Deep Reinforcement Learning for Geometry Dash

Dylan Lu
UCSB
dylanlu@ucsb.edu

Ishan Katpally
UCSB
ishankatpally@ucsb.edu

Surya Gunukula
UCSB
suryagunukula@ucsb.edu

Vigyan Sahai
UCSB
vigyan@ucsb.edu

Om Mahesh
UCSB
ommahesh@ucsb.edu

Demo: https://youtu.be/PKDMGPf-PEA

GitHub: https://github.com/ThePickleGawd/geometry-dash-ai

Abstract

With over 700 million downloads worldwide, Geometry Dash is notoriously one of the hardest platformer games in the world. Despite its difficulty, however, the controls are actually quite simple: either jump, or don't jump. Thus, Geometry Dash is an interesting challenge for deep learning, fitting nicely into the class of Atari-style games with sparse rewards. In this paper, we introduce a state-of-the-art RL agents to tackle Geometry Dash, beating the first two levels and outperforming all other existing image-only models.

1 Introduction

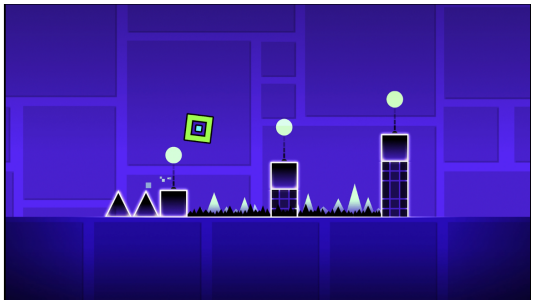

Figure 1: Geometry Dash Gameplay

Geometry Dash is a fast-paced 2D platformer where players must time their jumps to avoid obstacles and survive through increasingly complex levels. In Geometry Dash, the player constantly moves through a level filled with spikes, gaps, and other hazards; one mistake results in restarting from the beginning of the level. Unlike traditional turn-based or grid-based games, Geometry Dash's continuous, high-speed gameplay presents a unique challenge for reinforcement learning (RL) agents.

Previous work in Deep RL has featured games such as Atari 2600 games, as demonstrated in the landmark Deep Q-Network paper, "Playing Atari with Deep Reinforcement Learning"[6], where agents learned to play games like Breakout and Space Invaders directly from pixels. These environments offer standardized benchmarks and well-defined rewards, making them ideal for early research in

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

visual RL. In contrast, Geometry Dash has no pre-built environment and poses challenges such as sparse rewards, rapid failure, and precise timing, requiring custom solutions for observation, control, and exploration.

2 Related Work

Most previous attempts to train AI for Geometry Dash have been informal, typically demonstrated through YouTube videos. One example is a video by the YouTube channel "CodeNoodles" [1], where an agent learns to play a clone of Geometry Dash using neural networks. However, this approach does not use reinforcement learning techniques and instead employs a neuroevolution algorithm, mutating networks that make it further in the level. Most importantly, this method does not take in visual input, receiving only input about how far away the next obstacle is; our model gets no such convenience.

There was also a project from Stanford CS231 in 2017 [5] that aimed to train an RL agent using visual input and a Deep Q-Network. The authors modded the game to extract frames and simulate keypresses, but encountered significant difficulties, as their model was prone to overfitting and was incentivized to do nothing for a higher reward. The project was also unable to beat the first level of the game, even when overfitting to it.

Additionally, there have been various open-source attempts at Geometry Dash RL. GDind [7] was published two weeks ago and provides an environment to train your own model on the game. However, it does not release model weights, and its setup relies on rudimentary screen recording and simulated mouse inputs. This motivates the need for a more robust, environment-aware framework—one we aim to address in this work.

3 Background

We employ a Deep Q-Network (DQN) to learn the game. The DQN is a value-based method that approximates the Q-function, using a deep neural network to map state-action pairs to expected future rewards. At each step, the agent selects an action using an epsilon-greedy policy and learns to minimize the difference between predicted and target Q-values. To stabilize training, the DQN trains off a target network using a memory buffer, which samples from past experiences. We use the following loss function:

$$\mathcal{L} = (\mathcal{R}_{t+1} + \gamma \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2$$

$\mathcal{R}$: reward
γ: discount factor
$q_{\overline{\theta}}(S_{t+1}, a')$: target Q-value function
$q_\theta(S_t, A_t)$: predicted Q-value function
S_t: current state
A_t: action taken at time t

Our environment is particularly interesting because of the sparse nature of the reward system in Geometry Dash. The only obvious achievements are surviving longer and completing a level, making learning far harder. Furthermore, there are many failure states, where any move results in death, so the agent must also plan over longer horizons to avoid such states. These failure states can introduce discontinuities in the reward structure, making common RL methods unstable.

