Breakout AI w/ Deep Q-Learning

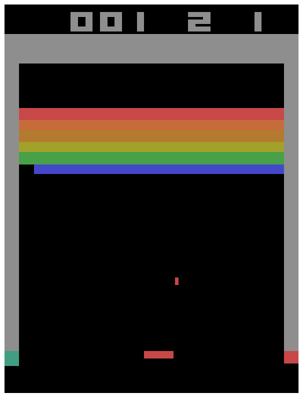
Group 8:
Jovan Ko, Samuel Huang, Joanna Doan, Smriti Davey, Lisa Verma

I. Introduction

Overview

Artificial intelligence, or AI, has opened up exciting opportunities to push the boundaries of machine learning, especially in a domain such as video games. This intersection between AI and the world of gaming has become significant as it presents a challenge to AI and its capabilities, pushing it towards achieving human-like thinking and decision-making. Video games, far beyond their role as sources of entertainment, have evolved into powerful tools for advancing society and simulating human cognitive abilities. They essentially serve as grounds for testing AI algorithms, evaluating their adaptability and strategic thinking closely resembling the way humans think.

In our project, we are utilizing the iconic video game, Breakout, an Atari game introduced in 1976. While it might appear straightforward and simple– maneuvering a paddle to hit a ball against a wall of bricks– the game's simplicity masks its complex and strategic nature. In Breakout, a lot of spatial awareness, planning, strategic moves, and precision is necessary and are qualities we seek in our AI agent.



Breakout, video game by Atari (1976)

Objectives

1. **Exploring the power of Deep Q-Networks (DQN):**

We will delve into DQN (Deep Q-Network), an advanced reinforcement learning algorithm introduced in 2015 by DeepMind. Reinforcement learning (RL) serves as the framework where agents will master the skill of decision-making within an environment to optimize rewards.

Within RL, there exists two central components: the environment, which is the challenge, and the AI agent, which embodies the learning algorithm. The agent actively engages with its environment, where each action yields a reward contingent on its choices, all while simultaneously observing its own decision-making processes. The primary objective is the enhancement of the agent's performance within the game and driven by its motivation for self-improvement and improve its precision to receive its rewards.

2. **AI agent's performance:**
   During our extensive training sessions, we will undertake a comprehensive analysis of the agent's progress, testing how well it adapts and its decision-making capabilities. This analysis will provide us a deeper understanding of the AI's true problem-solving abilities within its environment.

Motivation

Our project is driven by a strong desire to test the capabilities of Deep Q-Networks and reinforcement learning algorithms. It highlights the remarkable abilities of artificial intelligence, not only in performing complex tasks but also in showcasing its excellence within unpredictable and complex real-world situations. Breakout is proven to be an ideal testing ground, providing us an environment to test the agent's abilities and decision-making processes. By training the AI agent to excel at Breakout, we demonstrate AI's adaptability and ability to tackle challenges.

II. Background

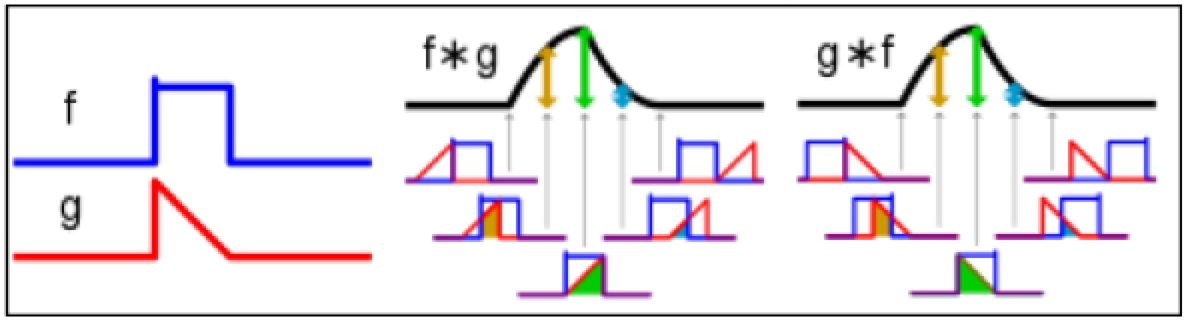
Deepmind's 2013 paper, "Playing Atari with Deep Reinforcement Learning", was a breakthrough for Reinforcement Learning (RL), being the first instance of combining the Reinforcement Learning technique of Q-Learning with Convolutional Neural Networks, a technique Deepmind has named "Deep-Q Learning." Deepmind was able to train a single model with this technique to play a number of video games for the Atari 2600 in an emulated environment, and was able to match and even outperform human players in some games. For the scope of this paper, we attempt to implement the same Deep-Q Learning technique to replicate Deepmind's results specifically for the game Breakout.

Breakout

Breakout was a popular game for the Atari 2600 that gave players the goal of destroying a wall of bricks by bouncing a ball off a moving paddle. At the start of the game, a ball is spawned in the middle of the screen, heading towards the player. The player is able to control the paddle at the bottom of the screen with the left and right directional pad buttons, and bounce the ball back up towards the breakable brick wall at the top of the screen. If the ball misses the paddle and goes past the player, the player loses a life, and is able to spawn a new ball with the red controller button. This game, and the Atari system, has since been adapted to an emulated environment API suitable for a RL model to interface with.

