Aegis Wing: A Study of Game Decision Algorithms

Daniel Varivoda, Jose Mari Ian Lou, Ramzi Adil
Northeastern University, Khoury College of Computer Science
Varivoda.d@northeastern.edu, lou.jo@northeastern.edu, adil.r@northeastern.edu

Abstract

Games have been used to test the capabilities of machine learning since its inception. Checkers was one of the first games used to test one of the earliest learning algorithms-minimax (K. D. Foote, 2022). There have been many algorithms that have developed since then, and today one of the most popular is deep learning. Our goal was to test the capabilities of early machine learning techniques, expectimax in particular against a modern technique; specifically deep learning, on the game "Aegis Wing". We chose this game because it has more dynamic states and complex agents than traditional games such as checkers or chess. We learned that expectimax works better than the deep learning agent before the deep learning model reaches competency. However, once the deep learning model reaches competency, it performs significantly better than expectimax and can make decisions significantly faster than the expectimax agent. In the future, it would be interesting to see how expectimax fares against deep learning with a larger depth, but for now it is clear that in regards to time and success, deep learning outperforms expectimax.

Introduction

Games are a fascinating method used to test a machine learning algorithm's ability due to their endless range of complexity and the common interest in the games among people. Since the inception of computers, people posed the question of whether they could match or even exceed human intellect. Since measuring or even defining human intelligence is a difficult task, many researchers decided to tackle the question by challenging human intellect in specific tasks. As humans are constantly playing and inventing new games with ever more complex state spaces, they are seen as an invaluable research tool in benchmarking the computational ability of artificial intelligence (AI).

The use of machine learning in solving game tasks has been present from its very inception. As early as 1949, the "father of information theory", Claude Shannon, published a paper detailing how he theorized a computer might be programmed to play chess. The approach suggested by Shannon is today known as the Min-Max Algorithm and was one of the very first steps to modern and complex AI systems that are common today (Shannon, 1950). Echoing the sentiments of many future researchers he believed the game of chess was an ideal challenge for computers to start with since chess has sharply defined operations (chess moves) and terminal states (check mate). Chess is a game that is neither too trivial or too complex for a solution to find a solution. The game requires "thinking" for skillful performance which "force us either to admit the possibility of a mechanized thinking or to further restrict our concept of 'thinking' (Shannon, 1950).

In one of the very first experiments of its kind, Arthur L. Samuel, who popularized the term "machine learning", while working under IBM, was able to implement Shannon's Minimax algorithm that was able to beat a human in the game of checkers, using Alpha-Beta pruning to reduce the huge state space (Samuel, 1959). While the program was still based on the Minimax algorithm it incorporated a groundbreaking step: learning. The program became better over time without human intervention, through the use of two novel methods; rote-learning, meaning it could store the values previously evaluated positions and not waste time exploring previously explored branches and learning-by-generalization, which modifies the evaluation function based on previously played games. The parameter weights were changed to lower the difference between the evaluation function and the actual strength of a certain board state. Learning by generalization was a completely groundbreaking idea as it allowed the program to learn what a good game position is without expanding branches. By 1962 Samuel and IBM had enough faith in the program that they demonstrated its ability and generated public interest in AI through a televised checkers match against a true human opponent. Though Samuel's algorithm did not "solve" the game of checkers, as was mistakenly reported at the time, it was an incredible step forward in the world of a AI. Before the multilayer peceptron that is so unbiquitous in modern AI was even invented, Samuel was utilizing early decision theory algorithms to test the capabilities of AI against humans.

Checkers was largely forgotten for a time, due to the untrue but lasting impression that Samuel solved checkers. Chess was seen as the next big challenge for AI. Many teams tried and achieved significant results, however no programs were able to consistently beat master level players of either checkers or chess. The first program to truly surpass humans at a complex game was the checkers program CHINOOK and was developed starting in 1989 by a team led by Jonathan Schaeffer at the University of Alberta, nearly three decades after Samuel's groundbreaking checkers program (Schaeffer, 1996). Building off Sameul's work, a team lead by Feng-hsiung Hsu developed a novel program called Deep Thought (Berliner, 1989). The most groundbreaking aspect of this program was that its evaluation function did not have hard coded parameters and was instead tuned automatically using a database of games between master chess players. Unlike previous chess programs which all utilized ideas such as alpha-beta pruning and selective extensions, most had no learning component and ultimately derived all their intelligence fully from their human creators.

The next great breakthroughs in AI were derived from a different game than the common chess or checkers examples that were close to being solved: backgammon. Backgammon has a huge branching factor and the traditional tree-search-with-handcrafted-evaluation-function approach does not work well. Renowned AI researchers Gerald Tesauro and Terrence Sejnowski explored an alternate approach based on learning a good evaluation function; a goal that had been abandoned since Arthur Samuel’s work (Tesauro and Seinowski, 1989). They trained a neural net to accept a backgammon game position as an input and a potential move, and then output a score measuring the quality of that move. Since the model no longer requires an evaluation function, their approach removed the need for hard coding human intuition into the program logic and instead allowed it to use human intuition in the form of features.

