A Planning-Based Reinforcement Learning Approach to Numerical Optimization

Uma Maheswari Natarajan[1,†] Sai Shruti Prakhya[1,†] Shivani Sanjiv Shukla[1,†]
Chandramouli Kamanchi[2] Raghuram Bharadwaj Diddigi[1]

Abstract

Optimization lies at the core of scientific computing and machine learning, where algorithms iteratively update parameters using function values and gradient information. In this work, we investigate whether effective optimization strategies can be learned from fixed datasets of such trajectories and transferred to new problem instances without further learning. We formulate numerical optimization as a Markov Decision Process (MDP), where each state consists of the current iterate, function value, and gradient information, and actions correspond to update directions and step sizes. We then adapt the popular planning-based Reinforcement Learning (RL) paradigm, AlphaGo, to numerical optimization to train policy and value networks offline using a family of Rosenbrock functions. At deployment, the learned policy and value functions are kept fixed and used within a Monte Carlo Tree Search (MCTS) procedure to perform trajectory-level planning. This enables the method to combine offline-learned optimization priors with test-time lookahead, allowing decisions that account for long-term effects of updates rather than relying solely on local heuristics. We evaluate the approach on a diverse suite of optimization problems beyond the training distribution. Empirical results show that the learned policy, when coupled with planning, generalizes effectively and achieves reliable convergence with competitive or improved optimization efficiency. Our results demonstrate that AlphaGo-style learning, when applied to optimization provides a promising framework for learning adaptive strategies from offline data.

1 Introduction

Optimization concerns the problem of identifying the best possible solution from a set of feasible alternatives and forms a foundational component of scientific computing and machine learning [6, 10]. To address the diversity and computational demands of real-world optimization problems, a wide range of optimization methods have been developed in the literature, spanning classical gradient-based techniques [2], Quasi-Newton methods, and their variants [22]. These approaches have demonstrated strong performance in well-behaved settings, particularly for smooth and convex objectives [7]. However, their effectiveness often degrades on more challenging landscapes, including those characterized by non-convexity, noise, ill-conditioning, flat regions, or the presence of multiple local minima [13, 29, 32]. In such scenarios, optimization procedures based on locally informed update rules can exhibit slow progress [23] or become sensitive to hyperparameter choices [33]. Motivated by these limitations, we investigate whether an artificial intelligent agent can be trained to solve numerical optimization[1] problems in a fundamentally different manner, with the goal of achieving reliable convergence while reducing the overall time to solution.

[†]Equal contribution [1]International Institute of Information Technology Bangalore [2]IBM Research, Bangalore, India
Correspondence: umamaheswari.natarajan@iiitb.ac.in

Dr. Raghuram Bharadwaj is supported by the Anusandhan National Research Foundation (ANRF) under the Prime Minister Early Career Research Grant ANRF/ECRG/2024/005235/ENS

[1]Throughout this paper, the term "optimization" refers to the problem of minimizing a real-valued objective function.

Accepted at ICML 2026 Workshop on Decision-Making from Offline Datasets to Online Adaptation: Black-Box Optimization to Reinforcement Learning.

Reinforcement learning (RL) [31] offers this fundamentally different paradigm for developing sequential decision-making strategies, emphasizing learning from experience rather than relying on explicit rules. In this framework, an agent interacts with an environment, receives feedback through rewards, and progressively refines its behavior to optimize long-term outcomes. Such learned decision strategies have been shown to rival, and in some cases surpass, carefully hand-designed algorithms across a range of complex domains. In particular, planning-based reinforcement learning systems such as AlphaGo Zero [28] have demonstrated the effectiveness of combining learned value functions with explicit look-ahead to master complex board games through long-horizon reasoning. More recently, similar ideas have been applied beyond games to algorithm discovery, where learning-driven search has uncovered improved routines for fundamental computational tasks such as sorting [18] and matrix multiplication [12]. Together, these successes highlight the ability of reinforcement learning to explore rich spaces of strategies and exploit long-term feedback, establishing it as a powerful and general framework for sequential decision making.

Motivated by the reinforcement learning framework for learning effective decision-making strategies, we explore its applicability to numerical optimization. Iterative optimization solutions can be naturally viewed as a sequential decision-making process, in which each iteration corresponds to a decision whose consequences extend beyond immediate objective reduction and influence the trajectory of subsequent updates. Specifically, each step involves two coupled choices: selecting a direction of movement and determining a step size that controls the magnitude of the update, with the shared objective of achieving fast and reliable convergence. This perspective motivates us to investigate the following questions:

- Can a reinforcement learning agent learn to select update directions and step sizes directly from feedback?
- If so, can such learned decisions lead to improved optimization behavior in practice, particularly in terms of convergence speed?
- Can a learned optimization strategy generalize across objective functions with differing characteristics, such as non-convexity, noise, or ill-conditioning?

To address these questions, we first model optimization as a sequential decision-making problem. Under this formulation, the problem of designing an optimizer is recast as the problem of learning a policy that maps observed optimization progress to update decisions. Building on this MDP formulation, we propose AlphaGrad, a planning-based reinforcement learning agent that employs Deep Monte Carlo Tree Search (MCTS) [28]. AlphaGrad explores possible future update sequences through simulation and selects actions that appear promising under this lookahead over resulting trajectories. By planning over entire optimization trajectories rather than making purely local decisions, AlphaGrad explicitly accounts for the long-term consequences of its actions. This enables the policy to favor updates that may be locally suboptimal; however can lead to faster overall progress under a fixed evaluation budget.

We first train AlphaGrad on an offline family of Rosenbrock functions and then evaluate on a suite of challenging benchmark functions. Across these problems, AlphaGrad consistently drives the iterates toward correct solutions and often matches or improves upon the convergence speed of classical optimization methods under the same evaluation budget. The learned policy further demonstrates robustness to variations in objective landscapes, including stochasticity, and strong non-convexity, indicating that planning-based reinforcement learning can acquire optimization strategies that generalize across problem instances rather than encoding a single local heuristic.

We note that we are not the first to view numerical optimisation through the lens of RL. Prior work has explored casting numerical optimization as an RL problem [9, 17, 5], notably [17], which learns reactive update rules mapping gradient histories to optimization steps. In contrast, our approach adopts a planning-based perspective, using MCTS to reason over future optimization trajectories. This allows the learned policy and value functions to guide long-horizon decision making.