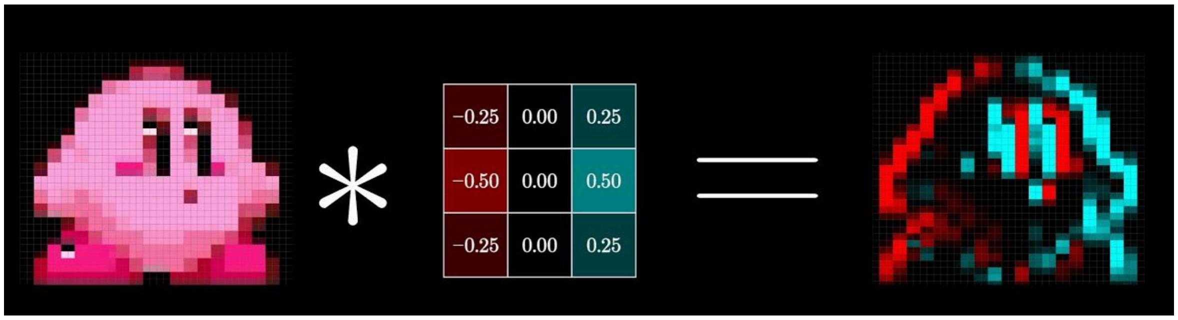
Convolutional Neural Networks

To parse the image data from each frame of the Atari environment, the model must be able to extract various important features from the frame, such as the player paddle position, the ball location, the amount of unbroken bricks, etc. A common method to extract features from image data in machine learning uses Convolutional Neural Networks. In functional analysis, a Convolution of two functions f and g is an operation that takes the function g and "drags" it across the function f, and outputs a new function, $f * g$ that plots the overlapping area of the two functions as it was "dragged" across.



Convolution of two functions

When applied to discrete data points, such as pixel intensities across an image, we find that choosing an appropriate "kernel" function g to "drag" across the image f is able to extract features from the image, such as edges of shapes.



Convolution of a "kernel" function over an image to extract edge information

A Neural Network with a layer of various different kernels can then implicitly learn the significance of these different features, and be trained to generate different outputs based on the image and its learning objectives. Deepmind's Atari model (and ours) uses a Convolutional Neural Network (CNN) to parse each frame image of the Breakout game.

Q-Learning

To unpack Deep-Q Learning, we must first understand Q-Learning. In Reinforcement Learning, the environment must be interpreted by the RL agent as a state space modeled as a Markov Decision Process (MDP), which is a probabilistic map of all the different states an agent can reach with various actions at every state, with a defined Reward Function for every transition from one state to the next. Reinforcement Learning aims to learn a Policy Function that can estimate which action to take to maximize this reward. The Reward function is typically not defined for every possible state, so to search this space for rewards, we typically define a Value Function that assigns an expected reward to each state, and the Policy Function is then developed to find an action that takes the agent to the state with the highest Value function.

Q-Learning, then, combines the Policy Function with the Value Function into a Quality Function that determines an expected reward for each state-action pair. In "shallow" Q-Learning, this is typically done by calculating backwards from a state that gives a reward, and by defining a "discount factor" $\gamma < 1$ to discount this future reward in favor of possibly better immediate rewards, the exact expected reward for each action at each state (state-action pair) can be calculated. The calculated Q-value is then stored in a Q-table, a look-up table for the Q-function. Defining this Q-table over the entire state-action space, the agent can then simply choose which state-action pair has the highest Q-value to continue. This Q-table can also be updated as the agent explores more of the state space. This removes the need for a Policy Function that must model the entire environment and predict future rewards to determine the best probable action, giving us an “Off Policy” or “Model Free” algorithm.

Deep-Q Learning

A limitation of “shallow” Q-Learning is the need to calculate a Q-value for every state-action pair in the state-space, which becomes difficult for games with larger state-spaces, and impossible for games with near-continuous state-spaces, like Breakout. Deep-Q Learning takes Q-Learning a step further by using a Neural Network to approximate the Q-function, removing the need for a large Q-table. In the case of Deepmind’s Atari model and our project, the CNN used to parse the frame data is trained to approximate our Q-function. By calculating a Loss Function after a certain number of frames, which is simply the discrepancy between expected rewards from the Q-function and the actual reward received, the Q-function can be continuously updated over our training period.

In training our Deep-Q Network (DQN), there are 3 main parameters that affect how the Q-function is updated: The discount factor γ, the learning rate α, and the exploration rate ϵ. The discount factor γ, as explained above, gives a reasonable discount to future rewards to incentivise the model to prioritize actions with immediate rewards. We will not adjust this parameter in our training, leaving it at a value of 0.99. The learning rate α determines how strongly the weights of the Q-function CNN will be affected by the current calculated loss function, at the risk of overwriting previously learned information.

The discount value and learning rate are factors in updating our Q function, which follow a modified version of the Bellman equation:

$$Q_{new}(s, a) = Q_{old}(s, a) + \alpha\left[R(s, a) + \gamma \cdot max(Q'(s', a')) - Q_{old}(s, a)\right]$$

where (s, a) is the state-action pair to be updated, $R(s, a)$ is the given reward from that state, and $max(Q'(s', a'))$ is the maximum expected future reward from the current Q-function. In the DQN, this is calculated implicitly with gradient descent over all the weights of the neural network.

The third parameter is the exploration rate ϵ, which is a factor in the technique of Epsilon-Greedy Exploration to determine the exploration/exploitation tradeoff of attempting unexplored state-actions or to continue down a known path. The exploration rate ϵ determines the percentage of “exploration” actions taken that will be completely random. In our DQN, ϵ follows a linear exploration schedule, starting at 1.0, or 100% random actions at the start, and linearly decreasing to a specified percentage, around $1\sim5\%$, over the span of a certain fraction of the total training run, after which ϵ remains at its final percentage for the rest of the training run.

III. Methodology

We spent a large portion of our project trying different implementations and examples, which all had issues with quickly outdated dependencies and no longer maintained APIs. We eventually managed to find a few APIs that worked for our project, using the Gymnasium environment from Faruma, the DQN class from StableBaselines3, and Weights and Biases for logging training results.

Game Environment

For the game environment, we used the Gymnasium environment API currently maintained by Faruma. Gymnasium is the currently maintained branch of OpenAi's Gym environment, previously OpenAi's database of environments to benchmark new RL algorithms. The Gymnasium API runs an emulator of Atari 2600 to play the original Breakout ROM file. It takes in frame-by-frame image data as the observation space, and provides a list of possible actions to take for the learning model to interface with. It also provides a render method that allows us to display the model's gameplay to visually evaluate its performance.

