Game AI - IDG5301 - Project Report

Development of a Q-Learning and a visual Deep-Q-Learning agent for the Fighting Game competition DareFightingICE

Emil Dobetsberger
Digital Games MSc Course
University of Malta
Msida, Malta
emil.dobetsberger@gmail.com

***Abstract*—This report documents the development of two agents for the fighting game competition DareFightingICE, held on https://www.ice.ci.ritsumei.ac.jp/ ftgaic/. Both agents are based on reinforcement learning algorithms, the first one using tabular Q-Learning, and the second one Deep Q-Learning. The agents were trained and tested against a provided Monte-Carlo-Tree-Search algorithm, which they managed to surpass in several occasions.**

***Index Terms*—Q-Learning, Deep-Q-Learning, reinforcement learning, deep reinforcement learning, neural network, FightingICE, fighting game**

I. Introduction

FightingICE [1], [2] is an environment for developing artificial intelligence (AI) agents playing fighting games. The environment, developed in Java, provides interfaces for creating agents both in Java and in Python. For this paper, two agents were written in Python and their performance was measured against a provided agent that utilizes a Monte-Carlo-Tree-Search (MCTS) algorithm. The game inside the environment is completely symmetric [8], featuring exclusively one-versus-one combat, with both players starting out at 400 health points, and 0 energy points. Each game is played over 3 rounds, each round lasting either 60 seconds or until one of the agents has lost all health points. The player with the most health points left wins. There are, in total, 37 possible actions, some of which require energy. Energy is accumulated both through hitting the enemy and getting hit, the latter generally yielding more energy than the former. The provided MCTS agent demonstrates very goal-oriented behavior but seems to perform differently depending on the hardware it is run on. Although the competition asks for fighting game AI that is strong against any opponent and as both player 1 or 2, this project has been focused exclusively on being Player 1 and fighting the provided MCTS agent, which controls player 2. Via the interfaces, agents receive updates about the game state with a delay of 15 frames (amounting to one-fourth of a second), while limited visual data of the game is available in real-time, in the form of a 32x32 image of the game state in two different versions.

II. Q-Learning algorithm

The first agent that was developed, is based on tabular Q-learning, based on a tutorial by Abid Ali Awan [3].

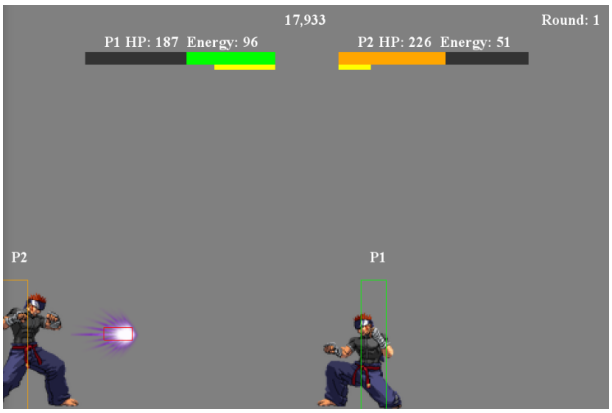

Fig. 1. The FightingICE environment

A. Description of Approach

The Q-Table of the Q-Learning agent has an entry for each action in a limited state-space. At first, the action space was defined as all possible keyboard inputs (7 keys are used in the game), although this technique was quickly discarded, as the system allows for selecting actions made up of key combinations directly, the performance of which would be difficult to learn for the AI otherwise. With key presses as actions, it would also have been impossible for the AI to learn to use the special action, which requires a high amount of energy and a combination of three keys (one key having no other use than being part of this action), which leads it to not be explored much by the AI. Hence, the action space was defined as all 37 named actions that the environment supports. Even though some of the actions are not possible in certain states, and thus some entries of the table will never be explored, this table structure was decided for the sake of simplicity. The state is composed of the vector from the player position to the enemy position, player energy, as well as enemy energy. The positional vector consists of the y-component in very coarse quantization and the x-component in more precise logarithmic quantization, which is centered around 0 (see figure 2). The current health points were not included in the state, as they have a lesser impact on the gameplay situation, and a small state space was desirable for training efficiency. After later development, it was decided to also

include the current player state and enemy state (”Standing”, ”Air”, ”Crouch”, ”Down”), because the possible actions and the hitbox size depend on it.

