AI in Games - DRL

Álvaro Miranda
Faculty of Engineering
University of Porto
Porto, Portugal
Email: up201603694@g.uporto.pt

Miguel Carvalho
Faculty of Engineering
University of Porto
Porto, Portugal
Email: up201605757@g.uporto.pt

***Abstract*—Nowadays, there exists a part of artificial intelligence and machine learning with very good application in games, which is Deep Reinforcement Learning, where a machine can train itself based on their actions' results. Usually, to perform this kind of learning, it is necessary to own/replicate the game environment to perform multiple learning episodes in a small amount of time. Our work's objective is to develop agents to play HaxBall, a 2D football online browser game, without the possession of the game's source code. Thus, we decided to use Supervised Learning techniques, collecting data from real players, and training Neural Networks to control the agents. We will explain our method and how we managed to overcome the challenges of the work.**

Keywords: Artificial Intelligence Deep Reinforcement Learning Supervised Learning.

1. Introduction

In this work firstly we will talk about what was referred in the abstract, that is deep reinforcement learning. After that we will talk about a supervised learning.

We will talk about our methodology work on the bot for Haxball, including the data gathering, the setup for the experiments, and the different approaches to the Supervised Deep Learning techniques

2. Deep Reinforcement Learning

Reinforcement learning want to achieve a process similar to the humans. Like to a human do a complex math problem, he first needs to learn how to solve it. If during that process he fails he learn from that until he can finally solve the problem the right way. The machine will do the same using some well-designed computing algorithms. Using things like inputs and outputs the machine will train itself to achieve his goal by seeing if the results of his actions bring good or bad rewards. It do that until it achieves its final goal which depends on what it is. It can be playing a game or solving a difficult math problem like referred before.

As we can see it's a part of machine learning. Later we will talk about the Supervised part.

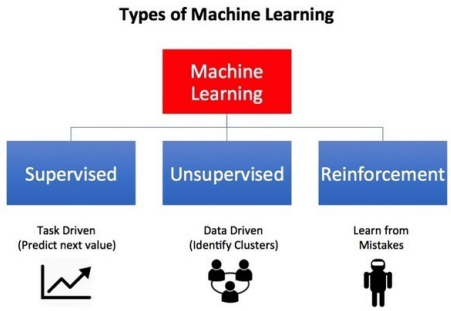

Figure 1: types of machine learning

2.1. Important aspects in DRL

Here we will talk about some important definitions to understand better the process used in DRL.

- **Agent**: easy to understand that it is the system which will interact with the surrounding environment;
- **Action**: the action that the agent will choose to execute next.
- **Environment**: it's where the agent is inserted and will do it's actions. It takes the agent state and it's action and returns it's reward.
- **State** situation the agent encounters itself in the the present moment
- **Reward**: the reward is how the agent will evaluate if an action is good or bad in a given state.
- **Discount factor**: it essentially determines how much the agent cares about rewards in the distant future relative to those in the immediate future. if its value is 0 it only looks for a immediate reward. Otherwise if it is 1 it will look for all the rewards of its future actions.
- **Policy**: it defines the learning agent's way of behaving at a given time based on its state.
- **Value**: it is the long term reward. According to the current state and policy we define the value.
- **Q-value or action-value**: the same as value but it takes another parameter which is the current action.

2.2. Mathematics and Algorithms Used

Here we will talk about some methods and algorithms DRL uses.

- **Markov Decision Process**: As we see in figure 2, it summarizes the reinforcement learning. It says that the futures states only depend on the actual state and not the previous ones.

