Real-World Reinforcement Learning through an Attempt to Learn The Web-Game "Slither.io"

Abstract

We present an attempt to apply deep reinforcement learning methods to train a model to learn the web-game "Slither.io", while exploring approaches and techniques applicable to other real-world problems.

Introduction:

Reinforcement learning (RL) is a branch of machine learning concerned with creating an agent that achieves desired behavior in a defined environment. The agent achieves this goal by interacting with the environment, choosing an appropriate action for a given state in order to maximize some well-defined cumulative reward.

Reinforcement learning application can fit myriad tasks ranging from human communication using natural language (i.e. chat bots), autonomic cars, training a bot to walk and playing games.

In some cases, RL methods can be used *in silico* to train a model faster by emulating the agent-environment interaction. However, in the real-world such methods might be impossible to implement – for example when one wishes to train a model within a real-world environment or when one attempts to train a model by observing an internet based environment– a case in which one does not have full control regarding the emulation speed, furthermore the reliability of the data is questionable.

In this mini-article we will attempt to overcome these two issues by training a model, using RL methods, to play Slither.io [2] – a multiplayer snake game in which the player tries to reach the highest score possible by collecting "food" and avoiding getting tackled by other players. We will present the different approaches we Implemented, show their results and discuss our conclusions.

Our motivation approaching this project was to explore deep RL techniques in a real-world environment and examine effective ways to implement them, speed up the training process and successfully train a model to learn how to optimally perform within the Slither.io environment. Those techniques can be found effective solving other real-world problems as well.

Background: see appendix [a]

Related Work:

One distinguished previous work in the field is heuristic based model created by ErmiyaEskandary et al [3]. This JavaScript based bot performs well and able to achieve high score (the highest score we measured was 18,000~), aiming for large food clusters within its view while prioritizing avoiding enemies over anything else. We have decided to use this bot's infrastructure as a base to our model.

An often recited RL based work was done by DeepMind Technologies in their Atari article [1], In which they demonstrated a new RL deep learning method called **Deep-Q-learning**. This method will be presented later in this article.

Our approach:

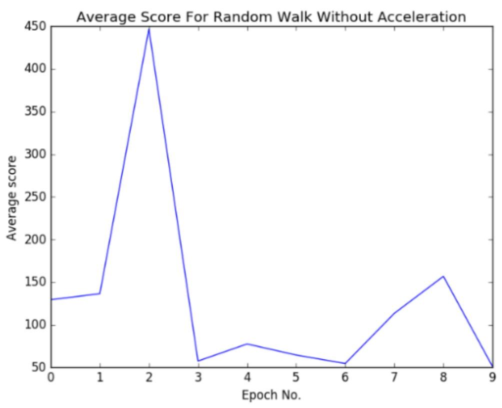
To establish communication between our model and Slither.io server, we used a client-server practice over HTTP protocol [c]. The client was implemented in JavaScript while the server was written in Python.

The Input consisted of 4 stacked frames (similar to [1] page 5). Each frame is a 24X24 matrix of labeled points, each label was determined given the closest objects to its point. Possible labels: {(0, **enemy**), (25, void), (50, **our bot**), (100, **sparse food cluster**), (150, **medium food cluster**), (200, **intense food cluster**)}, thus creating a grayscale image representing a game frame (see figure 2).

Our Architecture was based upon convolutional neural networks (**CNN**) consisting of 2 sets of a convolutional layer followed by a pooling layer, after them a flattening layer followed by 2-3 fully connected layers.

The Output vector, representing the policy or Q-values of the possible actions given a certain state (depending on the implemented model), consists of 64 entries. Describing 32 possible angles the bot could turn to while accelerating or not. (see figure 7)

To obtain a baseline, we ran the bot twice for 100,000 steps, performing random actions, enabling and disabling acceleration for each experiment. A model will be considered a learning model if it consistently outperforms the baseline.



