Amidar-v0

Author: Bulaichi Ionel-Daniel

Faculty: UPT ML 2023 RL

Game Description

Amidar-v0 is an OpenAI Gym game environment based on the classic arcade game Amidar-v0. In Amidar, the player controls a character that moves on a grid-like playfield. The goal is to color or "fill" all the squares on the playfield while avoiding enemies.

The game takes place on a rectangular grid, consisting of multiple interconnected squares. The player character, represented by a small sprite, starts at one corner of the grid. The player's objective is to traverse the grid, painting adjacent squares to fill them. Once a certain percentage of the squares have been filled, the level is considered complete, and the player progresses to the next level with a more challenging layout.

In the Amidar-v0 game environment, the action space consists of a discrete set of actions that the player can take at each time step. The actions available to the player are as follows:

- Move Up: The player moves one square upwards.
- Move Down: The player moves one square downwards.
- Move Left: The player moves one square to the left.
- Move Right: The player moves one square to the right.
- Paint: The player paints the current square they are occupying.

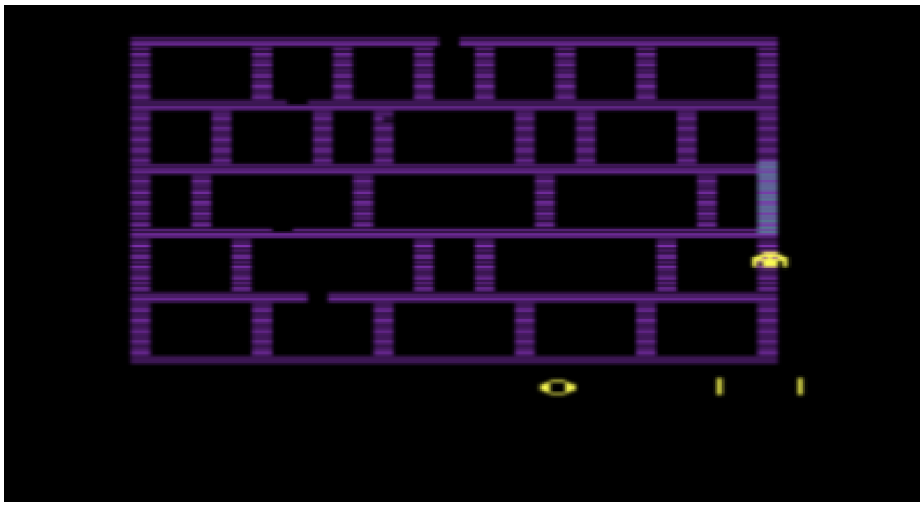

Figure 1 Amidar-v0 game

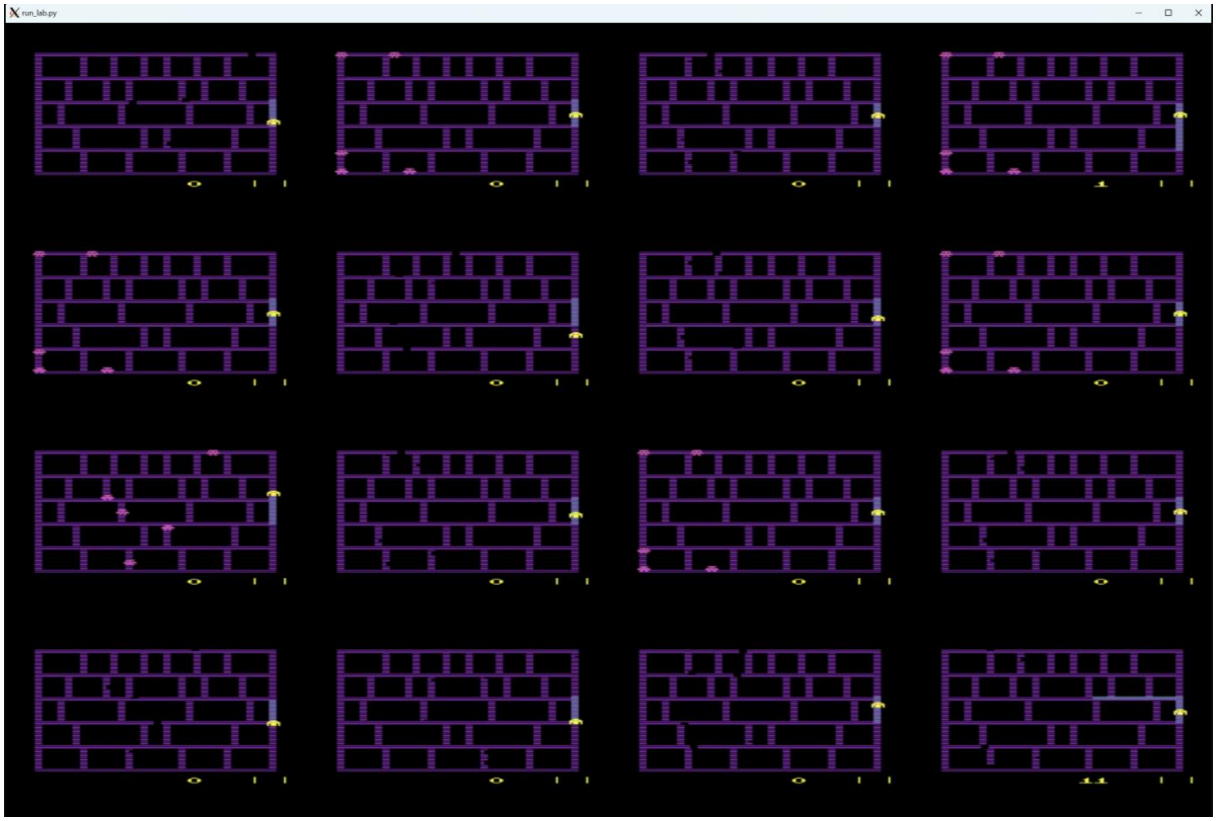

Figure 2 Amidar-v0 game divided in multiple executions

A session of Amidar typically involves a player controlling a character within a grid-like playfield. Here's a description of how a typical Amidar session might look:

- **Game Start**: The game begins with the player's character positioned at one corner of the playfield. The grid is displayed on the screen, showing empty squares that need to be filled.
- **Player Movement**: The player can use the controls to move their character horizontally or vertically across the grid. The character moves from square to square, and their position is updated on the screen accordingly.
- **Square Filling**: As the player moves over empty squares, they are filled, changing their appearance to indicate progress. The player's goal is to fill a certain percentage of squares to complete the level.
- **Enemy Movement**: Enemies, represented by sprites, move around the playfield as well. They may have random movement patterns or follow specific paths. The enemies aim to capture the player, and their positions are updated as they move across the grid.
- **Avoiding Enemies**: The player must navigate the grid strategically to avoid coming into contact with enemies. Colliding with an enemy results in losing a life and potentially a decrease in score.

