# 2048 Deep Reinforcement Learning

**Tej Shah**[*]
Rutgers University
tej.shah@rutgers.edu

**Gloria Liu**[*]
Rutgers University
gl492@@rutgers.edu

**Max Calero**[*]
Rutgers University
mc2104@rutgers.edu

## Abstract

This paper presents and assesses two implementations of deep reinforcement learning agents using double deep $Q$-learning networks (DDQN) and proximal policy optimization (PPO) by training agents to play the challenging stochastic game 2048. One agent is trained using DDQN by estimating the $Q$-values of actions, and the other agent is trained using PPO by updating the policy network directly. In testing, the mode of the maximum tile value the DDQN and PPO agents achieved were $1024$ and $256$ respectively, with scores between $9000$ and $30000$, and between $1000$ and $3500$ respectively. Over all the testing runs, the largest maximum tile achieved by the DDQN and PPO agents were $2048$ and $512$ respectively. Therefore, the DDQN agent could "win" the game sometimes. Though DDQN and PPO perform better than the random baseline, DDQN appears to be more sample efficient than PPO in training an agent to play 2048.

CS462 Deep Learning Spring 2023 Final Project
* equal contribution

# 1 Introduction

## 1.1 Gameplay

2048 is a challenging stochastic game that involves shifting tiles up, down, left, and right on a $4 \times 4$ game board to make tiles with larger powers of two, with a goal to achieve the maximum tile possible before entering the terminal state. Reaching a tile of at least 2048 counts as a win, but the agents can keep playing and attain higher maximum tile values.

In the first state of the game, two tiles will be spawned at two random locations on the game board:

Current Score: 0.0

Max Number: 2.0

| | | | |
|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 |
| 2.0 | 2.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 |

Last Move: no move

Figure 1: Possible starting state

Current Score: 4.0

Max Number: 4.0

| | | | |
|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 |
| 2.0 | 0.0 | 0.0 | 0.0 |
| 4.0 | 0.0 | 0.0 | 0.0 |
| 0.0 | 0.0 | 0.0 | 0.0 |

Last Move: left

Figure 2: Possible second state

A game movement shifts all tiles in one direction as far as possible. During the shift, if two tiles of the same number will combine, then they combine to form one new tile that is twice as big as the two original tiles. After the shift, if there are any open spaces on the board, a new tile will be randomly generated in one of the open spaces. The sum of all newly generated tile values gets added to the game score. For example, Figure 2 is a possible new state after taking a left move from Figure 1.

The game enters a terminal state when moving left, right, up nor down will change the board state (the board is filled with tiles, but no moves will result in two tiles of the same value combining to form a new, larger valued tile).

Current Score: 668.0

Max Number: 64.0

| | | | |
|---|---|---|---|
| 16.0 | 2.0 | 8.0 | 2.0 |
| 8.0 | 16.0 | 64.0 | 16.0 |
| 4.0 | 32.0 | 8.0 | 4.0 |
| 2.0 | 4.0 | 16.0 | 2.0 |

Last Move: right

Figure 3: Terminal state, loss

Current Score: 20260.0

Max Number: 2048.0

| | | | |
|---|---|---|---|
| 2048 | 0 | 0 | 2 |
| 8 | 16 | 2 | 0 |
| 2 | 4 | 2 | 0 |
| 2 | 0 | 0 | 0 |

Last Move: left

Figure 4: A "win" but agent can keep playing

Here are other important clarifications of our implementation of the game:

1. Whenever a tile is generated randomly anytime in the game (at the start or after a move), there is a $90\%$ chance the tile is valued at 2, and $10\%$ chance the tile has value $4$.
2. Sometimes, a board will only change to a different state when certain game movements are taken (Fig 5). If a 'useless' movement is taken, nothing about the game state will change, including the score, but it still counts as a movement during gameplay.
3. The game state is represented by a $4 \times 4$ numpy matrix. The training models use this matrix to make decisions rather than an image of the game board.
4. The goal of this game is to achieve a maximum tile value. The game score is helpful for measuring progress but is irrelevant otherwise. Trained agents' success is only measured by the percentage of games played that reach at least a tile of 2048 and the maximum tile value that the agent can consistently reach.

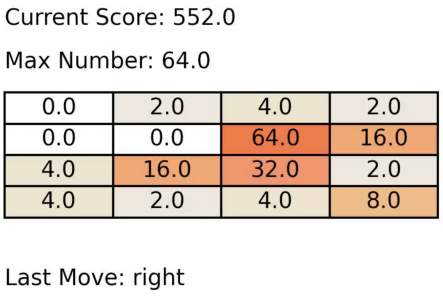


Figure 5: Example of a game state where moving right won't result in any change in game state

### 1.2 Motivation

In the past several years, Deep RL has had numerous breakthroughs, from biology (AlphaFold) to math (AlphaTensor) to games (StarCraft, Atari, etc). Crucially, it is important to explore the strengths and limitations of standard value-based and policy-based Deep RL methods all in one project: Double Deep Q-Networks (DDQNs) and Proximal Policy Optimization (PPO). Furthermore, it is extremely valuable that, as students, we implemented these state-of-the-art techniques from scratch using PyTorch, giving us strong practical exposure to PyTorch and Deep RL. Games serve as a test-bed for developing general-purpose techniques and identifying the failures of existing methods; hence, we hoped to provide insight into developing better Deep RL solutions for the 2048 game.

We chose 2048 as our game of choice because it is easy to understand the gameplay but still challenging to master. The transition between states is relatively straightforward but still has elements of randomness regarding where the new tile will generate. The goal of the game, which is to reach a tile valued at $2048$ or greater, is simple but not easy to achieve. There are at most four possible actions at each state, but each new action creates a new state the model likely has not seen before. Theoretically, there are more than $10^{16}$ possible states in the game's state space before reaching 2048 (There are ten choices of number, namely $\{2^i | 1 \leq i \leq 10\}$, for each of the 16 tiles), and even more if the game reaches 2048 and continues! Despite the appearance of the simplicity of the 2048 game, given its small game board and easy-to-understand game mechanics, the game is actually challenging for deep-learning models to master because of the sheer number of possible states and the elements of randomness. So, this game would truly test the limits of the Deep RL methods mentioned above, but there would not be a lot of overhead when we evaluate the progress and efficacy of the models during training or testing. Using 2048, we could truly focus on the deep learning methods themselves rather than get bogged down with gameplay mechanics.