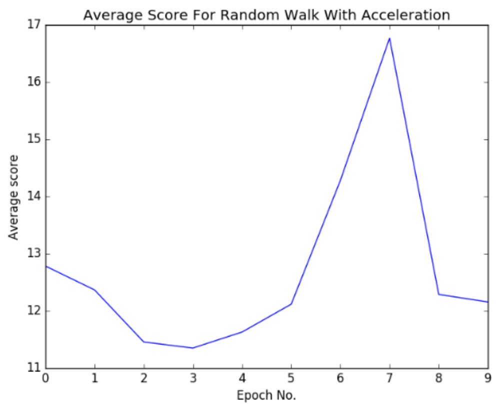


Figure 1 – random walk performance graphs with and without acceleration. Each epoch consists of 10,000 observations.

Attempts and Experiments:

In all the experiments in this paper, 10,000 steps are one epoch which runs for 1~ hour, in each step one frame is observed. In the DQN experiments the models were trained with an ε-greedy behavior, with ε decaying from 1 to 0.05 or 0.001 along the game.

Initially we tried to implement a neural network based policy gradient. We abandoned this model as we have seen that theoretically it is inefficient given the problem ([1] page 5) and as it did not yield significant results.

Later, we implemented a **deep-Q-network (DQN)** with experience replay memory [b], as presented in the Atari article ([1] Page 5) Running this model without any modifications did not yield significant results.

Generating additional observations - To expand the set of observed transitions **(s_t, a_t, r_t, s_{t+1})**, we applied rotational, symmetrical and transpose transformations resulting in equivalent transitions. This insight has allowed us to augment a given transition to an additional 7 observations (see figure 1). Effectively allowing us to accelerate our learning rate, overcoming the limited sampling rate which is inherent to the real-world environment.

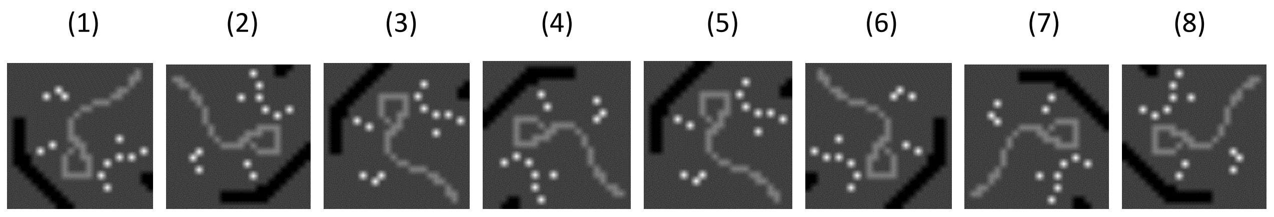


Figure 2 – image (1) is the original frame observed during the game. Images 2-8 are rotated versions of image (1) that were added to the memory reply. The action chosen got rotated accordingly. The images are enlarged 25 times their original size.

Improved reward function – in order to convey more data to our model we defined the following reward function:

$$R = \begin{cases} -k, & if\ died \\ R_{food} + P_{enemies} + P_{dcenter} + R_{length} + R_{gain}, & else \end{cases}$$

Where $R_{food}, P_{enemies}, P_{dcenter}, R_{length}$ and R_{gain} are reward or penalty functions which allows traits of the environment other than the raw score to affect the reward, and thus affect the learning process. The idea behind this is to accelerate the training process by combining it with basics heuristics (like reaching for food), using reward which better describes the implications of an action **a** in some state **s**. See also appendix [d].
In addition, past work has successfully demonstrated the use of heuristics to accelerate Q-learning [7], which encouraged us to try it in our work as well.

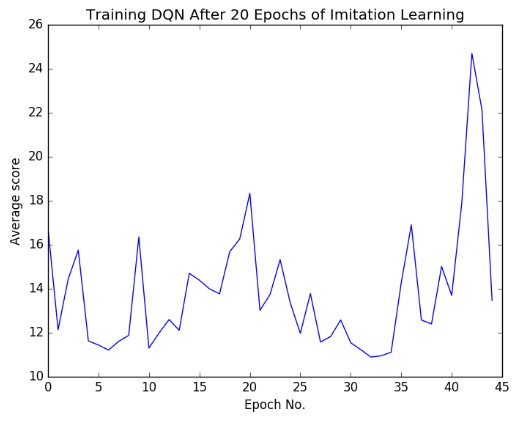
With the described approach we tried to train our model with and without acceleration for 20-30 epochs. From the results we couldn't conclude if we saw the beginning of a learning progress, or a sequence with some successful games. As the results are not distance from the baseline, we don't consider them as a successful learning. (see figure 5)

In addition to the deep Q learning we implemented an **Imitation learning** method to try to learn from the heuristic bot's behavior. In imitation learning we provide the agent with demonstrations generated by a supposed expert, from which it should derive desired behavior.

