Incremental Reinforcement Learning in Video Games

Maxwell Rider
UML CS Undergraduate
University of Mass. Lowell
Maxwell_rider@student.uml.edu

Alp Yüksektepe
UML CS Undergraduate
University of Mass. Lowell
omer_yuksektepe@student.uml.edu

Ethan Karner
UML CS Undergraduate
University of Mass. Lowell
Ethan_karner@student.uml.edu

Abstract -- Our goal for this project was to come up with a game that utilizes Incremental Reinforcement Learning (IRL) for the AI agent. We chose IRL because most of the major techniques that we see being utilised in games are A* searching algorithms, so we wanted to see how well IRL would do compared to A*. We made a simple dungeon crawler game with the idea of having the enemy chase the player around in order to kill them and win the game. The purpose of the player is to kill the enemy and escape the dungeon.

I. Introduction

Our objective was to implement a Q-Learning based algorithm for the enemy AI. The enemy operates using 2 major algorithms alongside Q-learning. Initially, the AI performs standard Q-Learning to find the optimal policy for the original environment. Next, the agent detects when a reward has changed for a given state-action pair to generate a “drift environment”. After that, there is an algorithm that performs a prioritized sweep of the new environment that updates state-action pair values and merges new and existing information. Finally, we perform Q-Learning on the new environment using the new data to find the new optimal policy. By using this technique, we allow the AI agent to be in constant lookout for the player.

The AI is rewarded for taking actions that move it closer to the player location. The problem, however, is that the agent must know where the player is in the map even while the player is moving. Our chosen method of Incremental Reinforcement Learning allows for the use of Q-Learning in a dynamic environment such as this by updating the Q values each time a change in rewards is detected. In our case, the change in reward values comes from the changing position of the player as they move around the world.

In order for this technique to be effective, the AI must be trained to seek stationary targets in various locations and in various environments. We created several different environments to train the AI on for several episodes before finally allowing the player to control the character and move through the environment while the AI agent attempts to chase them. Training the AI in different environments and different goal locations within those environments allows it to build on previous attempts and make better decisions in a real environment.

II. Literature Review

Our primary source for this project was “A Novel Incremental Learning Scheme for Reinforcement Learning in Dynamic Environments” [1]. This paper includes extensive research and explanations of how to effectively use reinforcement learning in dynamic environments as well as providing possible use cases. Their research shows that by using this approach of incremental learning, they were able to make large performance gains in the average amount of learning steps for each episode. They were able to show that by fusing new and old data the AI agent had improved adaptability to dynamic environments.

The algorithms discussed in the paper were as follows: Drift Detection (Algorithm 1), Prioritized Sweeping of Drift Environment (Algorithm 2), Integrated Incremental Learning (Algorithm 3). Drift Detection is the process of detecting when the reward value of a given state-action pair has changed from its previous value. This works by maintaining a copy of the environment and systematically moving through the environment with a detector agent, comparing the values seen to the values stored in the copy. If any of these values differ then use the changed (state, action) pairs to help generate the "drift environment". The Prioritized Sweeping of the Drift Environment is used to gradually spread the changes caused by the drift environment, fusing this new information into the existing information. Finally, the Integrated Incremental Learning starts by using standard Q-Learning to find the optimal policy of the original environment. It then runs a drift detection agent using Algorithm 1. If a drift is detected then Algorithm 2 is used to update the drift pairs and their neighbors and fuse this new information with the existing information. After these two algorithms are run, standar Q-Learning is then restarted using the values from the new environment.

The problem we aimed to solve differed from the one discussed in the paper in that rather than the environment itself changing (adding or removing walls for example), our goal state location was being changed. We were not able to find papers on this specific problem but found this paper to be sufficient for our purposes.

III. Methodology

We used Python as the chosen language for this project, we chose python for its ease of use in high level programming like this and everyone in the group was familiar with using it for other AI projects. For a graphics engine we chose to use PyGame [3]. We made this choice because it is designed to work in 2D and it was the easiest engine to learn on short notice unlike more well known 3D engines like Unity or Unreal. We took advantage of PyQlearning [2], a Q-Learning library that is available for free on Github. This provided us with several abstract functions to use as well as several completed functions that we used to build our Q-Learning environment. The technique and algorithms used are described in detail in our primary source paper[1] used for this project. In short they are Drift Detection, Prioritized Sweeping of the Drift Environment, and Integrated Incremental Learning. The combination of the three algorithms would allow for the agent(s) to learn to navigate a dynamic environment with large increases in learning performance over standard Q-Learning.

At its core, the game is a very simple one. One human player versus an AI enemy player. The goal for the AI is to catch the player before the player can successfully kill the AI. For our human player movement, we bound the W A S D keys to movement making use of some built in functions from PyGame. The player can click and drag the mouse to fire projectiles at the location of the cursor with the purpose being to hit the enemy. The enemy has a large health bar to give the human player more of a challenge than simply hitting it once.

The player has to weave around the map, avoid the enemy AI, and successfully kill it to win the game. The AI enemy is rewarded for moving closer to the player's location with the goal of reaching the same tile as them. The win state for the AI is when the it and the player share a tile. This is a "kill" for the AI. Using the algorithms described previously the AI would be able to quickly determine new optimal paths to take in order to catch the player.

In order for the AI to perform well in a dynamic environment it must be trained in a static environment first. We created a variety of training environments (figures 1 - 5) that increase in complexity. These would be used to test the AI in a variety of environments and a variety of static goal locations within those environments. After several training episodes are completed, the AI would then be put into a "real-world" test (figure 6) where the player would have control of their character and be free to move about the world.