The reward function is not explicitly documented in Gymnasium's documentation, but it seems to be based off of internal game states that provide the current score of the game. The documentation links to the original Atari 2600 game manuals with regards to rewards, which explains the different scores given from breaking different colored bricks. If our assumption is correct, the Gymnasium environment takes this current score variable from the internal game state and directly outputs it as the reward function at the current state. Stable Baselines 3, the API we use for our learning model, gives a reward function in its environment wrapper that bins rewards to +1, 0, or -1 by its sign. We assume that means that the reward is +1 for each brick it hits, -1 for losing a life, and 0 otherwise. Since we are using the Stable Baselines 3 environment wrapper, we assume that this is the reward function taken in by our model. A deep-dive into the source code for these APIs is needed to better understand the how the reward function is implemented

Learning Model

For the learning model, we used the DQN class from Stable Baselines 3, a pre-built Deep Q Network class with a number of model hyperparameters to tune. Stable Baselines 3 (SB3) is a currently updated collection of RL algorithm implementations based on OpenAI Baselines, again, a now defunct baseline benchmark for other RL algorithms to compare. SB3 attempts to maintain a stable implementation able to replicate OpenAi's baseline results. The DQN class allows us to choose different learning policies according to the environment it is acting in. In our case, we use the 'CnnPolicy' to read in the image data from the game environment. In addition to the available parameters for tuning the DQN model itself, SB3 also provides a FrameStack method which allows us to stack four running instances of Breakout within a single frame, allowing the model to train on four games at once, effectively increasing our training efficiency by 4x.

The DQN model follows the same architecture dictated in the Deepmind paper, with an input and output layer and 3 hidden layers:

- **Input layer**: The input layer is of size 84x84x4 to take in the frame-by-frame image data.
- **1st Hidden Layer**: The first hidden layer is one of two convolutional layers, with 16 8x8 filters with stride 4 and rectifier activation.

- **2nd Hidden Layer**: The second convolutional layer has 32 4x4 filters with stride 2, also with rectifier activation.
- **3rd Hidden Layer**: The third hidden layer has 256 fully connected rectifier units
- **Output Layer**: The output layer is simply the 4 possible actions that the agent can take in Breakout: 'LEFT', 'RIGHT', 'NO-OP', and 'FIRE'

The two convolution layers are essential for extracting significant game features from the frame, such as the locations of the paddle, ball, and unbroken bricks. The fully connected layer, we presume, is what encodes the Q-function approximator during training. It is also possible that the convolution layers also encode some information about approximating the Q-function, but that is beyond the scope of this project.

Evaluating Performance

With this basic implementation, we were able to independently train a few models, but we did not know how to properly extract the training logs. These logs contained essential information such as episode lengths, episode rewards, exploration rates, and total timesteps. Thus, to evaluate the model's performance, we wrote a simple script using Pandas and MatPlotLib in Pyplot to extract the data from our output logs and plot the results. Pandas, a data manipulation library in Python, is our data extraction process that extracts the data from multiple log files into a list. Then we use matplotlib for data visualization. The combination of Pandas and Matplotlib allowed us to efficiently extract, structure, and visualize the training statistics from multiple log files. This streamlined our analysis, enabling us to select multiple models to evaluate the performance of our RL agents over time.

With our trained models, however, we were able to simulate a new game, and render the gameplay in a graphical window to qualitatively evaluate the model's performance. Using screen-capture software, we recorded the performance of each manually logged model in a simulated game, and were able to notice what strategies, or lack thereof, the model was attempting.

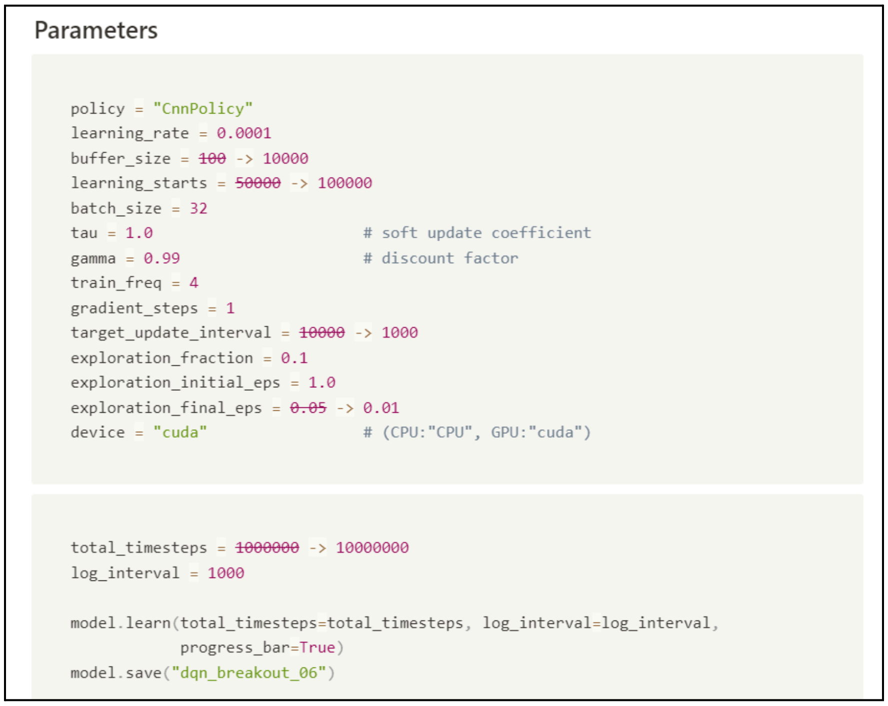
Eventually, we were able to discover that Weights and Biases, an online platform with many tools for machine learning development, provides an API for SB3 that allows us to collect and compile training data in real time, and provides interactive plots comparing each of our runs for us to use. Implementing the Weights and Biases (WandB) API allowed us to streamline our training process greatly. Unfortunately, this was near the end of our project timeline, and we were unable to retroactively fit our previously trained models onto the WandB platform. Instead, we trained one new model based on the parameters given by Deepmind's paper, which ran for 10 million frames and took about 11 hours to train, and a number of smaller models, around 2 million steps/~3 hours training, each experimenting with different hyperparameter changes to compare their results.

