Comparing Single & Double Deep-Q Networks for AI Learning to Play *Galaga*

Braeden Stewart

Utah State University CS 5640 Student

a01976976@usu.edu

Abstract

Successful automated defense against airborne threats requires that it can identify these threats and react accordingly. There is a need for developing learning methods that an Artificial Intelligence agent can use to train to accomplish these goals. This work investigates the efficiency of using Double Dyna-Q Networks vs Dyna-Q Networks to help these agents learn. It uses the environment of the game *Galaga: Demons of Death* which simulates a similar problem against the opposing alien invasion force. In this environment, the agent learns to identify enemy spacecraft as well as the projectiles they fire based off image samples of the environment. The agent also observes the behaviors of said enemy spacecraft. Using this information, the agent determines a course of action to avoid and/or eliminate threats to the spacecraft that the agent is controlling. These studies hope to figure out the optimal architecture between the two neural networks for training agents in charge of Anti-Air defense systems.

General Terms. Convolutional Neural Network (CNN), Deep-Q Network (DQN), Double DQN (DDQN), Artificial Intelligence (AI), Reinforcement Learning (RL)

Introduction

My study involves the use of CNNs, as a DDQN and DQN, to take in images of the game of *Galaga* and returns values that helps the agent playing the game to decide which actions will result in a higher score. This environment mirrors real world issues that Automated Anti-Air (AAA) emplacements face internationally when engaging hostile planes, missiles, and nukes. I knew that I wanted to learn more of how Reinforcement Learning agents behaved in these roles without needing to code an entirely new environment[1] so that I could focus more on studying how Neural Networks work. I was already very familiar with the environment of *Galaga* from competing against my mother for a high score in the game in my youth.

Specifically, the purpose of this research was to observe the differences of exploring and exploiting behavior between a DDQN and DQN agent given the environment of the game. In theory, one of these neural networks would get a consistently higher score than the other indicating its dominance. Hopefully, these small results could be used in relation to larger scope problems like the AAA systems.

[1] OpenAI Gym-Retro provided an emulator to interact with *Galaga* for neural networks.

Methods

Only a basic summary of how CNNs and Dyna-Q Learning work will be given here[2]. a basic knowledge of Reinforcement Learning concepts is also assumed. CNNs are defined by their Convolutional layers which take an input matrix, usually representing a pixel image, this is then used in a dot product calculation between the matrix and a set of learnable parameters called a kernel. The kernel slides across the image in strides, performing these calculations. An activation map is returned that records the response of the kernel at each spatial position of the image (Source 3).

For this problem, the input matrix is made more complex since a single frame from playing *Galaga* is not sufficient to represent the actual state the agent needs to observe and learn from. This is because a single frame does not show the agent the velocity of moving objects in the game. This means that a RL agent will most likely never learn to dodge missiles or enemy spacecraft. It will also not hit moving enemy spacecraft with its own missiles. All these factors result in lower high scores. For this reason, the implemented neural networks stack four frames together so the agent can gleam a velocity from the change of position between frames. This serves as the input for the agent to learn from (source 4). The pixel images are resized to 112 x 120 before being stacked with the others and put into the neural network.

Specifically, the implementation for this project uses a neural network made up of the layers displayed in Figure 1 below. Most of this project was done with the help of the python package of TensorFlow.

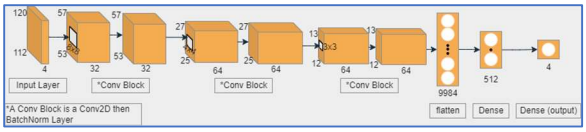

Figure 1: Diagram of Neural Network The white squares are the filters and the rectangles at the end are fully connected layers. A larger image can be found inside of the documentation folders of my GitHub page for the project

