ENGG*3130 Final Project Report

Deep Reinforcement Learning with Arcade Games

LAURA MURPHY, ERIC FAGUY, and JOSH NEAR

This project focuses on deep reinforcement learning. Specifically deep Q learning algorithms being used in the classic arcade games of Tetris, Pong, and Space Invaders. Deep-Q learning is a value-based reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. Typically deep reinforcement learning algorithms outperform more traditional methods in both processing and performance. In this project we found the Deep-Q learning will produce very different results in different games.

CCS Concepts: • **Computing methodologies** → **Reinforcement learning**; Neural networks.

ACM Reference Format:
Laura Murphy, Eric Faguy, and Josh Near. 2020. ENGG*3130 Final Project Report: Deep Reinforcement Learning with Arcade Games. 1, 1 (May 2020), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This project is focused on analyzing the performance of Deep Q-learning on three arcade video games, this game are Pong, Tetris, and Space Invaders. We will examine why each game produces the results they do as well as the similarities and differences between the three games. Figure 1 is a visible representation of our project, it is being used to show both the deep Q networks as well as the reinforcement learning taking place in the arcade games. All the code used for this project can be found at https://github.com/efaguy27/engg-3130-final.

1.1 Background Information

1.1.1 Q-learning. Q-learning is an off-policy reinforcement learning algorithm used to find the best action to take when given a current state [1]. What considers this off-policy is the fact that the Q-learning function learns from actions outside the current policy, such as taking random actions, which then shows that a policy is not necessary. This means Q-learning searches to locate a policy that is able to maximize the total reward. The "Q" in Q-learning stands for quality, which represents how effective the action is in gaining a prospective reward.

For Q-learning to be performed a Q-table or matrix needs to be created to follow the structure [state, action] where the values are initialized to zero. The values are then updated and stored after an episode. This table then becomes a reference for the agent to select the best action due to the Q-value. The next step is for the agent to communicate with the environment and make updates to the Q-table. The agent can interact with the environment in two ways. The first option is to use the table as reference and analyze each action for each state. The agent then chooses the best action from the maximum value of the actions. This option has been given the name exploiting as the information available is

Authors' address: Laura Murphy, lmurph05@uoguelph.ca; Eric Faguy, efaguy@uoguelph.ca; Josh Near, jnear@uoguelph.ca.

Manuscript submitted to ACM

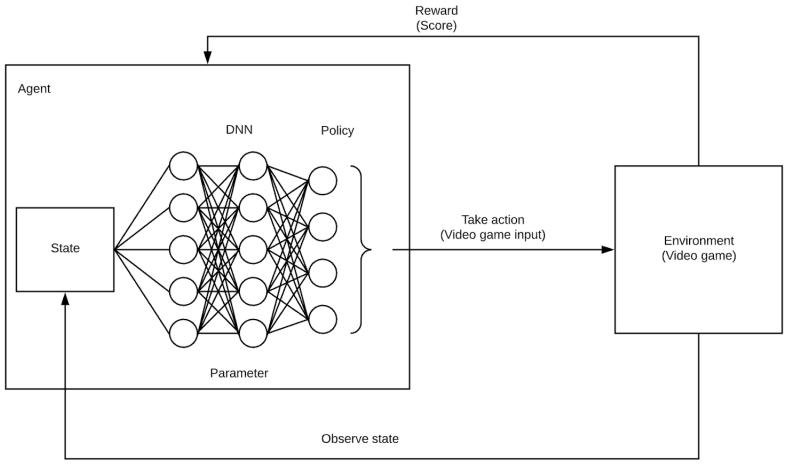

Fig. 1. A representation of our final project.

being used in order to make the decision. The second option is to take a random action. The agent explores new states that would have not been selected in the exploiting option. This process has been termed, exploring. The two processes are balanced by using the value epsilon (and setting it to how often the user would like to exploit or explore. In order to update the Q-table, updates occur after each step and ends once the episode has been completed, which is when a terminal point has been reached. A terminal state can mean a variety of things such as reaching the end of a game or completing a desired objective. The agent does not tend to learn much after a single episode but will continue to learn more upon further exploration. It will then converge and learn the optimal Q-values. The three basic steps are as follows: firstly, the agent begins in a state, takes an action and then receives a reward. In the second step, the agent selects the action by referencing the Q-table with the highest value or by random, and the third and final step is to update the Q-table.

