Capstone Project

Dillon Connolly

Machine Learning Engineer Nanodegree

December 23th, 2017

Definition

Project Overview

As there are many different literary genres, there also exists many different video game genres. Some of the competitive genres include first person shooters (FPS), fighting, multiplayer online battle arena (MOBA) and real-time strategy (RTS). All competitive games share a common flaw. They all include a system for players to learn the game while playing against computer controlled players frequently referred to as bots. As the players become accustomed to the game, the bots are no longer a challenge and are neglected for the longevity of the game. Often times as an attempt at a solution to overcome this issue, bots will include various difficulty levels. As the bots difficulty level increases, the delay between their inputs decreases and sometimes they feel like they are reading the player's input commands. This leaves players feeling cheated or forces them to come up with strategies that are only practical against a bot.

It has been shown that some of the strongest bots can beat the world's best in other competitive games. In this project I attempt to apply a Deep Q Network with Double Learning and Prioritized Experience Replay to the fighting game Tekken 7 to try to create a more human like bot that responds solely off of visual input.

Problem Statement

I would like to create a bot that is able to continue to challenge Tekken 7 players in a constructive way. Currently there is no alternative to playing online against other human players if a player is seeking to improve their skill level. The only way to get better is to play against other players of similar skill level. By using a Deep Q Network with Double Learning and Prioritized Experience Replay I wish to create a bot that learns solely from visual input. A bot that can effectively play from visual input alone is the closest thing to a real player that a gamer can get. Having more difficult bots will enable a player to raise their skill ceiling and get better at the game without playing online.

Metrics

The metrics for this problem are very straight forward. Seeing how the agent performs against myself will be a strong indicator of the agent's success with learning how to play the game. I have no prior Tekken 7 experience. I actually bought the game specifically for this project. I will also compare the agent against the computer controlled bots to see how it fares against them. This will be presented later on as a Win:Loss ratio.

Analysis

Data Exploration

The data used for this experiment was all extracted from the environment I created. The environment consists of five main components. I will start with some definitions of these key components so that my environment is clearly conveyed.

My laptop:

> Consists of 32GB of ram, Intel i7-6700k CPU, Nvidia GTX 980M GPU, and running Windows 10 for the test.

Playstation 4:

> Playstation 4 (PS4) was on the same wired home network as my laptop. Tekken 7 was ran on the PS4 to offload the computing power from my laptop.

PS4 Remote Play:

> Software that was written by Sony to allow players to connect a PS4 controller to their computer and play remotely. The software creates a window on the user's computer where they are able to see the video output from the PS4.

REM4P:

> Software written by TMACDEV that intercepts the PS4 Remote Play software to allow for keyboard input instead of controller input. When this software is active, users can bind buttons from the controller to the keyboard.

Python script:

> Part of my python script handles latching onto the PS4 Remote Play window for keyboard inputs and also for recording the screen. The latter allows for the agent to receive visual input of the game to be used as the state and also to calculate the reward

the agent should receive. The former is used for the agent to produce keypress outputs that can be interpreted by the PS4.

To extract the inputs I use a combination of python libraries mss and pillow to get images of the game window. The images are used by the model to calculate the reward and the current state. Further elaboration on how these were calculated can be found in the Data Preprocessing section. For my experiment I had planned on changing characters but because there are 81 potential outputs and many states, I decided to keep it simple. This meant only using the same 2 characters and remaining on the same map for the entirety of the learning. I did this as an attempt to reduce the total amount of data the agent needed to process.

Exploratory Visualization

Fig. 1 Agent on the left vs the computer controlled bot on the right

Screen captures such as this are what is used as input data. With 36 playable characters and 21 different stages, Tekken 7 has a large variety of input data for an AI controlled combatant. To try to simplify this, I only used the two fighters featured above and limited my training to the stage they are currently shown in. When testing the agent it still performed the same on other stages which lead me to believe that the algorithm was not able to create an actual strategy. More on this later.