### 1.3 Preliminary Work

First, we had to build the game itself. Then we had to create functions for plotting model progress. Then we had to create a visualization of the board, in the form of videos, to make it easier to understand and evaluate the actions of the models.

For the game itself, we created a Board class that is initiated every time a new gameplay sequence begins. It keeps track of, and updates the following after each move:

1. **state** - the $4 \times 4$ matrix that keeps track of the tile values on the game board
2. **score** - the game score
3. **max_number** - the maximum tile value in the state
4. **last_move** - the last move played in the gameplay (either none, left, right, up, or down)
5. **terminal** - whether or not the game is in a terminal state

There are also functions for changing these variables according to the rules of 2048 after a player moves left, right, up, and down. There is also a terminal class that can be run in order to play 2048 in the terminal, and entering 'W', 'S', 'A', and 'D' in the terminal corresponds to an up, down, left, and right move. This was used to test that the board class behaved as expected.

In our Plotter class, we use matplotlib to display data collected from having an agent play the game (from a new board to a terminal state) repeatedly. This way, we can determine how well the agent performs. The Plotter class can create a histogram of the frequency of game scores (grouped into bins), a bar chart of the max tile reached, and a histogram of the number of game moves (steps) each game took (grouped into bins). It first creates a JSON file from multiprocessor dictionaries containing the necessary information, then creates the plots from the JSON file and saves them into a file.

We developed visualization functions to enable the representation of game states at every move, facilitating the understanding of the underlying mechanisms and decision-making. To employ these visualization functions for recording gameplay, two tensors are required to be stored: (1) the gameplay tensor, which sequentially captures the $4 \times 4$ board of the game, and (2) a tensor containing the chronological sequence of game scores and moves. During the visualization process, an image of the game state is generated at each move, utilizing the Matplotlib library. These images display a color-coded table of tile values, along with the latest game move, the max tile, and the current game score. Subsequently, the function saved the images for further analysis. Once all images corresponding to the game steps have been generated, the function compiles a video, providing a comprehensive visual representation of the entire gameplay. Then the individual images are then deleted to conserve storage space. We incorporated functionality to execute multiple episodes and retain the gameplay tensor corresponding to the episode with the highest score. This feature enables us to capture videos of the most successful gameplay instances, helping us optimize model behavior.

### 1.4 Cart Pole Testing Environment

Before testing the agents on the $2048$ game, we trained them using the Cart Pole environment as a sanity check. Cart Pole is part of the OpenAI Gymnasium library, which provides environments to test reinforcement learning implementations. In the Cart Pole environment, an agent has to move a cart left or right at each time point in order to balance a pole attached to the cart. The pole starts in a different position each time, so the agent has to learn which action to take based on the state of the environment. For each unit of time the pole remains balanced, the score increases by $1$. The maximum reward in Cart Pole v1 is $475$, and depends on the duration that the pole is balanced (capped length $500$). Cart Pole v1 was used to test the DDQN and PPO agent. Getting the optimal reward/maximum score on the Cart Pole environment should take significantly less time than reaching a 2048 tile in the $2048$ game. Cart Pole has many fewer possible game states and possible actions. So, this was a good way to test the agents prior to training them for a long time on the $2048$ game, although the hyperparameters and reward functions would still need to be adjusted to suit 2048 training.

### 1.5 Random Agent

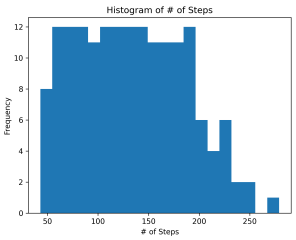


Figure 6: Game Steps

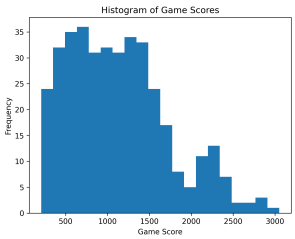


Figure 7: Total Scores

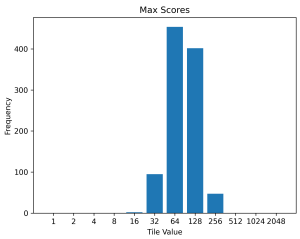


Figure 8: Max Scores

As a baseline to exceed for DDQN and PPO agents, we implemented a random agent that randomly selects any one of the four cardinal directions as its actions until the episode terminates. We simulated $1000$ times the performance on three key metrics for the Random Agent: the number of steps the agent takes in an episode, the game score of an agent, and the frequency of the max score in a particular episode. Generally, games are around 50-200 steps long with nearly uniform probability and 200-275 steps occasionally. Generally, the game score is between 100 to 1500 with equal probability, occasionally between 1500 and 2500, and rarely between 2500 and 3000. Generally, the max tile in all the simulations is $64$ and $128$ but is rarely $16, 32$ and $256$.

## 2 Related Works

### 2.1 Bellman Equations & Stochastic Dynamic Programming

The Bellman equations form the foundation of traditional reinforcement learning by recursively relating optimal values to optimal actions. Within a computing paradigm, these equations can be iteratively solved for the optimal values and the optimal policy using stochastic dynamic programming. The optimal Bellman equations for state values $V^*$ and action values $Q^*$ are given below:

$$V^*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')], \tag{1}$$

$$Q^*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} Q(s', a')], \tag{2}$$

where $p(s', r|s, a)$ is the transition probability of reaching state $s'$ with reward $r$ after taking action $a$ in state $s$, and $\gamma$ is the discount factor. The Bellman equations are the foundation for many reinforcement learning algorithms, including value iteration, policy iteration, and $Q$-learning.

### 2.2 Value & Policy Based Methods

Reinforcement learning algorithms are broadly categorized as value-based or policy-based. Value-based methods estimate the expected cumulative reward for any particular state. They can be used as a backbone for selecting an optimal policy: at every state, take the action that leads to the next highest value. On the other hand, policy-based methods directly map states to actions by computing a probability distribution parameterized by $\theta$ over actions. $V(s)$ is the state-value function, $Q(s, a)$ is the action-value function, and $\pi_\theta$ is a policy parameterized by the weights of a neural net. Some popular value-based include Q-Learning, SARSA, and Deep-Q Networks. Similarly, some popular policy-based methods include REINFORCE, TRPO, and PPO.

### 2.3 Deep Q-Networks (DQNs)

DQNs are a reinforcement learning technique that uses a neural network to approximate the $Q$-value function. The $Q$-value function is defined as the expected future reward of taking action at a given state in a Markov Decision Process. The goal of DQN is to learn a value function that maximizes the expected return from any initial state by computing state values, which gives the optimal policy .