4 Methodology

4.1 Model Architecture

The input for our basic DQN model is a 128x128 image of the game, which we convert to gray HSV scale. We send this image to our CNN, which consists of three convolutional layers, a fully connected layer, and two output neurons corresponding to jump or no jump.

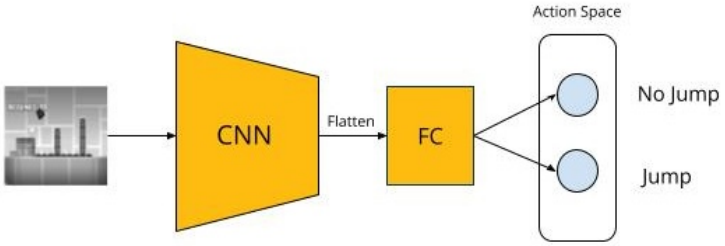

Figure 2: Basic DQN Model

We also implemented a Double DQN (DDQN), which has the same architecture as the DQN; the main difference is when calculating the loss function. It has been shown that DQN will often overestimate the Q-value, so to counteract this, when computing the max term in the equation, we decouple the choosing of the action from the evaluation of the action.

$$\mathcal{L} = (\mathcal{R}_{t+1} + \gamma q_{\bar{\theta}}(S_{t+1}, \operatorname*{argmax}_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2$$

Finally, we imlemented a "Mixture-of-Experts" agent, where the model decides which classification head to route into based on whether its in the ship or cube state. This is motivated by the fundamental differences in play styles for each mode and the difficulty to learn both at the same time. During training, we give the model the correct state directly from the mod so it can route into the correct head module. Simultaneously, we train a classifier to predict the current state. Together, our loss term is as follows:

$$\mathcal{L} = (1 - \lambda) \cdot \text{MSELoss}(Q_{\text{pred}}, Q_{\text{target}}) + \lambda \cdot \text{BCELoss}(\hat{y}, y)$$

λ: weight for classification loss

Q_{pred}: predicted Q-values

Q_{target}: target Q-values

$\hat{y}$: predicted ship labels

y: ground truth ship labels

We found that setting $\lambda = 0.1$ was a good balance of classification and DQN objective, achieving about 98% accuracy.

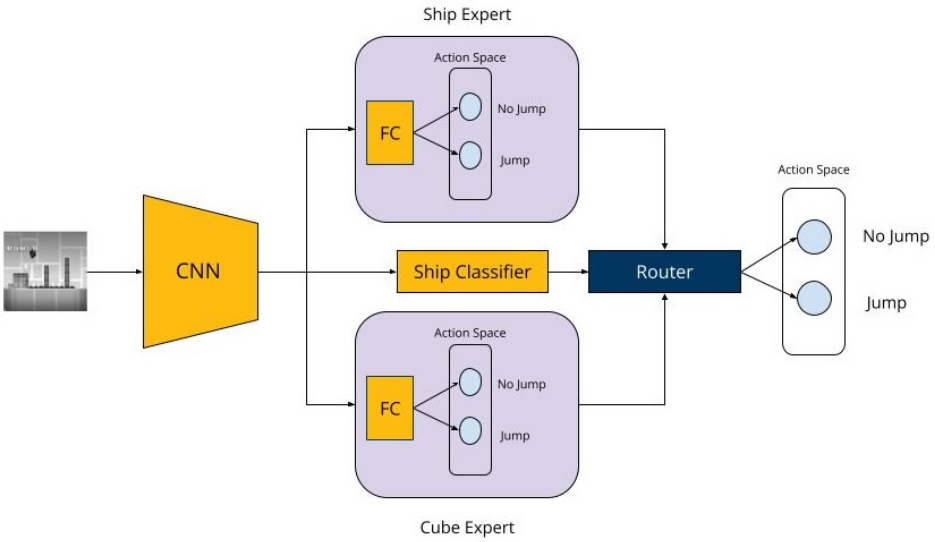

Figure 3: Expert DQN Model

4.2 Modding Geometry Dash

Unlike games with existing reinforcement learning environments, such as those built on OpenAI Gymnasium or Unity ML-Agents, Geometry Dash does not offer a built-in interface for training agents. To address this, we developed a custom training environment by directly modding the game client using C++ and the Geode modding framework [3].