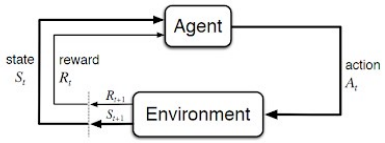

Figure 2: agent-environment interaction in a Markov decision process

- **Bellman Equations**: it's a set of equations that decompose the value in the the immediate reward plus the discounted future values.
- **Dynamic Programming**: when the agent and the environment are fully known, using the Bellman equations, Dynamic Programming can be used to evaluate value functions in interactions and update policy.
- **Value iteration**: this algorithm wants to compute the optimal state value function. To do that it iteratively improves the estimate of the value. It begins with random in the value function and then it updates the Q-Value and value function until they converge.
- **Policy iteration**: this algorithm only wants to find the optimal policy, so to achieve that it re-defines the policy each step and compute the value according to the new policy until the policy converges.
- **Q-learning**: first, this model assumes that the agent doesn't know anything about state-transition and reward models. So, the agent will have to learn what is good and what is bad by trial and error. It works by storing in a set of Q-Tables with the state-action pairs. It starts with the value 0. After each interaction it stores the values there and as it is being filled it will hold the best actions for the agent. Normally the actions are chosen randomly.
- **Deep-Q-learning**: A problem that Q-Learning has is that the Q-Table can be enormously. Deep Q-learning, instead of maintaining a large Q-value table, utilizes a neural network to approximate the Q-value function from the given input of action and state.

2.3. Uses in Games

An well-known example of DLR being used in games is when DeepMind developed an IA capable of beating the best player in the world at the game Go, which is considered the most challenging classical game for AI, as

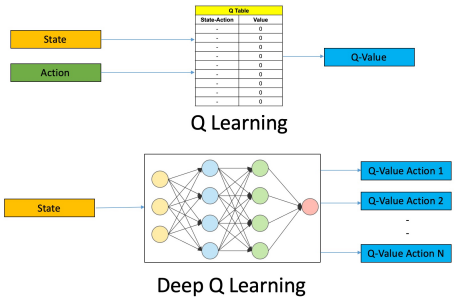

Figure 3: difference between Q-Learning and Deep Q-Learning

it has too many move options. The AI was called AlphaGo. A better version of AlphaGo is also being developed called AlphaGo Zero, where it starts with a neural network with zero knowledge of the game and in each interaction it increases it's performance. Using powerfull search engines, each interaction it improves a litle more.

This shows the power of Deep Reinforcement Learning.

3. Supervised Learning

Supervised Learning is the most common area of machine learning and the easiest to learn.

It is an approach to creating artificial intelligence by being given input data and the expected output results. The AI then knows what to look for. With this, the model is trained to detect the underlying patterns and relationships in the training phase. After this, the trained model is presented with test data to verify the accuracy of the training. As it is being tested, it can be adapted to fine-tuning the system to get better results. This accuracy depends on the algorithm used and the data used too.

But this adaptation to the results can be dangerous because it can cause a phenomenon called overfitting, where the model is overtuned to the data set and fails in new data in the test data. This can be fought by diverse training data, and guarantying that the test data is different from the training data to check if it is not drawing answers from its previous experience, but instead, the model's inference is generalized.

3.1. Classification and Regression

There exists two kinds of results in supervised learning algorithms: the classification and regression.

The classification algorithm first in the training is given data points and an assigned class. Then after being presented with the test data it has the function to label it with the correct class accordingly with the input data, based on the previous training data. An example of this is checking if an email is spam or not. The algorithm is initially trained with a set of emails labelled as spam or not spam. Then it will check new emails and will label them as being spam or not taking in account its precious training.

However many times it is not a binary classification and a certain data can have multiple classes. The algorithm in addition to giving the predicted class also gives and confidence in the prediction.

In regression, it deals with numerical values. First it is trained to try to create relations between the data and create ways to predict future relations in the test data. It has many utilities like for example calculate the odes in sports bets. It checks the teams previous results, its standings and taking that information in account they produce a numerical value so that the gamblers feel inclined to bet, and the company behind the bets being able to gain money independently of the match result.

3.2. Supervised Algorithms

There exists many algorithms that can be used in this process. It depends on what the person using them wants and most importantly what the data used requires to be handled the best way.

