AI Semantle

Team Members: Harris Greenstein (hag65), Steven Urdaneta (sau9), Simon Kapen(sak299)
AI Keywords: Reinforcement Learning, Natural Language Processing
Application Setting: Semantle
External Students: None
Presentation Link:
https://drive.google.com/file/d/1Z2H_FXM7ViGIDVOmaozgDngUAEcA0D1E/view?usp=sharing

Project Description

Initial Goals

With the recent Wordle craze, many similar games began rising in popularity. One such game is Semantle, where the goal is to guess a hidden word each day. When a user inputs a guess, the game calculates the similarity between the guess and the hidden word, then displays that similarity to the user. Afterwards, the user keeps inputting guesses until the target word is found.

Due to the open-ended nature of Semantle, the game tends to be very difficult for a human to complete. Meanwhile, the game presents interesting challenges for a project in artificial intelligence. The flow of the game is simple: the game takes in simple input (the user's guessed word) and gives simple output (a similarity score to the target word). This discrete input and output makes Semantle a game that is very fitting for building an AI that can play it. Calculating the similarity of two words is as simple as computing the corresponding word vectors' cosine similarity. Additionally, the creators of Semantle are very transparent about the inner workings of the game. Information about the implementation of the game, as well as the dataset used for word vectors, are both available at the game's website.

At a high level, our original goal was to create an AI that could solve Semantle for any word in a three million word dataset within a reasonable amount of time. From the start, we knew that we needed a reinforcement learning algorithm that could handle extremely large data spaces. We found Q-learning to be the best option for this. [explain Q-learning more here] However, the main problem with Q-learning that we aimed to tackle was the size of the action space. If each word is its own action, then the size of the action space would be extremely large. One of the first goals of the project was to find ways to make the action space smaller so implementing Q-learning was more feasible.

Q-Learning

From the beginning of the project, we were directed towards Q-learning as the main reinforcement learning algorithm that would drive the higher level of narrowing down potential words to guess. Q-learning is often used for games like this, where there is a well defined environment with clear actions and the AI gains information about the game which is described as the current state. Although Q-learning in theory could capture the full relationships between all words and given all previous words guessed in some state representation be able to pick a single word to play. This was something that started as an initial naive assumption that if we have the right Q-table and state representation, the Q-learning could play the game on its own. We quickly realized its impracticality for reasons described in the state representation and q-table size section. However, we did briefly revisit the idea as we thought about the tradeoffs of adding additional q-layers and layers of word clusters later on.

The advantages of Q-learning were obvious from the start. We were not playing a complete information game like Chess or Go where it is possible to look several moves ahead and make strong assumptions about the value and likelihood of those states. Thus we needed a reinforcement learning algorithm that looked at the current state of information and decided what action to take based on what it knows so far. Semantle is a game about using past information to predict the future and more knowledge is gained with every move in the game. However, future moves must still be accounted for, so the highest expected future reward in the update of the Q-table is a good way to estimate that in an incomplete information game.

At first, we believed that if we could master the state representation, then we could make each word an action and the Q-learning agent could directly decide which word to guess. This was naive and a good representation of our lack of knowledge about Q-learning going into this project because we didn’t realize the curse of dimensionality that would later plague and restrict us in terms of both what we chose as state representations and actions.

Regardless of the state representation that we decided, we knew that there would be a huge exploration vs exploitation trade off in our AI as it tried to choose between words it knew might be good and words it has no information about but could be the mystery word and can not go overlooked. Q-learning almost always incorporates an exploitation exploration trade off and for our function we went with an epsilon greedy approach that would ensure that given enough time, every state should be eventually explored. The results of this state exploration can be found in the Epsilon-Decay subsection.

Additionally, we wanted a reward driven algorithm. This is because Semantle gives the similarity score to the mystery word as a float from approximately 0 to 100. It was our belief that the similarity of the word was a perfect representation of the reward and Q-learning would adapt and get increasingly higher similarity scores because it wants to maximize its reward. Although this is not entirely inaccurate and a different implementation might use the similarity score as a very effective reward, this was an inaccurate way of judging the actions of the q-agent because it only operated at the top level. We will go further into detail on this limitation in the reward function subsection.