Both implementations of DQN and DDQN use the same type of neural network model. A summary of said model's outputs and learnable parameters is displayed below in Figure 2.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 112, 120, 4)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 53, 57, 32)        8224
_________________________________________________________________
batch_normalization (BatchNo (None, 53, 57, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 25, 27, 64)        32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 25, 27, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 13, 64)        36928
_________________________________________________________________
batch_normalization_2 (Batch (None, 12, 13, 64)        256
_________________________________________________________________
flatten (Flatten)            (None, 9984)              0
_________________________________________________________________
dense (Dense)                (None, 512)               5112320
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 2052
=================================================================
Total params: 5,192,996
Trainable params: 5,192,676
Non-trainable params: 320
_________________________________________________________________
```

Figure 2: DQN & DDQN neural network model summary

The output is a shape of four to represent the four actions the agents could take in the environment. An agent can go left, go right, fire a missile, or do nothing.

This study also used the common following RL methods: Target Networks, Skipping Frames, and Experience Replay.

Target networks are nearly identical to the networks they derive from. The target network is updated frequently from the latest network intermittently to fix the parameters of the unstable target function in Temporal

[2] Please see source 3 to learn more about how CNNs operate.

Difference[3] error or loss calculation (source 5).

Skipping Frames is a technique which doesn't require Q-values to be calculated every frame. This harmonizes with earlier requirements of the agent needing at least 4 frames to observe the velocity of moving objects on the screen. This technique also allows the neural network to gain more experiences with less computational cost (source 6).

Experience Replay helps reduce risk by storing experiences that include the actions, rewards, and state transitions which are then used in mini batches to update the neural networks. This should increase learning speed, reduce overfitting by minimizing correlations between experiences, and reuse past state transitions to stop catastrophic forgetting (source 6)[4].

The batch size for the implementation is 16, meaning 16 states are passed into the fit function

The major difference between DDQN and DQN is how they use the Bellman equation[5] to calculate the Q-values. Unlike DQNs, DDQNs use the target network for evaluating the action and the primary network for selecting the action to evaluate. DQNs uses the target network for selecting and evaluating the action (source 7). See figures 3 and 4 below. The array done flags stores a Boolean that determines whether the memory frame was at the end of an episode,

$$labels[i][action\ index] = rewards[i] + (\boldsymbol{not}\ done\ flags[i]) * \gamma * \max(values[i])$$

Figure 3: DQN -Q value function pseudocode

$$labels[i][action\ index] = rewards[i] + (\boldsymbol{not}\ done\ flags[i]) * \gamma * \text{target values[i][argmax (values[i])]}$$

Figure 4:DDQN Q-value function pseudocode

The reward for both agents is based off the score they earn at each step. In *Galaga* points are only earned by destroying enemy ships, but extra points are given for hitting moving enemies. This rewards the player for skillful behavior. The cumulative reward is equal to the score of a single episode or playthrough. For these experiments, *Galaga* ends by losing all available lives, two in this case, or reaching a maximum number of steps taken.

To fit the neural network with the loss function, both agents use the Adam optimizer and the Huber loss function provided by TensorFlow (source 8). The Huber Loss function is depicted below in Figure 5. For each value x in error = y_true – y_pred; d is delta, which is where the loss function changes from a quadratic to a linear function.

