CS 615 - Examination of Deep Q Learning Networks and the Potential Benefits with Respect to Video Game Development

Marcus Sullivan, Ishtiaq Shariar, Matthew Merenich

September 1, 2021

1 Abstract

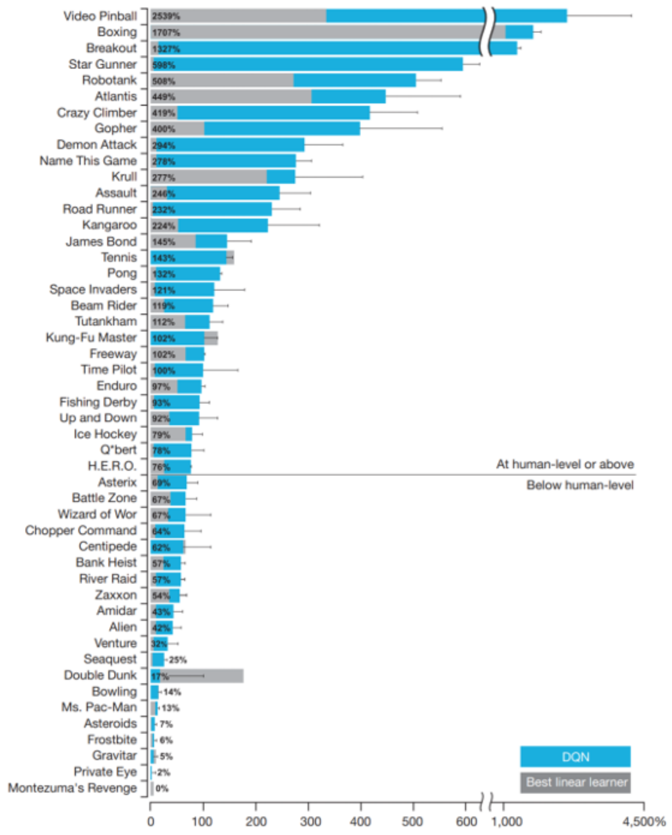
Video games have been an ideal platform to test the latest AI algorithms on how quickly and efficiently they can learn, and how realistic they can be. Little research has been done to identify the applicability of AI algorithms to video game development. Our project will try our hand at training an agent to learn how to play common video games environments using a reinforcement learning technique called Q-learning. The randomness of the learning phase of Q-learning is ultimately replaced by the output of a deep learning network that determines the optimal Q value (i.e., maximum predicted future reward) based on previous moves – that is, we implement a form of memory. Most published research with regard to Deep Q Networks (DQNs) utilizes a standard, open-source machine learning library, like Python's PyTorch, but utilizing existing libraries may not be ideal for video game development projects. Our team examined the comparability of custom neural network classes to the PyTorch implementation of an existing program. We then sought to examine the potential benefits that such an architecture could provide to the game development process by applying the resulting network to a new, internally-developed platforming game.

Our team was able to successfully train an implementation of the classic game Snake by replacing the existing PyTorch libraries with our own machine learning code. After 200 episodes, the custom implementation achieved a final rolling average (10 episodes) 20.5 points which was greater than the average score of the PyTorch implementation, 15.0. The PyTorch implementation did, however, have a greater learning efficiency. A 2D platformer, referred to as Hungry Hungry Pogo (HHP), was then developed in which the AI was required to collect pieces of food in an environment within a limited amount of time. The game added challenges of simulated gravity, platforms, and a constantly jumping character. When attempting to apply the same DQN network to the new game, the increased complexity of the game environment proved the existing DQN to be insufficient. Attempts to improve the network by tuning the reward signal were similarly unsuccessful. The final rolling average score achieved for each implementation ranged from 0 to 1.8 points. Our research showed that any attempted application of a DQN to the game development process should be approached with caution. Implementation of a DQN can take considerable time and effort and the resulting network may not be portable to multiple game scenarios.

2 Background

In 2015 Google DeepMind engineers developed a Q-learner based on human neurological behavior to replicate the complex decisions a human makes when faced with new situations. Their approach using high-dimensional sensory input data to train the Q-learner agent is different from past attempts to train a similar model; useful features were time consuming to produce, and the state data contained a smaller subset of data. The combination of reinforcement learning and neural networks allowed for high dimensional data to training the agent more effectively.

To test their model, the Atari 2600 emulator was utilized across dozens of games and would output frames and specific values – such as change in score and number of live remaining – for the agent via a deep learning architecture. What they found was a simpler approach to previous attempts was most effective: focusing on the input data and a timely representation of policy produced a more accurate model. This coincides with human interactions with each game; humans playing the game are also only privy to the current environment and a sense of reward for actions taken. Their final Q-learner agent proved to learn to a level of understanding and scores past even the best masters of the game (see below).



3 Description

3.1 Basics of reinforcement learning

Reinforcement learning, at its core, follows the Markov Decision Processes (MDP). This entails the iterative decision to select an action in each state that yields a reward and produces a new state. It's the agent's goal to maximize the reward, both immediately and in the long term.

- A state $\mathrm{S_t}$ at time t is observed by the agent, who take an action A. A new state becomes $\mathrm{S_{t+1}}$, and a reward $\mathrm{R_t}$ is granted to the agent.

The probability of a reward R given a state $\mathrm{S_t}$ are determined by the preceding state $\mathrm{S_{t\text{-}1}}$ and the action taken given A(s). Expected return is the probability the agent is learning to maximize, given a current state. The goal is to maximize the cumulative reward over time – this is the sum of future rewards. The probabilities an agent will select an action given a specific state is called *policy*, and the functions used to extract the best policy are called *value functions*. The goal of reinforcement learning is to find the optimal policy, which has an expected return greater than or equal to other policies policy for all states.

