An Application of Deep Q-Learning to Playing Pac-Man

ECS 171 Fall 2018 Final Report

Tanushree Bisht, Rishi Dutta, Pu (Percy) Feng, Calvin Leng, Jingwen Low, Alex Mirov, Louie Shi, Prajakta Surve, Mithun Vijayasekar, Julian Wilkinson, Nikhil Yerramilli, and Jacob Young

Abstract

We construct an agent to play the classic 80s arcade game, Pac-Man, through an implementation of the Deep Q-Network Algorithm—a machine learning algorithm first introduced in 2013 that combines ideas from reinforcement learning and artificial neural networks to play Atari games. We find that our agent performs at a level comparative to humans. We elaborate on the architectural decisions in our implementation and the possibility of generalization to other classic arcade games.

Contents

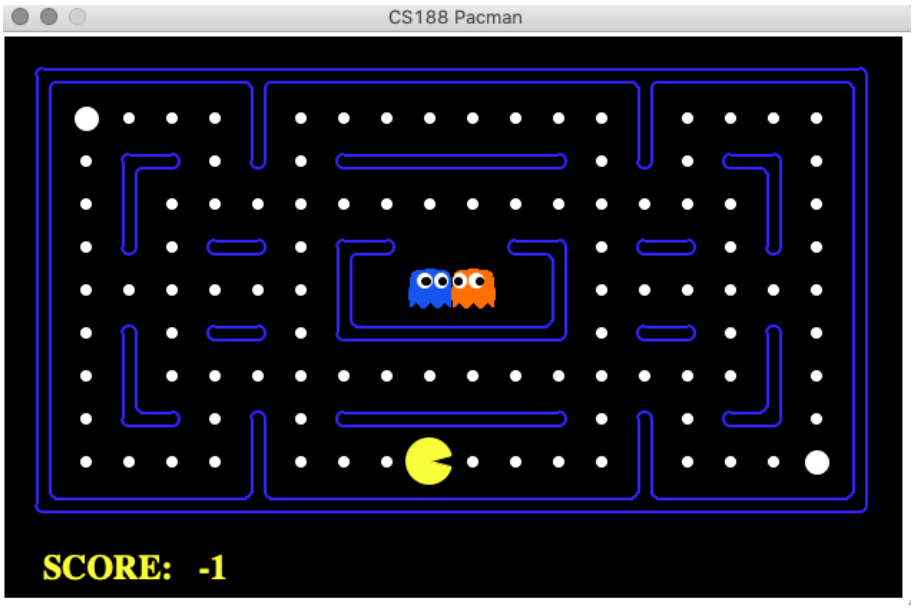


Figure 1: The displayed information during a game of Pac-Man.

1 Introduction

Reinforcement learning algorithms have recently found major success in creating agents that are capable of outperforming humans in various retro arcade games, such as Space Invaders, Pong, Breakout, and many others [8, 11]. In this paper, we implement the Deep Q-learning algorithm, a machine learning algorithm introduced in 2013 by Mnih et al. [8] that borrows notions notions from reinforcement learning and artificial neural nets, to develop an agent capable of learning how to play the popular arcade game—Pac-Man.

For the uninitiated, Pac-Man is an arcade game where the player controls a yellow character, named Pac-Man, and navigates a maze containing various items, i.e. pellets, fruits, and ghosts, as in Figure 1. If Pac-Man steps on an item, we say it *eats* it. The goal of the game is to attain the highest score, which can be achieved by eating pellets, fruits, and ghosts (assuming that a special power-up pellet has been eaten). We choose to use a version of the game implemented from Stanfords CS221 course, which can be in found in the following link:

http://stanford.edu/~cpiech/cs221/homework/prog/Pac-Man/Pac-Man.html.