2 Methodology

In this section, we begin by formulating the problem of finding the global minimum of an objective function as a Markov Decision Process (MDP) [4], which we then solve using a Deep Monte Carlo Tree Search (Deep MCTS) [28] algorithm. Deep MCTS combines Monte Carlo Tree Search [8]

with a neural network that is trained to guide the planning process by providing policy and value estimates, thereby improving action selection over long planning horizons. Section 2.1 describes the MDP formulation, while Section 2.2 briefly describes the proposed algorithm and the training strategy employed.

2.1 MDP Formulation

We define the optimization process as an MDP by specifying the state space, action space, and reward function. The state space is a five-dimensional vector[2] consisting of the two variables x and y, the function value $f(x, y)$ and the partial derivatives f_x and f_y. This state representation ensures a fair comparison with traditional first-order based gradient methods, that operate with the same information [22]. The partial derivatives provide local first-order slope information, while x, y and $f(x, y)$ serve as absolute references. Integrating these components into the state representation at time step t, denoted by s_t, allows the agent to construct an approximate mapping of the underlying functional landscape:

$$s_t = [x_t, y_t, f(x_t, y_t), f_x(x_t, y_t), f_y(x_t, y_t)]. \tag{1}$$

The resulting state space is continuous, with x and y being initialized randomly at the beginning of each training episode. During an episode, the agent navigates the space within a predefined boundary for both variables.

The action space is represented as a tuple (a_x, a_y), where each component represents the action chosen for a particular dimension and is expressed as the product of a step size (α) and a direction (d). This independence of actions for each axis allows the agent to execute separate updates, which is crucial for navigating landscapes where curvature varies significantly across dimensions (e.g., narrow valleys).

The step sizes are chosen from a predefined set of values, while the direction is restricted to a binary choice indicating the sign of the update. This formulation explicitly decouples the step magnitude from the gradient's norm, preventing the vanishing gradient typically associated with flat regions of the objective landscape [25, 20]. Consequently, the agent gains a multi-resolution search capability: it can independently select large steps to traverse flat regions and then switch to smaller steps for precise convergence, distinct from the local gradient magnitude.

Once an action is taken, the state transition takes place as given below:

$$x_{t+1} = x_t + a_x,$$

$$y_{t+1} = y_t + a_y,$$

$$s_{t+1} = [x_{t+1}, y_{t+1}, f(x_{t+1}, y_{t+1}), f_x(x_{t+1}, y_{t+1}), f_y(x_{t+1}, y_{t+1})]. \tag{2}$$

While the above transition describes a deterministic update, the formulation can naturally extend to stochastic settings. In the presence of noise or uncertainty in function evaluations or gradient estimates, the state transition induces a probability distribution over next states rather than a single deterministic outcome. This is consistent with the general MDP framework, where transitions are defined by a conditional probability distribution $P(s_{t+1} \mid s_t, a_t)$.

Based on the current state, the action taken by the agent, and the resulting state transition, an immediate reward is provided as a feedback to the agent. We constuct the reward in a way that accounts for both correctness and latency. It includes a per-step penalty that encourages the agent to reach the goal efficiently, along with a term proportional to the change in function value between consecutive states, defined as $\Delta f = f(x_t, y_t) - f(x_{t+1}, y_{t+1})$. The second component of the intermediate reward encourages a decrease in the function value between the current and next states. Thus, the intermediate reward encourages monotonic reduction in the function value across successive iterations.

[2]For simplicity of exposition, we define the MDP for a two-variable optimization problem; the formulation naturally extends to higher dimensions.

There are three terminal scenarios in our formulation. The first occurs when the agent exhausts the maximum number of steps allowed for reaching the optimum. The second occurs when the agent violates the predefined boundary constraints, resulting in a large negative terminal reward and termination of the episode. This penalty reflects the fact that divergence in any single variable leads to failure of the optimization process, and thus teaches the agent to avoid unstable trajectories. The final scenario corresponds to the agent reaching a point within an ϵ-neighborhood of the global minimiser (x^*, y^*), upon which a large positive terminal reward is assigned and the episode terminates. Thus, the reward function is mathematically formulated as:

$$r(s_t, a) = \begin{cases} R_g & \text{if } |x_{t+1} - x^*| \leq \epsilon,\ |y_{t+1} - y^*| \leq \epsilon \\ R_p & \text{if } \max(|x_{t+1}|, |y_{t+1}|) > B \\ R_s + c \cdot \Delta f & \text{otherwise} \end{cases}, \tag{3}$$

where R_g denotes the large positive reward obtained upon reaching the ϵ boundary of the global minimiser (x^*, y^*), R_p represents the penalty incurred by the agent upon violating the boundaries of the search space (B), R_s denotes the per-step penalty that encourages faster convergence, Δf represents the reduction in the function value between two consecutive states ($f(x_t, y_t) - f(x_{t+1}, y_{t+1})$) and c denotes a scaling hyper-parameter.

Note that Eq. 3 requires the knowledge of optima. During offline training, knowledge of the optimal solution (x^*, y^*) is used to define terminal rewards, providing a strong learning signal. During the inference, however, the MCTS tree is guided exclusively by the intermediate reward, i.e, the third term in Eq. 3 and does not make use of the other two terms (does not assume the knowledge of (x^*, y^*)).

2.2 Proposed Algorithm

Having formulated the optimization problem as an MDP, we introduce AlphaGrad, an agent designed to find the global minimiser of an objective function, inspired by the superhuman performance of AlphaGo Zero [28]. AlphaGrad combines a deep neural network with Monte Carlo Tree Search (MCTS) [8] to guide action selection. The neural network outputs policy distributions over actions along each dimension and a value estimate, which serve as priors for the MCTS search. Using the PUCT formulation, the search explores future trajectories and refines actions based on simulated outcomes, enabling reasoning over long-term optimization behavior. The refined policy is used to sample actions and generate trajectories, which are used to train the network via a combined value and policy loss under a temporal difference framework. The network parameters are optimized iteratively to improve both policy and value estimates. The complete proposed methodology, including the Deep MCTS algorithm and training procedure, is presented in the Appendix A section.

3 Experiments

We benchmark our proposed AlphaGrad algorithm against established optimization algorithms, including Barzilai–Borwein (variants 1 and 2) [2], Steepest Descent [7], Quasi-Newton [22], and Adam [16]. Section 3.1 outlines the experimental setup used for training and evaluation and in Section 3.2, we analyze performance on deterministic benchmark functions to address the research questions posed in the Introduction Section 1.