This revolutionary approach led Tesauro to take another leap forward in AI, one of the first hugely successful applications of reinforcement learning. Unlike supervised learning, which approximates some function with very specific inputs and outputs, reinforcement learning deals with finding optimal choices in different situations. More specifically, the program attempts to learn different possible states and how agent actions can affect those states. With the stage set for reinforcement learning and such powerful algorithms available, researchers began work over the next few decades to solve different games like the deep neural network based AlphaGo which beat a grandmaster Go player. Go is largely thought to be one of the most difficult board games left for AI to master. Other difficult and interesting games which AI researchers worked to solved include DOTA,a popular real time strategy game with an almost infinite state space and dynamic conditions.

As solving games has been at the forefront of AI innovation, our paper aims to explore the techniques used by previous researchers when applied to a relatively modern video game with dynamic state changes from start to finish. The game “Aegis Wing” can have complex and dynamic states based on whether new agents are added to the state and varying agent behaviors. Some agents can act stochastically, others can act deterministically, while other agents act following heuristic based policy. This makes it different from traditional games such as chess where the amount of pieces can only ever decrease as the game progresses and there is a set amount of piece types on the board (i.e. each team only ever has 2 rook or less at any time in the game). The “Aegis Wing” game also takes relatively simple action inputs. These traits make Aegis Wing an interesting game and an ideal simulation to test the computational power of our decision making algorithms. We pit an upgraded version of the classic Minimax algorithm, the Expectimax algorithm, against a simple feed-forward Neural Network (NN) with Deep Q-learning. We expect to see that the Expectimax algorithm performs significantly worse due to computational limitations. We would like to test the ability of a simple neural network to consistently win a complex and dynamic games.

Methods

The Aegis Wing Game and Game State

Aegis wing is a 2D side-scrolling arcade game. In the game, a player must survive an onslaught of enemy ships before the ships destroy the player. Avoiding enemy ships, projectiles and shooting down enemy ships are key to survival. This game is interesting because it offers more complex and dynamic states as compared to traditional games like chess or checkers. Our goal was to model this game and see how adversarial algorithms and reinforcement learning compare in terms of effective decision making in a complex and dynamic environment.

Environment

The environment of the model can largely be defined as the borders of the game as well as the positions of any existing agent(s) on the board as well as “bullets”/projectiles. The gameboard can be thought of as a 2D array that models all available positions in the gamestate. Other parameters of the environment include the amount of turns left and the amount of lives a player has left. This information is stored in our program in the GameState class as the following attributes: current_agents; a list of AgentInterface type objects, current_projectiles; a list of ProjectileInterface type objects, gameboard; a GameBoard class object, and turns_left; an integer.

Agents

There are 2 kinds of agents in the game; a “ship” (AgentInterface type object) agent and a “projectile” (ProjectileInterface type object) agent. Projectile agents are instantiated when a ship agent takes a “fire” action.

Ship Agents:

This model contains 4 general classes of agents; Player agents, SimpleGoLeftAgent, BasicFireAndMoveAgent, BasicCounterAgent, and a HeuristicAgent. These classes can be differentiated generally on the basis of actions they can take and their strategies. Ship agents have a set amount of hit points (hp). In the case of player agents, the hp is set to 3, in all other cases the hp is set to 1. Ship agents are only destroyed/removed from the gamestate if their hp reaches 0.

We built 3 different kinds of player agents in this model: standard Player agent, expectimax agent, and a reinforcement learning Agent. All player agents can choose from the following actions: move left, move down, move up, move right, stay in place, and fire. The standard player agent's strategy is to take inputs from the user and perform the corresponding action. The expectimax agent chooses its own action based on predictions generated by the expectimax algorithm. The reinforcement learning agent takes in an input of the current state and outputs a vector of predicted Q-values for each possible action in that state. From this vector the reinforcement learning agent chooses the action corresponding to the highest Q-value.

There are 4 types of enemy/antagonizing agents; SimpleGoLeftAgent, BasicFireAndMoveAgent, BasicCounterAgent, and a HeuristicAgent. The SimpleGoLeftAgent can only take the move left action. The BasicFireAndMoveAgent can choose from the following actions: move left, stay in place, move down, move right, fire. It chooses an action among those actions randomly using a normal distribution. The BasicCounterAgent can choose from all available actions. Its goal is to go to some ideal position (generated randomly) and stay in that position for some number of turns. It will fire bullets during the duration of its stay at the ideal position. Once that time period is over it will repeatedly take the move left and fire action until it is destroyed or exits the map. The HeuristicAgent's can also choose from among all possible actions. However, every time it moves, it will also perform a fire action. Additionally, it will "chase" the player for a set amount of turns and take the fire action if the player agent is in the same row (y-position) as itself. Once the time duration ends, it will act like the BasicCounterAgent and move left continuously until leaving the board.

