A Comparison of Infexion Gameplay between Game Tree Search Minimax and Deep Reinforcement Learning (TD Learning)

Introduction

In trying to look for an optimal agent to for competitive Infexion gameplay, we decided to explore multiple approaches that seemed viable to tackle the challenges that come with the complexity of the game. The initial ideation process was to make use and improve upon our current knowledge of machine learning and deep reinforcement learning methods by creating a Duelling Double Deep Q learner that utilises a convolutional neural network that would play against agents and update its parameters through rewards. In our tests we concluded that a minimax agent that utilises a model-based policy fined tuned by human intuition beforehand, outperforms the neural network by a landslide, resulting in a form of reinforcement learning that some would consider cruel punishment. Nevertheless, we have detailed all the different methods, approaches and implementations that were explored throughout the process of submission. We would like to acknowledge some of the great minds that we have taken lessons upon during our progress namely: Richard Sutton, Steve Brunton, Phillip Tabor, Andrew Ng, along with our universities own Chris Leckie and Lea Frermann for all we have learned about machine learning and artificial intelligence.

Approach 1: Minimax

In finding the next best move, our choice of a tree search algorithm was minimax. Minimax returns an optimal solution (assuming the opponent is perfectly rational), making it preferable over Monte Carlo Tree Search (MCTS). Although minimax falls short in being time and space efficient, implementing an alpha beta pruning algorithm along with other optimisations later explained produces a sub-optimal solution for a drastically reduced search time.

The optimality of Minimax is heavily reliant on an accurate evaluation function, to measure the numeric reward based on an action taken. Using the rules and strategies of the game, our evaluation function is defined by the following four heuristics and its strategic motivation:

1. Difference between the number of players (*Risk of Losing*) - In the game of Infexion, power has no influence on whether the cell can be infected. As a consequence, a player with a cell that has a power of 5 faces a greater risk of losing than a player with 5 cells that have a power of 1. Therefore, having fewer number of cells should be recognised as unfavourable.

2. Difference between the total power of players (*Score*) - Power plays a role in determining the winner. When the power of a player is 0, the player has lost. Where the number of turns exceeds 343, the winner is determined based on power. Hence, maximising the players power and minimising its opponents can be simplified as the goal of the game.

3. Power that can be captured – Power at risk of being captured (*Capturing Strategy*)- Another objective of the agent is to reduce the opponent's power, of which the only way is by spreading. This heuristic incentivises moving to a state that captures the most power of the opponent yet does not put the player in the position prone to being attacked.

4. Incentivises forming clusters (*Clustering Strategy*) - Like a biological infection, a host with a smaller surface area is less prone to being attacked. The notion behind this heuristic is that by forming clusters of cells in the game, an opponent will be less likely to infect the player's cluster, since in the next move the player can capture the opponent's attacking cell (as shown in figure 2 below).

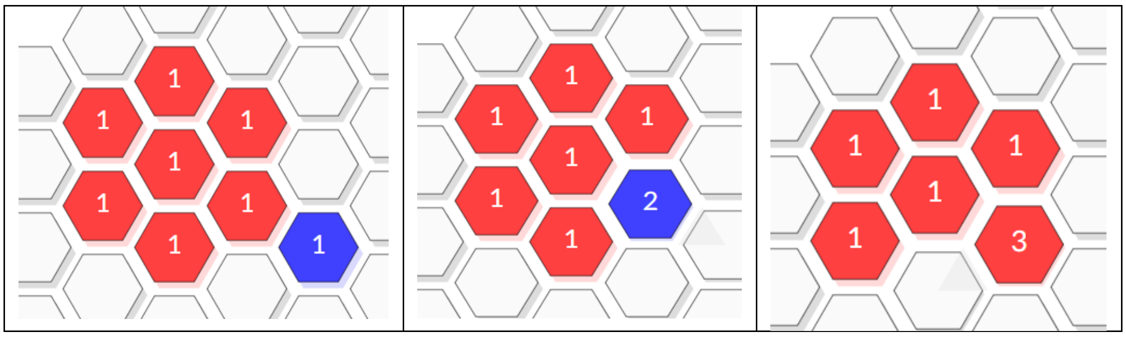


Figure 1: By clustering cells together, rational opponents are likely to attack.

