Overview

My Masters Project was an attempt to train a bot to play Hearthstone with Deep Reinforcement Learning. It is inspired by similar projects such as Deep Learning for Atari, Alpha-Go, and TD-Gammon, and largely involves modifying the algorithms used in such project to accommodate differences Hearthstone has with these other games.

Hearthstone

Game Description

Hearthstone is a popular digital trading card game created by Blizzard Entertainment.

Each player starts the match with a deck of cards which each player constructed using 30 out of the hundreds of cards available in the match. Each card has a number of numerical elements (attack, health, mana cost), as well as text describing an effect they possess.

Throughout a match, each of a player's cards can be in one of a few locations: their deck, their hand, the field, or the graveyard. Each player has knowledge of their own hand, all the cards on the field (both theirs and their opponent's), the contents of both graveyards, and the contents of their own decks (but not their opponent's). They do not know the contents of their opponent's hand and deck, but they do know their size. This is the main source of imperfect information in the game.

In addition to the cards themselves and their location, a game-state is described by a number of other values: each player has a health value, a max and remaining mana value.

In Hearthstone, the players take turns. A player's turn starts with them drawing a card from their deck and adding it to their hand. They then can perform performing an arbitrary number of moves, mainly consisting of moving cards from their hand to the field or using cards on the field to attack

enemy characters. Which moves a player can take is limited by the cards available to them at that moment and by the amount of mana they have to spend that turn. The player ends their turn by hitting the “End Turn” button. A player cannot do anything on their opponent’s turn. The majority of these moves are deterministic, but several have stochastic outcomes due to randomness inherent to the game (shuffled decks) or random effects of many cards.

The game continues until the health of one player drops to 0. When that happens, they lose and their opponent wins.

Hearthstone Board

Hearthstone Cards

Hearthsim and the Hearthstone AI Competition

Hearthsim is a fan-made emulator for Hearthstone, specifically designed for making bots for Hearthstone. It features a forward model for the game.

The Hearthstone AI Competition was a competition held over in 2019 and 2018. It had contestant create bots utilizing Hearthsim.

This competition had two tracks. In the “Premade Deck” track, each contestant would submit their bot and would compete against all other bots in the track using decks published by the tournament runners beforehand. In the “User Created Deck” track, each player would submit a deck alongside their bot, and it would use that deck in competitions against the other bots in the track using their own decks.

Deep Q Learning

Reinforcement Learning

For this problem, an Agent agent is acting in an environment. Given a current “state” *s* and a set of actions *{a}*, the agent chooses one “action” *a* to take. By taking the action, it may observe a “reward” *r,* and the state changes to a “successor” state *s’*, from which the agent may have to act again using a new action set *{a}’*.

Reinforcement Learning consist of a set of algorithms used to train such agents. Usually, they do so by training the agent to take actions that maximize its expected accumulated reward while acting in the environment. In particular, most algorithms have the agent learn a “policy” function $\pi(s)$, in which a state is input and an action is output, such that the expected accumulated reward is maximized by following that policy. This function is then used by the agent to choose an action every time it is given a state.

Q Learning:

Q Learning is a common and popular reinforcement learning algorithm. It learns a policy function by learning the Q function, which is:

$$Q(s,a)=\sum_{s'} T(s,a,s')\left[R(s,a,s')+\lambda\, max_{a'} Q(s',a')\right]$$

where:

- *s* is a state
- *a* is an action taken from state *s*
- *s'* is the successor state: a state which may follow from taking action *a* from state *s*
- $T(s,a,s')$ is the probability that taking action *a* from state *s* results in successor state *s'*
- $R(s,a,s')$ is the average reward gained from taking action *a* from state *s* when *s'* is the successor
- λ is a "decay parameter", ranging from [0,1], which lowers the contribution of future reward to the overall Q value.

In plain English, this function calculates the average expected accumulated reward following taking action *a* from state *s*, and then acting optimally afterward.

The policy that uses this function is: $\pi(s)=argmax_a Q(s,a)$; or simply, when in state *s*, take the action with the highest Q value.

Q Learning is designed for situations where neither the transition probabilities $T(s,a,s')$ nor the reward function $R(s,a,s')$ are known, called *model-free learning*. Learning is done by viewing samples consisting of state, action, reward and successor tuples (*s,a,r,s')* that are gathered from simulations.

Deep Q-Learning: Training

Some specifics of the training depend on how the Q function is modeled. In Deep Q learning, a neural network is trained to predict the reward function. This Deep Q Network is trained in pretty much the same way any neural network is trained: gradient descent via back-propagation.

Given a data point consisting on an input *x* and a corresponding true target value *y,* the prediction of the network for that data point $\hat{y}$, and a loss function $L(y,\hat{y})$ which quantifies the error between the two, back-propagation updates the weights in a neural network toward minimizing this loss function. It does so by calculating the gradient of the loss function with respect to each weight in the network, then subtracting each weight by some portion of its gradient.

In the case of Deep Q learning, the data point is a state and the output is predicted *Q* values for the actions. One aspect that separates Deep Q-Learning from most supervised applications of neural networks is that the target is not fixed, but is also calculated using the network. This target is given as:

$$y = r + \lambda \, max_{a'} Q^N(s', a')$$

where Q^N is the Q-Network, and *r* is the reward observed by taking action *a* from state *s* and resulting in successor state *s'*. Here, when updating the network, the target calculated in part using the current version of the network.

