Reinforcement Learning for Atari Game: Asteroids

Moritz Grüss[1] and Kristian Ruth[2]

[1]gruss@kth.se
[2]krirut@kth.se

February 17, 2025

1 Abstract

In this paper we train a software agent to play the Atari game Asteroids using reinforcement learning. The learning algorithm used was Deep Q-Learning, which was implemented in Python using mainly the PyTorch library. The results of this was an agent which could destroy at least 250 asteroids. The agent achieved impressive aiming and accuracy however the movement of the agent was less significant in the chosen policy. Deep Q-Learning proved to be challenging to work with as the performance of the agent was correlated to the complexity of determining the reward function. Our findings showed that crafting rewards to be given often to inform agents in complex state spaces. This suggests that a preliminary strategy should be determined before training and that the reward system, in addition to the state representation, should reflect this strategy by rewarding behavior closely aligned to it.

2 Introduction

In this report we investigate the use of reinforcement learning (DRL) in creating a program which can play the classic game Asteroids. Reinforcement learning is a subset of machine learning, which involves an agent (initially with zero knowledge) interacting with and learning about an environment in order to improve decision making [1]. This process is differentiated from supervised learning as there is no labeled dataset for training, and from unsupervised learning as the model learns from a reward mechanism rather than attempting to extract patterns in data [1]. In our case we use a specific method of RL called Deep Q-learning, which implements neural networks as the drivers of learning.

Our implementation of RL (in the context of learning a game) requires 4 main components: an agent which receives the state of the game as input and predicts

the best action in relation to predicted reward, an interactive environment (i.e. the environment responds to actions), state representation i.e. the information given to the agent, and a reward mechanism, which is used to guide the model's behavior. The agent corresponds to the model, value function, and policy, and the reward mechanism corresponds to the reward signal according to IBM's definitions [1].

Related works include Mnih et. al's *Playing Atari with Deep Reinforcement Learning*, which uses a more complex convolutional neural network in order to feed raw sensory data to the model, rather than a simplified state representation as we do [3]. OpenAI has also tackled deep reinforcement learning with Dota 2, a famous esports game using techniques such as self-play [4].

3 Contribution

3.1 Requirements and Design

Below we list the requirements of the project:

1. Asteroids-like game environment including: asteroids which move at constant speed in a set direction (towards the player's position at the time they were created) and controllable agent with four actions (shoot, turn clockwise, turn counterclockwise, accelerate in direction of player's heading).
   (a) Input: action array (length 4 corresponding to action space). Picks the action with the highest predicted value
   (b) Activities: update game environment according to action, update asteroid positions, spawn new asteroids
   (c) Output: state representation, performance metrics, updated environment
2. State representation of the environment of the game, including information such as the position and velocity of the player, as well as positions and velocities of a certain amount of asteroids (to be chosen as required for performance and training purposes)
   (a) Output: state array to neural network
3. Neural network with one input layer corresponding to the size of the environment information (state), one output layer corresponding to the size of the action space (four actions), and a certain amount of hidden layers of variable size.
   (a) Input: state array
   (b) Activities: evaluate state, produce values for each action

(c) Output: array of size of action space, each item has a predicted value based on the state the network believes will be created due to the action

4. Reward functionality that can provide feedback to the agent about the quality of the actions taken in the environment
   (a) Input: state array, action chosen
   (b) Activities: evaluate quality of action given the current state
   (c) Output: number corresponding to calculated quality
5. Replay buffer that allows the agent to evaluate actions taken in various states to improve quality of learning and performance
   (a) Input: list of states, list of actions, list of rewards
   (b) Activities: evaluate actions taken in states and the rewards gained and update network parameters accordingly
6. Logging metrics such as average score to evaluate the performance of the agent during training and testing (presented as a graph)