3.1 Experimental Setup

Training: We train AlphaGrad in a custom optimization environment implemented using the OpenAI Gym framework, based on a subset of the two-variable Rosenbrock family of functions [26], parametrised by constants a and b that determine the location and curvature of the global minimum, defined as $f(x, y) = (a - x)^2 + b(y - x^2)^2$. The Rosenbrock function, characterized by an ill-conditioned Hessian and a long, narrow parabolic valley, is a standard benchmark for optimization, where gradient-based methods often exhibit slow convergence. We evaluate the learned policy on four challenging benchmark functions [30]: Beale [3], Ackley [1], Bukin N.6 [15], and Three-Hump Camel [15] functions and an overview of these evaluation functions is provided in Table 1.

The agent is trained for 3000 episodes, with a maximum of 200 steps per episode. In each episode, a function is randomly sampled from the Rosenbrock family [26], with parameters $a \in [-1.5, 1.5]$ and $b \in [0.1, 1.5]$, yielding a global minimiser at (a, a^2) and promoting generalization across objectives. At each step, actions are selected from 20 equally spaced step sizes in $[0.001, 0.999]$ along each coordinate, combined with a binary update direction. Episodes terminate upon reaching the target region, violating state-space bounds, or exhausting 200 steps. The state variables x and y are initialized uniformly in $[-15, 15]$ and constrained within $[-20, 20]$, with violations incurring a penalty of -100. A terminal reward of $+100$ is assigned upon reaching the ε-neighborhood ($\varepsilon = 0.2$) of the global minimiser (x^*, y^*), while non-terminal transitions receive an intermediate reward of $-1 + \alpha\Delta f$, and $\alpha = 0.3$ is a scaling hyperparameter selected empirically. To balance exploration and exploitation, we use 20 MCTS simulations per step for the first 2000 episodes and 50 for the final 1000. Additionally, Dirichlet noise ($\alpha = 0.3$, $\epsilon = 0.25$) is incorporated at the root of the MCTS tree [28] to encourage exploration and mitigate early policy bias.

Evaluation: To evaluate the generalization capability of AlphaGrad, we construct custom OpenAI Gym environments for four benchmark optimization functions and their stochastic variants. Table 1 summarizes their functional forms, evaluation intervals, and global minimisers. During evaluation, the neural network weights are frozen, and adaptation occurs solely through the MCTS look-ahead mechanism using only intermediate rewards and a per-step penalty, preventing any leakage of the true optimum (x^*, y^*) or search boundaries. The agent operates within the same action space as in training, selecting from 20 discrete step sizes and binary directions. The benchmark functions exhibit diverse challenges, including multiple local minima, ill-conditioning, non-differentiability, and sharp landscapes. AlphaGrad performs 50 MCTS simulations per step with a maximum of 200 steps, and this corresponds to a total of 10,000 function evaluations per episode. Accordingly, to ensure fair comparison, the number of steps per evaluation episode is fixed to 10,000 for Barzilai–Borwein variants 1 and 2 [2], Quasi-Newton [22], and Adam [16]. For Steepest Descent [7], which uses variable evaluations due to line search, the step count is adjusted per function to match the same evaluation budget.

For consistent comparison, step counts for traditional methods are scaled to the Deep MCTS time scale. The success rate is defined as the percentage of 1000 evaluation episodes in which the agent reaches the ε-boundary ($\varepsilon = 0.2$) of (x^*, y^*). Failure occurs upon exhausting the step limit or breaching the search boundary, defined as $[-20, 20]$ for all functions except Ackley [1], where the range is $[-35, 35]$.

Hardware: All experiments, including both training and evaluation, were conducted using the standard CPU runtime provided by Google Colab.

Function Name	**Expression**	**Evaluation Interval**	**Global Minimum**
Beale	$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$	$x, y \in [-4.5, 4.5]$	0 at $(x, y) = (3, 0.5)$
Ackley	$f(x,y) = -20\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right) - \exp\left(0.5[\cos(2\pi x) + \cos(2\pi y)]\right) + 20 + e$	$x, y \in [-32.768, 32.768]$	0 at $(x, y) = (0, 0)$
Bukin N.6	$f(x,y) = 100\sqrt{\lvert y - 0.01x^2\rvert} + 0.01\lvert x + 10\rvert$	$x \in [-15, -5]$ and $y \in [-3, 3]$	0 at $(x, y) = (-10, 1)$
3 Hump Camel	$f(x,y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2$	$x, y \in [-5, 5]$	0 at $(x, y) = (0, 0)$

Table 1: Benchmark functions used for evaluation.

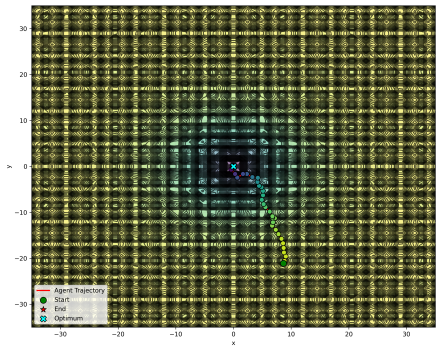
3.2 Results and Discussion

We now analyze the empirical results obtained from the evaluation procedure described in Section 3.1.

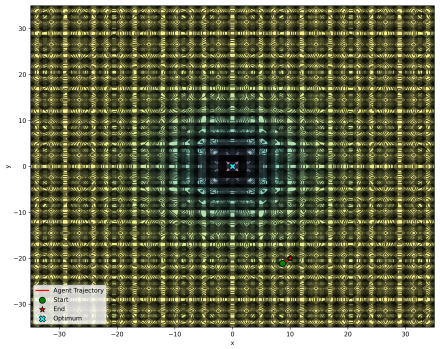
3.2.1 Ackley Function without Stochasticity

The Ackley function [1] in two-dimensional space consists of an outer region characterized by shallow sinusoidal variations and a deep, narrow hole at the center containing the global minimum. Despite this challenging landscape, AlphaGrad achieves a 97.7% success rate with an average of 62.09 ± 47.07 steps per episode in navigating these innumerable local optima to locate global minimiser

(Table 2). A key observation is the smoothness of the agent's trajectory (Figure 1a), which cuts directly past thousands of local optima regions. This indicates that the agent's actions are not myopic, but instead reflect long-horizon planning enabled by the MCTS look-ahead mechanism. In contrast, none of the traditional methods (Figure 1b) successfully cross the outer region to reach the center. The flat, sinusoidal nature of the landscape causes these methods to stagnate in local optima, resulting in success rates below 5% across all baseline optimizers considered.