3.2 Basics of Q-learning

Q-learning is a type of reinforcement learning that utilizes an algorithm to update the maximum expected return (policy) given any state of play based on the reward of an action taken. Specifically, a Q-function determines the ideal action for the agent for a policy and outputs a q-value. States differ depending on the game, and include but not limited to speed, position, and boundaries. After an initial knowledge exploration period of almost random moves, our agent sees increasingly better results. This process is known as knowledge exploitation and extremely popular in video game simulation.

Our model uses the Bellman equation to update the state-value function.

$$\underbrace{NewQ(s,a)}_{\text{New Q-Value}} = \underbrace{Q(s,a)}_{\text{Current Q-Value}} + \underset{\text{Learning rate}}{\alpha}[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\text{Discount rate}}{\gamma} \underbrace{max\, Q'(s',a')}_{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s,a)]$$

The current Q-value is updated by the reward and maximum value of the action policy. A neural network tracks each state and their associated rewards. As previously noted, the exploration phase promotes randomness for the initial episodes. As the agent progresses, the state will increasingly call on past states' maximum reward. Eventually, the current Q-value becomes accurate enough that updating it results in diminishing returns. The discount rate ensures the current states are more influential that the expected value of future rewards.

3.3 Epsilon Greedy strategy

We previously discussed the exploration and exploitation processes in the previous section, and how our agent chooses to explore during the initial stages of the learning process. The numerical representation of this is the exploration rate, or epsilon ε, which is initially set to 1 or 100 percent certainty of exploring. This number slightly decreases as the model learns more about its environment. This switch between exploration to exploitation causes the agent to become "greedy" in finding the reward – it'll turn to its policy and highest Q-value to choose the next action. This process is called the epsilon greedy strategy. We implement exploration or exploitation based on a random number between 0 and 1; if the number is greater than ε, the agent will choose exploitation (and exploration for a number less than ε). In theory, as εdecreases, the chances of "greedy" behavior increase.

3.4 Deep learning replacing the Q-table

Traditional Q-learning algorithms will compare Q-values in a Q-table to select the maximum value. For simpler projects with smaller state and action pairs, this may be a feasible way to obtain and compute the maximum return. As learning time and state/action pairs increase, however, efficiency of updating the entire table after each iteration becomes computationally expensive and takes too long in gameplay. To address this, we implement deep learning networks to approximate the optimal Q-function, called deep Q-learning.

A neural network estimates Q-values is called a deep Q-network, or DQN. Given a state as input, the DQN will estimate Q-values for each action that can be taken in that state. The input consists of captured images from the game, which consists of the environment the agent faces at a specific time t. Multiple images are given to act as a mini video to provide more information to the network such as speed and direction of the agent. The DQN's primary objective is to minimize the loss between the estimated Q-values and the target Q-values as determined by the output of the Bellman equation. A DQN follows most ANN processes, forward and backward propagation to update the weights and values associated with each action. We'll talk about our specific ANN architectures in a later section.

3.5 Replay memory

Experience replay is a technique utilized during the training process of the DQN that stores the agent's experiences at each time t within a dataset called the *replay memory*. Each experience is comprised of the state of the environment S_t, the action A_t taken from this state, the reward R_{t+1} given to the agent at time t+1, and the next state S_{t+1} after the reward was given.

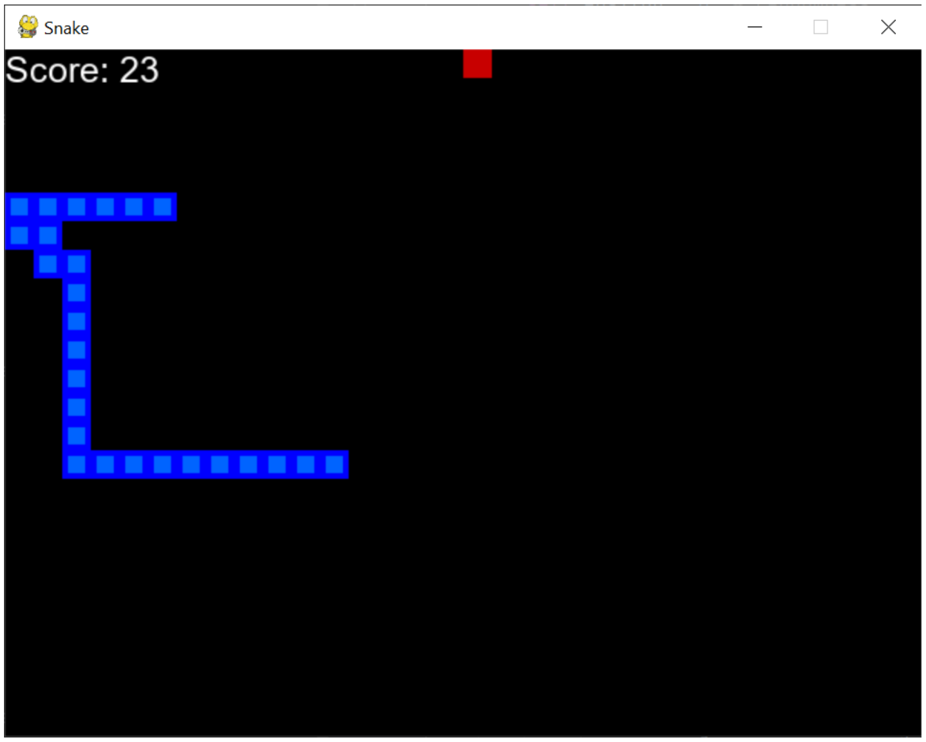
The biggest advantage to using replay memory in the training process is to disassociate the order of action to reward and alleviate correlation. Taking random memories and training produces effective and unbiased data.

