Mastering Curve Fever with Double Deep Q-Learning and MiniMax with Alpha-Beta Pruning

Mira Finkelshtein, Daniel Rotem, Aviad Sar-Shalom, Gabrielle Marmur

***Abstract*—Curve Fever, also known as Achtung die Kurve, is a simple multi-player game played on a shared screen where the winner is the last competitor standing. The goal of this project was to use double deep reinforcement learning and the MiniMax algorithm with alpha beta pruning in order to train bots to play and master the game.**

Keywords—Curve Fever, Achtung Die Kurve, MiniMax, alpha beta, Double Deep Q-learning

I. Introduction

Curve Fever is a simple multiplayer game where the winner is the player who survives the longest. Each player starts at a different point on the screen, once the game begins, each starting point elongates into a line that continues growing, moving forward, and leaving a trail behind them. Each player can navigate their line either left or right or neither which is forward. A competitor loses when they bump into any of the surrounding walls or the trails of any player, including their own. The aim is therefore not to bump into anything, the player who survives the longest, wins.

We found a version of the game implemented in the PyGame platform on GitHub[1] and adapted the code to fit our needs. We added an opening screen that allows you to choose which players you want to play, including our players, and a finishing screen with an option of playing again – be sure to check it out.

The simplicity of this game is perhaps what makes it so enjoyable and what drew us to try and build an artificial agent (or multiple agents) that could master this game and perhaps introduce new interesting strategies we have not discovered yet. We chose to employ two different strategies for this problem. The first reinforcement learning approach included training agents with a Double Deep Q-learning (DDQN) algorithm and the second used the MiniMax algorithm with alpha-beta pruning. Each approach models the problem in different ways and resulted in different kinds of agents.

II. Double Deep Q-learning

The first approach, DDQN, is a model-free, off-policy algorithm using two Neural Networks (NN) approximating the operator; Q-value.

A. Q-learning

Q-learning is an algorithm used by an agent to learn the optimal policy with regards to the expected sum of rewards in an environment. An environment is characterized by a set of states S, a set of possible actions A, a set of transition probabilities $P: S \times A \times S \to (0,1)$ such that $\sum_{s' \in S} P(s'|s,a) = 1$, a discount factor $\gamma \in (0,1)$ and a reward function $R: S \to R$.

A policy $\pi: S \to A$ is a mapping from the state space to the action space of the environment. We define the environment as $< S, A, P, \gamma, R >$. The Q value of a state action pair, $Q(s,a)$, is defined to be the expected sum of rewards received by taking action a from state s and following the optimal policy $\pi *$ from then onwards.

Formally,

$$(1)\ Q(s,a) = R(s,a) + \gamma \cdot \left(\sum_{s' \in S} P(s'|s,a) \cdot R\left(s', \pi * (s')\right)\right) + \gamma \cdot \left(\sum_{s'' \in S} P(s''|s',a) \cdot R\left(s', \pi * \left(s''\right)\right)\right) + \gamma \cdot \ldots$$

Seeing as $Q(s,a)$ is somewhat a recursive function, it can be re-written as

$$(2)\quad Q(s,a) = R(s,a) + \gamma \cdot \left(\sum_{s' \in S} P(s'|s,a) \cdot Q\left(s', \pi * (s')\right)\right)$$

So, given the optimal policy $\pi *$, we can find the Q-value of each state-pair action (2), and given the correct Q-value for each state-action pair we could find the optimal policy

$$\pi * (s) = \max_{a \in A} Q(s,a)$$

In Q-learning, the agent holds a Q-table of size $|S| \times |A|$ which is initialized arbitrarily. The cell in the i-th row and the j-th column is supposed to represent $Q(s^i, a^j)$. The agent starts in a random starting point and proceeds to take actions in the environment. At time step t, the agent uses an epsilon-greedy method to choose action a_t and takes that action from state s_t. Based on the reward r_t and the next state viewed s_{t+1}, it then updates the value of $Q(s,a)$ to be

$$(3)\ Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \alpha \cdot \left(R(s_t, a_t) + \gamma \cdot \left(\max_{a \in A} Q(s_{t+1}, a) - Q^{old}(s_t, a_t)\right)\right)$$

By taking a very large number of steps, this process eventually converges to representing the Q-value of the environment with respect to the optimal policy.

B. Deep Q-learning

Although Q-learning is guaranteed to converge for any finite MDP, in real life environments the state space can often be so large that running it is computationally infeasible. This causes a problem not only with the running time but also with the memory needed to store the Q-table used in the algorithm. For example, even a simple game as checkers has over 10^{20} possible states.

In deep reinforcement learning (DRL), instead of holding a Q-table in memory, the agent uses a neural network (NN) that approximates the actual Q-value of each state-action pair. Assuming the NN is a close enough approximator, the optimal policy can then be extracted as

$$\pi * (s) = \max_{a \in A} NN(s,a) \approx \max_{a \in A} Q(s,a)$$

[1] Link to code implementing of the game

In traditional deep learning, the network is usually trained by comparing its output on a set of training samples to the ground truth and using the loss between the two to backpropagate the partial derivatives through the network, and update the weights.