(a) AlphaGrad (ours): Successful optimization.
Start: $(8.65,\ -21.10)$, Final: $(-0.08,\ 0.14)$
Optimum: $(0.00,\ 0.00)$, Distance: $(0.08,\ 0.14)$
Steps: 30, Success: **True**



(b) Steepest Descent: Failed optimization.
Start: $(8.65,\ -21.10)$, Final: $(10.00,\ -20.00)$
Optimum: $(0.00,\ 0.00)$, Distance: $(10.00,\ 20.00)$
Steps: 1254, Success: **False**

Figure 1: Optimization results on the Ackley function [1] starting from the same initial point. AlphaGrad reaches the global minimum in only 30 steps, whereas Steepest Descent [7] fails to do so.

Table 2: Ackley Function Without Stochasticity

Optimization Algorithm	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	**97.7**	$\mathbf{62.09 \pm 47.07}$	$\mathbf{58.84 \pm 42.54}$
BB2	0.2	199.60 ± 8.92	0.35 ± 0.07
BB1	0.0	200.00 ± 0.00	0.00 ± 0.00
Steepest Descent (Armijo Conditions)	1.3	197.55 ± 22.57	0.86 ± 0.32
Quasi Newton	0.8	198.40 ± 17.79	0.26 ± 0.19
Adam	0.0	0.00 ± 0.00	0.00 ± 0.00

3.2.2 Beale Function without Stochasticity

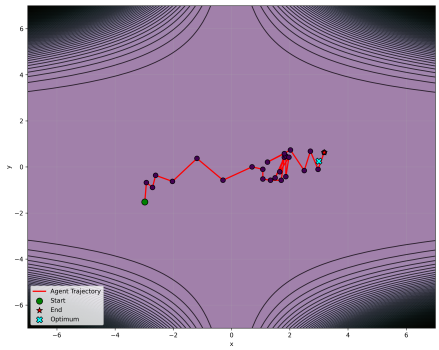
The performance of AlphaGrad on the Beale function [3], as shown in Table 3, highlights its ability to locate the global minimiser despite the presence of local minima, sharp spikes, and extended flat valleys, conditions under which traditional optimization methods often struggle. AlphaGrad achieves a 96% success rate, while the second-best performing method, Steepest Descent [7], lags significantly behind with a success rate of only 55.6%. As illustrated in Figure 2a, AlphaGrad adopts a distinct zig-zag trajectory, most pronounced at the beginning and near the end of the search. Even within the flat region, the agent actively explores different directions. This strategy is effective in such regions, where many actions yield similar rewards, allowing the agent to broadly sample the state space while still making steady progress towards the target, and ultimately converging to the optimum.

In contrast, the Steepest Descent [7] algorithm shown in Figure 2b initially takes a large step but subsequently stagnates in the function's flat region. This limitation arises from the fundamental update equation governing traditional optimizers:

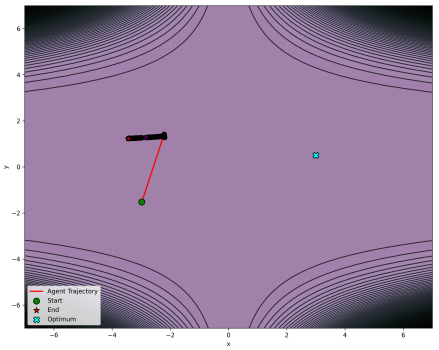
$$x_{\text{new}} = x_{\text{old}} - \alpha\, f'(x_{\text{old}}).$$

Here, the gradient vector $f'(x_{\text{old}})$ determines the descent direction, while α controls the step size or learning rate. Traditional methods primarily differ in how α is computed: Quasi-Newton methods [22] approximate curvature information via the inverse Hessian, Barzilai–Borwein methods [2] derive

α from differences between successive iterates, Adam [16] incorporates gradient moment estimates, and Steepest Descent [7] selects α using a line search. However, as these optimizers approach the optimum, the gradient magnitude $f'(x)$ tends toward zero, leading to increasingly conservative updates. A key novelty of our approach lies in decoupling the gradient magnitude from the update step by incorporating gradient information directly into the state representation. In AlphaGrad, this decoupling allows the update direction to be chosen discretely from $\{-1, +1\}$ along each axis, enabling the agent to maintain meaningful step sizes even near the minimum.



(a) AlphaGrad (ours): Successful optimization.
Start: $(-2.98,\ -1.52)$, Final: $(3.18,\ 0.63)$
Optimum: $(3.00,\ 0.50)$, Distance: $(0.18,\ 0.13)$
Steps: 24, Success: **True**



(b) Steepest Descent: Failed optimization.
Start: $(-2.98,\ -1.52)$, Final: $(-3.44,\ 1.23)$
Optimum: $(3,\ 0.5)$, Distance: $(5.43,\ 0.81)$
Steps: 1453, Success: **False**

Figure 2: Optimization trajectories on the Beale function [3] starting from the same initial point. AlphaGrad converges successfully to the optimum in 24 steps, whereas Steepest Descent [7] fails to do so.

Table 3: Beale Function Without Stochasticity

Optimization Algorithm	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	**96.0**	$\mathbf{28.11 \pm 35.60}$	$\mathbf{21.57 \pm 15.03}$
BB2	24.3	151.49 ± 85.61	0.38 ± 0.16
BB1	25.3	150.88 ± 86.82	0.30 ± 0.12
Steepest Descent (Armijo Conditions)	55.6	92.98 ± 96.60	7.45 ± 17.50
Quasi Newton	1.3	197.40 ± 22.63	0.26 ± 0.16
Adam	43.1	143.53 ± 74.69	68.99 ± 58.19

3.2.3 3-Hump Camel Function without Stochasticity

AlphaGrad demonstrates robust performance on the 3-Hump Camel function [15], a multimodal landscape characterized by three local minima. As we observe in Table 4, the agent achieves a 100% success rate, converging in an average of 7.84 ± 3.79 steps per episode. In contrast, the next best-performing method, Steepest Descent [7], lags significantly behind, attaining only a 60.5% success rate and requiring 80.24 ± 96.96 steps on average. This substantial disparity highlights AlphaGrad's superior convergence efficiency in a setting where the presence of an xy interaction term complicates independent variable optimization (Table 4). Notably, since AlphaGrad is trained exclusively on a unimodal objective, this result underscores the agent's ability to generalize effectively to previously unseen multimodal landscapes with different structural characteristics.