Some common variables that are used in Q-learning are the learning rate, gamma, reward, and max. The learning rate is how accepted the new value is versus the old one. Once the difference between the old and new value is determined, it is multiplied by the learning rate. That value then gets added to the previous Q-value which then moves it in the direction of the latest update. Gamma is the discount factor. It is what is being used to balance all rewards. Typically, this number ranges from 0.8 to 0.99. Reward is the value received when an action is completed at a given state. Lastly, max uses the Numpy library and takes the maximum of the prospective reward and applies it to the reward of the current state. The reason for this is that it impacts the current action by the possible future reward.

The agent performs a sequence of events that eventually generates the maximum total reward [2]. The Q-value being calculated states that the maximum future reward is the reward the agent received for entering the current state (s) plus the maximum future reward for the future state (s') [3]. The equation shown in Figure 2 is known as the Bellman equation.

The Q-value is generated from the current state (s) and action (a) is the reward (r(s, a)) plus the highest q-value possible from the next state (s'). Gamma is the discount factor that controls the addition of rewards in the future. Being

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

Fig. 2. The Bellman Equation

shown below Q(s', a) depends on Q(s", a) which has a coefficient of gamma squared. Therefore, showing that the Q-value depends on the Q-value of future states.

1.1.2 Deep Q-learning. Deep Q-learning is an extension of Q-learning used when the environment has too many states for normal Q-learning to be effective as the Q-table would be too large [3]. In deep Q-learning a deep neural networks is used to approximate the Q-value function. The state is given as the input and the Q-value of all the possible actions is then returned as the output. The steps involved in reinforcement learning using deep Q-learning networks are that first, all past experiences are initially stored in the user memory. Secondly, the next action is then determined by the maximum output of the Q-network. Finally, the loss function, which is the mean squared error of the predicted Q-value and the target Q-value (Q*), which therefore makes this a regression problem. At this point though, the target value is unknown as this is a reinforcement learning problem. Now, back to the Q-value update equation derived from the Bellman equation shown below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Fig. 3. TEMP

The section circled in green represents the target. R is the true reward and the network updates the gradient using back-propagation to finally converge. All of the steps in deep Q-learning are to first preprocess and give the game screen (state s) to the deep Q-network which will then return the Q-values of all actions of this state. An action must then be selected using the epsilon-greedy policy. With this probability epsilon a random action (a) is selected and with a probability of 1-epsilon, an action is then selected with the maximum Q-value. The action is then performed in the state s and moved to the new state (s') and receives a reward. S' is the preprocessed screen of the next game screen and the transition in stored in the replay buffer. Random batches of transitions are then sampled from the replay buffer and the loss is calculated using the equation below.

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

Fig. 4. Loss Function

The loss equation is the squared difference between the target Q and the predicted Q. A gradient descent is then performed with respect to the actual network parameters to minimize loss. After every C iteration, the actual network weights are copied to the target network weights. These are all then repeated for an M number of episodes.

2 EXPERIMENTS

2.1 Pong

The code used for Pong was created in the Autonomous Learning Library, a deep reinforcement learning library for PyTorch [4]. The Autonomous Learning Library simplifies the building of reinforcement learning agents as it is agent based rather than algorithm based. The Deep Q-learning described and used expands on the algorithm behind Q-learning, which allows it to work effectively with modern convolution networks. The main innovation implemented is the replay buffer which allows for batch-style updates of decorrelated samples. This program also includes a "target" network which will improve the stability of the updates. The experiments that were run on this program were; changing the minibatch size from 32 to 16 and 64.

2.2 Tetris

The code used for tetris was found on https://github.com/nuno-faria/tetris-ai [5]. This code works by implementing a basic Q learning algorithm in the form of $Q_{state} = reward + discount(Q_{nextstate})$.

The best the action is chosen is slightly different from most deep Q-learning implementations. Normally there is an output vector, from the neural network, of all values for a certain state, where each value corresponds to an action, and the action with the highest value is chosen. In this implementation in each state all the possible next states for all the possible actions are collected, and the states are inputted into the neural network to predict the reward and the action that leads to the state with the highest reward will be chosen.