So to train this network, data consisting of state, action, reward and successor tuples (*s,a,r,s')* is gathered from simulations. During these simulations, the true policy isn't used; rather, a policy which balances *exploitation* of the best currently learned valued *exploration* of unknown ones us used. A common implementation, which I use, is *epsilon greedy*: where when making a decision, there is an *epsilon* percent chance that a random action is taken, and a 1- *epsilon* chance of choosing the best move according to the current Q function. This epsilon parameter starts high and decays as training continues.

For each data point (or more typically, samples), prediction $\hat{y}$ is calculated with the Q-network, and target *y* is calculated with the above equation. These two values are put through some loss function, like Mean Squared Error for instance:

$$Loss_{MSE}(y,\hat{y})=\frac{1}{2}(y-\hat{y})^2=\frac{1}{2}(r+\lambda\, max_{a'}Q^N(s',a')-Q^N(s,a))^2$$

Then, the gradients of the loss with respect to each weight are calculated:

$$\nabla_w Loss_{MSE}(y,\hat{y})=(y-\hat{y})\nabla_w\hat{y}=(r+\lambda\, max_{a'}Q^N(s',a')-Q^N(s,a))\nabla_w Q^N(s,a)$$

And each weight is updates via gradient descent:

$$w\leftarrow w-\alpha\nabla_w Loss(y,\hat{y})$$

Deep Q-Learning: Improvements

Over time, many improvements for Deep Q-Learning have been developed, some of which I implemented in my project.

Experience Replay:

When training a neural network, it is important that the data used be independently and identically distributed. In a reinforcement learning problem, since states are not independent from their predecessors, this assumption does not hold.

To rectify this, generated data is held in a buffer. As new data is generated, older memory is removed from the buffer. Whenever an update to the network is made, tuples are sampled randomly from that buffer. This helps break up the temporal dependencies inherent to reinforcement learning data.

In my application, I used a relational database to store these tuples, so the training could be paused and resumed without the need to create new data from scratch.

Target Network:

When the same version of the network is used to calculate both targets and predictions, this has been found to create unstable training. To rectify this, two different versions of the network are used. The *online network* Q_O^N is used to make the predictions $\hat{y}$, and is what is updated during each gradient descent step. The *target network* Q_T^N is used in the calculations of the target *y*. It is not updated during the gradient descent step; rather, at regular intervals during training, the weights of the *online network* are copied to the *target network*. As such, these networks need to have the exact same structure.

Updating the above loss function, we have:

$$Loss_{MSE}(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2 = \frac{1}{2}(r + \lambda\, max_{a'} Q_T^N(s',a') - Q_O^N(s,a))^2$$

Double Q-Network:

Using target network itself has been found to result in the Q-network overestimating the Q score. To rectify this, the calculation of the target value is changed to:

$$y = r + \lambda\, Q_T^N(s', argmax_{a'} Q_O^N(s',a'))$$

In plain English, the Q component of the target uses the action that the *online network* says is the best, but the score that the *target network* gives that action is used. This was found to correct for the overestimation of the Q value.

Updating the above loss function, we have:

$$Loss_{MSE}(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2 = \frac{1}{2}(r + \lambda Q_T^N(s', argmax_{a'} Q_O^N(s',a')) - Q_O^N(s,a))^2$$

Training and Testing in Parallel:

The running of simulations for data generation and the training of the network is able to be done in parallel. Simply create one thread for training the network, and several separate threads for playing games to generate data.

Doing this requires a thread-safe replay buffer (which a Relational Database provides) and a thread-safe neural network that the threads share. The game-playing threads push data to the database as it is created, and the training thread samples from it whenever it needs data to make an update to the network. A mutex for the neural network is given to any thread that needs it: when a game-playing thread for making Q calculations for playing games, and when the training thread needs it to calculate the target values from the data or perform an update on the network.

Applying Reinforcement Learning to Hearthstone

State, Action. Reward, Successor:

In most applications of using Reinforcement Learning toward games that I have seen, it is used either to make a decision every frame (like with Deep Learning Atari) or to make a one move per turn (like with Alpha-Go or TD-Gammon). In Hearthstone, however, players take an arbitrary number of moves during their turn.

Since the vast majority of these moves are either deterministic or lightly stochastic, and a forward model for the game exists, it appeared unnecessary to require an agent to learn to choose among individual moves. What could not be explicitly modeled efficiently, however, was what

happened after the end of a player's turn, since this was the result of an opponent's decisions based in part on unknown information. Therefore, my main decision was not to have an agent learn the value of individual moves, but to learn the value of the collection of moves taken in a turn. In reinforcement learning terminology:

- the start of the player's turn is the "state"
- the sequence of moves taken during the turn is the "action"
- the start of the player's following turn is the "successor"
- any reward gained between the state and successor (reward gained by making the moves plus reward gained due to the opponent's actions on their following turn) the "reward".

This introduces another difficulty, however. Most applications of deep reinforcement learning have a small, constant action space. The network they would use to score actions would input the state and output one score for each of those actions. Since the possible action-state combinations are way too numerous in total, and not constant over the states, that solution is not feasible for this application. Instead, I have my score network input both a state and an action, and output a single score value. When making a decision, the agent would put each of the possible actions into the network individually, and choose an action based on the scores.

Reward Function:

In order for Reinforcement Learning to work, a way to evaluate the immediate reward of a state and action is required.