- **Energizers**: Throughout the playfield, there may be energizers available for the player to collect. When collected, energizers temporarily grant the player the ability to capture enemies by touching them, earning bonus points in the process.
- **Level Completion**: The player continues moving and filling squares while avoiding enemies until a certain percentage of the grid is filled. At this point, the level is considered complete, and the player progresses to the next level, which may feature increased difficulty and new challenges.
- **Scoring**: The player's score is displayed on the screen and increases as they successfully fill squares, capture enemies, and collect bonus items.
- **Game Over**: The game continues with increasing difficulty and multiple levels. However, if the player loses all their lives by colliding with enemies too many times, the game ends. The player's final score is displayed, and they may have the option to start a new session or try again.

The visuals and specific gameplay details may vary depending on the version or platform of Amidar being played. These actions allow the player to navigate the grid and color or "fill" the squares while avoiding enemies. The player can choose one action from the available set at each time step to influence their movement and progress in the game.

However, the player is not alone on the grid. Enemies, represented by other sprites, roam around the playfield, trying to capture the player. The player must avoid coming into contact with enemies, otherwise they will lose a life and respawn at the starting position. To defend against enemies, the player can collect, and use "energizers" scattered throughout the playfield. When collected, energizers temporarily grant the player the ability to chase and capture enemies for a limited time. This provides an opportunity to earn extra points and clear the path for filling squares without being hindered by enemies.

Amidar-v0 in the OpenAI Gym framework provides an interface for interacting with this game environment. Reinforcement learning agents can observe the current state of the game, take actions to move the player character, and receive rewards based on their performance. The goal is to train agents that can learn effective strategies to fill squares and avoid enemies, maximizing their scores and completing levels efficiently.

Algorithms

This section provides a description of algorithms used to train the agent in Amidar-v0 environment using reinforcement learning algorithms.

A2C (Advantage Actor-Critic) is a reinforcement learning algorithm that merges policy and value-based approaches. It utilizes an actor network to learn action selection and a critic network to estimate state values. By iteratively updating both networks, A2C optimizes the policy using advantages to encourage actions leading to greater rewards. It offers stability and efficiency for training sequential decision-making tasks.

A3C (Asynchronous Advantage Actor-Critic) is a parallelized variant of the A2C algorithm. It uses multiple agents, each with its own copy of the network, to explore and collect experiences simultaneously. These experiences are then used to update the shared network parameters, enhancing learning efficiency. A3C enables scalable and faster training of reinforcement learning models by leveraging parallel computation.

DDQN (Double Deep Q-Network) is a variant of the DQN algorithm that addresses the overestimation bias issue. It uses two sets of network parameters, one for action selection and the other for action evaluation, to mitigate the overestimation of Q-values. By periodically updating the target network with the online network's parameters, DDQN improves stability and convergence during Q-learning.

DQN (Deep Q-Network) is a reinforcement learning algorithm that combines deep neural networks with Q-learning. It learns an optimal policy by approximating the Q-values of state-action pairs using a deep neural network as a function approximator. DQN achieves efficient learning by leveraging experience replay and target networks.

PPO (Proximal Policy Optimization) is a reinforcement learning algorithm that balances exploration and exploitation by updating the policy gradually with a proximity constraint. This constraint helps maintain stability during the policy updates. PPO achieves efficiency and robustness through multiple epochs of minibatch updates, making it suitable for various reinforcement learning tasks.

REINFORCE is a policy-based RL algorithm that optimizes expected rewards via gradient ascent. It directly updates policy parameters to maximize cumulative rewards, making it suitable for continuous or high-dimensional action spaces. It uses policy gradients to learn an optimal policy and improve performance in reinforcement learning tasks.

SARSA (State-Action-Reward-State-Action) is an on-policy RL algorithm that learns action-value estimates by updating Q-values using experiences gathered while following a chosen policy. It iteratively updates Q-values for state-action pairs in a tabular manner, making it well-suited for problems with discrete state and action spaces.

Experimental Results

A2C

The algorithm used for training is **Advantage Actor-Critic** with a set of hyperparameters used to train the model, these values can be found in a2c.json in **Amidar/configrations/a2c.json**

The discount factor is set to 0.98, and the lambda value is set to 0.95, the entropy coefficient is specified from line 13 to 19.

Network architecture is a ConvNet, line 27, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 256 units, the used activation function is ReLU, also normalization is enabled and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and the game is executed for 500000 steps.

Next graph is showing that the agent is behave random at the beginning of the game and after 100000 steps agent learn and is going to be more stable.