Clustering and Q-agent Hierarchy

It became clear early on that Q-learning could not reach the precision level of single word guesses, so instead it was going to have to generalize to a group of words. In order for the group of words, or cluster, to be useful, the word would need to be semantically similar to if a very good word is guessed in a cluster, the mystery word is likely also in that cluster. We already had the pre-trained Word2Vec model used by Semantle, so the words already had embeddings ready to be clustered. However, there was no clear way of supervised clustering because we knew very little about the semantic relationship between words. Therefore we resorted to unsupervised clustering, which suited our purpose well, but required trial and error to create closely sized clusters.

The first unsupervised clustering algorithm we went with was K-means clustering. This was a good first start, but we found that the clusters created through this process did not capture the semantic similarity of the Semantle game. Thus we needed something that could better use the cosine similarity to cluster. Through multiple cluster method experimentation, it was found that Agglomerative hierarchical clustering with cosine distances and complete linkage created the best clusters of nearly equal sizes and highly semantically similar. Agglomerative is a bottom up approach where each word starts as its own cluster, and continuity combines with nearby clusters to create increasingly larger clusters until the number of clusters is reduced from the number of words to the desired number of clusters.

Due to Q-learning's curse of dimensionality, mentioned more in the Q-table size section, we could only have 4 clusters at each level. Originally, we only had one level of clustering which did not sufficiently reduce the number of potential words our heuristic function would have to evaluate. We then tried up to 3 layers of clustering each cluster subdividing in 4 at each level. Because Q-learning learns slower on each subsequent level, the sweet spot was 2 levels of clustering to reduce the number of words by a factor of 1/16.

We then needed Q-agents to make decisions at each level, thus leading to the 5 agents, one at the top level to choose a sub-agent, and then 4 at the sub level to choose a sub-cluster. Our Q-learning system is structured with two levels of Q-agents, each contributing to different aspects of the decision-making process. At the top level, the primary Q-agent selects a sub-agent from four options, each responsible for a unique set of subclusters of words. This hierarchical structure effectively divides the problem space into smaller, more manageable segments. Once a sub-agent is chosen, it then takes over, deciding on the best subcluster to pick a word from. This two-tiered structure allows our system to manage the complexity and dimensionality of the game by breaking down decisions into higher-level cluster selection and lower-level subcluster selection within that cluster. We did keep track of the original 4 clusters however, in order to check for the mystery word in the cluster to reward the top agent properly and not penalize the top agent for the sub-agent choosing the wrong sub-cluster.

Environment Representation

The environment in Q-learning is the space in which the Q-agent will try to maximize their rewards and create policies to make decisions based on the current state. The environment serves to update the state of the Q-agent whenever it guesses a word and should also provide the reward for the Q-agent based on the action that they just took. One environment supports all 5 Q-agents in this implementation and they all receive their rewards from the same reward function that propagates to both levels of Q-tables. The environment gets initialized with the mystery word for the first game and all the clusters and subclusters get stored in the environment. The environment keeps track of the previously guessed words for the round which is all the information the whole AI has to work on to guess a word in that round. This is stored as a list of tuples of word, similarity score and sorted so that the word with the highest similarity score is at index 0 and the worst similarity score word is at index -1 or length of guessed words - 1. Additionally, the environment stores 5 states for each of the 5 Q-agents so that it can appropriately update the correct state for the right Q-agent when an action is taken and also give the state to the Q-agent to make their action decision.

State Representation and Q-Table Size

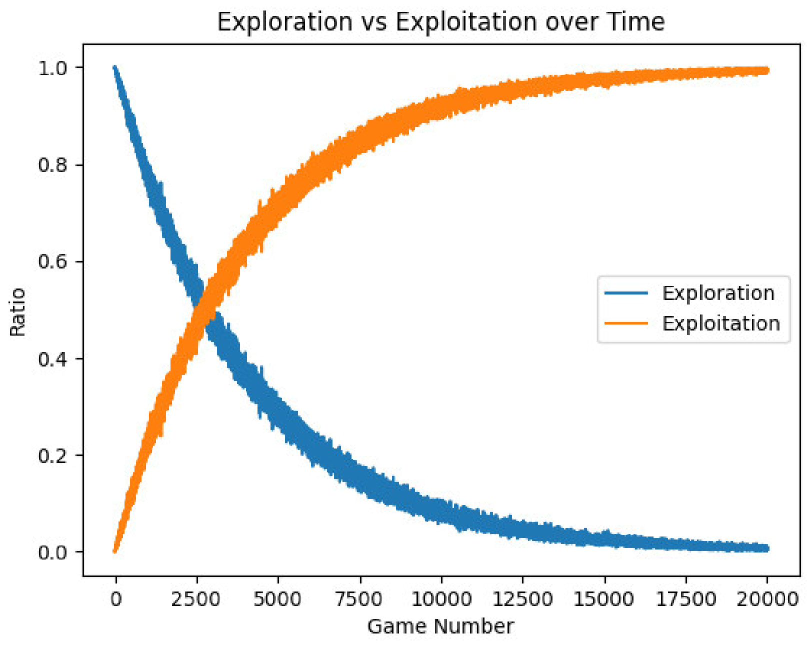
The state representation was one of the most important decisions in the Q-learning process because of the curse of dimensionality. The Q-table size grows exponentially with the number of states and actions, so we had to come up with an efficient state representation that could fully capture what the Q-agent knows about the game at that time, while generalizing in a way that allows the same state to be reached multiple times. It was clear that the state had to keep track of how good or bad the words guessed from taking an action were, but we couldn't use the raw similarity score. The raw similarity score is a unique float value that is never the same for two different words. If we keep track of the state as a list of the last similarity scores from each action, we would never repeat the same state twice and the number of states would be nearly infinite.