Through repeated simulations of the game, we arrived at the following weights:

$$Evaluation\,Value \;=\; 3\,(Heuristic\;1)\;+\;8\,(Heuristic\;2)\;+\;10\,(Heuristic\;3)\;+\;10\,(Heuristic\;4)$$

These optimal weights were found by looking at whether the weight produced the consistent wins in the least number of moves. With the consideration of weights, the evaluation value is mainly made up by the difference between then total power (heuristic 2) and the strategic clustering of cell (heuristic 4), both heuristics were significant in determining an action that increasing the chances of winning.

Time and Space Optimisations - Minimax

Minimax is a time and memory exhausting approach. The branching factor can be expressed as $(49 - n) + 6n$, where n is the number of the player's cells. In the worst-case scenario, the branching factor on a minmax search tree is 289 (where n = 48). To counter this shortfall, we reduced the search space minimax had to evaluate through alpha-beta pruning and depth cut-off.

In further optimising alpha-beta pruning, games states that took on a spread action as the next move were searched before spawn actions. This is because, in early stages of the game, spawning in a particular cell (e.g. (2,2)) compared to the cell adjacent often produces evaluation values that are indifferent. By considering distinct values generated from spread actions early on, we increase the likelihood of branches getting pruned.

At the start of the game, the search for the next best move can be simplified to a random agent. Due to the infinitely repeating nature of the board, spawning in any position on the board for the first move will be the same. As such, we can reduce run time by having player red (the player who moves first) simply spawn in a random cell, rather than calling minimax. As for the second move by player blue, to avoid a loss (shown in the diagram below), we can confine the search space for blue to spawn in any cell other than the cells surrounding player red's first move to reduce run time.

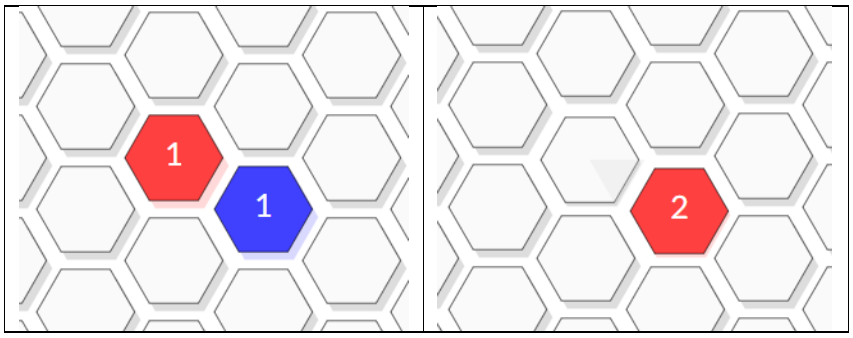


Figure 2: Spawning in the cell adjacent to the first move may result in a loss.

Additionally, we found that restricting the number of spawn actions a player was allowed to play had an improvement on run time. With less spawn actions made, meant the player had fewer cells on the game board which translates to a smaller branching factor (as given by $(49 - n) + 6n$). Though this improvement on run time is at the expense of finding the optimal next best action. As shown in the table 1 below, when restricting the number of spawn actions to 5 against a game with the random agent, changes in the winning rate were negligible, the minimax agent would always win. However, against more complex agents like a power greedy agent, the same restriction to 5 spawn actions showed a decrease in the games the minimax agent won. Depending on the agent played against, there is room for the run time of the minimax agent to be more efficient. For the purposes of submission, the number of spawn actions allowed was not restricted, so that the agent could have the highest likelihood of winning.