Figure 3 Total reward for a2c

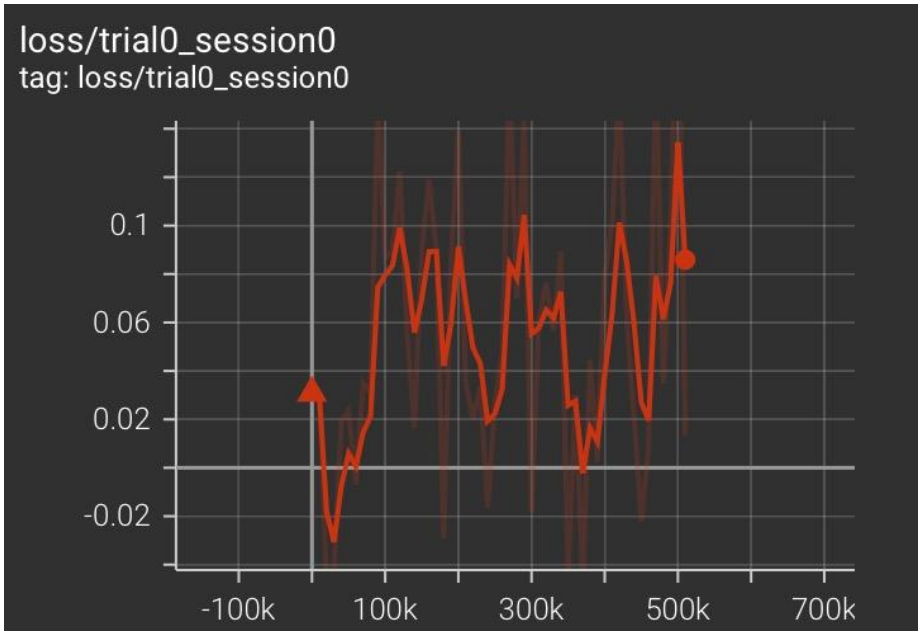

Figure 4 Loss for a2c

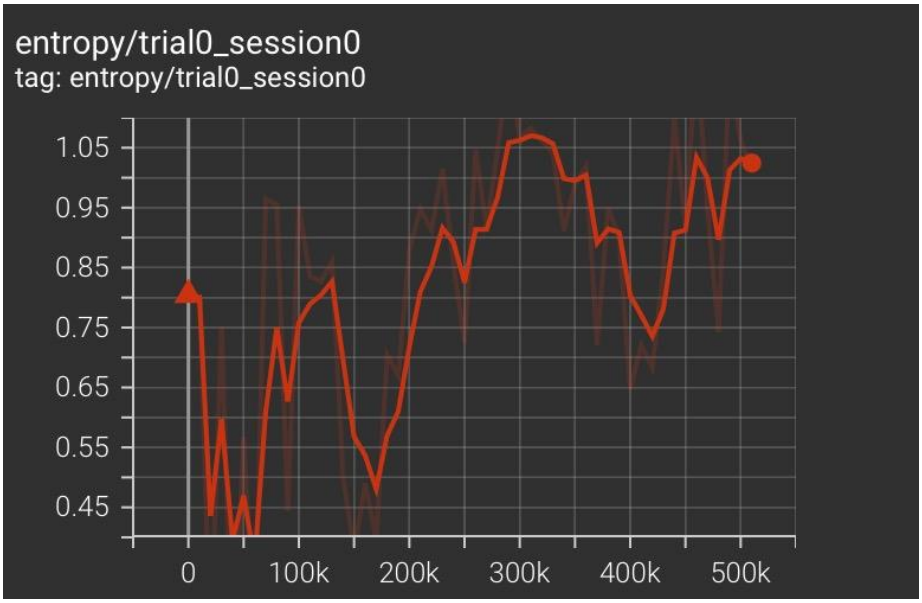

Figure 5 Entropy for a2c

A3C

The algorithm used for training is **Asynchronous Advantage Actor-Critic** with a set of hyperparameters used to train the model, this values can be found in a3c.json in **Amidar/configrations/a3c.json**

The discount factor is set to 0.97, and the lambda value is set to 0.95, the entropy coefficient is specified from line 13 to 19.

Network architecture is a ConvNet, line 34, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 512 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps, and as loss function used is Mean Squared Error.

Next graph is showing that the agent is learning continuously at the beginning of the game and after 100000 steps agent learn and is going to be more stable and in 300000 is decreasing the receiving reward.