Hearthstone is a zero sum game. There is only one winner and one loser, and an overwhelming victory is just as much a victory as a close game. Therefore, my initial thought was to only grant a positive reward to actions that result in a win, a negative reward to actions that result in a loss, and no

reward otherwise. This was the approach taken by TD-Gammon. However, I believed that in order to get the agent to act more intelligently in every turn, a way to get a non-zero reward to measure the utility of each turn would be needed. Since there is no explicit reward measure in the Hearthstone (like a score in many Atari games), I had to design one myself.

Hearthstone is often viewed by its players as a kind of recourse management game. Cards in your hand and on you field are all tools available to you at that moment. When a card is added to your hand or the field, that is a resource which is “gained”. When a card is sent to the graveyard, it is a recourse that has been “spend” and is no longer available. If you have more cards available to you thank your opponent does, then you are said to have “card advantage”. Having more “card advantage” generally means that you have more options than your opponent, and thus are more likely to win in the long run. Therefore, good moves are ones that cause your opponent to lose more resources that you expend, or cause you to gain more resources than your opponent.

In addition to cards, a player’s health and mana can also be considered a resource. It is best to limit the amount of health you lose while maximizing the health lost by your opponent. Also, during each turn, a player is given an amount of mana which is spent to play cards. If the player doesn’t use all that mana, then it is “lost”. Therefore, is it more resource efficient to spend as much mana you can during your turn, and so not doing so should be penalized.

While winning and losing is ultimately the only thing that matters in Hearthstone, using your resources efficiently over the course of the game is often the best way to ensure victory in the long run.

So, I designed my reward function to encourages having more resources than your opponent. Features of this function include: (1) the number of minions which each player has on the field, (2) the number of cards each player has in their hand, (3) the number of cards each player has in their decks, (4) each player’s remaining health, (5) the turn player’s remaining mana, (6) the total attack value of the minions on each player’s field, (7) the total health value of the minions on each player’s field. What

I call the *resource component* $\phi(\text{game state})$ of a given game-state is a linear combination of these features using hand-crafted weights.

However, this alone might make the reward accumulated by a long game better than a short one, since more turns with abundant resources could result in greater accumulated reward than a quick game with a win. Since I would prefer that the agent prioritize quick victories over long games, I add a *turn penalty T* to every turn, which penalizes taking too long to win, counterbalancing the accumulation of

The complete reward function, then, is a large score if this is the last action before a victory, a large negative score if this is the last action before a defeat, and otherwise the *resource component* at the end of the player's turn, minus the *resource component* at the start of the following turn, minus the *turn penalty*. Written as an equation:

$$R(s,a,s') = \phi(\text{end player's turn}) - \phi(\text{start following turn}) - T$$

State Representation:

Due to the sheer number of cards available in the game, the number of possible game-states is huge. Therefore, a way to encode a game-state is needed. The design of the encoding is informed by its use as a neural network input and as a tree search equality comparison (discussed later).

A game-state in Hearthstone is largely defined by the features of cards and their location, plus some additional information about the game board.

Values related to each individual card include: (1) an integer base Attack value, (2) an integer current Attack value, (3) an integer base Health value, (4) an integer current Health value, (5) an integer base Mana Cost, (6) an integer current Mana Cost (7), a categorical Type (Minion, Spell, Weapon, Hero), (8) a categorical Tribe (e.g. None, Beast, Dragon, …), (9) Effect Text. Each card is represented by a vector of these values; single values for the integers, one-hot encodings for the categorical values, and a series of Boolean values for the existence of certain, hand-picked words and phrases in the Effect

Text. Certain other values, such as name and card rarity, are excluded due to their irrelevance to the game.

Values related to the board include: (1) an integer for each player's Health, (2) an integer for each player's Hand Size, (3) an integer for each player's Deck Size, (4) an integer for each player's Field Size, (5) an integer for each player's Base Mana, (6) an integer for each player's Remaining Mana, (7) an on or off Hero Power, (8) the stats of each player's Equipped Weapon. In addition, I add (9) the sums for the total Attack value of the minions under each player's control, (10) the sums for the total Health value of minions under each player's control. This is also represented as a vector of these values: single values for the integers, and a Boolean for the Hero Power.

For the sake of clarity and convenience as a neural network input, the entire state representation is divided into three groups: *Hand Information, Field Minion Information, Board History*.

Hand Information consists of all the cards in your hand. A player can have between 0 and 10 cards in their hand, so this is represented as the concatenation of these card vectors, plus 0 vectors to represent unfilled space.

Field Minion Information consists of the cards on the field. Each player can have between 0 and 7 on the field, so this is represented so this is represented as the concatenation of these card vectors, plus 0 vectors to represent unfilled space.

Board History is meant to represent the current board values, plus their recent history. It consists of the vector of the above board values, plus the board values that existed at the end of the last few turns. While this section is mostly constant within a turn, the history may be helpful for the neural network to evaluate state.

Information hidden in the game, which includes the order of cards in your deck, the cards in your opponent's hand, and the order and contents of cards in your deck, are unknown and largely unchanging within a single turn. Thus, they are ignored when constructing state information.

Both the minions on the field and the cards in the hand have an ordering. However, in the vast majority of cases, the ordering of cards is irrelevant. So, in order to make two game-states with only different card orderings equivalent, both for equality comparisons and for neural network input, the card vectors in the *Hand Information* and the *Field Minion Information* sections are sorted.