DQN uses a neural network with weights $\theta$ to represent the $Q$-value function, such that $Q(s, a; \theta) \approx Q^*(s, a)$, where $Q^*(s, a)$ is the optimal $Q$-value function. The neural network takes the state $s$ as input and outputs the $Q$-value for each possible action $a$ in that state.

The network is trained by minimizing a loss function that measures the difference between the predicted $Q$-value and the target $Q$-value. The target $Q$-value is computed with the Bellman equation:

$$Q^*(s, a, \theta) = r + \gamma \max_{a'} Q^*(s', a', \theta)$$

where $r$ is the immediate reward, $\gamma$ is the discount factor, and $s'$ is the next state.

The loss function is defined as:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2]$$

One issue with the above Q function is the instability caused by using a moving target to calculate the loss. This was addressed by creating a second set of weights such that the Bellman equation is:

$$Q^*(s, a, \theta^-) = r + \gamma \max_{a'} Q^*(s', a', \theta^-) \tag{3}$$

Then, the loss function is defined as:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2] \tag{4}$$

where $(s, a, r, s')$ are transitions sampled uniformly from a replay buffer $D$, and $\theta^-$ are the weights of a target network that are updated periodically from $\theta$.

### 2.4 Double Deep-Q Networks (DDQNs)

DDQNs extend DQNs by reducing the overestimation bias of Q-learning. DQN uses a neural network to approximate $Q(s, a)$ and updates it by minimizing the temporal difference (TD) error:

$$\delta_t = r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)$$

where $\theta$ are the network parameters, $\theta^-$ are the parameters of a target network that is periodically updated to match $\theta$, $r$ is the reward for taking action $a$ at state $s$ resulting in $s'$, and $\gamma$ is the discount factor. The TD error measures the difference between the current estimate of $Q(s, a)$ and the target value obtained from the environment.

However, $Q$-learning has a tendency to overestimate the value of $Q(s, a)$ because it always takes the maximum over the actions in the next state. This can lead to suboptimal policies and slow convergence. Double DQN addresses this issue by decoupling the action selection and action evaluation in the target value. Instead of using the same network to choose and evaluate the best action in the next state, Double DQN uses the online network to choose the action and the target network to evaluate it:

$$\delta_t = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s, a; \theta)$$

Now the loss function is defined as:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s, a; \theta))^2] \tag{5}$$

where $(s, a, r, s')$ are transitions sampled uniformly from a replay buffer $D$, and $\theta^-$ are the weights of a target network that are updated periodically from $\theta$. This way, Double DQN reduces the positive bias of Q-learning and improves the performance and stability of DQN.

### 2.5 Policy Gradients

Policy gradients maximize expected to return by doing gradient ascent with respect to the policy by using the expected return of sampled actions in a trajectory to make a policy update.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right], \tag{6}$$

where $\tau$ is a trajectory, $A^{\pi_\theta}(s_t, a_t)$ is the advantage function, and $\pi_\theta$ is the policy parameterized by $\theta$.

### 2.6 Generalized Advantage Estimation (GAE)

How do we compute the advantages? Ideally, we want to select actions leading to higher average rewards. Generalized Advantage Estimation is a general-purpose technique to designate advantages to actions, trading off between bias and variance. Advantage functions weigh the relative importance of different observations, updating the network to prioritize states with higher values. In the simplest case, if you have some state $s_t$, an actual reward value of $R_t$, and a predicted value for that state of $V(S_t)$, the advantage of that state could be defined as the difference $R - V(S_t)$. For this particular example as well as other examples of similar types, the main challenge we face is high variance in sampled returns. The key insight behind Generalized Advantage Estimation is to increase the horizon upon which we determine advantages: particularly, an $n$-step lookahead discounted to timestep $t$ may be more advantageous. GAE introduces a new estimator for the advantage function that combines multiple estimators with different levels of bias and variance. The GAE advantage estimator is:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V, \tag{7}$$

where $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference (TD) residual for value function $V$, $\gamma$ is the discount factor, and $\lambda$ is a hyperparameter that controls the trade-off between bias and variance.

The GAE advantage estimator can be seen as a weighted sum of $n$-step advantage estimators, with weights determined by the hyperparameter $\lambda$. When $\lambda = 0$, GAE reduces to the 1-step advantage estimator (i.e., the TD residual), which has low variance but high bias. When $\lambda = 1$, GAE becomes equivalent to the Monte Carlo estimator, which has low bias but high variance.

With experimentation, we can vary the value of $\lambda$: then, GAE can balance the bias-variance trade-off and provide a more accurate and stable estimate of the advantage function. Using the GAE advantage estimator in the policy gradient update can lead to improved learning performance and stability.

### 2.7 Trust Region Policy Optimization (TRPO)

Since samples have high variance, we don't want to overfit any particular batch of data. How can we update the policy based on general-purpose features in the sample? TRPO addresses this problem by constraining the gradient update to a trust region, particularly limiting the gradient update to at most some distance $\delta$ between the old policy distribution and the new policy distribution as measured by a distance measure like the KL divergence. The update rule for TRPO is shown below:

$$\text{maximize} \quad \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A^{\pi_{\theta_{\text{old}}}}(s_t, a_t) \right] \quad \text{subject to} \quad \mathbb{E}_t \left[ KL(\pi_{\theta_{\text{old}}}(s_t) || \pi_\theta(s_t)) \right] \leq \delta, \tag{8}$$

where $\delta$ is a hyperparameter controlling the size of the trust region, and the expectation is taken over the state-action pairs $(s_t, a_t)$ sampled from the old policy $\pi_{\theta_{\text{old}}}$. This constrained optimization problem provides more stable policy updates as opposed to vanilla policy gradient methods, which can take too large of a gradient step and result in the failure of the learning goal.

### 2.8 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a family of algorithms that simplifies TRPO by using a surrogate objective function, which approximates the constrained optimization problem.

We can reformulate the TRPO objective by looking at its dual form and get $L^{\text{PEN}}$ as a result:

$$L^{\text{PEN}}(\theta) = \mathbb{E}_t \left[ r_t(\theta) A_t - \beta_t KL(\pi_{\theta_{\text{old}}}(s_t) || \pi_\theta(s_t)) \right] \tag{9}$$

where $\beta_t$ is an adaptive penalty coefficient, encouraging the new policy to stay close to the old policy. While flexible, it requires more hyperparameter tuning. What if we could approximate the behavior of constraining gradient updates with some pseudo, surrogate objective? That's the idea behind $L^{\text{CLIP}}$:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[ \min \left( r_t(\theta) A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t \right) \right] \tag{10}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and $\epsilon$ is a hyperparameter that controls the degree of clipping. By optimizing the clipped objective, PPO discourages overly large policy updates, thus providing a more stable learning process compared to vanilla policy gradient methods.