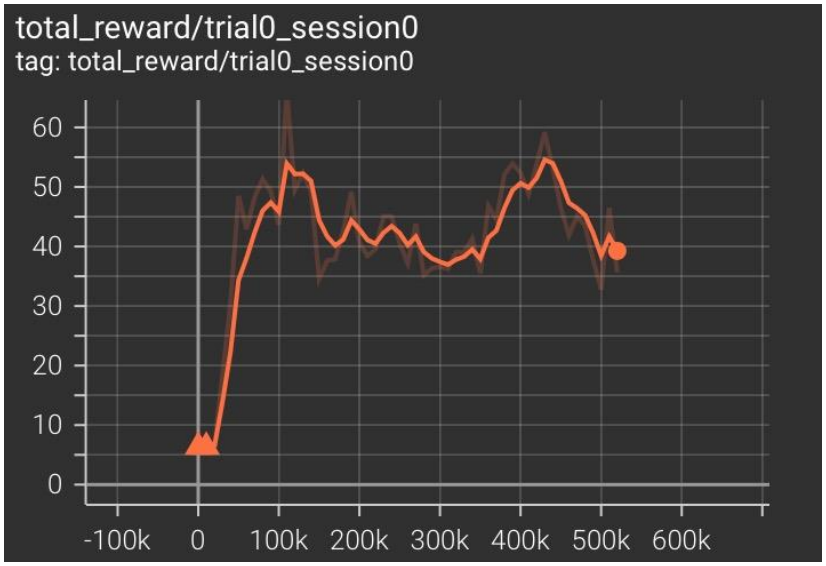

Figure 6 Total reward for a3c

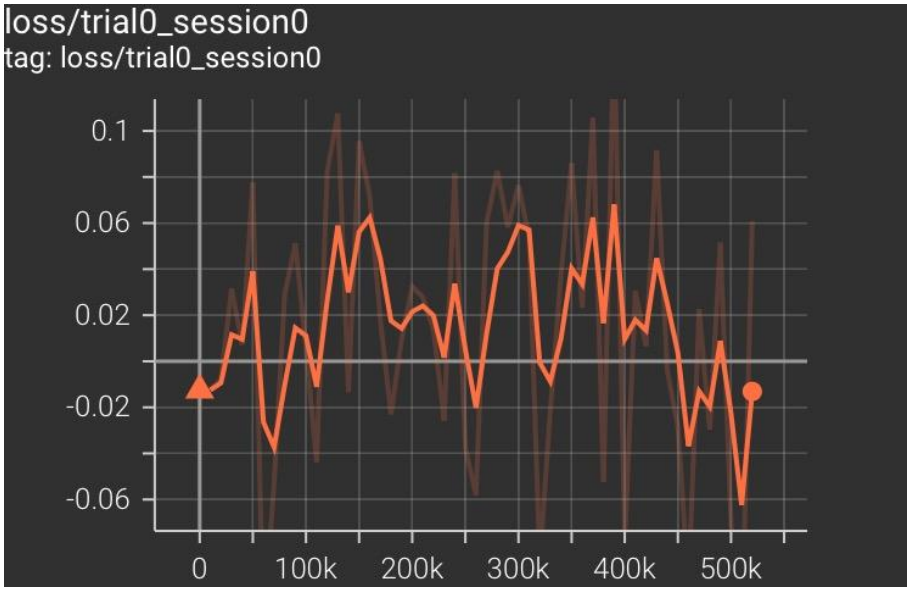

Figure 7 Loss for a3c

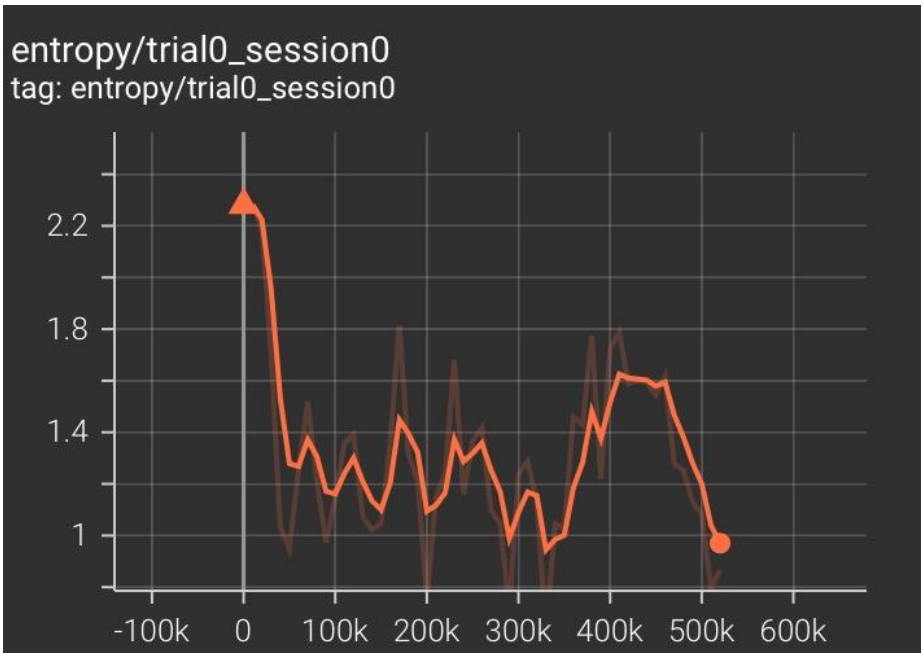

Figure 8 Entropy for a3c

DDQN

The algorithm used for training is **Double Deep Q-Network** with a set of hyperparameters used to train the model, these values can be found in a3c.json in **Amidar/configrations/ddqn.json**

The discount factor is set to 0.99, action policy is epsilon-greedy, the memory is Prioritized Replay with batch size equal to 32 with maxim size of 200000.

Network architecture is a ConvNet, line 31, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 512 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps, and as loss function used is Mean Squared Error.

Next graph is showing that the agent is behave poorly at the beginning of the game and after 300000 steps agent learn and is going to learn and gather more reward.