Identifying Actions with Search Trees

As mentioned previously, an "action" in this application is a sequence of moves taken during a turn. To find all of these sequences, search trees are used. Given a game-state, Hearthsim's forward model is used to find the outcome of each move available from that state. Using A-Star, a tree is constructed using where game-states are nodes and moves. After this tree is constructed, each node is scored according to the Q function, and the sequence of edges that lead from the root to the node with the highest Q score is selected. However, if a sequence of moves that result in a win are found (or in Hearthstone terminology, if the player "has lethal"), then scoring each value is skipped, and the path to the winning node is returned instead.

This solution also solves another issue. In order the update the Q value, the max Q value for the successors is needed. Since the action space is not constant per turn, a way to find the actions available in the successor state is necessary. This is solved by simply storing the tree in the data tuples; *(s,a,r,s')* becomes *(s,a,r,s',{a}')*, where *{a}'* is all the actions that were found for *s'*.

However, using search trees introduces some issues that need to be solved.

One issue is time constraints. I was restricting myself to the time limit imposed by the competition (70 seconds per turn), and sometimes a tree could not be completed in that time. So, sometimes I would have to make due with a partial tree, and this tree would have to prioritize expansions in a way that the best moves would be discovered. This was the primary motivation for using A-Star. Whenever a new node is discovered, it is placed in a priority queue. This priority queue was ordered using the *resource component* $\phi(\text{game state})$ of the reward function, since I believe that

states with high scores are likely to have successors who also have high *resource components*, which would mean they are likely to have higher Q values.

Another difficulty is the existence of stochastic move outcomes. The majority of moves have deterministic outcomes: only one result. But many actions (such as playing cards with randomness in their effect, and drawing cards) have many possible outcomes. Furthermore, the amount of stochasticity in these moves vary wildly; some only have a number of outcomes in the single digits, while others can have thousands.

My solution for this changed over the course of my project. In my most recent solution, the successor of an action with a stochastic outcome would simply be one successor randomly sampled from the possible outcomes.

The search tree algorithm works like this:

MoveSearch(Root, Start, T): Returns Queue of moves

- Given a root game-state Root, a start turn game-state Start, and a time limit T,
- Run A-Star with Root as the root until either Lethal is found, time runs out, or all nodes are expanded.
- If Lethal was found, return the path from Root to Lethal.
- Otherwise, use the Q function *Q(Start, N)* to for every node N in the tree.
- Return the path from the Root to the node with the highest score.

One thing to note is that I differentiation between the Root and the Start. The Start is the state from which the turn started, and is thus the value to use as the state *s* in the Q calculation. Root is the value to start the tree from. These two values are often the same, but there are some circumstances where the Root will be different than the Start, such as when the agent takes a stochastic action that results in an outcome that wasn't previously simulated.

Reward Decomposition:

As stated earlier, the reward of a given state-action pair is the *resource component* at the end of the player's turn, minus the *resource component* at the start of the following turn, minus the *turn penalty*. The first portion is dependent only on the end of the turn after taking the action, the second portion is dependent on the start of the player's following turn, and the final portion is constant. Since the agent sees the end of the turn after taking its action due to the tree search, and the *turn penalty* is constant, the agent already knows a portion of the total reward function.

Given this, it felt redundant to require the agent to learn that portion through reinforcement learning. The agent could start with this knowledge already, and simply learn the second portion of the reward (the *resource component* at the start of the following turn) as well as the future reward. I call the former portion of the reward function the *Deterministic Reward* R^{det} and the later portion the *Stochastic Reward* R^{sto}, such that $R(s,a,s')=R^{det}(s,a)+R^{sto}(s,a,s')$. Note that R^{det} only depends on the state and action, things which would be known by the agent when choosing an action, while R^{sto} is based on the successor as well, something not known to the agent when selecting an action.

Rewriting the Q equation with these terms, we get:

$$Q(s,a)=\sum_{s'} T(s,a,s')\left[R^{det}(s,a)+R^{sto}(s,a,s')+\lambda\, max_{a'}Q(s',a')\right]$$

$$=R^{det}(s,a)+\sum_{s'} T(s,a,s')\left[R^{sto}(s,a,s')+\lambda\, max_{a'}Q(s',a')\right]$$

Since a portion of the Q value is known given the state and action, we can have the network learn only the unknown portion instead of having the network learn the entire Q function. Terming this Network N, we get:

$$Q(s,a)=R^{det}(s,a)+N(s,a)$$

$$N(s,a)=\sum_{s'} T(s,a,s')\left[R^{sto}(s,a,s')+\lambda\, max_{a'}\left(R^{det}(s',a')+N(s',a')\right)\right]$$

Making the new policy function:

$$\pi(s) = argmax_a Q(s,a) = argmax_a \left(R^{det}(s,a) + N(s,a) \right)$$

Some modifications to the training algorithm are made to have the network predict the new value. First, instead of storing the entire observed reward value *r* in the *(s,a,r,s')* tuples, we store only the portion corresponding to R^{sto}, named r^{sto}. So, the tuples are not *(s,a,*r^{sto}*,s')*.