To solve this infinite state problem and reduce it to a reasonable finite number of states, we resorted to binned similarity scores. Instead of the raw similarity score, we would instead assign ranges of similarity scores the values 0 to 4 where 4 represents the similarity scores of the closest words and 0 is the absolute worst words. This created 5 bins of similarity scores. The state then becomes a list of 4 integers from 0 to 5 where the 4 integers correspond to the last binned similarity score from taking each of the 4 actions available to a Q-agent. This reduced the number of states in the game to 5^4 or 625 states which gets multiplied by the number of actions, 4, to determine the number of Q-values in the Q-table, 2500.

The equation for the number of entries in the Q-table is $x^y y$ where x is the number of bins and y is the number of clusters. This is the main reason why we couldn't increase the number of bins or clusters at each level without drastically increasing the Q-table size. Just one more cluster at each level would increase the Q-table from 2500 entries to 15625 entries and just one more bin would increase it to 5184 entries. Therefore, given the curse of dimensionality, the state representation that we chose we still believe to be the best trade off between known information and limiting the size of the Q-table enough so that effective policies can be learned.

Epsilon-Decay

Once the number of states was reduced to a reasonable size, we needed to ensure that all states had a chance to be explored throughout the game and that the Q-learning wouldn't get stuck in local maxima having only explored a subset of states. Epsilon-Decay was the perfect way to balance this exploration exploitation trade off. We start at a very high value of epsilon and explore with a probability of epsilon so we highly explore at the beginning of the learning process. As games progress, we decay epsilon exponentially relative to the game number. This will encourage more exploitation of the information in the Q-tables as the information gets better. The crossover point is approximately 2500 games as seen in the graph below. Through testing, this point is approximately when enough states had been explored that the performance wouldn't improve through more exploration. Although it decays exponentially, it never reaches zero, always leaving a chance of new states being explored and the probability of all states being explored only increases with time.