This is problematic for reinforcement learning, as we do not have a ground truth for the function we are trying to approximate. Instead, two similar networks are used in the training process, one for choosing an action and for evaluating the Q-value of the next action. In the process of training, the first network (NN_1) is constantly compared with the other (NN_2) in order to follow a similar logic as in the Q-learning algorithm.

At time step t the agent takes action a_t from state s_t and observes the reward $R(s_t, a_t)$ and the next state s_{t+1}.It stores this transition $(s_t, a_t, R(s_t, a_t), s_{t+1})$ in a memory buffer. After accumulating enough transitions, it samples a batch of transitions at random from its memory and for each transition $(s_t, a_t, R(s_t, a_t), s_{t+1})$ It calculates the loss

$$Loss(NN_1(s_t, a_t),\ rt\ + \gamma \cdot \max_{a \in A} NN_2(s_{t+1}, a))$$

and updates the first network's (NN_1) weights accordingly using back propagation. Every few training steps, the second network's (NN2) weights are replaced with the weights of the first network. This is very similar to the Q-learning algorithm, in which the current value of $\max_{a \in A} Q(s_{t+1}, a)$ is taken into consideration when calculating the new value of $Q(s_t, a_t)$.

III. Modelling the Environment

A. Markov Decision Process (MDP)

An environment satisfies the Markov property if its state signal compactly summarizes the past without degrading the ability to predict the future [1]. In Curve Fever, seeing as each player leaves a trail behind him that does not change, the state of the game at each time step holds all the information of the state at the previous step. This means that it is possible to represent the state in a way that ensures the Markov property, allowing us to treat the task as a Markov Decision Process (MDP).

Technically, our problem can be modelled as a finite fully observed MDP if we represent each state fully, as the number of possible states is finite. The number of possible states is approximately the number of colors used in the game to the power of the size of the board:

$$(4)\ |possible\ states| = |colors|^{\theta(Board\ Size)}$$

Seeing as the RGB board we were working with was $750 \times 600 \times 3$ pixels it was not computationally plausible to learn in a state space of this magnitude. We therefore chose to model the problem as a Partially observed Markov decision process (PoMDP), splitting our efforts into two branches, each representing the state of the game differently; one used the image of the board and the other extracted relevant features. For each type of representation, we used a different model as described in the sections ahead (see section IV and V).

IV. Learning from Raw Input

Learning from raw sensory input, such as images, poses a great challenge for any kind of learning scheme, and for RL in particular. On one hand, dealing with the raw input preserves all the relevant information a system might need in order to learn (as opposed to learning from extracted features, which often reduces the dimensionality of the input and loses information). It stands to reason that methods that learn from raw input will be able to achieve the best results on some tasks, because these methods use intact information, and do not rely on feature extraction that may vary in quality. Learning from raw input achieved impressive results in various fields, including RL [2].

On the other hand, learning from raw input has some clear disadvantages. The high-dimensionality and large volume of raw data requires matching computation power to process. Thus, training takes a long time to complete and requires expensive resources in terms of memory and GPU. The high dimensionality of the input also makes the learning less likely to converge. This is specifically a nuisance for RL with Deep Q-learning, which is infamous for its tendency to diverge, as we will discuss later.

A. Input Preprocessing

After a limited time of exploration, the distribution of states out agent visits is largely determined by its current policy, which may not be optimal. In order to avoid overfitting to this current-policy-distribution, our agent implements an experience-replay mechanism. That is, the agent saves a large dataset of previous states from past games (back from a different time in training when the policy was different), and randomly draws batches from this dataset to train on. Clearly, such a dataset increases in volume very fast, and consumes a lot of the program's memory. This calls for shrinking the input (the images) as much as possible before inserting them into the dataset. Obvious computational restrictions that exist in every network call for the same need.

To reduce the size of the input, we subsampled every image by a factor of 6, and reduced the color scheme from RGB to grayscale. Ideally we would like to blur the image before subsampling, but for efficiency reasons we chose subsampling to be the first step of preprocessing, so that all other steps can be performed on a smaller image (blurring the entire image before subsampling is surprisingly costly. Blurring after subsampling is pointless).

To ease the training process for our agent, we cropped a box around the head of the "snake" our agent controls. This allowed the agent to always see itself as if it were positioned in the middle of the board, with the environment changing around it. We also rotated the box in an opposite direction to the one the agent is currently facing, so that the agent always saw itself as if it were facing up. This special point of view of the agent - always at the center of the board, facing up - does not only ease training. It is in fact necessary to let the agent learn its location and direction this way, because the first steps of subsampling the image and reducing it to grayscale erase the differences between the different colored players in the game, as well as the head of each player. It is thus impossible to learn from the image alone where the agent is going, where is its head and tail, or even which "snake" is played by the agent. Thus, centering and rotating the box around the snake's head solves this problem efficiently and even offers an advantage in learning.

Besides reducing the sample space of the distribution of states our agent sees, rotating the picture has an additional side-effect: the rotation gives us the motion information about the picture (the motion of the snake after the rotation is always at angle 0). This frees us from the need to save a few consecutive frames for each input sample in order to account for motion: motion (at least of our snake,

which is the most relevant in this game), is now given to us for free with a single frame.