3.6 Architectures

As mentioned above, a DQN is a proven architecture when it comes to teaching an AI to play video games. Two games were utilized, in order to examine the benefits of machine learning architectures for video game development. The first, a version of the classic Snake game that was developed by Gremlin in 1976, was used to obtain a general understanding of the DQN learning process. The second, a simple, internally-developed game titled Hungry Hungry Pogo (HHP), was used in order to assess the potential benefits of deep learning architectures for video game development as a whole. Two different architectures, namely SnakeAI and PogoAI, were generated to tackle each problem. The games and the accompanying architectures are described in detail below. All architectures were programmed using Python 3.9.5.

3.7 Snake and SnakeAI

Figure 1: Snake Screenshot

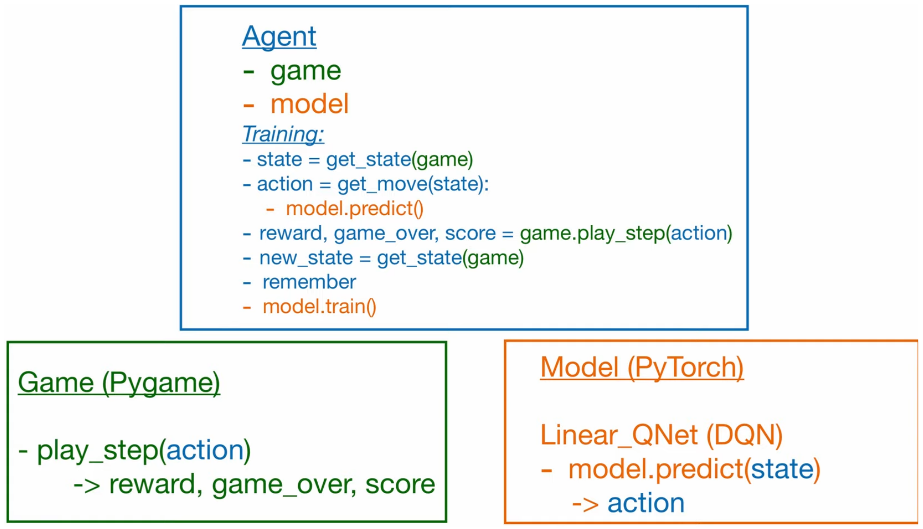


Snake is a 2D, single-player game in which the player controls a snake in a grid-based environment by selecting the direction (left, straight, or right) of the snake's next movement. The snake will automatically move toward the selected direction each game step. The player's goal is to collect food represented by a single red block. When the food is collected, the player receives a point, the length of the snake increases by 1 block, and a new food is spawned in an empty space. If the player steers the snake into the edge of the screen or itself, the game is over.

3.7.1 Architecture

Since Deep Q Learning was a new topic for our group, an existing implementation was utilized as a foundation for our Snake game. The Snake implementation was based on a GitHub repository (python-engineer/snake-ai-pytorch) pushed by Patrick Loeber (known as python-engineer on Github). The repository provided a Snake environment (game) that was programmed using the PyGame library as well as a working DQN programmed using the PyTorch library. Our goal was to analyze the existing modules and replace the PyTorch components of the network with our own neural network components.

Figure 2: Snake Game Architecture[1]



[1]Source: https://www.python-engineer.com/posts/teach-ai-snake-reinforcement-learning/

Modules: The basic architecture is described in Figure 1, above. The program consists of 3 primary modules: the Game, the Agent, and the Model. High-level descriptions of these modules are included below. Note, the basic learning procedure is the same as that described in Section 3.

1. *Game:*
   Encompasses the environment and receives player inputs that effect the game state.

2. *Model:*
   The actual DQN. It receives the game state and provides the next action (i.e., player input) for that game that theoretically maximizes the potential reward. The model in the source code was built primarily using PyTorch. The overall PyTorch architecture was generally shallow and depicted in Table 1, below. The model was evaluated using the Squared Error loss function. Note, the output of the DQN is a 1x3 vector to provide a Q value for each potential action (Left, Straight, Right).

Layer	In Size	Out Size
Fully Connected 1	11	256
ReLu	256	256
Fully Connected 2	256	3

Table 1: Snake DQN Architecture

3. *Agent:*
   Acts as the connective tissue between the Game (environment) and the Model (DQN) and has many responsibilities. It is responsible for coordinating the training process, passing inputs/outputs between the Model and the Game, and tracking the learning progress. The Agent is also responsible for getting the state of the Game and compiling it into a row vector that can be received by the Model and used to generate a new action.

3.7.2 Hyperparameters

Table 1, below, shows the final hyperparameter values for the architecture. All values, with the exception of the reward signals, were left unaltered from the original source code.

Hyperparameter	Details
ADAM	η: 1 ρ_1: 0.9 ρ_2: 0.999
Learning Rate	0.001
Reward	+50 for eating food[2] -10 for hitting a wall
Epsilon Greedy	ϵ : 0.4 Decay rate: 0.01
Replay Memory	Batch size = 1000
State	See Description Below

Table 2: Snake DQN Hyperparameter Summary

ADAM and Learning Rate The ADAM parameters and global learning rate were left unchanged from the original implementation.

Reward: The reward for the SnakeAI architecture was simple. The reward for eating the food on a timestep was +50. The Reward for ending the episode by hitting collision or running too long without collecting food was -10. The reward for eating food was increased from +10 in the original implementation to +50 in out implementation in order to account for a drop in learning efficiency that was observed after replacing the PyTorch components. These changes and the analysis of them are discussed in detail in Section 4.

