# Pixelcopter with Alternative Q-Learning Agents

Millin Gabani
*University of Waterloo*
Waterloo, Canada
mgabani@edu.uwaterloo.ca

Nathan Leung
*University of Waterloo*
Waterloo, Canada
neleung@edu.uwaterloo.ca

Benjamin Lo
*University of Waterloo*
Waterloo, Canada
b7lo@edu.uwaterloo.ca

Joshua Sayavong
*University of Waterloo*
Waterloo, Canada
jsayavon@edu.uwaterloo.ca

***Abstract*—This paper applies various reinforcement learning techniques to a side-scrolling game called Pixelcopter, namely Q-Tables with sparse and dense discretization as well as Double Deep Q-Learning. The motivation is to characterize and understand the Artificial Neural Network based improvements to table-based Q-learning agents in continuous state domains. In the Pixelcopter environment, we conduct experiments with each reinforcement learning agent to quantify and compare performance with respect to reward, model convergence time, and model size. We also explore Q-table discretization issues that deep Q-learning agents avoid to achieve significantly better performance. The results demonstrate the potential of deep Q-learning for reinforcement learning in applications with continuous domains with high-dimensionality, such as autonomous navigation.**

***Index Terms*—Reinforcement learning, Q-learning, artificial neural networks, game playing, autonomous navigation**

## I. Introduction

Reinforcement learning (RL) is a field of machine learning in which agents aim to maximize reward by taking actions in an environment. RL has roots back in the 1900s, with the fields of animal learning psychology, optimal control with value functions and dynamic programming, and temporal-difference methods coming together to produce the RL we know today. RL's resurgence in popularity within the past six years has sped up development. Since DeepMind's Deep Q-Learning paper in 2013 [1], the field has exploded with a variety of clever techniques to help agents learn.

One application of RL is autonomous navigation, where the goal is to find an optimal movement behavior and path by balancing exploration and exploitation of the environment. Typically, huge quantities of data from sensors like lidars and cameras act as high-dimensional input, leading to intractability of RL due to the large dimensionality. Accordingly, the autonomous navigation application can be reduced to a game of a similar nature. Game playing lends itself well to reinforcement learning since the environment is well-defined, actions are clear-cut and limited, and the reward can be easily specified according to the goal of the game.

The motivations for this paper are the following: 1) We would like to attempt to quantify the effect of a selection of modern reinforcement learning improvements, including the use of Artificial Neural Networks and Double Deep Q-Learning, and 2) we would like to apply reinforcement learning to a simplistic version of autonomous navigation in hopes that the findings scale to the full-scale, useful problems faced in the real world.

In this work, tabulated Q-learning agents with different levels of discretization are compared against a deep Q-learning agent. The paper attempts to provide context and understand the principles behind the proposed agents, adopting the Pixelcopter game as the evaluation environment.

### A. Pixelcopter

Pixelcopter is a side-scrolling game in which the player, depicted as a white pixel, attempts to navigate through a tunnel. A game ends when the Pixelcopter collides with anything green, either the top or bottom walls of the tunnel or any of the barriers in between. The environment is randomly generated for each game, and a screenshot is shown in Figure 1. The only action that the player can take is to accelerate the Pixelcopter upwards. If the player chooses not to do so, the Pixelcopter accelerates downwards. The reward/score increases by 1 for every vertical green block that the Pixelcopter passes and decreases by 10 every time the game ends.

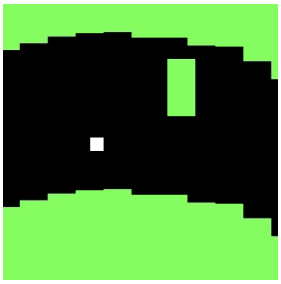


Figure 1: Pixelcopter screenshot

The PyGame Learning Environment (PLE) allows for reinforcement learning development in Python and features 9 available games, which includes Pixelcopter. PLE provides the game interface, valid action space, reward function, and a non-visual representation of game state.

## II. Related Works

The most expansive reference on the current state of reinforcement learning is the book by Richard Sutton [2]. This book describes the details of various algorithms which are used in this paper, specifically Q-learning and its modifications. In another paper, Sutton [3] introduces the Dyna architecture, which includes an internal model of the world in order to interleave planning and learning by trial-and-error.

One of the most significant milestones of reinforcement learning in the gaming space was TD-Gammon in 1992 [4]. This used classical reinforcement learning techniques, known as Temporal Difference Learning, to play backgammon at a "strong master level" that rivals that of top human players. However, extending the model-free reinforcement algorithms to other games like checkers or Go proved unsuccessful due to the divergence of the Q-learning.

Perhaps the most notable and well-known leader in the field of reinforcement learning today is Google's DeepMind. Most recently, their work on games such as Go and Starcraft II have exhibited the potential behind Deep Reinforcement Learning. Their first foray into game playing comes from their 2013 paper [1] in which they were able to train an agent to play various Atari games, using Deep Q-Learning and raw pixel data. This was the first successful application of an Artificial Neural Network (ANN) to simulate a Q-table. They used performance on an Atari emulator introduced by Bellemare, Naddaf, Veness, and Bowling [5], and since then, these Atari games have been the de-facto method to benchmark performance on reinforcement learning [6] [7] [8] [9].

The DeepMind team also introduced the concept of Experience Replay (ER) to reinforcement learning. This prevents the Deep Q-Network (DQN) from overfitting from its most recent experience and "forgetting" previous experiences. Liu and Zou [10] explored the specific effects that ER has on learning. They found that ER outperforms the averaged fixed memory strategy in 71%, up to 86%, and up to 100% of trials of the CartPole, MountainCar, and Acrobot games respectively.

Another improvement with the DQN algorithm was the advent of the Double Q-Learning algorithm, introduced by van Hasselt, Guez, and Silver [11]. The use of two neural networks, called Q-Network and Q-Target, was shown to greatly reduce bias. Using an integrated perception approach to build the environment, Zhang, Sun, Yin, Lin, and Wang [12] applied DDQN to autonomous vehicle speed control, scoring 271.73% better than DQN. Translating agents from virtual environments to the real world has also been explored using both real captured and synthetically generated images [13] [14] [15]. To combat the challenge of high dimensionality of complex domains like autonomous navigation, efforts have also been made to evaluate and improve upon the following: exploration strategies [16] [17], policy selection and planning [18], deep neural network architectures [19], and neural network encoding and compression [20] [21].