Previous work suggested techniques for combining imitation learning and Q-learning [8], so we attempted to combine it with deep Q-learning as well.
We implemented this method by observing the heuristic bot play the game, tracking its states, actions and rewards and feeding them to the experience replay memory. The rest of the DQN algorithm hasn't been modified.

We trained the model for 10-20 hours using that approach and evaluated the results. This experiment also gave us an idea about the heuristic bot performance (see figure 6). The results were exceptional, yet we couldn't conclude from them that the model had learned (see appendix [e])

In a follow-up experiment we loaded the weights resulted from the previous one to the model and trained it for 45 epochs, a total of 65 epochs of training. Unfortunately, this experiment didn't yield any significant results beyond the initial baseline. The results can be seen in the following figures:



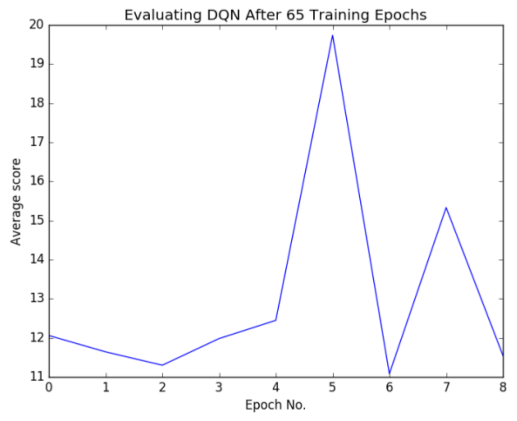


Figure 3 – The left plot describes the training process after 20 epochs of imitation learning. The right graph is the evaluation of the resulted weights.

Lastly, we implemented DQN with optimality constraints but were unable to test it under our computational constraints (see appendix [g]).

What could have gone wrong:

Despite our efforts we did not manage to train the model to perform better than the random walk baseline. Assuming no flaws in our implementation, there can be several reasons for that:

- We didn't give the model enough time to learn. Training a model to preform reasonably in an Atari game with a DQN algorithm takes hundreds of millions of observations ([4] page 1), while our experiment reached around 650,000 steps at most (which we augmented to 5.2 million observations).
- Slither.io game states are very noisy and random – i.e. there is no guarantee that preforming action **a** in a state **s** will lead to another state **s'** or even to small set of such states, as the rival snakes are controlled by human players and can appear suddenly and the food appearances also does not have an obvious pattern. This implies that the amount of observation needed to train our DQN model may be even larger than an Atari game like Pong.
- The bot might choose to accelerate while decreasing its points. When at minimal length, choosing to accelerate will not affect the neither bot nor the score. This might make it difficult for the model to learn the basic correlation between accelerating and point loss, as it experiences 'inconsistencies' in the environment's behavior.
- Communication issues between the client and the server might have reduced the performance of the model. It was most noticeable on the death count – by comparing the death count on the client and the server side, we noticed the model missed around 0.25-0.33 of the deaths.
- It is possible that the parameters of the model weren't tuned correctly, the architecture was not suitable for the problem at hand or the models weren't suitable for the problem.

What we could have done better?

- Let the model run for a longer time and parameter tuning.
- Test other net architectures, specifically RNN architecture. RNNs can overcome the delay between the chosen action and reward by using feedback, thus 'remembering' previously chosen actions [9].
- Implement different approaches that might be more successful – like DQN with constraint optimization [4] or A3C [5]. We didn't implement the latter model for the lack of resources.
- Producing the input in a fashion that can overcome the need for a client-server communication and the data loss which it entails – like using machine vision or implementing the whole algorithm on the client side [6]. We did try the latter approach, but unfortunately as far as we checked the library we used couldn't handle this task. Initially we have attempted to implement computer-vision techniques, but as we have seen that our RL model didn't yield significant results we abandoned this aspect of the project.
- As last resort, we would reconsider the assumption that it is impractical to emulate the environment. Then we would try to train the model first in a game with other bots, doing that would allow us to run the game in un-human speed.