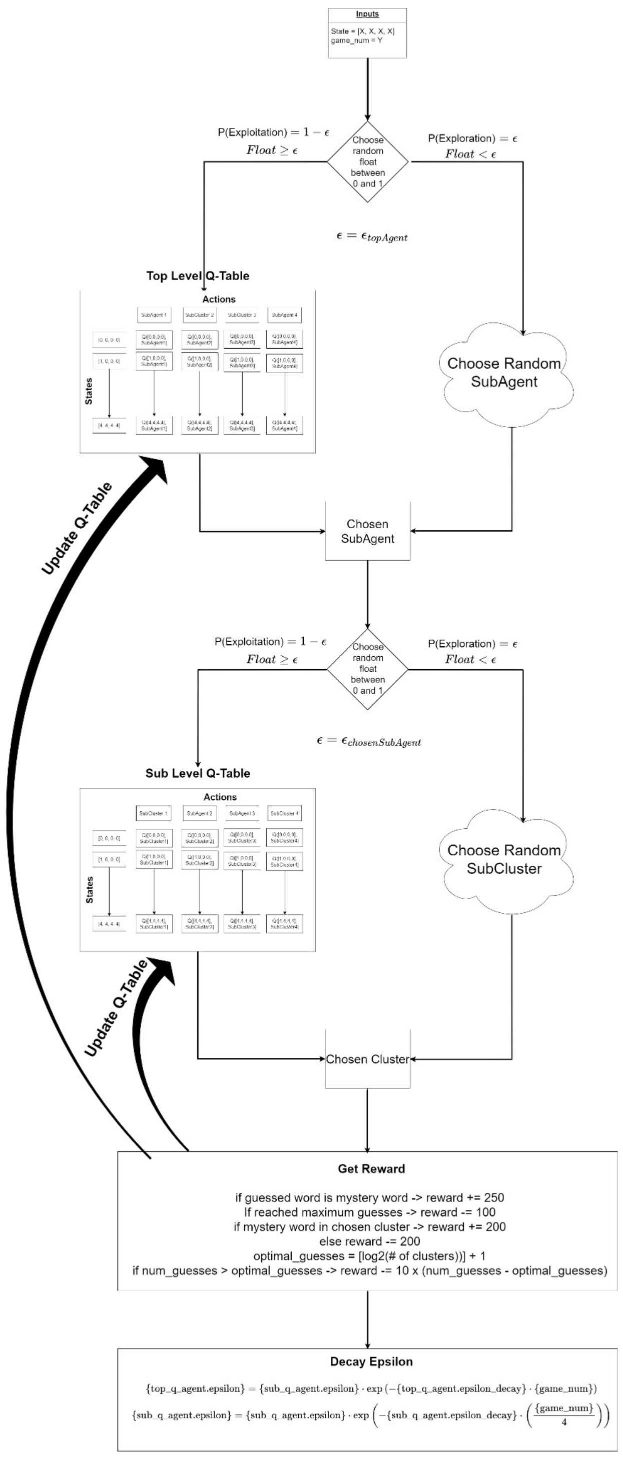
Epsilon Decay has to be considered at each level of Q-learning to ensure that every Q-agent has a chance to fully explore. Therefore, each Q-agent keeps track of its own epsilon value, and we only decay the epsilon value of sub-Q agents when they take an action and have a chance to explore a new state. We also decay the sub-q-agents at a rate of the game number divided by 4 to account for the fact it only gets chosen about ¼ the amount as the top agent makes decisions. Thus our top agent will explore faster while the sub-agents explore when they are given the opportunity. You can see how successful this epsilon decay is at exploring new states in the evaluation section.

Reward Function

The reward function was a major setback that we didn't see for a while when working on the Q-learning agent. At first, we were rewarding the Q-agent almost strictly based on the similarity score of the word chosen. This seemed like a reasonable approach as the similarity score was directly proportional to how good of a guess the word is. When this reward function wasn't guiding our Q-agents properly, we thought it required additional factors to moderate and add to the reward. Thus we increased the number of parameters in the reward function to include penalties for duplicate guesses, a reward equal to double the similarity score, a penalty for reaching the maximum guesses, and more. This even further misled the Q-agent, but we were unaware of it until we came to a realization.

The realization we had to come to in order to reach the optimal reward function was that the Q-agent doesn't pick the word so its reward should have nothing to do with the similarity score of the word. Rather it chooses a cluster, so we need to reward it based on if the cluster chosen contains the mystery word or not. This is exactly what we went with, and the simplified reward function doubled the performance of the Q-agent. The previous reward function had it select the cluster with the mystery word about 27% of the time, just slightly better than the 25% you would expect from random decisions. The new reward function allowed the Q-agent to choose the cluster with the mystery word 50-60% of the time including early guesses with limited information.

Overall Structure



Inputs
State = [X, X, X, X]
game_num = Y
Choose random float between 0 and 1
P(Exploitation) = $1 - \epsilon$
$Float \geq \epsilon$
P(Exploration) = ϵ
$Float < \epsilon$
$\epsilon = \epsilon_{topAgent}$
Top Level Q-Table
Actions
States
Choose Random SubAgent
Chosen SubAgent
Update Q-Table
$\epsilon = \epsilon_{chosenSubAgent}$
Sub Level Q-Table
Choose Random SubCluster
Chosen Cluster
Update Q-Table
Get Reward
if guessed word is mystery word -> reward += 250
If reached maximum guesses -> reward -= 100
if mystery word in chosen cluster -> reward += 200
else reward -= 200
optimal_guesses = [log2(# of clusters))] + 1
if num_guesses > optimal_guesses -> reward -= 10 x (num_guesses - optimal_guesses)
Decay Epsilon
$\{\text{top\_q\_agent.epsilon}\} = \{\text{sub\_q\_agent.epsilon}\} \cdot \exp\left(-\{\text{top\_q\_agent.epsilon\_decay}\} \cdot \{\text{game\_num}\}\right)$
$\{\text{sub\_q\_agent.epsilon}\} = \{\text{sub\_q\_agent.epsilon}\} \cdot \exp\left(-\{\text{sub\_q\_agent.epsilon\_decay}\} \cdot \left(\frac{\{\text{game\_num}\}}{4}\right)\right)$