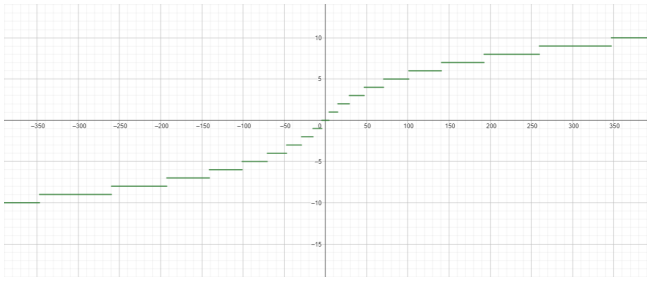

Fig. 2. Logarithmic quantization function used for positional x-component

Due to the delay in the provided frame data, it was discovered that the agent can get stuck in a position, specifically in ”Crouch”, since after a single ”Stand” action, the agent would still think they are in ”Crouch”, and continue with more crouching actions. To fix this behavior, a policy was added, that, upon changing stance, makes the agent repeat the stance-changing action (possibly ”Idle”) until the new stance was reflected in the state retrieved from the system. This method was then also used to force the ”Forward Walk” action to be called at least twice, since only walking forward for one frame would have little to no impact on player position.

The values of the Q-Table, which contains the expected rewards when taking a specific action in a certain state, are initialized at 0, and are updated over the course of the training episodes, an episode being one round of a game. With the exploration-exploitation principle [9] the agent then decides, whether to perform the best action in the current state (the action with the highest reward according to its Q-Table), or a random action to gather new information. Before taking any action, the agent is given a reward for the current game state, that it can relate to its last action taken. The rewards are defined by a reward function that will be described later on.

B. Algorithm Parameters

The training parameters for the algorithm were initially taken from the tutorial [3]. Some adjustments were made after more experience was gained. First of all, the training parameter $learning\_rate$ is suggested at 0.7 but was lowered to 0.1, since an episode in the fighting game is much longer than in the example, and the agent thus benefits from learning slower over time.

The environment parameter $\gamma = 0.95$ was left as the tutorial suggests. It defines how much future rewards influence the reward in a state. If the environment might yield rewards far in the future, and the agent should therefore have non-immediate rewards in mind, a higher value is beneficial.

The exploration parameters ε_{max} and ε_{min} were set at 1.0 and 0.05, such that in the beginning all actions are random, but the more training the agent undergoes, the smaller the ε value becomes until it reaches a minimum, that still allows for a small degree of randomness. The parameter $decay\_rate$ defines how fast the ε value decays, i.e. how random actions decrease over episodes. Since the simulation of an episode in our environment takes much longer than in the example, this value had to be adjusted, such that at least after 300 episodes, some results of the training would become evident, to assess the performance of the agent. Hence, the decay rate was set at 0.005. This value was chosen, such that a high enough portion of the decisions of the agent was made by evaluating the Q-Table, i.e. that after 300 episodes less than 30 percent of decisions were made at random. The behavior of the variable ε with these parameters can be seen in figure 3.

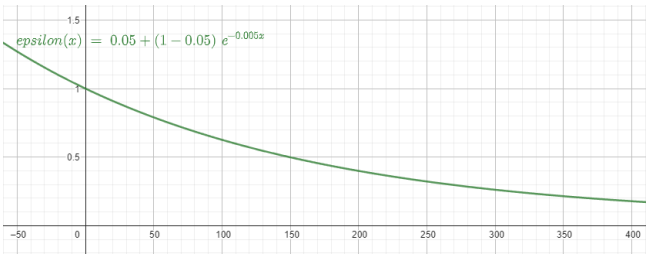

Fig. 3. The epsilon function defines the portion of random decisions after training a certain number of episodes

For some quick experiments, where the agent was only going to be trained for 100 episodes, a decay rate of 0.01 proved useful, as after training the agent would already base 60 percent of their decisions on their training. As the rewards are less reliable with less training, a higher portion of random decisions is reasonable. After some initial tests on lower-end hardware, the performance of the agent was empirically satisfactory, such that these values were not tweaked further.

C. Performance measure of the algorithm

For evaluating the performance of the agent, the difference between the player health points and the enemy health points at the end of each round was logged. The higher the difference, the better the performance of the agent. As the results of 300 rounds would generally be very noisy, the average of each game (3 rounds) was first calculated and plotted, and then the average of 12 games would be calculated as a moving average across the game averages. The behavior of this curve gives information about whether the agent was improving its performance over time, or if its performance was getting worse. The data of the training of the final agent, treated in this manner, is shown in figure 4.

