Pixel-Perfect Piloting: Superhuman Control of Pixelcopter via Reinforcement Learning

Tai Vu
Computer Science
Stanford University
taivu@stanford.edu

Brad Nikkel
Symbolic Systems
Stanford University
bnikkel@stanford.edu

Jenny Yang
Computer Science
Stanford University
jjyang1@stanford.edu

Abstract

The field of reinforcement learning has been growing constantly in the last few years, with a wide range of practical applications in automated game playing. Reinforcement learning enables an agent to learn the game environment without supervision and then devise optimal strategies to beat the game. This study provides a comprehensive comparative analysis of reinforcement learning (RL) algorithms for mastering the arcade game Pixelcopter, a sparse-reward environment. We implement and evaluate a range of value-based and policy-based methods, including variations of Q-Learning, Sarsa, Action Value Function Approximation, and Policy Gradients, enhanced with techniques like eligibility traces. All implemented models learned to play the game effectively and achieved superhuman performance. The Sarsa(λ) algorithm proved most effective, achieving a peak score of 379 and an average score of 47.03. This work offers valuable insights into the relative strengths and convergence properties of different RL approaches, highlighting the critical impact of hyperparameter optimization in complex, dynamic control tasks.

1 Introduction

Reinforcement Learning (RL) has emerged as a leading paradigm in artificial intelligence, enabling agents to achieve superhuman performance in complex sequential decision-making tasks, most notably in the domain of games. From mastering the ancient game of Go to conquering a suite of classic Atari titles, RL algorithms have demonstrated a remarkable ability to learn optimal strategies directly from environmental interaction, often without any prior human knowledge. These games serve as powerful and challenging benchmarks for RL research, particularly those with sparse reward structures—where feedback is infrequent—which mirrors the complexities of many real-world problems [11].

This paper investigates the application and comparative performance of various RL algorithms on Pixelcopter, a dynamic arcade game that, despite its simple controls, presents a significant challenge. In Pixelcopter, an agent must navigate a continuously scrolling, procedurally generated cave, avoiding obstacles with only two possible actions: "ascend" or "do nothing." [3]. Figure 1 shows a screenshot of the gameplay as the player avoids the vertical barrier as well as the cave floor and ceiling. This combination of a continuous, high-velocity state space and a sparse reward signal (a point is awarded only for clearing an obstacle) makes it an ideal environment for testing the limits of modern RL techniques.

To this end, we implement and rigorously evaluate a comprehensive suite of foundational RL algorithms. We explore both value-based methods, including several variations of Q-Learning and Sarsa, and policy-based methods, specifically Policy Gradients. Our analysis demonstrates that all implemented agents successfully learn to play Pixelcopter at a level that surpasses human performance. Notably, we find that Sarsa(λ), an on-policy method enhanced with eligibility traces, achieves the highest performance, reaching a maximum score of 379. This paper will detail our modeling of the

environment, delve into the specifics of each algorithmic approach, and present a thorough analysis of our experimental results and key findings.

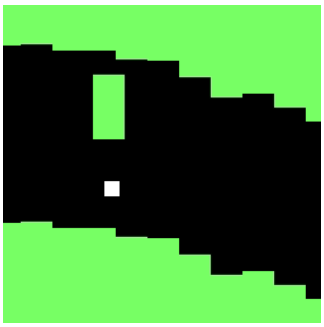

Figure 1: An Example Game of Pixelcopter

2 Related Work

The application of reinforcement learning (RL) to master video games has a rich history, serving as a critical benchmark for algorithmic development. One example is Brendan et al., which combined Q-Learning with policy gradients to attain high scores in a number of Atari games [9, 15]. Another example is Volodymyr et al., which offered a seminal, powerful approach of combining convolutions neural networks with Q-Learning by inputting arcade games' image pixels to find a value function that estimates future rewards [7].

Extant machine learning (ML) literature using Pixelcopter specifically is rather sparse. However, Pixelcopter is structured similarly to Flappy Bird, a popular game where the player must avoid the a continuous stream obstacles to survive. Flappy Bird has the same action space as Pixelcopter and can be described with the same state features. The wealth of research on Flappy Bird therefore provides a valuable foundation for our study. For example, Thurler et al. implemented genetic algorithms, Q-Learning, and actor-critic models. While all approaches could learn the game, Thurler et al. found that the genetic algorithms could reach the goal of never losing [13, 14].

Furthermore, there is some recent research on generalizing gameplay from simple computer games like Flappy Bird to more complex games in the real world. For example, Du et al. utilized modifications of Flappy Bird as a preliminary proof that machine learning can train human players to learn games. This paper and other research shows that reinforcement learning methods have potential beyond computer games and could potentially transform the way that we learn in everyday life [5, 16, 12].

Specific prior work on Pixelcopter includes a study by Mysore et al., which investigated the efficiency of actor-critic models. They found that using independent network structures for the actor and critic was more effective in resource-constrained environments than sharing a single network structure [8, 17]. Our work builds upon these foundations by providing a broad, comparative analysis of several foundational RL algorithms—including Q-Learning and Sarsa variants—applied specifically to Pixelcopter. We aim to fill a gap in the literature by systematically evaluating and optimizing these methods in the Pixelcopter environment to determine their relative efficacy.

3 Model

3.1 States

In our model, each state is a dictionary that contains 7 features, as described in Table 1. The terminal state occurs when the agent hits the ceiling, floor, or barrier and thus ends the current game.

In addition to using raw feature values, we experimented with a discretization approach in order to reduce the size of the state space and speed up the training process. This corresponds to dividing the original game screen of size 48×48 into discrete grids of size $s \times s$ for a discretization level

Feature	Description
y	The player's vertical position
v	The player's velocity
d_c	The player's vertical distance to the ceiling
d_f	The player's vertical distance to the floor
d_n	The player's horizontal distance to the next block
y_t	The vertical position of the top part of the next block
y_b	The vertical position of the bottom part of the next block

Table 1: State representation of Pixelcopter

s. Every point in a grid would share the same feature value. Particularly, for a real number n and a discretization factor s, we applied the following discretization function.

$$f(n, s) = s \left\lfloor \frac{n}{s} \right\rfloor$$

3.2 Actions

At each time step, there are 2 possible actions: 0 and 1, which represent "moving down" and "moving up" respectively.

3.3 Rewards

The original game emulator produces a reward of 1 every time the player passes a block. In all other cases, the reward is 0. We believed that this sparse reward scheme would slow down the convergence of our algorithms, because the agent received a reward of 0 most of the time. Therefore, we defined the reward function as follows:

$$R(s, a) = \begin{cases} 3 & \text{if the agent successfully passes a block} \\ -2000 & \text{if the agent hits a block and loses the game (terminal state)} \\ 1 & \text{if the agent survives at the current time step without hitting anything} \end{cases}$$

Intuitively, a negative reward of high magnitude like -2000 would discourage the model from entering the terminal state. Meanwhile, a small "survival bonus" of 1 encourages life preservation, even if the player has not yet passed a block, so it would avoid moving randomly between consecutive blocks. Hence, this approach helps speed up the training process.