Using Geode, we hooked into the game's internal update loop, allowing us to implement our own functions for the training environment. In particular, we developed the following features: 1) "Step" frame by frame through the game, 2) Jump via a command, 3) Load the level from a desired percentage.

We also capture the screen frame directly from OpenGL, sending every fifth frame to our AI model through a local TCP port. We tested varying frequencies for sending frames, including sending every frame—sending every five frames was empirically the best balance for speed and efficiency.

From the Python side, we send actions to the mod through a different TCP socket. We have two different commands: "step" and "step hold," corresponding to staying still and jumping. Together, these modifications provide the full environment for training our agents with RL.

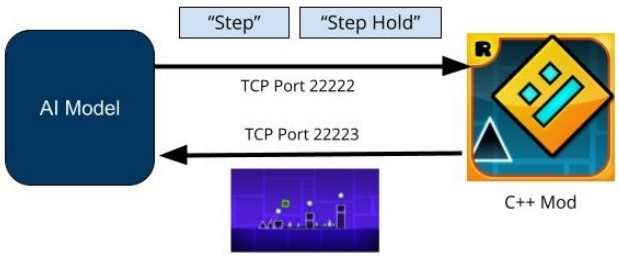

Figure 4: TCP Data Pipeline

5 Experiments

We tested a wide variety of models and present a select few that show promise. In addition to the differing models discussed above, such as DQN, DDQN, and Dueling DQN, we implemented NoisyNets, New-State randomness, N-Step Learning, and differing architectures of the DDQN, increasing and decreasing model complexity.

NoisyNets [2] are a type of linear feed-forward layer that adds a randomness term to each weight and bias, multiplied by a learnable model parameter. This addition has been shown to improve many models trained on Atari games, as it allows the model itself to control how much randomness it should have for a given state, while the default ε-randomness does not.

$$y = (\mu^w + \sigma^w \cdot \varepsilon^w) + \mu^b + \sigma^b \cdot \varepsilon^b$$

The NoisyNet formula for the linear layer is above, μ^w and μ^b are the regular weight and bias. We then add noise, ε^w and ε^b, multiplied by learnable parameters σ^w and σ^b.

New-State randomness is a change to how the agents take a random action; instead of using ε-randomness, which tapers off from a hyperparameter, we add randomness based on how often the model has reached the state. For our specific environment, we define a state to be a truncated percentage, as having a state be a different image fed into the network would lead to many "different" states that only differ by a few insignificant pixels.

N-Step Learning is a slight tweak to the DDQN's loss function, where we unravel the expression for future reward based on the model's prediction of the future state N steps into the future. The idea behind this addition is to attempt to make the reward landscape less sparse for the model by propagating out rewards over more training steps.

$$\mathcal{Z}_{t+1} = S_{t+1}, \operatorname*{argmax}_{a'} q_\theta(S_{t+1,a'})$$

$$\mathcal{L} = (\mathcal{R}_{t+1} + \gamma\mathcal{R}_{t+2} + \gamma^2\mathcal{R}_{t+3} + \cdots + \gamma^n q_{\bar{\theta}}(\mathcal{Z}_{n+1}) - q_\theta(S_t, A_t))^2$$

The main architectural tweaks made to the DDQN were experimenting with the capacity of the model. With the default being the standard Atari architecture, we tried adding more filters and layers to the convolution block and residual layers, adding extra linear layers to the fully connected layer, and reducing model capacity in total.

The modification to the convolution block was the addition of 16 extra filters in the beginning convolution and an extra hidden convolution layer of 64 filters to 64 filters with a kernel size of 4 and a stride of 2.

To reduce model capacity in total, we halved every value in the standard Atari architecture.

In addition to changes to the capacity, we also implemented an architecture that takes in additional input to the fully connected layer of the model's previous actions. The motivation for this change was an observation that during the ship section, to fly straight, the model would need knowledge of the previous input.

We also tested a wide variety of reward functions and reward parameters. We experimented with continuous reward functions, discrete reward functions, New-State reward functions, and reward clipping. The number of reward parameters tested was numerous, and we will only provide a select few configurations in the results section. The continuous reward functions were a small reward given to the model at each time step. We experimented with a constant reward, a linear reward, and a squared reward. We also scaled the reward with respect to the death reward. The percentage that the model has crossed is given as input to the linear and squared rewards. The idea behind the non-constant rewards was to encourage the model to make progress in the level, as the rewards were much greater later in the level compared to the beginning of the level, due to the percentage difference.

The discrete reward function was a simple reward given to the model every 3 percent. Though extremely simple, this proved to be the most effective.