- **Linear Regression**: the most simple, it tries to correlate two factors, one independent and the other dependent. It is used one table and the algorithm tries to find a linear function that most accurately predicts the relation between the two factors.
- **Polynomial Regression**: like the linear regression it is a form of regression analysis in which the relationship between the independent variable x and the dependent variable y is modelled as an nth degree polynomial in x, instead of a linear function
- **Logistic Regression**: like in the linear regression i tries to model a binary dependent variable. It uses a logistic function to model a binary dependent variable, it is usually used to produce a binary regression that shows that if the value is on one side of the line it is 1, if it is on the other it is 0.
- **Decision Trees**: it uses a tree-like model of decisions to classify the data. Are most commonly used in decision analysis to help identify the best strategy to reach the goal.
- **Support Vector Machines**: first it is given training examples, each marked as belonging to one of two categories. Then it builds a hyper plane or set of hyper planes in a high- or infinite-dimensional space, which can be used for classification or regression or outliers detection. This plane separates the two classes and, the largest distance between the two classes(also called margin) the better the future classification will be in the future.
- **K-Nearest Neighbor**: this algorithm used for classification and regression. It consists on classifying the new data according to it's k neighbors. It sees its k nearest neighbors of the new data in the space, and attributes it accordingly to those k neighbors.
- **Random Forest**: consists on constructing a multitude of decisions trees when being trained, and outputting and class in case of classification or an average prediction in regression. It corrects the decision trees problem of overfitting so it outperforms them.
- **Artificial Neural Networks**: It has a hundred of thousands of artificial neurons called processing units connected by nodes. These are made accordingly to inputs and outputs.It goes though the training phase, where it learns to recognize patterns in data. It then checks if the actual output is equal to the desired output, by using backpropagation. This is done to adjust the weight of the connections until the difference between the two outputs is minimal. Then new data is classified accordingly with this neural network.

3.3. Uses in Games

An well-known example of DLR being used in games is when DeepMind developed an IA capable of beating the best player in the world at the game Go, which is considered the most challenging classical game for AI, as it has too many move options. The AI was called AlphaGo. A better version of AlphaGo is also being developed called AlphaGo Zero, where it starts with a neural network with zero knowledge of the game and in each interaction it increases it's performance. Using powerfull search engines, each interaction it improves a litle more.

This shows the power of Deep Reinforcement Learning.

4. Haxball Bot

Subsection text here

4.1. Haxball

The game consists of a 2D soccer game with a standard field, each team with a goal. It is played by two teams which can have a variable number of players. The objective is to score more goals than the other team. When the game time finishes, the team with more goals is the winner. All the entities are discs: circles with various radius. The player is controlled with five keys: 4 to move for each direction, and one more to shoot the ball when the ball and the player discs are in contact. When the ball hits the limit lines, it bounces back to the playing field. As this is a multi-player, world-wide game, in the main page there is a lobby with rooms from all over the world. The player has the option to create a room, where he have control over the players, being able to kick them or to join a previously created room to play.

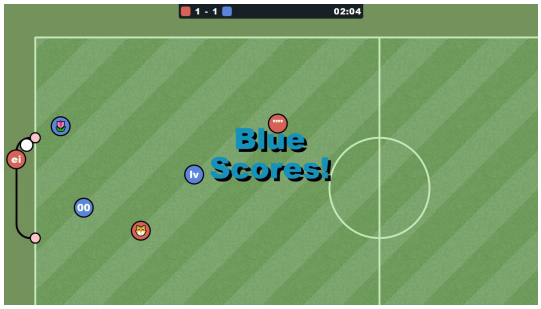

Figure 4: example of a haxball game

4.2. Environment

As we previously mentioned, we do not have access to the HaxBall source code, which lead us to two options: to simulate the environment, which would give us the option to use Deep Reinforcement Learning techniques and more control over the game and experiments, or to find a way to retrieve information about the state of the game in real-time to gather data in order to use Supervised Deep Learning to train the agents. We chose to follow the second path. For this, we needed to find a way to retrieve the game state at each moment (used as input data for the Neural Networks) and the keys pressed by the player (label input for the Networks).

Each HaxBall map (football field) can have different sizes and ball and player physics, so we created a map with a slow and big ball to make it easier for the agents to play.

4.3. HaxBall Headless Host

The HaxBall Headless Host gives us the opportunity to host a manageable HaxBall room locally on the browser through an API that provides control of many aspects of the room. This rooms are commonly used by the community to make autonomous rooms that don't need an admin to organize the players in teams and may even persist ranks, banned players or other useful systems. With this, we have access to the state of the game, that is, the position, the velocity and direction of the players and the ball, and other general information about the room. It has some limitations as we have no way to persist data in our computer directly from the code of the room without the use of another server running locally. We use the host to gather data of games, as we will explain in the dataset section.

