924.31$	-1219.39 $\sigma = 333.84$
15	-169.39 $\sigma = 326.45$	-476.37 $\sigma = 1111.24$	-1920.29 $\sigma = 535.8$	-40.1 $\sigma = 362.45$	-331.74 $\sigma = 1257.03$	-1790.73 $\sigma = 418.25$
18	-220.7 $\sigma = 440.83$	-647.74 $\sigma = 1411.7$	-2634.45 $\sigma = 681.03$	-28.67 $\sigma = 448.44$	-463.05 $\sigma = 1670.95$	-2448.66 $\sigma = 521.35$
20	-251.54 $\sigma = 532.88$	-761.29 $\sigma = 1611.92$	-3150.8 $\sigma = 784.43$	-14.43 $\sigma = 510.16$	-549.11 $\sigma = 1941.95$	-2917.15 $\sigma = 583.28$
25	-321.64 $\sigma = 784.82$	-1163.38 $\sigma = 2270.41$	-4541.38 $\sigma = 1088.18$	19.14 $\sigma = 669.91$	-830.31 $\sigma = 2714.89$	-4203.11 $\sigma = 731.11$
30	-385.0 $\sigma = 1086.42$	-1583.15 $\sigma = 2904.94$	-6114.47 $\sigma = 1422.46$	57.5 $\sigma = 829.12$	-1134.13 $\sigma = 3468.62$	-5665.52 $\sigma = 895.07$
40	-395.16 $\sigma = 1832.61$	-2585.81 $\sigma = 4318.27$	-9598.73 $\sigma = 2296.22$	151.48 $\sigma = 1171.86$	-1970.63 $\sigma = 5193.29$	-9026.96 $\sigma = 1220.0$
50	-246.61 $\sigma = 2759.43$	-3797.87 $\sigma = 5817.81$	-13424.65 $\sigma = 3287.45$	268.82 $\sigma = 1453.98$	-3282.46 $\sigma = 6999.78$	-12894.06 $\sigma = 1516.93$
60	57.38 $\sigma = 3847.75$	-5371.03 $\sigma = 7490.82$	-17541.63 $\sigma = 4468.25$	408.77 $\sigma = 1806.53$	-4888.4 $\sigma = 9124.39$	-17217.32 $\sigma = 1846.58$
70	399.14 $\sigma = 4974.45$	-6870.58 $\sigma = 8990.63$	-22076.33 $\sigma = 5702.67$	593.39 $\sigma = 2164.3$	-6663.37 $\sigma = 11146.38$	-21956.68 $\sigma = 2172.3$
80	1159.67 $\sigma = 6220.38$	-8872.08 $\sigma = 10713.05$	-26636.19 $\sigma = 7095.49$	747.89 $\sigma = 2433.59$	-9067.86 $\sigma = 13376.8$	-27058.83 $\sigma = 2494.53$
90	1901.38 $\sigma = 7377.76$	-10686.87 $\sigma = 12455.67$	-31543.13 $\sigma = 8332.7$	936.19 $\sigma = 2770.54$	-11630.04 $\sigma = 15703.27$	-32502.3 $\sigma = 2841.04$
100	2900.73 $\sigma = 8808.37$	-12832.01 $\sigma = 14403.12$	-36527.55 $\sigma = 9916.24$	936.19 $\sigma = 2770.54$	-11630.04 $\sigma = 15703.27$	-32502.3 $\sigma = 2841.04$

Switching between Adam and Random Search Next, we compared our three-case switching optimizer to a switching optimizer with access only to random search and Adam as its possible actions. Since Adam was our best static optimizer that generalizes better to higher dimensions, the only way that this learned optimizer could outperform Adam would be if RS added value. Experimentally we see in the right subtable of Table 3 better performance in the low-dimensional range from 5 to 20, and we see that the action distribution favors Adam over RS in high dimensions. The particulars of the action distribution

show what looks like an initial searching phase, then a period of near-100% Adam and an increase in random actions again later in the tail. This agent seems to switch to using Adam almost exclusively as the dimension of the problem increases after an initial switching period. Given this result, we suspect that our agent cannot predict when restarting would be valuable or that restarting randomly is so unlikely to produce a better starting point on average that doing so is not worthwhile. A possible resolution to this issue would be to train an agent that has access to a more classical learned optimizer and use that as a way to find better-restarting locations.