The New-State reward function was a reward given to the model every time it reached a state; however, the more often it had visited this state, the less the reward would be. We specifically taper off this reward proportionally to the inverse square root of the number of times the state has been visited. Just like the New-State randomness, we define a state to be a truncated percentage. We also added reward clipping, where we limited the reward that is saved in the model's memory to some range (a,b).

6 Results & Analysis

6.1 Best Results

Our best models, trained on "Stereo Madness" and "Back On Track", are capable of beating the entire levels. These models, specifically the cube models, took about 5000 and 2500 attempts, respectively. To train the "Back On Track" model, we took model weights from the "Stereo Madness" model and trained from there, hence the significant decrease in attempts.

This may seem bad for generalization, but "Back On Track" has many different elements than "Stereo Madness" that significantly affect the model, such as jump pads, new structures, and a longer run time for the cube section, thus, the fact that the weight distillation was able to bring training down by a factor of two shows great potential.

Our most successful model is a mix of two separate models trained on the cube and ship. The cube model consists of the extra convolution filters architecture, while the ship has the same architecture with extra input for the previous action. These were trained with the following hyperparameters in Appendix A.

Notably, all rewards were scaled down between -1 and 1, which led to increased stability while training, in addition to massive performance improvements.

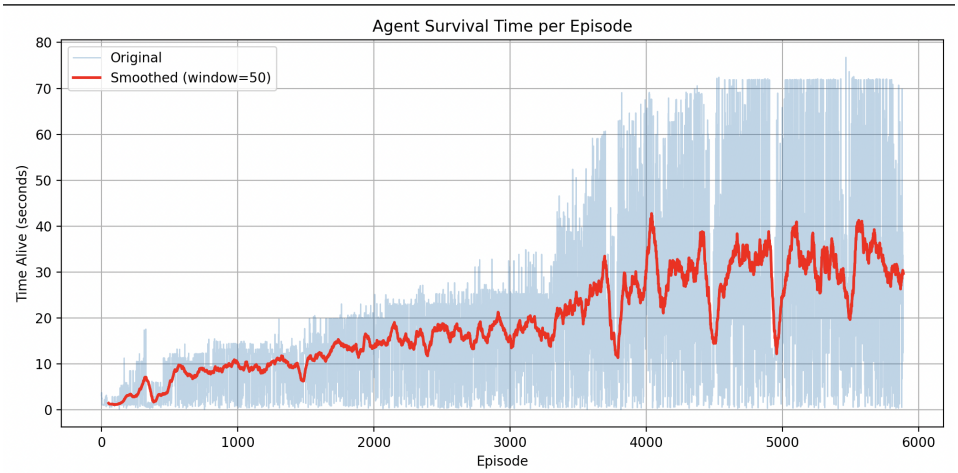

Figure 5: Cube Model on Stereo Madness

Above is a graph of the cube model trained on Stereo Madness. After about 4500 attempts, the model beats the level quite often. Saving the weights of such models yields a model that regularly beats the level. However, we have observed that the win rate from these models is not 100%, and they often make seemingly random mistakes near the end that ruin the attempt. We think this is due to potentially inconsistent frame rates during testing and training from the TCP server, leading to random triggers that the model misinterprets and makes one small mistake. However, in Geometry Dash, one small mistake is life or death.

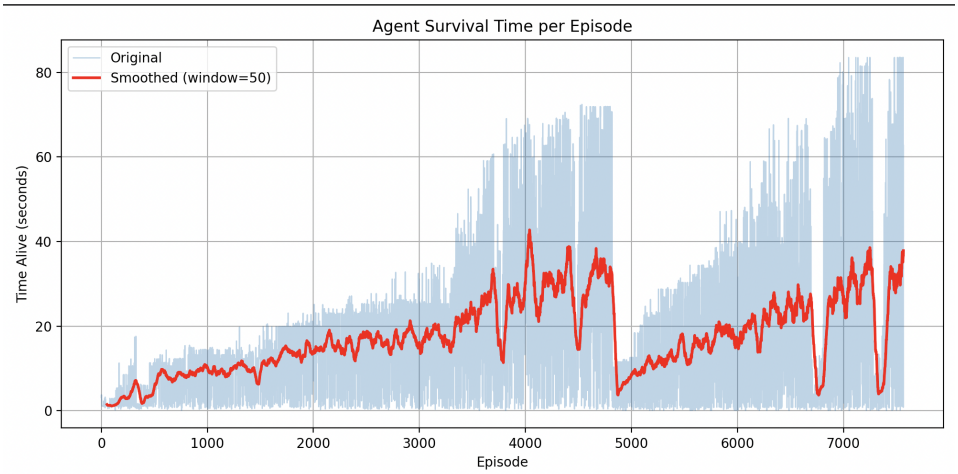

Figure 6: Cube Model on Back on Track

Above is a graph of the cube model trained on Back On Track. This model has imported the model weights of a perfect cube model from Stereo Madness at attempt 4900; everything past that attempt is the model training on Back On Track. After attempt 7000, the model beats the level quite often. The two massive dips in the graph at the end are the GPU throttling and do not reflect the model's instability, although the training process is quite unstable.