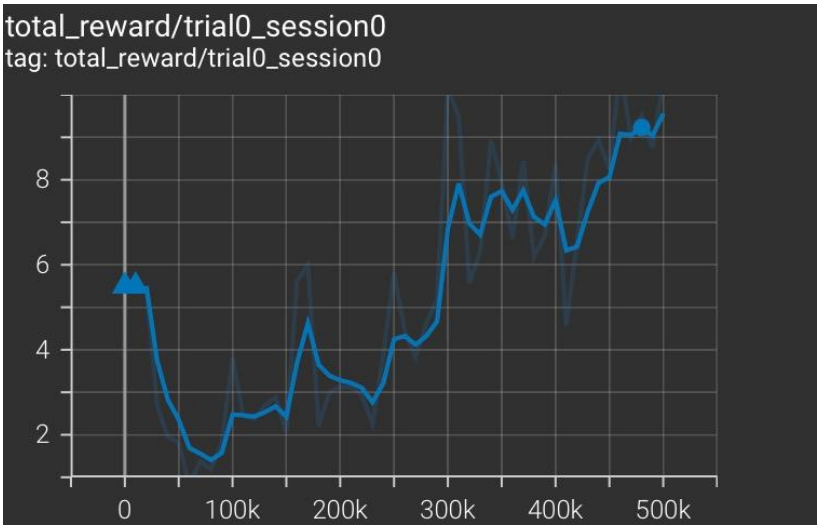

Figure 9 Total reward for ddqn

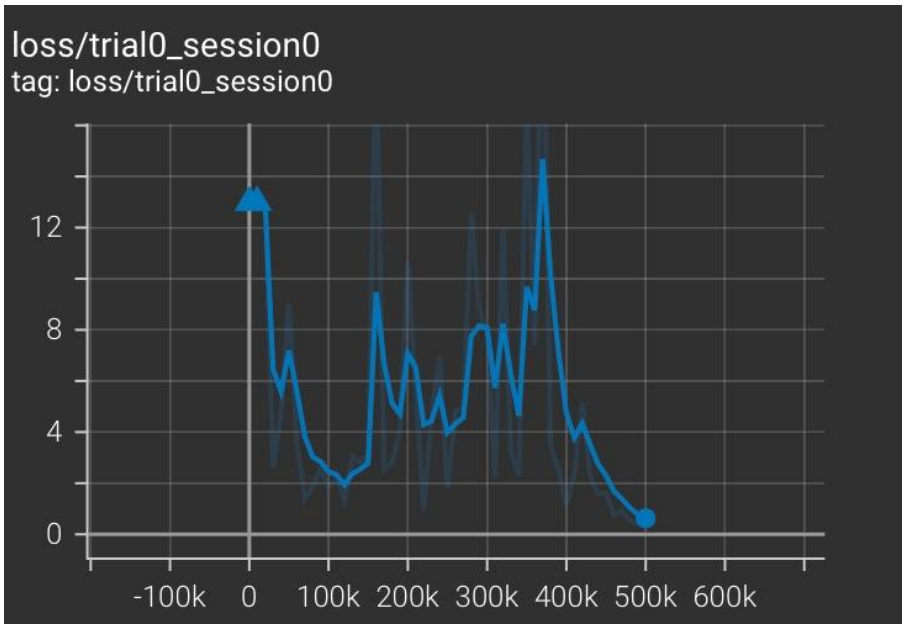

Figure 10 Loss for ddqn

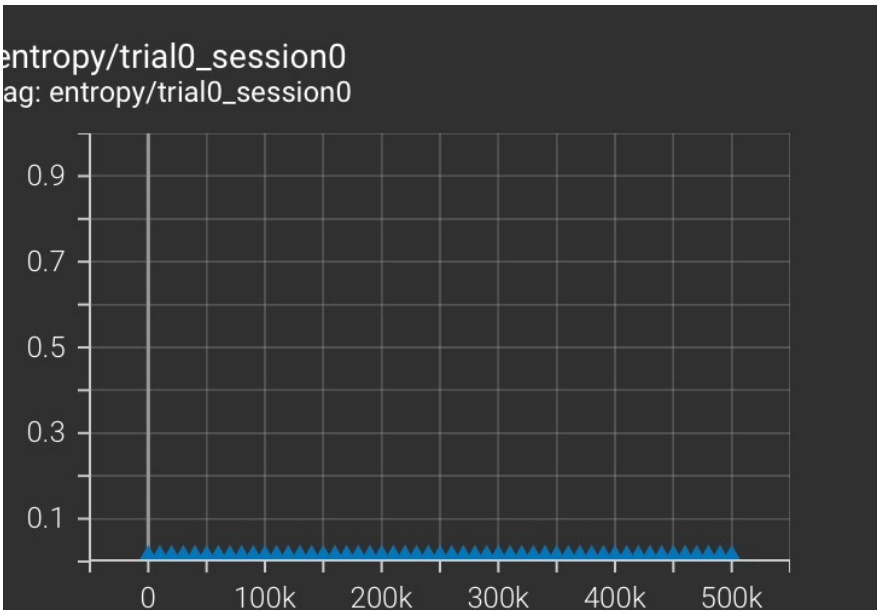

Figure 11 Entropy for ddqn

DQN

The algorithm used for training is **Deep Q-Network** with a set of hyperparameters used to train the model, these values can be found in a3c.json in **Amidar/configrations/dqn.json**

The discount factor is set to 0.99, action policy is epsilon-greedy, the memory is Replay with batch size equal to 32 with maxim size of 200000.

Network architecture is a ConvNet, line 31, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 512 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps, and as loss function used is Mean Squared Error.

Next graph is showing that the agent is learning slowly at the beginning of the game and after 350000 steps agent learn and is going to be more stable.

