# Utilizing REINFORCE to solve Wordle

**Jeongmu (Daniel) Hahn (jdh6372)**
University of Texas at Austin
Austin, TX
jeongmu.hahn@utexas.edu

**Ryan Chhong (rc47259)**
University of Texas at Austin
Austin, TX
ryanchhong@utexas.edu

**Abstract:** Applying deep reinforcement learning to games has been an extremely common task. The introduction of Deep Q-Learning by Mnih et al. [1] showed that the usage of artificial neural networks were viable in the domain of reinforcement learning. Since then, a variety of methods and techniques have been invented based off this idea of utilizing a neural network as a non-linear function approximator. In this work, we try to apply some of these methods and techniques to create an agent capable of playing "Wordle." The game of Wordle was an interesting problem, mainly due to the massive size of the action space. In order to overcome the large action space, we combine reinforcement learning techniques with techniques from natural language processing in order to train an agent to play Wordle.

**Keywords:** NLP, Reinforcement Learning

Source Code: https://github.com/rchhong/cs394r-final-project
YouTube URL: https://youtu.be/8V6ZE1kT6Hg

## 1 Introduction

### 1.1 The Wordle Environment

Wordle is a word-based game created by John Wardle. The game's popularity has exploded in recent months, becoming a viral phenomenon. Recently, the game was acquired by the New York Times.

Wordle is a fully observable, single agent, deterministic, episodic, static, discrete, known environment. The goal of the game is to determine a predetermined word that changes daily. This word will be referred to as "the solution." The player has a maximum of six guesses to guess the solution. Upon guessing a word, the game returns various information about each of the letters in the guess. The color encoding is as follows:

- Green tiles indicate that the letter is both present in the solution and is in the correct position.
- Yellow tiles indicate that the letter is present in the solution, but is not in the correct position.
- Grey tiles indicate that the letter is not present in the solution.

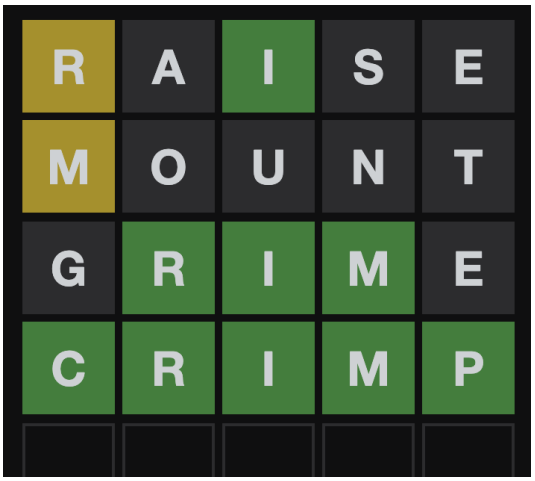



Figure 1: An example game of Wordle

## 2 Related Works

As stated in the introduction, deep reinforcement learning methods have been commonly applied to various types of games. One of the first works exploring this combination was conducted by Mnih et al. [1] in 2016. The methods utilized to learn how to play Atari were based off of Q-Learning, a relatively simple off-policy reinforcement learning algorithm. The approach we utilized in this paper also involves the usage of deep neural networks. However, there are many problems with this approach. Although Q-learning is shown to converge to a global optimum, the algorithm in practice does not tend to perform well sometimes. The approach that we takes utilizes a more sophisticated method to learn how optimal play that moves away from the traditional action-value methods, such as Q-Learning.

The approach that our agent utilizes to learn how to play optimally is similar to the approach outlined in a paper by Sutton et al. [2]. In essence, we utilize policy gradient methods instead of the traditional action value methods utilized in previous work. The agent utilized the one of the simplest policy gradient algorithms: REINFORCE. Additionally, our agent utilizes a learned baseline in order to speed up the rate at which the agent learns the optimal policy. The learned baseline in our case is the estimated state value for the current state. A deep neural network is also utilized to learn to predict this value. One of the main disadvantages of this approach is the fact that the REINFORCE update is a Monte-Carlo update, meaning that termination of episodes is necessary in order to facilitate network updates. However, due to the extremely short maximum possible length of episodes in Wordle, we did not find this to be an issue. A combination of the algorithm's simplicity and surprising performance led us to choose this algorithm to form the basis of our agent.