4 Algorithms

4.1 Baseline

We applied a random policy as a baseline algorithm. Specifically, at each state, the agent randomly picks one of the two actions (up or down) with probabilities p and $1 - p$ respectively. We selected $p = 0.5$ in this case. This was a good starting point, because it was simple to implement and clearly defined a baseline strategy that a well-trained algorithm should outperform.

4.2 Q-Learning

Q-Learning is an off-policy algorithm that aims to estimate the action value function $Q(s, a)$. Specifically, for each training example (s, a, r, s') that contains a state s, an action a, a reward r, and a subsequent state s', this method applies the following incremental update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

In this formula, α is a learning rate, and γ is a discount factor.

4.3 Sarsa

Sarsa is an on-policy algorithm that also estimates the action value function $Q(s, a)$. However, instead of maximizing over all possible actions, Sarsa utilizes the next action a' to update Q. For each training example (s, a, r, s', a') that contains a state s, an action a, a reward r, a subsequent state s', and a subsequent action a', this technique performs the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma Q(s', a') - Q(s, a)\right)$$

where, as with Q-Learning, α is a learning rate, and γ is a discount factor.

4.4 Action Value Function Approximation

In this approach, we model the action value function using a parametric form $Q_\theta(s, a)$ with some parameter θ. Then, we minimize the loss between the estimate $Q_\theta(s, a)$ and the optimal action value function $Q^*(s, a)$, which is defined as:

$$l(\theta) = \frac{1}{2} \mathop{\mathbb{E}}_{(s,a)\sim\pi^*} \left[\left(Q^*(s, a) - Q_\theta(s, a)\right)^2\right]$$

where $Q^*(s, a)$ can be estimated as $Q^*(s, a) \approx r + \gamma \max_{a'} Q_\theta(s', a')$.

This can be done by performing the following gradient descent update:

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)\right) \nabla_\theta Q_\theta(s, a)$$

In this study, we model $Q_\theta(s, a)$ with a feed-forward neural network with 2 hidden layers. The network takes as inputs the state representation of s and then outputs two values in the last layer, which corresponds to the action values for $a = 0$ and $a = 1$. We chose this modeling technique instead of inputting (s, a) into the network and outputting a single number because our approach takes advantage of parameter sharing, which allows the model to learn the action value function for both $a = 0$ and $a = 1$ at the same time in each gradient descent update.

4.5 Policy Gradients

Policy gradients differ from Q-Learning and Sarsa's value-based approach by instead optimizing the policy directly by learning a probability distribution for actions given the state space. We used a "rewards-to-go" policy gradient that generates trajectories $\tau = \{(s_1, a_1, r_1) \cdots (s_t, a_t, r_t)\}$ where t is the total steps until Pixelcopter reaches its terminal state, and s_i, a_i, and r_i are the i-th state, action, and reward, respectively, in trajectory τ. Then, where $R(\tau)$ is the discounted rewards of trajectory τ and π_θ is our parameterized policy, we find:

$$\max_\theta \mathbb{E}_{\pi_\theta} R(\tau)$$

by performing a gradient ascent update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau)$$

where α is the learning rate. This is simplified by employing logarithms to get: [6]

$$\nabla U_\theta = \mathbb{E}\left[\sum_{i=1}^{t} \nabla_\theta log \pi_\theta(a_i|s_i) \gamma^{i-1} R(\tau)\right]$$

where γ is the discount factor. Policy Gradient gradually takes steps towards a policy that maximizes the expected discounted return of trajectories.

4.6 Additional Strategies

4.6.1 ϵ-greedy Exploration

In order to manage the trade-offs between exploration and exploitation, we leveraged ϵ-greedy exploration. Specifically, for each state s, the agent chooses a random action with probability ϵ; otherwise, the agent picks the greedy action $a^* = \arg\max_a Q(s, a)$. For our ϵ-greedy strategy, we experimented with decaying ϵ slightly per training iteration (with some decay factor) toward some minimum floor ϵ.

4.6.2 Forward and Backward Updates

Normally, in Q-Learning and Sarsa, we performed the update formula in a forward order (from the first frame to the last frame in a sampled trajectory). However, we also experimented with backward updates, which apply the update rule in the opposite direction. We believed that this practice would allow important information (like hitting a block or hitting the cave's floor or ceiling) to propagate faster through the state space.

4.6.3 Eligibility Traces

We also implemented Q-Learning and Sarsa with eligibility traces (Q(λ) and Sarsa(λ)). Eligibility traces can speed up learning of sparse rewards by propagating rewards back in time with an exponential decay parameter λ. To do this, we modify the above Q-Learning and Sarsa algorithms by adding the temporal difference update δ:

$$\delta = r + \gamma Q(s', a') - Q(s, a)$$

where s' and a' and the current state-action pair and s and a and the previous state-action pair.

Every state-action value function is then updated as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta N(s, a)$$

where $N(s, a)$ is a count of the times a state-action pair has been visited.

Finally, state-action visit counts are decayed via the the discount, γ, and the exponential decay, λ:

$$N(s, a) \leftarrow \gamma \lambda N(s, a)$$

Given that eligibility traces are most impactful in sparse-reward environments [6] like Pixelcopter, we hypothesized that eligibility traces may yield an improvement upon the vanilla Q-Learning and Sarsa models.

4.6.4 Nearest Neighbors

Because each state has 7 features, when the model begins training there are many unseen states. As a result, we implemented nearest neighbor approximation to approximate unseen states to a seen state. For each unseen state, we found the nearest seen state, calculated with Euclidean distance using all 7 features of the state. Thus, the approximate value function is

$$U_\theta(s) = \theta_i$$

where

$$i = \arg \min_{j \in 1:m} d(s_j, s)$$

and $d(a, b)$ represents the Euclidean distance between two states a and b.

The performance with nearest neighbor approximation was similar to randomly selecting an action for unseen states. We hypothesize that this is due to two factors. First, the distance function abstracts 7 features into a single number, which can only generally capture the similarity between two states. Second, the action space has only two actions, so randomly selecting an action is a sufficiently good baseline.

4.6.5 Experience Replay

In our action value function approximation algorithm, we applied experience replay in order to mitigate the issue of catastrophic forgetting (which means that the model forgets about variable information it learned in the past). In particular, a replay memory is used to store a fixed quantity of the most recent experience tuples during the training pipeline. After that, this method samples a batch of tuples uniformly and then feeds the batch to the gradient descent update.

5 Implementation

We implemented the reinforcement learning algorithms using Python [2] and PyTorch [10]. In addition, we leveraged the Pixelcopter emulator provided by PyGame Learning Environment (PLE) [1] and OpenAI Gym [4]. The code can be found at https://github.com/taivu1998/Pixelcopter-RL.