The reason we choose the Deep Q-learning algorithm for our purposes is because the algorithm allows the agent to learn solely based off its environment, specifically, in the form of visual input. This is perfect for the purposes of Pac-Man, since the game map can easily be visualized as a discrete bitmap of various objects, i.e. Pac-Man, ghosts, fruits, and walls. Moreover, the Deep Q-learning algorithm excels in environments where rewards are often sparse, noisy, and delayed [8], which all holds true for Pac-Man.

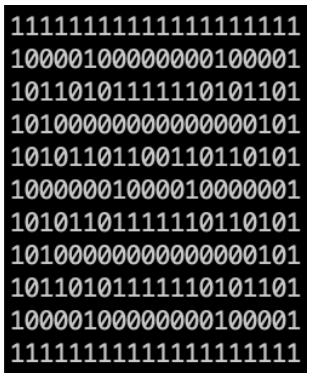


Figure 2: (Top) The matrix representation of the walls at a given state of the game. (Bottom) A kernelled 5×5 grid around Pac-Man.

We find that our agent performs at a level comparative to humans, scoring up to 800 points in our version of Pac-Man.

As for the implementation of the project, there are two main portions to implement: the game controller and the neural net. Specifically, the game controller will be the part of the agent that sends over controls to the game, just as a human would, while the neural net portion of the agent that learns and decides the next actions. As a result, we decided to split the project into two smaller sub-projects: the game controller and the neural network itself.

2 Background and Overview

The game controller is responsible for generating and formatting the inputs into the Artificial Neural Network (ANN). In order to feed the data into the ANN, we chose to represent the data in the following manner. In the `gameState` class, a hash table `boards` holds the string representation of the locations of each object such as the walls, Pac-Man, ghosts, pellets, and power-ups at each step of the game where a `0` in the string indicates that the object is not there, while a `1` indicates otherwise. After converting to this representation, since feeding in entire grids of the maze would be too slow for the ANN, the grids were additionally kernelled into smaller 5×5 and 7×7 grids around Pac-Man. An example of such representations can be seen in Figure 2.

The neural network side of the project is implemented with the Deep Q-learning algo-

rithm. The foundation of the algorithm is simply an ANN appended with some memory which remembers the sequence of states and actions previously taken in the game. The feature of this algorithm that lets it excel for the purpose of learning various of these old school games boils down to how loss is calculated which in turn affects how the weights are updated. The rough overview of the algorithm is that it attempts to learn the function Q^* where $Q^*(s, a)$ is the optimal value of the sequence s for the next action a. Thus we see that the optimal strategy is to select the action a which maximizes

$$r + \gamma Q^*(s, a)$$

where r is the current score and γ is the decay rate. From this, we see that

$$Q^*(s, a) = \mathbb{E}_{s\sim\mathcal{E}}\left[r + \gamma \max_a Q^*(s, a)\middle| s, a\right] \tag{1}$$

where $\mathcal{E}$ is the environment that the agent interacts with. In practice, updating with this loss function is impractical, thus it is common to approximate this loss function with a linear function. However, the deep Q-learning method chooses to use an ANN to approximate this loss function Q^*. To this end, the Q-Network is trained by minimizing a sequence of loss functions

$$L_i(\Delta_i) = \mathbb{E}_{s,a\sim p(\cdot)}\left[(y_i - Q(s, a; \delta_i))^2\right]$$

for each iteration i where y_i is the predicted value given in (1) [8]. With this loss function, we take the gradient as usual and update the weights with stochastic gradient descent.

The entire implementation of this method is all encapsulated in the method `learn`. Lastly, the purpose of the memory mechanism is to smoothen the training distribution over many past behaviors through a random sampling of the remembered sequences of states and actions from the memory at each training step. With the ANN trained, it is thus an approximate of the function Q^*, which provides the optimal move given a sequence of states and actions. With this, the game controller can simply call the function `getNextMove()` which will essentially predict the next move given the current state of the game, i.e. categorize the next move $a \in \{\text{North}, \text{South}, \text{West}, \text{East}, \text{Stop}\}$ where the input to the ANN is a vector representation of the game state, including game map and score.