6.2 Failed Results

Interestingly, many of the "improvements" we added in the hope of improving model performance led to drastic decreases in results. We present a select few insightful failed experimental results.

Continuous Rewards

One of the most intuitive things to do was to add an increasing reward for time alive, encouraging the model to venture further and complete the level. However, this led to massive performance losses. Although stability of the model increased, it never seemed to get past 15% on Stereo Madness. This was in comparison to the other best model at the time, which got to nearly 70–80% at its peak and an average of 40%. We added two main time-continous rewards, a linear reward, and an exponentail reward, we played around with the constants and hyperparameters of each reward, but it never got past 15%.

The current positive reward given to the model is a discrete reward given every 3%, although this discrete reward resulted in instability compared to the time-continous reward, the model performed better. Our hypothesis for this counterintuitive result is that the discrete-time reward gives the model freedom between the 3% to explore the action space, while the continuous-time reward encourages the model to stick to its potentially flawed weights, as deviation often leads to an early death, which is a massive penalty considering the reward it would have had it had simply stayed alive, even it was headed for certain death.

New-State rewards

Another intuitive addition to the reward function was to add a reward based on how new the state was to the model, once again in hopes that it encourages exploration, and in Geometry Dash's case, since we make the states the truncated percentage, it would always encourage getting further in the level. However, this also performed horribly.

Our hypothesis as to why this occured is that the varying reward confused the model, as since we are using a DQN architecture that predicts the expected reward of a state, this state's true reward changes, and the inflated reward that the model attempts to predict is no longer present, leading to unstable gradients.

NoisyNets

One of the classic implementations of Rainbow DQN [4] is the addition of NoisyNets. As discussed eailer in the experiments section, it allows the model to control how much randomness it gets. However, as it kept training, the model seemed to "want" randomness, and the gradient of the σ^w and σ^b matrices kept increasing, and soon randomness completely dominated the model's actions, even if we initally set σ^w and σ^b to 0! It never got past 10%, in fact, performance decreased the longer it trained.

Our hypothesis for why this occured, is that the noise added to the model was too complex for the model to handle. At the beginning, when the model anyways needs more randomness to explore, the weights of the random noise increase; however, the noise added to the final fully connected layers is not very helpful, as since the action space is only two-dimensional and the noise added is complete gibberish. In comparison, the ε-greedy noise is simply and allows for the model to easily grasp what to acconplish.

Dueling

Another one of the classic implementations of Rainbow DQN [4], the Dueling architecture decoupes the prediciton of the action from the value of the state, however, while Dueling DQN does well in other scanrios, performance was poor in Geometry Dash.

Our hypothesis as to why is that since there are only two actions for Geometry Dash, the separation was not very useful and only added extra parameters in the network to train.

7 Conclusion

There are many ways to expand upon this project. The simplest improvement would be to try training on more levels, including base game levels and custom levels created by community members. Training on more levels can expose the model to more obstacles and structures, potentially allowing it to both learn more patterns and generalize better to any given level. However, doing this can be very

time-consuming for each level, and it may be challenging to find out which levels help the models generalize best.

Another improvement is to get the model to know exactly where the player is in the image we feed. With our implementations, the model is not aware of the exact position of the player, and this can cause issues with training and deciding on actions. If the model can figure out the player's position, it can be more precise and accurate with its outputs, leading to better performance in general.

One of the major time sinks of this project was the training time. Since the game runs slower than normal speed and it takes thousands of attempts to train properly, we often had to leave models training overnight to see any results. It is possible to speed up the game using the Geode modding framework, but any attempts we made to do so did not integrate with our training setup. If we could train properly while speeding up the game, it would considerably reduce training time, allowing models and levels to be tested more easily and more often.

We can also improve by sending each frame of gameplay to the model rather than every five frames. Currently, if we try doing this, we experience major performance drops, which we theorize is due to issues with sending images through TCP. If we can send every frame without drops in game speed, the model will have a lot more game states to work with and will potentially perform better.