# 3 Double Deep Q-Networks (DDQN) Agent

## 3.1 Implementation Details

The Double DQN model is a variant of the Deep Q-Network (DQN) model that aims to reduce the overestimation bias of the Q-values. The main idea is to use 2 networks: an online network and a target network. The online network is updated by gradient descent using the SmoothL1Loss and AdamW as the optimizer . In contrast, the target network is updated by copying the weights of the online network periodically or through soft updates, in which this paper considers the latter. The Double DQN model also modifies the action selection process by using the online network to choose the $\arg\max_{a'}$, i.e the best action and have the target network evaluate its Q-value and taking the Q-value of what the online network considers the best action. This way, the action selection, and evaluation of that action are decoupled, which reduces the overoptimism of the Q-values.

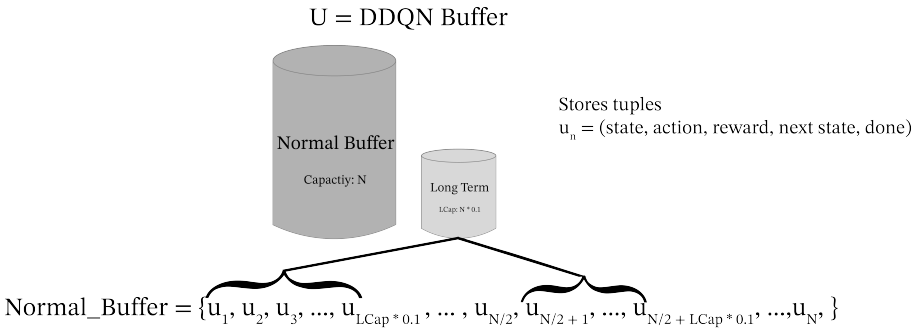


Figure 9: Representation of the DDQN Buffer

We used a replay buffer to store and sample the agent's experiences. The replay buffer was altered to hold early memories sparsely in a smaller long-term buffer. This gives the agent the opportunity to learn from earlier memories more effectively. The replay buffer helps break the temporal correlations of the experiences and improve the data efficiency of the learning process. We also considered using prioritized replay experience as it would've possibly solved this issue as well.

## 3.2 DDQN on Cart Pole Environment

| **Hyperparameters** | **Value** |
|---|---|
| Replay Capacity | 10000 |
| Batch size | 128 |
| Epsilon start | 0.9 |
| Epsilon end | 0.01 |
| Epsilon decay | 1000 |
| Gamma | 0.99 |
| Tau | 0.005 |
| Learning Rate | $1 \times 10^{-4}$ |
| Number of Shared Layers | 1 |
| Shared Hidden Layer Size | 128 |
| Activation Function | nn.ReLU() |
| Gradient Clip Value | 100 |
| Episodes | 500 |

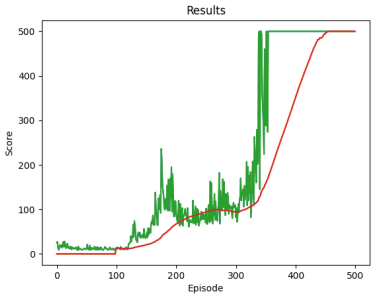


Figure 10: DDQN Cart Pole V1 Training

One of the challenges of DDQN is to ensure that the algorithm converges to a stable solution. To test the performance and robustness of the algorithm, it is common to use a simple and well-known environment such as Cartpole. Cartpole is a classic control problem where an agent has to balance a pole on a cart by applying left or right forces. The goal is to keep the pole upright as long as possible. The game is considered solved if the running average is close to the maximum time possible.

Altering the replay buffers mechanics stems from an issue that arose while testing in cart pole. The agent periodically forget how to recover from failures. To prevent this, we modified the replay buffer to keep some of the early episodes and periodically saved episodes in the long-term buffer. This seemed to have addressed the issue and enabled the agent to learn without forgetting more frequently.

By testing the DDQN algorithm with cart pole first, we can verify that the algorithm works correctly and can learn a good policy for this environment. We can also tune the hyperparameters of the algorithm, such as the learning rate, the discount factor, the target networks' $\tau$ value for soft updates, and the exploration rate. Once we are satisfied with the results on cart pole, we can then apply the same algorithm to a more complex and challenging environment, such as $2048$, although proper hyperparameter tuning was still necessary.

### 3.3 2048 Environment

#### 3.3.1 State Representation

One way to represent the state of the $2048$ board is by using one-hot vectors of length $18$. A one-hot vector is a vector that has only one element equal to $1$ and the rest equal to $0$. In the case $2048$, the position of the $1$ indicates the value of the tile on the board. For example, if the first position is $1$, it means there is a $0$ tile, which is an empty space. If the second position is $1$, it means there is a $2$ tile, and so on. The last position corresponds to a $131072$ tile, which is the highest possible tile value achievable in the game. For example, the one hot vectors for tiles are shown here:

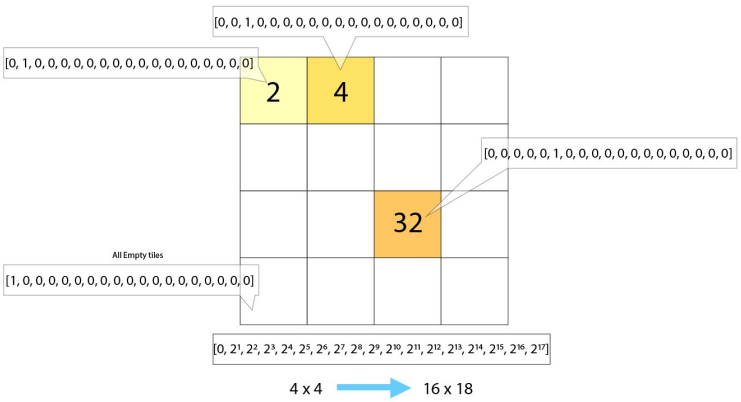


Figure 11: Example of one hot encodings of a board

To encode the whole board, we convert the $4 \times 4$ matrix representing the board to a $16 \times 18$ matrix where each row represents a spot on the board.