IV. Results

After more than 77 hrs of combined training time from all of our different models, the results were quite fascinating.

In our early experiments, our model faced significant challenges and performed poorly. It exhibited difficulties in learning from its past attempts, where it does not have the ability to hit the ball back. During this early part of the experiment, the episode mean reward was averaging around 1.0 to 3.0. It means that during the five lives the model has given, it could only hit the target with a maximum of three bricks most of the time. We can see the same result from training the model with up to 7 hours of training session. Therefore, it became evident that the total timesteps isn't the problem here. After some more experiment tweaking the parameters, we discovered that we had set the buffer size to a relatively low value of 100. This limited buffer size meant that our model lacked access to an enough number of frames from the past experience to learn effectively.

To address this issue, we increase the buffer size from 100 to 10000, allowing our model to store and learn from more extensive history frames. Additionally, we made further adjustments to promote more frequent updates to the model's target network by reducing the target_update_interval from 10,000 to 1,000. Moreover, we start using CUDA cores from NVDIA GPU which give us a performance boost of more than x2.3 compare from using CPU to massively improve our training efficiency. Later, we also made additional changes to increase model stability and convergence by decreasing the exploration final epsilon from 0.05 to 0.01 while increasing the total training time steps from 1M to 10M.


Parameters

```
policy = "CnnPolicy"
learning_rate = 0.0001
buffer_size = ~~100~~ -> 10000
learning_starts = ~~50000~~ -> 100000
batch_size = 32
tau = 1.0                       # soft update coefficient
gamma = 0.99                    # discount factor
train_freq = 4
gradient_steps = 1
target_update_interval = ~~10000~~ -> 1000
exploration_fraction = 0.1
exploration_initial_eps = 1.0
exploration_final_eps = ~~0.05~~ -> 0.01
device = "cuda"                 # (CPU:"CPU", GPU:"cuda")
```

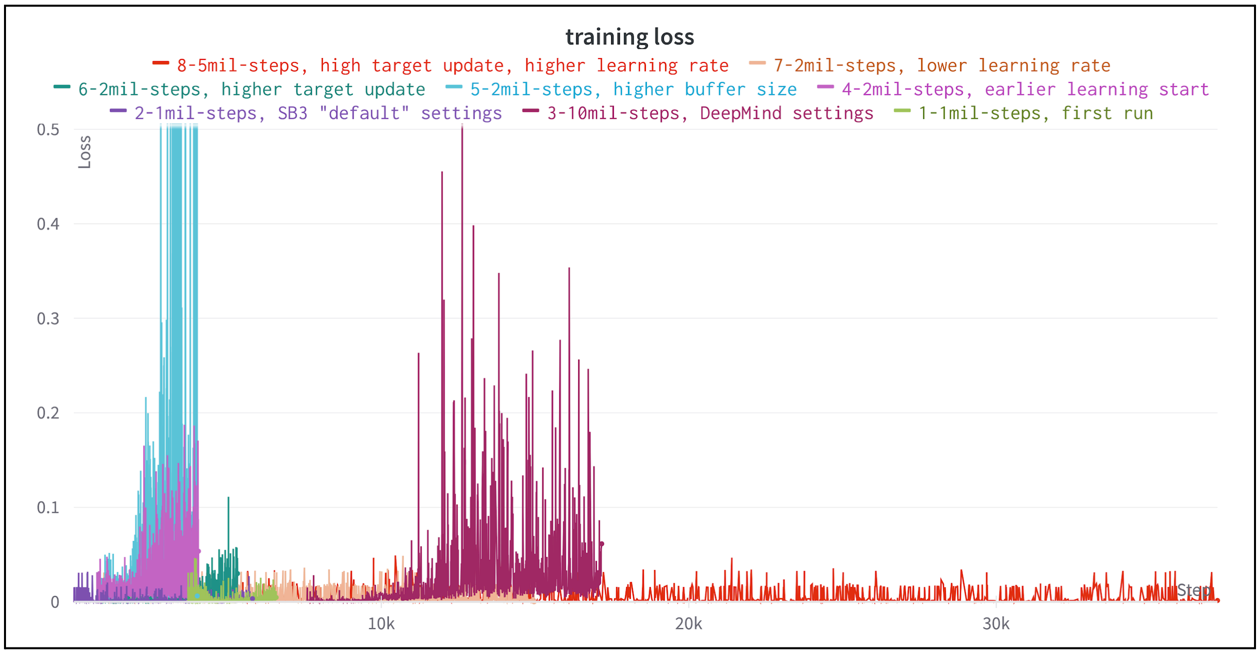
```
total_timesteps = ~~1000000~~ -> 10000000
log_interval = 1000

model.learn(total_timesteps=total_timesteps, log_interval=log_interval,
            progress_bar=True)
model.save("dqn_breakout_06")
```