3 Game Controller

3.1 Pac-Man Game and Version

We implement this project in pure Python 2.7, i.e. this project does not depend on any packages external to the standard Python distribution.

We use a version of Pac-Man from Stanford University's Artificial Intelligence Multi-Agent Pac-Man projects codebase. This version and its consequent API was developed at UC Berkeley by John DeNero (denero@cs.berkeley.edu), Dan Klein (klein@cs.berkeley.edu) and Pieter Abbeel (pabbeel@cs.berkeley.edu).

The functionality is the same as the original game's asides from the following:

- there are no extra lives;
- there are no fruits;
- there are no exits on the sides of the board and, consequently, no way to "teleport" to the other side;
- the score decreases as a function of time; and
- the ghosts are random and do not attack algorithmically.

Though there are differences from the original Pac-Man game, this version was chosen because it was written in Python and contained skeleton code that allowed for easier development of the neural network agent. In fact, the code was originally used to allow students to write an algorithm for the game; instead, we trained a neural network. More information can be found at the web page below:

http://inst.eecs.berkeley.edu/~cs188/Pac-Man/home.html.

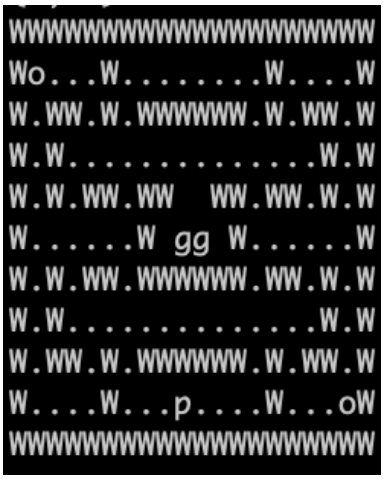
3.2 Custom Pac-Man API

In order to feed the game state into the ANN, we must construct a custom Pac-Man API to organize the data of the current state of an existing Pac-Man game. This data is then used by the ANN to train the model and is passed into the ANN API. At each frame of the game, the API is used to provide the ANN with information, such as the position of Pac-Man on the board, the activity of the ghosts, the location of the food (pellets and power-ups), the score, and so on. Here the game state is parsed and formatted in a manner that will be easy for the ANN to use and train.

The custom Pac-Man API formats the state of the Pac-Man game into a long string that the neural network can understand. The state is transformed into a grid, separated into grids of separate elements (yielding equally sized grids for pellets, ghosts, etc), and is then flattened into a string (as in Figure 3) used to represent each character on the grid. Additional information is also supplied, like the game state after an action is performed, the score of the game, and the time the ghosts are vulnerable after Pac-Man consumes a power-up. The grids are also reduced relative to Pac-Man's position, via a kernelling technique, and are supplied along with the full sized grids. This is discussed in the section that follows.

Because this assignment was originally used for an AI class, there was already code in place to allow for custom agents to control Pac-Man. The preexisting code structure of custom agents allows each agent the ability to `getAction()`, a method which takes in the current gameState object and returns a legal move, i.e. north, south, east, west, or stop. The structure is set up so that each agent can customize their own versions of this method. For this project, a new agent, named `NNTrainingAgent` was created that both parsed the game state and acted as a bridge between the Pac-Man game instance and the neural network.

The internal Pac-Man game code is already equipped with several methods pertaining to the acquisition and usage of game states. Most notably, the default code contains a class



Character	Game Object
W	wall
.	pellet
o	power-up
p	Pac-Man
g	ghost

Figure 3: (Top) Matrix representation of game map. (Bottom) Legend for game map.

called `gameState`, which when stringified, returns a matrix of characters that corresponds to the current running state of the game. This matrix is a 2-dimensional representation (with character tokens) of the absolute and relative positions of all entities, walls, and spaces in gameState for the current iteration. For example, a `W` at a position (x, y) indicates that a wall block exists at the coordinate (x, y). Because the matrix and the stringification of `gameState` can be constructed dynamically while the game is running, the matrix can be parsed into formats that are suitable for the neural network at every iteration. The ANN would then be able to use this stringified data in order to output a legal move. This is done in the `getAction()` method that is common to all agents.