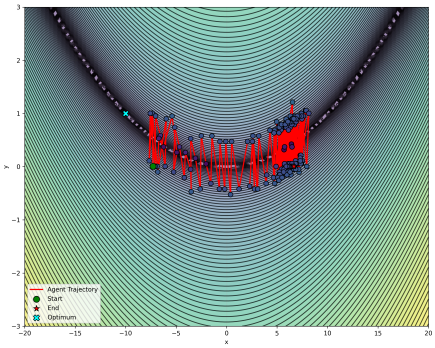
3.2.4 Bukin N.6 Function without Stochasticity

The Bukin N.6 function [15] is the only non-differentiable landscape among the evaluated benchmarks. It features a sharp, valley-shaped ridge containing multiple local optima, with the global minimum lying directly along this non-differentiable seam. As shown in Table 5, this structure proves challenging for all optimization methods. Nevertheless, AlphaGrad achieves a success rate of 46.3%, outperforming most classical optimizers, with Barzilai–Borwein 2 [2] being the closest competitor

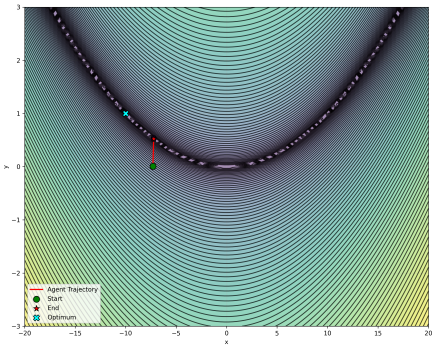
Table 4: 3 Hump Camel Function Without Stochasticity

Optimization Algorithm	Success Rate (%)	Avg Steps	Avg Steps (Success)
AlphaGrad (ours)	**100**	$\mathbf{7.84 \pm 3.79}$	$\mathbf{7.84 \pm 3.79}$
Barzilai Borwein 2 (BB2)	40.7	118.65 ± 98.19	0.13 ± 0.06
Barzilai Borwein 1 (BB1)	33.7	132.65 ± 94.48	0.12 ± 0.07
Steepest Descent (Armijo Conditions)	60.5	80.24 ± 96.96	1.92 ± 0.64
Quasi Newton	19.5	161.03 ± 79.19	0.13 ± 0.07
Adam	19.8	162.03 ± 76.46	8.22 ± 5.73

at 43.2%. As illustrated in Figure 3a, AlphaGrad is able to navigate local minima and points of discontinuity, in contrast to the Steepest Descent method [7] shown in Figure 3b. Notably, the agent's trajectory is consistently non-monotonic, reflecting substantial exploration of the search space. While AlphaGrad often identifies the correct descent direction early in the episode, the absence of smooth gradient feedback limits its ability to transition effectively from exploration to exploitation. As a result, the agent frequently changes course and terminates away from the global minimum within the evaluation budget.



(a) AlphaGrad (ours): Failed optimization.
Start: $(-7.30,\ 0.01)$, Final: $(6.66,\ 0.01)$
Optimum: $(-10.00,\ 1.00)$, Distance: $(16.66,\ 0.99)$
Steps: 200, Success: **False**



(b) Steepest Descent: Failed optimization.
Start: $(-7.30,\ 0.01)$, Final: $(-7.22,\ 0.52)$
Optimum: $(-10.00,\ 1.00)$, Distance: $(2.78,\ 0.48)$
Steps: 133, Success: **False**

Figure 3: Optimization results on the Bukin N.6 function [15] from a common initial point. Both AlphaGrad and Steepest Descent [7] fail to converge. AlphaGrad deviates substantially from the optimum, while Steepest Descent remains closer but still unsuccessful.

Table 5: Bukin N.6 Function Without Stochasticity

Optimization Algorithm	Success Rate (%)	Avg Steps	Avg Steps (Success)
AlphaGrad (ours)	46.3	113.31 ± 93.61	12.77 ± 10.03
BB2	43.2	121.18 ± 90.83	17.55 ± 13.78
BB1	0.8	198.40 ± 17.81	0.03 ± 0.01
Steepest Descent (Armijo Conditions)	5.3	189.95 ± 44.21	3.07 ± 1.81
Quasi Newton	0.4	199.54 ± 7.68	84.50 ± 38.12
Adam	0.9	182.97 ± 55.03	10.80 ± 32.77

3.3 Discussion

Based on the empirical results in Section 3.2, we interpret AlphaGrad's performance through the lens of reinforcement learning and contrast it with classical optimization methods. Across diverse benchmark functions, AlphaGrad demonstrates strong performance, particularly on challenging

landscapes where traditional methods struggle. This behavior reflects a core reinforcement learning principle: prioritizing long-term gains over greedy, short-term improvements.

Classical optimization methods largely operate in a myopic manner. Steepest Descent [7] is inherently greedy, selecting updates that maximize immediate reduction in function value. Similarly, methods such as Barzilai–Borwein [2], Quasi-Newton [22], and Adam [16] incorporate historical gradient information to refine updates, but remain focused on improving the immediate step rather than explicitly reasoning over future trajectories. As a result, these approaches are fundamentally local, optimizing short-horizon objectives without planning over the long-term evolution of the optimization process.

In contrast, AlphaGrad selects actions through explicit look-ahead. At each step, it performs multiple MCTS simulations to explore alternative update sequences and evaluate their expected outcomes using the learned value function. These simulations yield a refined policy that guides action selection. This trajectory-level planning discourages locally optimal but globally suboptimal updates, while the stochastic policy promotes sustained exploration of the search space.

Overall, these results indicate that AlphaGrad learns a generalized optimization strategy that adapts across diverse landscapes. Importantly, this behavior emerges from offline training, demonstrating that AlphaGo-style learning can effectively acquire transferable optimization strategies from offline data. We also train and evaluate our model on stochastic variants of the benchmark functions, where stochasticity is introduced by adding Gaussian noise with zero mean and a standard deviation of 0.01 to each element of the input state vector independently. Detailed results for these variants are presented in the Appendix B section, and the implementation code for both deterministic and stochastic settings is available in the anonymous repository: `https://anonymous.4open.science/r/RL-Optimization-A8C7/`

4 Related Works

Traditional Methods: Early work in optimization focused on hand-crafted gradient-based methods [7]. Techniques such as Barzilai–Borwein introduced adaptive step sizes, while Nesterov's Accelerated Gradient [21] achieved faster convergence rates of $O(1/k^2)$. With the rise of deep learning, challenges such as saddle points and hyperparameter tuning became prominent, leading to optimizers such as Momentum [24], AdaGrad [11], RMSprop [27], and Adam [16], which incorporate gradient averaging and per-parameter adaptation. Despite these advances, such methods rely on fixed, hand-crafted update rules that may struggle to dynamically adapt to complex, non-convex landscapes.