Hyper-parameters from Ver.06 model which introduce significant improvement

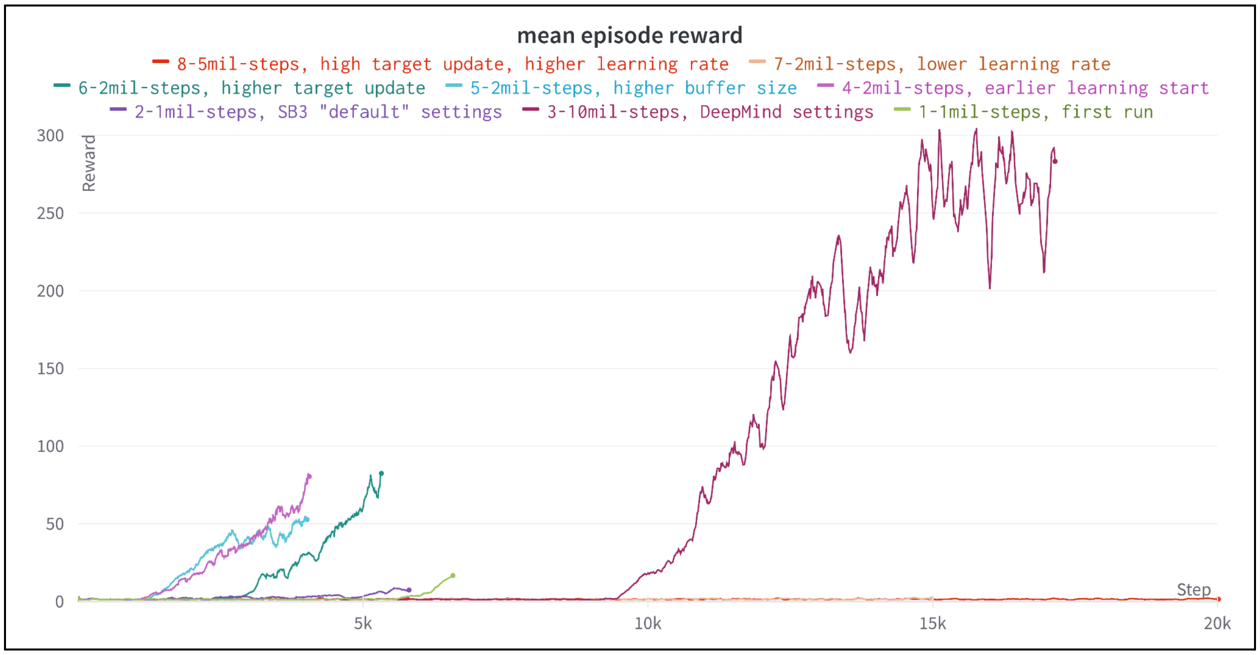
- All the other experimented parameter settings history and result logs can be found here

These changes collectively led to significant improvements in our model's performance. In contrast to our early models, our improved models result in a consistent mean reward of over 300 where our final model has a mean reward of 380. These optimizations and parameter adjustment collectively improved our model performance enabling our AI to actually play the game and very occasionally win the game.

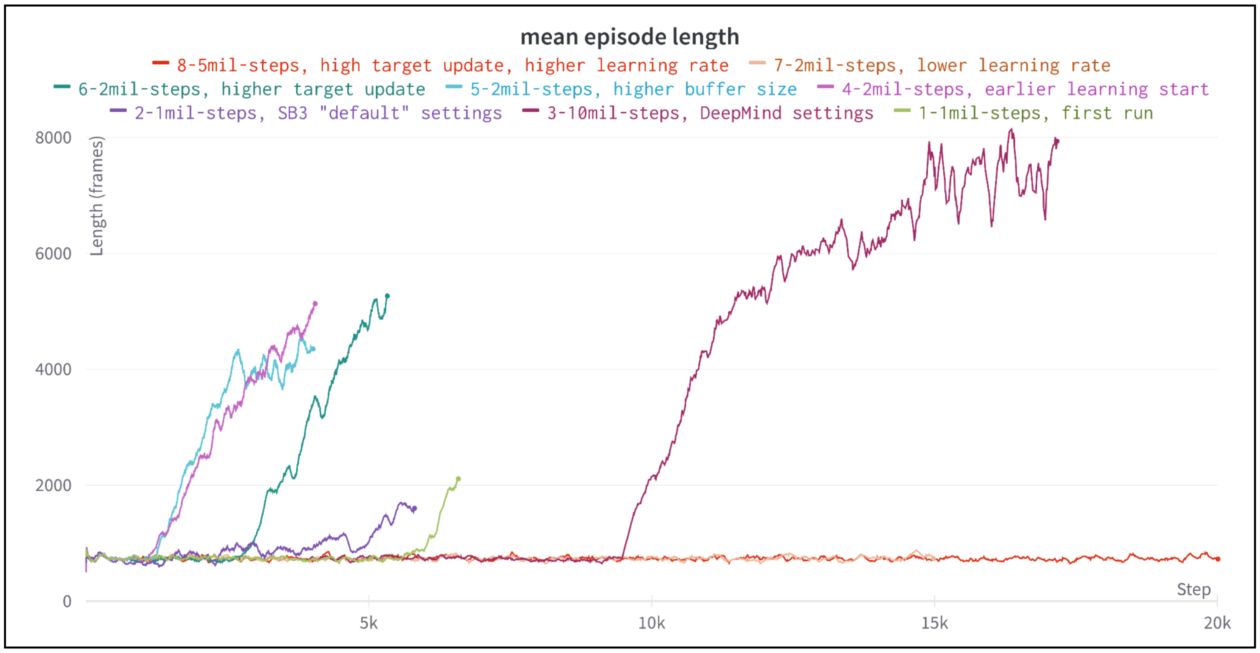
For the last models trained and logged with WandB integration, we trained 8 new models with a select few hyperparameter settings to compare their results. We mainly looked at 3 variables for evaluating performance: Training loss, the log of the loss function at each model update step; mean episode length, the average game length during each training epoch; and mean episode reward, the average received reward per game during each epoch. These results are plotted below:



Compiled training loss from WandB

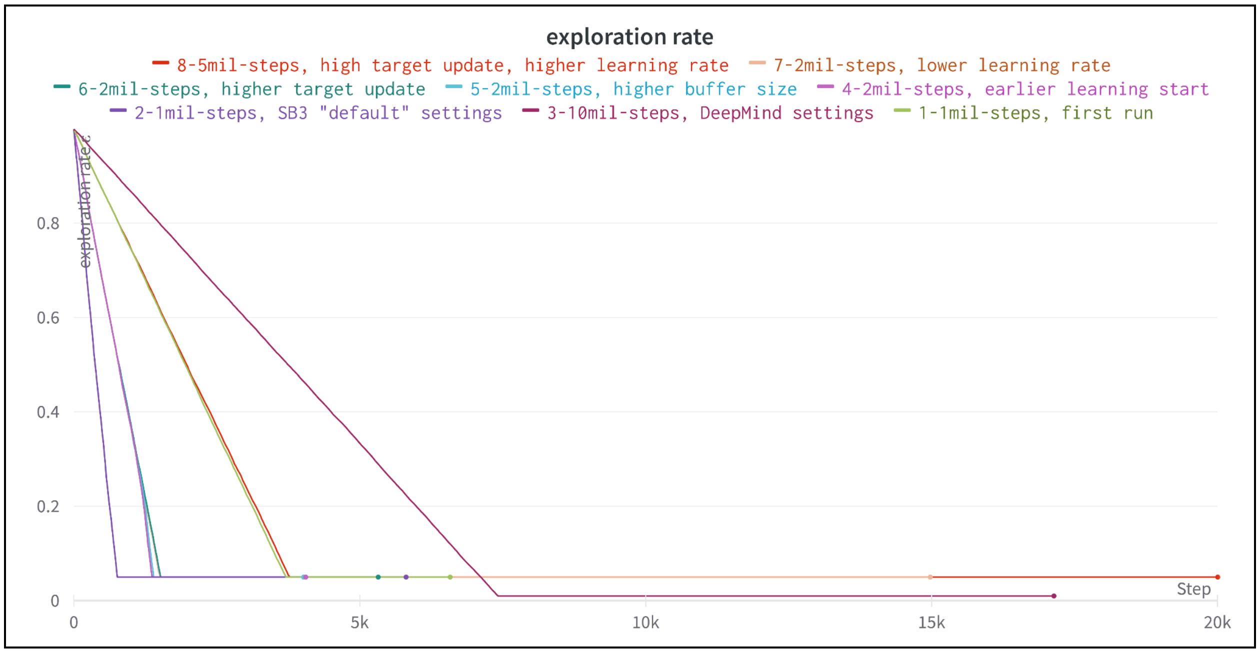


Compiled mean episode reward from WandB



Compiled mean episode length from WandB

However, we also find the plot of exploration rate decrease to be helpful to visualize when the "exploration phase" ends compared to when learning seems to start in the above plots:



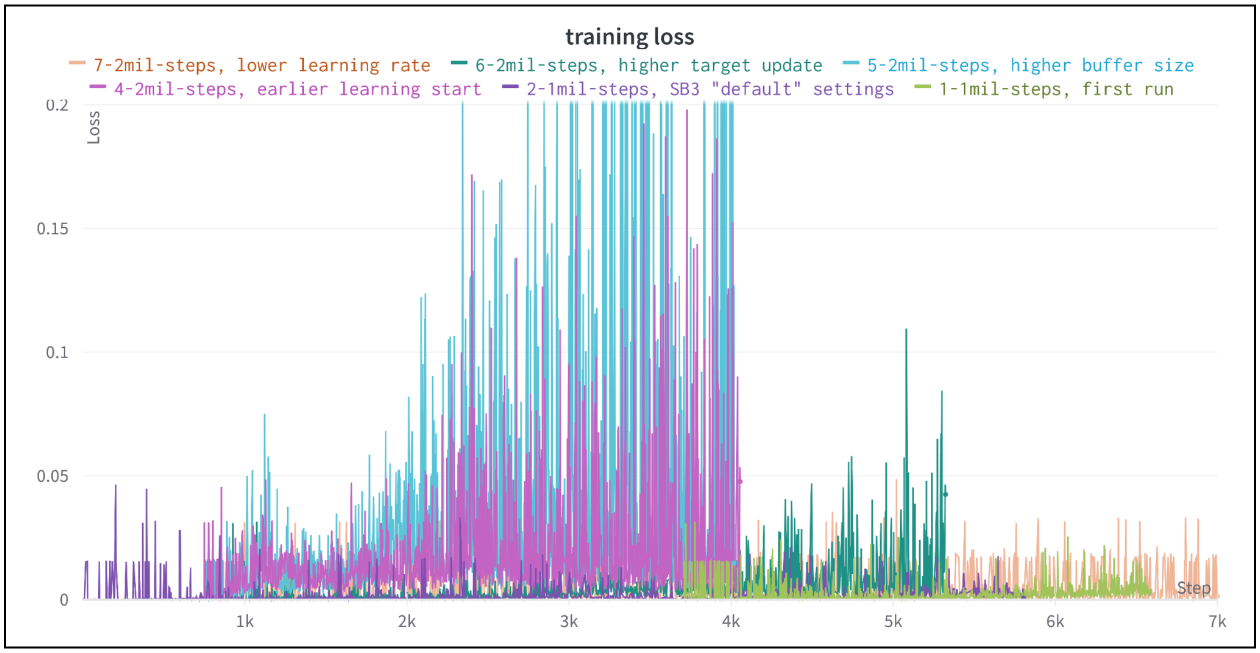
Compiled exploration rate plot from WandB

- *An interactive compilation of our training plots on Weights and Biases can be found [here]*