6 Experiments and Results

6.1 Evaluation Metrics

We use the total score as a evaluation metric for our models. In each game, the player gets 1 point when passing each vertical block, so the final score is equal to the total number of blocks that the player passes during that game.

6.2 Experiment Setup

Given our goal was to maximize the score for each model, we extensively experimented with hyperparameters to optimize each model for a high score. We utilized the same random seed throughout our experiments and evaluated the mean, standard deviation, minimum, and maximum scores for one hundred evaluation epochs per every thousand training epochs to get a sense of how each model learned over time.

Beyond the discretization-factor, which we discuss later, we found one of the more influential hyperparameters to be the discount factor. The ideal discount factor varied significantly between models. For example, 0.96 and 0.95 discount factors produced the highest mean scores for Policy Gradient and Sarsa respectively, while a 0.9 discount factor fared better for Q-Learning and 0.92 was ideal for Q-Learning Nearest Neighbors. Typically, if we lowered the discount factor too low, the training time decreased in concert with the mean score.

Next, the amount of training epochs needed to reach a high score varied considerably amongst models. We found that Q-Learning and Sarsa, and Q-Learning Nearest Neighbors took approximately 50,000 training epoch reach their maximum scores, whereas Q(λ) and Sarsa(λ) took approximately 10,000 training epochs. Q-Learning Function Approximation took approximately 500,000 training epochs to achieve a score modestly better than random. This demonstrated to us that our different models would be more or less appropriate depending on computing resources and time allocated to training.

Our models' optimal learning rates varied considerably too. Our 'rewards-to-go' Policy Gradient needed a small initial learning rate of approximately 0.0001 to train well and we found that decaying Policy Gradients learning rate very slightly each training epoch helped achieve a higher mean score. Policy Gradient's optimal learning rate range was considerably smaller than our Q-learning and Sarsa variants' optimal learning rate range of 0.01-0.04.

The optimal setting for Q(λ) and Sarsa(λ)'s eligibility traces was generally in the range 0.9-0.92 and our optimal epsilon range, 0.01-0.02, was the hyperparameter was the most consistent across all our models. Since we decayed our epsilon slightly after every training epoch, we found that an optimal epsilon was best adjusted slightly higher with increased training epoch and vice versa to encourage more early exploration or less dependent on the number of training iterations.

Algorithm	Min	Max	Average	Std. Dev.	Train Time
Baseline (random)	0	4	0.117	0.371	N/A
Q-Learning	2	253	30.963	25.853	45:46
Q(λ)	6	249	35.127	28.549	19:39
Q-Learning (nearest neighbors)	0	213	33.241	27.852	28:39
Sarsa	6	319	39.215	34.081	42:40
Sarsa(λ)	4	**379**	**47.030**	36.245	2:21:48
Policy gradients	3	221	22.141	18.492	08:25
Action value function approximation	2	36	6.264	5.174	1:01:41

Table 2: Performance of Different Algorithms

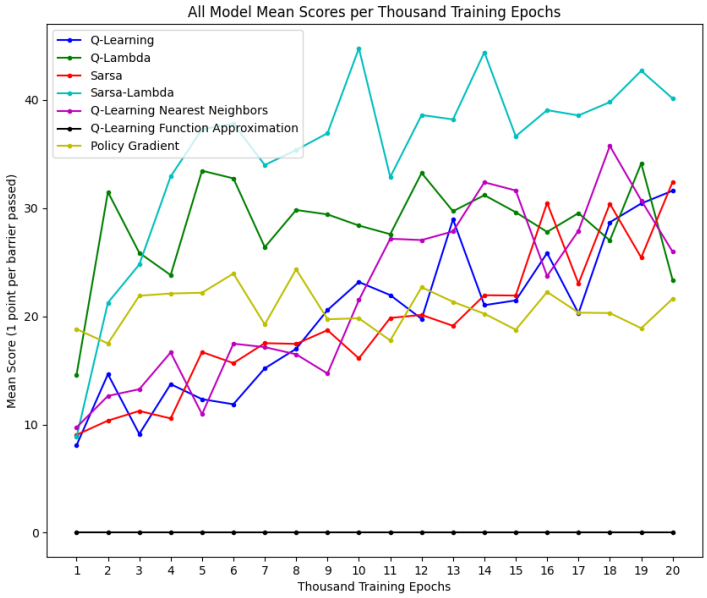

Figure 2: All Models' Mean Scores (per 1000 iterations)

6.3 Results

6.3.1 Model Performance

We ran all our models with many different hyperparameters and recorded their respective maximum score results in Table 2. We can compare different techniques based on their maximum and average scores, which represent the agents' peak performance and performance consistency respectively. We can see that all the models significantly outperformed the baseline algorithm. In addition, the Sarsa(λ) agent with (eligibility traces) performed the best, with a maximum score of 379 and an average score of 47.03.

Figure 2 illustrates the curves for the mean scores produced by our models during the training process. The general upward trends of the curves demonstrate that the agents gradually learn the patterns of the game environment and make better decisions as we train them for more iterations.

Meanwhile, the use of nearest neighbors did not seem to impact the Q-Learning algorithm, as the models with and without nearest neighbors had relatively similar outcomes. At the same time, we can see that traditional reinforcement learning techniques like Q-Learning, Sarsa, and Policy Gradient performed much better than a complex, deep learning-based approach like action value function approximation. This might be because deep neural networks are significantly harder to train, as they

must learn good representations for all states in a large state space. We were able to train the deep learning model for a small number of epochs due to limited resources, so more training iterations might be required in order for its parameters to converge. Were we to employ more computing resources to achieve this, it is likely that the deep learning model would outperform our model, but with limited training epochs, our models outperformed the deep learning model.

6.3.2 Q-Learning versus SARSA

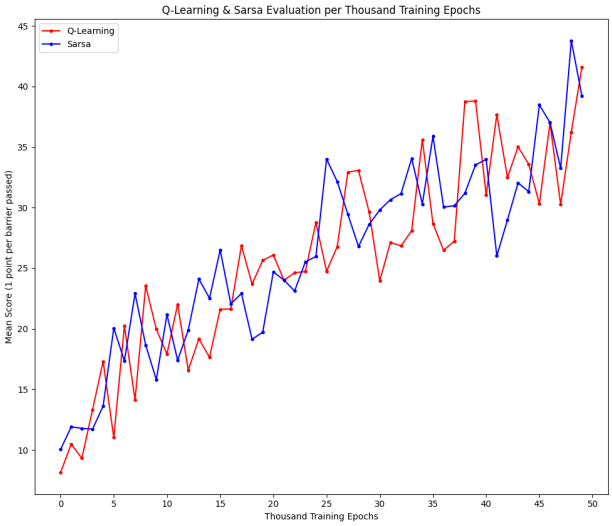

Figure 3: Q-Learning and Sarsa Mean Scores per 1000 iterations

As shown in Figure 3, vanilla Q-Learning and Sarsa learned at similar rates and achieved similar mean scores. To achieve their highest mean scores, they took similar period of time to train at approximately 45 minutes for 50,000 training epochs. In the Q-Learning and Sarsa error bar plots, we see that Q-Learning (Figure 8) and Sarsa (Figure 10) gradually, continuously learned more optimal policies over 20,000 training epochs.