However, one of the main issues of the previous method is the fact that it requires great computational power in order to store the replay buffer utilized to train the agent. There are many more sophisticated and robust policy gradient methods proposed in recent years. One of the simplest advancements to the REINFORCE algorithm was to introduce bootstrapping to policy gradient methods. Actor-Critic methods bring this improvement to the REINFORCE algorithm. However, a more important development is the asynchronous version of this algorithm, outlined in a paper by Mnih et al. [3]. One of the advantages of following the approach outlined in this paper is the fact that the amount computational power required is greatly decreased, as the author's of the paper were able to train their agent on similar tasks found in Mnih et al. [1] with significantly less computational power. Although we tried this approach with our agent, we found that the positive bias in the updates caused a large amount of instability, and our agent was unable to learn anything. Because of this, we decided to utilize the simpler REINFORCE approach.

There are many more extensions to the algorithms listed in the previous paragraph. One of the main disadvantages to both REINFORCE and Actor-Critic methods is the fact that a fixed learning rate is generally applied when utilizing these algorithms. However, finding a learning rate suitable for the given problem can be quite difficult. However, methods such as Trust Region Policy Optimization (TRPO) are able to solve this problem. Proposed by Schulman et al. [4], the TRPO algorithm is able to maximize the magnitude of parameter update without overshooting the optima by selecting an optimal set of parameters within a region. Proximal Policy Optimization (PPO) is an approximation of this method, which was proposed by Schulman et al. [5]. These two algorithms for the current state of the art for on-policy agents. Although these algorithms would be able to perform better and learn faster than our agent, we found that the REINFORCE algorithm was sufficient to learn near-optimal performance.

As for the off-policy case, Soft Actor-Critic (SAC) and Deep Deterministic Policy Gradient (DDPG) form the current state of the art algorithms. Soft Actor-Critic, proposed in Haarnoja et al. [6] adds entropy regularization to the optimization problem in order to encourage greater exploration and prevent premature convergence. Our agent also utilizes this trick for the same purposes as Soft Actor-Critic. Finally, Deep Deterministic Policy Gradient (DDPG) combines Actor-Critic methods and Q-learning to produce an algorithm that learns both a value function and a policy, but is able to work in off-policy settings.

The idea of applying neural networks and reinforcement learning together has exploded in the past couple of years. A survey of these methods and their applications can be found in a work by Shao et al. [7]. Although these papers dive more in depth and utilize more complex reinforcement learning algorithms and neural network architectures to solve increasingly complex games, none of the approaches combine these algorithms with natural language processing techniques. Due to the unique nature of our game being a word game, the agent we created utilizes techniques from both of these domains in order to learn to play Wordle optimally.

Although Wordle is a very novel game, there has been some work exploring the optimality of the game. A work by Anderson and Meyer [8] explores finding an optimal strategy that humans can utilize to play the game. The approach we utilize is significantly different from the approach outlined in the paper, as we wish to create an optimal solver, not an optimal strategy for humans. However, we will utilize the statistics produced in the paper to form a baseline for optimal human play. Another work by 3Blue1Brown [9] utilized information theory in order to write an optimal solver for Wordle. The approach they took involved utilizing a greedy algorithm to select words that would maximize the entropy of the probability distribution of potential outcomes by selecting that word. The agent that we create does not take this approach, but we will explore whether or not there are parallels between the entropy values for each word and the estimated state value produced by our agent.

## 3 Methods

### 3.1 Policy Gradient Methods

Policy Gradient methods provide an alternative to the traditional function approximation methods utilized in reinforcement learning. Policy Gradients can be applied to reinforcement learning problems to make agents learn to act optimally in a manner similar to gradient descent methods utilized to train neural networks. Policy Gradients methods parameterize the policy utilizing a vector of parameters $\boldsymbol{\theta}$ and a performance measure $J(\boldsymbol{\theta})$. The basic update for policy gradient methods is as follows.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla(\boldsymbol{\theta}_t)} \tag{1}$$

Where $\widehat{\nabla(\boldsymbol{\theta}_t)}$ is an unbiased estimator of $\nabla J(\boldsymbol{\theta}_t)$ and $\alpha$ is the learning rate. In general, $J(\boldsymbol{\theta}_t)$ in the episodic case is the value of the initial state $s_0$ following the policy parameterized by $\boldsymbol{\theta}$ ( $\pi_{\boldsymbol{\theta}}$).