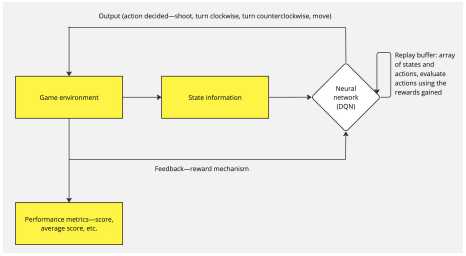

Figure 1: Requirements design

The above figure demonstrates the design of the system in order to fulfill the requirements outlined previously. The game environment produces the state representation, namely player and asteroid information, the deep q-learning (DQN) network has a number of layers including the input (state representation) and output (action decision), which is fed back to the game environment in order to allow the agent to interact with the environment. Additionally the game environment evaluates the action taken and provides feedback to the network through some reward mechanism and graphs the performance metrics. The neural network additionally trains on samples of previous states, actions, and rewards in order to improve decision making.

3.2 Architecture

3.2.1 Initial system design

Figure 2 shows the initial design of the system. It was made to be generic in order to be adaptable to changes made during development. However it contains the necessary components in order to fulfill the requirements. The main game function generates a state representation of the environment every frame and queries the neural network in order to choose an action. The current environment is then updated based on the action (e.g. shoot→spawn projectile). The action and state are passed to the reward mechanism which outputs some floating point reward which is fed back to the network in order to indicate its predicted q-value quality. The network also uses a replay buffer to train itself on series of states, actions, rewards, and resulting states.

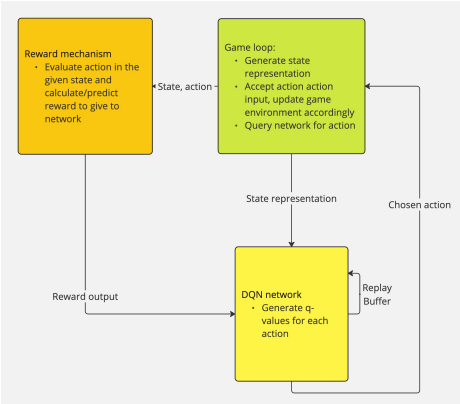

Figure 2: Initial system design

3.2.2 Deep Q-Learning Network Functionality

The neural network is set up using PyTorch, with a linear input layer of size corresponding to the size of the state representation, a linear hidden layer of alterable size, and a linear output layer with output size according to the size of the action space (i.e. four). The first two layers use ReLU activation functions. The training is done by calculating the Q-value of the state resulting from the action, discounting it by γ (discount factor), and adding the reward of the action in the current state. The network predicts the Q-values of the actions in the current state, and the two values are fed through a loss function (in our case `torch.nn.SmoothL1Loss()`, though mean-squared error can be used as well). The loss is finally propagated back through the network and the weights of the network parameters are adjusted in order to minimize the loss. The training function accepts a list of states, rewards, actions, and next states in order to facilitate batch learning as well.

3.2.3 Final system design

Figure 3 describes the final system design with data flows according to our finished project.