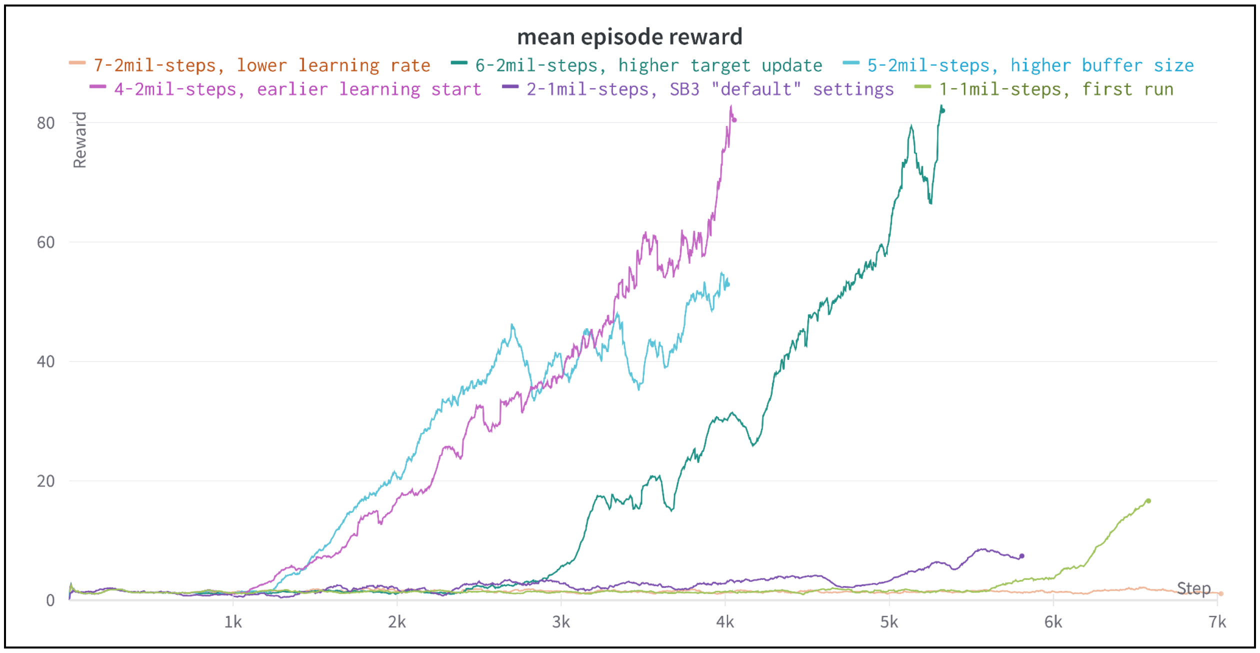
The first two runs used settings close to the default settings given by SB3 to test the WandB functionality, each one training for 1 million steps, and taking around 1 hour to train. This provided a baseline result that we could compare the rest of our runs to.

The 3rd run used hyperparameter settings from Deepmind's paper, with the exception of learning rate, which was not provided by Deepmind and set to 0.00025, and the buffer size, which ran into memory issues when attempting Deepmind's setting. Deepmind used a buffer size of 1 million, we instead opted for 100,000, which our memory was able to handle. This model trained for 10 million steps, and took 11 hours to train. Expectedly, this was a dramatic increase in performance from our baseline runs, and it matches our larger models from our pre-WandB models. However, this model, like our previous models, was still unable to finish the game, and as we can see in the performance plots, that episode length seems to plateau after ~5000 frames, which we assume is indicative of what we consider an "endgame" state, where bricks are more sparsely populated and must be aimed at to finish the level.

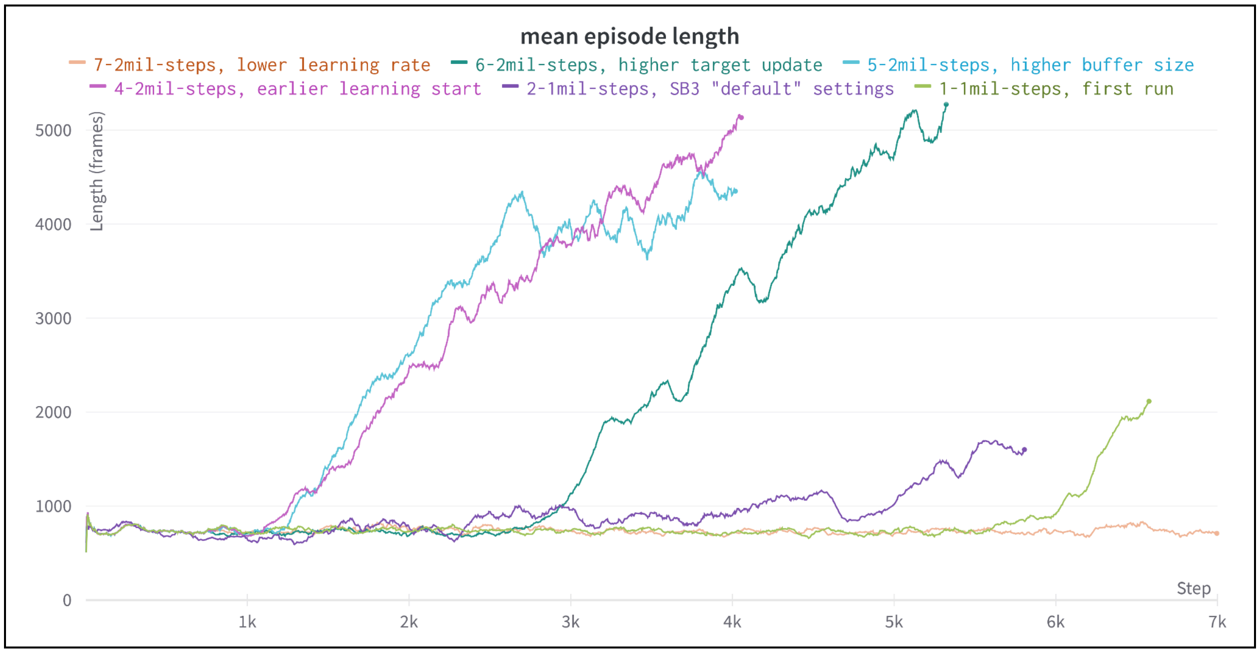
The remaining 5 runs, due to time constraints, were shorter runs of 2 million steps, about 2.5 hours of training each, aimed at noticing changes in training behavior when tweaking a few select hyperparameters. One parameter was changed for each successive run. A closer look at these runs are shown below:



Training loss of smaller tweaking runs



Mean episode reward of smaller tweaking runs



Mean episode length of smaller tweaking runs

An interesting observation we see between these 4 runs is specifically between runs 4 and 5, where we increased buffer size from 1,000 to 100,000 to match our replicated Deepmind model. We see a huge increase in training loss between these two runs, so much so that the plots above are clipped at 0.2 to show necessary detail, while run 5 has a peak loss of 36.244. Despite this huge difference in loss while training, there doesn't seem to be markedly any difference in the performance of the two runs. We do see a slight plateauing effect in run 5 that we don't see in run 4, but it is too small of a difference to make any significant conclusions.