Algorithms and Techniques

After much research, I came across an approach to my problem that is an improvement of the approach used by DeepMind to play Atari games. This new approach is called Double Learning Deep Q Network with Prioritized Experience Replay. The algorithm, originally proposed by Hado van Hasselt, is unique in that it uses two independently learned Q functions to improve the stability of Deep Q Learning. In theory, this should enable it to reduce over optimization.
I have chosen this algorithm over other approaches such as Asynchronous Actor Critic Agents (A3C) because of the nature of my environment. Firstly, PS4 Remote Play prohibits two players from playing at the same time on one computer. Secondly, I only have 1 PS4 so I am unable to run multiple instances of my agent in parallel. I inadvertently limited the approaches available to me by choosing a project that relies on hardware.
The Prioritized Experience Replay aspect of the algorithm is supposed to help the model learn more rapidly by providing a recall. It works as a memory cache for the agent to look back on when the presented situation differs from the expected situation.

Benchmark

Due to the nature of a fighting game it is actually very straightforward to benchmark the agent. I will simply load the model after it has been trained and compete against it. After each match I will record the results and present a table showing the total rounds and victor of each round. In Tekken 7 each match is best of three. Although I do have experience in various other fighting games, I have very little experience with Tekken 7. At the time of writing this, I have played Tekken 7 for less than one hour. If I am able to consistently beat the agent then it is proof that it either A) the agent did not learn how to play; or B) it only learned how to exploit a weakness in the computer controller bots.

Methodology

Data Preprocessing

States and rewards are captured using the mss and pillow libraries. Whenever a state is captured, the resultant image is downsized with pillow using the Lanczos algorithm to an 84x84 pixel image and then converted into a numpy array. Before being used as a state, it is first converted to grayscale using a commonly used dot product equation:

$$np.dot(img[..., :3], [0.299, 0.587, 0.114])$$

Fig. 1 Agent on the left vs the computer controlled bot on the right

Fig. 2 How the image looks after processing

This reduces the total amount of input variables for the agent to interpret.

To calculate the reward, I wrote a function that focuses on the part of the window that contains health bars. It simply stores a previous health image and then subtracts it from the current health image. If the opponent's health has changed (ie.There exists at least 10 pixel values higher than 125), the agent receives a +1 reward. If the agent's health has changed then the agent receives a -1 reward. If both health values change in an exchange, the agent will receive a reward of 0.

Fig 3. Example of how the reward is calculated. If agent is left this would yield a +1 reward.

Implementation

This project was implemented using the Atom IDE. By writing all the code in Atom I was able to reduce the memory requirements that would have come with running jupyter notebook in chrome while my model was running. Instead, I closed out of all programs besides the ones that were necessary to run the program and executed it from the command line. To better illustrate the design of the program, I will list out the steps and provide additional details as required.