```
loss = 0.5 * x^2                  if |x| <= d
loss = 0.5 * d^2 + d * (|x| - d)  if |x| > d
```

Figure 5: Huber Loss function. This was defined at https://www.tensorflow.org/api_docs/python/tf/keras/losses/Huber

In addition, the agents use a greedy-ε algorithm for exploration with an initial ε of 1 and a final ε of 0.01. The exploration rate decays according to the equation below in Figure 6 for every training episode.

[3] Please see source 5 to learn more about Temporal Difference (TD).

[4] Catastrophic Forgetting is when the agent forgets how to do a previously learned task almost completely

[5] Please see source 9 to learn more about the Bellman equation

$$\varepsilon = \varepsilon - \left(\frac{\varepsilon}{\varepsilon_{decay}}\right), \varepsilon > 0.01$$
$$\varepsilon = \varepsilon, \varepsilon \leq 0.01$$

Figure 6: Epsilon decay function where ε_{decay} is equal to 50 for both DQN and DDQN implementations

The learning rate is locked at 0.0025.

For the following experiments, the independent variables were the two types of CNNs, DQN and DDQN. The control variable was the Gym Retro environment for *Galaga*. The dependent variables were the scoring and rewards achieved by each agent which will be used to compare the two later. There are a few changes that were made between experiments which will be detailed in the following paragraphs. The results will be discussed in the next section.

The initial DDQN implementation was planned to be heavily based off George Fidler's implementation provided at source 1. However, their code was dependent on TensorFlow 1.0 which was drastically different from the principles behind the current TensorFlow 2.0 version. Unsuccessfully, I attempted to migrate their code to TensorFlow 2.0 and then decided to write both the DDQN and DQN agents based off implementations for the Cart-Pole[6] and *Pong* environments[7]. This is Version 1 of the implementation.

Version 2 of the implementation changed the speed at which the exploration rate decayed. Referencing Figure 6, the ε_{decay} value was increased from 50 to 500. Future experiments kept this modification.

Version 3 is where the hard time limit of 4000 steps per episode was removed to be replaced with an idle-steps limit of 4000. I defined an idle-step as a step where the agent took an action that changed neither the score or lives values. This was needed to allow the agent to play till the actual end of the game but also still keep it from infinitely looping with strategies that made no progress.