The target value *y* must also be transformed from the whole Q value to only the portion that the network is meant to predict. This is, translating $y = r + \lambda Q_T^N(s', argmax_{a'} Q_O^N(s',a'))$ to:

$$y = r^{sto} + \lambda \left[\phi(s',a') + N_T(s',a') \right] \quad \text{where} \quad a' = argmax_{a'} \left(\phi(s',a') + N_O(s',a') \right)$$

Notice that $\phi(s',a') + N(s',a')$ is equivalent to $Q^N(s',a')$, and also that the previously mentioned *online* and *target* networks are still used, but for the new network type.

Incorporating all this into a loss function for network training, we have:

$$Loss_{MSE}(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2 = \frac{1}{2}(r^{sto} + \lambda \left[\phi(s',a') + N_T(s',a') \right] - N_O(s,a))^2$$

$$\text{where} \quad a' = argmax_{a'} \left(\phi(s',a') + N_O(s',a') \right)$$

<u>*Neural Network*</u>

The network being trained is part of the Q calculation. As previously mentioned this network inputs a state *s* and action *a*, and outputs a single score value. It is meant to output an estimate for

$N(s,a) = \sum_{s'} T(s,a,s') \left[R^{sto}(s,a,s') + \lambda \, max_{a'} \left(R^{det}(s',a') + N(s',a') \right) \right]$, from which Q is calculated by $Q(s,a) = R^{det}(s,a) + N(s,a)$. I implemented this network with "Tensorflow.Net", a library of mappings for write Tensorflow in C#.

When deciding what to input as *s* and *a*, I make one major presumption: The game-state that the player ended their turn on is ultimately the only thing that matters when evaluating the score. The

particulars of the game-start at the start of the turn, and the particular sequence of moves that got you to that end-of-turn game-state, offer no information that the end-of-turn game-state itself doesn't. Therefore, the neural network input for a state and action is the representation of the game-state at the end of the turn.

As previously mentioned, a game-state representation is split into three sections: *Hand Information, Field Minion Information, Board History*. Each of these sections is put as input into different parts of the network.

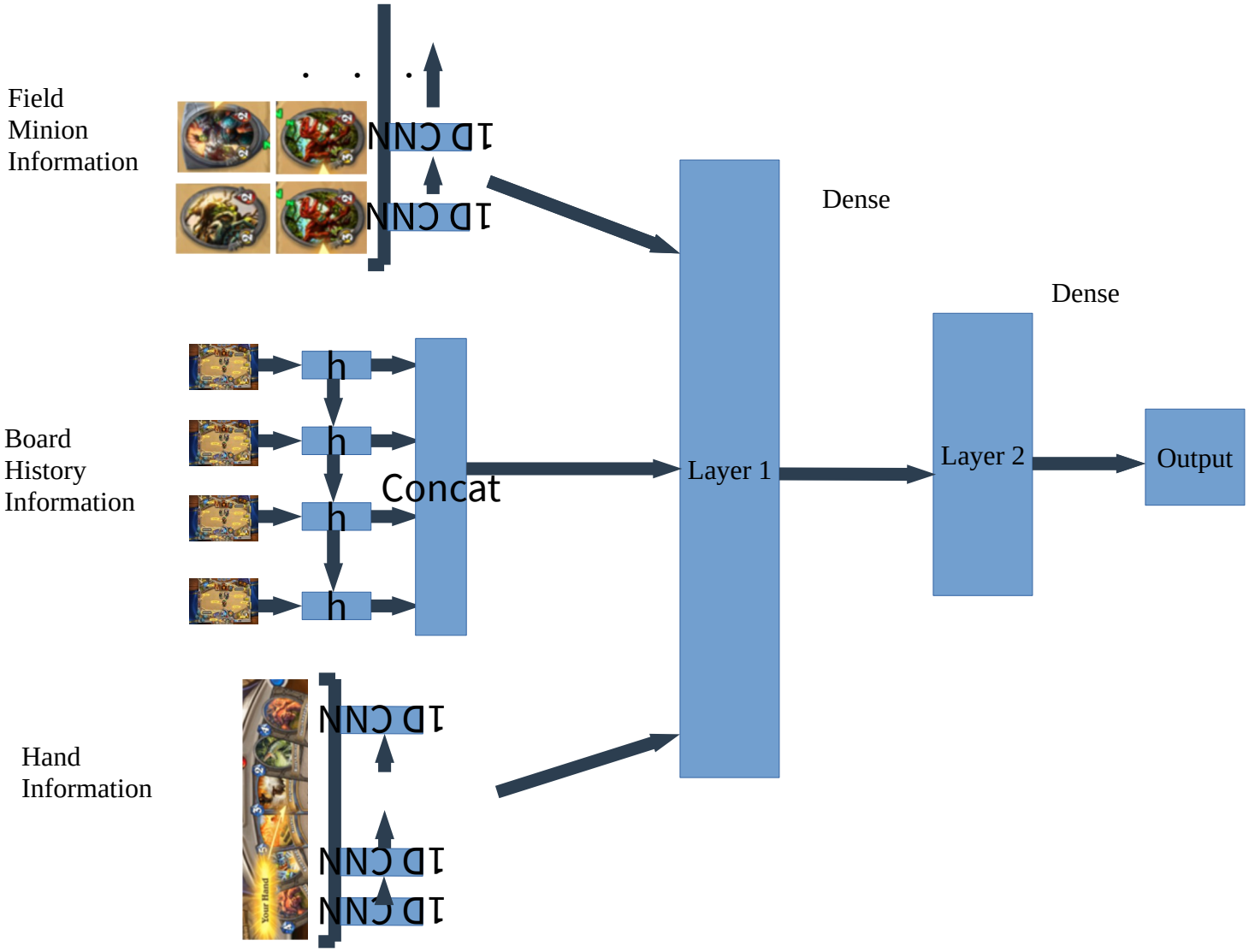
Hand Information consist of a matrix of card vectors; each row represents an individual card and each column represents a particular feature. To analyze these, I put this matrix as an input into a 1D convolutional layer which convolves over the rows. If I simply concatenating the vectors and putting it though a standard dense neural network, the number of weights would be extremely large. By using the convolutional layer instead, each card is analyzed the same way and the number of weights is reduced significantly.