Apart from deep reinforcement learning, there have also been other RL strategies for the Atari 2600 games. Liang, Machado, Talvitie, and Bowling [22] developed the B-PROS, B-PROST, and Blob-PROST shallow RL agents that cut down computational and memory costs almost threefold compared to DQN. They devised a benchmark for the Atari Learning Environment and found that Blob-PROST and DQN have similar performance. Jaderberg et al. [23] proposed an RL agent that learns from auxiliary pseudo-reward functions in parallel with the cumulative reward, achieving 880% mean and 250% median performance and surpassing A3C and DDQN.

## III. Technical Background

### A. Reinforcement Learning

Reinforcement Learning is one of the three paradigms of machine learning besides supervised and unsupervised learning. Reinforcement learning comprises of an agent and an environment. The agent's goal is to explore and exploit the environment to maximize user-defined rewards. The environment is typically formulated as a Markov Decision Process (MDP) and the representations of environments are called states [24]. The agent takes action based on the state of the environment, and this action updates the state of the environment and the agent receives a reward for the state-action combination. The agents updates its action policy for a given state using the user-defined reward.

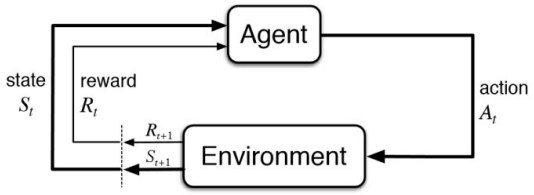



Figure 2: Typical reinforcement learning reward loop

The agent has an option to explore or exploit its interaction with the environment. The exploration versus exploitation trade-off in reinforcement learning is a stochastic multi-armed bandit problem, and many strategies like $\epsilon$-greedy, Boltzmann exploration, and pursuit algorithms exist to address it [25].

### B. Q-Learning

First developed by Watkins [26], Q-learning is quite a primitive, but effective, method of learning how to solve a problem in a finite, discrete world. Q-learning allows agents to evaluate actions for a given state and optimize outcomes through Markovian domains. In Q-learning, the Q-value is meant to determine the expected reward from taking a particular action. In an agent's policy, the agent is meant to take the next action which would yield the highest Q-value in order to maximize reward.

Watkins created the Q-learning algorithm which allows an agent to learn a policy by attempting all possible states and all possible actions. The algorithm is as follows [26]:

- Observe the current state, $x_n$
- Select and perform an action, $a_n$
- Observe the next state, $y_n$
- Receive immediately a reward, $r_n$
- Adjust $Q_{n-1}$ using a learning factor, $\alpha_n$, according to the following equations:

$$Q_n(x,a) = (1-\alpha_n)Q_{n-1}(x,a) + \alpha[r_n + \gamma V_{n-1}(y_n)] \quad (1)$$

$$V_{n-1}(y) = \max_b\{Q_{n-1}(y,b)\} \quad (2)$$

By performing this algorithm, the Q-table iteratively builds Markov chains of states, actions, and rewards. As states are repeatedly visited, the Q-table will converge, and an optimal

policy for each state will be possible when picking the action that maximizes expected reward.

Of all of the methods mentioned in this paper, Q-learning is most simple method of storing actions and states because nothing more complicated than a table needs to be used. Watkins [27] describes the use of a lookup table in his form of Q-learning. Table 1 shows an example of a Q-value table. Each row represents a perceived state. The columns represent the predicted Q-value for each possible action for each state.

Table I: Example of Q-value table

| State ID, $x_i$ | $Q(x, a_1)$ | $Q(x, a_2)$ |
|---|---|---|
| 0 | 0.1425 | -0.324 |
| 1 | 0.141 | 0.212 |
| 2 | -0.271 | 0.912 |
| 3 | 0.437 | 0.296 |
| 4 | -1.253 | 0.576 |
| ... | ... | ... |

Tabulated reinforcement techniques, such as classical Q-learning described here, were originally developed for discrete domains. To translate Q-learning to continuous domains, the continuous states need to be discretized to discrete ones; which will be explored in this paper. Another possible solution is to use function approximation of the Q-Table to handle these continuous states.

### C. Deep Q-Learning

The function approximation of the Q-table is the other solution to domains with large or infinite state spaces. As explored by Tadi [28], this approximation was classically a linear function; however, it struggled with generalization and efficient parameter tuning. DeepMind was the first to successfully use an Artifical Neural Network for function approximation and has since become the conventional method for Q-learning, now called Deep Q-Learning. The ANN used is similarly named as a Deep Q-Network.

These networks are trained similarly to how the Q-table is updated in Equation 1. Equations 3 to 5 formally express the ANN training process. In this case, the Delta Rule is used for weights on the output layer of the DQN. Standard backpropagation algorithms can then be used to train the entire network. As training is performed and similar states are visited multiple times, the DQN will evolve to output increasingly more accurate rewards for each action until convergence. At this point, choosing the action with the highest reward will yield an optimal policy.

$$y_{des} = r_n + \gamma \max_b \{Q_{n-1}(y, b)\} \tag{3}$$

$$y_{pred} = Q_{n-1}(x, a) \tag{4}$$

$$\Delta w = \alpha_n (y_{des} - y_{pred}) \tag{5}$$

However, in practice, the DQN approximation faced some issues that were not experienced with the tabulated method. The first is how the model could overfit the most recent states and "forget" how to perform in previous states that haven't been seen recently. The solution is a method called Experience Replay (ER), which allows the DQN to train on a batch of randomly selected previous experiences. This memory of experiences, defined as the state, action, and reward from a frame of an episode, is written to on each new step of the algorithm.

### D. Double Q-Learning

Double Q-Learning (DDQN) is method proposed by Hasselt, Guez and Silver to address overfitting in DQN. The Google DeepMind paper [11] proves that DQN suffers from substantial over estimations in some games in the Atari 2600 domain.