Projectile Agents

Projectile agents can only take one action; a movement on the horizontal or x-axis and only in one direction. Projectile agents are allowed to overlap with each other. If a projectile agent created by a player agent collides with an enemy agent i.e. the position of the bullet and the enemy agent overlap, then the bullet gets destroyed and the enemy agent loses some hp. This also applies in the case of an enemy projectile and a player agent.

Actions

An agent can perform any number of actions. Most actions will update the agent's position. All the possible actions are: move left, move down, move right, move up, fire, and stay in place. In the model, these are represented as the Actions enumeration class i.e. Action.Left, Action.Down, ..etc. An agent may also be able to perform the fire action simultaneously along with a directional action like Action.Up. In regards to legal actions, there are only a few conditions to be wary of. Agents are not allowed to take an action that would move them beyond the y-axis (row) boundaries of the board. The player agent types are specifically prohibited from taking actions that would move them beyond the game board (both the y and x axis). Enemy agents however, are allowed to enter (spawn) and exit the board via the x-axis (column) boundaries. All enemy agents will eventually be removed from the board. This is because either they are destroyed or because the enemy agents are biased to move towards the left side of the board and eventually exit.

The Game State

The state of the game can be thought of as the environment and all existing agents. This information is encapsulated in the GameState class. Much of the game logic resides in this class. The most important methods of this class are generateSuccessorState, getStateAfterAction, and getStateAtNextTurn. The rest of the methods are mostly set up or helper methods to these functions. All of the important methods will create the subsequent game state which contains updated information on agent positions and agent existences as a consequence of some individual agent taking some action. This is useful for progressing the game and also for creating predictive models, especially in the context of expectimax.

The Game Loop

The set up for the game starts with creating an initial gamestate. Set up methods are called which will define the basic parameters of the game such as length and height of the game space/game board, the amount of turns a player needs to stay alive to win the game, and the amount of lives a player has. In our model, the gameboard is 8 units long, 7 units high, the amount of turns to survive is 300 and we always set the amount of lives a player has to 1. The amount of lives only decreases if the hp of the player hits 0.

Lastly, one must configure the enemy spawn parameters. This is done through the EnemyPicker class. This class must be configured with the general spawn rate, as well as the enemy types allowed to spawn along with the specific spawn rates of those enemy types.

When the game loop begins, all existing agents choose their actions and perform the action. Projectile agents all move first, followed by ship agents. Once all agents have taken their action, calculations are performed to check for any ship to ship collisions (player and enemy ship collisions) as well as projectile and ship collisions. The list of existing agents is updated as necessary, following collision calculations (i.e. if ships are destroyed or enemy ships exit the board). The turns left to survive is decremented and then enemies will be spawned (i.e. added to the current_agents list) based on general spawn rate, specific spawn rates of enemies, as well as enemy limits. In our model, the general spawn rate is 50%, the maximum enemies allowed at any single time is 5 and the following contains the specific enemy spawn rates: HeuristicAgent; 15%, CounterAgent; 20%, BasicFireAndMoveAgent; 20%, and SimpleGoLeftAgent; 45%. Essentially, there is a 50% chance that a new enemy will spawn at any given turn if there are less than 5 enemies in the current gamestate. If an enemy will spawn, then the enemy type being spawned will be chosen from the distribution listed above. This distribution was chosen to closely mimic the arcade game this model is based on and retains a challenging level of dynamic complexity to our algorithms. Once the spawning process is completed, then the next state can begin. This game loop continues until the player wins or loses. The player loses if they lose their life and the player wins if they survive the amount of turns the set at the start of the game (in the case of our model, 300 turns). Examples of this set up and game loop can be seen in the following files "RL/gameRL.py" and "GameLoopExpectimaxTestingV3.py"

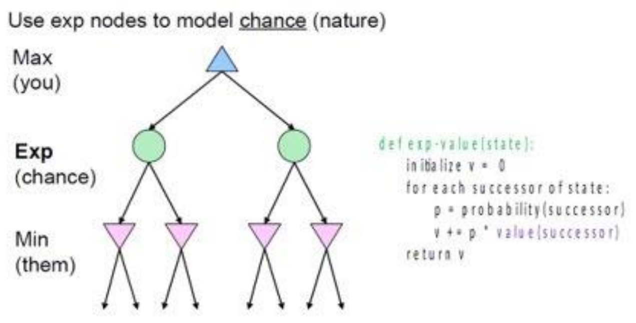
View

The game can optionally be viewed using our basic view. The GameState class has methods like print_board, print_agent_locations, print_projectile_locations, and print_score. These can be used to create basic visual representations of the game at any state. The print_board method for example will print a 2D array filled with numbers or strings that represent agent and bullet positions. Empty spaces are denoted by the value 0. The player agent is denoted by the number 1. Enemy agents will be represented by a value > 1. Player bullets are represented as X and enemy bullets are represented as B. If bullets or ships overlap, only one will be visually represented on the board. However, calling the print_agent_locations method and print_projectile_locations will provide all existing agents, bullets, agent types and positions regardless of overlap. We were building another view using tkinter and turtle, but unfortunately we ran out of time to complete it.