D. Experiments

The agent was tested against the MCTS agent, as well as an idle agent, and an agent that would only spam the ”Kick” action. Due to the environment not providing more sophisticated algorithms, the MCTS agent was generally regarded as the benchmark, and the other agents were only tested to see if the agent’s training was effective. Training first against the ”less sophisticated” agents, and then against MCTS tended to worsen the performance of the agent, and thus future training would only be carried out against MCTS. Before and after

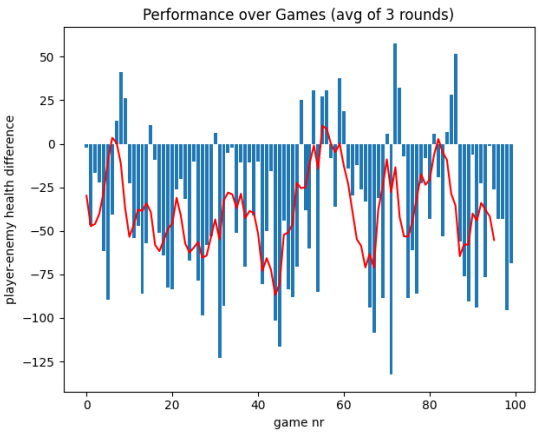

Fig. 4. Performance of the Q-Learning agent over training games

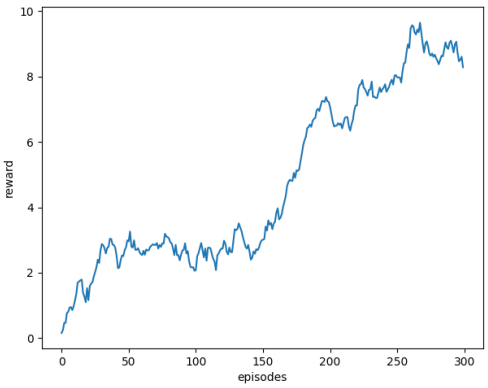

Fig. 5. Development of the cumulative episodic reward over episodes

the changes described in the previous sections, the agent was trained over usually 300 episodes against MCTS, and after training, an additional evaluation was made, for 10 games (30 episodes) where no further learning was gathered. Observing the win-loss ratio of the agents during this evaluation, the empirically more promising agent was then taken and developed further, until the final version.

The experiments were mostly used to decide on the best reward function. In the first iteration, the reward was only taken as the missing health of the enemy, then was changed to the missing health of the enemy minus the missing health of the player, which makes the agent more likely to defend from enemy actions, although not necessarily improving the end result. It was observed, however, that the agent would often prefer to use actions with a low or medium energy cost, and never reach energy levels high enough for expensive special actions. For this reason, in another attempt, the reward was changed to include a factor for the energy the player currently stored.

While not only showing improved results, this agent was found to use the special action (fireball) with a frequency that was about 25 percent higher than previous agents. The way the received reward develops over time and episodes can be seen in the following figure 5.

After training the final agent for 100 games against the MCTS agent on the best hardware available (AMD Ryzen 9 3900XT, Nvidia GeForce RTX 3070), out of which it ended up winning 20, the final evaluation in 10 games with no further training led to a result in equal proportion of 2 wins and 8 losses against MCTS.

E. Conclusion

While the agent can be seen learning and improving to some extent, one important task that it could not perform well, and which is probably the reason for its low success, is dodging projectiles. As the presence of projectiles and their positions are not included in the state space, the agent has no way of knowing about and taking countermeasures against an approaching projectile. This issue is difficult to tackle without largely increasing the state space, however, the problem could be solved by simply hardcoding the behavior in the presence of powerful projectiles. Since the projectile spawned by the ”Special Action” (a fireball) deals vast amounts of damage, and is often round-decisive, dodging it should always be a high priority, and heuristic logic might be perfectly sufficient to achieve this goal.

Another approach to better assess the quality of learning of the agent would be to disable projectiles completely for all players. Although this would require alterations to the environment, this can be taken into account for further research, as in our environment, round results often depended on whether a fireball was cast or not.

Finally, although training showed some satisfactory improvements in behavior, it was also not further explored, how the agent would evolve over extended periods of training (thousands or tens of thousands of episodes), as this was not possible in this project due to time and resource constraints.