This diagram shows the overall structure of the Q-learning implementation in this AI. It starts with the inputs to the Q-learning, which are the current state and the current game number. We then move on to step 1, picking a sub-Q agent. It chooses a number between 0 and 1 to decide if it will exploit or explore. If it exploits, it picks the sub-Q agent with the highest Q-value based on the state, otherwise it picks a random Q-agent. We have now chosen a sub-Q agent that handles 4 out of 16 of the subclusters of words.

This moves on to the second step: choosing a subcluster of words. We then pass the same inputs to the sub-Q agent who again decides to exploit or explore. If it exploits, it will choose the subcluster with the highest Q-value based on the current state, otherwise it chooses a random subcluster. This choice of subcluster then gets passed on to our heuristic function described later.

After we guess the word, we move on to the third and final step, the update step. This is where we decay epsilon for the two Q-agents involved and get the reward to update their respective Q-tables. Q-learning makes updates based on several factors: the current Q-value, the learning rate, the reward, and the maximum future Q-value moderated by a discount factor. These hyperparameters were heavily adjusted and experimented with to extract the best performance and policies from the Q-learning implementation.

Heuristic Functions

Once the AI determines which cluster to pick from, it must choose a word from that cluster to actually use as a guess. To do this, we tried various methods.

Greedy Approach

Our most successful heuristic involves a greedy approach. Throughout the game, the AI keeps track of the previously guessed words, as well as their similarity scores. When choosing a word from a cluster, the word with the highest similarity score (the "best word") and the word with the lowest similarity score (the "worst word") are obtained. Next, an intermediate vector is calculated by subtracting the worst word's vector from the best word's vector. Finally, the vector in the cluster with the closest similarity to this calculated vector is obtained and returned.

We also experimented with a different approach if the best previously guessed word wasn't very close to the target. Presumably, if this best guess isn't very close to the target, we wouldn't be sure it's on track to be a correct guess. Therefore, if the best word's similarity is below a certain threshold, the AI would instead calculate the intermediate vector by finding the element-wise average between the current and target vectors. This approach performed much worse, cutting the win rate roughly in half.

RNN Approach

Overview

While the greedy approach worked well, it often got stuck in local minima, leading to it not being able to guess the correct word within the maximum amount of guesses. Due to this, we attempted an approach involving a neural network. Given an input of a certain number of words representing previous guesses and their corresponding scores, as well as a word representing the next guess, we want to guess the similarity score of the next word.

Due to the recursive nature of the input, an RNN was a natural first approach. Figure 1 depicts the general design of the model. Each previously guessed vector is concatenated with its score, and the next word vector is concatenated with 0 to fit within the same amount of dimensions. Next, each vector-score concatenation is passed into the RNN in succession. Once the next word vector is passed in, the output is used as the predicted similarity score. The diagram depicts two hidden layers being used, but various amounts of hidden layers were experimented with.