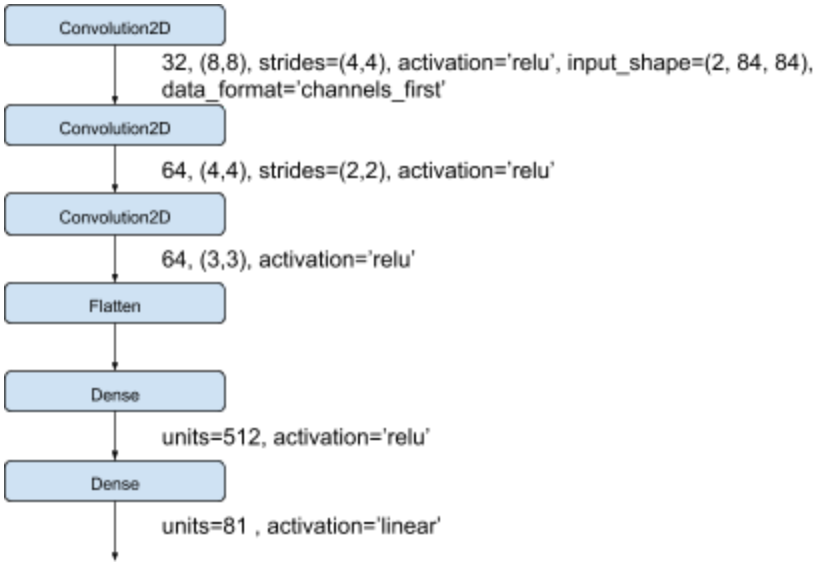
1. The REM4P application is opened and a dualshock 4 controller is connected to the computer via a micro usb to usb type A adapter.
2. The executable is selected from the drop down and “Launch” is pressed. This allows REM4P to latch onto the PS4 Remote Play process for keyboard inputs.
3. The PS4 Remote Play application remotely turns on the Playstation 4 and I navigate to the offline practice mode in Tekken 7 where there is no time limit on matches. The window is placed in the top left corner of the screen so that it is easier to capture what is happening in game.
4. Command line is opened and the python script is ran.
5. Upon being ran, the python script creates a new learning agent. The agent has the following default parameters:
    a. **Input_handler:** An object to the InputHandler class. This allows the agent to execute keyboard actions.
    b. **Learning:** Whether the agent is learning or not. Defaults to False but I set it to True in the parameters when declaring the agent.
    c. **inputShape:** A tuple of (image_stack, image_width, image_height); (2, 84, 84).
    d. **numActions:** The length of the valid_actions array. There are 9 possible directional inputs and 9 possible button inputs for a total of 81 possible actions. They are declared as:

Fig 4. REM4P application.

```
# available inputs by type
direction = [0, DPAD_UP, DPAD_DOWN, DPAD_LEFT, DPAD_RIGHT, DIAG_DOWN_LEFT, DIAG_DOWN_RIGHT, DIAG_UP_LEFT, DIAG_UP_RIGHT]
attack = [0, TRIANGLE, CIRCLE, CROSS, SQUARE, CROSS_SQUARE, CROSS_CIRCLE, SQUARE_TRIANGLE, TRIANGLE_CIRCLE]
```

Fig. 5 Output arrays.

In these arrays, 0 means do nothing. The attack actions that are a combination of 2 buttons correspond to a tuple of pressing both buttons. In Tekken 7, two buttons can be pressed to execute a different move than each button does on its own.



   e. **Model:**
   f. **Memory:** The memory object. It is implemented using a SumTree data structure for O(log n) sampling and O(1) insertion and update.
6. Next the agent imports a previously saved model if it is available in the folder the script resides in. Tekken 7 kicks the player out of the practice mode after a certain time frame so tests were done in 12 hour bursts. After 12 hours, the model was backed up and then the weights were reloaded into the agent for continued testing.
7. Now the agent runs.
8. When running, the agent goes through some preliminary set up steps.
   a. First, the agent activates REM4P. To do this, the agent simply presses the Ctrl + Alt key and REM4P takes over the keyboard handler. Any keys that are mapped to controller buttons will be interpreted as controller input to PS4 Remote Play.
   b. Now a vision object is created with the parameter 'left' meaning that the agent is the left health bar. Vision handles getting the current screen as well as determining the reward the agent should receive.
   c. The vision object grabs the current screen, and creates the preliminary state array. This contains 2 copies of the current screen for the initial 2 stack image. In the deep mind study they used a stack of 4 images but I chose not to in favor of increasing the memory size. I am limited to 32gb of ram.
   d. Next, some variables are initialized. They are the **start_time** (used for tracking episodes), **rewardTotal**, **episode** (The current episode number), **reward_per_episode** (tuple to keep track of episode, reward, steps and Q value), and **i** (used to maintain reward_per_episode index).
   e. Lastly the agent executes the input_handler function to reset the players. While in the practice mode of Tekken 7, when the user presses Touchpad + Cross it resets the players to the starting point in the middle of the arena.
9. Now the main loop for the program starts. The agent chooses an action based on the state. The action is then executed. Immediately after execution, the screen is captured and reward is recorded. The new screen capture is stored in a **statePrime** numpy array

that contains the previous state and the new state. The agent updates its models using the state, action, reward and statePrime. The agent updates its prioritized experience replay tree.