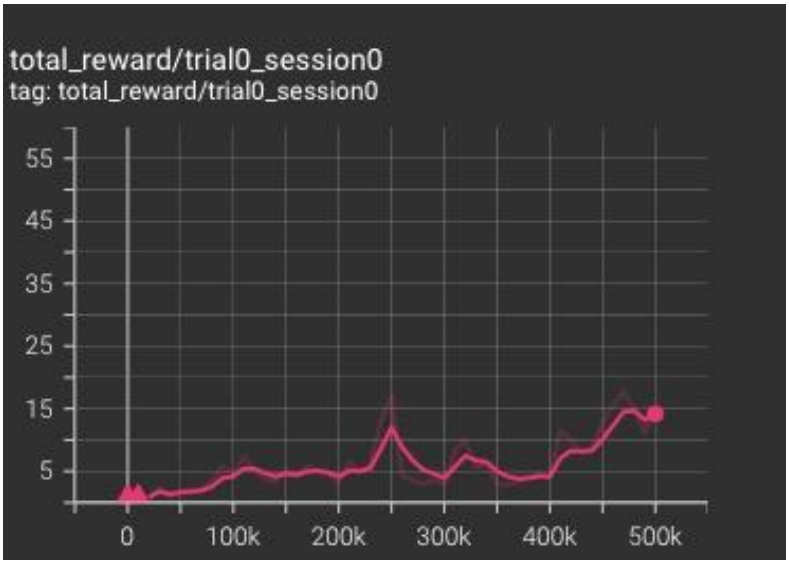

Figure 12 Total reward for dqn

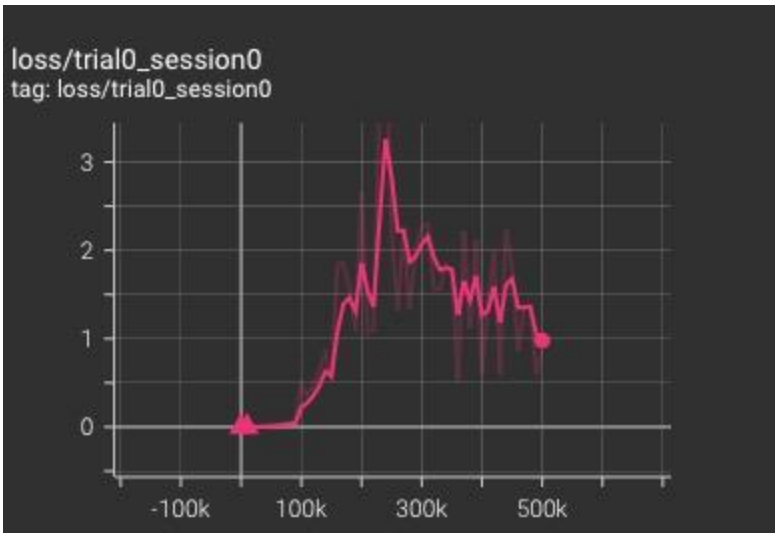

Figure 13 Loss for dqn

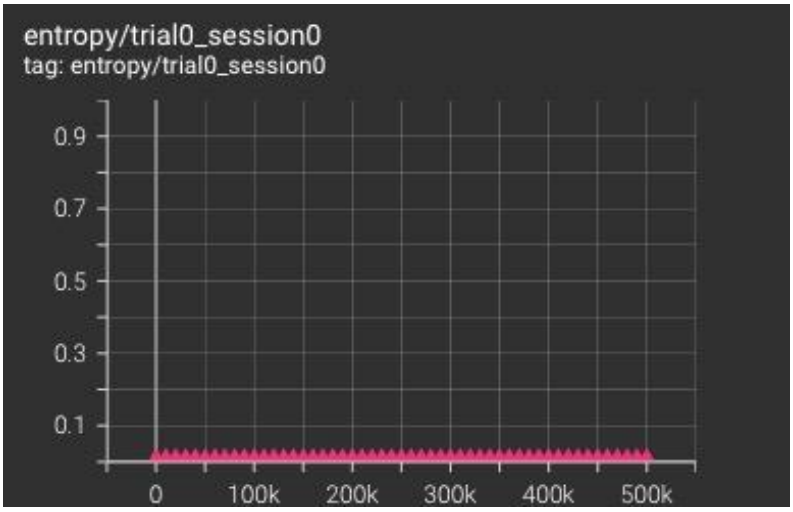

Figure 14 Entropy for dqn

PPO

The algorithm used for training is **Proximal Policy Optimization** with a set of hyperparameters used to train the model, these values can be found in a3c.json in **Amidar/configrations/ppo.json**

The action policy is default policy, discount factor is set to 0.95, value loss coefficient is 0.5.

Network architecture is a ConvNet, line 31, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 512 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps.

Next graph is showing that the agent is learning faster the beginning of the game and after each 100000 steps agent learn more and more until run is finished.

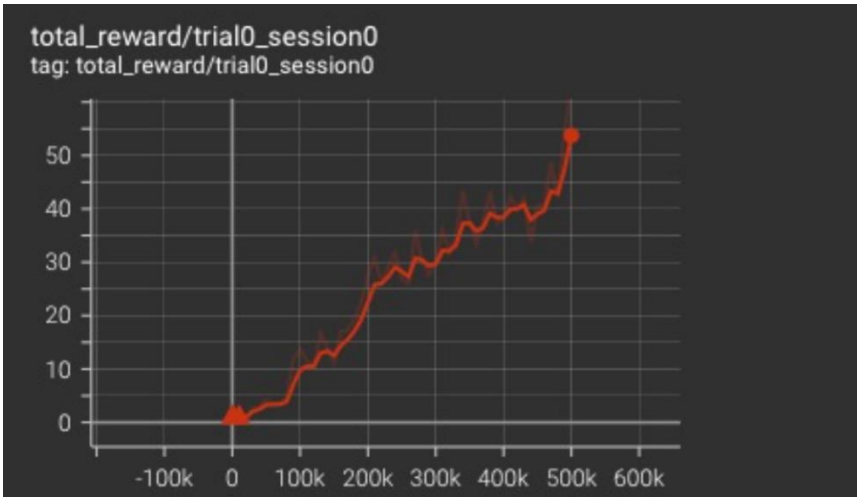

Figure 15 Total reward for ppo

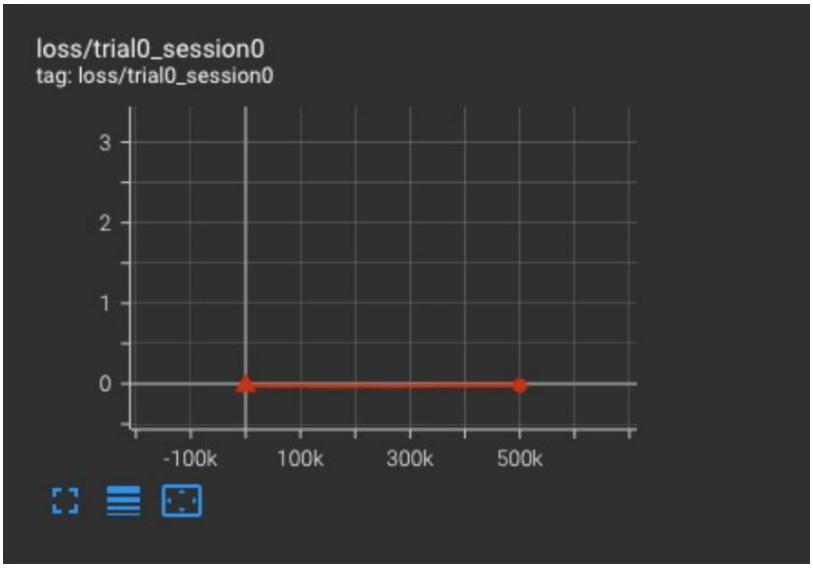

Figure 16 Loss for ppo

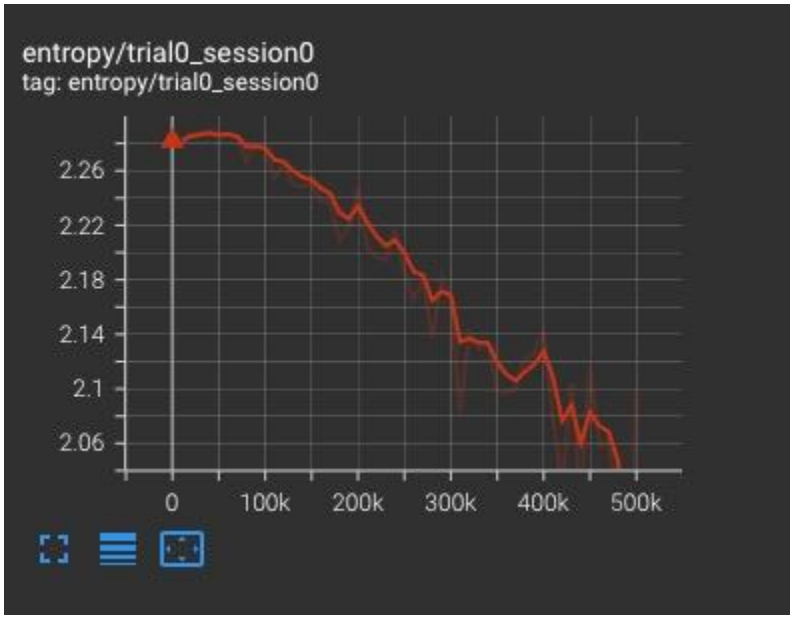

Figure 17 Entropy for ppo

REINFORCE

The algorithm used for training is **Reinforce** with a set of hyperparameters used to train the model, these values can be found in a3c.json in **Amidar/configrations/reinforce.json.** The discount factor is set to 0.99, the memory is OnPolicyReplay.