DDQN addresses overfitting by decoupling selection from evaluation. DDQN uses two neural networks unlike DQN, which only uses one. The Q-Network functions similar to the network used by DQN; it is responsible for selection of the next action. However, a second network called Q-Target is used to evaluate the action selected. The trick here is that the target value is not automatically produced by the maximum Q-value but by the Q-Target network.

DDQN uses two sets of weights $\theta$ and $\theta'$. Update to Q-Network's weights ($\theta$) is similar to that of DQN. However, update to Q-Target's weights ($\theta'$) is done using the following target value:

$$y_{des}^{Q-Target} = r_n + \gamma Q(S_{t+1}, \max_b Q(S_{t+1}, b; \theta_t); \theta'_t) \tag{6}$$

For a clear comparison, we can first untangle the selection and evaluation in Q-learning and rewrite its target (Equation 3) as the following:

$$y_{des}^{Q-Network} = r_n + \gamma Q(S_{t+1}, \max_b Q(S_{t+1}, b; \theta_t); \theta_t) \tag{7}$$

The update to weights in both network is done by using the loss function described in Equation 5 and backpropagation.

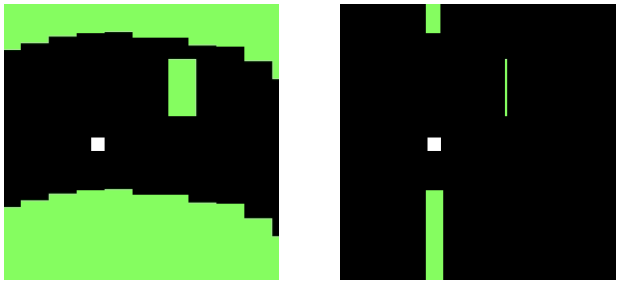
## IV. Agents

The reinforcement learning techniques described in the previous section were implemented as separate agents to compare performance. Each agent uses the PLE interface, providing them with the current state, possible actions, ability to perform each action, and the reward from each action. The current state is represented as a 7-dimensional feature vector, summarized in Table II. The possible actions are the following: 1) no action, which accelerates downward due to gravity (denoted as $a_1$), and 2) accelerate upwards (denoted as $a_2$).

Table II: Pixelcopter state dimensions

| Dimension ID | Description | Units |
|---|---|---|
| 0 | Player Position | pixels |
| 1 | Player Distance To Ceiling | pixels |
| 2 | Player Distance To Ceiling | pixels |
| 3 | Next Gate Distance to Player | pixels |
| 4 | Next Gate Top Height | pixels |
| 5 | Next Gate Bottom Height | pixels |
| 6 | Player Velocity | pixels/frame |

One important note is that the state of the game that is given to each model is extremely limited. Figure 3 illustrates the limited information that the agent receives, except velocity.



(a) Pixelcopter display

(b) Pixelcopter state seen by RL agents

Figure 3: Visual representation of Pixelcopter state seen by RL agents

For consistency, each agent used the same hyperparameter values described in the previous section. These values are summarized in Table III.

Table III: Q-table approximation

| Parameter | Value |
|---|---|
| $\epsilon_{start}$ | 1.0 |
| $\epsilon_{decay}$ | 0.02 / episode |
| $\gamma$ | 0.97 |
| $\alpha$ | 0.0004 |

### A. Q-Value Table, Sparse Discretization

An agent called Q-Table Sparse (QTS) was created for Pixelcopter which uses Q-learning to determine which decision to make. In order to adequately use the Q-table, the game state dimensions described in Table II must be discretized. To do so, each dimension was first explored to determine the possible range of values. Then, a scheme for discretization of each dimension was created. Each possible range of each dimension was first bounded based on the dimension characteristics. For example, place position is bounded to [8, 40] because the map has walls as a top and bottom border which blocks the player from reaching the very edge of the screen. A step size was determined for each dimension, and buckets were created. Dimensions could now be grouped into these buckets for discretization. The player position dimension ended up having 5 steps by rounding the value to the nearest value of the following set: $\{8, 16, 20, 24, 28, 32\}$.

Table IV: Discretization of Pixelcopter state dimensions

| ID | Possible Range | Allowed Range | Step Size | # Steps |
|---|---|---|---|---|
| 0 | [0, 48] | [8, 40] | 8 | 5 |
| 1 | [0, 48] | [4, 20] | 4 | 5 |
| 2 | [0, 48] | [4, 20] | 4 | 5 |
| 3 | [0, 48] | [8, 40] | 8 | 5 |
| 4 | [0, 48] | [8, 40] | 8 | 5 |
| 5 | [0, 48] | [8, 40] | 8 | 5 |
| 6 | [-10, 10] | [-1.5, 1.5] | 0.5 | 7 |

Now that each dimension of the state was bucketed, the state of the game was able to be represented by some permutation of dimension buckets. The game state now must be mapped to a state ID in order for the state to fit distinctly into a table row. The following mapping is used to determine the row number of each possible game state.

$$i_{row} = d_0 + 5 \cdot d_1 + 5^2 \cdot d_2 + 5^3 \cdot d_3 + 5^4 \cdot d_4 + 5^5 \cdot d_5 + 5^6 \cdot d_6$$

$$d_0, d_1, d_2, d_3, d_4, d_5 \in [0, 4], \qquad d_6 \in [0, 6]$$

$$i_{row} \in [0, 109374]$$

Using this lookup table, the Q-values were encoded, and the agent was able to optimize actions for every state of the game.

### B. Q-Value Table, Dense Discretization

Another agent called Q-Table Dense (QTD) was created which was similar to the previous agent. It uses the Q-value table like the previous agent, but the number of discretizations for each dimension were doubled. The purpose of this agent is to explore the effects of an increase in number of states on Q-learning. The following tables show the new discretizations of this agent and their subsequent mapping to state ID.