10. Once per minute, data from the currently running episode is stored in a numpy array to be recorded at the end of the test.
11. After 12 hours or 720 episodes, the model and target model are saved to .h5 files. The tuple containing information recorded from each episode is saved to a text file.
12. To continue training, the models are first copied to a backup folder. Then I manually go back to character select and re-enter the game. I found this to be a quick solution to keep the game from exiting on its own to the title screen while I am away from my computer. Now I just re-run the python script and it starts over with the previously trained models.

Refinement

For the most part I utilized the algorithm posted by Jaromír Janisch on his blog. The only parameters I changed from his model implementation were the maximum memory capacity for the prioritized experience replay, the exploration stop value and the gamma value for DDQN. I doubled the maximum memory capacity from 200,000 to 400,000. This resulted in my model using roughly double the amount of RAM than the referenced model. My reasoning for this was that in Tekken 7 there are so many more potential states that having a much larger experience replay will be beneficial to the agent. The exploration stop value was adjusted to reflect the amount of time the agent would be running for. This proved to not really have an effect on the outcome of the agent. The gamma value also proved to not really have an effect on the outcome of the agent. From my experience working with convolutional neural networks, I found that it is very difficult to create a model from scratch and get great performance. I felt that the other parameters he proposed were sound and would work well with my very limited hardware.

Results

Model Evaluation and Validation

The model chosen is a technique proposed by Hado van Hasselt called Double Deep Q-learning. It consists of using two independently learned models to reduce maximization bias that normally arises from the agent overestimating just a single model. This solution has been shown to outperform the DQN algorithm I originally intended to use in other applications. When the models are combined, their function is as follows:

$$Q(s,a) \rightarrow r + \gamma \widehat{Q}(s', argmax_a Q(s',a))$$

As an attempt to simplify the problem, I had the agent play the same character throughout all learning and tests. The agent played the same against all opponents and on all maps. This result shows that the agent did not learn how to maximize the reward. Between a 60 hour trial, 4 36 hour trials and a 27 hour trial, all agents yielded similar results. The Q values and rewards hovered around the x axis instead of increasing.

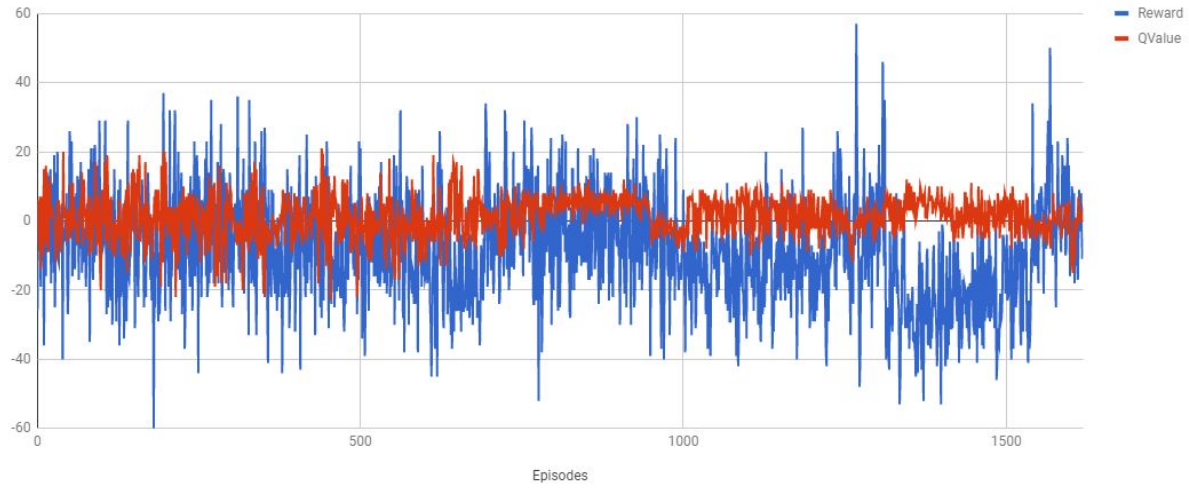


Fig 5. A graph showing the Qvalue and Reward over 1619 episodes or 27 hours.