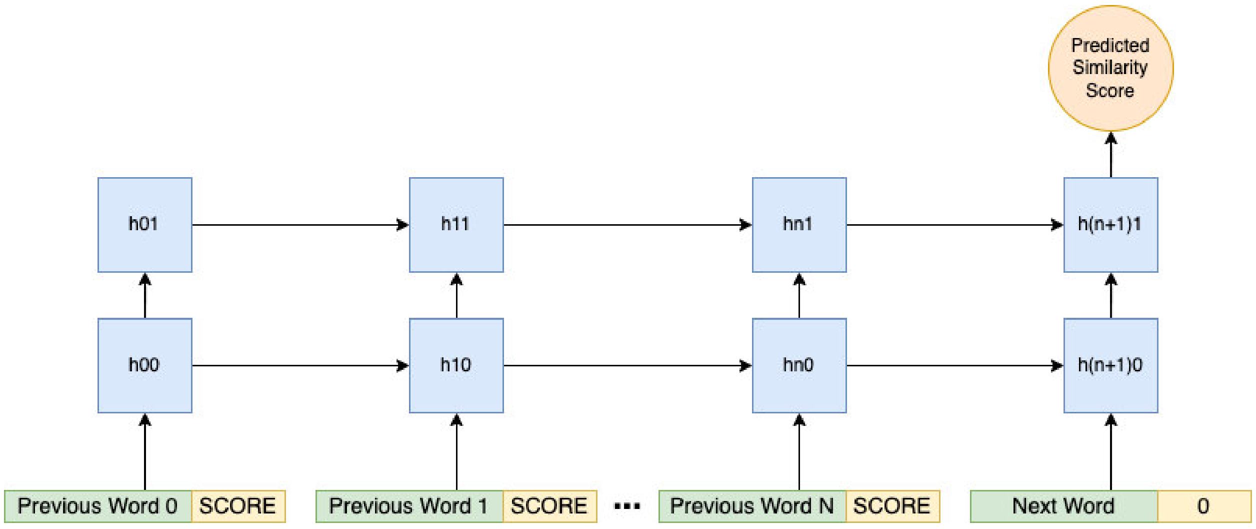


Figure 1: Diagram representing the RNN heuristic.

When incorporating the RNN into the algorithm, we realized that running the RNN on each word in a cluster would be too slow. Therefore, we combined this RNN approach with the greedy approach. First, the AI would take the five closest words to the best guessed word. Next, the AI would feed each word into the RNN and return the word with the highest predicted score.

Training

Two methods to train the RNN were attempted. First, we tried a more traditional approach, randomly generating samples of data with 5 previously guessed vectors, a vector to guess next, and a target word. The RNN was trained on this data with a fixed amount of epochs.

Next, we tried training our RNN in-game. After each round of guesses, the RNN would compare the predicted scores of the five vectors with their actual similarities to the target word. It would then use this loss to backpropagate, then repeat.

Results

Overall, our resulting project ended up being close to our original vision, and we ended up meeting most of our original goals. However, there is one goal that we never achieved which was to reach the full 3 million word dataset. The number of Q-layers and clustering needed combined with the curse of dimensionality problem meant that we knew that no computers that we had access to could feasibly train this 3 million word implementation. Thus, we always stuck to the 4,200 set of possible answer words which span the whole vector space and thus give a good representation of performance amongst the larger wordset. Additionally, there is little reason to think that besides time and required computing power, our current implementation couldn't scale to the full 3 million word data set, and thus a theoretically scalable solution is the best we were able to accomplish.

Evaluation

Goals

When evaluating our AI, we wanted to answer two main questions. First, how often can our AI solve games? It's important that a Semantle AI is actually able to solve games within a reasonable amount of guesses. Next, how fast can our AI make guesses and complete games? We needed to make sure that our AI didn't take too long solving games. Finally, how well does our AI learn? We need to make sure that our AI explores the action space efficiently.

Our original goals for the AI were to be able to solve any game, and to be able to do so while completing each game in under a minute.

Methods

Our evaluation methods can be seen in the file qlearning.ipynb. We split our evaluation into two different areas: evaluating the greedy heuristic and evaluating the RNN heuristic.

To evaluate how often our AI can solve games, we first chose a limit to the number of guesses per game. In this case, we set a limit of 50 guesses. We then ran a series of 3,000 games and tracked the win percentage for each interval of 100 games. To evaluate how fast our AI can solve games, we simply timed this 3,000 game interval for the AI using both heuristics.

Results

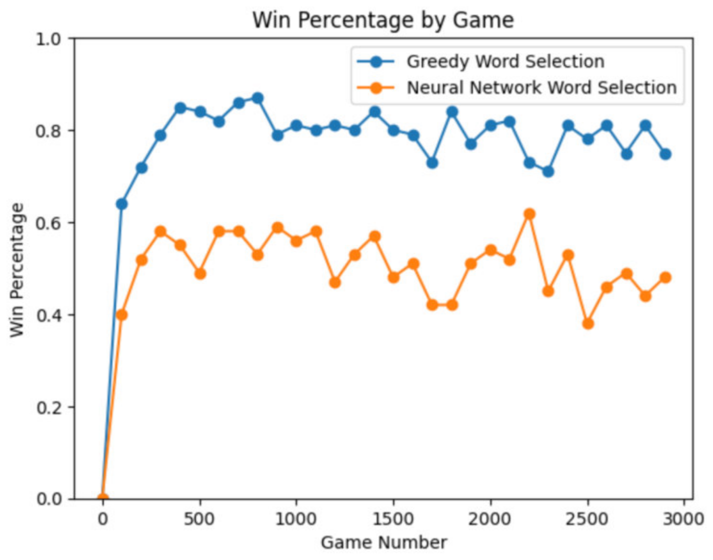


Figure 2: Graph representing the accuracy over time with different word selection techniques.