If the game state of the entire game board were to be parsed into a single or multiple strings, there would be a lot of redundant and excessive data. In order to remedy this, we decided to give the ANN more localized information via a *kernelling* technique. Kernelling refers to the process in which data outside a certain radius from the Pac-Man is stripped, leaving only the local and immediate environment to be used. The ANN was provided with extra information by using grids centered around Pac-Man that was tunable to be 5×5 or 7×7. Said kernelling was done to the one-hot encoding of the 4 main features in `gameState`: walls, pellets, power-ups, and ghosts. Instead of passing in matrices into the ANN, the kernels were represented as bitmap strings with `1`'s indicating the feature is present and `0`'s indicating it is not.

In order to transform the original game state character matrix (which is represented as the matrix of different characters), each character is parsed into a separate string. For example, there is a string that represents the one-hot encoding of all the positions of walls that exist in the game, where each 1 in the string represents the existence of a wall block at that particular mapped coordinate. There is another string that represents the one-hot encoding of where the pac-man is, where the 1 in the string corresponds to the location that the string maps to in the 2-D coordinate of the position of pac-man in gamespace. In the end, all these strings are formed from the kernelled version of each game instance and concatenated together. The result is a long string of bits that contains the kernel of all objects within gamespace. This string of bits forms the basis of the inputs into the ANN.

4 Artificial Neural Net

State information from the game controller is fed to an ANN using reinforcement learning, which is a type of machine learning algorithm that allows an agent to learn from an ever-changing environment through trial and error. After careful research into the various types of reinforcement learning algorithms (e.g. Q-learning, SARSA, DDPG, Deep Q-learning), the Deep Q-learning algorithm was chosen to power the decision-making process in Pac-Man.

In the context of the Pac-Man game, the environment consists of the grid/walls and the states consists of the locations of ghosts and of Pac-Man itself (encoded in a bitmap). Pac-Man faces the dilemma of maximizing its reward (game score; the feedback against which we measure success or failure of the agent) while exploring new environments. The intent of the ANN is to create the policy (a method of mapping an agents state to its actions) used in the reinforcement learning algorithm. The ANN also acts as the agent, as it drives the decision making process for Pac-Mans behavior.

4.1 Deep Q-Learning

Deep Q-learning, a variant of Q-learning, combines deep neural networks with reinforcement learning. DQN agents have been shown to surpass human performance in various reference games by Google's DeepMind; hence, DQN is considered a general-purpose agent that continually adapts its behavior in new environments without human intervention.

Q-learning, the reinforcement learning algorithm which forms the basis for Deep Q-learning, learns an action-value function $Q(s, a)$ which measures the reward of taking a particular action a given a state s. A memory table of $Q[s, a]$ is learned to store Q values for various state-action pairs. The next action a' the agent takes is chosen so that $Q(s, a)$ is maximized. Q-learning works best for small state-action spaces. Because the Pac-Man environment is constantly changing with ghost and Pac-Man movements, the state-action space is much larger. Accordingly, a Deep Q network (DQN) was used to approximate $Q(s, a)$ and to decide on the best next action based on the current game state in Pac-Man.

In Deep Q-learning, the future reward is expressed as a formula that can be optimized

on. The loss, which is minimized, indicates how far the prediction is from the target value. The loss is expressed as

$$\text{loss} = \left(r + \gamma \max_a \hat{Q}(s, a') - Q(s, a)\right)^2.$$

An action a is executed, reward r is observed, and a new state s is reached. The maximum target Q value is calculated and then discounted so that the immediate reward is valued more than the future reward.