Table V: Discretization of Pixelcopter state dimensions

| ID | Possible Range | Allowed Range | Step Size | # Steps |
|---|---|---|---|---|
| 0 | [0, 48] | [4, 44] | 4 | 11 |
| 1 | [0, 48] | [4, 44] | 4 | 11 |
| 2 | [0, 48] | [4, 44] | 4 | 11 |
| 3 | [0, 48] | [4, 44] | 4 | 11 |
| 4 | [0, 48] | [4, 44] | 4 | 11 |
| 5 | [0, 48] | [4, 44] | 4 | 11 |
| 6 | [-10, 10] | [-2, 2] | 0.5 | 9 |

$$i_{row} = d_0 + 11 \cdot d_1 + 11^2 \cdot d_2 + 11^3 \cdot d_3 + 11^4 \cdot d_4 + 11^5 \cdot d_5 + 11^6 \cdot d_6$$

$$d_0, d_1, d_2, d_3, d_4, d_5 \in [0, 10], \qquad d_6 \in [0, 8]$$

$$i_{row} \in [0, 17715609]$$

### C. Double Deep Q-Learning

This agent implements the DDQN approximation with a 5000 item Experience Replay. The specifics of the Q-Network and Q-Target architecture are described in Table VI, which corresponds to 3952 weights, or 7904 weights for both networks.

Table VI: Deep Q-Network layers

| Layer | Number of Neurons | Activation |
|---|---|---|
| Input | 7 | N/A |
| 1 | 32 | relu |
| 2 | 48 | relu |
| 3 | 32 | relu |
| 4 | 16 | relu |
| 5 | 8 | relu |
| Output | 2 | linear |

## V. Performance

Experiments were conducted in order to characterize the performance of each agent. The purpose was to determine the behaviour of how each agent learns over time.

The experiments were set up as follows. The agent was exposed to the Pixelcopter game and was given thousands of randomly generated games as sample data. As the agent played each game, its model learned from the data. A snapshot of each model was taken at a certain interval of episodes. Using the snapshots of the trained agents, the agents were reconstructed at each snapshot. The performance of each agent was assessed by allowing each agent to play the Pixelcopter game 100 times, and instructed to pick the greediest reward each time. The total reward for each game was saved, and the mean and median reward of each agent was calculated.

### A. Q-Value Table, Sparse Discretization

Figure 4 shows the reward of the sparse discretization Q-table implementation of Q-learning.

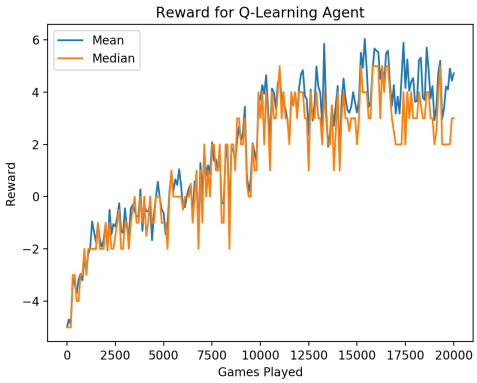



Figure 4: Average reward for Q-Learning Sparse agent over training data

The average reward seems to converge after 11000 episodes of gameplay. The mean reward plateaus at a value of 4. Figure 5 shows the minimum and maximum reward witnessed for each Q-learning agent. The agent accomplished a maximum score of 50. The minimum score for each iteration was lower than 0 reward despite the volume of training data used. This means that each of the agents experience games in which they lose the game immediately. This could be a result of the random initialization of the course path and the obstacle locations. There may be certain initializations that the agent does not know how to succeed in.

It was also of interest to observe the behaviour of how the Q-table is populated. The Q-table can be evaluated by how often each state is visited. If the majority of states are visited quite uniformly, it can be a good indication of a good selection of discretizations. A high number of visits for each state would allow the Q-table to have a better estimation of next immediate reward. Figure 6 shows the number of distinct state IDs which have been visited over number of training episodes.

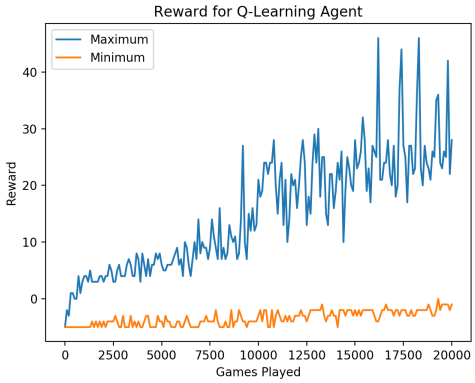



Figure 5: Minimum and maximum reward for Q-Learning Sparse agent over training data

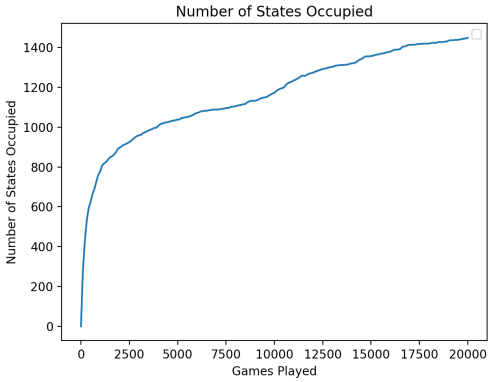



Figure 6: Number of visited states in the Q-table

By the 20000th episode, only 1.3% of all of the possible states have been visited. This would be a good indication that the discretization steps for each dimension were not chosen appropriately. It would have been beneficial to explore the probability distribution of each dimension in order to determine more useful ways that the state could be represented. This way, each state could be visited more uniformly, and a smaller table could be made as a result of the more precise discretization.

As well, around the convergence point of 110000 games played, the slope of the graph drastically decreases. This could attribute to the concept that the agent is converging because it does not discover new states as fast anymore. More training data would not have as much as a high impact to the agent because fewer new states would be explored.

### B. Q-Value Table, Dense Discretization

The same reward experiment was conducted for the Q-Value table agent with dense discretization. In this case, the number of episodes that were used for training was up to 200000, which is ten times the number of episodes of the previous agent. A larger timestep of 10000 was used in order to reduce computation time of snapshots.