Epsilon Greedy Strategy: In a reinforcement learning architecture, it is typical for the Agent to utilize an epsilon greedy strategy in order to balance exploration (random actions) and exploitation (DQN derived actions). This architecture, however, chooses to use a modified version of the strategy. Instead of discounting the probability of a random action by multiplying the current probability by a fractional discount rate, the Agent starts at a 40% random probability and decreases linearly by 1% each episode (i.e., each round of the game).

[2]Value altered. Original value was +10.

Replay Memory: The architecture utilized a replay memory. Disabling this component slightly reduced the overall rate of improvement per episode.

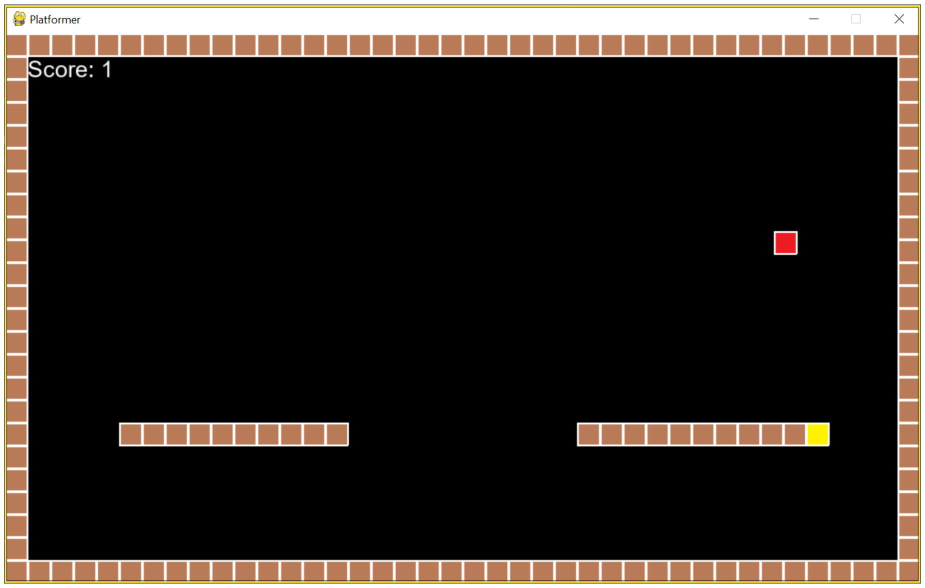
State: The state is collected by the Agent from the Game and consists of 11 Boolean values that attempt to accurately describe the environment to the Model. The following state values were passed to the model in the form of a single row vector to determine the predicted optimal Q value for each action (i.e., best action to take):

- Collision forward
- Collision right
- Collision left
- Moving left
- Moving Right
- Moving Up
- Moving Down
- Food is left
- Food is right
- Food is up
- Food is down

3.8 Hungry Hungry Pogo and PogoAI

Hungry Hungry Pogo (HHP) is a simple, internally-developed 2D-platformer. In its current state, the game was designed as a single-player experience that expands on a few of the ideas of Snake and introduces new challenges. The player controls a single block on an imaginary Pogo stick and once again is tasked with collecting food. The player must collect the food within a specific amount of time or the character will starve and the round will end. Collision no longer ends the game. The player can choose from just two actions, move left or move right. If the food is eaten, the countdown timer to the end of the round will be reset. The player will need to account for the constant change in vertical position as well as navigate around platforms in order to reach the food and achieve a high score. The overall goal was to determine how easy it would be to train an DQN network to control an AI for a video game. This network could theoretically be applied across multiple level designs without the need for retraining effectively reducing the development time for the developer.

Figure 3: Hungry Hungry Pogo Screenshot



Note: the player is the red block and food is the yellow block

3.8.1 Architecture

The same basic architecture from the Snake game implementation was used with a few customizations as described in Section 3.8.2, below. Although a CNN was considered, we were unable to get the CNN working with the DQN within a reasonable amount of time. In addition, a CNN would likely be impractical for use in an actual game since the image processing and CNN pipeline would likely result in a significant degradation of game performance.

3.8.2 Hyperparameters

Table 1, below, shows the final hyperparameter values for the architecture.

Hyperparameter	Details
ADAM	η: 1 ρ_1: 0.9 ρ_2: 0.999
Learning Rate	0.001
Reward	See Below for Details
Epsilon Greedy	ϵ : 1.0 Decay rate: 0.95 min: 0.05
Replay Memory	Batch size = 1000
State	See Description Below

Table 3: Snake DQN Hyperparameter Summary

ADAM and Learning Rate The ADAM parameters and global learning rate were left unchanged from the original implementation.

Reward: The reward for eating the food on a timestep was +50. The Reward for ending the episode was -10. Multiple additional rewards were tested as described in Section 4.2.

Epsilon Greedy Strategy: Since the Epsilon Greedy strategy did show a minor improvement relative to the no tuning strategy in Figure 5, a greedy epsilon randomness strategy was utilized for this network. See the evaluation of the Snake game for more details.

Replay Memory: Since turning off the replay memory showed a minor reduction in performance relative to the no tuning strategy in Figure 5, the replay memory was utilized for this network. See the evaluation of the Snake game for more details.

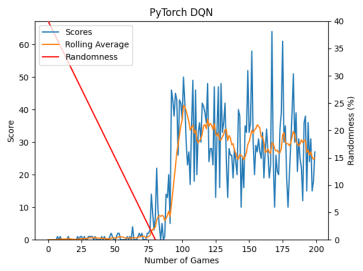
State: The following state was collected and sent to the Model. In total, 19 Boolean values were selected in order attempt to provide parity with the type of information that was used in the Snake implementation.