Keras was used to implement the agent ANN and abstract away the implementation-specific details for calculating and handling loss. As the DQN agent trains, the approximation of the Q values converges to the true value, resulting in lower loss and higher game score. The remember and replay/learn features (to be discussed in Section 4.2) of a DQN distinguish it from a traditional Q-learning agent.

4.2 Training and Neural Net API

The DQN agent is trained over 1500 individual Pac-Man simulations in `main.py` and its object instance is saved to play future games. In every game iteration, the DQN agent gathers state-action-reward information and trains on a sample of the information from the respective game.

The DQN agent maintains a memory, which allows it to avoid forgetting past experiences and reduce correlations between experiences. The memory is implemented as a list of previous experiences in the form of (`currentState`, `nextState`, `reward`, `actionTaken`) tuples, which is used to re-train (fit) the model. The `remember()` method in the `NnAgent` class simply keeps track of state, action, reward, etc. in its memory.

The `learn()` method in the `NnAgent` class trains the neural net with a randomly sampled subset of experiences from its memory. Training the network in sequential order runs the risk of the agent being influenced by likely-correlated sequences of experience tuples; randomly sampling from the memory avoids this issue. In our implementation, a batch of 32 random experiences from memory are used for each training iteration.

The `learn()` method is invoked on every training iteration. The agent keeps track of both the immediate and future rewards in order to perform well over the long-term. `NnAgent`'s γ value (which is 0.95 in the implementation) serves as a discount rate. Accordingly, the agent maximizes the discounted future reward based on a state. For each memory in the randomly sampled batch, the DQN agent predicts the future discounted reward and approximately maps the current state to the discounted reward. The DQN is then trained with the current state and the future discounted reward.

Initially the agent acts randomly by an exploration rate, which is dependent on the epsilon value in the `NnAgent` class. This way, the agent leverages the power of exploration before finding patterns in the relationship between its actions, states, and rewards. The agent gradually transitions into predicting the reward based on the current state rather than random exploration and picking the action which yields the highest reward. This behavior is

implemented in `getNextMove()`, which returns an index value that corresponds to the type of movement Pac-Man should pursue, i.e. 0 for North, 1 for South, 2 for West, 3 for East, and 4 for Stop, based on the DQNs prediction. For example, given the neural net output of

$$[0.2, 0.4, 0.1, 0.1, 0.2],$$

`getNextMove()` returns 1 to the game controller, which corresponds to a southward movement.

4.3 Neural Net Design

Keras was used to implement the ANN used for the DQN agent. Categorical cross entropy was used as the loss function with softmax activation in the output layer, as this is a classification problem in that Pac-Mans best next move is chosen based on its current environment/game state. ReLU activation was chosen for the hidden layers as research shows it results in faster convergence. This is desired behavior, as it is ideal to train the agent over fewer Pac-Man games. Research also demonstrates that around 60 nodes per hidden layer is appropriate for this application based on an average of the number of input layer and output layer nodes [9] [10].

At a high-level, the ANN in the DQN agent consists of the following sequential densely connected layers:

i) (Input Layer) 127 input nodes, each corresponding to a value in numpy array of the bitmap consisting of locations of ghosts, Pac-Man, and other data relating to state/game environment;

ii) (Hidden Layer 1) 60 nodes with ReLU activation;

iii) (Hidden Layer 2) 60 nodes with ReLU activation; and

iv) (Output Layer) 5 output nodes, each corresponding to an action for Pac-Man to take (move North, South, West, East, or Stop), with softmax activation.

In additional to the traditional neural net specific hyperparameters, a couple of DQN-specific hyperparameters are specified in the NnAgent class that aid the DQN in learning from its experiences.

- `gamma`: the discount rate, used to calculate the future discounted reward;
- `epsilon`: exploration rate by which the agent acts by random action rather than prediction;
- `epsilon_end`: how much the agent explores at minimum;
- `epsilon_decay`: how much the agent reduces its exploration rate as it improves its performance over games; and
- Learning rate: the rate at which the Keras-implemented neural net learns in each iteration.