Reinforcement Learning Methods: Reinforcement learning (RL) [31] offers an alternative paradigm by learning optimization strategies directly. Its success in large search spaces, demonstrated by Deep Q-Networks [19] and AlphaGo Zero [28], has inspired applications to algorithm discovery, including AlphaDev [18] and AlphaTensor [12]. Prior work has formulated optimization as an RL problem [9, 14, 17, 5], with approaches such as [17] learning reactive update rules from past information. However, these methods remain retrospective, lacking explicit reasoning over future trajectories. More recent approaches such as Prompted Policy Search (ProPS) [34] incorporate large language models but still require advances for effective deep RL representations. In contrast, our work adopts a planning-based approach, using MCTS-driven look-ahead to learn optimization strategies that account for long-term effects, bridging reactive and planning-based paradigms.

5 Conclusion

Optimization lies at the core of scientific computing and machine learning, yet classical methods often struggle on complex, non-convex landscapes. In this work, we formulated numerical optimization as a Markov Decision Process and proposed AlphaGrad, a planning-based reinforcement learning approach that combines learned policies with Monte Carlo Tree Search. Across diverse benchmark functions, AlphaGrad achieves reliable convergence under fixed evaluation budgets and often outperforms strong classical baselines, particularly on challenging landscapes. These results highlight the benefits of trajectory-level planning in prioritizing long-term progress over greedy updates. Overall, our findings demonstrate that planning-based reinforcement learning can learn transferable optimization strategies, and that AlphaGo-style learning provides a promising framework for developing adaptive optimizers from offline data.

References

[1] Ernesto P Adorio and U Diliman. Mvf-multivariate test functions library in c for unconstrained global optimization. *Quezon City, Metro Manila, Philippines*, 44, 2005.

[2] Jonathan Barzilai and Jonathan M Borwein. Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148, 1988.

[3] Evelyn Martin Lansdowne Beale. *On an iterative method for finding a local minimum of a function of more than one variable*. Number 25. Statistical Techniques Research Group, Section of Mathematical Statistics, Princeton University, Princeton, NJ, USA, 1958.

[4] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[5] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pages 459–468. PMLR, 2017.

[6] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

[7] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[8] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[9] Moésio Wenceslau da Silva Filho, Gabriel A Barbosa, and Péricles BC Miranda. Learning global optimization by deep reinforcement learning. In *Brazilian Conference on Intelligent Systems*, pages 417–433. Springer, 2022.

[10] Kalyanmoy Deb. *Optimization for engineering design: Algorithms and examples*. PHI Learning Pvt. Ltd., 2012.

[11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[12] Fawzi, Alhussein and Balog, Matej and Huang, Aja and Hubert, Thomas and Romera-Paredes, Bernardino and Barekatain, Mohammadamin and Novikov, Alexander and R. Ruiz, Francisco J and Schrittwieser, Julian and Swirszcz, Grzegorz and others. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[13] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[14] Eloghosa Ikponmwoba and Opeoluwa Owoyele. Deephive: A reinforcement learning approach for automated discovery of swarm-based optimization policies. *Algorithms*, 17(11):500, 2024.

[15] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013.

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

[17] Ke Li and Jitendra Malik. Learning to optimize, 2016. URL `https://arxiv.org/abs/1606.01885`.

[18] Mankowitz, Daniel J and Michi, Andrea and Zhernov, Anton and Gelmi, Marco and Selvi, Marco and Paduraru, Cosmin and Leurent, Edouard and Iqbal, Shariq and Lespiau, Jean-Baptiste and Ahern, Alex and others. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[19] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Rusu, Andrei A and Veness, Joel and Bellemare, Marc G and Graves, Alex and Riedmiller, Martin and Fidjeland, Andreas K and Ostrovski, Georg and others. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[20] Ryan Murray, Brian Swenson, and Soummya Kar. Revisiting normalized gradient descent: Fast evasion of saddle points. *IEEE Transactions on Automatic Control*, 64(11):4818–4824, 2019.

[21] Yurii Nesterov. A method for solving the convex programming problem with convergence rate o (1/k2). In *Dokl akad nauk Sssr*, volume 269, page 543, 1983.

[22] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 2006.

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

[24] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[25] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE international conference on neural networks*, pages 586–591. IEEE, 1993.

[26] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The computer journal*, 3(3):175–184, 1960.

[27] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[28] Silver, David and Schrittwieser, Julian and Simonyan, Karen and Antonoglou, Ioannis and Huang, Aja and Guez, Arthur and Hubert, Thomas and Baker, Lucas and Lai, Matthew and Bolton, Adrian and others. Mastering the game of go without human knowledge. *nature*, 550 (7676):354–359, 2017.

[29] James C Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons, 2005.

[30] Sonja Surjanovic and Derek Bingham. Virtual library of simulation experiments: Test functions and datasets, 2013. Available at: https://www.sfu.ca/~ssurjano/optimization.html.

[31] Sutton, Richard S and Barto, Andrew G and others. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[32] Aimo Törn and Antanas Žilinskas. *Global optimization*, volume 350. Springer, 1989.

[33] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 2002.

[34] Yifan Zhou, Sachin Grover, Mohamed El Mistiri, Kamalesh Kalirathnam, Pratyush Kerhalkar, Swaroop Mishra, Neelesh Kumar, Sanket Gaurav, Oya Aran, and Heni Ben Amor. Prompted policy search: Reinforcement learning through linguistic and numerical reasoning in llms. *arXiv preprint arXiv:2511.21928*, 2025.

Appendix

A Proposed Algorithm

We introduce AlphaGrad, an agent designed to find the global minimiser of an objective function, inspired by the superhuman performance of AlphaGo Zero [28] in games such as Go and Chess. The core idea of Monte Carlo Tree Search is to enable an agent to explicitly look ahead by simulating the future consequences of its actions and selecting those that provide the greatest long-term benefit. In the context of numerical optimization, this paradigm allows the agent to reason over the downstream effects of step-size and direction choices, evaluating entire optimization trajectories rather than isolated updates. As a result, the agent can favor decisions that may be locally suboptimal but lead to faster and more reliable global convergence.