Cropping a box around the snake's head forces us to add a margin to the entire board of the game, so that we can crop the box centered around the head even if the head is at the edge of the board. Rotating the box to fit the direction the snake is heading requires an additional crop after the rotation, to ensure the final result contains only data from the original frame, and no black margins that result from the rotation, and vary in size according to the rotation's angle, which could hurt the training process.

The entire preprocessing on a single image looks like this; (1) The full board of the game (400 x 400 pixels), surrounded by a wide margin that will be cropped later (the margin has to be of a fixed size regardless of the box size, because the width of the margin determines the size of the cropped image that will eventually be fed to the NN). (2) The cropped box centered around the head of the white snake (the head is the little green dot). (3) The box subsampled by a factor of 6. (4) The box Reduced to grayscale. (5) Rotated so the head points at angle 0 (a line in angle 0 is parallel to the X axis, pointing to the positive direction). (6) Final crop before feeding the image to the dataset of experience-replay.

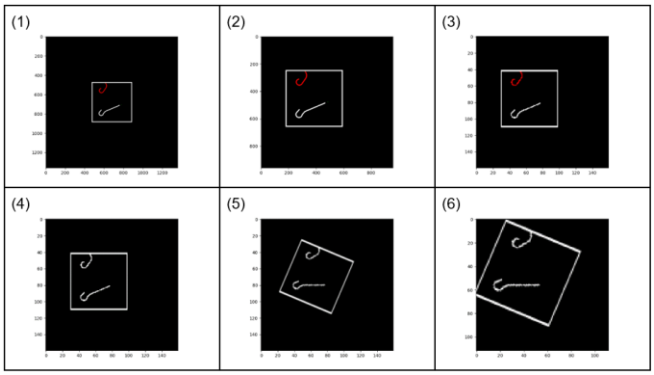

Fig. 1. The entire preprocessing on a single image

B. *Network Architecture*

The model we chose for learning from the image of the board is a Convolutional Neural Network (CNN), with a (3,3) kernel and a linear rectifier (ReLU) as an activation on all the convolution layers and no activation on the output layer. The structure is as follows:

$$
\begin{aligned}
&Input \rightarrow Convolution\ Layer\ (output\ channels: 16)\\
&\rightarrow Convolution\ Layer\ (output\ channels: 16)\\
&\rightarrow Max\ Pooling(stride: 2)\\
&\rightarrow Convolution\ Layer\ (output\ channels: 32)\\
&\rightarrow Convolution\ Layer\ (output\ channels: 32)\\
&\rightarrow Max\ Pooling(stride: 2)\\
&\rightarrow Convolution\ Layer\ (output\ channels: 32)\\
&\rightarrow Convolution\ Layer\ (output\ channels: 32)\\
&\rightarrow Global\ Average\ Pooling(\,)\\
&\rightarrow Fully\ Connected\ (output\ size: 3) \rightarrow Output
\end{aligned}
$$

We chose this structure following mostly the works of [2] on the DQN for playing ATARI games, and the more modern VGG16 model for image classification [3]. The DQN model is a bit simpler than ours in terms of size and depth. It consists of only two convolution layers and two dense layers, as can be seen in this graph:

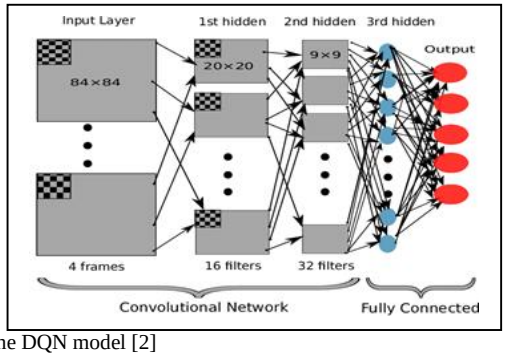

Fig. 2. The DQN model [2]

Since we were working with images of similar resolution to the frames from the ATARI games DQN handled, we tried to use a similar size input for our network (112 x 112), and the same amount of output channels on the convolution layers. The idea was that if 32 output channels are enough to capture the image complexity of an ATARI game, it should be enough to do the same with curve fever.

Since hardware and technology improved greatly since 2013, we thought we can afford a larger, deeper model then the one of DQN, which might achieve better results. To deepen our model, we took inspiration from this beautiful VGG16 model:

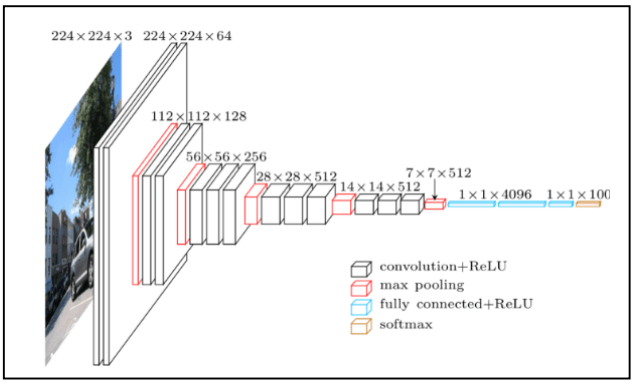

Fig. 3. The VGG16 model [3]

The VGG16 model is meant to work with significantly higher-resolution images than what we need and classify them for real-world complex objects. We wanted the advantages of this reach model, but some of the features seemed unnecessary when dealing with simple images such as the ones in curve fever - mainly, the vast amount of output channels in the convolution layers, that are fully qualified to capture the fine details of real-world objects, and seem completely redundant when dealing with simple curvy lines on a black background.