- If the player is grounded
- Velocity direction
- If the food is left
- If the food is right
- If the food is up
- If the food is down
- Out of reach
- Collision left
- Collision right
- Blocked up
- Blocked down
- Under Platform
- Over Platform
- Near Platform 1
- Near Platform 2
- Food Under Platform
- Food Over Platform
- Food Near Platform1
- Food Near Platform2

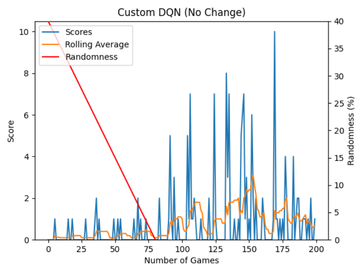
4 Evaluation

4.1 SnakeAI Evaluation

PyTorch vs Custom DQN: In order to evaluate the effectiveness of our DQN relative to the PyTorch DQN, each DQN was run for 200 episodes using the same hyperparameters (see Table 2). After running each DQN, the maximum score from a single episode and the average score of the last 10 episodes were compared.



(a) PyTorch



(b) Custom DQN - No Tuning

Figure 4: PyTorch vs Custom DQN

Figure 4 shows plots displaying the score and random action percentage (randomness) with respect to the number of episodes. Both plots show distinct increases in the maximum score achieved after the randomness reaches 0%. Based on the rolling average, the PyTorch implementation appears to reach a plateau shortly after the randomness reaches 0%. By contrast, the variability of the rolling average for the Custom DQN is far greater with respect to the maximum score achieved.

Custom DQN Hyperparameter Tuning: The effectiveness of hyperparameter tuning on the custom DQN was examined by running the DQN for 200 episodes with the following changes:

1. Increase the Learning Rate from 0.001 to 0.01
2. Decrease the Learning Rate from 0.001 to 0.0001
3. Remove Replay Memory
4. Increase Replay Memory batch size from 1000 to 2000
5. Implement true Greedy Epsilon (start=1.0, decay= 0.9, min=0.1)
6. Increase the positive reward from +10 to +50
7. Scale both positive and negative reward from 10 to 50

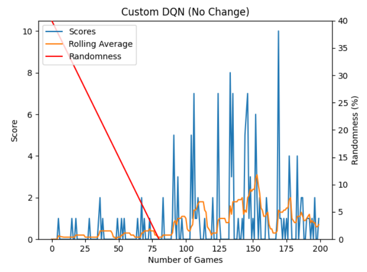
After running each DQN, the maximum score achieved (Max Record) across all episodes and average score of the last of the last 10 episodes (Final Rolling Average) were compared (Table 4). In addition, for each test, plots displaying the score and relative randomness with respect to the number of episodes were generated and are included in Figure 5.

Change	Max Record	Final Rolling Average
PyTorch	64	15.0
No Tuning	10	0.7
Learning Rate Increased	1	0.1
Learning Rate Decreased	2	0.2
No Replay	3	0.1
Replay Memory Increased	6	1.1
Greedy Epsilon	7	2.2
Positive Reward Increased	38	20.5
Both Rewards Scaled to 50	3	0.2

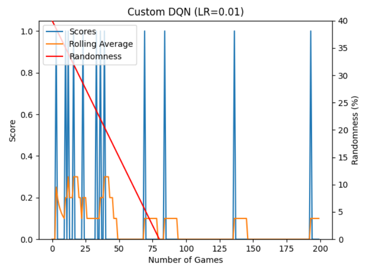
Table 4: Snake Custom DQN Hyperparameter Tuning

The data in Table 4 shows that PyTorch DQN performed the best with respect to the max record and had the second highest final rolling average with values of 64 and 15, respectively. The PyTorch DQN was also the most efficient learner with significant improvements to the average score seen immediately after the randomness percentage reached 0% (80 games). The next best implementation was the custom DQN in which the positive reward was increased; although the max record was about half that of the PyTorch implementation with a record of 38, the final rolling average was actually over 5 points greater than that of the PyTorch implementation. Notably, when both the positive and negative rewards were scaled to 50, the improvement in learning efficiency that was achieved by just scaling the positive reward up was completely lost. All other implementations performed poorly relative to the Pytorch implementation.

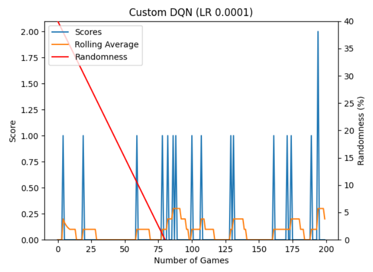
The plots from Figure 5 largely reflect the data in Table 4. Notably, for plot (g), *Positive Reward Increase*, the increase in the rolling average appears to still be on an upward trend at the conclusion of testing. It is possible that further training beyond 200 episodes would have resulted an even greater final rolling average.