Results

In Version 1, both agents were trained for 36 hours, roughly 900 episodes. The maximum number of total steps allowed per training episode for both was 4000, which gave the agent enough time to beat roughly level one and level two of Galaga. figures 7-10 show the results for both agents. Each order of weight represents a checkpoint, so the score at order of weight 1 is what the agent achieved using the trained weights that the agent had learned after the previous 25 training episodes. The score at order of weight 35 is what the agent achieved using the learned knowledge of the previous 875 training episodes. This way it lines up with the average scores recorded during its training with an exploration rate > 0.

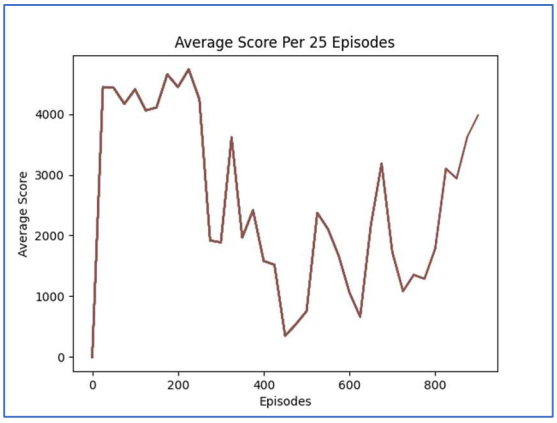

Figure 7: DQN Average Score per 25 episodes with epsilon > 0. Every 25 episodes, the weights of the agent at that moment were saved.

[6] Cart-Pole is a classic control environment provided by OpenAI Gym (source 7)

[7] *Pong* RL agent implementation at source 4

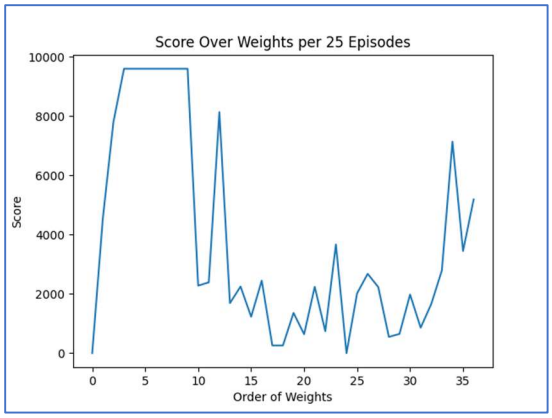

Figure 8: Results of running the DQN agent with epsilon = 0 using the weights it learned after training.

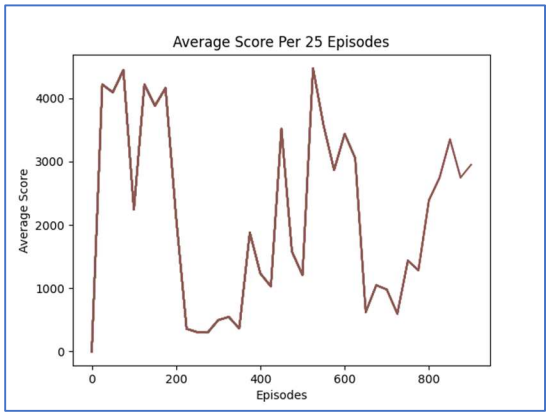

Figure 9: DDQN average score per 25 episodes with epsilon > 0. Every 25 episodes, the weights of the agent at that moment were saved.

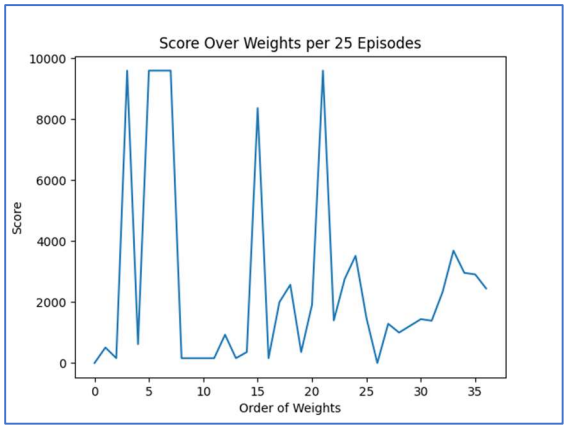

Figure 10: Results of running the DDQN agent with epsilon = 0 using the weights it learned after training.

Both agents appear to plateau at around 25 to 225 training episodes with the DQN agent getting to and staying at that plateau score of 9,600.

Interestingly, both agents learn the same strategy to achieve this high score. The strategy it chooses to get that score is to keep the ship completely still in the middle firing missiles at a consistent rhythm to destroy enemies as their flightpaths cross the center of the screen; see Figure 11 below.

Figure 11: Lazy but effective tactic where agent just sits in the middle and fires missiles periodically in Galaga

It makes no attempts to avoid dying to missiles or collisions which causes it to die midway through level 2. This might be because the game was set to end after 4000 steps anyway. This meant that the agent derived less value from preserving lives to score potentially higher scores. It only cares about getting the highest score possible in that time frame.