6.3.3 The Effects of Eligibility Traces

As shown in Figure 4, our hypothesis that adding eligibility traces would improve Q(λ) and Sarsa(λ)'s mean scores compared to their vanilla counterparts was correct. We also see in Figure 4 that eligibility traces for Q(λ) and Sarsa(λ) have similar performance in the first few thousand training epochs, but after approximately 7,000 training epochs, Sarsa(λ) achieves significantly higher mean scores. Finally, it is worth noting that while Q(λ) and Sarsa(λ) learned optimal play in fewer training epochs, their training times (to achieving their maximum scores) did not vary significantly from regular Q-Learning and Sarsa, meaning each training epoch tended to take significantly longer with eligibility traces than without eligibility traces.

6.3.4 The Effects of Forward/Backward Updates

We found that backwards updates impacted the training process of Q-Learning and Sarsa, but in ways we had hypothesized. For normal Q-learning, forward and backward updates did not significantly affect convergence to the optimal policy. In normal Sarsa, however, we see that forward updates vastly outperform backward updates (see Figure 6). In Q(λ) and Sarsa(λ), we see a slightly different pattern; both Q(λ) and Sarsa(λ) converged toward optimal policies significantly faster with forward updates than they did with backwards updates (see Figure 5). This is counter-intuitive since we know that the last state in a sampled trajectory should contain the most important information, namely the frame where the player hits a block, initiating the terminal state. By updating the Q-Learning or Sarsa

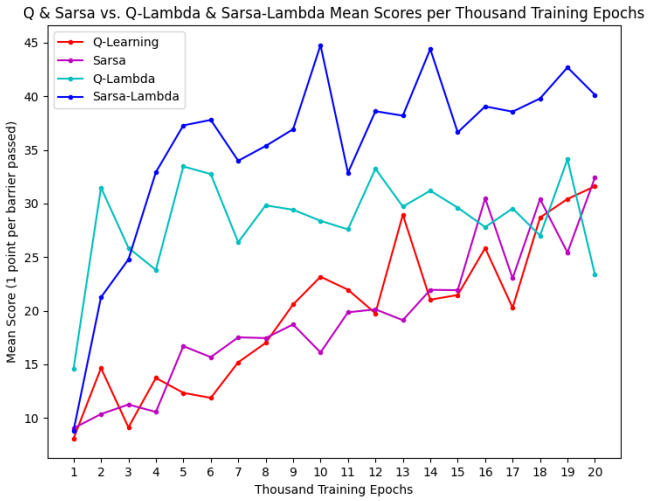

Figure 4: Q-Learing and Sarsa versus Q(λ) and Sarsa(λ) Mean Scores per 1000 iterations

in a backward order, we expected this more important experience to propagate through all the earlier states in a single iteration, allowing the agent to quickly gain knowledge about the bad states that led to hitting barriers and thus avoid these states in subsequent iterations. It seems that this was indeed the case for Sarsa backwards compared to forwards.

Surprisingly, however, Q(λ) and Sarsa(λ) achieved significantly higher mean scores with the normal forward update order, as shown in Figure 5. We know that Sarsa considers the next action a' instead of the optimal action $\arg\max_{a'} Q(s', a')$ as in Q-Learning. Meanwhile, Q(λ) and Sarsa(λ) added the use of eligibility traces in their rule. We suspect that these differences may have negatively affected the behavior we expected from backward updates, but more experiments are needed to answer this question.

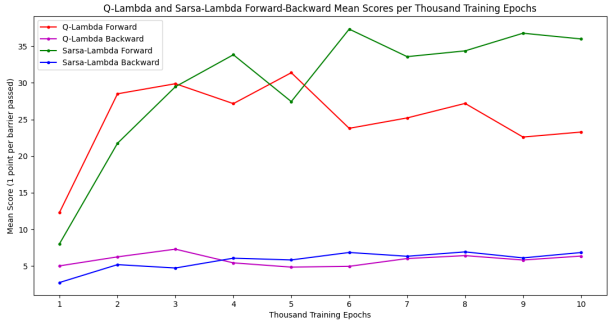

Figure 5: Q(λ) and Sarsa(λ) Forward-Backward Comparison: Mean Scores per 1000 iterations

6.3.5 The Effects of ϵ-greedy Exploration

We found that decaying epsilon slightly with each update, while maintaining a floor epsilon value, significantly benefited Sarsa and Sarsa(λ) and slightly benefited Q-Learning and its variants. Particularly when training for larger epochs, it was useful use epsilon decay to make Q-Learning and Sarsa more greedy as time progressed.

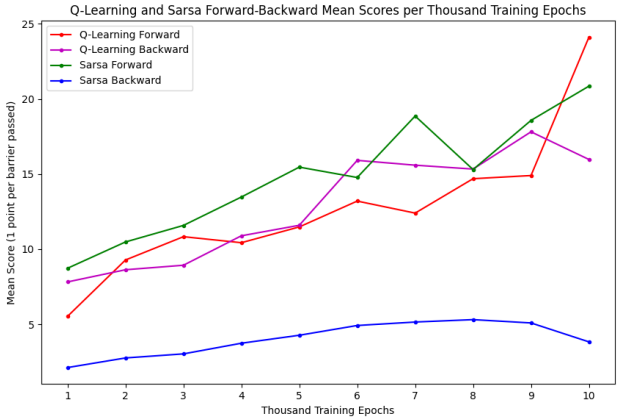

Figure 6: Q-Learning and Sarsa Forward-Backward Comparison: Mean Scores per 1000 iterations

6.3.6 The Effects of Discretization Levels

As described above, given Pixelcopter's 7-feature state spaces of floats, we discretized our state space by dividing the state-space floats by an integer, converting the results to integers, and then re-scaling them. We found our discretization factor to be the most impactful hyperparameter across our models, with the exception of action value function approximation, which was unsurprising as action value function approximation is more suited toward continuous state spaces. We found the ideal discretization factor range to be between 8 and 12. If we discretized our state space above this range, our models learned faster but could not attain as high a score. When we lowered the discretization factor below 8, the models tended to get significantly lower mean scores within the same amount of epochs, though with many more training epochs they may have matched or exceeded the highest means scores our models obtained in ideal 8-to-12 discretization range. This behavior can be seen in Figure Figure 7, which shows different discretization factors effects on Sarsa(λ)'s mean scores over training epochs.