4.4. Dataset

To solve a Supervised Deep Learning problem, we need to give some input and correspondent labels to train the Neural Network. Our approach was to extract the game state at each moment, which is the positions and velocity vectors of each disc (players and ball) in the game and using the player input as labels: moving left, up, right, down, and kicking as a binary vector of size 5 ([0,0,0,0,0]). This section will explain how we gathered the data, how we process and analyze it.

4.4.1. Data gathering

Firstly, we tried to host a room that sends the game state at a specific rate to a local python server through a WebSocket, which would collect each data frame and, simultaneously, read the player's keyboard using python's win32api package to detect which keys are pressed at each game tick. However, we find it hard to synchronize the game state and the input keys, making it impossible to create the dataset with this method. Secondly, we used computer vision techniques such as circle detection and template matching to detect, in the previously grabbed display screen displaying the game, each of the circles representing the players and the ball. At the same time, we analyze the keys pressed by the player.

In the end, we had a setup to record ourselves playing and gather the necessary data to train the Neural Network. However, one big problem of the Deep Learning problems is the lack of data, which leads to non-generalized classifiers with lousy performance. Thus, while we continued our experiments with the data recorded of our gameplay, we opened an automated public 1v1 room, which manages its game sessions and continually sends information through a WebSocket to a local server where we persist the data of these games, including which player was the winner in order to filter the data. With this approach, we continuously gathered a lot of data, even though we can't control what kind of style the players will follow and if it will be helpful or harmful to the agents.

We also created data where the red player is alone with the ball to start with an easier problem and understand what was the better architecture for the networks and features as input. We persisted sets of dribble, where the agent mission is to dribble the ball across the field, and a scene where the ball and the player are put randomly in the player's side of the map, and the agents needs to go to the ball and shot it to the blue goal. For the last one, we use the Headless Room for the recording and the experiments to test the agents.

4.4.2. Data processing

From the game screen, all we could detect from the game was each disc's position on each frame. However, we needed to process this data to achieve a higher amount of features. Thus, we calculated for each disc the velocity vector (vx and vy) and respective magnitude (disc speed), the acceleration vector (ax and ay), the angle in which direction the disc is moving, and the angle and distance between the red disc (player) and the ball, the blue disc and the enemy goal. We used these features in different situations, and they were

grouped differently in our experiments to test which set was more adapted for the Neural Networks training. Some of the values we normalized using zero mean unit standard deviation column wise, using the following formula:

$$z = \frac{x_i - \mu}{\sigma}$$

Figure 5: where xi is the feature value, μ is the mean and σ is the std

Also, the angles were encoded in the form of cos(a), sin(a) making it 2 inputs for each angle.

For the data acquired with the Headless Host, we had to take more steps in order to process the data because the game scale was different from our screen, and the rate of state recording was also different. To solve this problem, we converted each recorded game to the same scale of our data. Our biggest challenge was the fact that we have no access to the keyboards of the players recorded, so we had to find a way to overcome this problem. Analyzing each frame, the positions velocities, calculating each player's acceleration, and detecting the strength of the field's friction, we were able to predict the inputs of the players: the movement and the kicks.

Furthermore, we created a tool to cut the sequences of the gathered data sequences to prevent feeding the Neural Networks with bad performances, since many times, the players in the room weren't playing with the intention of winning.

4.5. Experiments

For the Deep Learning, we used python's package Skorch (PyTorch), "A scikit-learn compatible neural network library that wraps PyTorch." and we did multiple experiments of different play modes, Neural Network architectures, features to find the best way to teach the agents to play HaxBall. In this section, we will show the experiments, the results, and some conclusions about our work. In the following experiments, we used validation accuracy and loss as a metric, even though, in this kind of problem, it is better not to really a lot of this evaluations, since this is a continuous game with numerous possibilities in each situation. It is a good idea to test the agents in games.

We started our project by implementing a behavior-based hardcoded bot, which analyses the game state at each moment and chooses the best behavior accordingly. The first objective of coding this agent was to test our detection program and our environment. After, we could train the Deep Learning agents with the gameplay of this behavioral agent to have a fixed play style. We could teach the Deep Learning agents against this bot to play against a blue player with a fixed play style or even could test our agents against this bot since a human can easily win the Deep Learning agents.