The Expectimax Algorithm

Structure

The expectimax algorithm can be visually represented as maximizer, and chance nodes. Maximizer nodes try to choose the best action from all potential actions based on some heurisitic. In our case, the heuristic is the score of a state. Chance nodes return the average score for taking a particular action and are calculated by taking the average score from all potential actions of non-player agents. The chance node scores depend on the max depth limit for predictions.



```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Figure 1 Expectimax Structure

The expectimax algorithm is utilized in the autoPickAction method of the ExpectimaxPlayerAgent3 class. There is an expectimax function which is called recursively. It is used to determine the player's score at any given turn, taking a specific action by predicting the expected value of that action x many turns/depths ahead. The depth limit of the prediction can be set upon instantiating the expectimax agent.

The expectimax function takes in 3 parameters; the agent index in the state's current_agent list, the current_depth (0 at the start) and the GameState that the next predicted state is based off. There are 3 terminating conditions for the recursive call. The first is if the depth limit has been reached. The second is if there are no more agents in the state. The Third is if the player agent has been eliminated. In all three of these cases, the recursive call ends and returns the score at that state. Otherwise, the recursive call continues. The appropriate next agent index and next depth and GameState is passed along to the next recursive call.

Average or best scores are chosen depending on whether the call is occuring at a maximizer node (player) or chance nodes (enemy). At the end, the player can see the expected value of taking a particular action, viewing x turns/depths ahead. The player will then choose the best action. Our specific implementation dictates that if more than one action has the same largest expected value, then the expectimax player agent will randomly choose among those best actions.

For our model, we decided to use a depth = 1. This means our agent will only look one turn ahead. Early testing showed that the expectimax algorithm can run quickly when there are only a few agents on the board (5 or less) and when those agents have a small amount of potential actions to choose from. The tree grows exponentially for every new agent and for every action that agent can take. Short games with few complex agents can run relatively quickly. However, even at depth=2, a 300 turn game takes longer than 2 hours to complete. With our limited time resources, we opted to use a depth=1. At this depth for our model, games may take around 10 minutes to complete at the longest. With this decision, we can grab more game samples for our analysis.

The Neural Network Model

To implement reinforcement learning (RL) for an AI agent we decided to utilize a standard 4-layer feed forward neural network with Deep Q-Learning and a Rectified Linear Unit (ReLU) activation function. This approach was chosen for two reasons: 1. Q-learning allows us to train the model by simply playing games rather than having to create a huge training data set with the correct output marked as with many traditional RL methods, and 2. Deep Q-learning allows us to leverage the power of deep neural networks to generalize across states which is necessary due to the huge state space of the Aegis Wing game.

Neural Networks

In our experiment we utilize a feed forward network structure. It has a single input layer of neurons, an intermediary set of hidden layers, and a final output layer. Each neuron in the previous layer is connected to each neuron in the next layer, with the weights of each connection being learned by the network. The neural network continuously feeds off the input it receives from its training data by running its outputs through a set loss function. This function is defined by the user to create an estimate of distance from the true "correct" answer to the network's output. This information is then fed back into the network through back-propagation, which updates the connection weights of each neuron. The network then continues updating weights in response to the training data, attempting to minimize the loss function. After this training period, one can use the new weights in the neural network to predict outputs to some degree of accuracy.

In a feed-forward NN information flows only in one direction, from the input layer, through the hidden layers, and finally out through the final output layer. Due to this structure, feed-forward NNs have no memory of previous inputs and are bad at predicting future states. Moreover, many traditional neural networks require large marked sets of training data that is very resource intensive to collect. To combat these issues and allow the network to train solely from playing games without exterior intervention we turned to Deep Q-learning.

Deep Q-learning