Network architecture is a ConvNet, line 31, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 256 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps, and as loss function used is Mean Squared Error.

Next graph is showing that the agent is learn poorly at the all stages of the game and after 350000 steps agent have a big uphill learning and after that is still poor on performances.

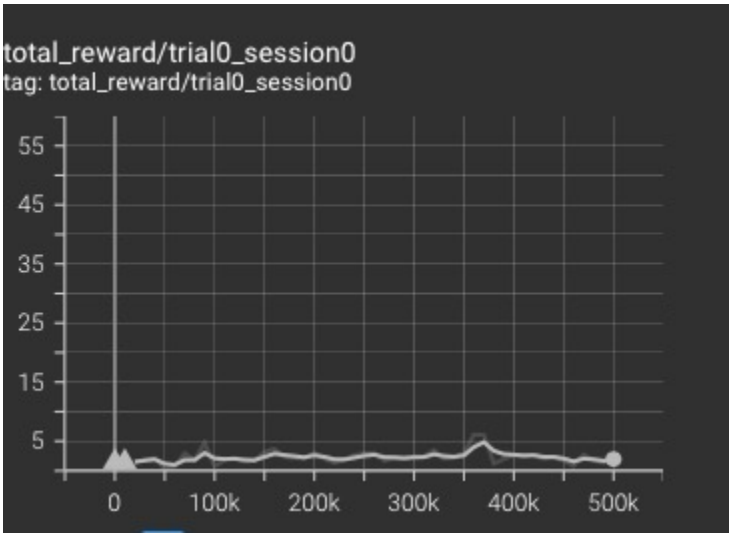

Figure 18 Total reward for reinforce

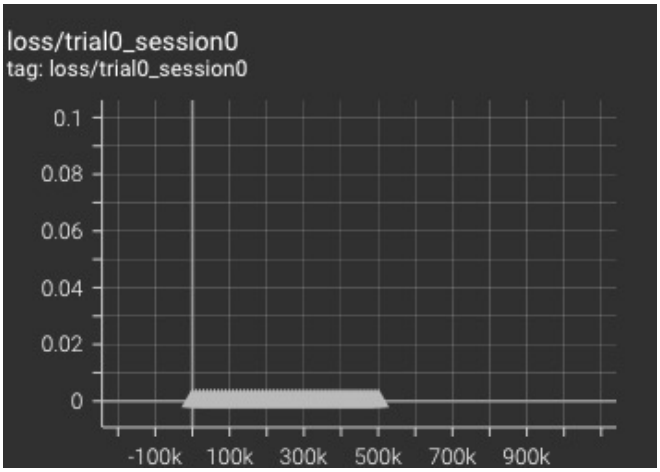

Figure 19 Loss for reinforce

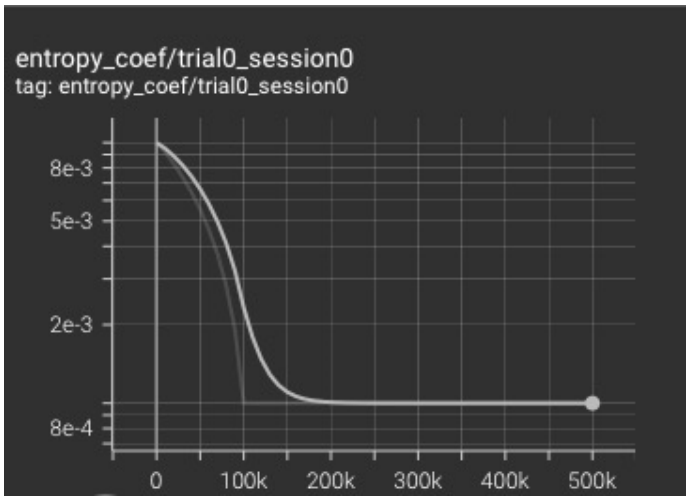

Figure 20 Entropy for reinforce

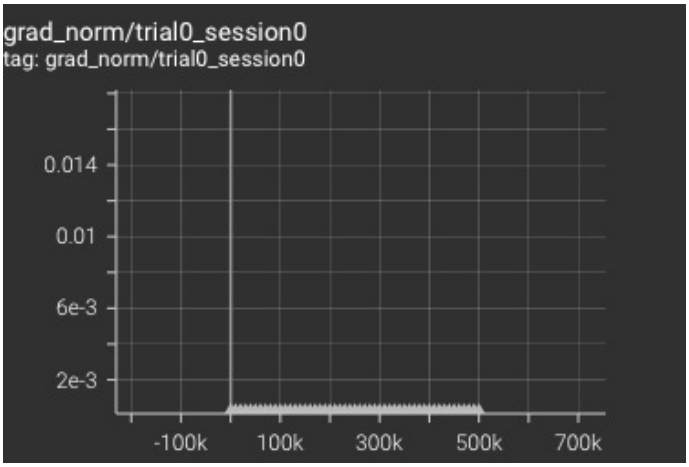

Figure 21 Grad normalization for reinforce

SARSA

The algorithm used for training is **Sarsa** with a set of hyperparameters used to train the model, these values can be found in a3c.json in **Amidar/configrations/sarsa.json**

The discount factor is set to 0.95, to explore more entropy with a linear is added to the loss.

Network architecture is a ConvNet, line 31, it made of convolutional and fully connected layers as is it described in configuration file:

1. **32 filters, 8 kernel size, 4 stride, 0 padding, 1 dilation**
2. **64 filters, 4 kernel size, 2 stride, 0 padding, 1 dilation**
3. **32 filters, 3 kernel size, 1 stride, 0 padding, 1 dilation**

The fully connected layers are made of 256 units, the used activation function is ReLU, also normalization is enabled, and batch normalization is disabled.