#### 3.3.2 Reward Shaping

One of the challenges in designing an agent that can play the game of $2048$ is to choose a well defined reward function. A reward function returns a numerical value that reflects how well the agent is performing at each step and can vastly impact the agents performance. We experimented with various reward functions, such as the change in score, how long it takes to reach a new max tile along with the $\log_2(\texttt{NewMaxTile})$, and the amount of empty spaces on the board after taking an action separately and all together. We found that solely using the amount of empty spaces on the board as the reward function made the most progress in terms of increasing the score and reaching higher tiles.

$$r_{\text{final}} = \texttt{EmpySpots(state')}$$

However, we are not certain if this is the best reward function. Other factors such as, non-tuned hyperparameters may have influenced the choice of this reward function.

| Hyperparameters | Value |
|---|---|
| Replay Capacity | 50000 |
| Batch size | 128 |
| Epsilon start | 0.9 |
| Epsilon end | 0.01 |
| Epsilon decay | 10000 |
| Gamma | 0.99 |
| Tau | 0.001 |
| Learning Rate | $1 \times 10^{-5}$ |
| Number of Shared Layers | 1 |
| Shared Hidden Layer Size | 256 |
| Activation Function | nn.ReLU() |
| Gradient Clip Value | 1000 |
| Episodes | 75000 |

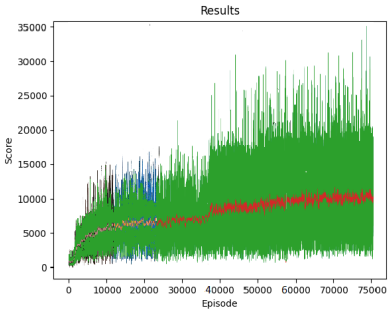


Figure 12: Culmination of all training sessions $r_{\text{final}} = \texttt{emptyTiles(state')}$

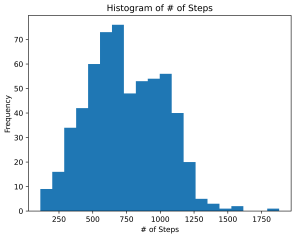


Figure 13: Game Steps

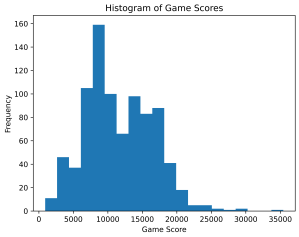


Figure 14: Total Scores

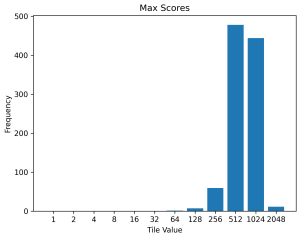


Figure 15: Max Scores

### 3.3.3 DDQN on 2048 Environment

The table above shows the hyper parameters used for training the DDQN agent to play the $2048$ game. The orange plotted line represents the rolling average score of the last $100$ episodes. The inconsistency of colors is due to multiple training episodes and not properly saving the graphs.

As previously mentioned, finding the optimal hyperparmetes was a challenge that arose after finding a suitable reward function, $r_{\text{final}} = \texttt{emptyTiles(state')}$ in our case. We found that adjusting the learning rate was a crucial factor in determining performance of the agent. At a learning rate of $1 \times 10^{-4}$ the agent worked well initially, but stopped learning after reaching an average score of $4000$ at $10000$ episodes. Adjusting the learning rate to $1 \times 10^{-5}$ resulted in faster and smoother learning in the earlier episodes and continuation of learning later on. As demonstrated in Figure 11, this adjustment allowed the agent to reach $4000$ average score way before the $10000^{th}$ episode.

# 4 Proximal Policy Optimization (PPO) Agent

## 4.1 PPO on Cart Pole Environment

| Hyperparameters | Value |
|---|---|
| Shared Hidden Layer Size | 64 |
| Number of Shared Layers | 1 |
| Activation Function | nn.ReLU() |
| PPO Clip Value | 0.20 |
| PPO Policy Learning Rate | $3 \times 10^{-4}$ |
| PPO Value Learning Rate | $3 \times 10^{-3}$ |
| PPO Epochs | 40 |
| Value Epochs | 20 |
| KL Target | 0.02 |
| Number of Rollouts | 4 |

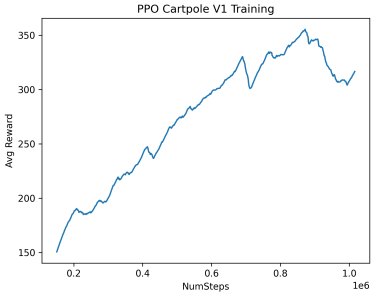


Figure 16: PPO Cart Pole V1 Training

Since deep reinforcement learning is highly sensitive to hyperparameter initialization, we first verified that our implementation of Proximal Policy Optimization runs and learns well on the Cart Pole test environment, where the maximal reward is $r^* = 475$. We verify that our agent is learning appropriately with Proximal Policy Optimization if the reward generally increases and the average reward in the last $k = 250$ episodes approaches $\overline{r} = 475$. Our hyperparameters are shown above.

Per Figure 6, over the training of approximately $1,000,000$ environment steps, we see a non-trivial increase in the smoothed average reward for the agent that learns with PPO. The average reward increased from around $150$ to $350$ throughout training. During the evaluation, we observe that this trained policy is optimal: in $1000$ simulations, the trained model always receives the optimal score of $500$. This difference can be attributed to how the policy network is used during training and testing. During training, we sample actions from the policy action distribution, whereas, during testing, we greedily select the highest probability action from the policy action distribution.

## 4.2 Implementation Details

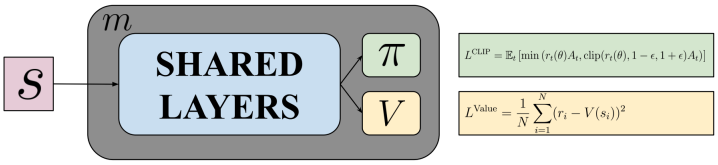


Figure 17: Actor Critic Model

For the model, we use an Actor-Critic architecture. The actor (policy $\pi$) and critic (value $V$) networks have shared layers and separate heads. For simplicity of computation, we use $L^{\text{CLIP}} = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$ to update the actor-network and use mean squared error with respect to the true rewards and expected rewards for the state to update the critic network. We use an Adam optimizer to optimize the Actor Network and the Critic Network. We hypothesize parameter updates to the Critic Network are more stable than to the Actor-network due to lower variance, so we can have the learning rate for $V$ be higher than that of $\pi$.