5 Results

In Figure 4, we have plotted the game score in the Pacman game obtained by our DQN agent after it was trained for 1500 game simulations with the hyperparameters shown in the table below. The results show that our DQN agent gaining higher scores after each training epoch, which signals that the neural net appears to be learning to play the game, albeit slowly. The highest score achieved by the DQN agent is 883. The fluctuating nature of the game score can be attributed to the exploration rate during the early stages of the training.

What is most peculiar is that this DQN agent almost always ends the game in a loss, possibly because of the high focus of the DQN agent on maximizing the game score (rather than focusing on winning), along with possible shortcomings the agent faces near the end of the game where the map is sparsely filled with pellets.

The DQN agent does falter in some cases, where it would play normally, and eat pellets while avoiding ghosts, but when most of the pellets on the map have been cleared, the agent would get stuck in a hallway because it was unable to discern that there were remaining pellets on the other end of the map.

6 Discussions

We see that the Pac-Man scores tend to increase with iteration. The agent appears to have occasional spikes of good games, having scores spike up to nearly 800. The high variance can be attributed to the randomness factor of the algorithm, allowing the agent to get out of the corridors where it gets stuck.

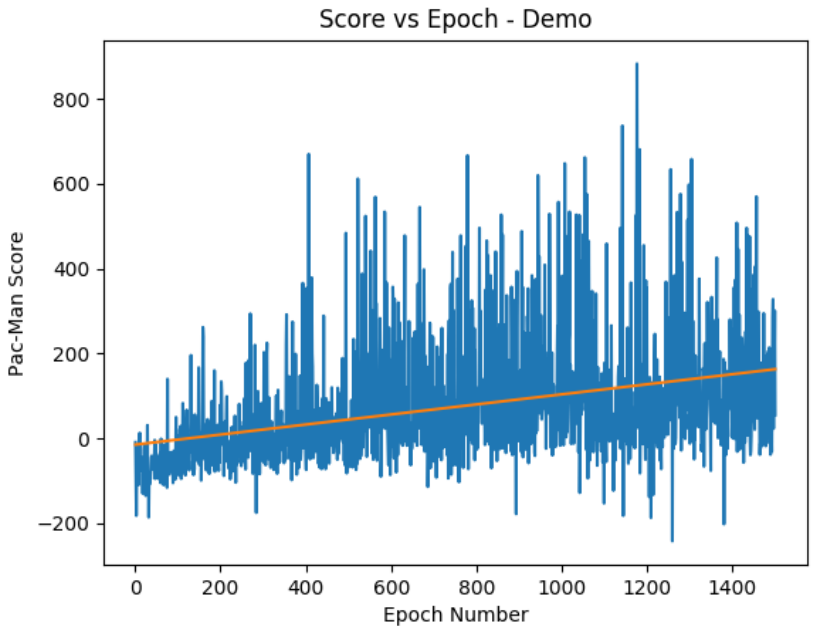
The reason our agent rarely performs well can be seen from observing the agent in real-time. We see that the agent ends up getting stuck in corridors, going back-and-forth when it runs out of pellets to eat in its general vicinity. Although we ran out of time to fix such an issue, this issue can be fixed by changing the reward function by introducing some parameters that are sensitive to time, since the current reward function depends only on the change in score.

Once this problem is alleviated, it is likely that the agent will perform at a level comparative to humans, since the agent performs like how a normal human point up to the point where it gets stuck.

7 Future Work

Going forward, there are multiple areas of our project that can be further researched and improved upon.

From the game controller perspective, it would make sense to experiment with different kernel sizes. Only a kernel of 5×5 and 7×7 were evaluated due to time constraints of the project. However, the relationship between kernel size, efficiency, and training speed would be useful to explore.