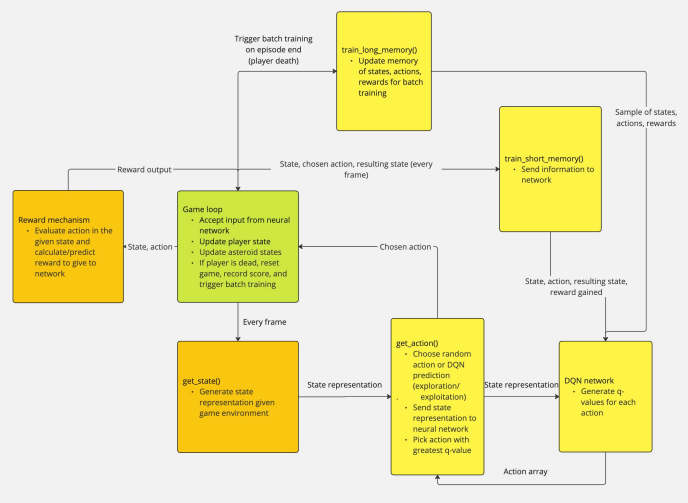

Figure 3: Final system design

3.3 Results

The first stage of development considered a case where the agent is fixed in the middle of a square screen. In figure 4, we can see the results of the training. In this case the agent has only three actions: Shooting, turning clockwise and turning counter-clockwise. Figure 4 shows two curves, in red we can see an arithmetic mean score. This was calculated by taking the sum of every single score from each respective game, and dividing by the number of games. The green curve shows the Sliding mean, which was calculated by taking the arithmetic mean of the last ten games. It should also be mentioned that destroying a single asteroid, is equivalent to a score of 20. As an example, destroying 5 asteroids in one game, would give a final score of 100. For every game recorded, the final score is used to calculate the respective averages.

If we consider the first 100 games, the result might indicate that the agent already knows a somewhat successful strategy. This could be a result of the exploration done by the agent. The exploration in this case is done by picking a random action, at random play-steps. In the shown training example, the randomness was set up such that the probability of taking a random action would be 0.25. This probability was reduced via a linear function which ensured the probability of a random action would go to 0 by game 200. If we consider the

behavior of the agent using highly randomized actions, it seems reasonable that the agent would maintain a relatively low score. If we now extend our view to the first 200 games, we can see a clear decline in performance, in parallel with reduced randomness. The average score is at an all time low around 200 games. At this point, the agent could also be noted to only point at the incoming asteroids, without actually shooting. It takes about another 70 games for the agent to get back up to the highly randomized performance. It then takes roughly 100 games to reach the all time high average of 140. It should also be said that the highest individual score achieved by this agent, during training, was 840.

In addition to the randomness being reduced throughout training, the `train_short_memory()` function was not used after game 200. From this point we were only training using the `train_long_memory()`, which implied sampling batches from all experiences currently stored in the replay buffer. After the 400'th game we also notice some intense drop-off in performance which could be a result of over-training, meaning that the agent has trouble generalizing to unseen circumstances. We might also consider the highest average of 140. This implies that at its best, the agent shot down 7 asteroids on average.

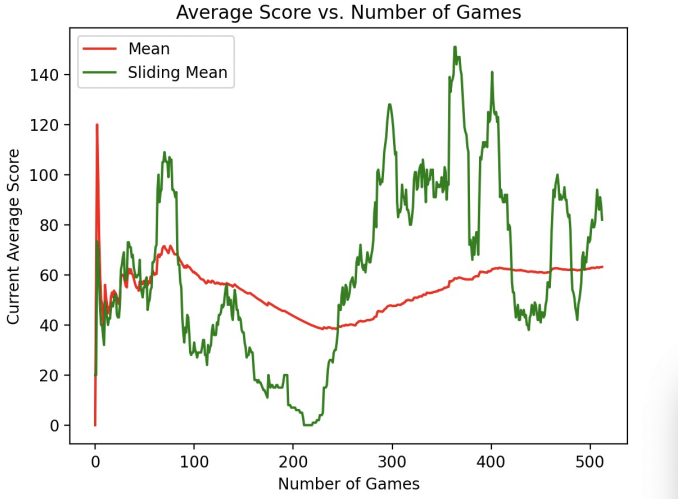

Figure 4: Agent's performance during training, without acceleration

After completing this we added a fourth action to the agent, which was the acceleration. In this way the agent could choose to accelerate in the direction it was currently facing. For this we changed the number of states as well as the information encoded in the states. The state information which was fed into the neural network trained in figure 4 for example, recorded 8 normalized distances between agent and asteroids, in addition to 2 directional indicators of the stationary agent. The updated network would now include 9 states, all of

which were normalized values corresponding to come property of the environment. Here the states were defined as such: x and y position of nearest asteroid relative to agent, asteroids velocities in x and y directions, cosine and sin of players direction, agents position in x and y, agent speed. These states were all normalized as well.