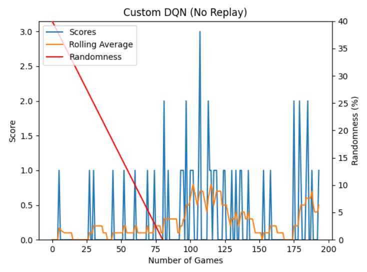
(a) No Tuning



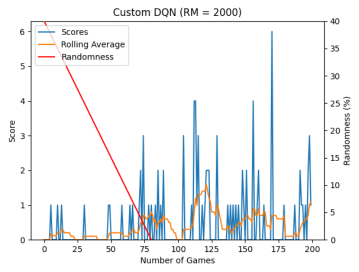
(b) Learning Rate Increased



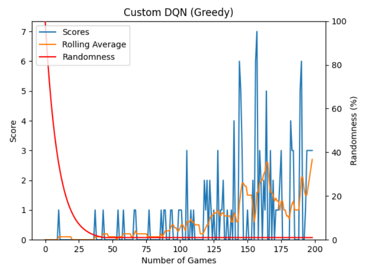
(c) Learning Rate Decreased



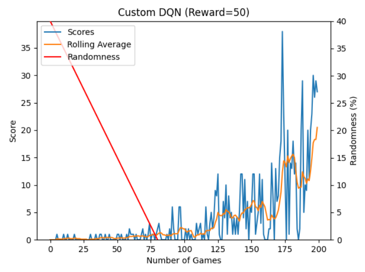
(d) No Replay



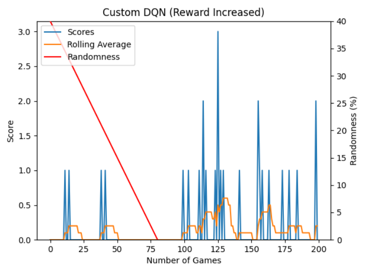
(e) Increase replay batch size



(f) Greedy Epsilon



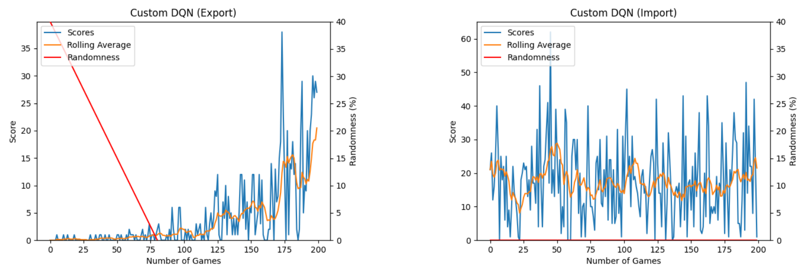
(g) Positive Reward increased



(h) Scale both Rewards

Figure 5: Hyperparameter Tuning for Custom DQN

Export and Reimport of Stored Values In order for a trained DQN to be useful for game development, a developer must be able to export the trained weights and re-import them into the project that they are working on. As such, the practicality of exporting and importing trained weights was examined. The DQN was modified to save the fully connected layer objects as a binary file using the Pickle library. After training for 200 episodes the final objects were re-imported into a new DQN instance with the learning functionality turned off. The resulting plots of score/rolling average relative to episode were then compared. The hyperparameters match those detailed in Table 2.



(a) Training

(b) Imported

Figure 6: Exporting and Importing Fully Connected Layers

As shown in Figure 6 above, the rolling average for the DQN using the imported fully connected layers closely matches the ending rolling average from the training DQN. This confirms that, once trained, re-utilization of the trained DQN would be simple and easy.

4.2 Hungry Hungry Pogo Evaluation

Examination of the effect of increased complexity The goal for training a DQN for HHP was to examine the benefits that a DQN could provide to the game development process. Since Snake is a fairly simple environment, we wanted to first identify the impact of increased environmental complexity on the learning process. This could give us an indication of the portability of a DQN from game to game. To test this, the AI was trained in the HHP environment with equivalent hyperparamenters from Table 2. The results shown in Figure **??** show that the reward signals and hyperparameter values from the Snake game are insufficient for teaching the DQN. Upon review of the training process, three primary causes of failure were identified and are depicted in Figure 8.

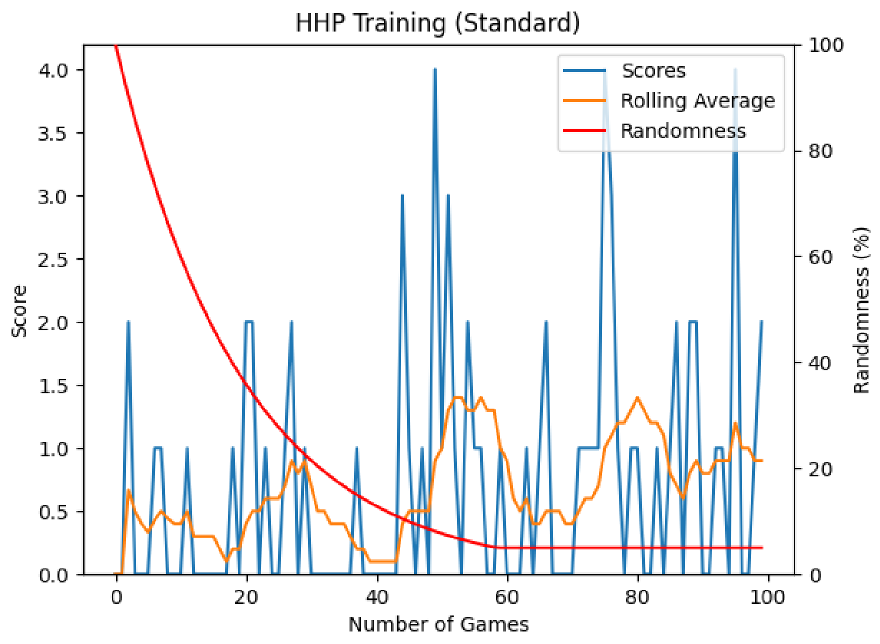
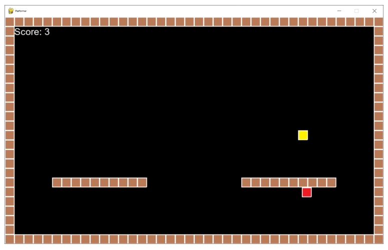
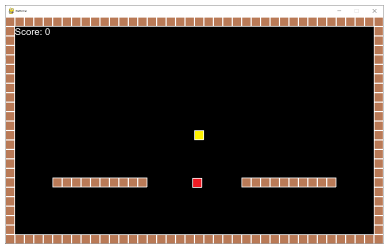