	Features used	**Architecture**	Result
NN1	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 100 Output: 5 Optim: SGD	Val_acc: 0.553 Val_loss: 0.232
NN2	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 32 Hidden: 64 Output: 5 Optim: SGD	Val_acc: 0.534 Val_loss: 0.23 Aparently dribbles in circles
NN3	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 8 Output: 5 Optim: Adam	Val_acc: 0.526 Val_loss: 0.267
NN4ang	6 inputs: angle.red.ball.cos, angle.red.ball.sin, distance.red.ball angle.red.goal.cos, angle.red.goal.sin, distance.red.goal	Input: 6 Hidden: 8 Output: 5 Optim: Adam	Val_acc: 0.508 Val_loss: 0.261 Aparently dribbles very good

TABLE 1: Dribbling experiments

A. The first problem to be solved by our agent was the dribble. We recorded us dribbling the ball inside the field without using the kicks. We used Binary Cross Entropy in all the experiments, and we gave 15% of the data for a validation set. A total of 3562 rows from which 3028 were used for training and 534 for the validation.

Even though the dataset is small and some of the agents move in rounds around the ball, it was easy to find a agent with good results, even though the accuracy is not very big.

B. Since the dribbling problem was too simple, it was created a second problem consisting of moving to the ball and shooting to the blue goal. 4848 samples were collected, from which 4121 for the training set and 727 for the validation. For each shoot, the ball and the red disc were set in a random position on the red side of the field, with a random velocity in a random direction. The red player was recorded moving to the ball and shooting to the blue goal. The results can be seen in Table 2. It was notable that it was hard for the agents to learn when to kick since the classes are highly unbalanced because the kick is much rarer. To solve this problem, when testing some of the Neural Networks, we reduce the classification threshold of the kick output to lower than 0.5, which made it kick more times but scoring goals.

To test the performance of the agents in the shooting, we set up our Headless Room to create shooting sets for the evaluation. In each set, the room set a counter, and it was considered timeout at 13 seconds. After this time, the agent had lost the opportunity to do a goal. The own goals were also taken into account. Each agent played 50 times unless it didn't score on the 20th attempt. The results are in Table 3.

	Features used	Architecture	Result
NN1	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 16 Hidden: 16 Output: 5 Optim: Adam Kick threshold: 0.2	Val_acc: 0.623 Val_loss: 0.212 Didn't shoot without reducing the kick threshold
NN3	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 16 Output: 5 Optim: Adam Kick th: 0.2	Val_acc: 0.612 Val_loss: 0.265 Didn't kick without the threshold
NN4	8 inputs: red.x, red.y, red.vx, red.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 8 Hidden: 16 Output: 5 Optim: Adam	Val_acc: 0.637 Val_loss: 0.249 Good performance Good goals
NN5ang	5 inputs: angle.red.ball.cos angle.red.ball.sin distance.red.ball angle.red.goal.cos angle.red.goal.sin	Input: 5 Hidden: 16 Output: 5 Optim: Adam Kick th: 0.3	Val_acc: 0.602 Val_loss: 0.287 Good dribble Didn't shoot without threshold
NN6ang	12 inputs, all normalized: angles and distances between each other disc and blue goal and red, speed of disc, direction of movement of each disc	Input: 12 Hidden: 16 Output: 5 Optim: Adam	Val_acc: 0.693 Val_loss: 0.218 Good performance
NN7ang	12 inputs, all normalized: angles and distances between each other disc and blue goal and red, speed of disc, direction of movement of each disc	Input: 12 Hidden: 16 Output: 5 Optim: Adam	Val_acc: 0.712 Val_loss: 0.181 Very good performance
NN8ang	12 inputs, all normalized: angles and distances between each other disc and blue goal and red, speed of disc, direction of movement of each disc	Input: 12 Hidden: 13 Output: 5 Optim: Adam	Val_acc: 0.731 Val_loss: 0.192 Aparently best performance