In conclusion:

In this mini-article we tried to use deep RL methods to train a model to play Slither.io. Although we tried several approaches, that seem theoretically correct, we did not achieve significant results that can provide a reasonable proof of concept. We also used several techniques in order to speed up the training process by dealing with the inherent problems of real-world tasks. These techniques can be useful when dealing with other problems with similar characteristics.

References:

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller. *Playing Atari with Deep Reinforcement Learning.*

[2] Slither.io game: http://slither.io/

[3] ErmiyaEskandary heuristic bot

[4] F. S. He, Y. Liu, A. G. Schwing, J. Peng. *Learning To Play In A Day: Faster Deep Reinforcement Learning By Optimality Tightening,* 2017

[5] A3C Tutorial by Arthur Juliani

[6] ConvNetJS – a deep learning JavaScript library

[7] R. A. C. Bianchi, C. H. C. Ribeiro, and A. H. R. Costa. *Heuristically Accelerated Q–Learning: a new approach to speed up Reinforcement Learning.*

[8] T. Brys, A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, A. Nowè. *Reinforcement Learning from Demonstration through Shaping.*

[9] Introduction to RNN's tutorial, *What are RNN's* section.

Appendices:

[a] Background:

The basic reinforcement learning methods are modeled as a Markov Decision Process (**MDP**) which are defined by **S, A, P(s, s', a), R(s, s', a)**:

- **S** – A set of possible states within the given environment.
- **A** – A set of possible action which can be taken by the agent to operate within the environment.
- **P(s, s', a) –** The probability that given state **s** and action **a** will yield the state **s'**.
- **R(s, s', a) –** The immediate reward moving from state **s** to state **s'** under action **a**.

Our goal is to learn an optimal policy **π** –as a probability function defined as: $\boldsymbol{\pi(a|s) = \mathrm{Pr}[a_t = a \mid s_t = s]}$, where $\boldsymbol{a_t}$ and $\boldsymbol{s_t}$ are the action and the state in step $\boldsymbol{t}$ in a step sequence of the MDP. An optimal policy maximizes the reward along the step sequence. We express this desired trait using the following notions: we define $\boldsymbol{R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}}$ as the reward for action $\boldsymbol{a_t}$ in a sequence of observations $\boldsymbol{s_0, \ldots, s_T}$, where $\boldsymbol{r_{t'}}$ is the immediate reward given in step ***t'***. We also define the **Q-function** as follows:
$\boldsymbol{Q^{\pi}(s,a) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t R_t | S_0 = s, A_0 = a]}$, which returns the expected reward given action **a**, state **s**, under policy **π.** We would like to use the optimal Q-function, which is defined as $\boldsymbol{Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)}$. So, the optimal policy will be the one which maximizes the Q-function, i.e. maximizes the reward.

The Q-function obeys an important and useful identity which is **the Bellman equation.** It can be shown that the Q-function satisfies the following property:

$$\boldsymbol{Q^{*}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q^{*}(s',a')} \qquad (1)$$

which is derived from the bellman equation. This property allows us to update the Q-function iteratively in various RL methods.

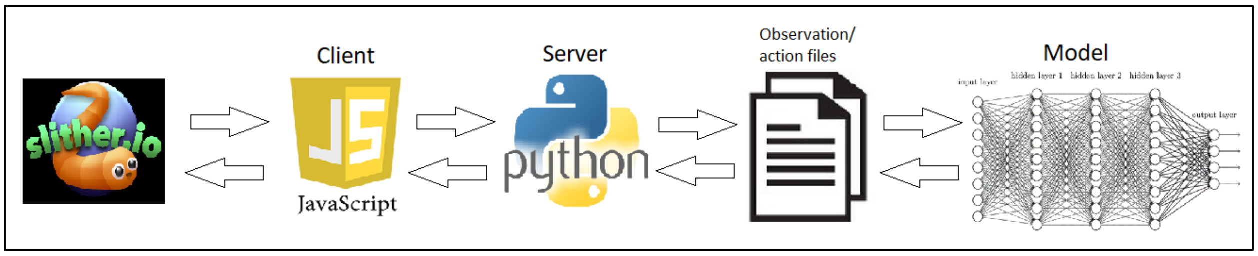
In games where the set of states **S** is small, we can simply fully represent the Q-function as a matrix of states and actions and use it to choose the action which yields the highest expected reward. Unfortunately, in many games (including Slither.io) the set of possible states is enormous, and therefore it is completely impractical to compute a matrix for the Q-function efficiently. Since we cannot explicitly compute it we chose **RL deep learning** methods which allows us to estimate the Q-function. These methods will be presented later in this article.