When training the Actor-Critic model using the update rule for the policy action distribution with the clipped proximal policy surrogate objective and the mean square error for the value function, we use a replay buffer to store samples and compute relevant characteristics about the data to train the model. We implement functionality to run $n$ episodes of the game until termination, storing important characteristics at each environment step: state, reward, action, value function prediction, and the action log probability. Using that information, we compute discounted rewards ($\gamma = 0.99$) for each particular action as well as their advantage estimates ($\gamma = 0.99, \lambda = 0.95$). With all the information now computed for each state in the $n$ trajectories, we randomly shuffle the order of the sampled

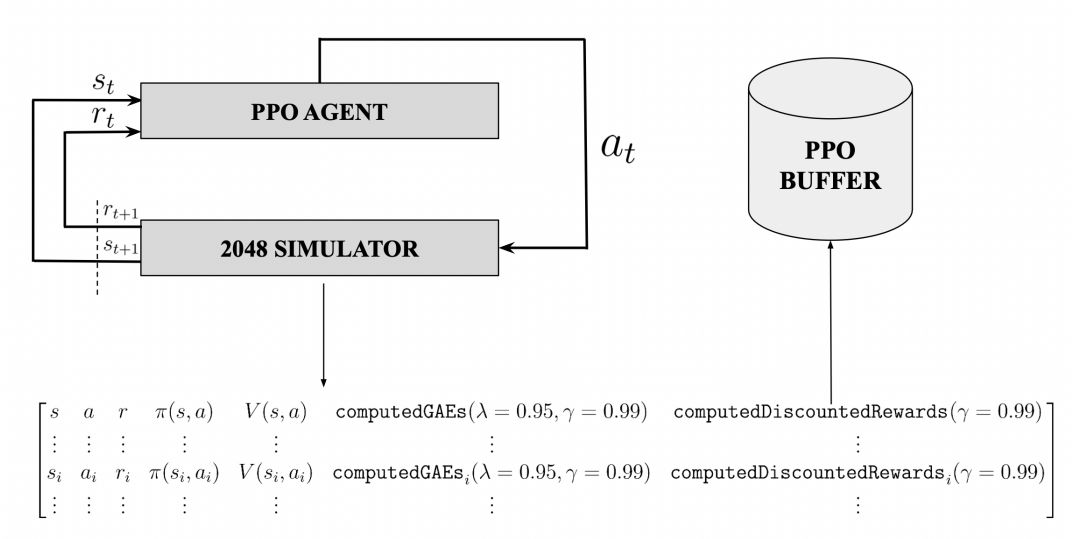


Figure 18: The PPO agent acts in the Markov Decision Process (MDP), taking action $a$ and ending up at state $s$ with reward $r$. We store all simulated trajectory samples with information regarding $s, a, r, \pi, V$, `computedGAEs`, `computedDiscountedRewards` in a PPO buffer.

data to avoid spurious autocorrelation during learning in order to make the training more stable and robust. To avoid too large of a shift in the distribution of the policy, even with clipping, we use an approximate KL divergence bound as a form of early stopping to limit the gradient update.

### 4.3 State Representation

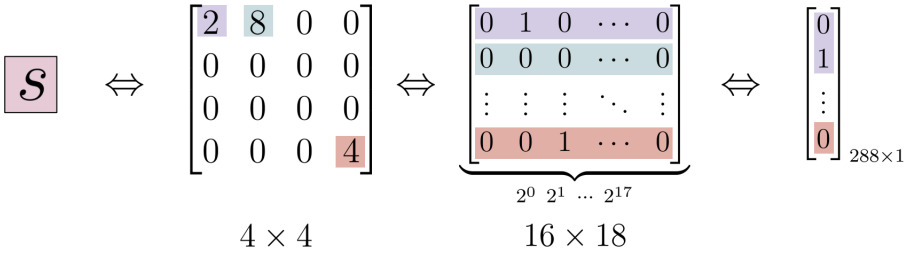


Figure 19: Example of State Representation

As we experimented with various hyperparameters, we adjusted our implementation. First, we observed during testing that our agents sometimes got stuck in an infinite loop. Because we greedily select the action with the highest log probability, if the action doesn't combine any tiles and the board is full, the game doesn't end if a possible action is available (refer to our simulator implementation details). To address this problem, we employed a simple heuristic: if the same action is selected 5 times in a row, the agent must select a new action. Second, we observed that inputting the $4 \times 4$ tiles flattened as a $16 \times 1$ vector into the Actor-Critic network provided little to no learning signal, no matter the hyperparameter setting or reward function. Hence, for each element $a_{i,j}$ in the $4 \times 4$ tile grid, we represent it as a one-hot vector $\mathbf{o}_{i,j}$ of length 18, where the $k$-th element of $\mathbf{o}_{i,j}$ is given by:

$$\mathbf{o}_{i,j,k} = \begin{cases} 1 & \text{if } a_{i,j} = 2^k \\ 0 & \text{otherwise} \end{cases}$$

This representation encodes the value of each element in the tile grid as a binary string, where the $k$-th bit is set to 1 if the element is equal to $2^k$, and 0 otherwise, a flattened $288 \times 1$ vector. By doing so, we can capture the hierarchy of values in the tile grid, where higher-valued tiles are represented

by longer binary strings with more significant bits set to 1. By using this one-hot encoding, we can better capture the state space of the game and provide a more informative input to the Actor-Critic network, which can then learn to make better decisions based on the encoded state information.

### 4.4 Reward Shaping

Besides the state representation being a major problem initially, we also had trouble determining the appropriate reward function. Since $\pi_{ppo}$ directly maps state to actions, we needed a reward that implicitly pushes the policy towards selecting certain actions beyond just scoring the state, which a value-based method does (implicitly giving a policy based on the value function for "free").

We tried various reward functions. Suppose the agent is in state $s$, takes action $a$, and ends up in a new state $s'$. Then, if $g$ represents the game score at some particular $s$, the $\texttt{deltaScore} = g_{s'} - g_s$. Suppose there are $\texttt{numZeros}$ on the $4 \times 4$ game board at state $s'$. The $\texttt{logMaxTile} = \log_2(\texttt{maxTile})$ and $t$ is the number of timesteps since the $\texttt{maxTile}$ last changed value. We tried various rewards functions such as $r^{(1)} = \texttt{deltaScore}$, $r^{(2)} = \texttt{numZeros}$, $r^{(3)} = \texttt{deltaScore} \cdot \texttt{numZeros}$. Unfortunately, all these reward functions had minimal learning signal over a few hundred thousand steps tested or had high variance between different sampled trajectories. After much experimentation, we found a relatively more stable reward function for our policy optimization:

$$r_{\text{final}} = \texttt{logMaxTile}^{\frac{1}{t}} \cdot \texttt{deltaScore}$$

The intuition for this is to give more rewards to actions that get to max tiles more quickly while maximizing the game score. We have used this reward for our final trained PPO policy in the results section of PPO for the 2048 game since it was the most stable in learning during the training start.