So, we tripled the depth of our convolution layers compared to that of DQN but kept its relatively small number of channels.

Finally, we added a little tweak of our own and preceded the final Fully Connected layer with a Global Max Pooling across the last convolution layer. Following Cook's [3] advice, (which is inspired by [4]), we further reduced the size of our heaviest layer (the Fully Connected layer) by a factor of 14^2, by simply applying Global Average Pooling to each filter right before the Fully Connected layer. As discussed in the article, this approach has been shown to achieve better results on classifying tasks, with less training time.

As for the activation on the output layer - our model is a regression model that tries to predict the expected reward the agent will get on a specific action in a specific state. According to the game definitions, the reward might be negative. for

example, in some of our experiments we used a reward of -1 to all state-action pairs that lead to a terminal state (when the player "dies"), and a reward of 0 otherwise. Simple activations like Softmax for this layer would not enable a negative reward prediction (of course, Softmax is generally ill-fitted for regression). We tried using PReLU instead (yes, we know it's not a good fit but hey, why not try?), but that led to an even faster divergence of the network. Eventually we opted to continue our experiments without an activation for this layer.

C. Training

We trained the network for 1000 sessions of 100 epochs (games) each, but in practice the process would get stuck on a "single-action-policy" or diverge and crash before completion. The divergence seemed to depend on the variance of the experience-replay dataset. After each divergence and crash we were able to load the weights from the last valid session (before the network started to diverge), reset the exploration rate to 1 and start creating a new experience-replay dataset, which allowed us to continue training from that point. We couldn't get the network to converge in any case, with any length of training.

The model described here is the best one out of dozens we tried. It does not diverge to infinity, and it occasionally learns some minor insight like "avoid the wall you come across in game, if it's in a certain range of angles". It's not horrible, but sadly we can't claim it converges to anything that resembles a real player. However, trying the different models was an opportunity for us to get some hands-on experience with CNNs and Deep Q-Learning. We had to overcome a lot of obstacles and make some interesting decisions just to prevent the model from getting stuck in a "single action policy" or diverge to infinity. The following are some of the more interesting obstacles we encountered:

1) Choosing an optimizer. We tried training our network with Adam, and with SGD (with and without) momentum with different learning rates. Adam seemed to be more stable in the sense that it didn't diverge all the way to infinity too quickly. This makes sense, since Adam is better suited to avoid exploding gradients than SGD. It still diverged on all cases where sgd diverged, although it took it much longer to crash the network completely. SGD diverged much faster in worst cases, but gave better results in best cases. We decided to keep training with SGD, and experimented with learning rates of 0.01 and 0.001. Of the two, 0.001 showed less tendency to diverge, and gave as good results as 0.01 learning rate when it didn't diverge. We chose 0.001 as a learning rate for the rest of the experiments.

2) Regularization. We tried using Dropouts or Batch Normalization after Fully Connected or Convolution layers respectively. We tried using these regularization methods on some layers or on all of them. Surprisingly, using even one layer of Dropout or Batch normalization lead to a very fast divergence of the net's prediction, resulting in NaN predictions and the collapse of the net after less than a 100 epochs. We researched this phenomenon but couldn't find a satisfying explanation for it (still looking for one if you have some insight). We continued without any regularization methods.

3) Rewards. We mainly considered two schemes for rewards in the learning process: 0 for state-action pairs that lead to terminal states and 1 for all others, or -1 for and former and 0 for the latter. Intuitively, these two schemes suggest a somewhat different reward system. The first encourages the agent to stay alive by offering ever-growing rewards, while the second does that by promising ever-shrinking penalties. One could think that this difference between very large reward values and very small reward values will have some mathematical significance for the training process. In our experiments, however, no one scheme did better than the other in terms of the quality of players they produce or their tendency to diverge. We currently train with the second scheme for rewards.

D. Curriculums

The usage of RL raises the question of curriculums. Should we use them, and if so, how? Our first thought was to train the agent on a very small board at first, and slowly increase the board size. This approach has several advantages, the main one being that a small board makes for shorter games, which enters more terminal states quicker into the experience-replay dataset.

At the beginning of training terminal states are the only ones that contain meaningful data, so we desperately need more of them so as to not drown in noise. On the other hand, a small board doesn't allow for a complex state of the game, which mostly consists just of the four walls of the board and one or two short lines before it ends. The agent might learn to overfit to specific cases, like avoiding only straight horizontal or vertical lines, not learning how to deal with anything more complex. In one of our training sessions our agent learned to actively seek the corners of the board and crash into them. It wasn't it's fault - the board it was trained on was too small, and didn't allow it to fully turn back and avoid a wall from most starting positions. It's best course of action was to aim for the corner of that wall, the farthest point it could reach, to get a few more steps of the game.

We tried several board sizes to start the training with. So far, the sweet spot seems to be around a (400 x400) board, which is not too big but still allows enough space for some evading maneuvers. Our best experiment used this size of board and managed to learn to avoid the first few obstacles in some cases.