AlphaGrad consists of two components: a deep neural network and a Monte Carlo Tree Search [8] that work in tandem to choose the best action for a given state (as shown in Fig. 4). The neural network takes the state vector s as input and outputs a policy over a_x, a policy over a_y, and a value estimate, V for the state. These policies π_x and π_y act as strong priors for the MCTS tree, which explores through multiple simulations the different possible actions that can be taken from s. The tree is expanded using the PUCT formula [28] in Equation 4, which accounts for both the state–action value $Q(s, a)$ and the number of times a node is visited. The search outputs a refined policy based on visit counts, which is then used by AlphaGrad to select an action in the real environment. Algorithms 1 and 2 describe the complete AlphaGrad algorithm and the MCTS procedure, respectively.

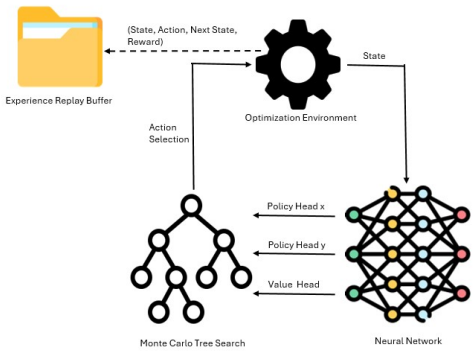


Figure 4: Deep MCTS Pipeline

To begin with, we parameterize the policy and value functions using a simple fully connected neural network with three output heads. One head outputs the policy for x, the second outputs the policy for y, and the third outputs the value of the state.

Algorithm 1 The AlphaGrad Algorithm

Input: Family of functions $f(x, y; a, b)$, learning rate α, discount factor γ, MCTS simulations M, batch size N

Initialization:
Initialize neural network f_θ with random weights θ
Initialize empty replay buffer $\mathcal{B}$

for episode $\leftarrow 1$ to MaxEpisodes **do**
    Sample function parameters a and b
    Initialize starting coordinates (x_0, y_0)
    Compute initial state $s_1 \leftarrow [x_0, y_0, f(x_0, y_0), f_x, f_y]$
    for $t \leftarrow 1$ to MaxSteps **do**
        $(\pi_x, \pi_y, V) \leftarrow f_\theta(s_t)$
        $(\Pi_x, \Pi_y) \leftarrow$ **Algorithm 2**(s_t, π_x, π_y, M)
        Sample action (a_x, a_y) from refined policy (Π_x, Π_y)

        Execute action: $x_{t+1} \leftarrow x_t + a_x$, $y_{t+1} \leftarrow y_t + a_y$
        Get reward $r(s_t, a)$ from Eq. (3) and termination flag $done_t$
        Compute next state s_{t+1}
        Store transition $(s_t, \Pi_x, \Pi_y, r_t, s_{t+1}, done_t)$ in replay buffer $\mathcal{B}$
        $s_t \leftarrow s_{t+1}$
        if $done_t$ **is** True **then**
            break
        end if
    end for

    if $|\mathcal{B}| \geq N$ **then**
        Sample minibatch $\{(s_i, \Pi_x, \Pi_y, r_i, s'_i, done_i)\}_{i=1}^{N}$ from $\mathcal{B}$

        $(\pi_x(s_i), \pi_y(s_i), V_{\text{pred}}(s_i)) \leftarrow f_\theta(s_i)$
        $V_{\text{target}}(s_i) \leftarrow r_i + \gamma(1 - done_i)V_{pred}(s'_i)$
        Use $V_{target}(s_i)$ and $V_{pred}(s_i)$ to compute value loss $\mathcal{L}_{\text{value}}$ through Eq. (5)
        Use $(\pi_x(s_i), \pi_y(s_i), \Pi_x, \Pi_y)$ to compute policy loss $\mathcal{L}_{\text{policy}}$ through Eq. (6)
        Compute total loss $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{policy}} + \mathcal{L}_{\text{value}}$

        Update network weights $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\text{total}}$
        Clear replay buffer $\mathcal{B}$
    end if
end for

Algorithm 2 MCTS Search Procedure

procedure MCTS($s_{root}, \pi_x, \pi_y, M$)
  Initialize tree T with root state s_{root}
  Initialize edge visit counts $N(s,a) \leftarrow 0$ and cumulative value sums $V(s,a) \leftarrow 0$ for all (s,a)
  Initialize temperature parameter τ
  for $simulation \leftarrow 1$ to M **do**
    $s \leftarrow s_{root}$
    path $\leftarrow [\,]$ ▷ Stores state–action pairs
    ▷ 1. Selection
    while s is fully expanded and non-terminal **do**
      Select action a with the highest PUCT score at state s using Eq. (4)
      Append (s,a) to path
      $s \leftarrow$ next state after applying a
    end while
    ▷ 2. Expansion
    if s is not fully expanded and non-terminal **then**
      Select an untried action a using priors π_x, π_y
      Expand the tree with edge (s,a) and next state s' as a new node
      Append (s,a) to path
      $s \leftarrow s'$
    end if
    ▷ 3. Evaluation
    if s is terminal (boundary violation) **then**
      $r \leftarrow R_p$
    else if s is terminal (goal reached) **then**
      $r \leftarrow R_g$
    else
      $r \leftarrow R_s + c.\Delta f$ (intermediate reward)
    end if
    ▷ 4. Backpropagation
    $v \leftarrow r$ ▷ Scalar return from simulation rollout
    for each (s,a) edge in reversed(path) **do**
      $N(s,a) \leftarrow N(s,a) + 1$
      $V(s,a) \leftarrow V(s,a) + v$ ▷ Same rollout value propagated to all edges
      $Q(s,a) \leftarrow \dfrac{V(s,a)}{N(s,a)}$ ▷ Monte Carlo action-value estimate
    end for
  end for

  Generate refined policies $\Pi_x(s_{root}, a_x)$, $\Pi_y(s_{root}, a_y)$ from root visit counts.
    For each action a_x: $\Pi_x(s_{root}, a_x) \propto N(s_{root}, a_x)^{1/\tau}$
    For each action a_y: $\Pi_y(s_{root}, a_y) \propto N(s_{root}, a_y)^{1/\tau}$
    Normalize Π_x and Π_y to form probability distributions
  return (Π_x, Π_y)
end procedure