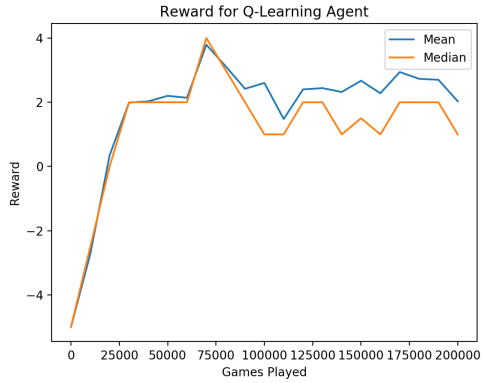



Figure 7: Average reward for Q-Learning Dense agent over training data

It appears as if the convergence of the reward for this agent requires much more data. It converges finally after 30000 episodes. Also, the convergence value of 2 is extremely low. The decrease in reward could be attributed to the fact that the number of states have been increased by over 100 times as a result of the discretization. This would make the values of the Q-table much more difficult to populate because each and every state would be visited far fewer times in a given timeframe.

The sparseness of the Q-table was also observed by counting the number of populated rows in the Q-table.

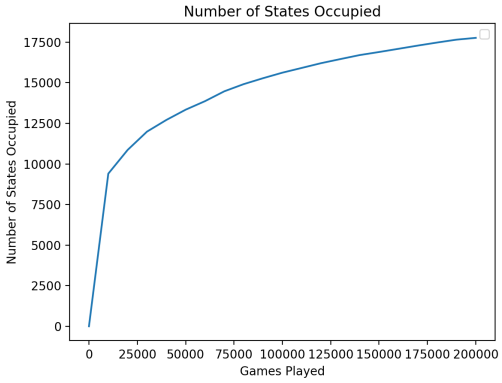



Figure 8: Number of visited states in the Q-table for Q-Table Dense agent

An extremely small percentage of states were visited for this agent. After 200000 episodes, which is 10 times the number of total iterations than the previous agent, only 0.095% of the possible states were visited. This is a natural consequence of increasing the number of possible states by over 100 times. This increase in scarcity for the Q-table reveals a fundamental consequence of using a Q-table with a large number of discretizations. Since the size of the table increases exponentially, it would be exponentially difficult to visit each and every state of the game at least once.

### C. Non-Visual Deep Q-Learning

Similar experiments were performed with the Deep Q-Learning Agent. The agent was trained over 15000 episodes, with snapshots of model weights being saved every 500. These results, illustrated in Figure 9, track the performance of the agent during training. The network converges after about 5000 games played with a mean reward of 37, which corresponds to about 8 seconds of play time. Training took approximately 3 hours to finish, with convergence at about 1 hour.

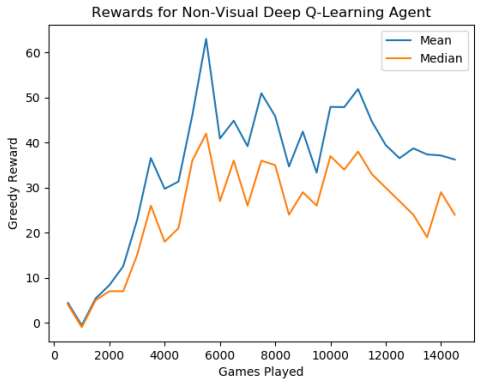



Figure 9: Average reward for non-visual deep Q-learning agent over training data

The minimum and maximum scores are displayed in Figure 10. Impressively, the model was able to obtain a high score of 355, or 75 seconds of play time. Also notable was that no matter how much the agent was trained, it still had the occasional “bad game” where it ended with a score of less than 10, indicating that the game ended almost immediately.

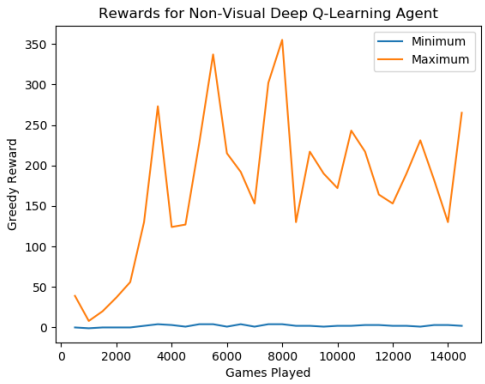



Figure 10: Minimum and maximum rewards for non-visual deep Q-learning agent over training data

The convergence of the model can also be seen in the model loss plot in Figure 11. This shows the loss plateauing around 3000 episodes, with the absolute minimum loss at 5000 episodes, corresponding to the convergence of training scores. This plot also displays how quickly the model responds to the high loss at the beginning, dropping two orders of magnitude

within 500 episodes. Fine-tuning the model from 500 to 5000 episodes is the most time-consuming.

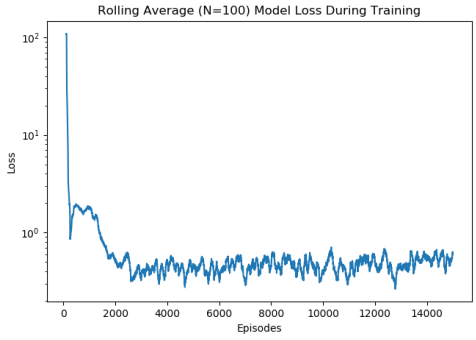



Figure 11: Rolling average (N=100) model loss over 15000 training episodes

Figure 12 shows the t-SNE embedding of game states represented by the last layer of DDQN. This plot was generated by letting the converged agent play 300 randomly generated episodes, resulting in around 50000 game states. The states were pre-processed using PCA and then used to find the t-SNE representation. The plot includes the maximum expected reward for each state; darker colours representing lower scores and vice versa. The following observations can be made:

- The distance to the next barrier is proportional to the x-coordinate of the graph.
- States where the pixel is above the next block, in line with the next block, and below the next block are near the top, middle, and bottom of the graph, respectively.

One can also see that the lowest expected rewards are in states where the next block is close and in line with the pixel. Rewards increase as the block gets farther away or the pixel moves up or down (relative to the block). Intuitively, this is exactly what we would expect and reveals that the agent is aware that it should not be perfectly in line with a close, oncoming block. This shows that the agent acts similarly to a beginner human player when a block is encountered. The agent marks it as a low reward scenario since it could potentially end the game. Obviously, an ideal agent would have high rewards in all scenarios.

