Is Transfer Learning Possible in Reinforcement Learning?

Scaling Difficulty on a Trained Agent in Super Smash

Abstract

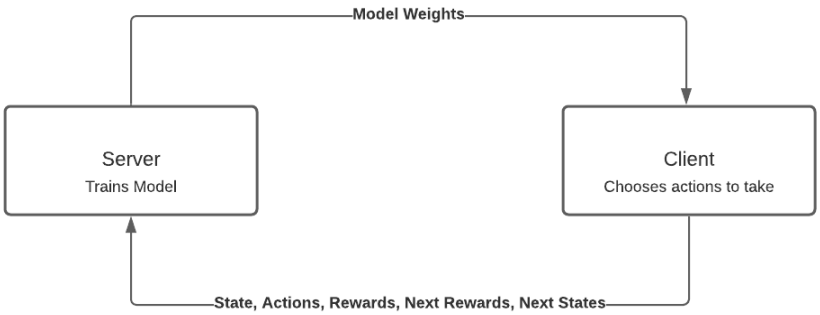
Playing video games is a complex system that requires planning and intricate strategies in order to successfully win against other players. There is a need to to quickly adapt agents to more difficult players or strategies. As such, this work introduced an agent that would originally learn against a simple opponent, then be trained against more difficult opponents and learn to adapt to them in decreased training times. This study demonstrates the possibility of transfer learning within video games and self-play to solve problems. This would allow a user or a company to have a baseline agent trained to play any game and would allow the user to change the desired behavior to learn to play with less time and computational power compared to retraining an agent from scratch. This could be powerful, allowing for users to create an AI that could potentially adapt to the level of difficulty of the person they are playing, providing a difficult, yet rewarding experience for everyone playing regardless of the player's skill level.

Introduction

Super Smash has been a classic video games since the 90s. From families playing together to the classic college dorm game everyone plays together, it is a popular and complicated game that has many varying and complicated strategies. As such, the question of the possibility to create a reinforcement agent to play the game and beat the prebuilt bots was raised. Thus, the idea was to explore the possibility of training an agent to play against a level one bot, then after it can beat the bot, train against a level three and train quicker than it would have if it had just trained originally from a level three bot. The weights after training on the level one bot were saved, then one was trained on a level two bot and one on a level three bot and the number epochs required in order to win were analyzed. This idea came from transfer learning in image recognition, which is taking a pretrained network and training it to recognize a new object that hadn't previously been known by the system. This is a parallel to that idea in reinforcement learning. The models were then trained with an AMD Ryzen 5 with a Nvidia GeForce 2060 6GB graphics card for the client and a 2 processor Intel Xeon computer with 128 GB of RAM for the server computer.

Methods

In this paper, a Deep Q Learning algorithm was used for training. A Server-Client model was also used to avoid RAM issues and emulator time out issues. This was necessary because if the emulator went about a half second without receiving a packet from the client, it would assume that the client had disconnected, and would shut the program down. This caused problems when training, since even starting the training process on another thread would also crash the emulator. Thus, it became necessary to migrate to a server-client model. I used Django as the server side which was convenient for saving states, even after starting since I was able to use a database and serve from there. Then, it became necessary to distribute the system across more than one computer because I was maxing out the RAM used, since Django needed to load a new model every time it trained, and sometimes it would be training more than one model at a time. The Server-Client interaction is as follows: The client makes a GET request that the server returns the weights of the model. It loads these weights, plays out a game, then POSTs the results to the server, and starts another game. The server, upon receiving these results, saves them, then trains on the data. This allows the client to make a GET request and receive the updated weights the next time it asks.



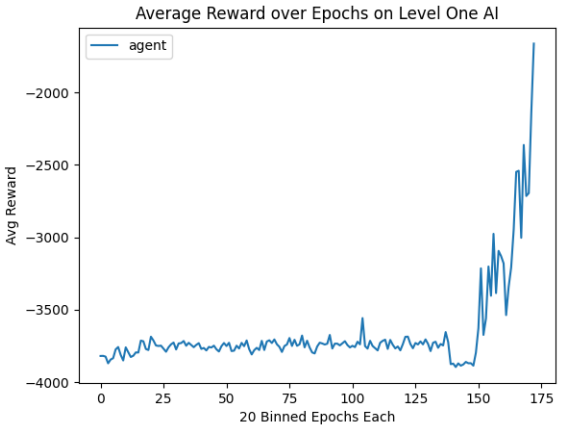
All of this is done concurrently to prevent emulator time out. The results that are posted consist of the player's lives remaining, the percentage they are at, the velocities and positions and knowledge of the same basic facts of its opponents as well and the action that they last took. Then, we take a batch of 128 different memories, which are stored on the server and applied the following algorithm:

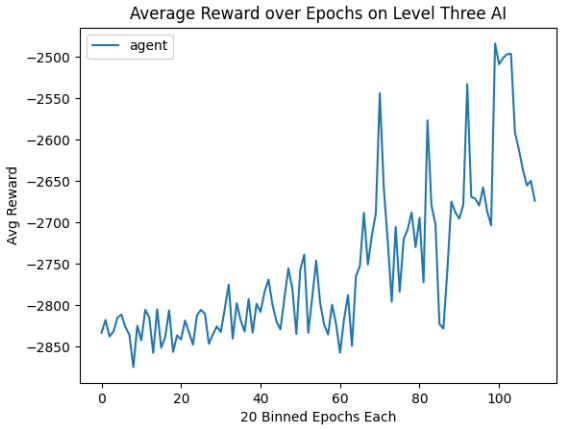
$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