Another question of curriculums was whether to start training the agent alone on a board, or to train it against other players. Training alone initially seemed natural, as it is a simpler task and is more likely to succeed. In practice, however, training alone showed more tendency to diverge quicker than training against an opponent. This might be due to the extra randomness the opponent adds to the game, that increases the variance of the experience-replay dataset and allows for a more even training process.

V. Feature extraction Network

The second route we took was feature extraction, where we represented each state of the game as a set of features extracted from the game's actual state (The image).

The features we chose to extract:

- Distance to obstacles: Denote the agent's angle θ. We divided the half space defined by $[\theta - \frac{\pi}{2}, \theta + \frac{\pi}{2}]$ into 25 evenly spaced angles, and

for each angle we extracted the Euclidian distance from the player's head position to the nearest obstacle (we bounded the search to 350 pixels). we then normalized the vector to be in the range [0,1].

- Position: We extracted the position of the player in the x axis and the y axis.
- Angle: We extracted the player's angle.

A. *Network Architecture*

We used a fully connected network with 4 hidden layers. The first hidden layer had 60 neurons and each other network had 40 neurons. each layer had a ReLU activation function, and we used a dropout of 0.5 when training the network.

B. *The Training Process*

At first, we tried training an agent in a full-size arena against a random opponent. In each game, the players' initial positions and angles were chosen uniformly at random. We decided to start with a rather complicated task to try and avoid overfitting to a specific board position or angle. However, after a few thousand playing episodes it did not seem like there was any learning going on. In fact, our agent did not seem to be any better than the random player, staying alive longer than its opponent in about 52% of their shared games:

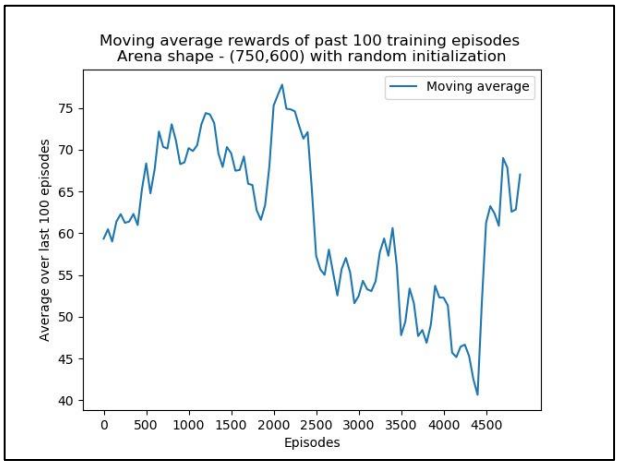

Fig. 4. A moving average of the rewards of 100 training episodes taken over 5000 episodes.

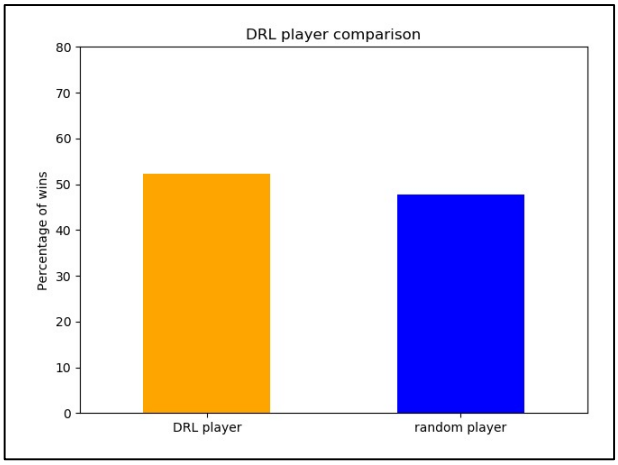

Fig. 5. The DRL player's percentage of wins compared to the random player. As we can see, the DRL player did not manage to significantly improve it's performance beyond chance.

After this failed attempt, we came to the following conclusions:

1) The randomization of the initial position, angle and the opponent made it too difficult for our agent to learn with the limited computational resources at our disposal; We trained our agent using Google Colaboratory which has a limit on daily GPU usage. We therefore were not able to keep training our agent for as long as we wanted.
2) Training an agent in a full-size arena meant that each playing episode was longer, and therefore we would be able to train the agent for less episodes in each GPU session. Since each episode only has 1 terminal state (The one in which the agent collides with an obstacle), we thought this might unnecessarily slow down the learning process.

To address these issues, we decided to try a different approach to the learning process, called "Curriculum learning" (See section V, D on curriculums).