In terms of game mechanics, Geometry Dash has six more game modes than just the cube and the ship. Many of the later levels in the game also use most, if not all, of these game modes in a variety of ways. We can follow a similar Mixture-Of-Experts style approach where we have an expert for each of the eight game modes and a gate that decides which expert to use. To train these experts we can follow the same procedure as the cube and ship: train on the parts of levels with that gamemode, or create a custom level to train on. Implementing and training each of these experts will allow for a lot more levels to be accessible.

8 Broader Impact

Our work demonstrates that deep reinforcement learning can be successfully applied to complex, real-time environments like Geometry Dash, which pose significant challenges due to sparse rewards and precise, frame-level control. Unlike previous approaches that relied on handcrafted inputs, we trained fully vision-based agents using modern RL techniques such as DQN, Double DQN, Mixture of Experts, NoisyNets, and N-step learning. By building a custom modded environment with full integration into the game client, we created a modular, open-source environment that allows future research and experimentation with other models. This contributes a novel benchmark for visual RL and highlights the best strategies for exploration, reward shaping and temporal reasoning in fast-paced, auto scrolling games.

9 Contribution Statement

Dylan:
Built the interface between our AI agent and the game, sending video frames and receiving commands via TCP. Set up the end-to-end training loop, implementing the DQN agent and the custom gymnasium environment. Trained and tested different hyperparameters.

Ishan:
Used the Geode modding framework to mod Geometry Dash and provide all the necessary functions needed for training. This includes: freezing and unfreezing level movement, jumping, "stepping" frames of gameplay, and loading from a specified point of the level.

Vigyan:
Implemented NoisyNets, N-Step Learning, all variants of the reward function, alternate DDQN architectures, implemented Mixture-Of-Experts, fixed bugs in the mod, extensive experimentation and hyperparameter tuning, trained many expert models.

Surya:
Implemented the dueling DQN framework in a custom gymnasium environment. Implemented Gated Mixture-Of-Experts network to help improve performance on ship and cube portions. Trained both these networks.

Om:
Trained initial networks. Constructed & analyzed network visuals. Experimented with DQN networks, helping train ship experts.

References

[1] CodeNoodles. I made an ai play geometry dash. https://www.youtube.com/watch?v=6cksxUwmrTw, 2023. YouTube, uploaded by CodeNoodles.

[2] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2019. URL https://arxiv.org/abs/1706.10295.

[3] Geode. Geode modding framework for geometry dash. https://github.com/geode-sdk/geode, 2023. GitHub repository.

[4] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017. URL https://arxiv.org/abs/1710.02298.

[5] Youngwook Kim and Bryan Williams. Game level generation using generative adversarial networks. https://cs231n.stanford.edu/reports/2017/pdfs/605.pdf, 2017. Stanford CS231n Course Project Report.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[7] SynvexAI. Gdind: Geometry dash reinforcement learning. https://github.com/SynvexAI/GDind, 2024. GitHub repository.