## VI. Comparisons and Conclusions

Figure 13 displays the box plot of the converged agent's rewards. Since there has been no prior published research for this specific game, the agents are benchmarked against a beginner human player level. Each participant (N=4) was allowed to play the game for 5 minutes before playing 50 games each. Each set of 50 games was used for this box plot, as a somewhat accurate representation of a beginner human skill level. Table VII includes a metric for the percentage of games that score above the first Human quartile (i.e. better than 25% of human games), and also gives a quantifies summary of all agents tested. Note that a percentage of 75% here is the "most human" an agent can be.

The first point of interest is the comparison between the Sparse and Dense Q-Table implementations. The Sparse Table was better trained on average, however both implementations did not learn any notable amount  average scores were only 2 and 4. This lack of performance reveals the difficulty of discretizing states in the continuous domain. There is an inherent tradeoff: too sparse and the states are too general to accurately predict rewards, too dense and the states are not visited enough to converge to an optimum  not to mention memory issues. Improvements here could involve manual discretization, where domain knowledge could indicate where state buckets could be more precise and more sparse. In any case, this shows the difficulty required with discretization and the problems that classical Q-learning encounters in continuous domains.

Unsurprisingly, mainly the result of discretization issues described above, the DDQN agent was able to perform much better than tabulated Q-learning agents  slightly above the average beginner human. The neural network in the DDQN agent operates on continuous states and is able to interpolate between states and apply learning to new, unseen states. The network is able to extract features from the inputs and more efficiently predict things based on those specific features. Discretization in Q-learning is often generated arbitrarily; however, in the real world, states are non-uniformly-distributed since there are higher chances of certain states to appear rather than others. This demonstrates the robustness they have in continuous domains.

One benefit that deep Q-learning agents have over classic Q-learning agents is the space efficiency. The size of the Q-tables for QTS and QTD implementations were about 200000 and almost 2 million cells respectively, and both had less than 2% of their states visited. This is an incredibly inefficient use of space. On the other hand, the deep Q-network had less than 8000 weight values. Even with the 5000 size buffer for replay memory, corresponding to 45000 values, there is still a 51% decrease in size compared to the QTS implementation. Generalizing, however, there may be cases where the buffer size need to be longer to store more memories, and the memory savings could dwindle. An improvement to the Q-learning agents could be to only allocate memory for Q-table rows when that state is visited. This would massively reduce memory costs but bring much more manual overhead and potentially increase training time.

One benefit that the Q-learning agents have is the training time. Despite training episodes being around 2 times as many, the training time is 15 times faster. This is because classic Q-learning algorithms act on constant time memory, rather than computing backpropogation and training a neural network.

Table VII: Observations for each agent

| Metric | QTS | QTD | DDQN |
|---|---|---|---|
| Mean Reward | 4 | 2 | 39 |
| Convergence (episodes) | 11000 | 30000 | 5000 |
| Convergence (minutes) | 17 | 31 | 255 |
| Model Size (floats) | 218748 | 35431218 | 7904 |
| % Above Human Q1 | 48 | 29 | 95 |

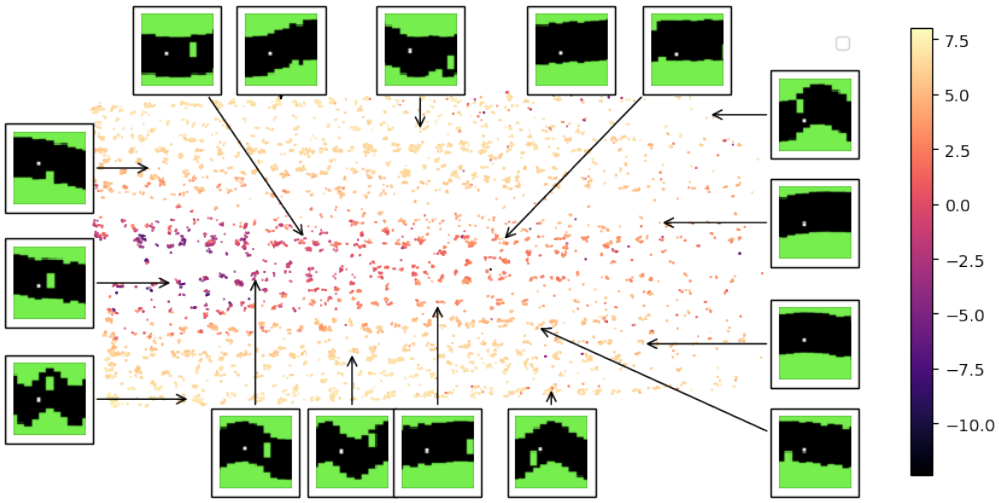



Figure 12: Two-dimensional t-SNE embedding of PCA reduced game states with maximum expected reward in the output layer of the DDQN

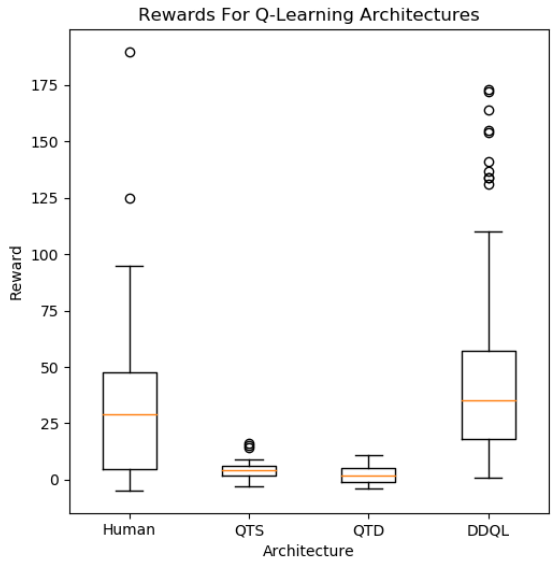



Figure 13: Box plot of rewards for the different architectures after convergence

The ANN-based improvements to Q-learning such as DQN and DDQN demonstrate the potential for RL in complex, continuous domains. Sidestepping discretization issues using deep Q-learning leads to significantly better performance, model convergence time, and space efficiency compared to that of table-based RL. Real-world applications are continuous, and if reinforcement learning is to be applied extensively, especially to autonomous navigation, it needs to be robust enough to handle continuous non-discrete domains.