We started from the simplest task we could think of. The agent trained as the only player in a $200X200$ size arena. The initial position and angle were the center of the board, parallel to the x axis pointing right, for all episodes. It became apparent that it was fairly easy for the agent to improve in such an environment, with the agent showing a significant increase in average rewards after only a few hundred episodes:

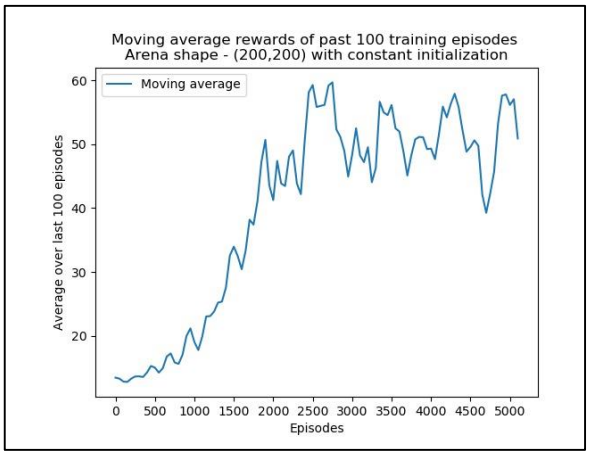

Fig. 6. A moving average of the rewards after training on a bigger board 200×200, as we can see, the player started to improve.

This was a rather comforting moment seeing as it was the first sign of actual learning we had witnessed after countless hours of work. Unfortunately, upon testing the trained agent in a larger environment, the learning did not manage to scale up and our agent exhibited inferior performance to the random agent (One agent proceeded to crash into the wall in front of him, another kept turning left until colliding with its own tail).

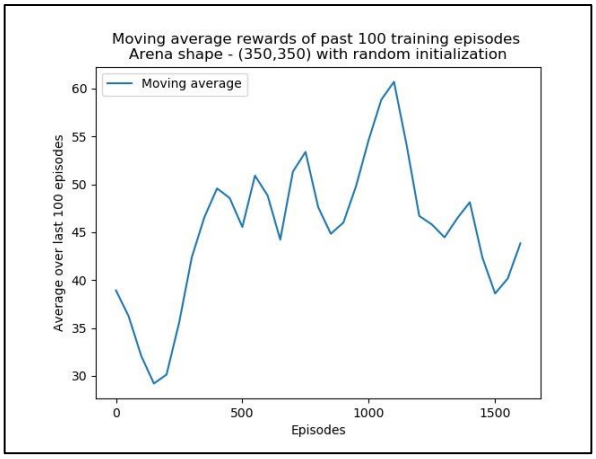

Fig. 7. A moving average of the rewards on a bigger board 350×350, as we can see, the learning did not scale up.

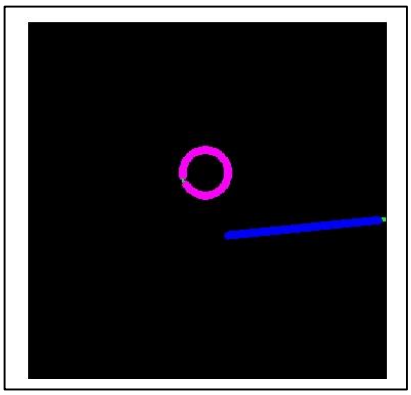

Fig. 8. The player on the 350×350 crashing into the first wall and into himself.

We believe that this was caused by the agent relying heavily on its own angle and position which were supplied in the feature representation of the game's state. We decided to omit those coordinates from the states representation and try to train the agent by only supplying it with the distance to the nearest obstacle in each of the 25 angles.

We trained another agent in a $200X200$ arena with a fixed initial position and angle. As observed before, the agent had no trouble learning to extend its life in such an arena:

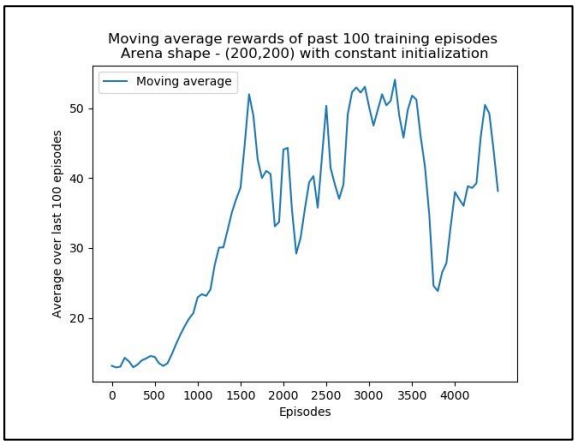

Fig. 9. The moving average of the reward on the 200×200 board when we did not give the position as a feature of the state.

When running a few games with this agent, we noticed that he was able to generalize this behavior to a full-size arena with some success. We therefore proceeded to train the weights in a 350X350 arena, only this time the initial position and angle in each episode was drawn uniformly at random from

$$(0, Arena\ width) \times (0, Arena\ height) \times (0, 2 \cdot \pi)$$

Upon starting the training in this different environment, we had to decide our agent's exploration rate. By the time the agent had finished training in the small arena, the exploration rate had decayed to its minimal value while training, 0.1. Letting the agent train in the new environment with the same exploration rate, would mean he would mostly exploit what he has already learned in the small arena, and might not explore his new environment enough to learn better strategies. However, setting the exploration rate too high might lead to excessive exploration and neglect of what he had already learned. We decided to train 2 agents, the first with an exploration rate of 0.1 and the second with an exploration rate of 0.5 (which decayed over episodes back to 0.1).