We now describe the Monte Carlo Tree Search [8] procedure used by AlphaGrad. The tree search consists of four phases: selection, expansion, evaluation, and back propagation. In the selection phase, the agent traverses the tree from the root node, which corresponds to the current state in the real environment. In this tree formulation, each node represents a state s, while statistics such as the action-value $Q(s,a)$, prior probability $P(s,a)$, and visit count $N(s,b)$ are maintained on edges corresponding to state-action pairs. Starting from the root, the agent repeatedly selects the child node associated with the action that maximizes the PUCT [28] score until a leaf node is reached. The PUCT score is given by:

$$U(s,a) = Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}, \tag{4}$$

where c_{puct} is an exploration constant. Once a leaf node is selected, the search proceeds to the expansion phase. If the selected node is non-terminal and not fully expanded, the tree is expanded by adding a new child node corresponding to a previously unexplored action. This expansion is guided by the prior policy obtained from the neural network. And in the evaluation phase, a value is assigned to the newly expanded node of the tree. If the simulation reaches a terminal state, either due to a boundary violation or due to entry within the target's ϵ-neighborhood, large negative and large positive rewards are assigned, respectively. If the node corresponds to a non-terminal state, an intermediate reward as defined in Section 2.1 is assigned. The evaluated values are then back-propagated from the leaf node to the root, where they are added to the cumulative value of each node along the path and averaged across visits to update the state-action value $Q(s,a)$, while simultaneously incrementing the visit counts. After the MCTS simulations are complete, these visit counts are used to generate a refined probability distribution. These counts are raised to the power of a temperature hyper-parameter to control exploration and then normalized to form a probability distribution, from which an action is sampled.

The sampled actions generate state transitions and rewards, which are subsequently used to train the neural network. Given the dense reward structure of the MDP, training is performed in a batch-wise Temporal Difference (TD) manner. The training loss includes two components: a value loss and a policy loss. The value loss is the mean squared error between the target value derived from the TD update and the value predicted by the neural network:

$$\mathcal{L}_{\text{value}} = \frac{1}{N} \sum_{i=1}^{N} \left(V_{\text{pred}}(s_i) - V_{\text{target}}(s_i) \right)^2, \tag{5}$$

where $V_{\text{target}}(s_i)$ is the TD target defined as $r_i + \gamma \cdot V_{\text{pred}}(s_i') \cdot (1 - \text{done}_i)$. Here, r_i denotes the reward for transition i, $V_{\text{pred}}(s_i')$ is the network's value estimate for the next state s_i', γ is the discount factor, and N represents the batch size.

The policy loss is the cross entropy loss between the prior policy predicted by the neural network and the refined policy obtained from the tree search:

$$\mathcal{L}_{\text{policy}} = -\frac{1}{N} \sum_{i=1}^{N} \left[\sum_{a_x} \Pi_x^{(i)}(a_x) \log \pi_x^{(i)}(a_x) + \sum_{a_y} \Pi_y^{(i)}(a_y) \log \pi_y^{(i)}(a_y) \right], \tag{6}$$

where $\Pi^{(i)}$ represents the refined MCTS policy distribution (target) derived from visit counts, and $\pi^{(i)}$ is the network's predicted prior distribution, summed over all discrete actions for both x and y axes. The final objective function is obtained by summing these two components:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{value}} + \mathcal{L}_{\text{policy}}. \tag{7}$$

The parameters of the neural network are optimized through minimisation of the above objective function.

B Performance on Stochastic Variants

We evaluate AlphaGrad on stochastic variants of the benchmark functions to assess its robustness under noisy conditions. The evaluation functions are summarized in Table 1. In this setting, the agent is exposed to the same functional families used during training and evaluation of its deterministic variants. Here, stochasticity is introduced by adding Gaussian noise with zero mean and a standard deviation of 0.01 to each element of the input state vector independently. Across these stochastic benchmarks, AlphaGrad outperforms all traditional optimization methods on the Ackley [1], Beale [3], and Three-Hump Camel [15] functions (Tables 6, 7, and 8). The exception is the Bukin N.6 function [15] with stochasticity (Table 9), where classical methods retain an advantage.

Overall, these results indicate that AlphaGrad maintains strong performance under stochastic conditions and that planning-based reinforcement learning can benefit from controlled noise by enhancing exploration, leading to improved robustness across diverse optimization landscapes.

Table 6: Ackley Function With Stochasticity

Optimization Algorithm	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	**85.3**	$\mathbf{76.50 \pm 59.83}$	$\mathbf{61.11 \pm 43.34}$
BB2	0.7	198.60 ± 16.65	0.33 ± 0.14
BB1	0.0	200.00 ± 0.00	0.00 ± 0.00
Steepest Descent (Armijo Conditions)	18.8	173.75 ± 60.37	59.83 ± 58.35
Quasi Newton	4.9	194.95 ± 25.35	97.03 ± 55.08
Adam	0.1	199.80 ± 6.30	29.00 ± 0.00

Table 7: Beale Function With Stochasticity

Method	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	**94.7**	$\mathbf{21.74 \pm 22.24}$	$\mathbf{20.17 \pm 14.61}$
BB2	26.0	148.19 ± 87.40	0.74 ± 0.88
BB1	70.7	73.87 ± 88.12	21.60 ± 40.70
Steepest Descent (Armijo Conditions)	57.7	92.79 ± 94.50	14.04 ± 28.49
Quasi Newton	0.6	198.80 ± 15.42	0.29 ± 0.22
Adam	50.3	135.45 ± 76.94	71.66 ± 59.85

Table 8: 3 Hump Camel Function With Stochasticity

Optimization Algorithm	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	**96.9**	$\mathbf{7.96 \pm 4.07}$	$\mathbf{8.02 \pm 4.10}$
BB2	63.4	82.42 ± 92.61	14.54 ± 3.61
BB1	45.9	135.13 ± 47.49	19.30 ± 47.49
Steepest Descent (Armijo Conditions)	62.9	74.97 ± 96.16	1.12 ± 0.40
Quasi Newton	96.0	36.58 ± 50.82	29.77 ± 39.14
Adam	20.2	162.47 ± 75.55	14.21 ± 26.67

Table 9: Bukin N.6 Function With Stochasticity

Optimization Algorithm	**Success Rate (%)**	**Avg Steps**	**Avg Steps (Success)**
AlphaGrad (ours)	33.1	135.43 ± 89.57	11.56 ± 10.04
BB2	54.5	113.24 ± 86.95	40.81 ± 48.41
BB1	9.0	199.80 ± 6.32	1.00 ± 0.00
Steepest Descent (Armijo Conditions)	8.8	185.66 ± 49.17	39.75 ± 58.88
Quasi Newton	0.1	199.01 ± 12.45	58.74 ± 40.10
Adam	28.9	157.48 ± 73.71	42.88 ± 58.40