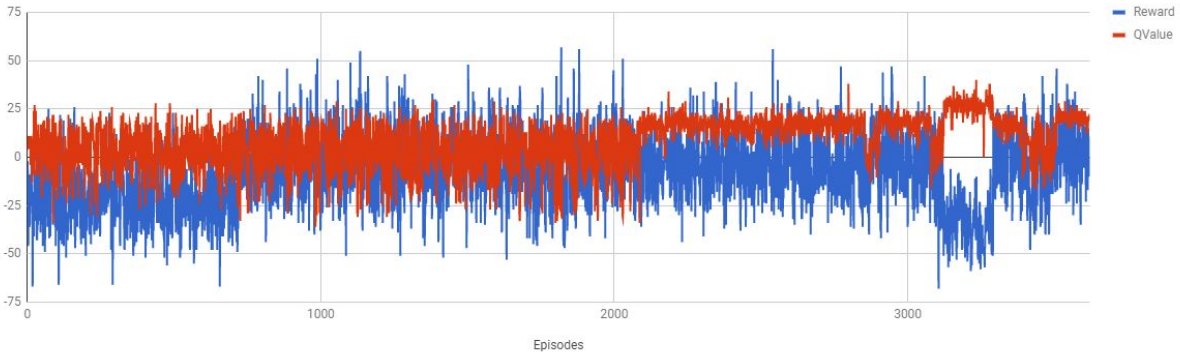


Fig 6. A graph showing the Qvalue and Reward over 3620 episodes or 60 hours.

Justification

To test the agent, I played it against the arcade mode bots and against myself. The results in arcade mode highlighted the design of the game pretty well. There are three matches where the bots are very low skill level and easy to beat. The fourth match serves as a wall to try to stop players from continuing in the arcade mode. The agent was able to win all three matches up until the fourth where it consistently lost every time. When playing against myself, the agent was able to win twice. Once I learned the agent's attack patterns, I was able to beat it consistently

with little to no challenge. The agent's main strategy is to spam the same move consistently which is very easy to play against. The strategy I chose was to wait for the moment of vulnerability between the attacks before striking. I do not claim to have solved the problem. I think with my limited set up and algorithm choice it would take many more hours and a much larger experience replay to create an agent that is able to consistently defeat a human player. The other obstacle is through remote play there is no way to have two agents fight each other. Having two agents learn against each other is one of the main strategies employed to expedite the learning process.

Conclusion

Free-Form Visualization

Fig 7. Demonstration of how easy it is to outplay the agent.

Fig 8. The agent lands a successful grab.

These two gifs demonstrate the agent running against myself. You can see in Figure 7 that for a few frames I am just standing still while the agent continually spams the grab button. I then counter in between the agent's attacks with a kick combo. In each test I conducted, the agent ultimately wound up choosing one button combination and spamming it. In Figure 8 I allow the agent to land a grab on myself to demonstrate what the agent is trying to achieve. When playing against the in game bots, the agent can actually have some success when enforcing this strategy. If it were to be trained against another agent I'm sure it would utilize a much more advanced strategy.