Field Minion Information consists of two matrices: where rows are minions and columns are features of those minions. One matrix contains the data for the player's minions, and the other contains the data for the opponent's minions. To input this into the network, I first create a new matrix using the pairwise matching of rows from the player's and the opponent's minion matrices. Since each original matrix contains 7 rows (for up to 7 minions) and *d* columns (for *d* minion features), the result is a (49)x(2d) matrix. This matrix should capture the pairwise interaction data between minions, which may be useful given that the player's and opponent's minions are likely to interact with each other though battle. This matrix is then put through its own 1D convolutional layer that convolves over the rows. The reasoning for this is much of the same as with the *Hand Information* network: saving weights and ensuring each pair is analyzed in the same way.

Board History consists of a matrix of board vectors, each representing the board information at the end of the last few turns. To model the temporal dependencies among these states, I put the these

board vectors through a recurrent neural network. In only put board information though an RNN, rather than all the card information as well, in order to make the whole network as small as possible while still capturing temporal data about major features of the game-state.

Each of these three components output a vector of features. These features are then concatenated into a sing vector and used as an input into a two layer dense neural network. The last layer of this network has a single node, which is used as the prediction output for the network.



Agent Decision Algorithm

Finally, here we put it all together. This is a description of the agent's move choice algorithm.

The agent always keeps track of a few things. It keeps a MoveQueue: a queue of moves which represents moves it plans to do in the future. It keeps track of a StartState: the most recently seen game-state which represents the start of a turn. It keeps track of EndState, which is the most recent game-state from which an "end turn" action is taken. Finally, it keeps track of a Timer: the time remaining in the player's turn.

When the agent observes a new state S: it first checks if the state represents a "start of turn" state. If it is, a new data tuple *(s,a,r,s',{a}')* is partially constructed as *(StartState,EndState,ϕ(S),S,_)*. This states that a transitions from the old StartState using the action sequence that lead to EndState, resulting in the stochastic portion of the *resource component ϕ(S)*, resulted in the successor state S. Then, is sets StartState to S, and reset the Timer.

Then, it checks if MoveQueue isn't empty, and that front move in MoveQueue is a valid move from S.

If so, that move is popped from the queue, and that move is taken.

Otherwise (which would be the case if S is the start of the turn, or a stochastic move was taken which resulted in a non-simulated state), it creates a new SearchTree with *MoveSearch(S, StartTurn, Timer*0.66)* to create a new MoveQueue. If S is StartState, then this tree is set to the created SearchTree. The front of the new MoveQueue is checked again.

If S was a "start of turn" state, then the partially constructed data tuple is completed by adding the recently created SearchTree to it. This tuple is then added to the Experience Replay Buffer.

Project Overview

In this section, I describe the general progression of my project, offer my reasons for certain decisions, and my thoughts in hindsight.

The overall progression of my project went:

Fall 2019 Semester: Research

January: Implementing SearchTree

Early February: Coding State Representation

Late February: Created initial reward function

Early March: Designing and Coding State Representation, changed reward function implementation to use that representation

Late March: Started making Nerual Network and training algorithm

Early April: Updated SearchTree algorithm, updated training algorithm for Parallelism

Mid April: Bulk of training

May: Writing Final Paper

Choosing the Project

As an undergraduate student, I was a dual major in Computer Science and Games & Simulations Arts & Sciences. Despite approaching graduation, I still felt there was more I had to learn. Since I found I was especially interested in AI, I decided to continue my education by pursuing a masters degree in Computer Science focused on Machine Learning.

To be accepted into the program, I had to choose a masters project, and one that the those who would have to approve my application would see as one I could do. An obvious solution would be to do a project applying machine learning to video game playing, as it combines both of the fields I study. I ultimately chose RL for Hearthstone because it would give me significant experience in Reinforcement Learning: an important field that appeared to me to be under-taught at RPI, and Hearthstone is a game I am very familiar with that I have yet to see RL applied to. I saw it as a significant challenge due to the complexity of the game compared to may others RL has been applied to.

Researching

While technically, my project was only supposed to be done over the Spring 2020 semester, I wanted to get all my research for the project throughout much of the Fall 2019 semester. I read a number of research papers regarding RL projects similar to this one, viewed many RL tutorials and articles.

The main results of my research was the development of theory for the aforementioned formulation of RL to Hearthstone, the tree-search algorithm for action finding, the details of the Deep Q-Learning algorithm and many of its improvements and the neural network architecture.

Despite my efforts from the start, these features did change throughout the project due to difficulties discovered along the way as well as time constraints.

Choice of RL Algorithm

Near the start of the project, when I was explaining my use of Deep Q Learning to you, you strongly recommended that I consider other algorithms: Policy Gradient methods in particular. You said this was because Policy Gradient has been shown to have many advantages over Q Learning.