TABLE 2: Shooting experiments

	Results		Result
NN1	Games played: 20 Goals scored: 1 Owl goals: 1 Timeout: 18	**NN6ang**	Games played: 50 Goals scored: 6 Owl goals: 5 Timeout: 39
NN3	Games played: 50 Goals scored: 4 Owl goals: 1 Timeout: 45	**NN7ang**	Games played: 50 Goals scored: 11 Owl goals: 2 Timeout: 37
NN4	Games played: 50 Goals scored: 10 Owl goals: 1 Timeout: 39	**NN8ang**	Games played: 50 Goals scored: 9 Owl goals: 1 Timeout: 40
NN5ang	Games played: 25 Goals scored: 0 Owl goals: 0 Timeout: 25		

TABLE 3: Shooting performance testing

C. Finally, the agents learned how to play a 1v1 game against another player. We recorded 15693 rows of gameplay against the same opponent for this phase, from which 13339 were used for training and 2353 for validation. As this is a much more complex problem, the results weren't so successful. Even though, it is notable that the agent learned and even scores goals. Evidently, the agent adapts to a specific playstyle against a particular opponent; thus, it is very hard to generalize the agent to play against any opponent and even correctly perform different defense/attack strategies.

	Features used	Architecture	Result
Our data			
NN1	12 inputs: red.x, red.y, red.vx, red.vy, blue.x, blue.y, blue.vx, blue.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 12 Hidden: 16 Output: 5 Optim: Adam	Val_acc: 0.454 Val_loss: 0.382
NN2	18 inputs, all normalized	Input: 18 Hidden: 32 Output: 5 Optim: Adam	Val_acc: 0.466 Val_loss: 0.324
Room data			
NN3	12 inputs: red.x, red.y, red.vx, red.vy, blue.x, blue.y, blue.vx, blue.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 12 Hidden: 16 Output: 5 Optim: Adam	Val_acc: 0.420 Val_loss: 0.388 In practice has very bad performance
NN2	12 inputs: red.x, red.y, red.vx, red.vy, blue.x, blue.y, blue.vx, blue.vy, ball.x, ball.y, ball.vx, ball.vy	Input: 12 Hidden: 24 Output: 5 Optim: Adam	Val_acc: 0.454 Val_loss: 0.382 In practice has very bad performance

Moreover, 113981 data samples were collected from the Headless Host room, representing different players, and processed in order to predict the key input at each frame. Some models were trained with this data without any filter, and the results were, as excepted, not very good since any style of play can be in that data. We developed a visualizer to inspect each frame and to cut the poor data, but this process would require a lot of time. Since we also gathered each game-winner, it is possible to train the Network only with the players that won the game. The results of both experiments are presented in Table 4.

5. Conclusion

The conclusion goes here.

References

[1] Comi, M. (2020, March 22). How to teach an AI to play Games: Deep Reinforcement Learning. Retrieved January 20, 2021, from https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a

[2] Contributors, T. (2020, July 08). What is Supervised Learning? Retrieved January 20, 2021, from https://searchenterpriseai.techtarget.com/definition/supervised-learning

[3] Frankenfield, J. (2020, August 28). Artificial Neural Network (ANN). Retrieved January 20, 2021, from https://www.investopedia.com/terms/a/artificial-neural-networks-ann.asp: :text=An%20artificial%20neural%20network%20(ANN)%20is%20the%20piece%20of%20a,by%20human%20or%20statistical%20-standards.

[4] Montantes, J. (2020, May 12). What You Need to Know About Deep Reinforcement Learning. Retrieved January 20, 2021, from https://towardsdatascience.com/what-you-need-to-know-about-deep-reinforcement-learning-3571ec8368f4

[5] Nielsen, A. (2020, July 22). Deep Reinforcement Learning for Video Games Made Easy. Retrieved January 20, 2021, from https://towardsdatascience.com/deep-reinforcement-learning-for-video-games-made-easy-6f7d06b75a65

[6] What is deep reinforcement learning? (n.d.). Retrieved January 20, 2021, from https://bernardmarr.com/default.asp?contentID=1902: :text=Deep%20reinforcement%20learning%20is%20a,penalised%20based%20on%20their%20actions.

[7] Wilson, A. (2019, October 01). A Brief Introduction to Supervised Learning. Retrieved January 20, 2021, from https://towardsdatascience.com/a-brief-introduction-to-supervised-learning-54a3e3932590

[8] Haxball. (n.d.). Haxball/haxball-issues. Retrieved January 20, 2021, from https://github.com/haxball/haxball-issues/wiki/Headless-Host