Figure 2 displays the win percentage per game over time of our AI using both heuristics. Overall, the AI using the greedy heuristic was able to solve 79.2% of games within 50 guesses, while the AI using the RNN heuristic was able to solve just 50.1% of games within 50 guesses. Clearly, the greedy heuristic showed much better results. While we weren't able to reach our goal of being able to solve any game within this interval, the AI is still able to solve most games within the maximum amount of guesses.

In terms of time, the AI with the greedy heuristic was able to play games in an average of 0.068 seconds and make a guess in an average of 0.002 seconds, while the AI with the RNN heuristic was able to play games in an average of 0.204 seconds and make a guess in an average of 0.005 seconds. While both versions of the AI clearly surpass our original goal of solving games in under a minute, the greedy heuristic AI was able to solve games much faster than the RNN heuristic AI.

Figure 3 displays how successful we were at solving the exploration vs exploitation tradeoff problem and at creating a reasonable sized Q-table that can effectively learn policies. What we see is the top agent reaching almost 600/625 states by game 10,000, and even then, still hasn't stopped exploring. The bottom four lines are the sub-Q agents and their lines are more piecewise because each sub-agent only decays their epsilon when they make a decision, about ¼ moves. Thus despite the graph not showing the subagents reaching the asymptote, we would expect this to happen with about four times as many games required as for the top agent. The graph is not entirely perfect, though, as we would hope to see the sub-agents with a higher slope of exploration in the second half of the graph as they still have over 200 states to explore. While they will still always explore more states with more games as epsilon never reaches 0, this could be too slow in practice to ensure reaching all states. There may exist a better mathematical equivalent than our

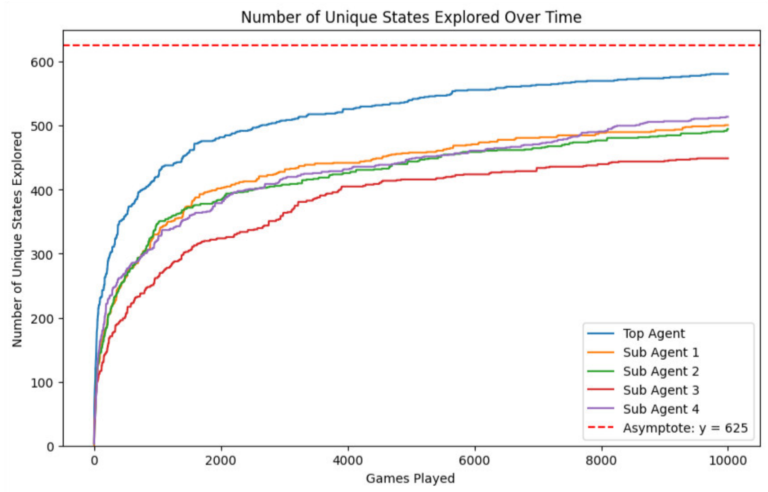


Figure 3: Graph representing unique states explored over time of each of the 5 Q-agents and their respective 625 states to explore, represented by the vertical asymptote. Epsilon decay = .00025

implementation to ensure that the sub-agents explore the same amount as the top agent regardless of how often it is chosen to perform an action.

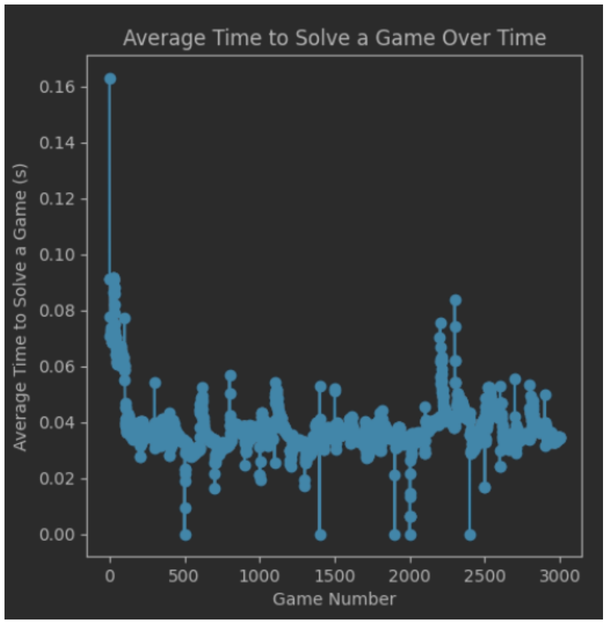


Figure 4: Graph showing Average time to solve a game vs time for 3,000 games and

Our original goal of 1 minute to solve a game was way overestimated. In Figure 4 (left), you see that the time to solve a game starts relatively high at the beginning, but even its slowest game is .16 seconds. It quickly settles out after about 500 games to its average around .04 seconds. The games with times near 0 are not errors but rather times when the first or second word guessed was the mystery word which is bound to happen given a large enough sample size. Outliers are likely rounds when all moves had to be taken and the largest clusters were chosen as many functions like our greedy heuristic have time complexity proportional to the number of words in the cluster.

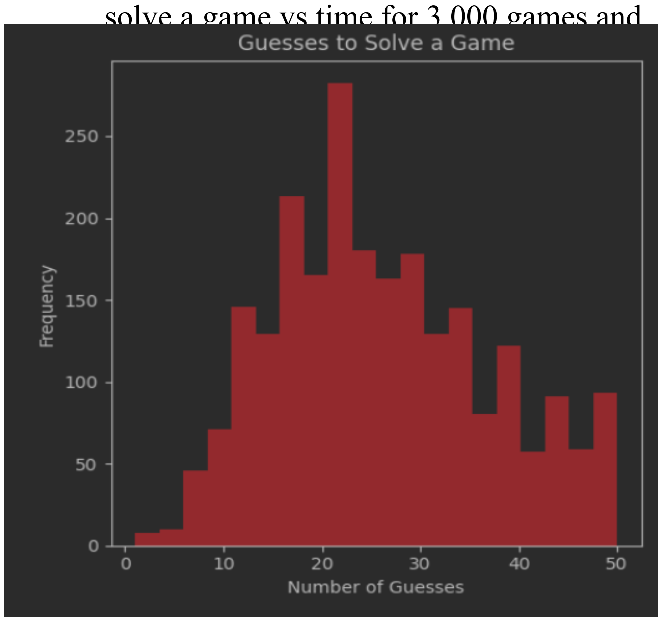


Figure 5: Graph depicting the frequency of each amount of guesses per game.

We were also successful in our goal of winning the game in a reasonable amount of guesses. When given 50 guesses, successful games were able to come up with a solution in about 22 guesses. As shown in Figure 5, the frequencies of each amount of guesses roughly follows a normal distribution. While this distribution is slightly skewed right, we still don't see a large spike at 50 moves meaning it doesn't often need to use all of its moves to gain enough information to solve the game. There are no publicly available statistics for average moves to solve Semantle, but in our anecdotal experiences, it is not uncommon to take 40, 50 or more moves thus this is very often performing far better than the project creators can solve the game, which was a big milestone for us while building the AI.

Figure 6
Furthermore, when we were considering speed, we also wanted to make sure that we were having our guesses done quickly. This is separately important from a fast game because we won't always be able to get the word in 5 guesses, or 10. Sometimes it took up to 500. However, with a fast guessing system we know that regardless of how many guesses are needed, our system will be able to solve quickly. As we see in this graph, we start off a bit high with 0.0025 seconds but quickly settle down around 0.0015 seconds per guess.

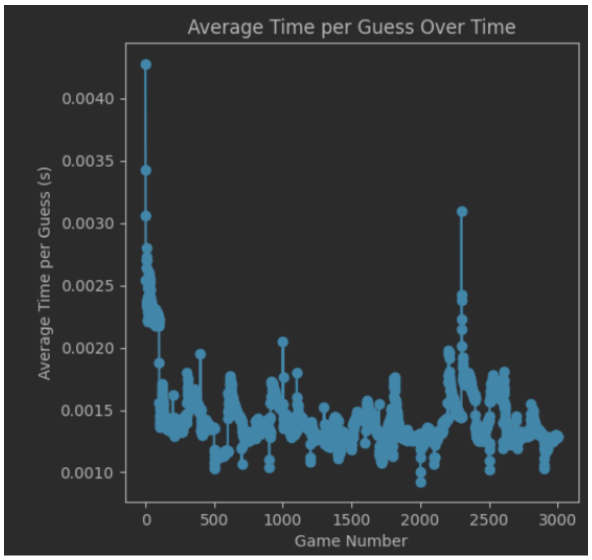


Figure 6: Graph showing Average Guesses vs time for 3,000 games and 50 moves/game

References

1. Turner, David. *Semantle*. https://semantle.com/