[b] Deep Q Learning

Deep Q learning attempts to estimate the Q-function using iterative updates, relaying on (1) which guarantees convergence to **Q***. An important method used by this approach is **Experience Reply**; This method maintains an observed set of transitions **(s_t, a_t, r_t, s_{t+1})**, each training step samples a random minibatch of transitions and updates the DQN's weights accordingly. This method utilizes the data more efficiently, allowing us to break the highly correlated sequential observations to create an IID set of transition, allowing us to learning more effectively [1].

[c] Communication Model:

The communication between the model and the game is described in the following scheme:



The game data is sent to the browser (the client side) from the Slither.io server. We use JavaScript to inquire the received data and process it yielding our input. The input is sent in a JSON format to the local server localhost on port 5000, using HTTP protocol. The server then writes the data it received to a file called “Observation.json”. Finally, the algorithm reads the data from the file and uses it to train the model.
When the model chooses an action, the opposite track is used – the algorithm writes the action to a file named “Action.json”, read by the server, that sends it to the client which then sends it to Slither.io server – determining the next action for the agent.

The communication was established using AJAX and jQuery library on the client side, and Flask and Flask-CORS libraries on the server side.
We used a layer of files between the server and the model as it allowed us to completely separate between these two elements – this ensures that if one side has crushed or had some issue, the other can keep running without crushing as well.
At first we feared that this method might create a large delay between the time the observation is obtained by the model to the time the action it chose took place, but as far as we checked the latency was minimal and did not raise issues.

As mentioned in the article, some information did get lost during this process; it was noticeable especially when we wanted to observe when the game has ended. We have yet to determine the cause to it.

[d] Reward Function:

As mentioned in the article, the idea behind the reward function was to convey more information to the model using the reward value.
Here we will present how it is calculated:

$$R_{food} = \begin{cases} \dfrac{\sum_{food} maxFoodSize * r - d(self, food) * food.size}{\max\limits_{food} d(self, food) * food.size * k}, & if\ there\ is\ food\ around \\ m, & else \end{cases}$$

Where $\boldsymbol{r}$ is the distance from the center (the bot) to the upper left corner of the matrix, and $\boldsymbol{d(self, food)}$ returns the distance between the bot the $\boldsymbol{food}$.
This function simply means that the closest the bot gets to some food and the heavier the cluster is, the higher the reward this function returns. This element should encourage the bot to get closer to highly dense food clusters.
$\boldsymbol{k}$ is a scaling constant enabling us to reduce the reward (here $\boldsymbol{k = 10}$), since without it this function returns a relatively large value to other rewards. The $\boldsymbol{maxFoodSize}$ we measured is 16.4. $\boldsymbol{m}$ is a punishment if the agent got a place without food at all (here $\boldsymbol{m = -1}$).

$$P_{enemies} = \begin{cases} \dfrac{\sum_{enemy} d(self, enemy) - r}{\max\limits_{enemy} d(self, enemy) * k}, & if\ there\ are\ enemies\ around \\ 0, & else \end{cases}$$

This function penalizes as the bot gets closer to an enemy. $\boldsymbol{r}$, $\boldsymbol{k}$ and $\boldsymbol{d}$ has the same roles as the previous function (here $\boldsymbol{k = 2.5}$).

$$P_{dcenter} = -d(self, centerOfTheMap) * k$$

This function gives a penalizes as the bot gets further from the center of the map. We have added it to help the agent avoid colliding with the edge of the map which ends the game. Here $\boldsymbol{k = 0.00005}$

$$R_{length} = self.length * k$$

This function gives a higher reward as the bot gets longer. Here $\boldsymbol{k = 0.01}$