## References


[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] A. G. R. Sutton, Richard S. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[3] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[4] G. Tesauro, "Temporal difference learning and td-gammon," *ICGA Journal*, vol. 18, no. 2, p. 88, 1995.

[5] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971

[7] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[8] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, vol. abs/1710.02298, 2017. [Online]. Available: http://arxiv.org/abs/1710.02298

[9] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *CoRR*, vol. abs/1611.01224, 2016. [Online]. Available: http://arxiv.org/abs/1611.01224

[10] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2018.

[11] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[12] Y. Zhang, P. Sun, Y. Yin, L. Lin, and X. Wang, "Human-like autonomous vehicle speed control by deep reinforcement learning with double q-learning," *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018.

[13] S. Kardell, M. Kuosku, and J. Karlsson, "Autonomous vehicle control via deep reinforcement learning," 2017.

[14] B. Tan, N. Xu, and B. Kong, "Autonomous driving in reality with reinforcement learning and image translation," *CoRR*, vol. abs/1801.05299, 2018.
[15] X. Pan, Y. You, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," *CoRR*, vol. abs/1704.03952, 2017.
[16] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *CoRR*, vol. abs/1507.00814, 2015. [Online]. Available: http://arxiv.org/abs/1507.00814
[17] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016.
[18] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3338–3346. [Online]. Available: http://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf
[19] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2863–2871. [Online]. Available: http://papers.nips.cc/paper/5859-action-conditional-video-prediction-using-deep-networks-in-atari-games.pdf
[20] J. Koutník, G. Cuccu, J. Schmidhuber, and F. J. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in *GECCO*, 2013.
[21] J. Koutník, J. Schmidhuber, and F. J. Gomez, "Online evolution of deep convolutional network for vision-based reinforcement learning," in *SAB*, 2014.
[22] Y. Liang, M. C. Machado, E. Talvitie, and M. Bowling, "State of the art control of atari games using shallow reinforcement learning," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, ser. AAMAS '16. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 485–493. [Online]. Available: http://dl.acm.org/citation.cfm?id=2936924.2936996
[23] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *CoRR*, vol. abs/1611.05397, 2016. [Online]. Available: http://arxiv.org/abs/1611.05397
[24] M. V. Otterlo and M. Wiering, "Reinforcement learning and markov decision processes," *Adaptation, Learning, and Optimization Reinforcement Learning*, pp. 3–42, 2012.
[25] V. Kuleshov and D. Precup, "Algorithms for the multi-armed bandit problem," *Machine Learning Research*, vol. 1, 2000. [Online]. Available: https://arxiv.org/pdf/1402.6028.pdf
[26] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698
[27] C. Watkins, "Learning from delayed rewards," 01 1989.
[28] V. Tadi, "Convergence analysis of temporal-difference learning algorithms with linear function approximation," *Proceedings of the twelfth annual conference on Computational learning theory - COLT 99*, May 1999.

## APPENDIX A
## CLASSIC Q-TABLE AGENT SOURCE CODE

```
import numpy as np

class QTable:
"""
    Implements a tabulated Q-Table.
"""
def __init__(self, num_states, num_actions,
             default_value=0, alpha=0.1,
             gamma=0.3, epsilon=0.1):
    self.q_table = np.ones([num_states, num_actions])
    self.default_value = default_value
    self.alpha = alpha
    self.gamma = gamma
    self.epsilon = epsilon

def instantiate_state(self, state, action):
    self.q_table[state, action] = self.default_value
    return self.default_value

def get_q_value(self, state, action):
    if (state, action) not in self.q_table:
        return self.instantiate_state(state, action)
    return self.q_table[state, action]

def get_best_action(self, state):
    return np.argmax(self.q_table[state])

def update(self, next_state, state, action, reward):
    old_value = self.get_q_value(state, action)
    next_max = np.max(self.q_table[next_state])

    new_value = (1 - self.alpha) * old_value + self.alpha * \
        (reward + self.gamma * next_max)
    self.q_table[state, action] = new_value


class QAgent:
"""
    Implements a Classic Q-Learning Agent.
"""

def __init__(self, env, num_frames, frame_skip):

    self.env = env
    self.num_frames = num_frames
    self.frame_skip = frame_skip
    self.rng = np.random
    self.current_frame = 0

    self.prev_state_id = None
    self.current_state_id = None
    self.current_reward = 0
    self.prev_action = 0
    self.state = self.env.getGameState()
    self.actions = self.env.getActionSet()
    self.num_actions = len(self.actions)
    self.num_states = 125000 # HERE
    self.q_table = QTable(self.num_states, self.num_actions)
    self.model = None

def get_state_id(self, state):
    player_y = self.get_player_y(state)
    player_vel = self.get_player_vel(state)
    player_dist_to_ceil = self.get_player_dist_to_ceil(state)
    player_dist_to_floor = self.get_player_dist_to_floor(state)
    next_gate_dist_to_player = self.get_next_gate_dist_to_player(state)
    next_gate_block_top = self.get_next_gate_block_top(state)
    next_gate_block_bottom = self.get_next_gate_block_bottom(state)

    return player_y * pow(5, 0) + \
           next_gate_block_top * pow(5, 1) + \
           player_dist_to_ceil * pow(5, 2) + \
           player_dist_to_floor * pow(5, 3) + \
           next_gate_dist_to_player * pow(5, 4) + \
           next_gate_block_bottom * pow(5, 5) + \
           player_vel * pow(5, 6)

def apply_bounds(self, val, min_val, max_val):
    return min(max(val, min_val), max_val)

def get_player_y(self, state):
    # Possible states: [<8, 16, 24, 32, >40]
    val = int(round(state['player_y']/8.0) * 8)
    return int(self.apply_bounds(val, 8, 40)/8) - 1

def get_player_dist_to_ceil(self, state):
    # Possible states: [<4, 8, 12, 16, 20]
    val = int(round(state['player_dist_to_ceil']/4.0) * 4)
    return int(self.apply_bounds(val, 4, 20)/4) - 1

def get_player_dist_to_floor(self, state):
    # Possible states: [<4, 8  12, 16,  20]
    val = int(round(state['player_dist_to_floor'] / 4.0) * 4)
    return int(self.apply_bounds(val, 4, 20)/4) - 1
```

```
def get_next_gate_dist_to_player(self, state):
    # Possible states: [<8, 16, 24, 32, >40]
    val = int(round(state['next_gate_dist_to_player'] / 8.0) * 8)
    return int(self.apply_bounds(val, 4, 40)/8) - 1

def get_next_gate_block_top(self, state):
    # Possible states: [<8, 16, 24, 32, >40]
    val = int(round(state['next_gate_block_top'] / 8.0) * 8)
    return int(self.apply_bounds(val, 8, 40)/8) - 1

def get_next_gate_block_bottom(self, state):
    # Possible states: [<8, 16, 24, 32, >40]
    val = int(round(state['next_gate_block_bottom'] / 8.0) * 8)
    return int(self.apply_bounds(val, 8, 40)/8) - 1

def get_player_vel(self, state):
    # Possible states: [ <-1.5, -1,  0.5, 0, 0.5, 1.0, >1.5]
    val = round(state['player_vel'] / 0.5) * 0.5
    return int(self.apply_bounds(val, -1.5, 1.5)/0.5) + 3

def update_q_table(self, reward=None):
    reward = self.current_reward if not reward else reward
    self.q_table.update(self.current_state_id,
                        self.prev_state_id,
                        self.prev_action,
                        reward)

def get_q_table(self):
    return self.q_table

def act(self, state, epsilon=0.05):
    self.current_state_id = self.get_state_id(state)

    if self.prev_state_id:
        self.update_q_table()

    if self.rng.rand() > 0.9:
        action = 0
    else:
        action = 1
    """
    if self.current_frame < 1000:
        print(self.current_frame)
    else:
        if self.rng.rand() > epsilon:
            action = self.q_table.get_best_action(self.current_state_id)
# exploit
    """

    if self.rng.rand() > epsilon:
        action = self.q_table.get_best_action(self.current_state_id)

    reward = 0.0
    for i in range(self.frame_skip):
# we repeat each action a few times
        # act on the environment
        reward += self.env.act(self.actions[action])

    self.current_reward = reward
    self.prev_action = action
    self.prev_state_id = self.current_state_id
    self.current_frame += 1

    return reward, action

def start_episode(self, N=3):
    self.env.reset_game()  # reset
    for i in range(self.rng.randint(N)):
        self.env.act(self.env.NOOP)  # perform a NOOP

def end_episode(self):
    self.state.clear()
```