Deep Q-learning is reinforcement learning built from the Q-learning algorithm. Q-learning is a model-free reinforcement learning algorithm that attempts to learn the expected value of an action in a particular state. More specifically, it tries to learn the policy that maximizes the total reward. It does this by trying to learn the Q-function:

$$Q(s,\ a)\ =\ E_{\pi}[\ G_t\ |\ S_t\ =\ s,\ A_t\ =\ a\ ]$$

Where S represents a particular state, A the action taken at that state, and π the policy. This expectation over π is the expected value of the action, a, in state, s, following some set policy π. In traditional Q-learning, this is done by creating a table over every possible state and action at that state, then assigning a value at each state. Originally, all the action values will be assigned 0 and actions will be taken at random. The algorithms learns new Q values for each state-action pair as it performs moves by updating the Q-values using the Bellman equation which is formulated as follows:

$$Q^*s,a = Q(s,a) + \alpha[R(s,a)+Q(\ s',a') - Q(s,a)]$$

Where Q* represents the updated Q value for a given state s and action a, R(s, a) represents the reward for the state-action pair, α represents the learning rate, and represents the future discount value. As the agent takes actions it can update the Q-table and 'learn' the optimal action to take for any situation. This approach works fine when the state-space is limited to a relatively few states, but as the state-space increases, traditional Q-learning quickly becomes infeasible. Since representing every state can be impossible, a function approximation to generalize across states becomes required. This is typically done using a deep neural network because of their expressive power, giving rise to the name 'Deep Q-Learning'. Unlike a traditional Q function, the Deep Q-Learning Network takes in just the state as input and outputs a real valued vector of size |A| where A is the action space. Each spot in the vector represents the calculated Q-values for each action the input state, with the agent picking the action corresponding to the largest Q-value.

Implementation and Training

The first challenge of Deep Q-learning was defining a proper loss function to allow our neural network representation to work with q learning. We chose our loss to be expressed as follows:

$$\boldsymbol{loss} = \left(\boldsymbol{R}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{\gamma}\max_{\boldsymbol{a'}}\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a'}) - \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})\right)^{\boldsymbol{2}}$$

Here we defined our loss as the square of the difference between current rewards plus the discounted optimal Q value action for the state and the outputted Q value action. The reward + discounted optimal next-state Q value represents our expected value for this state action combination and as such we would like to minimize the mean square error between the actual and calculated expected value.

The next challenge to solve was transforming our state-space from a 2-dimensional grid representation into a one-dimensional feature vector. Our final result grid had 7 rows and 8 columns with 7 possible objects for each position, which gave us a final 392 x 1 vector. Each space in the vector corresponds to a single position on the grid and a single agent type. If the agent was present at a certain position then that position is marked with a 1, otherwise it remains 0. This way we are able to represent the location and composition of each space on the board.

To define the reward function, we decided on a few different heuristic measurements that we believed would help the agent make good choices. First, we summed the number of agents and projectiles in "danger" positions around the player. The danger positions were defined as the positions in which there was a possibility of overlap after both the player agent and enemy agent/projectile moves in the next turn. For each projectile in a danger position the reward given was -5 points. For each agent in a danger position, the reward given was -10 points. The agent was also negatively rewarded 100 points for losing a health point in the last state. The agent is also positively rewarded 0.5 points for each turn it has survived in the game, increasing incentives for that agent to survive longer. Finally, if the player agent has killed an enemy agent, then it is positively rewarded the point value that was assigned to that agent. This way the agent is incentivized to get rid of the more dangerous enemies as a priority, as they have higher assigned point values.