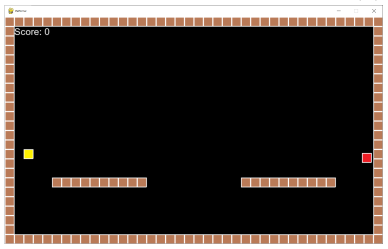
Figure 7: HHP DQN Performance



(a) Trapped



(b) Too High



(c) Stuck on Wall

Figure 8: Primary causes of failure

Causes of failure:

1. **Trapped:** The DQN gets trapped above or below a platform, with the food on the other side
2. **Out-Of-Reach:** The food is above the DQN's jump height when on the ground level
3. **Stuck-On-Wall:** The DQN gets stuck repeatedly trying to move passed a left or right wall

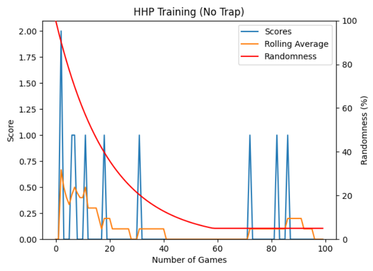
Examination of the reward signal: Based on the data from Section 4.1, the reward signals are the most important factor identified in relation to performance of the DQN. As such, the effects of different reward signals on learning efficacy and behavior deterrence were examined. The following reward strategies were introduced to the DQN in order to attempt to address the most common reasons. Each strategy was trained for 100 episodes

1. **No Trap:** The DQN was given a negative reward for being on the other side of the platform from the food.
2. **Too High:** The DQN was given a negative reward for when in a grounded state if:
$$jumpHeight + playerHeight < foodHeight$$
3. **No Walls:** The DQN was given a negative reward for being next to a wall.

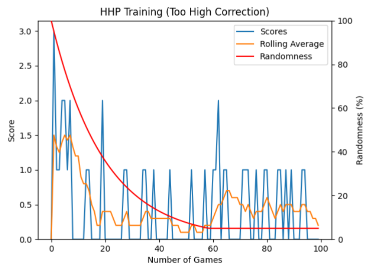
Results: When the No Trap strategy was implemented (Figure 9-a), the DQN was unable to learn any clear pattern. The DQN would typically fall victim to the stuck-on-wall problem and time out of the episode.
When the Too High strategy was implemented (Figure 9-b), the DQN, once again was unable to learn any clear pattern and would typically fall victim to the stuck-on-wall problem and time out of the episode. When the No Walls strategy was implemented (Figure 9-c), the DQN learned to perform a sweeping movement pattern, sweeping back and forth from each end of the level in order to eat the food. The DQN successfully learned to avoid the side walls. Overtime, however, the DQN still learned to prioritized the X location of the food and would fall victim to the Trapped problem, ending the run (i.e., trapped problem).

In order to determine if the DQN could overcome the Trapped problem if the DQN implemented the No Walls strategy, the two reward signals were combined into a single network. The network once again learned the sweeping movement approach and managed to avoid both the Stuck-on-Wall and Trapped problems. The results showed a moderate improvement in the final rolling average with an apparent upward trend towards the end of the learning process. The primary cause of a terminated episode was a lack in accuracy in which the DQN would repeatedly miss the food during its sweeps. The network also occasionally fell victim to the Too High problem. In recognition of the apparent upward trend at the end of training, this strategy was tested again with across 200 episodes. The new results (Figure 9-d) revealed that the learned strategy ultimately failed, with the DQN ultimately reverting and falling victim to the Trapped problem. The numerical data for all strategies is reported in Table 5. Only the data for the 200 training episodes is reported for the combined No Wall No Trap strategy

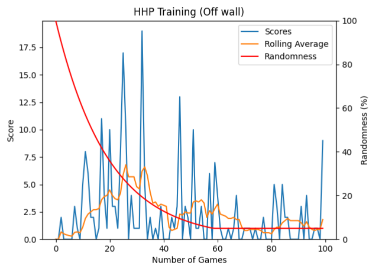
Based on the data in Figure 9 and Table 5, the moderate increase in environment complexity resulted in a complete loss in learning ability and new reward strategies needed to be developed. Although the reward signal can be tailored to encourage or discourage specific behaviors, as was seen when attempting to avoid the walls, identifying and addressing multiple issues with what is ultimately a single scalar reward value proved to be increasingly difficult with each new strategy implementation.



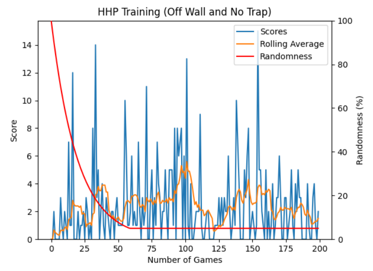
(a) No Trap



(b) Too High



(c) No Wall



(d) No Wall and No Trap

Figure 9: Performance of DQN After Reward Changes

Change	Max Record	Final Rolling Average
Standard	4	0.9
No Wall (avoid walls)	19	1.8
No Trap (avoid platform traps)	2	0
Encourage vertical movement	3	0.2
No Wall No Trap (200 episodes)	15	1.5

Table 5: HHP Reward Tuning

5 Conclusion

Replacing PyTorch with Custom Neural Network Classes: We demonstrated that a network trained using PyTorch could successfully be trained using custom Neural network classes. Although the performance of the custom neural network was lower relative to the PyTorch implementation, the network did manage to reach comparable average scored in the Snake game. The PyTorch implementation displayed a greater learning efficiency with respect to the number of games played (episodes). It is currently unclear why this is the case, however, some possibilities include the optimizations in the weight initialization for the fully connected layers and Adam parameters used for backpropagation.