Another interesting observation is the impact of a long "exploration phase" before actually allowing learning to start. Our smaller models had varying learning start times of 1000 to 5000 frames, while the Deepmind model started 1 million frames into the training period. However, once training starts for each of these models, the slope of the performance increase curves do not seem to differ too much, which makes us question whether learning needs to start after exploration.

Finally, a surprising result is that these hyperparameter settings may not be wholly independent of each other, and may need to change with scale. Having an understanding of our small 2-million-step models, we attempted to train a larger model for 5 million steps, taking settings combined from adjustments in run 6, and the opposite of run 7, which did not have good results. The result of this run ended up being completely inadequate, having an effectively flat performance curve across the board.

V. Discussion

Here, we summarize the key findings and insights from our observations of our various experimental training runs:

Hyperparameter Impact:

- Learning Rate: We discovered that the learning rate is crucial to prevent overfitting but must be carefully balanced for effective loss optimization.
- Buffer Size: Larger buffer sizes led to increased training due to a wider sample of frame, yet performed remained satisfactory.
- Exploration Rate: The impact of the exploration rate produced inconclusive results.
- Learning Start Time: We noticed that starting the learning process earlier, before exploration is complete, allows our agent to learn more quickly, challenging the idea of needing to wait for full exploration before starting to learn.

Comparison with Replicated Model:

Our agent, which uses a strategy similarly to the Deepmind model, often can't finish a full game, showing that this strategy has limitations.

Limitations and Future Directions:

In the future, we aim to explore opportunities to enhance our model's performance. With more time, it would be useful to try tuning a few more hyperparameters on larger models, as it would help answer a few questions we have from our current models. For example, if we let learning start before exploration rate fully depletes, could the model maybe learn to use "riskier" strategies, as it takes more random steps later into the game? There are also hyperparameters that we have not touched, such as batch size or discount factor, that could affect performance. With overcoming our late-game plateau-ing behavior, it may be useful to implement a different exploration schedule, maybe a damped sine wave to encourage periods of higher exploration followed by periods of exploiting current strategies. Our replicated Deepmind model starts to destabilize towards the late game, so with a longer training time we wonder if the performance will converge and fully plateau, or completely diverge? Continuous learning, increased data collection, and benchmarking against other models will be essential for further improvements.

Last but not least, we consider implementing an in-game time factor as a contributor to episode rewards will be a good approach in our next attempt. This approach involves reducing the episode reward as the game duration increases, discouraging the AI from using a strategy of continuously hitting the ball into the corner during the late stages of the game. While this idea holds promise for addressing this specific issue, it presents a challenge due to our reliance on pre-built game environments from OpenAI Gym. Building a custom game environment to seamlessly integrate this time-dependent reward mechanism would require quite some development effort, making it a complex undertaking within our current time constraints. Nonetheless, we believe that incorporating such time-based incentives could further enhance our model's performance and decision-making in the game of Breakout.

Our project contributes valuable insights into training AI agents, underscoring the importance of hyperparameter choices, learning start times, and strategies. This highlights the need for further exploration, particularly with larger training times to enhance agent performance and adaptability to different scenarios.

VI. Conclusion

In this project, our primary objective was to explore the capabilities of Deep Q-Networks (DQN) in training an AI agent to excel at Breakout, drawing inspiration from the Atari Deepmind Paper. Throughout all of this, it has shed light on the profound abilities of DQN and reinforcement learning algorithms within artificial intelligence (AI) through the context of video games. This exploration of AI and gaming has advanced the boundaries of machine learning, bridging the gap towards human-like thinking and decision-making.

After days of extensive training and analysis, our AI agent has demonstrated problem-solving abilities within the dynamic environment of Breakout, emphasizing spatial awareness, strategic planning, and precision–qualities we aim to instill in our agent. In achieving our goal, we've not only explored the abilities of DQN and RL algorithms, but also contributed to the understanding of machine learning, ultimately pushing the advancement towards human-like AI decision-making.

One more thing… (The collection video of most of our trained models and some final words)

VII. Contribution

- Jovan Ko: Implementation Research, Result Collection, Project Report
- Samuel Huang: Training Experiments, Result Collection, Project Report
- Joanna Doan: Presentation Slides, Project Report, Final Check
- Smriti Davey: Presentation Slides, Project Report, Final Check
- Lisa Verma: Project Ideation, Presentation Slides, Project Report

VIII. Citations

- "Playing Atari with Deep Reinforcement Learning", Deepmind. https://www.deepmind.com/publications/playing-atari-with-deep-reinforcement-learning
- "Convolution", Wikipedia. https://en.wikipedia.org/wiki/Convolution
- "But what is a convolution?", 3Blue1Brown. https://www.youtube.com/watch?v=KuXjwB4LzSA
- "Breakout", Gymnasium Documentation. Farama Foundation. https://gymnasium.farama.org/environments/atari/breakout/
- "DQN", Stable Baselines 3 Documentation. https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html
- "Introduction to RL and Deep Q-Networks", Tensorflow. https://www.tensorflow.org/agents/tutorials/0_intro_rl