Table 1: A comparison against other agents (based on a simulation of 20 games):

	Winning Rate (Average turns for a win)	
	Vs. Random Agent	Vs. Power Greedy Agent
Minimax Agent restricted to max 5 spawn actions	100% (taking on average 15 turns)	90% (taking on average 24 turns)
Minimax Agent with no restriction	100% (taking on average 23 turns)	100% (taking on average 39 turns)

Approach 2 - Duelling Double Deep Q Network

Summary

The inspiration for selecting a Deep Q network for the game of Infexion lies in the representation of the board state. As the state environment is fully observable and could easily be represented as images/multidimensional arrays. The initial approach was to be able to encode the state of the board and pass it through a Convolutional Neural Network (CNN) to be able to learn extract low dimensional features from the given high dimensional input. The number of output nodes are mapped to the total number of possible actions in the game (I.e. The total dimensions of the board * the total number of actions possible on a hex), and due to the large number of possible permutations of the board, contributing to an even larger state-action space, the approach of Deep Q Networks initially seemed appropriate to be able to estimate the advantage of taking an action given the value of being in a certain state. A simple goal resulted in a multitude of adaptations and optimisations in attempt to realise a working Temporal Difference model that abides to the programmatic constraints we were given. The remainder of the report would consist of an analysis of this approach and a discussion as to why it is outperformed by the minimax agent.

Approach

The overall implementation involves encoding the 7 × 7 board representation as a 4-channel image as the inputs to CNN and training it over multiple playouts while updating the parameters via gradient descent. As the training updates were computed over mini-batches stored in a replay buffer, the gradients of the Temporal Difference (TD) error were propagated through each layer using its first and second moment estimates calculated with an (Adaptive Moment)ADAM optimiser. The network was trained through a dedicated environment to be able to consecutively update its parameters over multiple games in succession to facilitate training with minimal supervision. All the code that was written for this training architecture was done using native python with the additional Cupy library, (a cuda enabled NumPy library) that would enable us to save and load the trainable parameters interchangeably with NumPy for the submission. The details each of each component of our implementation shall be discussed below with focus on details of the Duelling architecture, CNN, optimiser, training environment, hyperparameters and performance.

Convolutional Neural Network

The CNN layers were built from the ground up from a base python class that was unit tested from a simple XOR prediction model up to the MNIST handwritten digit dataset for layer validation. The inputs of the network are encoded into 4 channels of binary representations (ones and zeroes) 2 of the channels use Ones to include the position of each player on the grid and the other 2 channels encode a flattened view of the total amount of spread that would result from a player cell spreading across all its 6 possible Hex dimensions. To account for the torus like behaviours of the board wherein a cell can loop around from opposite sides of the board edges, the channels are

padded by its wrapped view in which its 7 × 7 image is expanded to dimensions 13 × 13 to take into account the "hidden" adjacent 3 cells around the edges which otherwise wouldn't be visible to the CNN. The network layer architecture consists of 3 convolutional layers followed by 1 fully connected layer after flattening and the output layer of which is split into the value and advantage streams as per convention of Duelling Networks. The convolutional layers each have kernel sizes of 3 × 3, and the number of kernels per input in sequence of layers are 7, 7 and 14. As zero padding isn't used during the forward propagation the reduction in dimensions from "valid" padding result in a more manageable size to be fed to the fully connected layer.

D3QN

The foundational concept of Q learning algorithms models the task of prediction as a finite Markov decision process in which the return at a time T can be estimated as the sum of rewards from time t to T. This allows for the estimation of future rewards to be reduced to the equation below:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

Where G is the return at time t, which is defined by the sum of the rewards from time t+1 up to a terminal state, multiplied exponentially by a discount factor γ. The equation can be reduced even further define the return at time t+1 to be the sum of rewards at from time t+2 onwards. The same follows for state value functions where:

$$v_\pi(s) \doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

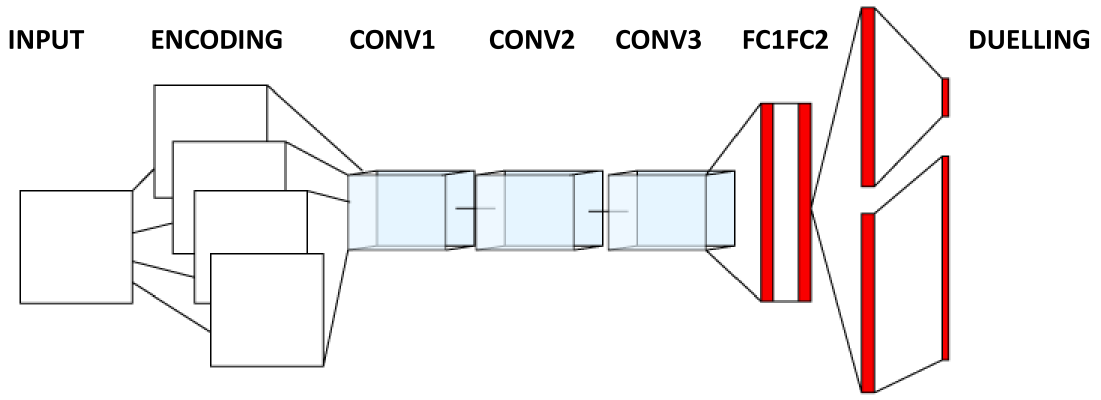
The predicted value at state s given a policy π, can be defined as the estimate of the reward at time t+1 summed with the predicted value of the next state discounted by a factor γ. This is also applied to the state action functions in Q learning to estimate the quality taking an action a given a current state s. Without going into further lengthy derivations, the version of Duelling Deep Q learning where the estimates are drawn from a set of parameters θ , and the quality of being in a state s is split into the value of being in that state plus the incremental advantage of taking an action at that state, parameterised by θ given by the equation below:

$$Q(s,a,\theta) = V(s,\theta_1) + \left(A(s,a,\theta_2) - \frac{1}{|A|}\sum_{a'} A(s,a,\theta_2)\right)$$

Where the advantage is subtracted by the mean or in some cases the highest advantage A to be able to recover the quality of a given state after the value and advantages are split. In Duelling Double Deep Q learning, there exists 2 separate networks where the online network makes predictions for every state observation and is subsequently updated after a given batch size threshold is reached. The other offline network is fixed and only used to compare the predictions of the online network at time t and the actual observation of the offline target network at time t+1. Hence the responsibilities for the two networks are split in which the advantage function is derived from the online network while the state action function is estimated by the offline network given the next state with the Q-target shown below is used to compute the loss function for the update rule by the immediate reward and the discounted estimate of the optimal Q value at state t+1.

$$\left(R_{t+1} + \gamma \max_a Q\,(S_{t+1}, a)\right)$$

The diagram below is an illustration of the network architecture attempted with the initial board representation split into four channels and fed through 3 convolutional layers and 2 fully connected layers to produce a dual stream output of state action representations. The Rectified Linear Units were used for the activation functions between the layers aside from the dual stream layer where no activation was included to preserve the linearity of the output. This network is duplicated in D3QN to provide both an offline and online network for training.



Gradient Descent with ADAM

As the training of the network was done over batches of samples in a replay buffer, an optimiser was used to calculate the exponential decay of the first and second moment estimates. This is used to as a method of discounting the contribution of previous gradients that were estimating the moving average from other optimisation methods that use Momentum. The target of which is to smooth out the oscillations of the gradient so that it converges towards a global optimum while enabling large enough step sizes to skip past local minima. The moment estimates are calculated by taking into account its exponential weighted average using the hyperparameters beta1 and beta2.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

On every update iteration t the parameters mt and vt are calculate by its value on the previous iteration and added upon by the current gradient by its estimated decay rate. A step is also included below to correct the bias related to stochastic gradient descent.

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} \qquad \widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

These values are then used to update the weights and biases for each layer where the both the weights and biases have their own respective moment estimates kept as non-trainable parameters. The constant epsilon is included to ensure that no 0 division errors occur.

$$w_{t+1} = w_t - \widehat{m_t} \left(\frac{\alpha}{\sqrt{\widehat{v_t}} + \epsilon} \right)$$