The reward for each state is calculated by $1+(linescleared)^2(boardwidth)$, where in this case board width is constant at 10.

Three different experiments were run each with a different training batch size.

The following parameters were used across all experiments:

- Episodes = 2000
- Epsilon stop episode = 1750
- Start Epsilon = 1
- Stop Epsilon = 0
- Memory size = 20000
- Discount = 0.95
- Epochs = 1
- Training start size = 2000
- Neurons = [32, 32]

The three batch sizes used were 256, 512, and 1024.

2.3 Space Invaders

The code used for Space Invaders was found on https://github.com/openai/gym [6]. The code used a form of deep q-learning from the open source artificial intelligence library OpenAI Gym. This library is set up to play a variety of Atari games including Space Invaders. The Space Invaders code works by performing an action for 2,3, or 4 frames. Possible actions in Space Invaders are move left, move right, do nothing, and shoot. In the case of shooting, that can be performed while moving and is a special case for this reason.

3 RESULTS

3.1 Pong

The first experiment completed was to run the program with a minibatch size of 32. See Figure 5. This experiment did not result much information except that it took approximately 1250 returns for the results to converge. The minibatch size is the number of experiences that are sampled in each training update. The trials started by receiving scores ranging from -19 to -21, after around 850 episodes the score then began to circle around the value zero, then from 1250 and after the values consistently ranged from 19 to 21. Due to the game having a maximum score of 21, there is plateau after 1250 returns seen in Figure 5. The epsilon value would start at one then decrease and hit zero after around 800 episodes when the values began to increase.

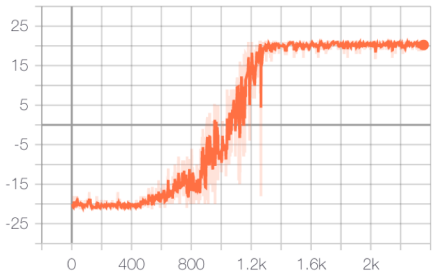

Fig. 5. Pong with a minibatch size of 32.

In the second experiment the minibatch size was halved to the value of 16. It was expected that the results would take longer to converge as the program is now looking at less values for each training update which allows the program less data to learn from. This, however, was not the case. As seen in Figure 6, the blue line representing the minibatch of 16 actually converged faster than the minibatch of 32.

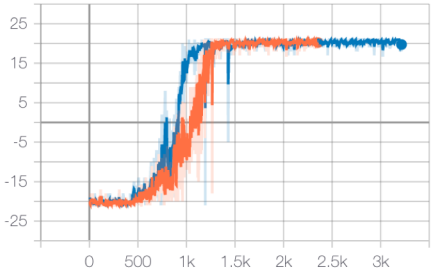

Fig. 6. Pong with a minibatch size of 32 (orange) and 16 (blue).

The experiment was then done again but this time by doubling the original minibatch size and changing it to 64. From Figure 7, with the orange line representing the minibatch of 32 and the grey line representing the minibatch of 64 it is clear that yet again there is inconsistent data.

From this figure it is clear that this time it is actually following the original hypothesis but when comparing this figure to Figure 6 it seems to be almost overlapping the blue line representing the minibatch size of 16. The results of

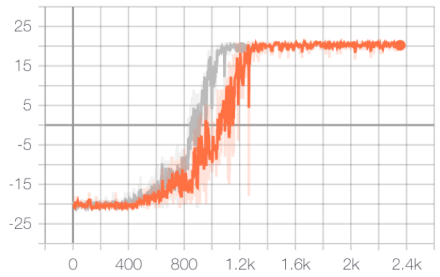

Fig. 7. Pong with a minibatch size of 32 (orange) and 64 (blue).

the three experiments show that the value of the minibatch has little to no impact on the rate at which the program converged. There is little consistency in the graphs and shows that even though there was double or four times the amount of points being evaluated, it still took the same if not longer amount of time to converge. Figure 6 does not just show how both took around the same time to converge, it also shows that with a larger minibatch size came larger fluctuations, showing that there is a high variance. From this outcome it is obvious that there is something else in play that is causing the large range of penalties for a given example. From here it was beneficial to review the expected value of the parameter. It is suspected that possibly when the program was randomly selecting the data points to evaluate, this is the reason behind the inconsistent data. Due to the different experiments not having the same input values that may be the reason for the unsuspected results. From this it may be beneficial to try to give the program a set random seed variable which would allow for some uniformity and to further evaluate the results.