Appendix

A Various Model Graphs

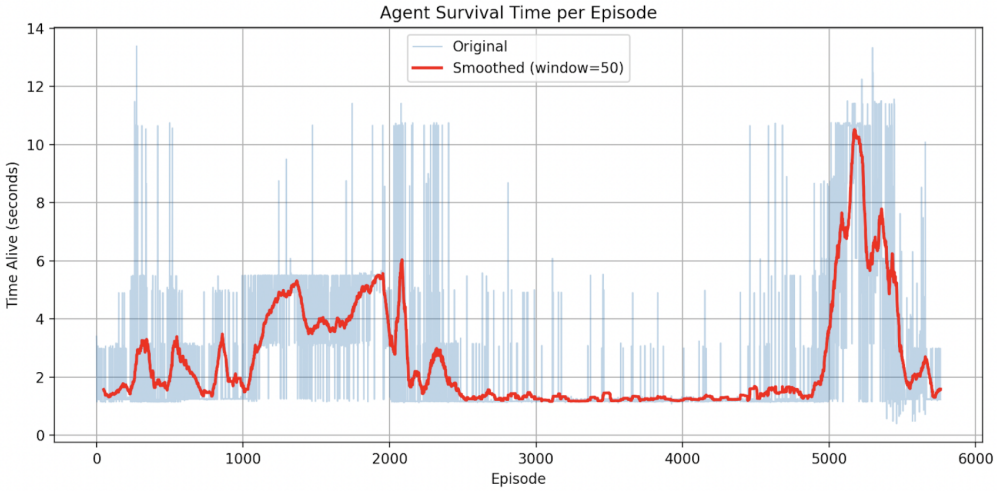

Figure 7: Continuous Time v1

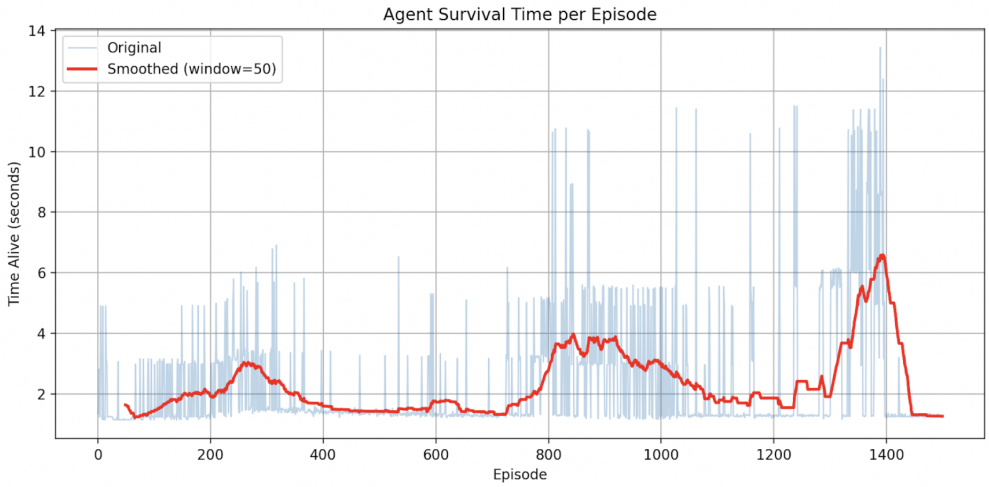

Figure 8: Continuous Time v2

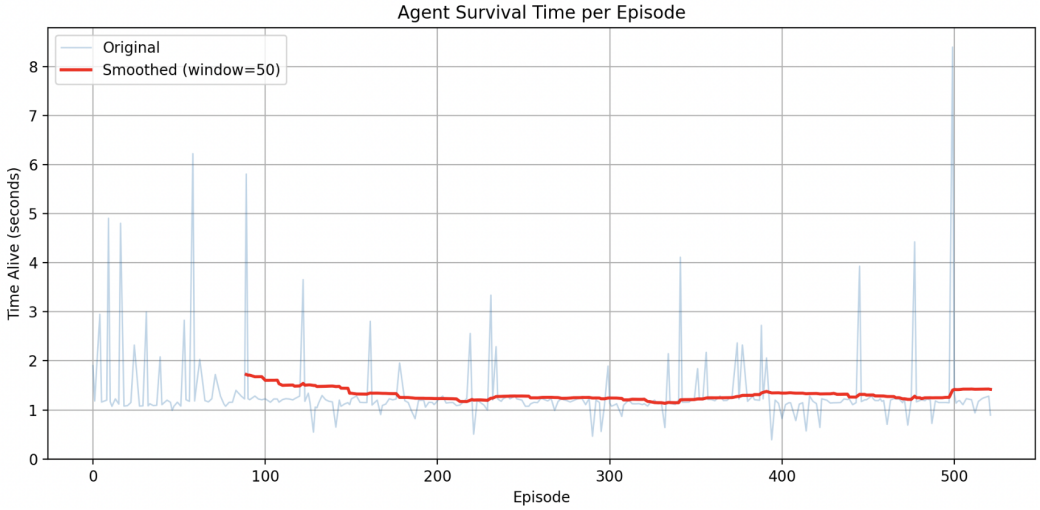