Our network was created using pytorch with 4 layers; a linear input layer, two linear hidden layers, and a final output layer. We defined the neuron size to be 200 in the first hidden layer, 100 in the second hidden layer, and 50 in the output layer, with a learning rate of 0.0001. The Q-network then trains performing random actions a certain percentage of the time defined by a parameter, ε. During training ε starts at a value of 1 (100% chance of random action) and decays at a value of 0.01 for each game played. For the first move in the current game during training, the agent always chooses a random action and updates its Q-values accordingly by calculating the loss and rewards from the new state. It then stores the state, action, reward, and new state in memory. This state is then replayed from memory a number of times (1000 in our case) and retrains the network weights using the calculated loss. After, it runs through each turn in the game choosing an action and updating the Q-Values. Every 100 iterations it stores a file of the updated weights for the training run and a graph of training scores, with our final model running for 5000 games. In testing we load our predefined weights and no longer update them, allowing our agent to independently run through games. We tested our agent's output at 100, 1000, 2000, 3000, 4000, and 5000 iterations with the results being shown in the following section.

Results and Discussion

Our final results compared data across 50 runs for the Expectimax agent with depth 1 and the Deep Q-learning agent at 100, 1000, 2000, 3000, 4000, and 5000 training games. A summarized table of results is shown below in Table 1 in the Appendix.

Training Results and Discussion:
After training our NN model over 5000 iterations we found that at a certain point the training ended up hindering model performance. We can see from Table 1 that at 100 training runs, the model wins no games, by 1000 the model begins winning games with increasing regularity until it hits 3000 iterations at which point it begins declining. At 4000 iterations the model performs significantly worse than it did at 1000 and at 5000 iterations the model can no longer win games. We can see this visualized in Figures 2 and 3. This phenomenon is fairly common in the world of machine learning and is known as overtraining/overregularization. While one might intuitively believe that more

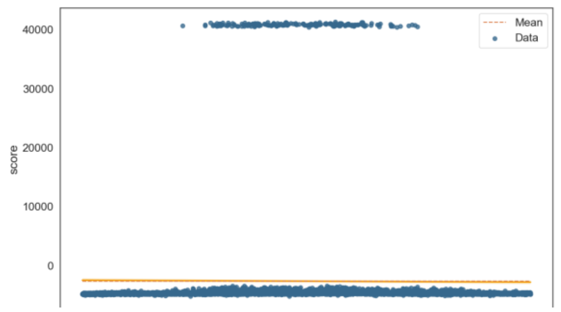


Figure 2 Neural Network Training Results at 2500 Iterations

training will always lead to better results, in fact the opposite can be true. The NN model is supposed to understand trends in the data (the state space) and react accordingly, however when the model trained too much it began trying to fit itself to the training environment instead and when it began being faced with dynamic environmental changes (even in training), the model begins to fail. Even though it was expected, it is still quite interesting to see this phenomenon unfold in real time.

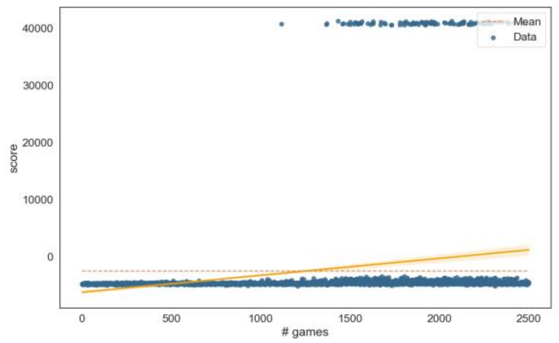


Figure 3 Neural Network Training Resuls at 5000 Iterations

Expectimax vs NN Results and Discussion

As expected, at optimal training times, the Neural Network heavily outperformed the Expectimax algorithm, as can be seen in Figure 4. While the superiority of the NN was to be expected, the fact that such a relatively simple NN could win over 50% of its games in such a complex state space was quite astounding. Considering that our 392-length feature vector still does not completely capture the complexity of the state space as it does not include overlaps and other potential heuristic features that could be used to further increase the model's "intuition" and the fact that hyper-parameter optimization was eschewed due to time constraints, the model was not expected to perform as well as

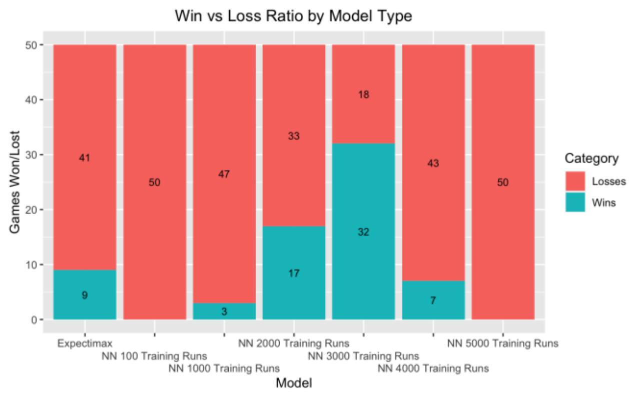


Figure 4 Wins/Losses in 50 Game Test Run by Model

it did. In fact, the only model which achieved an average positive score for the was the 3000 iteration NN model. While this is an extraordinary feat, it should be noted that this performance could also be due in part to general variance of the game state. s the game state is not fixed and random enemies spawn in semi-random locations some runs may end up being better suited to allow the model agent to win, while others have the chance of facing more challenging enemies like the Heuristic agent from the very beginning of the game. To truly dig into this, it would be ideal to gather more test run data, but due to the length of time that it took to run a game for expectimax our study was limited to comparisons for a maximum of 50 games.

Another major performance boost we discovered when comparing the two algorithms was the speed with which they compute the proper action to take. As seen in Table 1, the Expectimax Agent takes significantly longer to run a single game, even when the depth is limited to a single branch. The Expectimax Agent manages to run a single game at an average of 3.71 minutes per game, while the NN model ran all 50 runs within a single minute for each level of training recorded. While the Expectimax agent might fare better with some kind of pruning algorithm and with a larger depth value, this was not feasible to do with our time constraints. For this reason, we limit our experiment to depth =1 in order to limit the amount of time it takes to collect samples for our data. The drastic difference in speed between our expectimax and deep learning algorithm is due to the fact that the NN model offloads most of its costly calculations in its lengthy training step. While the NN model has to explore as many spaces as possible during training, once it is implemented it only needs to convert each game state to its feature implementation and perform relatively simple calculations to output the action Q-vector. On the other hand, at each turn/step of the game the Expectimax model has to expand all possible states of enemy actions, which for a dynamic game like Aegis Wing is a very large space even at a maximum depth of one.