The environment where algorithm is executed is named Amidar-v0, the agent is controlling the player and based on the environment response back will learn how to win the game. Agent is evaluated every 10,000 steps, and is executed for 500000 steps, and as loss function used is Mean Squared Error, RMSprop is the optimizer with a learning rate of 0.01.

Next graph is showing that the agent is learning faster at the beginning of the game and after 200000 steps agent learn and is going to be more stable.

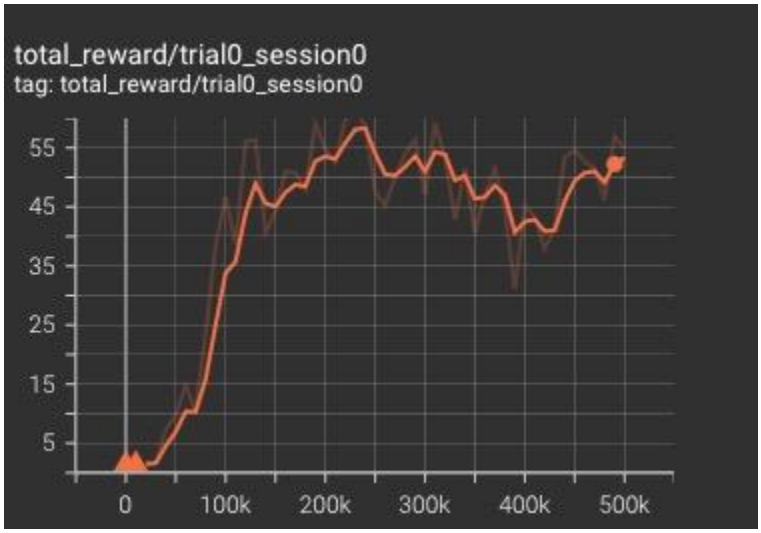

Figure 22 Total reward for sarsa

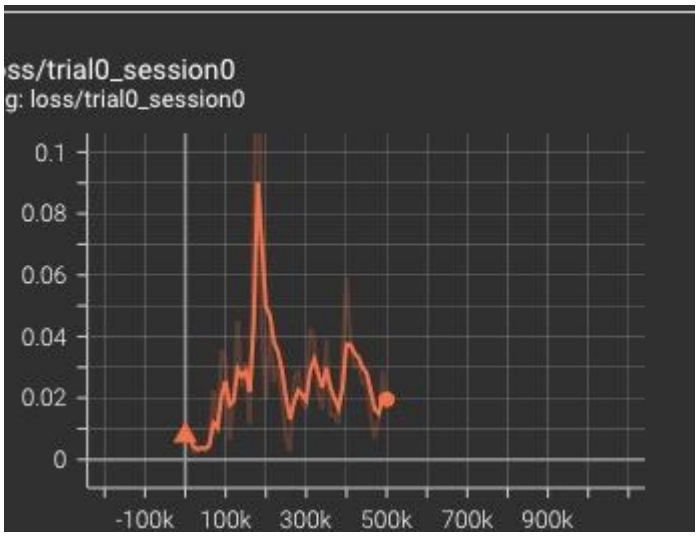

Figure 23 Loss for sarsa

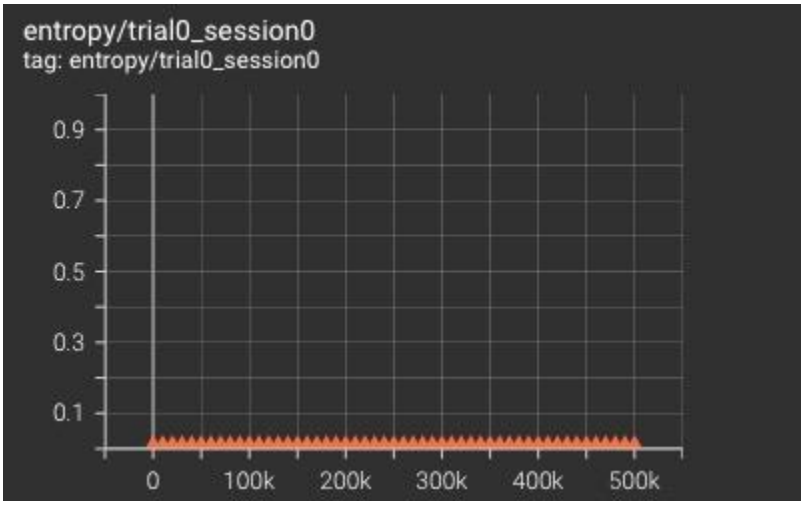

Figure 24 Entropy for sarsa

Conclusions

Based on the evaluation of the algorithms, it can be concluded that PPO outperforms the others, consistently achieving the highest rewards in the game. A2C and A3C demonstrate good performance, although not matching PPO's level. Sarsa performs decently but falls short compared to the top-performing algorithms. On the other hand, REINFORCE exhibits poor performance and struggles to converge to an optimal policy.

PPO consistently exhibits the highest reward accumulation among all the algorithms tested. Its ability to balance exploration and exploitation, along with the stability ensured by proximity constraints, contributes to its remarkable performance.

While A2C and A3C demonstrate commendable performance, they do not quite match the level of success achieved by PPO. These algorithms showcase efficient learning capabilities and have the potential to excel in a range of reinforcement learning tasks.

Sarsa, while not achieving the same level of success as PPO, performs decently. It shows promise in tackling certain problem domains but may struggle with more complex or high-dimensional environments.

On the other hand, REINFORCE falls short in terms of performance, exhibiting limitations in convergence to an optimal policy. Its reliance on policy gradients without utilizing value estimation may hinder its ability to effectively navigate the reinforcement learning landscape.

In summary, PPO stands out as the algorithm of choice in this evaluation, offering a strong combination of exploration-exploitation balance, stability, and high rewards. A2C and A3C show promise as well, while Sarsa performs decently, and REINFORCE lags behind in terms of performance and convergence to an optimal policy.

All graphs and configurations can be found in Github repo created to store all data: https://github.com/Bulaichid/Amidar-v0/tree/main