Episode 225 is also important because this is where both agents also happen to reach their minimum epsilon. This may be why both agents both go through a dramatic drop in performance over time. During these "valleys" of low scores, both agents would choose to stay at the borders of the screen refusing to move which caused them to get stuck on level 1 since a level in *Galaga* can only be beaten by destroying every enemy

(except for the bonus levels); see Figure 12 below. It is this problem that influences version 2 of the implementation.

Figure 12: Agent hiding like a coward in Galaga

The DDQN agent eventually remembers the previously discovered strategy at training episode 500, but quickly forgets and never recovers. The DQN agent starts to increase its score again towards the end of learning. It learns a new strategy that beats levels 1 and 2 but does not achieve as high of a score. However, it actively dodges threats and runs out of time rather than losing lives so it potentially could have scored more and showed behavior closer to that of a human player. This specifically occurs using the weights learned after 850 training episodes.

Version 2 decreased the decay rate for the exploration rate, but was only able to run for 12 hours, roughly 300 training episodes, for each agent. The results for this version of the implementation are below.

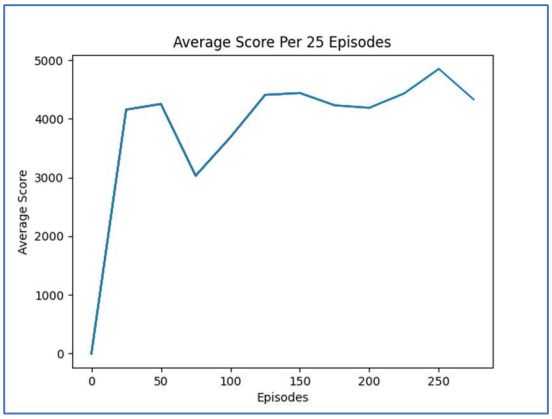

Figure 13: DQN_v2 Average Score Per 25 Episodes with epsilon (exploration rate) > 0.

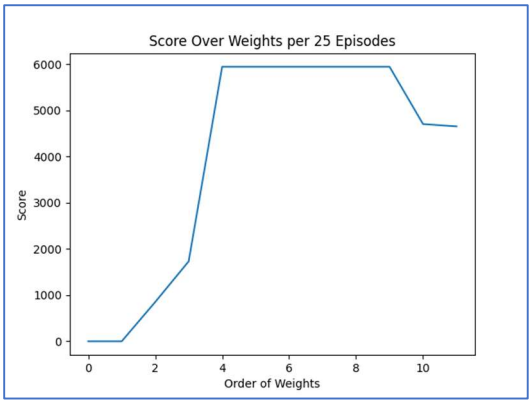

Figure 14: Results of running the DQN_v2 agent with epsilon = 0 using the weights it learned after training.

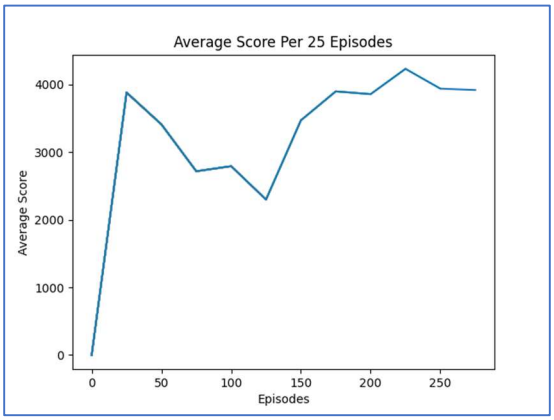

Figure 15: DDQN_v2 Average Score per 25 Episodes with epsilon > 0.

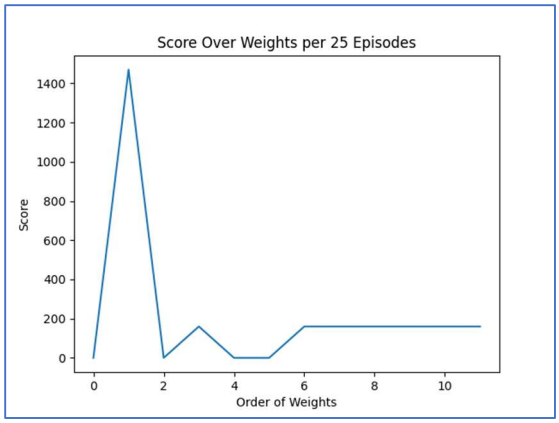

Figure 16: Results of running the DDQN_v2 agent with epsilon = 0 using the weights it learned after training.

This time, the DQN and DDQN agents acted drastically different from each other.

The DQN agent had an overall positive learning curve when run with an epsilon greater than 0. The graph with zero epsilon shows that it continues to learn and keep a relatively high score till the end. However, this was run before their epsilon reached the minimum exploration rate which would've been at about training episode 500. Unfortunately, I was unable to get enough requested time with the virtual computing service that I was using.

The DDQN agent had an increasing slope of average scores during training, but the agent performed terribly when loading the weights for each milestone of 25 steps repeating the "cowardly" strategies seen before. It even got zero points one time because it refused to fire missiles as well.