### 4.5 PPO on 2048 Environment

| Hyperparameters | Value |
|---|---|
| Shared Hidden Layer Size | 256 |
| Number of Shared Layers | 1 |
| Activation Function | nn.Tanh() |
| PPO Clip Value | 0.10 |
| PPO Policy Learning Rate | $1 \times 10^{-5}$ |
| PPO Value Learning Rate | $1 \times 10^{-4}$ |
| PPO Epochs | 60 |
| Value Epochs | 60 |
| KL Target | 0.02 |
| Number of Rollouts | 8/16/64 |


Figure 20: $r_{\text{final}} = \texttt{logMaxTile}^{\frac{1}{t}} \cdot \texttt{deltaScore}$

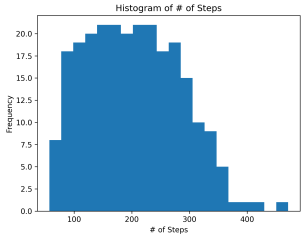


Figure 21: Game Steps

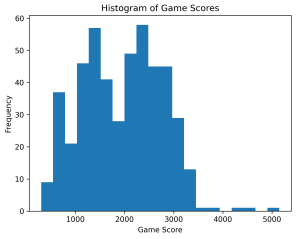


Figure 22: Total Scores

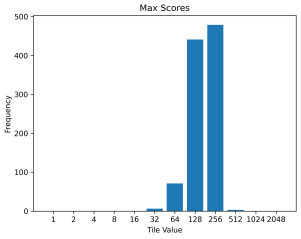


Figure 23: Max Scores

As mentioned previously, we experimented with various reward functions before deciding upon $r_{\text{final}}$, the model upon which we decided to commit to training the PPO model with tens of millions of environment steps. We also experimented with various hyperparameter settings before settling on the ones shown above, which worked best. The details of that experimental run are detailed in this section. We also include details of other non-successful training runs in the appendix for reference.

The number of rollouts was $16$ up to $20$ million steps, $64$ up to $24$ million steps, $8$ up to $35$ million steps, $16$ up to $65$ million steps, and $4$ for the remainder of the training. This was due to manual changes based on the reward training plot. We also experimented with increasing the learning rate from $65$ million steps to $75$ million steps to $5 \times 10^{-4}$ and $5 \times 10^{-3}$ for the policy and value networks, respectively. Since it generally decreased performance, we went back to the original learning rates.

We see that the reward increases from around $4000$ to $12000$ and back down to $10000$ over the course of around $100$ million in environment steps. The environment steps are generally between $100$ and $400$ steps. The total score is generally between $1000$ and $3500$.The max score is usually $256$, sometimes $128$, and occassionaly $64, 32, 512$. Regardless, the policy is better than the random agent in terms of max tiles, which shows evidence of PPO learning.

## 5 Comparison of Results

The DDQN and PPO agents are clearly better in the random agent in getting to higher max tiles. The DDQN agent is much better than the PPO agent and the PPO agent is much better than the random agent. We hypothesize that the PPO agent is more sample-inefficient than DDQN in the case of $2048$ because the only way for the optimal policy to get better is to adopt a different strategy at the start of the game, one which it hasn't explored (i.e. getting higher tiles in corners and make the tiles progression monotonic across the rows/columns). Since PPO directly maps states to actions, we hypothesize that the model we trained opted for a different strategy and achieved a suboptimal policy (in the start of games, the PPO model generally selects actions that go around in circles like up, down, and left, right). Since changing the policy to be more like the current trained DDQN strategy would require it to reduce its reward for a long period of time, we observe that PPO instead chooses to be locally optimal given that initial strategy. DDQN, by contrast, estimates state values, so it's able to change its optimal policy on the fly. That is one plausible reason for our difference in results.

## 6 Discussion

We initially underestimated the instability of deep reinforcement learning methods like DDQN and PPO. We had to experiment with various hyperparameter settings, model architectures, and reward functions for both value and policy-based methods to obtain working solutions.

Reinforcement learning can be data-inefficient, so it would be beneficial to develop efficient processes for data collection in future projects (e.g., utilizing multiple workers for data collection, leveraging GPUs for model updates, etc.). Parallelizing computations and dividing data collection into multiple processes proved more complex than anticipated for the scope of this project. As a result, we were unable to include this functionality for data collection, but we did incorporate it for model evaluation.

Another consideration to improve sampling efficiency would be to add a prioritized replay buffer to the DDQN algorithm. Prioritization makes use of the temporal differences to sample more efficiently and increase the stability of the algorithm. This would not be necessary for PPO since it contains built-in prioritization mechanism. For example, it uses a clipped surrogate objective that penalizes large policy changes and favors transitions that have a high advantage estimate. This way, PPO implicitly prioritizes the most informative and relevant transitions for updating the policy.

Due to constraints such as Google Colab timeouts and the unavailability of specific packages on Rutgers' iLab machines, we opted to focus on generating results rather than dealing with package manager overheads (e.g., conda). Consequently, we trained the models on our local machines, running them for several days and nights to accumulate tens of millions of environment steps or thousands of episodes as samples for the model. Our ability to do this was due in part to the development of an extremely efficient, vectorized 2048 simulator.

For checkpointing and logging the progress of models, we used manual methods. In the future, we will consider more sophisticated solutions, such as utilizing Tensorboard or Weights and Biases.

## 7 Conclusion

We examined how effective DDQN and PPO is in playing the game 2048. The results show that both DDQN and PPO are both promising and lead to much better game results than random guessing,

with DDQN achieving even better results than PPO. After $75000$ episodes, the DDQN agent learned to play 2048 in a way that most commonly achieves a maximum tile value of $1024$, but in around $2.5\%$ of games, the agent achieved a maximum tile value of $2048$, winning the game. The scores usually ranged from $9000$ to $30000$. The DDQN agent used a reward based on the number of empty tiles at the next state. After over $100$ million environment steps, the PPO agent learned a 2048 policy that most commonly achieves a maximum tile value of $256$, but has played games that achieved a maximum tile value of $512$. The games played by the PPO agent had scores generally ranging from $1000$ to $3500$. The PPO agent used a reward that considered the log of the maximum tile value and the change in game score between actions. Both agents performed better than the baseline random agent, which most commonly achieves maximum tile values of $64$ and $128$ and occasionally got up to a tile of value $256$; however, the DDQN agent outperformed the PPO agent in this game.