One very surprising result of the Expectimax agent is how many enemies it killed when compared to the NN agent (Figure 5).

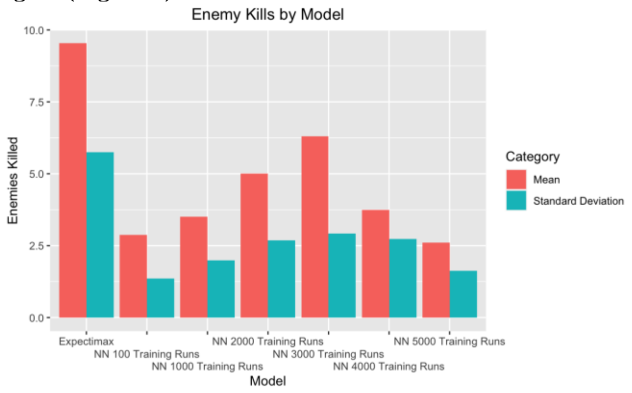


Figure 5 Enemies Killed in Test Runs by Model

Even at its best performance the NN only killed ~ 66% of the agents that the Expectimax agent killed in its run average. This is very likely due to the difference in strategies between the two types of agents. The Expectimax agent may value wiping enemies off the board more than the NN agent as it searches for the highest possible scores amongst subsequent game states. As surviving without getting hit yields the same score, regardless of the action. The only way to improve upon the game score when there is no immediate threat is to kill an agent, causing the Expectimax agent to try to kill agents instead of positioning itself better. On the other hand the NN agent has heuristic negative rewards that coerce it into attempting to stay further away from enemy agents and bullets along with a continuously increasing reward for each turn it survives. This heuristic 'teaches' the NN agent to move away from enemies as they attempt to get closer rather than to stay in place and fire on them as they move towards the player.

Furthermore, we also found that the NN agent, as is to be expected from RL, found a relatively efficient strategy utilizing the constraints we set for the enemy agents. It seems that the model was able to implicitly learn that enemy agents will not attack from the top and bottom and abused that constraint to survive as long as possible and stay away from enemy agents. We noticed that the 3000 training run NN tended to move around in the beginning but would ultimately move to the top left corner and begin firing a line of player bullets, making it virtually untouchable by enemy agents. Since enemy agents cannot fire in any direction except towards the left and no agents were programmed to crash into the player from vertical directions, there is virtually no way for an enemy agent to hit the player once the line of bullets is finished. The agent would have to spawn and immediately fire off a bullet in the same row as the NN player agent to do any damage to the player, which is very rare. The main challenge to this strategy comes in the form of the rarely spawned heuristic enemy which moves towards the player's location and fires from the new location in the same turn. This allows the enemy agent to charge into the line of bullets, killing itself in the process, but first shooting a bullet into the same row as the NN player agent which eventually travels and hits the player.

It was certainly fascinating that the NN agent was able to identify and exploit this game constraint, something that we as the designers did not realize when creating the game state. In order to get the NN model to behave more dynamically, we could introduce a penalty for staying in the same spot too long, increased the reward for killing enemies, limit the amount of times the player can fire, decrease the turn survival reward to a smaller fraction every turn, or just introduced another enemy that specifically attacks the player from the bottom or the top. It is quite extraordinary to see the NN model learning these underlying trends in the data, those that we as designers did not even realize at first, without them being explicitly encoded into the model design.

Expectimax Depth Differences

As a final look into our exploratory analysis of decision-making algorithms we wanted to dive deeper into the differences between different depths of the Expectimax tree search. To accomplish this in a feasible timeframe we limited all the agents to being Simple Go Left (SGL) agents and tested the agent at a depth of 1, 2, and 3 (Table 2). Since each enemy agent could only take a single action this drastically reduced the amount of time it took to calculate a full branch of the algorithm. We retained all other features for our model. We found that increasing the depth by 1 increased the average time to complete a game drastically increased the calculation time with the mean times for 50 games being 1.22, 6.12, and 55.10 seconds for depths of 1, 2, and 3, respectively. The increase in depth seemed to grow exponentially as the mean depth 2 time was ~ 5 times larger than the depth 1 time and the depth 3 time was ~9 times larger than the mean depth 2 time, leading to an overall ~45 times increase in elapsed game time between layers 1 and 3. This exponential growth makes sense as for each layer of depth, multiple new branches of possible state values will be needed for each branch in the previous layer. One unexpected result was how little the Expectimax agent outcome

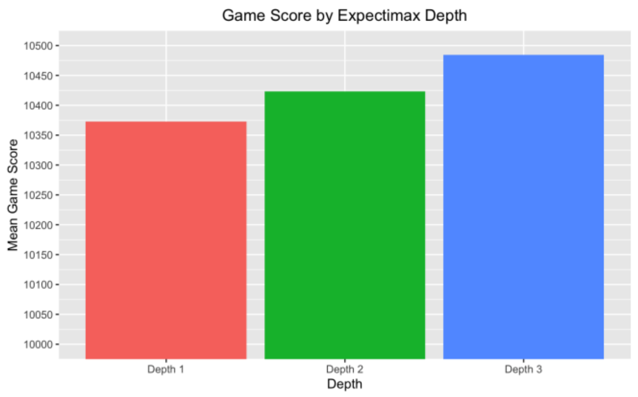


Figure 6 Mean Game Score for 50 Test Runs by Expectimax Depth