The results for Version 3, with the idle step limit instead of the total step limit, of both agents is displayed in the figures below.

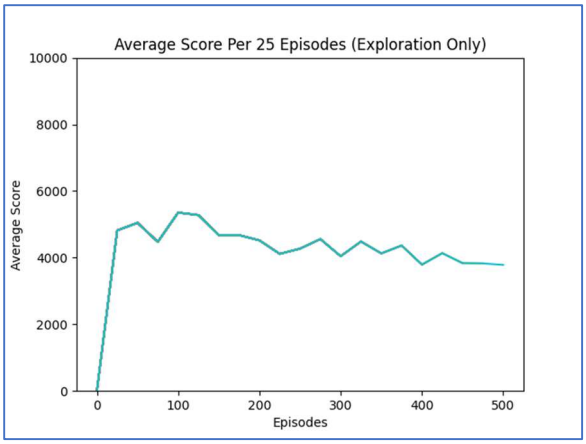

Figure 17: DQN_v3 Average Score per 25 Episodes with epsilon > 0.

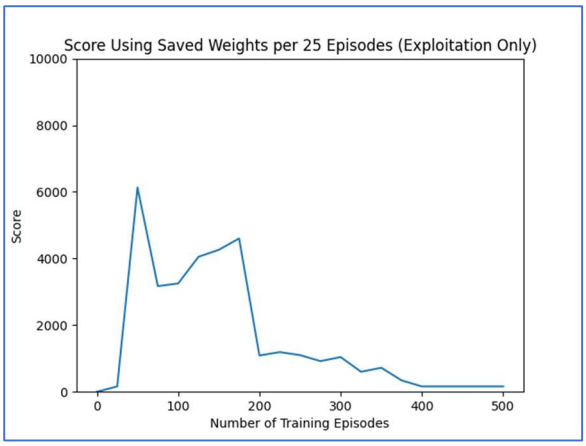

Figure 18: Results of running the DQN_v3 agent with epsilon = 0 using the weights it learned after training.

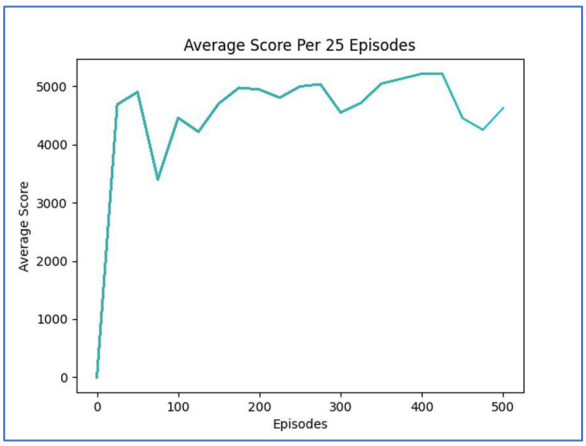

Figure 19: DDQN_v3 Average Score per 25 Episodes with epsilon > 0.

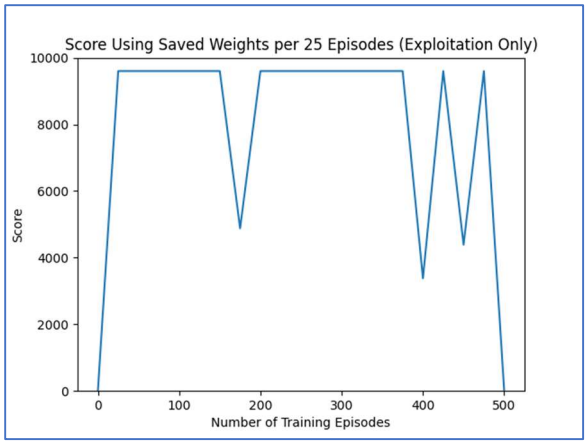

Figure 20: Results of running the DDQN_v3 agent with epsilon = 0 using the weights it learned after training.

This version also had very different results between the two, but the "victor" status flipped from the DQN to the DDQN agent.

The DQN agent appears to hold a steady score at least during training but performs poorly after training with a decreasing score meaning that it is learning to lose instead of win *Galaga*.

The DDQN agent appears to learn to score highly at around 10,000 points but then plateaus with occasional dips in score after training. During training, the DDQN agent shows a moderately increasing training slope suggesting that it is in fact learning to beat *Galaga*. However, it seems to favor using only the same "lazy" center strategy previously seen.

Summary

I implemented DQN and DDQN agents to play Galaga with the help of OpenAI Gym-Retro and TensorFlow. I then trained Version 1 of both agents for 900 episodes each. Out of the two DDQN performed the worst but not by much.

Dissatisfied with the forgetfulness of the agents, I decreased the epsilon decay rate so that they would spend more episodes exploring. This became version 2 of the agents which were both trained for roughly 300 episodes each. The DQN agent was the clear winner for this case having a positive learning curve with $\varepsilon \geq 0$.

Thankfully, I was able to get enough clear results using the Center for High Performance Computing resources provided by University of Utah before this project ended for Version 3 of the agents. I had desired to pit them against each other with a more faithful representation of the *Galaga* environment by removing the hard limit of 4000 steps per training episode. The original purpose was to prevent the agent from endlessly looping on a certain episode, but it restrained the agents from ever truly playing *Galaga* beyond the first few levels. To solve this, I added an idle steps limit to replace the original limit. This allowed the agents to experience more of the environment and prevented infinite looping. I then trained them for 500 episodes each.