While we didn't achieve a working model of this technique, we expect that some modification of the Drift Detection and/or Prioritized Sweeping algorithm(s) would be required for our purposes. Since our use case isn't exactly the same as the intended one we may need to alter the way the (state, action) pair reward values get updated.

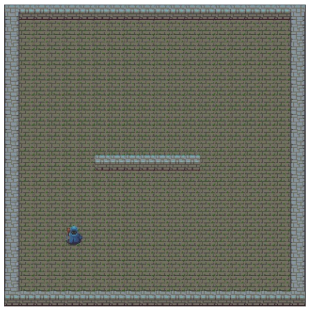
Figure 1

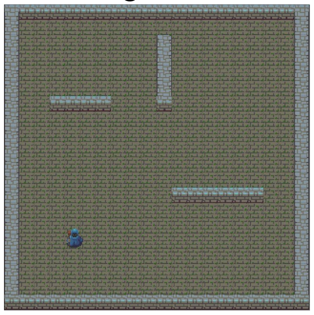
Figure 2

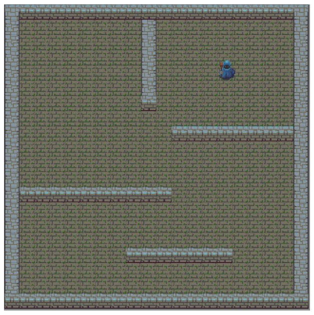
Figure 3

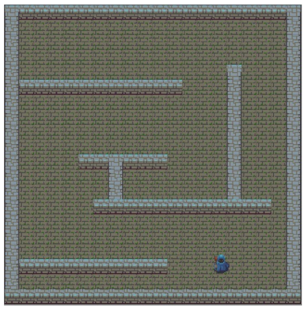
Figure 4

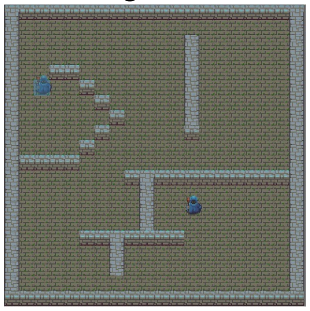
Figure 5

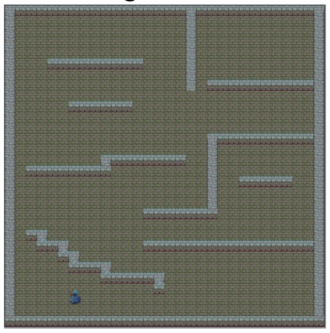
Figure 6

IV. Results

Despite not having a runnable game to show, we were able to figure out more about using Q-learning algorithms for games. In our research we reached the conclusion that while this technique would likely give us the desired results, it is not a practical solution for use in games. The space complexity of Model-free Q-learning is O(S, A, H); where S is the number of states in the environment, A is the number of actions and H is the number of episodes. This means that given a large state space and number actions, this technique will use large amounts of memory, especially with more than one AI agent. Memory is a very precious commodity in games so memory heavy algorithms are typically frowned upon.

We decided to compare our chosen technique to other popular searching algorithms that are used in games to compare both the space complexity and time complexity. We compared our incremental reinforcement learning time and space complexity with two very common AI algorithms, A* [4][5] and D* Lite [4][5].

A* is safe and reliable, however its time complexity is entirely dependent on the heuristic. It's space complexity is O(b^d) , where b is the number of average successors per state, and d is the depth. The reason why A* is entirely heuristic dependent because the heuristic can prune the b^d nodes that an uninformed search would expand. Overall, the space complexity of A* would be costly for our applications, since we have a lot generated states/nodes in memory.

D* Lite was another algorithm that we considered using, since that was also a common AI technique that's used in games. However, unlike A* searching, D* Lite starts looking backwards from the goal node, and keeps going. This is more useful for a complex setting as well as an environment with obstacles[4]. D* Lite could've been used in our setting, and most likely would've had a better space complexity than A* or Incremental Reinforcement learning.

V. Conclusion

In our project, we looked at the ease and practicality of using Incremental Reinforcement Learning as a method for making video game NPCs more intelligent and challenging to beat in a dynamic environment. We were able to partially answer the question we asked. We discovered that while this technique would have likely given us our desired result, Q-learning is not suitable for games with a large state space and even a moderate amount of actions. While this was our expected result, we were not able to prove this experimentally. While games are often used to showcase different Reinforcement Learning algorithms, we don't expect a technique like this to be used in full fledged games that aren't algorithm showcases due to the constraint on memory.

References

[1] Wang, Z., Chen, C. and Dong, D., 2016. A Novel Incremental Learning Scheme For Reinforcement Learning In Dynamic Environments. [PDF] Available at: <https://www.researchgate.net/publication/308987768_A_novel_incremental_learning_scheme_for_reinforcement_learning_in_dynamic_environments>

[2] Accel-Brain. "Accel-Brain/Accel-Brain-Code." *GitHub*, github.com/accel-brain/accel-brain-code/tree/master/Reinforcement-Learning/pyqlearning.

[3]"GettingStarted - Wiki." *GettingStarted - Pygame Wiki*, www.pygame.org/wiki/GettingStarted.

[4] Users.informatik.haw-hamburg.de. 2020. [online] Available at: <https://users.informatik.haw-hamburg.de/~schumann/BachelorArbeitCarinaKrafft.pdf>

[5]Choset, H., Robotic Motion Planning A* and D* Search. [PDF] Available at: <https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf>

[6]*Techwithtim.net*, www.techwithtim.net/tutorials/game-development-with-python/pygame-tutorial/.

[7]Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E., 2004 Difficulty Scaling of Game AI [PDF] Available at: <https://spronck.net/pubs/SpronckGAMEON2004.pdf