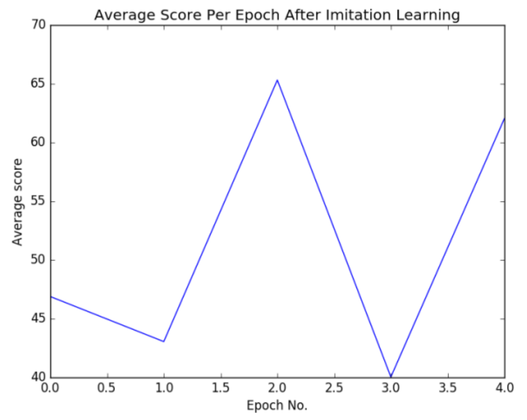
$$R_{gain} = \begin{cases} self.length_t - self.length_{t-1} - k, & (self.length_t - self.length_{t-1}) \leq 0 \\ self.length_t - self.length_{t-1}, & else \end{cases}$$

This function gives a reward if the bot got longer, and a punishment if it lost or didn't received length.
Here $\boldsymbol{k = 1}$

All these components which make up the reward are supposed to help the bot grasping the basic concepts of reaching for food, avoiding enemies and attempting to get longer. A drawback of this method is that it doesn't allow the bot to learn entirely by itself, but instruction it to certain heuristics. Since those elements are basic concepts of the game and we wanted to speed up the learning, we decided to give it a go.

[e] Imitation learning Evaluation:

To evaluate the results we loaded the weights learned during the I-L training session and ran them for 5 epochs. It can be seen in the following graphs that the bot moved from baseline with acceleration towards baseline without acceleration. Further inspection revealed that most of the bot's chosen action were **with** acceleration, but in the game itself it did not accelerate. Because of this bug we can't say for sure if the model did learn something or didn't (see appendix [f]). In later experiments we overcome that issue.



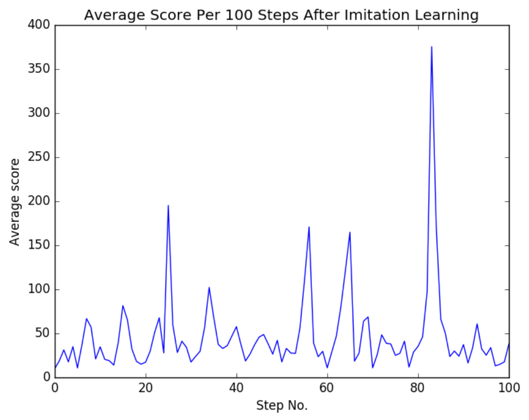


Figure 4 – In these plots we see the agent performance after 10 epochs of imitation learning. We notice that the left plot is between the two baselines – which is not surprising, considering the fact that most of the time the agent tries to accelerate and can't, but if it stops to accelerate from a moment and then tries again, it will succeed, and lose all it's points.

[f] Bug details:

Turns out that if you keep pushing the acceleration button after you lost almost all your points, the bot stops accelerate even after you earn new points. Only if the bot chose to stop accelerate for a moment then his acceleration actions would actually work. This can also explain why the result graph is between the two baselines (see figure 4). Since we wanted the bot to accelerate when he chooses acceleration action, we overcome that issue by forcing the bot stop accelerating when it lost almost all its points.

[g] DQN with constraint optimization:

DQN with constraint optimization is an extension of the DQN algorithm with experience replay.

It takes advantage of longer state-action-reward sequences available in the experience replay memory to estimate upper and lower bounds of the Q-function of a given state and action, and using them as constraints when optimizing the model parameters over the loss function.
Arriving to the following optimization program:

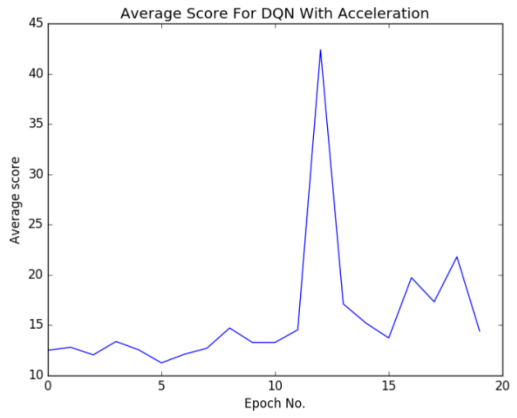
$$\min_{\theta} \sum_{(s_j,a_j,r_j,s_{j+1})\in\mathcal{B}} \left[(Q_\theta(s_j,a_j)-y_j)^2 + \lambda(L_j^{\max} - Q_\theta(s_j,a_j))_+^2 + \lambda(Q_\theta(s_j,a_j) - U_j^{\min})_+^2 \right]$$

This approach has shown faster and better accuracy on 49 games in the Arcade Learning Environment (for further information [4]).