Switching between GD and Random The last agent that we trained is restricted to using only GD and random search. This agent overall has the closest performance to our full 3-case switching optimizer; and before retraining the experiment on higher-dimensional data, the results were comparable. Unfortunately, because of space we cannot show those results here. By looking at both action distributions, however, one can see that the learned agents are not the same, since our 3-case switcher uses Adam 10–20% of the time depending on step number and dimension. Moreover, after retraining both agents, the 3-case switcher's performance is slightly better. We conjecture that GD could be exhibiting oscillatory behavior near a local minimum and that both restarting by RS and using Adam to get better local convergence are equally valid options in a lower dimension. In higher dimensions restarting provides less value, which results in the full 3-case switcher outdoing the restricted switcher.

Table 4: Retrained GD and RS agent's mean scores over time (computed using Eq. 6) with standard deviation σ. Note that this restricted agent's scores are good but, after retraining, the original full 3-case RLSO outperforms.

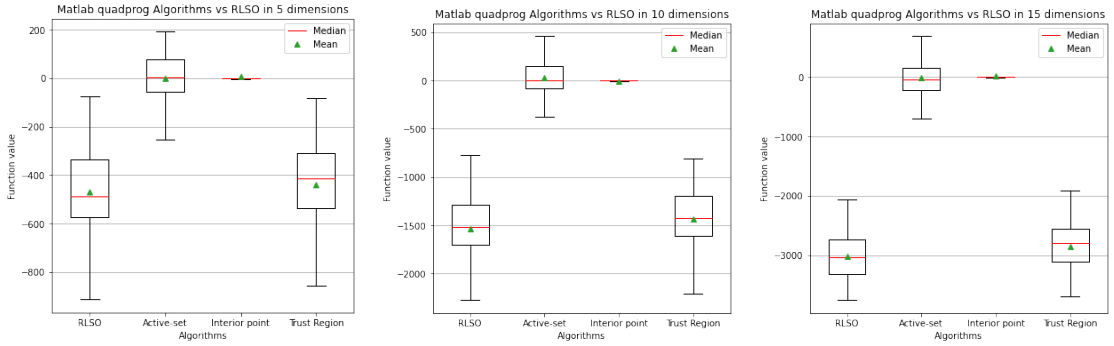
Dimension	Score - Adam	Score - GD	Score - RS
5, 10, 15, 25 mixed	-372.68 σ = 513.86	-722.92 σ = 1584.12	-2121.98 σ = 1879.46
5	-59.28 σ = 74.55	-117.52 σ = 246.22	-251.07 σ = 113.97
7	-114.76 σ = 125.53	-207.53 σ = 401.56	-519.72 σ = 180.59
10	-207.79 σ = 206.83	-365.44 σ = 664.59	-1034.56 σ = 269.8
12	-281.46 σ = 269.07	-509.45 σ = 885.7	-1448.58 σ = 336.55
15	-402.2 σ = 366.73	-712.18 σ = 1217.57	-2149.52 σ = 436.76
18	-523.15 σ = 472.27	-958.36 σ = 1566.09	-2933.61 σ = 539.75
20	-608.57 σ = 542.7	-1137.22 σ = 1853.89	-3495.36 σ = 605.23
25	-795.56 σ = 755.43	-1615.45 σ = 2524.61	-5020.0 σ = 811.08
30	-973.88 σ = 990.18	-2311.18 σ = 3394.68	-6688.25 σ = 1056.68
40	-1237.38 σ = 1567.67	-3430.15 σ = 4992.78	-10449.07 σ = 1641.02
50	-1314.71 σ = 2259.39	-4908.63 σ = 6734.34	-14483.15 σ = 2368.15
60	-1153.02 σ = 3116.67	-6390.82 σ = 8732.64	-18794.29 σ = 3203.42
70	-744.7 σ = 4123.61	-8022.7 σ = 10767.18	-23248.71 σ = 4238.39
80	18.68 σ = 5172.69	-9680.52 σ = 12920.85	-27793.53 σ = 5307.09
90	1083.3 σ = 6262.26	-11466.37 σ = 15054.57	-32435.11 σ = 6423.54
100	2599.92 σ = 7358.91	-13065.83 σ = 17162.49	-36902.39 σ = 7567.48