III. Deep Q-Learning

The second agent developed is based on Deep Reinforcement Learning, based on several tutorials online [7], [4].

A. Description of Approach

The Deep Q-Learning agent utilizes a Convolutional Neural Network (CNN) implemented with Pytorch [6] that calculates the expected rewards from a game state given as an image. The image provided by the system is a 32x32 downscale of the actual game, provided in two different versions as a 64x32 RGB byte buffer, see figure 6. The neural network then utilizes three convolutional layers, a max-pool layer, and two linear layers to convert the input image into float values that describe the goodness of each action. The values for this network were largely adapted from the ”Deep Q-Learning” tutorial from the Pytorch website [7].

The Deep Q-Learning agent uses two instances of this neural network, where one is updated continuously (the source network) and the other only copies the values from the source network at certain intervals (the target network). Additionally, it uses a replay memory, which serves as a storage of a batch of experiences. This technique is common of Deep Q-Learning

implementations [7], [9]. The expected rewards are calculated using an "AdamW" optimizer [5], utilizing the same reward function as described for the Q-Learning agent.

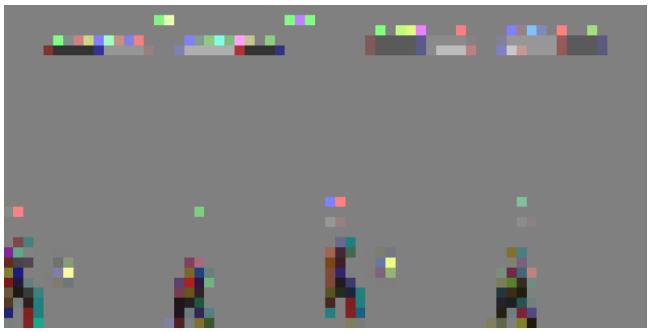

Fig. 6. The image data that the environment provides for agents

B. Algorithm Parameters

The Deep Q-Learning agent, similar to the tabular Q-Learning agent, utilizes an exploration-exploitation strategy [9] that switches between taking random actions and the best actions in the current state. The parameters for this were taken from the tutorial [7] with $\gamma = 0.99$, $\varepsilon_{max} = 0.9$ and $\varepsilon_{min} = 0.05$. It should be noted that even in the beginning the agent does not choose all actions at random with these values.

The decay rate was alternated between 0.01 and 0.05 in the same fashion as for the first agent. There are three other parameters that serve different functions, which were not present in the first agent. First of all, τ is the update rate for the target network. It was set to 0.005, the same as in the aforementioned tutorial. Another parameter called $learning\_rate$, which defines how fast the "AdamW" optimizer learns, was set to 0.0001. Finally, the $batch\_size$ defines, how many transitions are sampled from the replay buffer each time the network is optimized. As with the other parameters, the batch size of 128 has been left the same as in the tutorial.

C. Performance measure of the algorithm

In the same way as with the first agent, performance was measured in terms of the difference between the health of the agent and its opponent. As the neural network requires more computation, and the environment could barely be calculated faster than real-time with the available hardware, training the agent and observing the results for the assessment of changes was usually only done over 90 episodes. Even though this amount of training often did not yield reliable results, where performative differences were significant, any change that produced an empirically more promising agent was applied, and any other change was discarded. The experimented changes are discussed in the next section.

D. Experiments

The elements that were experimented with, were primarily the size of the network and its input. Due to the high computational cost of the neural network, training was more limited and a decay rate of 0.01 had to be used in most cases, and only 90 episodes of training could be carried out.

At first, simplifying the network was attempted. Instead of three convolutional layers, an attempt was made with just one. Then, the agent was tested again with three convolutional layers, but only with 16, 32, and 64 channels in their output, instead of the initial 32, 64, and 128. In another attempt, instead of 512 neurons in the first linear layer, this was reduced to 256. In a new try, the network was left untouched, but the image was input as greyscale instead of color. All these attempts led to the agent performing worse than in the initial attempt and were thus discarded, as they were empirically not promising. The only test, that gave promising results was providing the network with an input of only one of the two 32x32 images, specifically, the right part of the image seen in figure 6, instead of both. This approach was then used to further train the agent.