When training our implementation of this model we encountered a problem regarding its runtime (training step takes 17~ seconds in comparison to 0.25~ seconds on the DQN model) that rendered it impractical to use in this project, due to the on-line nature of the DQN algorithm.

Given more time we would have tried to refactor the code to improve its efficiency, and then run it on a GPU. Another solution, which is suggested in the article that was mentioned, is to reduce runtime of the training step by randomly sampling a subset of the constraints, rather than exhaustively using all of them.

Figures:



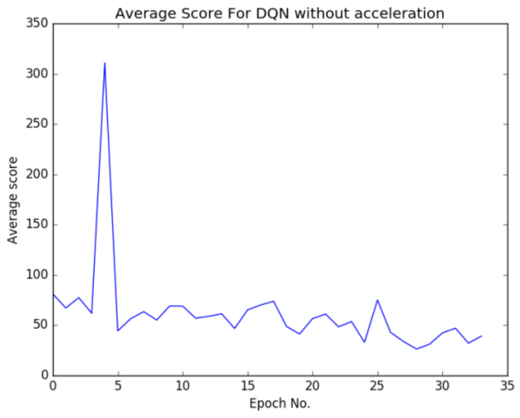
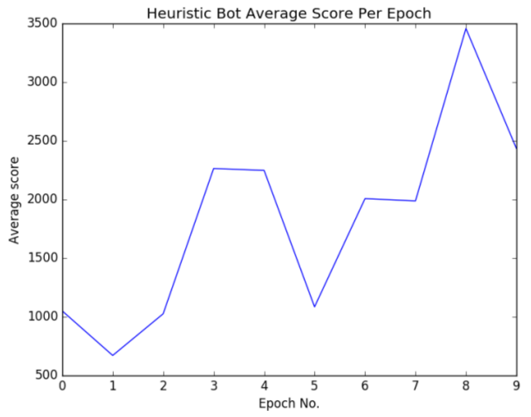


Figure 5 – Training progress of the DQN with and without acceleration. The left plot might look like the start of learning, but as one good game that happens by chance can significantly lift the average score, the results are not to distance from the baseline and it's unlikely that after 10 epochs there will be a significant progress, we can't count it as learning success.



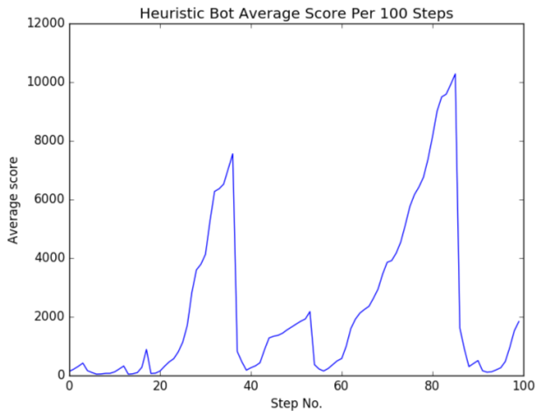


Figure 6 – These plots present the performances of the heuristic bot. These graphs gave us an idea of how a good game should look like, and to which performances we should strive.

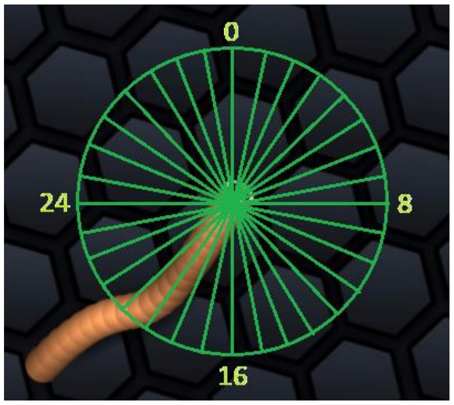


Figure 7 - Each slice represents a direction the bot can choose to turn to. Each angle represents $\frac{2\pi}{NumberOfSlices}$ sector, we marked 0 as the positive y-axis moving clockwise.