Randomness and Epsilon Greedy Strategy: Random actions in a DQN are intended to force the network to take actions that it would not normally have taken, thereby encouraging exploration of the environment and creating new learning opportunities. The environment, however, appears to have a significant impact on the usefulness of randomness for exploration. The data in Figures 4 and 5 show how prolonged randomness could be detrimental to the learning process. In Figure 4, the PyTorch Implementation only really begins to learn once the randomness reaches 0%. The plot for the positive Reward increase (subfigure g) in Figure 5 displays a similar trend. In Snake, an episode ends if the snake hits itself of the wall. Since the snake grows after each point, the environment makes a random action inherently more dangerous every time the snake eats the food. The opposite relationship was observed in the HHP testing, in which the DQN performance was greatest when the randomness was still high. This is likely because there is no inherent danger to random movements in the HHP game aside form wasting time. As such, the benefit to of environment exploration may have outweighed the potential time loss.

Reward signal Tuning: Based on the data from Section 4.1, for simple environments, the reward signal can effectively be tuned to increased the performance of the DQN. In the Snake environment, the reward signal was most influential when it was large and infrequent relative to the percentage of random actions. Notably, scaling both the negative reward for terminating an episode and the positive reward for eating the food resulted in a loss of the learning improvements that were achieved by just implementing the latter. This indicates that the effectiveness of tuning the reward is related to the balance between all potential rewards, rather than just the magnitude of the reward signal. It is possible that a similar result could be achieved if the food was larger and, thus, the frequency of receiving a positive reward was larger, but more testing would need to be performed in order to confirm this.

For more complex environments, however, attempting to manually tune the reward signals in an effort to improve learning efficiency was determined to be prohibitively difficult. Although the data from Section 4.2 did demonstrate that the reward signal could be tuned to discourage specific behaviors, we were unable to successfully implement multiple reward strategies at the same time. This is likely due to the fact that the reward signal is reported to the model as just a single scalar value that may not be suitable for reinforcing complex behaviors on its own.

Game Development: Despite demonstrating that Fully Connected layers could easily be exported and imported into DQN architectures, based on our data and the discussion above, we have determined that the complexity of DQN architectures prevents them from being readily utilized for game development. Even with our simple platforming example, HHP, the DQN architecture was exceedingly difficult to train by through manual modification of the hyperparameters. This is not to imply that a DQN cannot be a useful tool for game development, but instead indicates that any attempt at implementation would likely require a not-insignificant amount of research, patience, and comprehension.

6 Future Work

The most obvious course of action moving forward would be to further study the affect of other DQN aspects such as network depth and state signals on learning efficacy. Although we were unable to implement the CNN with the DQN architecture, the DeepMind project demonstrated that a CNN front-end is capable of achieving superhuman performance in the video game playing domain. Consideration should be given, however, to the impact that implementing a CNN-based DQN would have in terms of processing power (CPU and GPU). Even modern 2D video games can struggle with performance issues. The image processing required and extra propagation time for the CNN could prove to be costly to the available CPU and GPU bandwidth.

It is also possible that a non-CNN-based DQN could still be reasonably effective for a strategy or tactics games (e.g., the Civilization series from Firaxis or Final Fantasy Tactics from Square Enix). In these types of games, the state data and reward signals are more obvious and readily available. Moreover, the actions that the DQN would need to make in strategy games are typically not time dependent in the same way that a platformer is. The strategy game domain could be a intellectually lucrative research area in relation to the benefits that a DQN could provide during game development.

7 Bibliography

References

[1] *Deep Q-Learning Tutorial - Explained and Coded from Scratch.* Mar. 13, 2021. URL: https://www.reddit.com/r/learnmachinelearning/comments/m42lnz/deep_qlearning_tutorial_explained_and_coded_from/?utm_medium=android_app&utm_source=share.

[2] deeplizard. *Reinforcement Learning - Goal Oriented Intelligence.* Dutch. Mar. 20, 2020. URL: https://www.youtube.com/playlist?list=PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv.

[3] Kansal and Martin. *Reinforcement Q-Learning from Scratch in Python with OpenAI Gym.* 2021. URL: https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/.

[4] Abhav Kedia. *Creating Deep Neural Networks from Scratch, an Introduction to Reinforcement Learning.* Apr. 20, 2020. URL: https://towardsdatascience.com/creating-deep-neural-networks-from-scratch-an-introduction-to-reinforcement-learning-6bba874019db.

[5] Python-Engineer Loeber. *GitHub - python-engineer/snake-ai-pytorch.* 2020. URL: https://github.com/python-engineer/snake-ai-pytorch.

[6] M. Matta. *QRTS: A real-time swarm intelligence based on multi-agent Q-learning.* May 1, 2019. URL: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/el.2019.0244.

[7] Python Engineer. *Teach AI To Play Snake - Reinforcement Learning Tutorial With PyTorch And Pygame (Part 2).* Dec. 21, 2020. URL: https://www.youtube.com/watch?v=5Vy5Dxu7vDs&ab_channel=PythonEngineer.

[8] undefined. *Teach AI To Play Snake - Reinforcement Learning Tutorial With PyTorch And Pygame (Part 2).* Dec. 21, 2020. URL: https://www.youtube.com/watch?v=5Vy5Dxu7vDs&ab_channel=PythonEngineer.

[9] Volodymyr, Kavukcuoglu, and Silver. *Human-level control through deep reinforcement learning.* Feb. 26, 2015. URL: https://www-nature-com.ezproxy2.library.drexel.edu/articles/nature14236.pdf.