As a result of the increased size of the board and the fact that the agents were actually better at avoiding obstacles, each training episode took longer in this arena. This led to us only being able to train the agents for ~1500 episodes (Due to Google Colaboratory GPU limits). From previous training sessions we learned that agents usually need about 2000 episodes to start showing improvement. So 1500 is rather a small number of episodes, and considering the agents had to also generalize to various initial positions and angles, we believe the results of these training sessions do not imply that this direction is hopeless, we believe that given more time these agents would have been able to prolong their lifetime even more. nevertheless, the rewards the agents received during these sessions are shown in the following figures:

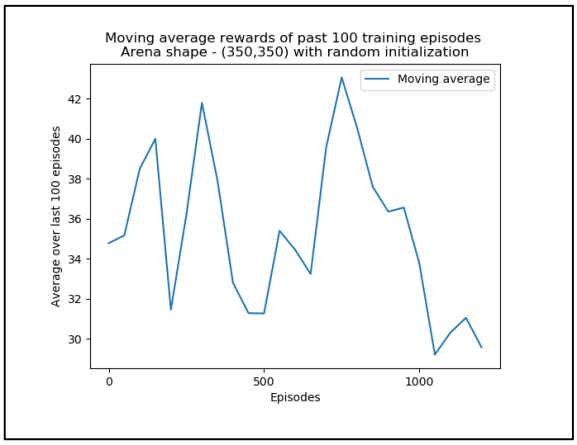

Fig. 10. Moving average of the agent with a high exploration rate.

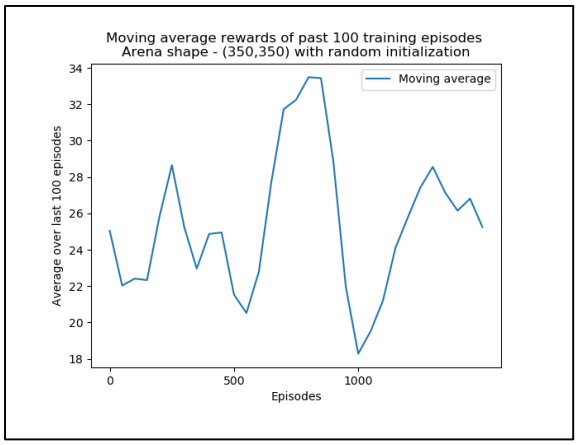

Fig. 11. Moving average of the agent with a low exploration rate.

C. Future Work

If we had more time and resources to train our agent. These are the additional training steps we would have liked to try:

- Taking a slower approach to increasing the board size. We believe a big jump in board sizes hinders the agent's learning in a very significant way, and a more gradual increase would have made the transition easier to learn.
- Adding permanent obstacles to the board. By placing permanent obstacles in the form of other snakes around the arena, we believe the agent would have learned to better avoid other snakes, and to aim for the "holes" in their tails.
- Training the agent against the min-max player. After training the agent with static snakes, we would have loved to try and train our agent against our min-max player. The goal of the game is obviously not to play alone, and if our agent were to train against an intelligent player it may have actually developed strategies trying to block the other player. We assume this type of training would have required a change in the reward function, as living longer doesn't always guarantee winning the game.
- Training against another DRL player. We would have absolutely loved to train our agent against a previous version of itself, and as it gets better, to keep matching it against better versions of its previous self. thus pushing it to constantly improve.

VI. Minimax and Alpha-beta

In game theory, a game tree is a directed graph whose nodes are positions in a game and whose edges are moves. Curve fever is not a sequential game as all players make a move simultaneously. Using a game tree however, allows our agent to consider future states Modelling the game as a sequential game.

The MiniMax value of a game tree is calculated based on the assumption that the two players, called Max and Min, each select their next move in order to maximize or minimize the max player's gain, respectively. MiniMax values are propagated from the leaves of the game tree to its root using this rule. Alpha-beta utilizes the MiniMax value to prune a subtree when it has proof that a move will not affect the decision at the root node.

A. Adapting to a multiplayer game

The MiniMax algorithm, Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, has also been extended to more complex games and to general decision-making in the presence of uncertainty. Curve fever is a multiplayer game where all players make a move simultaneously. We therefore decided to model the game as a 2-player game, where the agent is the max player and all his opponents are the min player.

B. Running Time and Optimization

Seeing as the Alpha-beta agent is an online player, traversing the game tree for each action, we had to ensure it would return quickly to preserve the game's normal flow. This turned out to be the main challenge we faced.

The minimax function returns a heuristic value for leaf nodes which are terminal nodes and nodes at the maximum search depth. We tested the running time and performance for different depths:

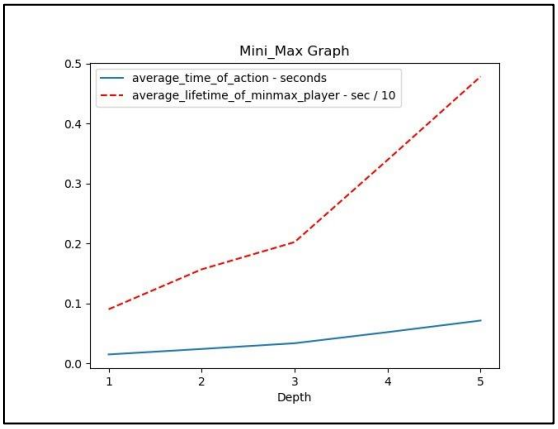

Fig. 12. The average time it takes to get the next move and the amount of time the player survuves as a function of the depth of the tree. As we can see, when we increase the depth we improve performance but increase our running time.