4.6. Comparison with QP solvers in MATLAB

We compare our RLSO with the standard QP solvers in MATLAB. We use three solvers: active set, interior point, and a trust-region reflective (TRR). Both the active set and interior point solvers are for convex QP problems and thus perform exceptionally poorly

on our problem distribution because of the relative infrequency of convex problems. The TRR algorithm is a generic algorithm for optimizing a function by iteratively minimizing proxy models of the underlying function. Building a proxy model is costly in terms of many function evaluations and scales quadratically with problem dimensions.

We used 100 newly generated test problems to reduce compute time, and we show the box plot of the final value after 100 iterations. Because nonconvex quadratic problems increase in frequency with dimension in our problem distribution, and because the TRR proxy model building step requires over 100 function samples for a single iteration, we stick to low-dimensional problems of size 5, 10, and 15. We see in Figure 5 that the interior point algorithm fails due to our problem distribution not containing enough convex problems with the necessary numerical robustness. Active set also fails after not being able to find a convex subproblem to address. We also see that RLSO is slightly better on average and distributionally better than TRR for all dimensions tested.



(a) (b) (c)

Figure 5: Box plots for the final solution found by RLSO, and MATLAB's active set, interior point, and trust-region algorithms.

5. Related Work

Prior work on learned optimization can be seen in various research going back at least 30 years but may not always be conceptually framed as optimization problems. Learning update rules for artificial neural networks (ANNs), which is implicitly learning optimization algorithms, was explored in (Bengio et al., 1990, 1995; Chalmers, 1991), but these approaches focused on the biological analogy of ANNs. A recurring theme in both the older literature above and in newer results is using genetic algorithms Metz et al. (2020) to search for optimizers. A majority of the recent work focuses on the use of recurrent neural networks (RNNs) to parameterize a stepwise update rule (Andrychowicz et al., 2016; Chen et al., 2017). RNNS have some natural advantages and include the possibility of the learned optimizer being able to generalize across both input and output dimensions. Modern architectures are essentially chaining the output of per-tensor inputs such as the gradient, or entry-wise mean of the gradient processed by an RNN into another RNN that produces

the update step Metz et al. (2020). The overall result is that optimizing the optimizer is a challenging task (Metz et al., 2019; Chen et al., 2020).

In contrast, our method naturally scales across dimensions without requiring an input RNN to learn low-dimensional representations for arbitrarily sized inputs, and our optimization problem is reducible to a standard reinforcement learning problem. In many respects, our work is a cross between Li and Malik (2016) in how we deal with the reinforcement learning task and Awad et al. (2020) work in Squirrel, a switching-based hyperparameter optimizer. Our work is the first learning to optimze approach that focuses on a optimization of QP problem instances.