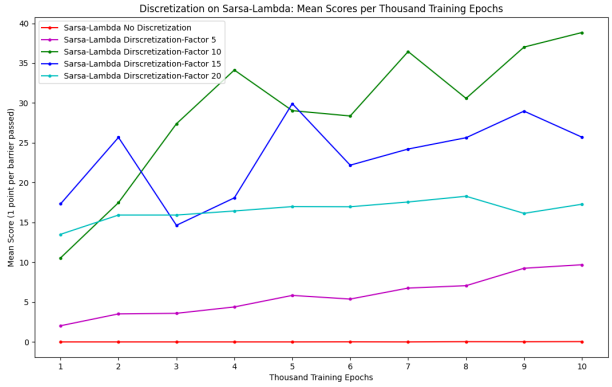

Figure 7: Discretization Factors on Sarsa(λ): Mean Scores per 1000 iterations

7 Conclusion

This research successfully demonstrates that the game Pixelcopter, despite its challenging dynamics and sparse-reward environment, can be effectively mastered using a range of reinforcement learning techniques. Our comprehensive investigation into value-based and policy-based methods confirms

that all implemented models, including variations of Q-Learning, Sarsa, and Policy Gradients, learn to play the game at a superhuman level, far surpassing the baseline performance.

The key finding of our comparative analysis is the superior performance of the Sarsa(λ) algorithm, which achieved a remarkable maximum score of 379 and a leading average score of 47.03. This highlights the significant benefit of employing eligibility traces, which accelerate learning by efficiently propagating reward information through the state space. Furthermore, our experiments with hyperparameter tuning, particularly discretization levels, underscore their critical role in balancing learning speed and peak performance. Ultimately, this work provides a rigorous case study on the relative efficacy of foundational RL algorithms, offering valuable insights for tackling similar sequential decision-making problems.

8 Future Work

Building on the findings of this study, several exciting avenues for future research could further advance performance and deepen our understanding of reinforcement learning in this context.

8.1 Advanced Algorithmic Exploration

While our study focused on foundational algorithms, exploring more advanced, state-of-the-art methods could yield even better results.

- Actor-Critic (A-C) Models: Implementing A-C architectures would be a logical next step, as they combine the strengths of both policy-based and value-based methods to potentially achieve more stable and efficient learning. Comparing a well-tuned A-C model against our existing results would provide a clearer picture of the performance hierarchy.
- Deep RL with Pixel Data: To eliminate the need for manual feature engineering, future work should explore an end-to-end learning approach using a Convolutional Neural Network (CNN). By feeding raw pixel data from the game screen directly into the network, the agent could learn its own hierarchical feature representations, a technique proven to be highly effective in mastering complex visual environments like the Atari suite.

8.2 Sophisticated Optimization and Analysis

Future efforts could focus on more systematic and exhaustive optimization to push the performance limits of the existing models.

- Automated Hyperparameter Tuning: Rather than manual tuning, employing automated optimization techniques like Bayesian optimization or genetic algorithms could systematically search the hyperparameter space to find the optimal configuration for each model, ensuring that each is performing at its absolute peak.
- Advanced Feature Engineering: New, informative features could be engineered by creating interaction terms from the existing state variables (e.g., the difference between the player's vertical position and the opening of the next barrier). This could provide the agent with more salient information for making decisions.
- Extended Training and Convergence Analysis: Given that some models were still improving at the 50,000-iteration cap, training the agents for a significantly longer duration would allow us to fully map their learning curves and identify their true performance plateaus.

References

[1] Pygame learning environment. URL https://pygame-learning-environment.readthedocs.io/.

[2] Python. URL https://www.python.org/.

[3] Pixelcopter, 2016. URL https://pygame-learning-environment.readthedocs.io/en/latest/user/games/pixelcopter.html.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. URL https://arxiv.org/abs/1606.01540.

[5] X. Du, X. Fuqian, J. Hu, Z. Wang, and D. Yang. Uprising e-sports industry: machine learning/ai improve in-game performance using deep reinforcement learning. In *2021 International Conference on Machine Learning and Intelligent Systems Engineering (MLISE)*, pages 547–552, 2021. doi: 10.1109/MLISE54096.2021.00112.

[6] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray. *Algorithms for Decision Making*. MIT Press, 2022. URL https://mitpress.mit.edu/9780262047012/algorithms-for-decision-making/.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[8] S. Mysore, B. Mabsout, R. Mancuso, and K. Saenko. Honey, i shrunk the actor: A case study on preserving performance with smaller actors in actor-critic rl, 2021. URL https://arxiv.org/abs/2102.11893.

[9] B. O'Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih. Combining policy gradient and q-learning, 2017. URL https://arxiv.org/abs/1611.01626.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

[11] H. Sheikh, S. Khadka, S. Miret, and S. Majumdar. Learning intrinsic symbolic rewards in reinforcement learning, 2020. URL https://arxiv.org/abs/2010.03694.

[12] M. Sun, T. Vu, and A. Wang. Privacy preserving inference of personalized content for out of matrix users, 2025. URL https://arxiv.org/abs/2508.14905.

[13] L. Thurler, J. Montes, R. Veloso, A. Paes, and E. Clua. Ai game agents based on evolutionary search and (deep) reinforcement learning: A practical analysis with flappy bird. In J. Baalsrud Hauge, J. C. S. Cardoso, L. Roque, and P. A. Gonzalez-Calero, editors, *Entertainment Computing – ICEC 2021*, pages 196–208, Cham, 2021. Springer International Publishing. ISBN 978-3-030-89394-1.

[14] T. Vu and L. Tran. Flapai bird: Training an agent to play flappy bird using reinforcement learning techniques, 2020. URL https://arxiv.org/abs/2003.09579.

[15] T. Vu and R. Yang. Bert-vqa: Visual question answering on plots, 2025. URL https://arxiv.org/abs/2508.13184.

[16] T. Vu and R. Yang. Ganime: Generating anime and manga character drawings from sketches with deep learning, 2025. URL https://arxiv.org/abs/2508.09207.

[17] T. Vu, E. Wen, and R. Nehoran. How not to give a flop: Combining regularization and pruning for efficient inference, 2020. URL https://arxiv.org/abs/2003.13593.