Character	Game Object
`gamma`	0.95
`epsilon`	1.0
`epsilon_end`	0.01
`epsilon_decay`	0.001
Learning rate	0.001

Figure 4: (Top) Game Scores vs. Training Epochs. (Bottom) Hyper-parameters used for this trial.

Additionally, the Pac-Man game emulator we used is not a "standard" Pac-Man game. In particular, the game board is much smaller and the ghosts behave differently. In the original Atari Pac-Man game, the ghosts have algorithmically encoded personalities which actively pursue Pac-Man, as opposed to the Stanford emulator that we used where the ghosts move randomly.

Regarding the machine learning aspect of the project, there are also numerous modifications and improvements that can be made. We investigated the optimal choice of hyperparameters for our Keras Neural Network instance (learning rate, number of hidden layers, number of nodes, loss function, etc.). However, whether or not our parameters were the global optimal remains unknown, as there was insufficient time and computational power to perform the extensive grid search required for extremely precise parameters. Allocating more resources to search for an optimal parameter configuration could increase neural network training efficiency and performance.

Another feature that was not implemented due to time constraints was parallelization of the main training script. It was extensively discussed how parallel instances of the Pac-Man game could be run to increase training speed. This would likely involve running multiple games in parallel, combining the memory of game steps and actions from all games, randomly sampling steps from the aggregate memory, and finally train on those sampled steps. Implementation ultimately proved too time consuming and complex given our limited amount of time. However, if implemented this could greatly improve the training speed.

A final major future improvement would be to refine the feature vector that was outputted by the game controller and fed to the neural network. While the agent was able to achieve a score in the 800s range before dying, it also frequently scored very poorly. This juxtaposing performance is likely due to non-optimal feature choices, as the current features are likely unable to represent all of the complexities of the game state to consistently play a successful game of Pac-Man.

8 Conclusion

We implemented a Deep Q-Learning approach to train an AI agent to play the classic Atari game, Pac-Man. This was achieved through the construction of three distinct modules: a game controller, a neural net Agent, and a training script. Through extensive iterative training, the neural network powered agent was able to successfully play a game of Pac-Man, achieving a decently competitive final score. As discussed previously, there are numerous improvements and modifications that can be made to enhance the project, including additional feature selection, hyperparameter tuning, parallelization, and overall code optimization. While an AI Pac-Man-playing agent is not a new idea, this unique implementation and modular design allow for future extensibility allow for future extensibility in a potentially limitless number of applications.

References

[1] Simonini, T. (2018, April 11). *An introduction to Deep Q-Learning: lets play Doom.* Retrieved from https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8.

[2] Seno, T. (2017, October 20). *Welcome to Deep Reinforcement Learning Part 1 : DQN.* Retrieved from https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b.

[3] Kim, K. (2016). *Deep Q-Learning with Keras and Gym.* Retrieved from https://keon.io/deep-q-learning.

[4] Bhatt, S. (n.d.). *5 Things You Need to Know about Reinforcement Learning.* Retrieved from https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). *Human-level control through deep reinforcement learning.* Nature, 518(7540), 529.

[6] Galway, L., Charles, D., & Black, M. (2008). *Machine learning in digital games: a survey.* Artificial Intelligence Review, 29(2), 123-161.

[7] Slater, N. (2017, July 19). *What is experience replay and what are its benefits?.* Retrieved from https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b.

[8] V. Mnih, K. Kavukcuoglul, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, https://arxiv.org/abs/1312.5602.

[9] Walia, A. S. (2017, May 29). *Activation functions and its types-Which is better?* Retrieved from https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f.

[10] doug (2010, August 2). *How to choose the number of hidden layers and nodes in a feedforward neural network?.* Retrieved from https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw.

[11] C. L. Martinez-Nieves, *Defeating the Invaders with Deep Reinforcement Learning*, http://cs229.stanford.edu/proj2017/final-reports/5244209.pdf.