Surprisingly, DQN was the dunce this time with a pitiful inconsistent score of around 6,000 points while DDQN excelled with a consistent score nearing 10,000 points.

Conclusions

The results are inconclusive for many reasons. One of these is the disparity of the results with neither agent showing a particular quality that gave it an edge over the other. Another reason is that I was unable to finish training sessions longer than 1000 episodes and my implementation of a DDQN may have been flawed. I believed that I was implementing it correctly using my sources, but I am less sure of that now. I will leave it up to the reader to figure out those answers themselves. Thank you for reading.

I will be presenting my results in class along with a live demo. For those who are unable to attend, my implementation code along with previous weights, measurements, and more can be found at my GitHub page:

https://github.com/bstew794/Galaga_RL_AI
.

Sources

[1] georgefidler1709. *Galaga_AI*. https://github.com/georgefidler1709/Galaga_AI. 2019

[2] M. Morales, *Deep reinforcement learning*. Shelter Island, NY: Manning Publications, 2020.

[3] Mishra, Mayank. *Convolutional Neural Networks, Explained*. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939. Aug 26, 2020

[4] Parmelee, Drew. *Getting an AI to play atari Pong, with deep reinforcement learning*. https://towardsdatascience.com/getting-an-ai-to-play-atari-pong-with-deep-reinforcement-learning-47b0c56e78ae. Jan 26, 2021

[5] Roy, Baijayanta. *Temporal-Differenace (TD) Learning: Reinforcement Learning using Temporal Difference (TD) Learning*. https://towardsdatascience.com/temporal-difference-learning-47b4a7205ca8. Sep 12, 2019

[6] Seno, Takuma. *Welcome to Deep Reinforcement Learning Part 1: DQN*. https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b. Oct 20, 2017

[7] Simmons, Leo. *Double DQN Implementation to Solve OpenAI Gym's CartPole v-0*. https://medium.com/@leosimmons/double-dqn-implementation-to-solve-openai-gyms-cartpole-v-0-df554cd0614d. Feb 5, 2019

[8] TensorFlow Core v2.70 Documentation. tf.keras.losses.Huber. https://www.tensorflow.org/api_docs/python/tf/keras/losses/Huber. Nov 5, 2021

[9] Torres.AI, Jordi. *The Bellman Equation: V-function and Q-function Explained*. https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7. Jun 11, 2020