Game	Winner
1	Agent
2	Me
3	Me
4	Me
5	Agent
6	Me
7	Me
8	Me
9	Me
10	Me
11	Me
12	Me
13	Me
14	Me
15	Me
16	Me
17	Me
18	Me
19	Me

20	Me

Fig 9. Table showing the win:loss ratio I recorded when fighting the agent

Reflection

The steps I took to try to solve this problem can be summarized as follows:

1. REM4P and PS4 Remote play applications were found and installed.
2. Purchased Tekken 7.
3. Develop code to capture screen data and process rewards.
4. Develop code to handle key presses that are mapped to controller buttons.
5. Implement DDQN agent with PER to play Tekken 7.
6. Connect a Dualshock 4 controller to the computer and then open REM4P and use it to launch the PS4 Remote Play application.
7. Start Tekken 7 and navigate to Offline > Practice mode. Next select the 2 fighters.
8. Once the game is loaded, press the “Options” button on the controller to pause the game and see the options menu. Set the computer player to hard difficulty and then press the “Options” button to resume the game.
9. In command prompt, start the agent.py file. It will automatically switch to the Remote Play window and start pressing input commands.
    a. *Note: Make sure before you start the agent.py you set the TRIAL, TOTAL_TESTS and TOTAL_EPISODES parameters. These will directly affect the duration the agent plays for and the naming of the files when the program saves the models and game data.

This project was definitely very eye opening to the reality of long training times. It’s hard to imagine how much work goes into training an advanced algorithm without experiencing it for yourself. The most difficult aspect of this project was dealing with network issues and not being able to directly access game data. Occasionally the network would hiccup and the test would have to be abandoned because my laptop would lose connection to my PS4 while I was away. My approach to the fighting game problem would have been much different if I could use the actual numbers from within the game. For example, basing the reward off of a number relating to the actual integer amount of hit points that were lost rather than using pictures. I thought that using the DDQN model my agent would become an unstoppable force but I did not think that it would actually never converge on a solution. I feel that using a simpler game that can be run at a faster clock rate on modern technology would yield better results with a DDQN network.

Improvement

I think the best way to improve this solution would be to switch to an older fighting game. That way the game could be emulated and the game memory could potentially be accessed. I would also lean more towards implementing an Asynchronous Actor Critic Agents (A3C) model instead of a Double Deep Q Neural Network. With A3C I could have n instances of the emulator running

with agents running on separate threads. All of the agents could then report their experience to a central node that uses gradient descent to choose and redistribute the best weights. By pitting the agents against each other, they would be forced to come up with more complicated strategies to win and would converge much more rapidly. I'm confident that with an approach like A3C the resultant agent would annihilate my current benchmark model. I was not able to implement an A3C model because I do not have multiple PS4's to parallelize the game. Not to mention only 1 player can connect through remote play at a time. These two factors eliminate the ability to have an agent fight against another agent.

References

[1] “Huber loss.” *Wikipedia*, Wikimedia Foundation, 7 Nov. 2017, en.wikipedia.org/wiki/Huber_loss.

[2] “Deep Convolutional Q-Learning.” *Deep Robotics*, 7 Aug. 2016, vmayoral.github.io/robots,/ai,/deep/learning,/rl,/reinforcement/learning/2016/08/07/deep-convolutional-q-learning/#convrl.

[3] Firoiu, Vlad, et al. “Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning.” *Cornell University Library*, 8 May 2017, arxiv.org/pdf/1702.06230.pdf.

[4] Fiszel, Ruben. “Reinforcement Learning and DQN, learning to play from pixels.” *Ruben Fiszel's website*, 24 Aug. 2016, rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html.

[5] Janisch, Jaromír. “Let's make a DQN: Double Learning and Prioritized Experience Replay.” *ヤロミル*, 16 Feb. 2017, jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/.

[6] Lau, Ben. “Using Keras and Deep Q-Network to Play FlappyBird.” *Ben Lau*, 10 July 2016, yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html?n=2.

[7] Mnih, Volodymyr, et al. “Playing Atari with Deep Reinforcement Learning.” *Cornell University Library*, DeepMind Technologies, 19 Dec. 2013, arxiv.org/abs/1312.5602.