Appendix A: Additional Plots for the Scores of All the Algorithms

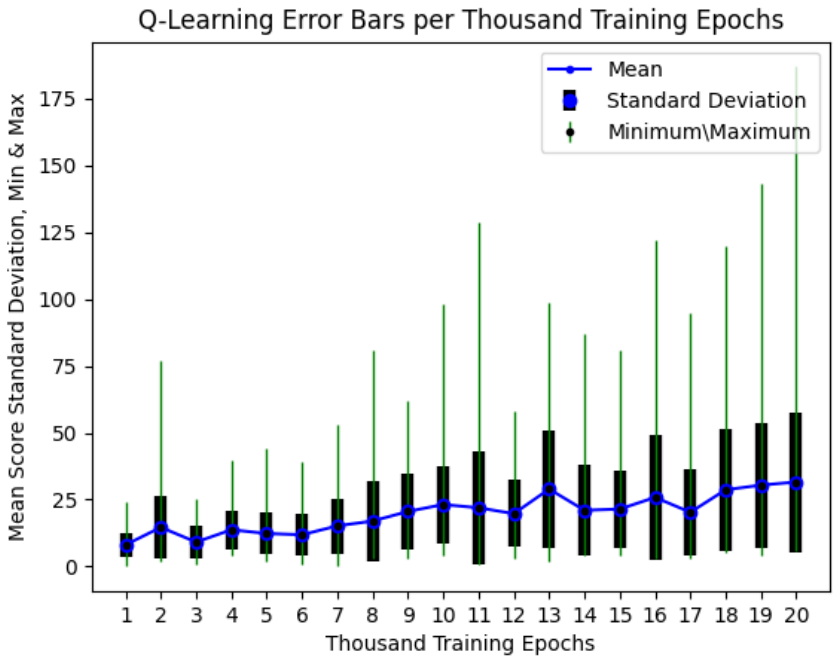

Figure 8: Q-Learning Means, Standard Deviations, and Extrema

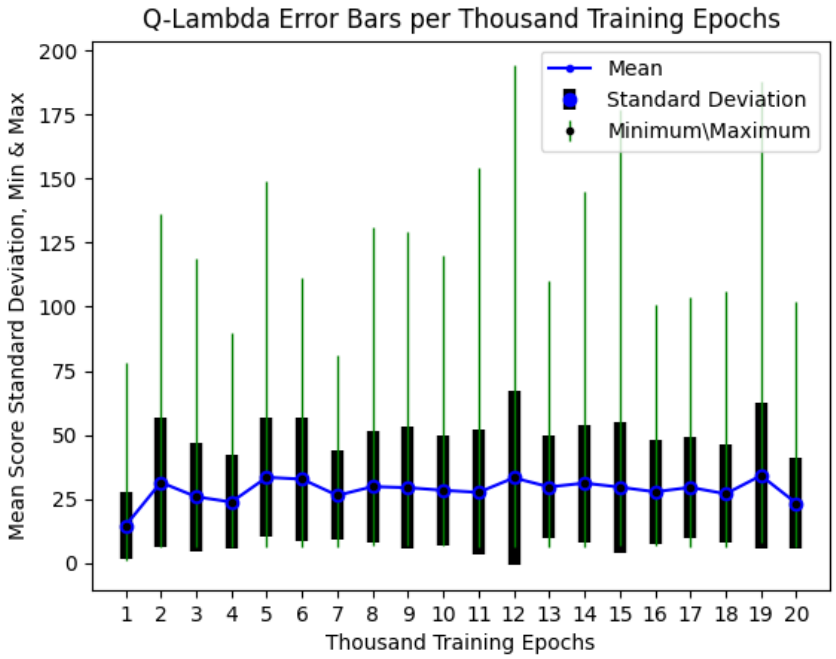

Figure 9: Q(λ) Means, Standard Deviations, and Extrema

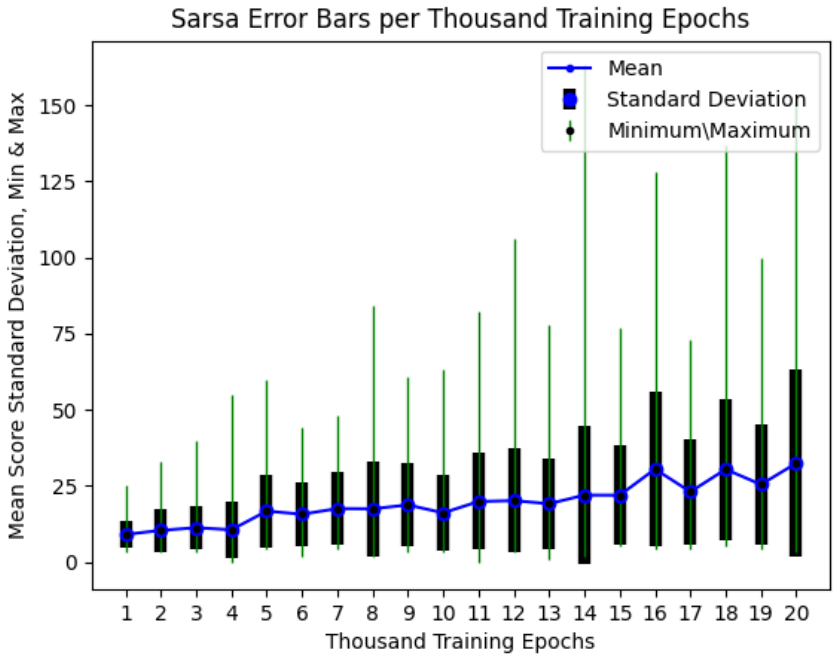

Figure 10: Sarsa Means, Standard Deviations, and Extrema