improved for the enormous amount of computation that was required. After removing the 10,000 given for winning the game (since each agent here won the game), the depth = 2 agent achieved an ~13.4% increase in score from the depth = 1 agent and the depth = 3 agent achieved an ~29.76% increase in score from the depth = 1 agent (~14.42% increase from the depth 2 agent). While a 30 percent increase in feasible score is a marked improvement from the depth 1 agent, it does very little to offset the 4500% increase in computational time required for a depth 3 agent. This is likely due to the fact that the only agents being tested against were SGL agents. These agents did not chase the player or try to shoot them, they just simply moved in a straight line, which requires very little thinking ahead. With a simple depth = 1 search, the Expectimax agent can see that an enemy agent will hit the player in their next move and either move into a space that will not be occupied or shoot the agent that will be moving into the player's position. In almost every case, as there are limited enemies on the board at any one time, the agent has no need to look further than one turn ahead as every situation can be solved in a single move. To truly see the benefits of a proper depth search we would have to change the enemy behavior to give the enemy agents more options of attack. In the case where enemies are coming from multiple directions or firing bullets at a player agent, we would expect to see drastic differences between the performance at different depths since the agent would have to "think ahead" in order to avoid being trapped in an inescapable situation.

Conclusion and Final Thoughts

As we expected the deep Q-learning NN model outperformed the expectimax algorithm during test runs of our game. Though it should be noted that the Q-learning model had to complete over 1000 training game iterations before it was able to outcompete the expectimax agent. While the NN model was extremely fast at on the fly computation, much of this cost is front loaded in the form of long resource intensive model training operation, but after completing enough training the NN model achieves spectacular results. To further improve upon our design we would like to implement more complex enemy agents to see how the neural network responds when there is no one easy winning strategy like sitting in the corner and shooting. This could be done by increasing the enemy HP or creating an agent that attacks from the vertical directions. Another next step would be improving the efficiency of the expectimax algorithm through pruning or some kind of state evaluation function so that we could test effectiveness at more depths without impossibly long run times. Lastly it would be interesting to implement more methods to see how they compare like Recurrent Neural Networks or a Monte Carlo Tree Search method.

Appendix

Model	**Mean Score**	**SD Score**	**Mean Turns**	**SD Turns**	**Mean Killed**	**SD Killed**	**% Won**	**Mean HP Loss on Win**	**Mean Run Time (Minutes)**
Expecti-max	-95.92	4974.69	170.80	92.63	9.54	5.74	18%	1.78	3.71
NN 100 Training Runs	-5736.68	62.51	27.32	18.97	2.88	1.35	0%	NA	0
NN 1000 Training Runs	-4743.00	3834.57	54.44	66.58	3.50	1.99	6%	1	0
NN 2000 Training Runs	-245.80	7682.79	124.86	127.81	5.00	2.68	34%	0.82	0
NN 3000 Training Runs	4564.24	7785.66	203.90	130.46	6.30	2.92	64%	0.94	0
NN 4000 Training Runs	-3448.14	5609.37	79.12	94.38	3.74	2.74	14%	1.43	0
NN 5000 Training Runs	-5705.08	70.83	42.62	25.04	2.60	1.63	0%	NA	0

Table 1: Expectimax vs Deep Q Learning Network 50 Game Test Results

Search Depth	Mean Score	SD Score	Mean Turns	Mean Ene-mies Killed	SD Killed	Games Won	Mean HP Loss on Win	Mean Run Time (Minutes)
Depth 1	10373	48.46	300	9.2	4.59	50	0.1	0.02
Depth 2	10423	54.93	300	14.96	5.46	50	0.18	0.10
Depth 3	10484	69.84	300	20.9	6.51	50	0.18	0.92

Table 2: Expectimax at Different Depths 50 Game Results with Only Go Left Agents

References

1. Berliner, H. J. Deep Thought Wins Fredkin Intermediate Prize. AI Magazine, 10(2), 89, 1989 https://doi.org/10.1609/aimag.v10i2.753
2. R. D. Greenblatt, D. E. Eastlake III, and S. D. Crocker, “The Greenblatt Chess Program,” Computer Chess Compendium. Springer New York, pp. 56–66, 1988. doi: 10.1007/978-1-4757-1968-0_7.
3. Samuel, A. L. “Some Studies in Machine Learning Using the Game of Checkers”. IBM Journal of Research and Development 3:3, 1959, pp. 210–229.
4. Samuel, A. L. “Some studies in machine learning using the game of checkers. II--Recent progress”. IBM Journal on Research and Development, 1967, pp. 601-617.
5. Shannon, C. E. XXII. Programming a computer for playing chess. In The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, Vol. 41, Issue 314, pp. 256–275, 1950, doi: 10.1080/14786445008521796
6. Schaeffer, J., Lake, R., Lu, P., & Bryant, M. “CHINOOK: The World Man-Machine Checkers Champion”. AI Mag., 17, 21-29. 1996,
7. G. Tesauro and T. J. Sejnowski, “A parallel network that learns to play backgammon,” *Artificial Intelligence*, vol. 39, no. 3, pp. 357–390, 1989.
8. G. Tesauro, “Temporal difference learning and TD-gammon,” *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.