Prior work on QP sought to assess the difficulty of solving the problem globally and to determine which algorithms, in particular, are best. QP naturally bifurcates into two problem cases: the convex case and the nonconvex case. The convex case occurs when all eigenvalues of the defining matrix Q are non-negative, in which case we call Q positive semidefinite. The nonconvex case can be further reduced into two cases. In one case, Q has both negative and positive eigenvalues, and it is called indefinite. In another case, when Q has all nonpositive eigenvalues, we call it negative semidefinite. The convex case is well behaved and can be solved in polynomial time. The nonconvex case is in general $\mathcal{NP}$-hard; and to determine what causes the difficulty in finding global solutions, researchers have looked at subproblems that retain the $\mathcal{NP}$-hardness property. In the general indefinite case, Murty and Kabadi (1987) showed that verifying that a feasible solution is a local minimizer is $\mathcal{NP}$-hard. Vavasis (2009) reviewed the complex issues of QP and a collection of restrictions that are still $\mathcal{NP}$-hard. In particular, QP with box constraints or simplicial constraints is still $\mathcal{NP}$-hard. Pardalos and Vavasis (1991) showed that in the indefinite case, even possessing one negative eigenvalue is enough for the problem to be $\mathcal{NP}$-hard. A result that gives more intuition about the nature of the problem can be found in (Pardalos, 1991). It provides a simple construction of an indefinite QP such that the number of local minima is 2^m, where m is the number of negative eigenvalues. This example uses only box constraints as well. Thus, while we have not explicitly shown that our problem distribution is in the $\mathcal{NP}$-hard set, we believe it is likely the case.

6. Conclusion

Our experiments show that combining solvers using a learned heuristic can outperform the individual solvers and result in a variant that outperforms the sum of its parts. Given that our chosen problem is an $\mathcal{NP}$-hard global optimization problem, it should be possible to generalize this work to optimizing neural networks or other stochastic problem settings. Further work could address the issues with restart policy and replace it with a more intelligent strategy. Two natural choices for different domains are to learn a biased restarting step as its own subproblem for the learned agent or in the quadratic programming case to learn a restart step, that is, a biased selector of an active step method.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-

AC02-06CH11357. We gratefully acknowledge the computing resources provided by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016.

Noor Awad, Gresa Shala, Difan Deng, Neeratyoy Mallik, Matthias Feurer, Katharina Eggensperger, Andre' Biedenkapp, Diederick Vermetten, Hao Wang, Carola Doerr, et al. Squirrel: A switching hyperparameter optimizer. *arXiv preprint arXiv:2012.08180*, 2020.

Thomas D Barrett, William R Clements, Jakob N Foerster, and Alex I Lvovsky. Exploratory combinatorial optimization with reinforcement learning. *arXiv:1909.04063*, 2019.

Samy Bengio, Yoshua Bengio, and Jocelyn Cloutier. On the search for new learning rules for ANNs. *Neural Processing Letters*, 2(4):26–30, 1995.

Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp, et al. A racing algorithm for configuring metaheuristics. In *Gecco*, volume 2, 2002.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

David J Chalmers. The evolution of learning: An experiment in genetic connectionism. In *Connectionist Models*, pages 81–90. Elsevier, 1991.

Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training stronger baselines for learning to optimize. *Advances in Neural Information Processing Systems*, 33, 2020.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.

Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando Freitas. Learning to learn without gradient descent by gradient descent. In *International Conference on Machine Learning*, pages 748–756. PMLR, 2017.

Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*. SIAM, 2009.

David S Dean and Satya N Majumdar. Extreme value statistics of eigenvalues of gaussian random matrices. *Physical Review E*, 77(4):041108, 2008.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

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

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

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

Luke Metz, Niru Maheswaranathan, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*, 2020.

K.G. Murty and S.N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical programming*, 39(2):117–129, 1987.

Panos M Pardalos. Global optimization algorithms for linearly constrained indefinite quadratic problems. *Computers & Mathematics with Applications*, 21(6-7):87–97, 1991.

Panos M Pardalos and Stephen A Vavasis. Quadratic programming with one negative eigenvalue is NP-hard. *Journal of Global optimization*, 1(1):15–22, 1991.

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

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

Terence Tao. *Topics in random matrix theory*, volume 132. American Mathematical Soc., 2012.

Stephen A Vavasis. Complexity theory: Quadratic programming., 2009.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan