View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Bard College

Bard

Bard College

Bard Digital Commons

Senior Projects Spring 2018 | Bard Undergraduate Senior Projects

Spring 2018

A Deep Learning Agent for Games with Hidden Information

Robert A. Mills
Bard College, rm5394@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2018

Part of the Other Computer Engineering Commons

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License.

Recommended Citation

Mills, Robert A., "A Deep Learning Agent for Games with Hidden Information" (2018). *Senior Projects Spring 2018*. 204.
https://digitalcommons.bard.edu/senproj_s2018/204

This Open Access work is protected by copyright and/or related rights. It has been provided to you by Bard College's Stevenson Library with permission from the rights-holder(s). You are free to use this work in any way that is permitted by the copyright and related rights. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. For more information, please contact digitalcommons@bard.edu.

Bard

A Deep Learning Agent for Games with Hidden Information

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Bobby Mills

Annandale-on-Hudson, New York
May, 2018

Acknowledgements

To all of my friends, family, and loved ones,
Thank you for your unending love and support
And for not laughing when I pitched my project

Abstract **4**

Introduction **5**

1.1 Pokémon 5

1.2 Game Theory 8

Data Processing **12**

2.1 Problem Description 12

2.2 Entity Embeddings 12

2.3 Team Builders 15

Game Playing Agent **17**

3.1 Problem Description 17

3.2 Tree Searching 18

3.3 Long-Short Term Memory Architecture 21

Results **24**

4.1 Team Builders 24

4.2 Entity Embeddings 25

4.3 Game-Playing Agent 28

Discussion and Future Work **29**

5.1 Overview 29

5.2 Team Builder 29

5.3 Entity Embeddings 29

5.4 Game-Playing Agent 31

5.5 Next Steps 33

Minds, Brains, and Behavior **35**

6.1 Overview 35

6.2 Philosophy 35

6.3 Psychology 36

Appendix **38**

Appendix A: Team Generation Methods 38

Appendix B: LSTM in JavaScript 39

Appendix C: LSTM Formulas 40

Bibliography **41**

Abstract

The goal of this project is to develop an agent capable of playing a particular game at an above average human level. In order to do so we investigated reinforcement and deep learning techniques for making decisions in discrete action spaces with hidden information. The methods we used to accomplish this goal include a standard word2vec implementation, an alpha-beta minimax tree search, and an LSTM network to evaluate game states. Given just the rules of the game and a vector representation of the game states, the agent learned to play the game by competitive self play. The emergent behavior from these techniques was compared to human play.

1

Introduction

1.1 Pokémon

Figure 1.1 Pokémon Game Covers[13]

The Pokémon franchise has existed since the early 1990's. The cute characters translated well from the pages of comic books to the Game Boy Color. While the trading card game has maintained a solid popularity among fans, the video game series has become a flagship series for Nintendo's handheld console line. After seven generations of games and countless spinoffs, the core gameplay remains unchanged. Twenty years since the first game stormed the market, a competitive scene has really found its footing. Each year Nintendo hosts worldwide tournaments in a variety of formats. Players of all ages compete to win prizes. When the games first emerged,

the lack of wireless capabilities on Gameboy devices meant that players had to meet up in order to play in person. This shortcoming meant that competitive Pokémon was relegated to weekend tournaments and casual play between friends. However, recent advances have made Nintendo's handheld gaming device capable of playing games via Wi-fi. The result has been a rapidly changing and well defined metagame.

In competitive gaming there are two layers of strategy that determine every match. On one level, there is the in-game strategy. This is generally the project that game playing AI aims to tackle. The techniques used here are usually either a reinforcement learning algorithm or tree search. However, no player walks into a competitive match without planning. In games like chess and go, the only decisions that define the game are made after the clock starts. However, in games such as *Magic: the Gathering* or Pokémon the players need to make a handful of decisions before starting a match. These are the decisions that define the metagame, sometimes referred to as just the meta. In chess, we may say that the meta is defined by the types of strategies that a player may employ. Knowing that your opponent is more likely to play aggressively or passively can help you find a winning strategy. Pokémon works much the same way. Knowing that the meta favors aggressive play styles will help you make decisions in the game. Pokémon also comes with a set of decisions made before the game begins. These decisions are based on the meta. Some AI techniques such as tree searches are not generally useful here.

Pokémon is a turn-based strategy game. Unlike in games like tic-tac-toe, decisions are made and execute simultaneously by each player. This game structure involves some hidden information. Because each player makes a decision without the knowledge of the other player's

decision, there is no way to be sure to which state an action will take the game. Furthermore, some actions have a stochastic element. The result is an extensive form of the game that is very difficult for humans to work on.

Before a game begins, each player chooses six pokémon(characters) out of a few hundred to comprise their team. Each of those six characters is given four moves each with a limited and preset number of uses. Each pokémon is also equipped with one item and a particular stat distribution. Every pokémon has six statistics which govern their effectiveness in a game: speed, attack, special attack, defense, special defense, and health points(hp). Once your team is assembled, you can take your crew online and look for an opponent.

The goal of each game is to knock out your opponent's pokémon by reducing their health to zero. On any given turn one pokémon from each team is active on the battlefield. Players can choose to have their pokémon use one of their four previously equipped moves to either damage the opponent or otherwise make the game more favorable. Players may also switch their active pokémon out for another pokémon in their team. This action does not allow either the pokémon leaving or entering the battlefield to use one of their moves. Learning when to switch and when to use a particular move are the critical parts of learning to play the game for human players. Each pokémon and each move have a particular type assigned to them. The interactions between those types can be found in the figure below. Note that, for example, ghost-type pokémon are immune to fighting type moves. Flying-type pokémon are weak to electric type moves. These interactions are central to playing the game well. Not only must a human player learn to make effective decisions, good players also learn to anticipate the moves that their opponent will take.

Professional players often choose to switch to a pokémon that resists a type of move when they believe that their opponent will try to use that type.

1.2 Game Theory

Game theory is a relatively young field of Mathematics. The objects of Game theory are the strategies used to play games. In order to best describe strategies, mathematicians appeal to the way that players value states. Though there are some examples of mathematical analyses of games going as far back as James Waldegrave's letters with his nephew regarding the card game, the field as we know it today began in the 1930's [11]. Shortly following John von Neumann's book *Theory of Games and Economic Behavior* in 1944, John Nash contributed what is perhaps the most recognizable finding of game theory: the Nash Equilibrium [6].

One helpful way of describing game states is by assigning a utility to an outcome. Real utility functions are not necessarily linear or logical. However we will use a simple example. Consider the following simple game. Each player has a penny. First, Player One chooses (out loud) heads or tails. Then, Player Two does the same. If the choices are the same, Player One takes both pennies. If the choices are different Player Two takes both pennies. The table below shows the value, otherwise called utility, of each state as a pair of values in the form (Player One's value, Player Two's values). This representation is called a strategic form. Note that each player's reward is simply the negative of the other player's reward. This kind of game is called a zero-sum game. Any utility that one player receives is utility lost by the other player. Not all games are zero-sum. Many games have cooperative strategies in which both players receive their highest reward by working together.

	Player Two		
Player One		Heads	Tails
	Heads	(1, -1)	(-1,1)
	Tails	(-1,1)	(1,-1)

Figure 1.2 A strategic form representation of the penny game

The example game above is called a “perfect information game.” This kind of game is defined by the availability of the entire game state to every player. Perfect information games can also be described with a game tree. This representation is sometimes referred to as an extensive form. Below is a game tree detailing each state that the game could reach and each action available to the players from that state. Note that the leaves of the tree are labeled with the winner of the game if that state is reached. Nodes are color-coordinated to show which player must make a decision from that state. The edges are labeled with the choice that a player makes from a state.

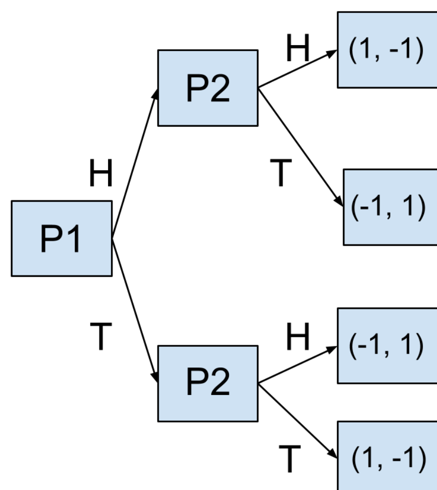


Figure 1.3 An extensive form representation of the penny game

Game trees, like the one above, help us determine “winning strategies” for a player. These strategies are ones that guarantee a win for a player. Player Two has a winning strategy in the game above. As long as Player Two chooses the opposite of what Player One chooses, Player Two will always win. Player one has no winning strategy in this game.

Some games lack this perfect information quality due to an element of chance or hidden information. Imperfect information games can still be described using a game tree. We can modify the rules of the game above to see how we might describe such a game. Suppose now, the players make decisions and reveal their choices at the same time. The outcome rules will remain the same. Now we will need to modify our game tree to represent what a given player knows about a game state. To do this, we will group states into “information sets”. Two states belong to the same information set if a player can not determine which of the two state they are in. Below is a game tree describing this game from the perspective of Player One.

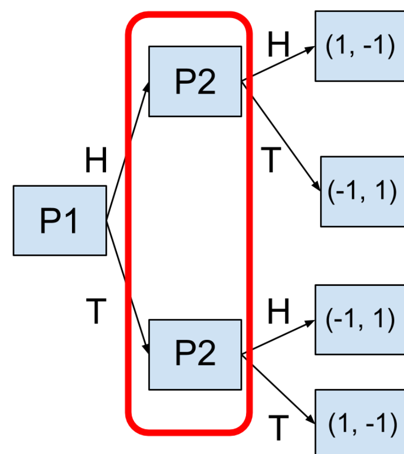


Figure 1.4 An extensive form game of the imperfect information variant of the penny game

Note that Player One cannot tell if they are in the state where Player Two has chose heads or tails when they make a decision. Therefore, we group these states into an information set. Now that we have a game tree, we determine that no player has a winning strategy in this game. As long as we can build an extensive form representation of a game, we can perform tree search techniques and find winning strategies even if that tree contains information sets.

2

Data Processing

2.1 Problem Description

This project is builds on research and applications of game playing techniques for artificial agents from the past few years [9]. In addition, it also is built on top of a recent project for a Pokémon playing agent. In 2015 a graduate student at Stanford University released a pokemon playing agent [1]. The program was built using the JavaScript runtime Node. The project included a minimax and greedy algorithm for playing the game. The majority of the thousands of lines of code were dedicated to the game engine and interfacing therewith. One early hurdle for this thesis was combing through thousands of lines of code to find the places that needed to be updated or otherwise modified. In the end the major contribution of this thesis besides general maintenance has been to develop a new function for describing and evaluating game states.

This project had two phases. The first phase consisted of collecting and analyzing data. In this step the goal was to determine what underlying features of the game could be exploited. The second phase involved choosing and implementing a model to evaluate the data.

2.2 Entity Embeddings

The first and most obvious problem for a game playing agent is determining a state description. Otherwise called feature engineering, we must first choose which features of a state best

encapsulate the relevant information for our agent. This problem is often referred to as "the framing problem." For agents and robots working in the real world choosing how to describe inputs for your agent is an extremely difficult problem. At any given moment, the human experience contains more information than is communicable. Luckily for game playing agents, the "world" is restricted to a much more easily described state. Indeed for game-playing agents it is not the selection of features that matter so much as the representation thereof. Early neural-network- based reinforcement learning agents, such as TD-Gammon, used a series of integer or binary numbers to describe positions of stones on a board as well as a few binary descriptions. Google's Alpha-Go Zero represents the board as a grid [5]. This representation of the game state allows their agent to use a convolutional neural network. Representation matters.

Previous iterations of Pokémon-playing agents have used easily accessible numerical descriptions to capture some of the information of a pokémon [1]. Such descriptions usually include values for remaining health and stat boosts. These descriptions are often augmented with a set of binary features to describe the state of the playing field as well as effects from previously used moves. Some researchers will often include some engineered features in their state descriptions such as binary values to answer question such as: Does a pokémonhave a supereffective move against their opponent's active pokémon? Is this pokémon faster than the opponent's active pokémon? Does this pokemon have a priority move? These kinds of questions are useful in that they catch many of the features that human players seek when making decisions while playing.

The above description leaves out one major aspect of pokémon descriptions: stats. Each pokémon has six stats that describe their functionality as well as their role. A pokémon could be

described by these six stats, their type, and their available moves. The problem with this representation is that the only information directly available to us in the game is the type of each pokémon. The stat description may be inferred as the game goes on, but the Bayesian inferences and computational power needed to leverage our partial information eats away at the limited time available to our agent during each turn. A simpler, more generalizable representation is preferred.

In order to improve upon the state representation from previous projects,we first use a common technique from natural language processing. Our goal is to find a representation of pokémon that captures not only their relative strengths but also the roles that those pokémon play on a team. Some teams rely heavily on specific synergies between two pokémon . Ideally our representation would be context sensitive. With this representation our model knows that pokémon likely fill different roles in different contexts. Supervised learning approaches were impossible here due to a lack of label data. Classifying pokémon into roles on a team is both arbitrary and not discrete. Pokémon can fill more than one role and there is some disagreement about what those roles are and which pokemon fall into which role. To solve this problem we employ the Word2Vec algorithm to generate embeddings for our pokémon .

Word2Vec(w2v) has many variations for different use cases. We used an out-of-the-box method that implements a continuous bag-of-words model (CBOW). This method means that the algorithm is learning to predict words given their context. Consider the word “cat.” Ideally, a vector representation of “cat” would be similar to a representation of “kitten.” Because the words are used in a similar context, they should have similar representations. Using a w2v model on large corpuses of English text often reveals words to be grouped together based on their meaning

and part of speech. Verbs are generally separate from nouns. Furthermore, nouns relating to travel are separated from nouns about food. The model learns to cluster entities based not only on data that can be labelled, such as part of speech, but also on more arbitrary ideas such as domain. This clustering is important for our use because we need to capture data that can be labeled(the stats of pokémon), yet is unknown, as well as information that is not easily classified. One other peculiarity of the continuous bag-of-words model is that words are represented as the average of their contexts. For example, “apple” could refer to either the fruit or the company. As a result of this ambiguity, the vector representation is somewhere between business-related words and food-related words. This means that our model will likely catch that some pokémon can fill more than one role. Hence one should consider average of those roles weighted by their frequency in a given context in our training data sets.

2.3 Team Builders

To implement the w2v algorithm using CBOW, we need to collect data to train and test on and we need to transform our problem into a representation that can take advantage of w2v. We will consider teams of pokémon akin to sentences. The pokémon are the words from which we are trying to learn a relationship between. In order to train our model we need examples of teams that might see play in the tier of the game we are playing. One way to collect this data would be to simply play the game and record the teams that we see. Unfortunately, this method requires that we already have an agent which can play at the level we require. If we could collect the data using this method, we would have already solved our problem and would have no need for the data. Collecting data by playing the game would also be an incredibly time consuming method. The program is only capable of playing roughly a hundred games in a day if it plays

non-stop. In order to have 10,000 team examples, the program would have to play continuously for three months. The time-scale of this project did not allow for such a data collection method. Instead, we opted to generate sample teams using publicly available data.

Smogon, the group that developed and maintains the website and platform on which we trained and tested our bot, publishes monthly statistics regarding pokémon usage and team play. For any given pokemon we have access to the percent of teams that used that pokémon this month. These usages are broken down by tiers. Furthermore, Smogon offers pairwise conditional probabilities - given that a Pikachu is on a team, how much more or less likely is that a Charizard is also on that team? This information gives us a simple technique to construct teams probabilistically using a hill-climbing algorithm. First, the usage probabilities are normalized. The program draws a pokémon from this set without replacement. The remaining pokémons' probabilities are updated using the pairwise-conditional probabilities of the pokemon just chosen. The set is re-normalized and is drawn from again. This process is repeated six times until a base team is constructed. Then the algorithm removes the first pokémon selected, re-normalizes and draws again. The algorithm will slowly climb to a more probable team. We found that this process results in a set of teams that roughly matches the expected distribution of pokémon found in the original statistics. These teams are fed to the word2vec algorithm to generate a vector representation for our model. Code for the team builders can be found in appendix A.

3

Game Playing Agent

3.1 Problem Description

The two techniques used to actually play the game, given our representation of game states, are a minimax tree search algorithm and an LSTM to evaluate game states. The tree of game states is too deep to reach a terminal node to get evaluations of previous states. To solve this deficiency of exhaustive tree search, we use an evaluation function approximated by a neural network. Figure 3.1 below shows a basic version of the network used to evaluate game states.

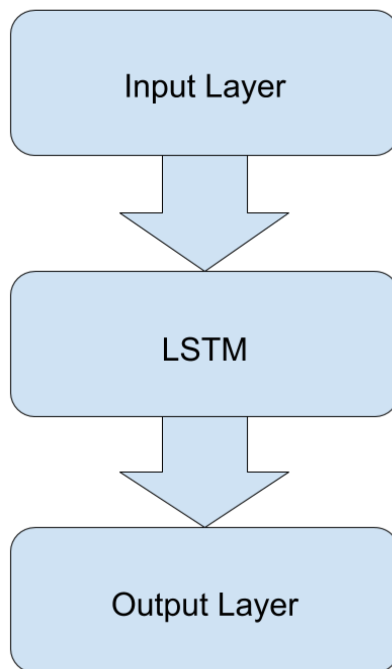


Figure 3.1 A simple illustration of the evaluation network

3.2 Tree Searching

One major advantage that game playing agents have over other kinds of reinforcement learning agents is access to a transition function. Our agent is not trying to learn the rules of the game but rather how to act under those rules. There exists a whole class of reinforcement learning algorithms for learning models of environments. Luckily the rules of the game are given to the bot ahead of time in the form of a game engine. We can pass the engine a state-action pair and receive a new state or set of states with probabilities for landing in each. The inclusion of a transition function allows us to use tree search algorithms to determine which action we should choose in a given state.

The tree search algorithm we implemented is called the minimax tree search with alpha-beta pruning. First, we will describe the vanilla minimax algorithm and then modify it with alpha-beta pruning to make it more efficient. The main intuition behind minimax is rather simple. We will assume that each player is acting to maximize their own expected value from the game. Because Pokémon is a zero-sum game, we know that the action that is best for one player is also worst for the other player. This insight allows us to predict moves that the other player is likely to make from a given state by assuming they will choose the move which gives us the lowest possible value. Likewise, we will always choose action which yields for us the highest possible value.

The minimax algorithm works as follows begin at a leaf node. Then, we label the parent node with this value. Now, we compare that parent's new value with the value of the leaf node's siblings. If this node represents a state from which player one makes a decision, then we label it with the highest value of its children. Else, we label this node with the lowest value among its

children. Once all children have been considered, we then propagate the value higher in the tree by comparing this node's value with that of its parent. If the parent is empty, we label it with the new value. We repeat this process until all nodes have been labeled.

Consider the example tree below. The leaf nodes are labeled with the values Player One receives at that state. The layer above is labeled with the minimum of the children of those nodes. The root is labeled with the max of its children. To perform minimax on this tree, we would begin with the leaf node label with a -1. We propagate this value up to its parent because that node is empty to begin with. Then, we compare -1 to -5 and choose the minimum. There are no children left so we propagate this value higher. The root node is temporarily labeled with -5. Now we move down the right side of the tree until we reach a leaf node. The node we have reached is labeled 3 so we push that value up the tree. Now we compare 3 with 4 and choose the lesser. Next we compare the value of the root to the value of last node we labeled. We maximize this value by choosing 3. We have determined 3 to be the value of our root node. Notice that Player One would prefer to play in the right branch of the tree, but Player Two would prefer us to receive the value 3 instead of 4. Recall from our previous discussion regarding game trees and extensive forms of games that we can represent games with hidden information using trees. The algorithm does not change when information sets are present in our game tree.

One way to make tree searches more effective is a method called pruning. As the name suggests, our goal when pruning a tree is to "chop off" certain branches by not searching them. We can do this by cleverly propagating information about the best and worst values we have seen from a particular branch. In doing so, we can should be able to tell whether or not searching down a branch will yield a better result than what we already have. This technique will allow us

to search deeper into a tree by saving time by not searching every branch. The pruning technique we used is called alpha-beta pruning. Alpha and beta are the best and worst values a given node has seen on the path to the root. So now we will not only be labelling each node with our best guess of the value of that node, but we will also label it with an alpha and a beta value.

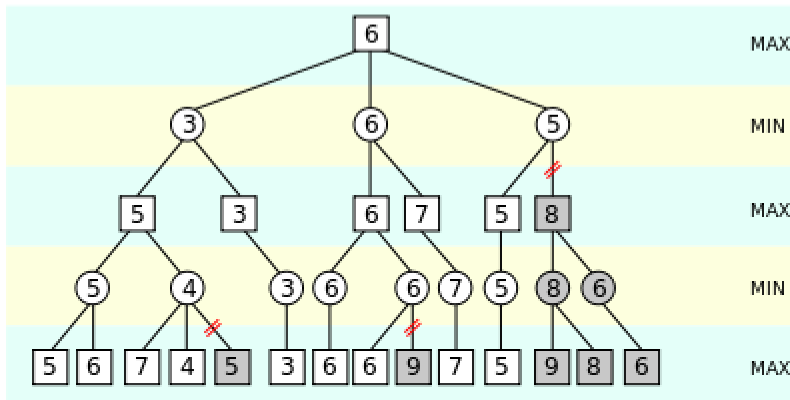


Figure 3.2 Note that this tee was searched in a left to right [14]

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 3.3 Psuedocode of minimax with alpha-beta pruning[2]

The psuedocode above describes the alpha-beta pruning algorithm. The tree in figure 3.2 shows the result of alpha-beta pruning. Note that the layers of the tree alternate between maximizer and minimizer nodes. The maximizer nodes are states where Player One must make an action. The minimizer nodes are nodes where Player Two must make an action. Where the beta value of an edge can guarantee a better worst case scenario by a path that it has already visited, the algorithm will prune the tree. Red dashes across an edge indicate a pruned branch. The leaf node with value 5 not visited by this tree search. The key intuition here is that pruning occurs when the algorithm can guarantee a better value by taking a different action earlier in the game tree.

3.3 Long-Short Term Memory Architecture

The network architecture we implemented uses Long Short-Term Memory (LSTM) cells. This architecture is a kind of recurrent neural network. It is often used in natural language processing projects because of its unique time-based activation and training algorithm. Recurrent neural networks have an advantage over traditional feedforward networks in that they can "remember" previous states. This advantage makes them very useful for time series prediction and predicting words that are likely to follow a given sequence in natural language processing. We are able to pass the network a series of time steps(game states) and it will return a probability that the given player will win from this state. This behavior more closely mimics the kind of decision-making that real players enact when playing a game. The previous states inform one's decisions. This architecture helps us fight the classic credit assignment problem in reinforcement learning. It is often said that some games can be determined by opening moves. However,

learning to recognize how decisions contribute to states that are many time steps away is a difficult problem. This learning is difficult if a network is only given the current state to evaluate. For this reason we decided to implement an architecture that can accept a series of game states.

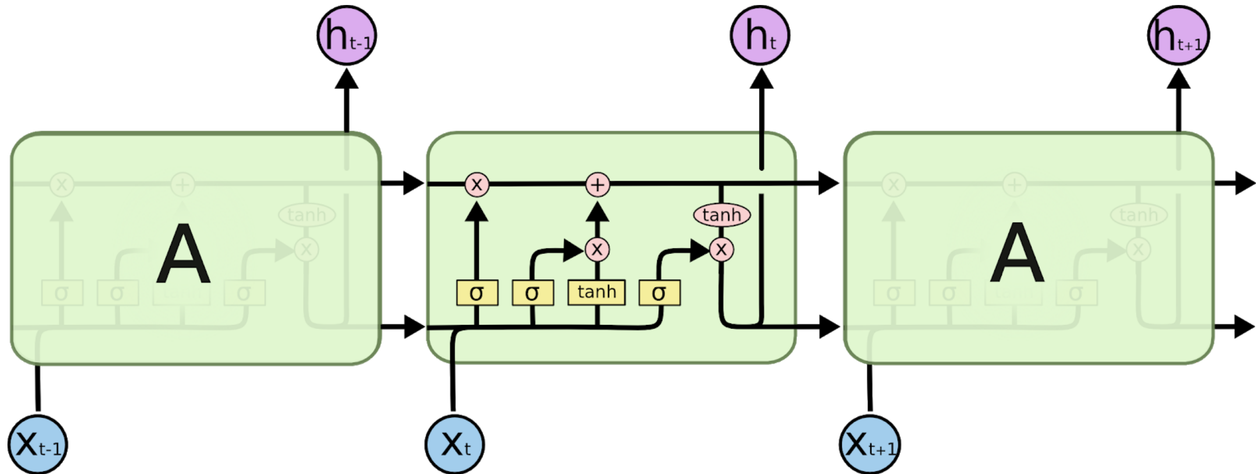


Figure 3.4 The inner workings of an LSTM cell in context of its neighbors[12]

The figure above shows the inner workings of an LSTM cell. Note that each cell receives a unique input and produces a unique output. Also note the way that information is propagated from a cell to its neighbor. The top lateral input is the state from the last cell. The new LSTM cell will multiply this vector by the result of activating the first "forget gate" on the bottom lateral input. A forget gate lets the cell learn which features need to be remembered to update the state before we pass it to the next cell. Next the cell applies another sigmoid gate and a tanh activation to the output of the last cell. This process determines the updates we want to add to the previous description of the state. Finally, the cell performs an activation function on the state description from the previous cell. This activation is filtered through a gate which determines what the cell will output to the next layer of the network. This vector also becomes the bottom

lateral input for the next cell. All of these steps and the relevant formulas can be found in the Appendix.

The LSTM cell described above works well because it can “remember” states that earlier cells have seen. However, this remembering is dependent upon gates that receive a lateral information. Our standard training algorithm for neural networks trains one layer at at time. Standard back-propagation cannot train the gates inside of the LSTM cells. So instead, we will use an algorithm called backpropagation through time(BPTT).

The calculus and linear algebra involved here are rather similar to that of standard back propagation. We will still use the chain rule of differentiation. The key insight for BPTT is that we first must “unroll” the LSTM layer. In order to calculate the error for the third cell we must sum the errors of cells from third cell down to the first. We must perform this technique for each output in the LSTM layer. The major departure from standard back propagation is that BPTT trains laterally. In standard backpropagation, the change in weights for one neuron do not affect the weights of other neurons in that layer. For further information on LSTM’s, BPTT, and recurrent neural networks in general, we recommend chapter 10 of Ian Goodfellow’s *Deep Learning* which is available online in full.[4]

4

Results

4.1 Team Builders

The team generator, featured in the code below, creates teams from a given set of probabilities. The generator has two iterations. The first iteration is called the “naive” generator which builds teams from only the usage probabilities of a each pokémon . These probabilities describe the number of teams in which a given pokémon is present. The second iteration, called the “smart” generator, builds teams with the usage probabilities of each pokémon and the “teammates” value described in section 2.2. Recall that this value is the change in the probability of a pokémon being in a team given that another pokémon is on that team as well. To test the efficacy of these team-generation methods, each method generated 10,000 teams. The usage data is drawn from is collected from a set of 30,000 teams. The usage rate of each pokémon is then calculated via each method. We then determined the difference between the generated usage rate and the original usage rate.

	Average Usage Difference	Variance of Usage Distance	Average Teammate Usage Distance	Teammate Usage Distance Variance
Naive Team Builder	0.18%	0.28%	1.16%	4.70%
Smart Team Builder	0.12%	0.29%	1.14%	6.76%

Figure 4.1 A comparison of the difference in percent between expected and generated usage and pairwise teammate usage values.

4.2 Entity Embeddings

To determine what the word embeddings had captured, we employed a few statistical tests. Although the meaning of the embedding is subjective, some numerical descriptions such as word similarity or cosine distance can show us how the entities are embedded. We describe the similarity of three descriptions of each entity. The first description is using only the publicly available stats of each pokémon . The second description is a result of performing the Word2Vec embedding described in section 2.1. Finally, we combine these two descriptions.

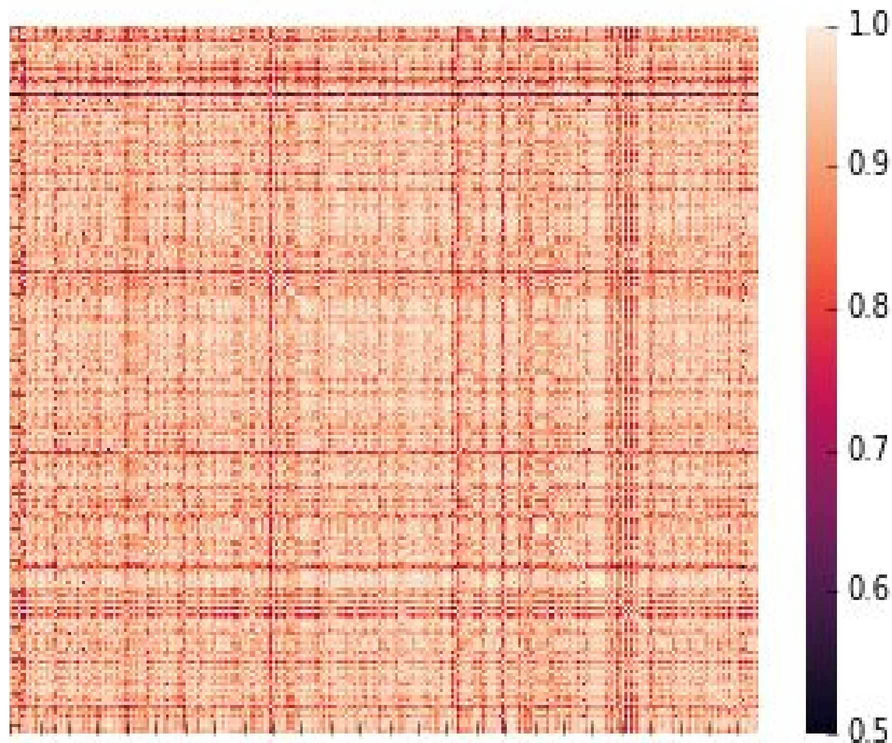


Figure 4.2 Heatmap of the pairwise similarities of all pokémon using Word2vec

The first feature description produced the heatmaps above. The first figure depicts the cosine similarity of all pairs of the 169 pokémon . Lighter colors indicate a shorter distance and therefore higher similarity.

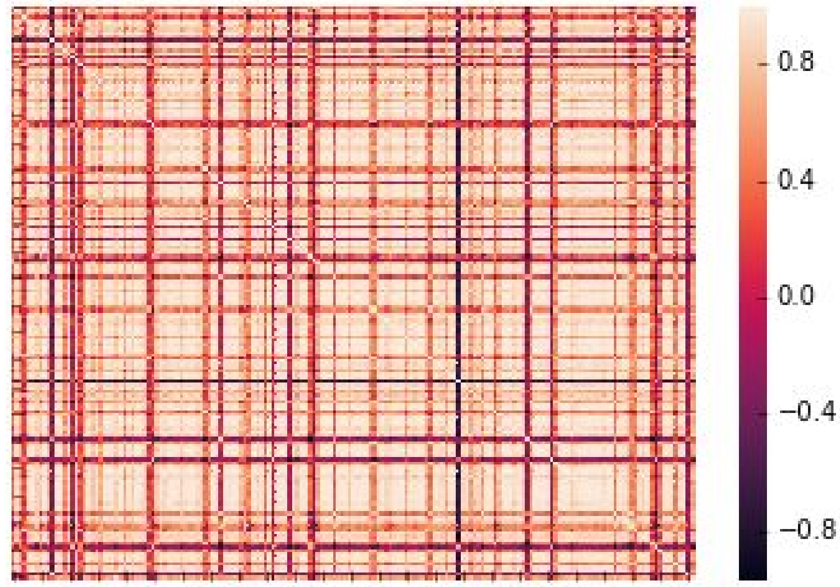


Figure 4.3 Heat map of cosine similarity of pokémon using just in-game stats

The second description is the result of the Word2Vec embedding. Here, dark colors indicate a negative relationship and light colors indicate a positive relationship. A negative relationship implies an opposite "definition." For instance, "up" and "down" would have a negative correlation in a Word2Vec model trained on an English language corpus. The light colors indicate that two pokémon are similar.

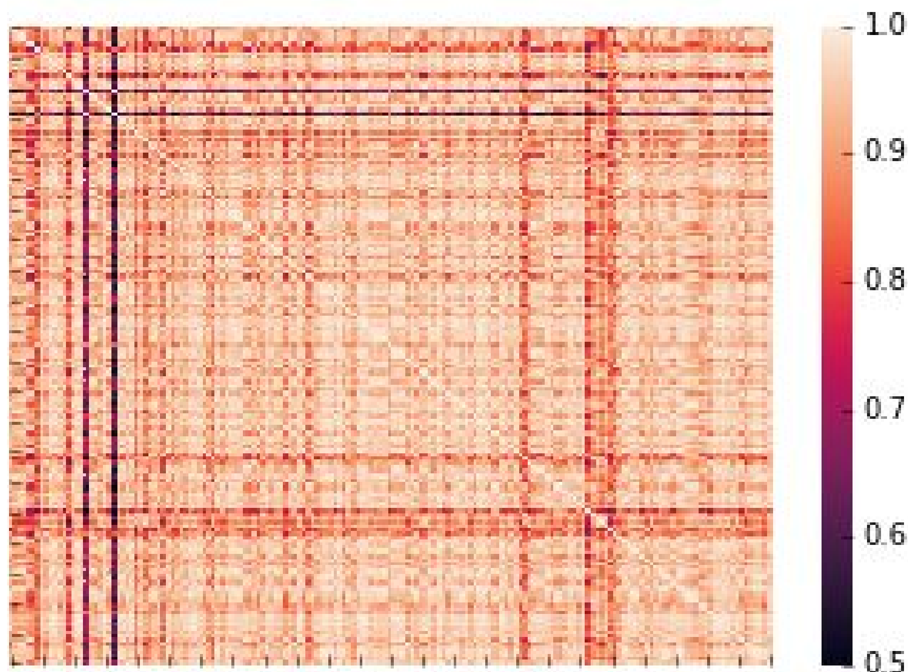


Figure 4.4 Heat map of an embedding using both Word2vec and the in-game stats of pokémon

The final representation of the pokémon is a combination of both earlier descriptions. Using both the Word2Vec embedding and the numerical description of pokémon via their in-game stats yielded the embedding shown above. This heatmap is using the same cosine distance measure as in Figure 4.2. Below is a table recording the averages and variances of pairwise distances of all pokémon using each of three embedding techniques.

	Average Distance	Variance in distances
Word2Vec only	.048	.094
Stats only	.043	.012
Word2Vec and Stats	.042	.012

Figure 4.5 the averages and variances of pairwise distances of all pokémon using each of three embedding techniques.

4.3 Game-Playing Agent

The network used to evaluate games states was successfully integrated with the existing JavaScript application for playing the game online. The network can accept a series of game states and produce a likelihood of winning from this position. The bot now also contains methods for collecting data such as series of game states paired with reward values and teams observed in play. At the end of each game, the bot can now train the network on either a single game or a batch of games.

Results regarding the loss values from training were not possible given the time scale of this project. See the chapter 5 for relevant details of behaviors demonstrated by the network, training times, and methods for reducing the burden of data.

5

Discussion and Future Work

5.1 Overview

Despite a successfully generated set teams, the none of the embeddings appear to successfully capture the semantic relationship between pokémon. However, with limited results to analyze, the state evaluation network does appear to be learning to play the game.

5.2 Team Builder

Surprisingly, the naive team builder performed just as well as the more complicated second iteration. There appears to be no significant difference between the results of the "naive" or "smart" versions of the team builder. This finding is against our intuition. However, these findings do validate our aim to replace the embeddings generated from these teams with an embedding based on teams actually observed while playing the game. Although the second iteration did not improve upon the naive team builder, the results of both versions are encouraging. The usage statistics of the generated teams do closely match those reported from smogon's public reports.

5.3 Entity Embeddings

Determining the effectiveness of an embedding is a matter of performance. An embedding is more or less effective than a different representation if it facilitates better learning. However, there are a few measures by which we study the embedding directly. The provided heat maps show how some of characters are related in our representation.

The new vector representation does not appear to successfully capture the relation between pokémon . The global heat maps are fairly bland. Both Figure 4.2 and Figure 4.3 show that most of the pokémon have high similarity values. The more complex representation did not help further differentiate pokémon as indicated by Figure 4.4. The large amount of lightly colored squares indicate that few of the pokémon are very dissimilar. This conclusion is further supported by the table in Figure 4.5. The low average distance between pokémon coupled with the low variance in these distances suggest that the overwhelming majority of pokémon are grouped tightly together in a singly cluster. A low average distance but high variance would suggest that the pokémon we embedded evenly throughout the space. Unfortunately this was not the case. From these results it is impossible to tell whether or not Word2Vec is simply not the right tool for the job or it was not used effectively in this implementation.

The Word2Vec model may have failed to capture the relationship between pokémon but it was successful in dramatically reducing the size of the network. Recall that the Word2Vec algorithm produced a five feature embedding for each entity. Each pokémon has eight stats. The game state otherwise has 51 features. A full explanation of the network structure can be found in the chapter 3.

Without this embedding, this network would have been overwhelmingly large. Some quick back-of-the-envelope math suggests the input layer would have been over 6,000 neurons large. Without some kind of feature engineering, the canonical way to represent a word in a vocabulary is by a one hot encoding. Using this technique would mean that each of the twelve pokémon is represented by a vector that consists of 168 zeros and a single 1. Without any of the other features of the game state, the input vector for the twelve pokémon alone would have been

just over 2,000 features long. Estimating conservatively that there are only 100 moves which see play, we could give each pokémon a four hot embedding one hundred features long. This would give us an additional 4,800 neurons in our input layers. The final network uses an input layer of fewer than 300 neurons.

5.4 Game-Playing Agent

On the burden of data:

Google's DeepMind group trained on roughly 5,000,000 games over the course of one month to achieve their results. However, much of the progress made in training occured within the first few days. After three weeks, they had surpassed the best human player. This project can produce one game of training data in about six minutes running on a single NVIDIA GTX 980. It would take this setup roughly fifty years to generate 5,000,000 games. Google used four tensor processing units(TPU's) to train AlphaGo Zero [5]. A single TPU is twenty times faster than a single GTX 980.

This bulk of data is not necessary to show progress or to learn behaviors. AlphaGo Zero exhibits many interesting strategies and appears to learn fundamental concepts after only a few hours of playing. These findings suggest that, even with the current unoptimized code, some progress can be made after only a week of training on 1,600 games. However, the time constraints of this project prohibited both training on this scale and tweaking hyperparameters.

Further code optimizations may also improve training times. If games were generated locally via the game engine instead of played online, game times could be cut in half or better. It is conceivable that a slight hardware upgrade coupled with a more efficient data collection

method may result in upwards of 1,000 games every two days. Distributed computing and online platforms such as Amazon Web Services may be one solution to facilitate such data collection. With these improvements, significant results could be obtained in just a few weeks. However this time scale remains outside the scope of this project.

On the exhibited behaviors:

After only a few days of training, the network began to exhibit behaviors consistent with beginner level play. For example, the network needed only a few days to learn the importance of stat boosting moves. The agent would frequently prioritize moves that boosted its active pokémon's stats over moves that deal damage to the opponent. This behavior was often exaggerated and lacked the nuance of average human level play. The agent would frequently use a stat boost move three or four times in a row when just once or twice would have been sufficient. Furthermore, the agent failed to learn that stats have a limit. The agent sometimes boosted itself multiple times after the boosting move no longer had any additional effect.

Similarly, the agent learned rather early on after just a day of training that health is critically important to winning. This learning resulted in the agent putting an overemphasis on healing moves. The agent would sometimes forego attacking the opponent to use moves that heal itself instead. This strategy is often referred to as a "stall" tactic. While it is a legitimate strategy, in order to be effective, the player must also use moves which damage over time or inflict lasting damage conditions on the opponent. The agent did not learn this asset of the strategy. Instead, the agent would attempt to force the opponent to run out of uses of its best moves. This strategy resulted in the opponent lacking the necessary tools to defeat the agent. So, in a sense, the agent did find a successful strategy. However, this strategy is easily countered by human players of an

average level. Overall, the quick emergence of recognizable strategy is encouraging for an LSTM-based line of research.

5.5 Next Steps

This project has laid the groundwork for a few interesting research projects. The most conservative research direction would be to simply improve upon the network used in this project. Due to time and hardware constraints, the network was not allowed to train for long enough to gather numerical test results. Future work may include optimization of the code suggested in section 5.4. A project of this nature should include a benchmark of simpler feedforward evaluation network. Each of these networks should be tested on a live server against human players.

A more ambitious but potentially more fruitful avenue for future work would be to implement a new tree search algorithm. Right now, the agent is using minimax with alpha-beta pruning. A future project may implement a Monte-Carlo Tree Search instead. This methodology is closer to the technique used by Google's Alpha Go Zero [5]. A project of this variety should also investigate the use of safe and nested subgames as used by Noam Brown and Tuomas Sandholm to play heads-up-no-limit Texas hold'em poker.[8] This line of research may also require a novel adaption of an existing tree search technique such as Monte-Carlo Tree Searches.

One other area that a future project could improve upon this work is by further investigating the issue of representation. Further research is needed to determine whether or not Word2Vec can be used to embed pokémon based on their appearance in a set of teams. This research may take the form of a more comprehensive study of the hyperparameters' effect on the

embeddings. Because the vocabulary size, training data, and sentence length are smaller than what is typical for a natural language processing project, it is likely that the model simply overfit to the given data set. One possible solution to this problem is to develop a novel adaptation of a known embedding technique.

6

Minds, Brains, and Behavior

6.1 Overview

This project can be divided into two sections - (1) creating and working with representations of game states and (2) playing the game. The first consists of the data processing and deep learning techniques used to find meaning in large amounts of data. The second section involves both the actual JavaScript application and the tree search technique implemented therein to play the game. Most importantly, these sections can be classified by the philosophical questions they investigate. The data processing techniques are in part an inquiry into how we determine the semantic value of entities. The parts of this senior project regarding the actual gameplay are aimed at the way that we can infer the semantic values that others hold. The particular insight of this project is the way that the neural network techniques bridge the gap between these two areas of research.

6.2 Philosophy

The Word2Vec(w2v) algorithm described in section 2.1 aligns closely to the pragmatism characteristic of Ludwig Wittgenstein's later work. In *Philosophical Investigations*, Wittgenstein develops a theory of what is for a sign to have a meaning [7]. A sign can be anything that can convey meaning. Words are the typical example of signs. However, we may also consider vocal inflection, gestures, and appearance choices to be signs as well.. Though other linguists and philosophers have stressed the important of context to the meaning of signs.

Wittgenstein takes this idea as step further. Wittgenstein writes clearly that “the meaning of a word is its use in language [7].”

The defining insight of w2v is a numerical representation of the semantic relationship of words based on their context. The vector representation of a word is constructed from the probabilities of other words to appear near it. The meaning of words are derived from their usage. The representation we generate for the Pokémon appears to be consistent with Wittgenstein’s pragmatism.

6.3 Psychology

The implementation of the game playing agent, including the tree search algorithm described in section 3.2, are examples of investigations into mind reading techniques. The tree search algorithm is attempting to predict the actions of another player. The same network is used to evaluate game states for both the agent and their opponent. This implementation choice makes two key assumptions. The first assumption is that, the opponent will always make the decision that gives them the best chance of winning. Secondly, that the opponent arrives at this evaluation similarly to the way that the agent does.

The assumptions that this model makes are consistent with both Wittgenstein and psychologists such as Jeremy Carpendale and Charlie Lewis. Wittgenstein asks in *Philosophical Investigations*, “Does a child learn only to talk, or also to think? Does a child learn the sense of multiplication before or after it learns multiplication [7]?” Carpendale and Lewis respond by saying that a child learns these things at the same time [3]. This response, in conjunction with the reading of Wittgenstein's *Philosophical Investigations* implies that the concept of multiplication

and the process or language thereof are the same thing. Language and thinking are inextricably interwoven.

From this position, Carpendale and Lewis describe how reasoning is explained in their model. Carpendale and Lewis suggest that "children understand talk about the psychological world as a pattern of activity…[3]" Furthermore, reasoning is not "based on the application of rules, but particular instances of rules would be manifest in the process of reasoning [3]." Reasoning, as Carpendale and Lewis describe here, is analogous to the tree search that our agent is performing.

If Carpendale and Lewis are correct, then the assumption that our game-playing agent makes regarding the way that its opponent evaluates states comes down to an issue of representation. Our agent cannot learn the way the that their opponent represents states. Although that representation is critical to reasoning, our agent has no choice but to use its own representation. Overcoming this semantic gap is not just an engineering problem, but it is also fundamentally an epistemological issue. By embracing pragmatism, we may also be able to disregard this problem. Though the agent cannot learn the representation that it opponent uses, if the agent wins the game, then it has done well enough. If the agent can consistently reason its way to victory, then it has learned a representation that provides enough sematic middle ground between its representation and that of its opponent. The insights provided by Carpendale and Lewis are the basis for claiming that winning games or the appearance complex strategies which rely on prediction are ample grounds for deciding the success of the agent and of this project.

Appendix

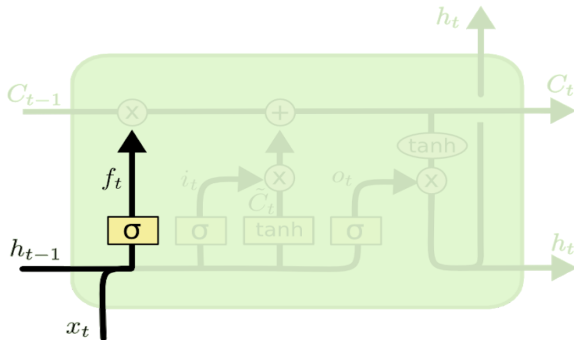
Appendix A: Team Generation Methods

```
def naiveTeam():
    return list(choice(list(pokeList), 6, p =
norm(baseUsage),replace = False))

def smartTeam():
    team =[]
    smartUsage = list(baseUsage)
    smartPoke = list(pokeList)
    smartNorm = []
    while len(team) < 6:
        smartNorm = norm(smartUsage)
        chosen = choice(smartPoke, p = smartNorm)
        if chosen not in team:
            team.append(chosen)
            for poke in data[chosen]['Teammates']:
                if poke in smartUsage:
                    smartUsage[smartPoke.index(poke)] +=
data[chosen]['Teammates'][poke]/float(sum(list(data[poke]['
Abilities'].values())))
    return team

def norm(given):
    return [i/sum(given) for i in given]
```

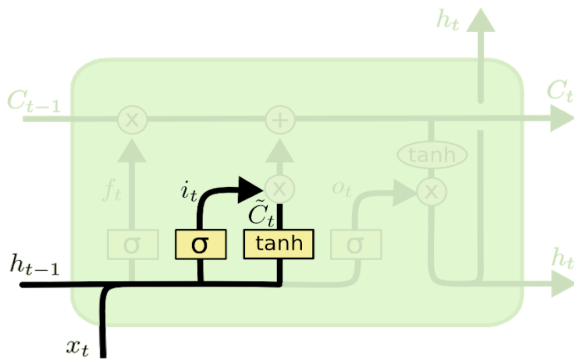
Appendix B: LSTM in JavaScript

```
var train_net = module.exports.train_net = function(win) {
    learnlog.info("Training neural network...");

    const trainSet = [];
    _.each(states, function(state){
           trainSet.push({input: state, output: [win ? 1 :
0]});
        });

    const results = trainer.train(trainSet, trainOptions);
    learnlog.info(results);
    states = [];
    fs.writeFileSync('lstm.json',
JSON.stringify(lstm.toJSON()));
}
function eval(battle) {
    var value = 0;
    var features = getVec(battle);
    console.log(features.length);

    value = lstm.activate(features);

    logger.trace(JSON.stringify(features) + ": " + value);
    return value;
}
```

Appendix C: LSTM Formulas

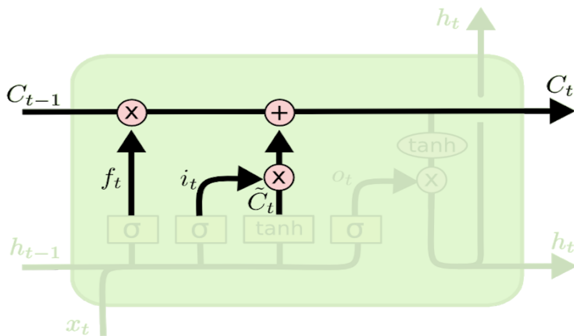


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

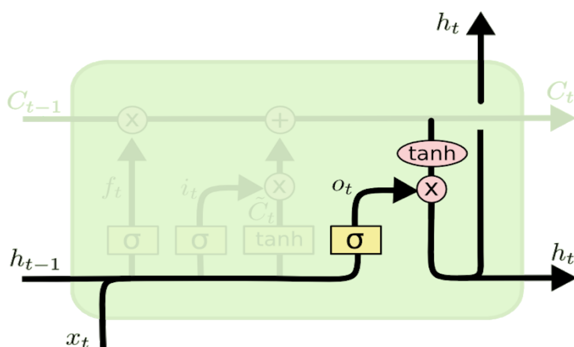


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Bibliography

[1]
K. Khosla, L. Lin, and C. Qi, “Artificial Intelligence for Pokemon Showdown,” p. 8.

[2]
S. J. Russell, P. Norvig, and E. Davis, *Artificial intelligence: a modern approach*, 3rd ed. Upper Saddle River: Prentice Hall, 2010.

[3]
J. I. M. Carpendale and C. Lewis, “Constructing an understanding of mind: The development of children’s social understanding within social interaction,” *Behavioral and Brain Sciences*, vol. 27, no. 01, Feb. 2004.

[4]
I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016.

[5]
D. Silver *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[6]
J. Nash, “Non-Cooperative Games,” *The Annals of Mathematics*, vol. 54, no. 2, p. 286, Sep. 1951.

[7]
L. Wittgenstein and G. E. M. Anscombe, *Philosophical investigations: the English text of the third edition*, 3. ed. Englewood Cliffs, N.J: Prentice Hall, 2000.

[8]
N. Brown and T. Sandholm, “Safe and Nested Subgame Solving for Imperfect-Information Games,” *arXiv:1705.02955 [cs]*, May 2017.

[9]
S. Lee and J. Togelius, “Showdown AI competition,” 2017, pp. 191–198.

[10]
M. Kawamura, “[Studies on low volume priming heart lung bypass (author’s transl)],” *Hokkaido Igaku Zasshi*, vol. 50, no. 2, pp. 137–167, Mar. 1975.

[11]

D. BELLHOUSE, “The Problem of Waldegrave,” p. 12.

[12]

C. Olah, “Understanding LSTM Networks -- colah’s blog,” *colah.github.io*. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 02-May-2018].

[13]

Jez9999 at English Wikipedia, *AB pruning.svg*. 2007.

[14]

@Pokemon, *What was your first Pokémon video game? #Pokemon20*. .