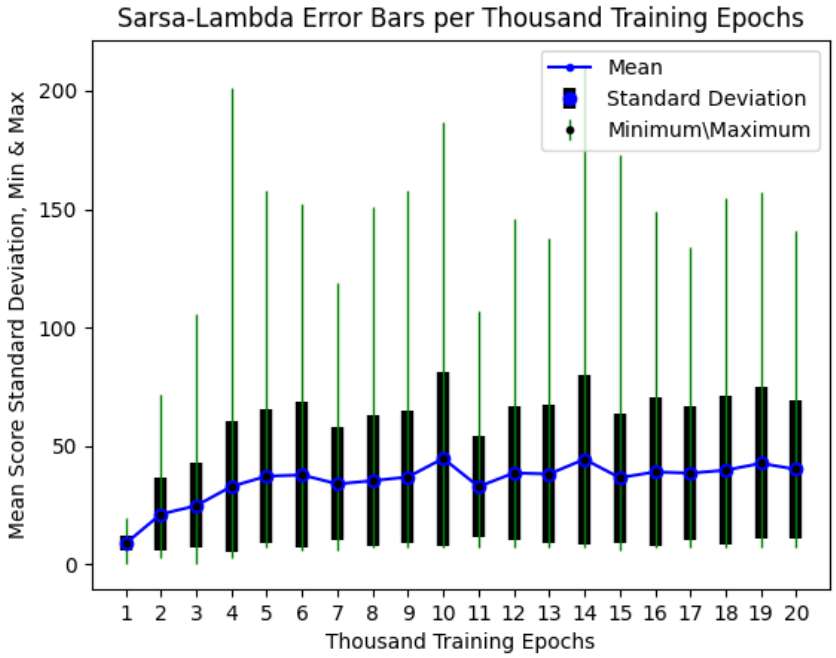

Figure 11: Sarsa(λ) Means, Standard Deviations, and Extrema

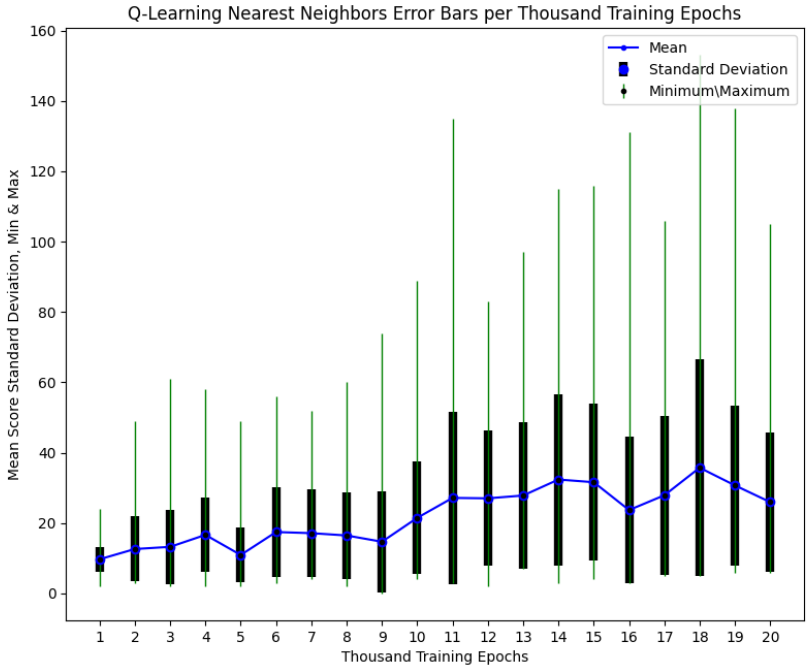

Figure 12: Q-Learning Nearest Neighbors Means, Standard Deviations, and Extrema

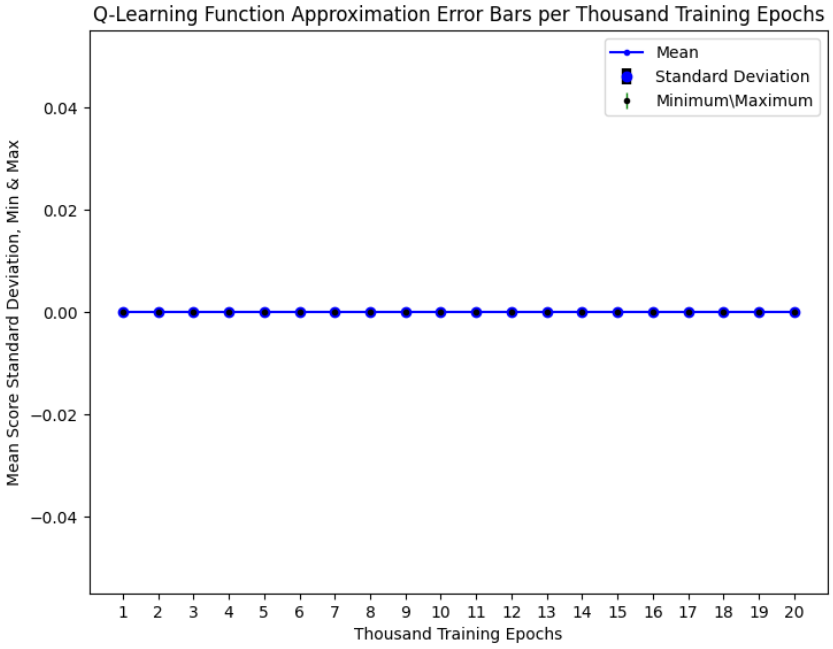

Figure 13: Action Value Function Approximation Error Bars Means, Standard Deviations, and Extrema