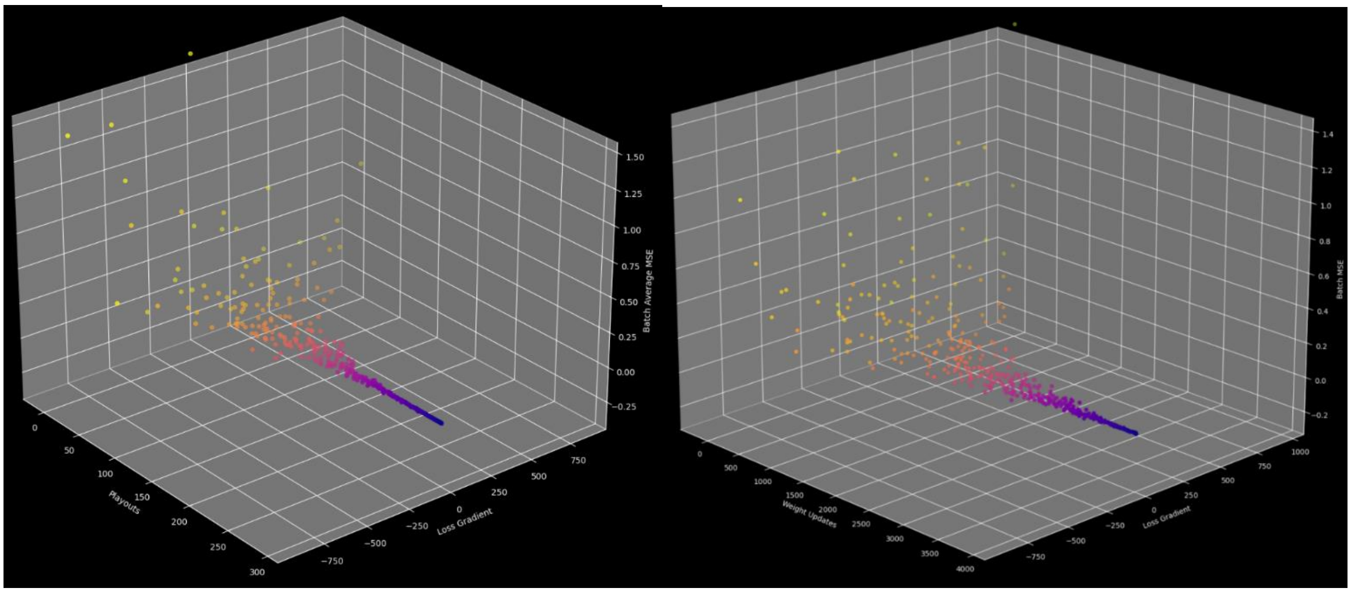
Training environment

We wrote a customised training environment to be able to run continuous tests while our agents were pitted against each other. This enables us to send customised callbacks to the agents at the same time while maintaining a persistent environment, such as notifying the neural network of a rewarded state.

Network Parameters and results

Network layers	Trainable parameters		**Training Hyperparameters**	Value
Conv1	1099		Step size	0.1
Conv2	1008		Epsilon Greedy, decay and minimum	1 /(1e-3) / 0.01
Conv3	784		State-Action Replay Buffer Size	1000
FC1	117649		Minibatch Size	20
FC2	117649		Weight Transfer Period	200 Turns
Duelling	117649		ADAM beta1/beta2	0.9/0.99

The graphs below highlight the amount of time it took to converge on a large training time with a small step size of 0.0001 that was tested to see how long it would have taken to converge. With a step size of 0.1 with the optimiser it took roughly 30 playouts for the gradients to converge while the plot below shows that nearly 300 playouts or 4000 weight updates were required for convergence.



Performance Discussion

In the end, after training the network while swapping out different hyperparameters layer architectures, and data-preprocessing, the end result of a duelling-network architecture for a dynamic action space underperformed against mini-max search in almost every playout simulated aside from the few instances in the beginning of the game where it is likely to clear the board with few moves. We hypothesise that the underperformance here was due to the dynamic action space that result from the ability to spawn on any part of the board as well as the difficulty of making an action prediction out of 343 actions. In hindsight other approaches may have been used to fully take advantage of the convolutional neural network architecture such as having it play the part of an observer as agents are playing against each other and having the network learn from the state actions of the winning agent of a given playout. If such a network were used as part of a curriculum based learning, or used in combination with a softmax activation as a multi-class prediction classifier, it would likely have resulted in a learner that could easily outperform the duelling architecture implemented.