## APPENDIX B
## DDQN AGENT SOURCE CODE

```
class DDQNAgent:
def __init__(self, state_size, action_size):
    self.state_size = state_size
    self.action_size = action_size
    self.batch_size = 64
    self.memory = deque(maxlen=20000)
    self.gamma = 0.97    # discount rate
    self.epsilon = 1.0  # exploration rate
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.99995
    self.learning_rate = 0.0004
    self.tau = .125
    self.train_on_TPU = False #Won't work if True
    self.model = self._build_model()
    self.target_model = self._build_model()

def _build_model(self):
    # Neural Net for Deep-Q learning Model
    model = Sequential()
    model.add(Dense(32, input_dim=self.state_size, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(self.action_size)) # default is linear activation
    model.compile(loss='mse',optimizer=Adam(lr=self.learning_rate))
    return model

def remember(self, state, action, reward, next_state, done):
    self.memory.append([state, action, reward, next_state, done])

def act(self, state):
    if np.random.rand() <= self.epsilon:
        return np.random.choice([0, 1], p=[0.25, 0.75])
    act_values = self.model.predict(state)
    return np.argmax(act_values[0])  # returns action

def replay(self):
    minibatch = random.sample(self.memory, self.batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target = (reward + self.gamma *
                      np.amax(self.target_model.predict(next_state)[0]))
        target_f = self.model.predict(state)
        target_f[0][action] = target
        self.model.fit(state, target_f, epochs=1, verbose=0)
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

def replay_batch(self):
    minibatch = random.sample(self.memory, self.batch_size)
    states, targets_f = [], []
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target = (reward + self.gamma * np.amax(self.target_model.predict(next_s
        target_f = self.model.predict(state)
        target_f[0][action] = target
        # Filtering out states and targets for training
        states.append(state[0])
        targets_f.append(target_f[0])

    history = self.model.fit(np.array(states), np.array(targets_f), batch_size=self.
    # Keeping track of loss
    loss = history.history['loss'][0]
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
    return loss

def replay_batch_gpu_optimized(self):
    # Check the above reference implementation for correspondance
    minibatch = np.array(random.sample(self.memory, self.batch_size))
    state = np.array(minibatch[:, 0].tolist()).squeeze()
    action = minibatch[:, 1]
    target = minibatch[:, 2]
    next_state = np.array(minibatch[:, 3].tolist()).squeeze()
    done = np.array(minibatch[:, 4], dtype=bool)
    Q_next_max = self.gamma * np.amax(self.target_model.predict(next_state, batch_si
    target = target + (Q_next_max * np.invert(done))
    target_f = self.model.predict(state, batch_size=self.batch_size)
    target_f[range(self.batch_size), action.tolist()] = target

    history = self.model.fit(state, target_f, batch_size=self.batch_size, epochs=1,
    # Keeping track of loss
    loss = history.history['loss'][0]
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
    return loss
```

```
def target_train(self):
    weights = self.model.get_weights()
    target_weights = self.target_model.get_weights()
    for i in range(len(target_weights)):
        target_weights[i] = weights[i] * self.tau + target_weights[i] * (1 - self.tau)
    self.target_model.set_weights(target_weights)

def load(self, name):
    self.model.load_weights(name)
    self.target_model.load_weights(name)

def save(self, name):
    self.model.save_weights(name)
```