Seeing as each layer of the game tree represents one of the players moves, we needed to represent the opponents' move as one action. Each action of the min player was therefore a move of all the opponents. This considerably increases the branching factor of the tree seeing as there are $3^{|opponents|}$ possible actions for each min layer. We decided to use a sample of 3 of the possible actions for each min layer. This does not affect the game tree for a game with only one opponent. We realized that the min opponents' move was less important for the agent overall, the main benefit of using the game tree was looking a few moves ahead.

Another challenge we faced was to update each successor state without making a copy of the state, seeing as when we sent a copy of the state to each child, the algorithm took too much time. We therefore created a method that updates the state according to the selected move, but then on the way back, we reversed all the changes made. Therefore, we could work with only one copy of the state, this cut down our running time considerably.

We used a 2 dimensional board of the game, representing each player as a different number, this also decreased our running time considerably, especially for the Varanoi heuristic that required us to calculate distance matrices.

C. Evaluating the state

Another important challenge for a good MiniMax agent is coming up with a good Evaluation function for the states. We chose to combine three features that we extracted from the board and combine them in order to assign a value for each leaf of the game tree.

1) Distance from the other players: Each player has his own head position and angle. For each player - we checked the head positions of his opponents and whether they were in the player's field of view spanning 90 degrees:

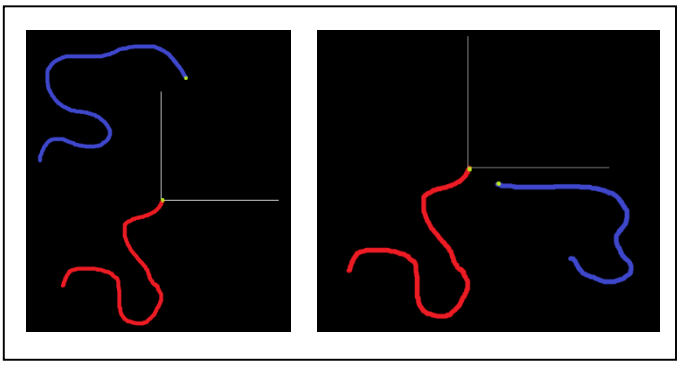

Fig. 13. On the Left: Blue opponent in the red player's field of view, red player gets a negative value. On the Right: Blue opponent out of the field of view, red player gets a positive value

2) The closest obstacle – for each state, we calculated the 40 steps forward in the current direction of the max player, checking if he meets an obstacle and penalizing states with close obstacles.

3) Voronoi Heuristic – this heuristic, based on a Voronoi diagram which is a partition of a plane into regions close to each of a given set of objects. In our case, we calculated the shortest path for each player to every point in the board grid, dividing the board into sections that each belong to a different player that can reach them the fastest:

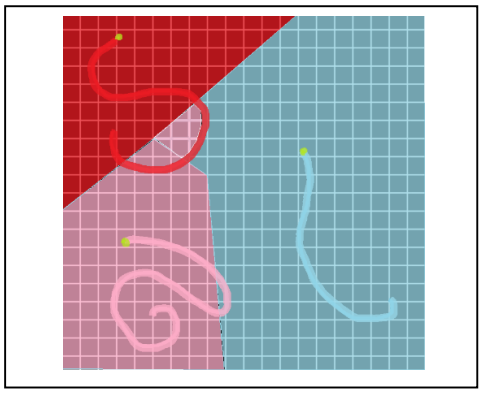

Fig. 14. The blue player is closer to more points on the board than the pink and red player.

Each player tries to maximize the space in his colored area. We give a higher value for boards that have a bigger area the color of our player, thus causing our agent to prefer boards where he can reach more of the board. The algorithm worked perfectly with 2 players, but when we added more players we needed to add adjustments – so we calculated the distances of all the players, and took the minimum value of all the opponents.

We saved a distance matrix in which for each position in the board a number is kept indicating which of the players this place is closest to. The calculation of the matrix is done on the basis of the Dijkstra's algorithm - which fixes every time another player and calculates the distance from this player to every position on the board. After calculating the distances - we took all the matrices and distances of all the opponents - and calculated the minimum value for each position on the board. Then we subtracted the opponent distance matrix from the minimax distance matrix and summed the distances – thus creating our partitions.

VII. Results

We let our players play 100 games against each other and summarized their percentage of wins in each match:

TABLE I. PLAYER'S PERFORMANCE

Players	**Percentage won by row player against column player**			
	DDQN	Alpha-Beta	Human	Random
DDQN		65%	50%	98%
Alpha-Beta	45%		40%	97%
Human	50%	60%		100%
Random	2%	3%	0%	

Fig. 15. Table of the players' performance

These results indicate that the DDQN was the better player of our two agents, performing just as well as the human player. We believe that if we had more time we would manage to improve its performance further.

References

[1] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[3] Jaderberg, M., Simonyan, K. and Zisserman, A., 2015. Spatial transformer networks. In Advances in neural information processing systems (pp. 2017-2025).

[4] Lin, M., Chen, Q. and Yan, S., 2013. Network in network. arXiv preprint arXiv:1312.4400.