The results of the training is shown in figure 5. Again we use the arithmetic mean of all games and a sliding window. Here we can see a dramatic increase in performance shortly after the 2500 game mark. This coincided with the point where `train_short_memory()` was no longer used and the agent continued to train using `train_long_memory()` only. As we can see the performance was much enhanced by this change if we compare to figure 4. The average score went well above 3000 and the maximum recorded score was 5000, although this was the maximum which we allowed, otherwise the games would go on for too long. When we observe the agent playing the game we see that the agent is extremely proficient at aiming and shooting. The same cannot be said for the movement of the agent, which is slow and infrequent. Regardless of this the agent still moves away from a fixed position and shows that it is able to aim and shoot from different positions on the screen as well.

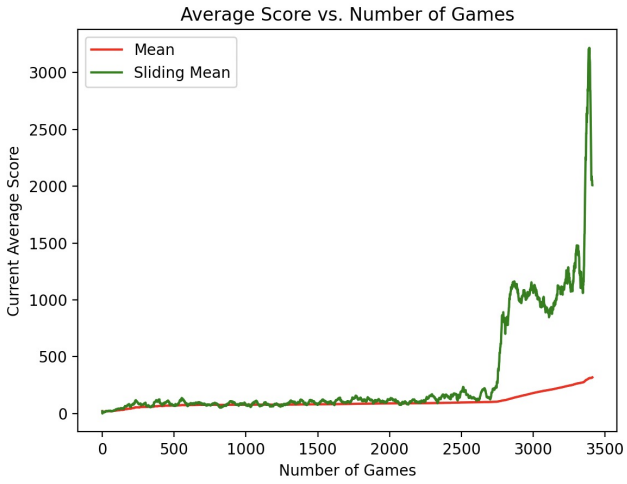

Figure 5: Agent's performance during training, with acceleration

3.4 Evaluation

For this project we set out to train an agent to play the Atari game called Asteroids using a reinforcement learning algorithm. The algorithm which was used to do this was Deep Q-learning. The implementation was inspired by a YouTube

Tutorial [2] originally intended for the Atari game Snake. During the development, another technique was also considered briefly before being considered to impractical. This technique was Reward Modeling which we hoped would enable a more complex reward system, but ultimately failed due to difficulties in manually sorting the data.

The core problem for this project was determining a reward model which would allow the agent to learn a good strategy. This was not so challenging for a fixed agent, but got considerably more challenging for the moving agent. Another thing which added to this problem was that the experiences stored in the replay buffer, had no relation to each other at all. This was a mistake we made initially by giving large rewards for gaining a score for example. Because the experiences capture only a frame of action, rewards would be very infrequent and often wrong, since the reward would be allocated to an experience which resulted after scoring, but not rewarding the action leading up to the score.

The result of this was that the agent had to be guided more strictly. By defining comprehensive reward functions, we taught the agent to play the game like we thought it should be played. As a result of this, the agent could not simply learn a strategy for the game, by figuring out what was making it lose for example. This was probably the biggest drawback of deep Q-learning for this task.

The final reward function rewards according to the dot product between player facing direction and direction to the asteroid (naturally penalizing looking away), and rewards shooting when aligned.

4 Conclusion

In conclusion, the project succeeded in training an agent to play Asteroids. The learning algorithm which was used was Deep Q-Learning and the highest recorded score with our trained model was 5000, which is equivalent to 250 asteroids shot down in one game. This was much higher than any score achieved by the two authors and implies that the agent performed better than a human player. Even though the algorithm performed well, future work should consider an alternative algorithm like a Genetic algorithm for example, to enable the agent to generalize using a less comprehensive reward model.

References

[1] Ph.D Jacob Murrel and Eda Kavlakoglu. "What is reinforcement learning?" In: *ibm.com* (25 March 2024). Accessed 4 Jan 2025. DOI: https://www.ibm.com/think/topics/reinforcement-learning.

[2] Patrick Loeber. "Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake". In: *freeCodeCamp.org* (accessed 28 November 2024). DOI: https://www.youtube.com/watch?v=L8ypSXwyBds&t=1042s.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[4] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG]. URL: https://arxiv.org/abs/1912.06680.