Results

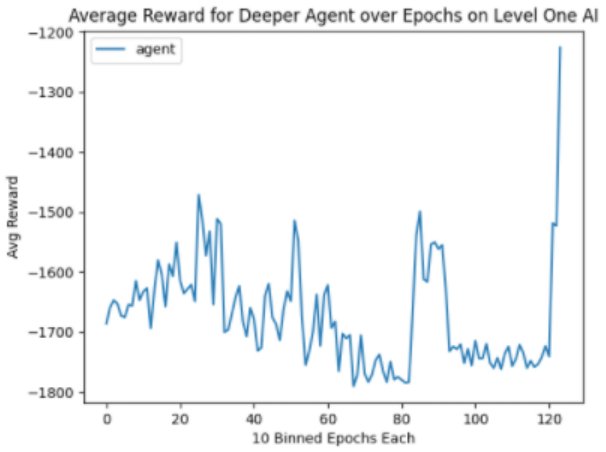
The results of the study were mixed. There were a lot of issues in the environment that required long setups, along with random, unexplained crashing and stops in the program. This hindered the ability to train effectively. However, many of these problems were alleviated by writing a Django server that would handle the training on a separate computer. Contact was established with the creator of the libmelee library (altf4) that had done some similar things with super smash, building an AI that predicted the percentage chance a player would win. His insight allowed me to know a little more of how to build a server that would do asynchronous learning and send back the results after learning, thereby eliminating some of the issues of the emulator timing out because it couldn't predict fast enough/train fast enough. Additionally, the server was ran on two separate computers to not crash the computer. With the nature of Django, the model had to be reloaded every time, and with many threads being spawned to POST and GET the requests, it would begin to take more RAM than was available. There were several times where the models would exceed 16 GBs, thereby crashing the computer. The Django server was moved to a separate server to a computer with 2 processors and 128 GB of RAM and sent the information to it through an SSH tunnel. This proved to be much more effective than before and allowed for good training.

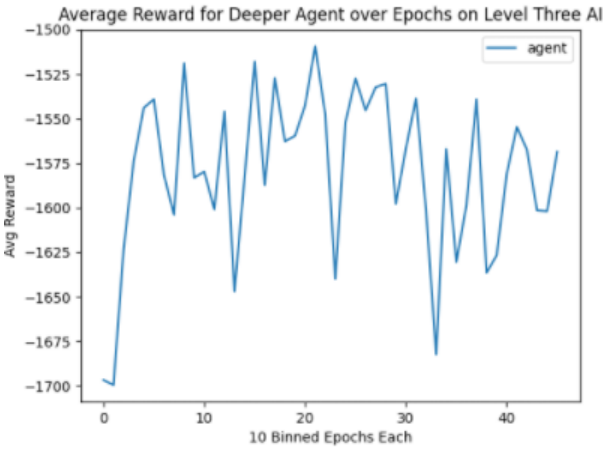
Next, after those problems were alleviated, training against the level one bot. This training took a very long time. To train to fight against the level one bot took around 3 days each time, with sometimes the agent not training very effectively against it, thus causing the need to just restart the training from scratch. After attaining a satisfactory agent, which is represented by the graph below, those weights were used to train against the level three bot. This is where many problems arose with the model trained previously. The agent that had trained against the level one bot had learned to use the same move over and over since the level one bot would just walk slowly towards it, then punch. Thus, if the agent just used a move over and over, the bot couldn't actually hit it. However, this isn't the case for the level three bot, as it is a little smarter. Thus, the agent was punished for everything it learned from the previous model and only began improving after it overwrote and changed how it was playing again.





This study was repeated once more, with a little quicker decay to alleviate the week long training period this previous experiment required and is currently being trained and I will have results by Friday.





Summary

Setting up the environment was incredibly tricky, with a lot of boiler plate code to even get the system running. However, even with the limited system that was implemented here, signs of transfer learning were apparent. The graphs showed that the agent began its average rewards a little higher than that of the original training of the agent from scratch. This at least helped a little bit in speeding up the agent. Additionally,

the fact that the agent was even able to defeat a level one bot was exciting to see the possibility. This showed that it was indeed learning and with more time and potentially a deeper network, it could learn to better exploit and defeat the built in bots.

Conclusions, Limitations and Future Work

Super Smash seems to be more complicated than a smaller network can train on. It seems the best solution to this problem will be to make a much deeper network and allow for long training time. However, the concept of the transfer learning in video games, specifically Super Smash, seems to be plausible, it just requires much more time to train on than most games. More training time and a larger network with potentially a different type of reinforcement learning algorithm could potentially help improve the agent. Implementing an Actor Critic or a PPO instead of DeepQ has been shown to converge quicker with better results, and could be one good route of improving the system developed here. Another improvement to such a complicated game is to have the agent originally train by watching games of a professional. The agent could take the state of the game, determine a move, and contrast it with the move that the professional chose, and once it trained to act like the professional, it then could train and improve on itself. This would allow the agent to skip a lot of the initial and long training that has to be done just to get it to understand the goal of killing its opponent. Additionally, rewards for individual moves can be implemented to bias the agent towards certain moves over others. This could add to the stylistic play of the agent itself.

Sources

https://www.freecodecamp.org/news/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8/
https://en.wikipedia.org/wiki/Q-learning
https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action
https://www.geeksforgeeks.org/sarsa-reinforcement-learning/
https://medium.com/swlh/introduction-to-reinforcement-learning-coding-sarsa-part-4-2d64d6e37617
https://libmelee.readthedocs.io/en/latest/
https://github.com/altf4/libmelee

This paper will be published on Github at: https://github.com/sonorousduck/DjangoSuperSmashServer and: https://github.com/sonorousduck/SuperSmashBot