For this implementation, we created our own game environment, plotting class, and visualization functions. Prior to training our agents to achieve a maximal tile in 2048, we trained the DDQN and PPO agents in the Cart Pole environment, where they always got the optimal reward/score after training for $500$ episodes and $1,000,000$ environment steps respectively.

Besides the assessment of these two approaches, there are several other insights we gained from training the agents. The reward function is extremely important to the success of any agent, and it is necessary to train the agent to work towards achieving a high tile value, rather than keeping itself from going into a terminal state (for example, by repeatedly going up on a board that won't be changed by an up movement). Besides just the score and current maximum tile, the final rewards also contained more subtle information about the game's state like the number of empty tiles on the board. How the state is represented to the agent is also very important. Changing the representation of the state from the $4 \times 4$ board matrix into one hot vector of the power of two provided the agents with more valuable spatial information about the game's state, increasing the scores achieved by the agents. So, when training agents using DDQN or PPO, or reinforcement learning in general, it may be valuable to consider less obvious aspects of the game's state in the reward function, and carefully evaluate how to best represent the game's state to the agent.

Overall, DDQN and PPO are both effective approaches to train agents to play the challenging game 2048 - both agents in our paper learned to play better than a random agent, and the DDQN agent occasionally achieved $2048$ in the game. The techniques are promising in the field of reinforcement learning, and further exploration and investigation of their usage in other contexts is warranted.

## 8 References


- Playing Atari with Deep Reinforcement Learning
- Deep Reinforcement Learning with Double Q-Learning
- Proximal Policy Optimization Algorithms
- Trust Region Policy Optimization
- High-Dimensional Continuous Control Using Generalized Advantage Estimation

# 9 Appendix

## 9.1 2048 Terminated PPO Experiment 1

| Hyperparameters | Value |
|---|---|
| Shared Hidden Layer Size | 256 |
| Number of Shared Layers | 1 |
| Activation Function | nn.ReLU() |
| PPO Clip Value | 0.10 |
| PPO Policy Learning Rate | $1 \times 10^{-5}$ |
| PPO Value Learning Rate | $1 \times 10^{-5}$ |
| PPO Epochs | 40 |
| Value Epochs | 40 |
| KL Target | 0.02 |
| Number of Rollouts | 4 |

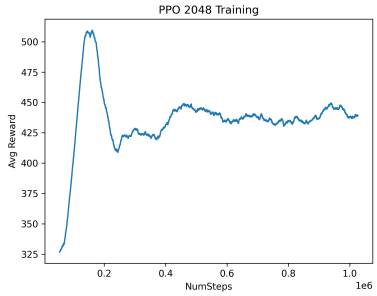


Figure 24: $r^{(2)} = \texttt{numZeros}$

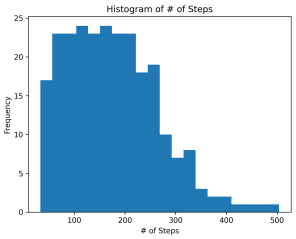


Figure 25: Game Steps

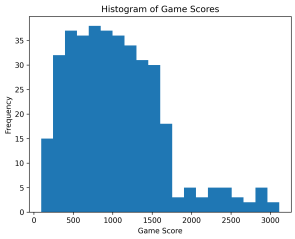


Figure 26: Total Scores

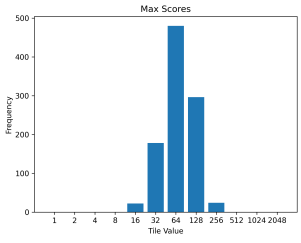


Figure 27: Max Scores

After the first couple hundred thousand steps of monotonic improvement, the score of the agent stagnates at a local minimum. The resulting model results in a max tile distribution that is, at best, equivalent to the distribution achieved by selecting random actions. As always, it's possible that we didn't train the model long enough to see positive improvement spikes in performance. But, given that other reward functions provided better learning signal, we worked on other experiments instead.

### 9.2 2048 Terminated PPO Experiment 2

| Hyperparameters | Value |
|---|---|
| Shared Hidden Layer Size | 256 |
| Number of Shared Layers | 1 |
| Activation Function | nn.Tanh() |
| PPO Clip Value | 0.20 |
| PPO Policy Learning Rate | $1 \times 10^{-5}$ |
| PPO Value Learning Rate | $1 \times 10^{-4}$ |
| PPO Epochs | 50 |
| Value Epochs | 50 |
| KL Target | 0.02 |
| Number of Rollouts | 8 |

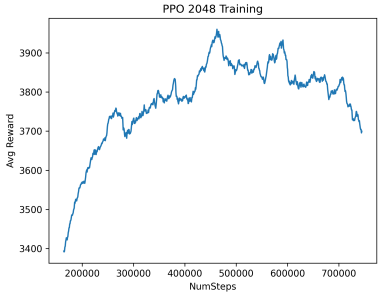


Figure 28: $r^{(3)} = \texttt{deltaScore} \cdot \texttt{numZeros}$

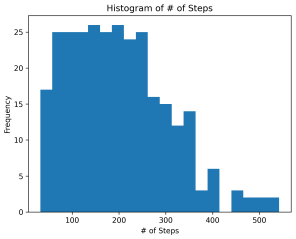


Figure 29: Game Steps

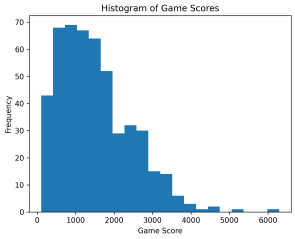


Figure 30: Total Scores

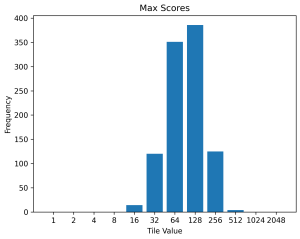


Figure 31: Max Scores

In hindsight, this experiment looks quite successful. There is clear evidence of learning, and the distribution over max scores is higher and better than simply random actions. At the time we did this experiment, we didn't realize that good results would be attributed to millions of data samples - since we saw a flat trend line in learning over the last couple hundred thousand steps, we terminated the experiment. Given what we know now, this method seems equally feasible to train the PPO policy with this particular reward function, depending on whether there is a jump in reward after a certain number of timesteps. It is possible we just didn't train this model long enough!