Figure 9: NoisyNet

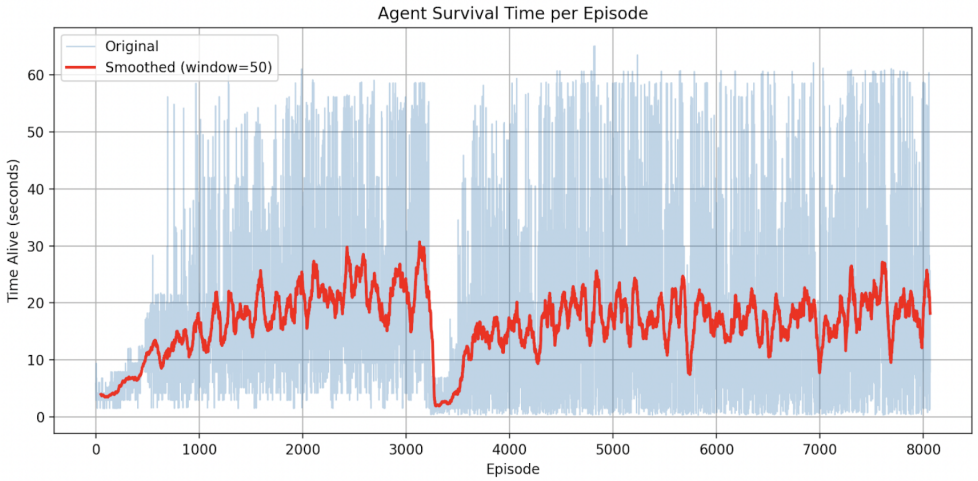

Figure 10: General Ship

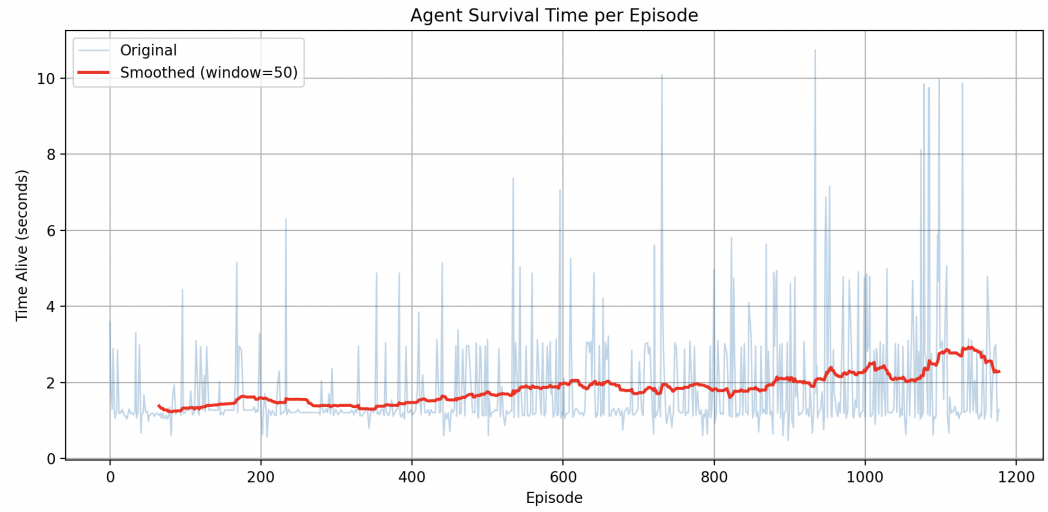

Figure 11: Small DQN Architecutre

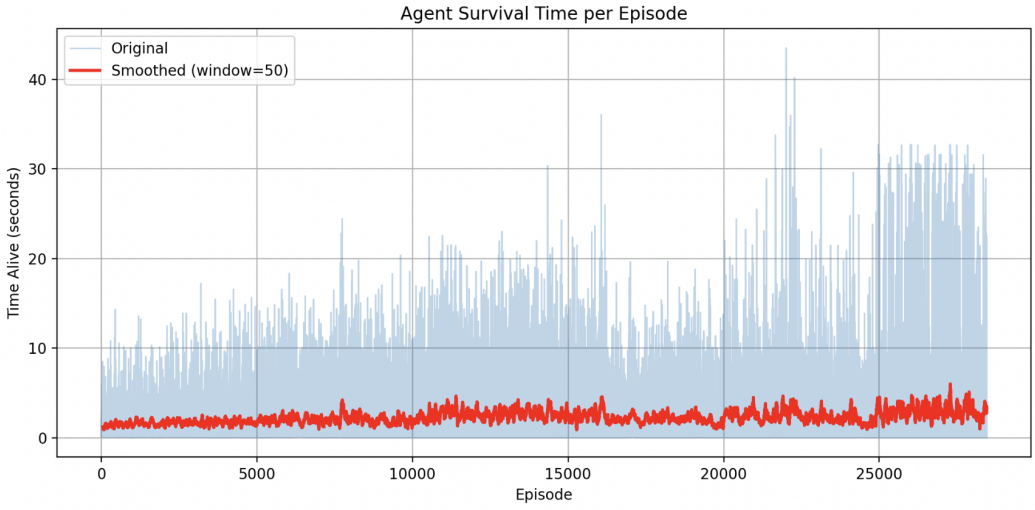

Figure 12: Dueling DQN