3.2 Tetris

Figure 8 shows the scores for the three tetris experiments that were run. Only the second half is shown to allow more detail to be shown, as the first half is consistently low scores. For the same reason extremely high scoring episodes are cut off. (A small note, the 512 batch size experiment crashed at episode 1859 due to a memory overload)

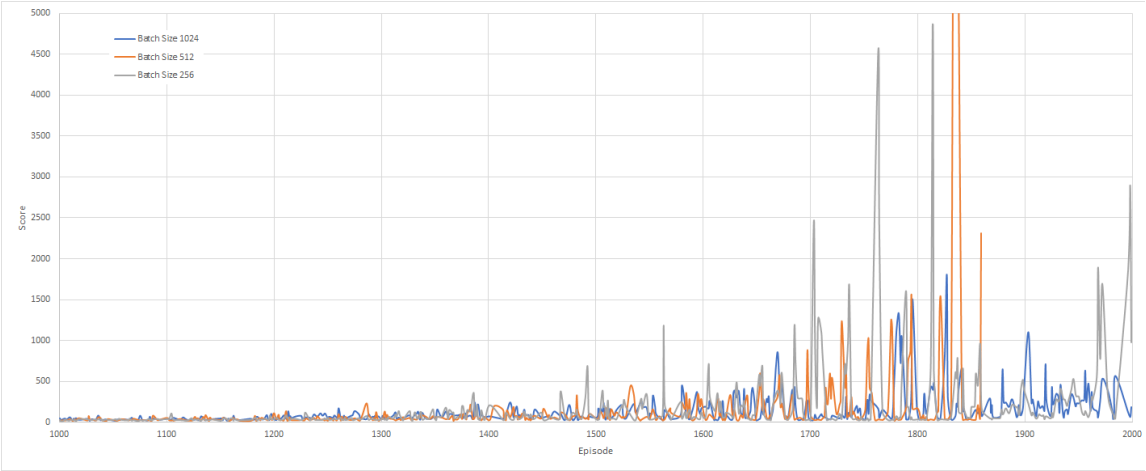

Fig. 8. The scores for the three tetris experiments.

From this graph we can see that even after the model if always choosing the best moves, after episode 1750, the model is still very unstable having some very high and some low episodes. This is most likely due to the large randomness and unpredictability in the game of tetris. We can also see that the batch size doesn't really affect the model in any discernible way as all the experiments return similar scores.

Q learning, and by extension Deep-Q Learning, is very good for environments where there is a relatively small number of possible states and actions. While tetris doesn't have a large number of actions, the possible state are very large especially compared to the other games examined. Due to the randomness and large number of state in tetris it is possible that the agent was put into a state that it did not have similar experience to, and therefor made a bad move. In Tetris a single bad move can also quickly cause you to lose the game. More detailed analysis of this can be found in the accompanying notebook.

We can also see that the 512 batch size experiment had a few runs that we much higher scoring than anything else, so much so that it caused the code to crash. This can be explained by same reasons listed above, the randomness inherit in tetris and in the training process. If these randomnesses "line up", i.e. the agent is given the same pieces in the same order multiple times, the agent can score extremely high.

4 CONCLUSION

From the results we can see that the same reinforcement learning algorithm will produce greatly different results for different games. For Pong we saw that the results quickly reach the max score after the training was complete, but for Tetris the results were extremely varied in the fully trained model.

REFERENCES

[1] A. Violante, "Simple reinforcement learning: Q-learning," Jul 2019.
[2] V. Valkov, "Solving an mdp with q-learning from scratch - deep reinforcement learning for hackers (part 1)," Apr 2019.
[3] A. Choudhary and I. B. Graduate, "Introduction to deep q-learning for reinforcement learning (in python)," May 2019.
[4] C. Nota, "The autonomous learning library." https://github.com/cpnota/autonomous-learning-library, 2020.
[5] nuno faria, "tetris-ai," Sep 2019.
[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.