Policy Gradient methods learn the policy directly. A Policy Gradient network output a probability distribution over the action space, and the policy function output is a sample from that distribution.

Despite these warnings, I still chose Deep Q Learning. I believed it was more fit for this project for two main reasons:

1. Policy gradient networks output a probability distribution over the actions. This is usually done using a softmax activation function over a constant output dimension. Since the action space was not anywhere near constant over the state space, having the network return a probability output was infeasible as far as I could tell.
2. Most policy-gradient methods are on-policy, in which each update to the function makes data previously generated to train it useless for later updates. Since I feared Hearthstone is too expensive to simulate, I believed an off-policy method, like Q Learning, was necessary due to it's data efficiency.

Implementing Tree Search

At the start of the Spring 2020 semester, the search tree algorithm was the first thing I started implementing, due to it being necessary to run all other parts of the agent. However, the version I started with was a bit of a different version that the one I ended with. In particular, it handled stochastic move results differently.

In the original version, if a stochastic move were discovered, expanding that move would be paused until later. After all move paths with solely deterministic outcomes were sampled, only then would the move with stochastic outcomes be evaluated. While time permitted, stochastic moves would be selected and several results would be sampled, each of which would be the root of a new

SearchTree. These trees would be expanded in much of the same way any other would, except stochastic outcomes in these subtrees would only have one outcome sampled. These subtrees would be scored using the max Q value of any of its nodes, and the score of the stochastic nodes in the main tree would be the average of the scores of its subtrees.

Some additional features this approach had were also helpful. If the agent ended up taking a stochastic move that resulted in a state that was already simulated, time could be saved because part of that move's decision tree was partially made already. It also dealt with the action maximization steps very elegantly. The values that would be maximized over would be the calculated Q values of the deterministic actions and the averages calculated for each of the stochastic actions in the root tree.

While I was very happy with this solution in theory, I would find it had an issue in practice. Namely, the runtime and the search heuristic.

Even when sampling only three or four outcomes, the time it would take to evaluate several stochastic nodes was simply too high: often only a small portion of all the stochastic nodes would be analyzed. The first solution I tried was to limit the time spend expanding deterministic moves. However, it was difficult to find a good way to balance the time between deterministic and stochastic expansions that gave a good portion for both. My second solution is the one I went with and I described above. If I wanted to create an evaluation for more stochastic nodes, I would need to sample even fewer for each: just one. Unfortunately, identifying and making this fix took more time than I would have liked, and arguably made much of the work I had done on the first version of the tree obsolete.

As for a search heuristic, I believe the one I chose was a good one: using the *resource component* of the game-state. Since this component is a portion of the total Q value, it makes sense that a game-state having a higher *resource component* means it is likely to also have a higher Q value, thus making it a reasonable heuristic. However, one place it falls short is exploration variety. Many non-equal states are still very similar to each-other and thus have similar *resource components* and Q

values. Thus, the search may prioritize looking at moves with similar outcomes at the expense of exploring different moves that may be discovered to also have good outcomes. I had the idea to rectify this by including some measure of state similarity in the search heuristic to penalize searching move with results similar to others that have already been explored. However, due to time constraints, I did not think of a way to formulate and implement this feature.

Designing and Implementing Reward Function:

I got around to starting this segment around late February. It proved to be a more difficult section; not from

As I mentioned int the *Reward Function* section, Hearthstone does not have an explicit way to reward individual states. Since I wanted one to improve the reward function, I had to design one myself.

I believe I identified important features with which to calculate this score, but I struggled somewhat with how to weigh each feature. I have a general idea of how to order the features in importance, but not their relative scale.

One idea I had to choose these features was to use an evolutionary algorithm. However, I feared that this would detract from the limited time I had to train my agent. The main way I ended up tuning these weights was to observe my agent playing games, identifying what I believe to be mistakes, and hand-tuning the weights to make those mistakes less likely. This was time consuming in its own right, and arguably imperfect and arbitrary, but I believe serviceable, and I hoped the reinforcement learning portion of the reward would pick up where the immediate reward fell short.

Another choice I made when designing the reward function was whether to reward simply possessing resources or rewarding a change to them. Initially, I thought the latter was the best

approach: to reward gaining and penalize loosing your own resources while reward causing the opponent to lose and penalize causing the opponent to gain resources. This would be calculated by taking the features of the end-of-turn game state and subtracting them by the features of the start-of-turn game-state, then multiplying that new vector by the weight vector.

In hindsight, there are a few things which are important to using resources efficiently that were not captured in the features I selected, most of which has to do with analyzing card text. After thinking about is some more, I may have some ideas how to implement them, but I feared running out of time.

Dropped Feature: NLP

As I mentioned previously, each card has a short sentence which describes what they do. To include this in the analysis, I was planning to do a Natural Language Processing component. A large part of how a card should be analyzed is based on this text. Since the length each piece of text was very short, and the set of words used to write this text was very small, I suspected this may be feasible.

Basically, when inputting a game state into the neural network. Each sentence, as a series of one-hot-vectors representing words, would be put through an RNN, resulting in a vector encoding. This network would be shared for all cards input into the network. Each card's vector would have their corresponding text encoding concatenated to it before being input into the network I described above.

The main reason this feature was dropped was due to time constraints. I found I was in early March when I can around to the time I would be trying to implement it. So, I instead opted for a set of Boolean values each representing the presence or absence of certain words and phrases common to Hearthstone card text.