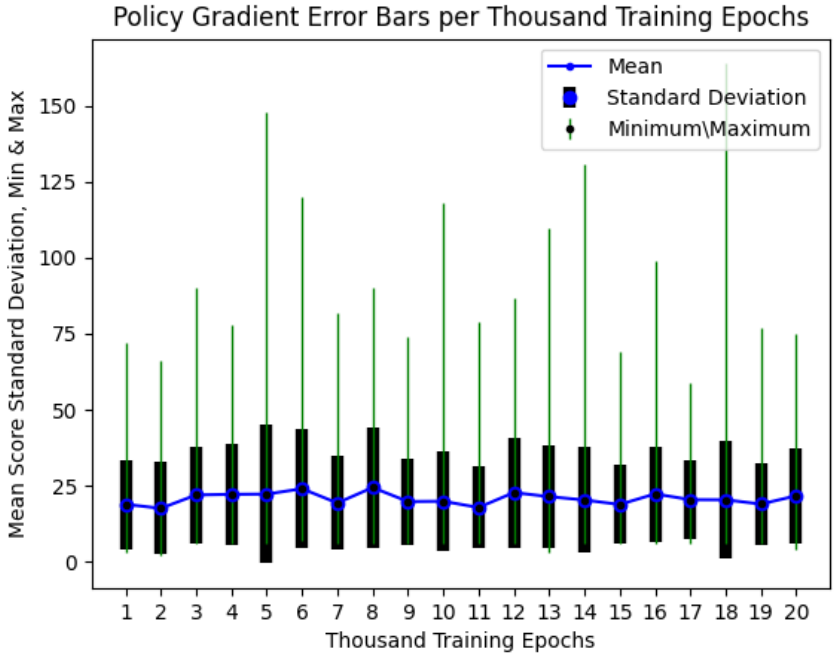

Figure 14: Policy Gradient Error Bars Means, Standard Deviations, and Extrema

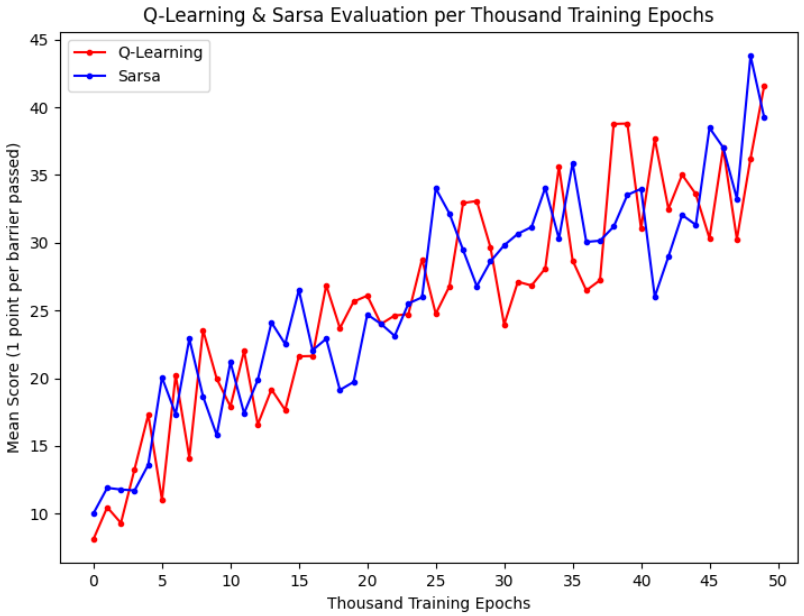

Figure 15: Q-Learning and Sarsa - 50k Training Epochs

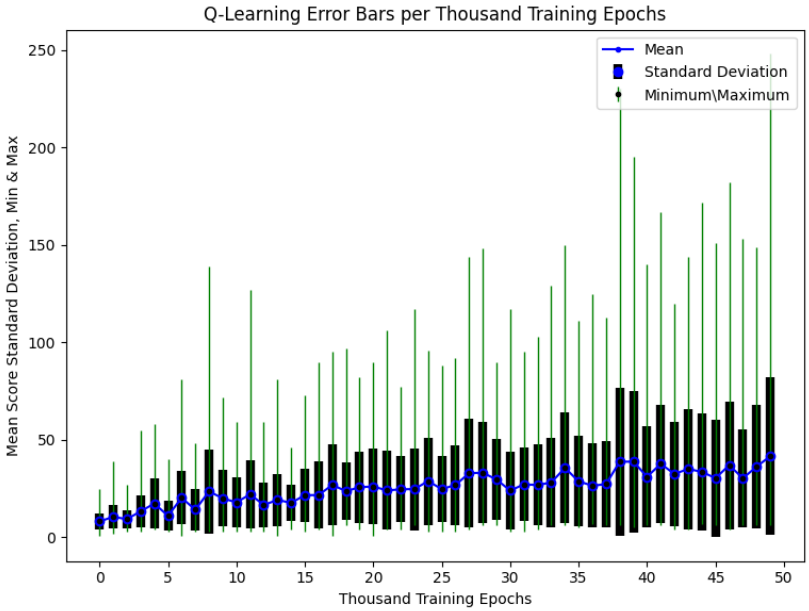

Figure 16: Q-Learning Error Bars - 50k Training Epochs

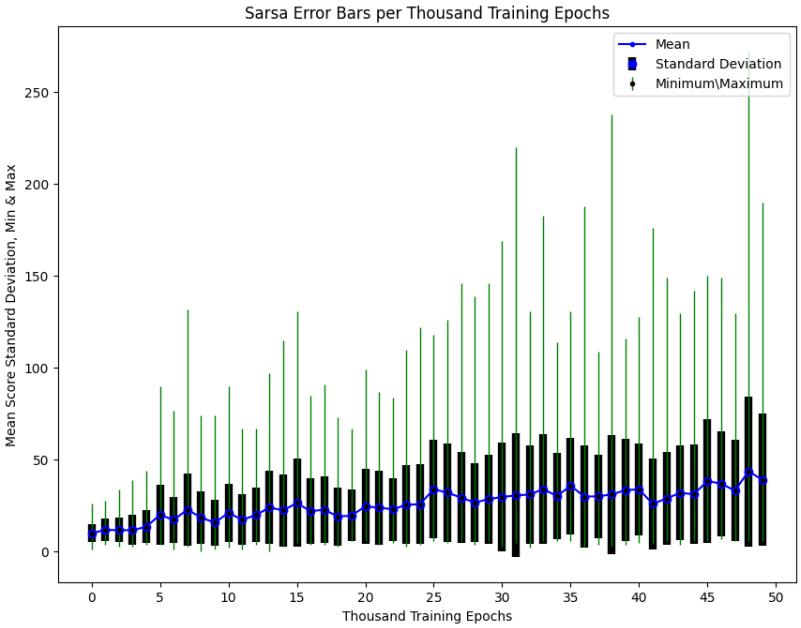

Figure 17: Sarsa Error Bars - 50k Training Epochs

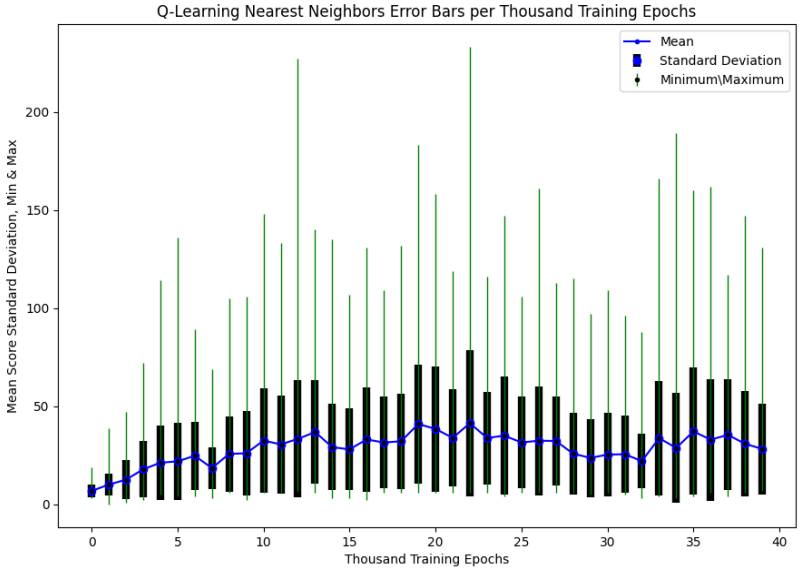

Figure 18: Q-Learning Nearest Neighbors - 40k Training Epochs

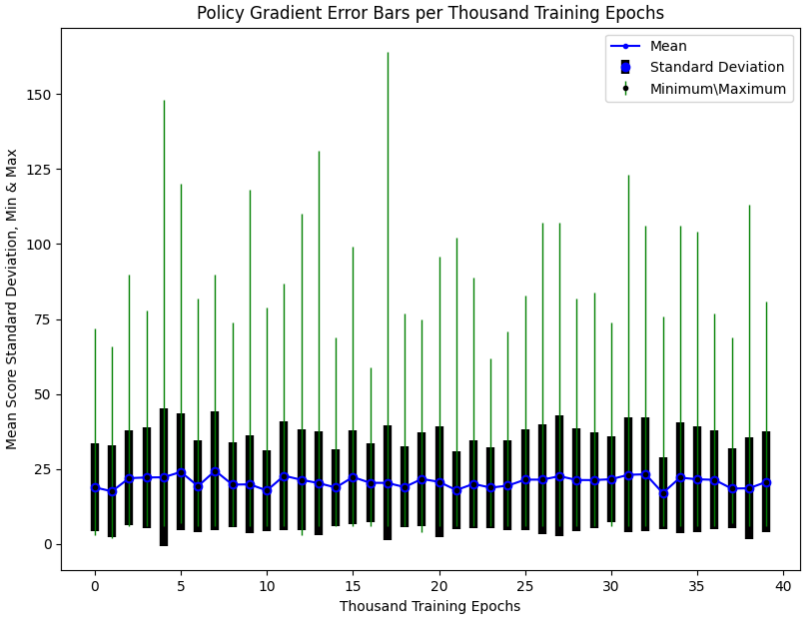

Figure 19: Policy Gradient Error Bars - 40k Training Epochs