Resetting the network, and setting the learning rate down to 0.05, the agent was left to train for several hours, until 300 episodes were completed. While still losing most games, the agent seemed to have improved performance slightly, as the average of the resulting health difference now mostly varied between -100 and 0, while most other agents barely ever achieved an average health difference above -50. In a final evaluation, the training of the agent was disabled, although still allowing for some degree of random actions according to the exploration-exploitation formula [9], and the agent was tested against the MCTS agent in 10 games. Out of these, the agent won 6 and lost 4; a substantial improvement over the 13 games it won during training out of 100, where one ended in a draw, and 86 in a loss.

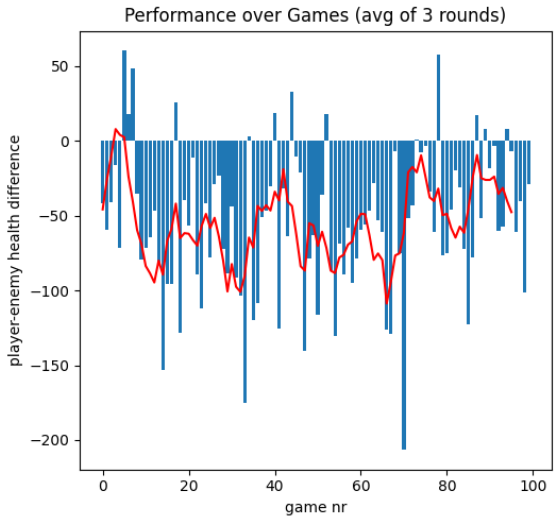

Fig. 7. Performance of the Deep Q-Learning agent over training episodes

E. Conclusion

Although visual differences between the behavior of this agent to the former are rather difficult to observe, the agent was seen successfully dodging the special action (the fireball

projectile) several times. As an (intentional) projectile dodge was not possible for the tabular Q-Learning agent, it is one of the main advantages of the CNN. Computational costs, however, remain very high for training, as most of the attempts that tried to simplify the neural network yielded worse results. Although additional training is suspected to improve results further, testing this was not possible due to time constraints. In future research, the agent could also be tested a higher number of times before the final evaluation, to confirm whether the difference in wins and losses is statistically significant or not.

IV. Discussion

Although Monte-Carlo-Tree-Search-based agents do perform well in this fighting game scenario, this report shows, that writing an agent that beats MCTS is not impossible. One factor that makes MCTS especially problematic is its different performance depending on the hardware it is run on. A pre-trained agent based on tabular Q-Learning is able to perform well at most times but fails when a gameplay situation includes an aspect that is not covered in the agent's Q-Table. Lower computational runtime costs and a higher degree of consistency across hardware are advantages of tabular Q-Learning, but the algorithm is highly susceptible to changes in the environment. If, for example, there were moving obstacles introduced into the environment, or other circumstances that would complicate a functional Q-Table greatly, adapting the agent would hardly be possible. An agent based on a convolutional neural network, however, can improve on this aspect. Although its training is time-consuming, it is believed to adapt well to environmental changes, as it, for example, demonstrated successful dodges of projectiles without the implementation targetting this situation specifically. Further improvement might be gathered through more rigorous training, and expansions of the neural network could also be attempted. Finally, there might be entirely different artificial-intelligence-based approaches, that might lead to even better results, which research yet has to reveal.

References

[1] Fighting Game AI Competition. https://www.ice.ci.ritsumei.ac.jp/~ftgaic/. [Online; accessed 21-Jan-2024].

[2] TeamFightingICE/FightingICE. https://github.com/TeamFightingICE/FightingICE, January 2024. [Online; accessed 21-Jan-2024].

[3] Abid Ali Awan. An Introduction to Q-Learning: A Tutorial For Beginners. https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial. [Online; accessed 21-Jan-2024].

[4] Branko Blagojevic. Learning from pixels and Deep Q-Networks with Keras. https://medium.com/ml-everything/learning-from-pixels-and-deep-q-networks-with-keras-20c5f3a78a0, March 2021. [Online; accessed 21-Jan-2024].

[5] The Linux Foundation. AdamW — PyTorch 2.1 documentation. https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html. [Online; accessed 21-Jan-2024].

[6] The Linux Foundation. PyTorch documentation. https://pytorch.org/docs/stable/index.html. [Online; accessed 21-Jan-2024].

[7] The Linux Foundation. Reinforcement Learning (DQN) Tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. [Online; accessed 21-Jan-2024].

[8] Stefano Gualeni. Game Design: theories, methods, and other lies. Institute of Digital Games, University of Malta, 2023.

[9] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. https://gameaibook.org.