Implementing Training & Testing

I got to this section around mid march. In order to implement the experience replay buffer so that I could pause training without losing all the previously generated data, I had to do some additional learning on how to use Relational Databases in C#. It didn't take too long, and was fairly simple to implement the pushing and sampling function necessary for the training algorithm.

Initially, I wasn't going to make training and testing parallel. However, after viewing how long playing games took, I felt it was necessary. So, I took the time to re-implement the training algorithm to work in parallel. This was the main cause of additional time

Training and Testing

For training, I went with the strategy used by similar projects: data would be gathered by running simulations of the bot playing against itself using random decks. While the simulations were being run and data is being gathered, the network would be updated in a parallel thread. As training continued, the epsilon parameter would decay, and old data points would be removed from the buffer. At regular intervals, the *online network* would be copied to the *target network*, and training would be paused as the bot would be bench-marked.

My main method of bench-marking my agent was though having it play matches against other bots. In particular, I chose the top 4 bots from the Hearthstone AI Competition's Premade Deck Player Track from 2019 as my bot's opponents. According to each of their descriptions, the agents in question used either Monte Carlo Tree Search, Dynamic Lookahead, or Greedy Lookahead. I had both my agent and the enemy agents use the decks made for this deck track.

From the very beginning, my agent had a roughly 50% win-rate against the enemy agents. This can almost certainly be attributed to the Search Tree algorithm and the Decomposition of Reward. Due

to these features, the agent started had knowledge of actions it could take, and had an already decent estimate for the value of these actions. By observing the actions myself, I would argue that the agent made reasonable moves overall, if only lacking “foresight”, which is what the reinforcement learning segment would learn. Since these bots represented to top bots in the competition, I believe I did a reasonably well job designing the reward function and the tree search algorithm.

Unfortunately, once I started training, it really showed me how slow this application is. Even with the tree search improvements I made, simulating a game simply took too long. Over the course of training, there was no noticeable improvement of the win-rate against the bots. Simulations ran very slowly, limiting the amount of data gathered. I know that Alpha-Go was trained using multiple Processors and GPUs over the course of weeks. This simply outperformed the hardware and time I had available. Another thing to consider is that due to the already present 50% win-rate, and the difficulty to get over a 70% win-rate due to the nature of the game, there may have not been much room to improve in the first place.

Dropped Feature: Transfer Learning

The last aspect of the project I planned to do was to use Transfer Learning. The idea was that the first model would be taught to play the game in general by training it using random decks. Then, that model would be used as a starting point for another model that learned to play one particular deck of cards, by playing games only with that deck.

This would be done by adding some extra features to the neural network. One more group of features would be added to the state representation: *Deck Features.* Basically, the player would always know the contents of their deck, and the location of those cards (deck, hand, field, or graveyard). This would be represented as a set of 30, 4-length one-hot vectors, resulting in a 120 feature vector. This

vector would be put through a dense network and result in a smaller vector output. This vector output would then be concatenated to the second to last layer in the already trained network.

During back-propagation of this new network, the weights of the old network would not be updates, with the exception of the weights connecting the second-to-last layer to the prediction output.

You once suggested to me that I only train the bot on a subset of the whole card set, but I didn't do this because it would reduce the relevance of this Transfer Learning experiment: do large problem first, then use to solve a smaller one.

Since I didn't end up with a useful final model, and I ran out of time, I didn't end up implementing this feature.

Final Thoughts

From the very beginning, I knew this was a very ambitious and challenging project. I knew reinforcement learning was difficult to get right and time consuming. However, I chose it in spite of these fears because it was a field I was interested in and believed was an important field that felt was under-taught at RPI compared to other machine learning paradigms.

From the beginning, I did have fears regarding the time it would take to complete this project: both time to code and time to train. I tried to reduce coding time by doing all my theorizing and problem formulation beforehand. While I would say that definitely helped, discoveries of issues during implementation forced me to go back to the drawing board occasionally, adding more time theorizing, coding, and testing than I anticipated. Also, I really didn't have an appreciation for the time and resources necessary to do Deep Reinforcement Learning well. But now I think I get what many professors and articles have been telling me: reinforcement learning is hard.

One aspect which took much longer than I would have liked was selection of various hyper parameters. The most annoying ones was the amount of time to allow the search tree to run and the weights in the reward function. I wish I had come up with a way to either calculate good values or to automate their discovery (like with evolution), however, time constraints were always in the back of my mind.

Despite difficulties in practice, I would still say this project was a positive experience for me. I learned plenty about this field and gained experience using it in a large scale project. Even with many of the features I didn't end up implementing (NLP, Transfer Learning), I still learned plenty about and practiced formulating them to a particular project. I'm particularly proud of my work in the *Decomposition of Reward* section due to me not seeing anything like it anywhere else, and it enabling me to create an agent with some knowledge already starting out. I'm also proud of the cleverness of my network design, which received several compliments from potential employers I showed it to. Even the participation in a long-term software project outside of making games was relatively new to me, so I'm happy to have gained more experience there.

If I wish to really spend the time continuing to train the bot, I now have a code base ready and waiting for me. On the other hand, given how computationally expensive the simulations are, I fear I won't have the hardware necessary to do so.

In conclusion, while I couldn't ultimately train a bot successfully due to hardware and time constraints, what I learned about RL in the process and the experience I gained attempting to utilize it in a unique way made the project very rewarding for me.