### 3.2 REINFORCE and the Policy Gradient Theorem

Finding $\nabla(\boldsymbol{\theta}_t)$ would be quite difficult in general, as it is hard to determine how to perturb $\boldsymbol{\theta}$ in a manner that would guarantee improvement. However, the Policy Gradient Theorem makes this quite easy. The Policy Gradient Theorem for the undiscounted case is stated below, where $\mu(s)$ is the standing state distribution (i.e. the proportion of time spent in state $s$).

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi_{\boldsymbol{\theta}}(a|s,\boldsymbol{\theta}) \tag{2}$$

The Policy Gradient Theorem gives rise to REINFORCE, which utilizes Monte-Carlo methods in order to update the parameters. The REINFORCE update (with baseline) is displayed below. As long as the baseline is solely dependent on the state or a constant, the estimator of the gradient remains unbiased.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(G_t - b(S_t)) \nabla \ln(\pi_{\boldsymbol{\theta}}(a|s,\boldsymbol{\theta})) \tag{3}$$

A finding by Mnih et al. [3] found that adding the entropy of the probability distribution created by the actor to the update above encourages exploration. Therefore, the final update rule that will be utilized with our agent is as follows.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left[(G_t - b(S_t)) \nabla \ln(\pi_{\boldsymbol{\theta}}(a|s,\boldsymbol{\theta}) + \beta_{entropy} H(\pi(\cdot|s))\right] \tag{4}$$

In our case, the baseline will be the value of the state. This quantity will be approximated utilizing a neural network as well. The update rule for the state value is shown below

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left(G_t - \hat{V}(S_{t,t})\right) \tag{5}$$

It is important to note that our use case is in the discounted domain. Therefore, the REINFORCE update rule shown above is actually incorrect. However, in our experiments, we found that this did not matter and the agent was still able to learn to play optimally.

### 3.3 Word Embeddings

Word embeddings are a technique proposed by Bengio et al. [10] in 2003. The curse of dimensionality is a common issue in Natural Language Processing. In the context of sentences, it is quite unlikely that the sequence passed into the model is the same to any sequence of words will be the same as any sequence that the model is trained on. An alternative to the n-gram methods, word embeddings are another way to combat the curse of dimensionality. Word embeddings are a learned feature representation for words whose dimensionality is significantly smaller than the original dimensionality of the problem.

In our case, we utilize word embeddings on the action space in order to significantly reduce the dimensionality of the action space. Instead of outputting an entire word, the agent is trained to output a representation whose dimensionality is the same as the dimensionality of the embedding. Therefore, by computing how similar the action the agent outputs and the embedded representation for each word, we can determine which words were most similar to the action the agent outputs. These similarities scores to create a probability distribution across the words. Much like a basic kernel method, the dot product will be utilized as this similarity metric.

## 4 Experiments and Results

### 4.1 Environment Details

For testing purposes, smaller versions of the Wordle environment were utilized at first before scaling up to the final version. 100-word Wordle refers to the Wordle problem featuring only 100 possible

valid word actions, where each of the words could be the solution. 1000-word Worlde refers to a similar concept, but with 1000 valid words instead of 100.

The full Wordle problem features around 13000 valid word actions, but only 2300 of these can be the solution to the Wordle.

#### 4.1.1 State and Action Encoding

The current state representation consists of a binary vector of length 422. The first 26 entries in the vector represent whether or not the letter has been tried yet. Then for each position we store information about which letters cannot be in that position, which letters can potentially be in that positions, and which letters must be at that positions. Each of the pieces of information listed is stored as a one-hot vector for each letter. Therefore, for the entire word, this consists of 390 entries (3 * 5 * 26). The final 6 entries are used to indicate which guess we are currently on.

We underwent several iterations prior to settling on this implementation. Prior to one-hot coding everything, we tried to utilize a state representation that encoded the status of each letter as follows: 0 for a letter not present in the word, 1 for a letter in the wrong place in the word, and 2 for a letter in the correct place. However, we found this representation to not learn well. Additionally, the turn number was left as a single number instead of being one-hot encoded. This again did not train well. We hypothesize that this occurs to to the fact that there is not enough of a discrepancy between the apparent value of each state and the numeric value.

As for the actions, the first representation we settled on ended up being our final representation. Each of the words was one-hot encoded to create a vector of length 130. These one-hot vectors were used as our action representation

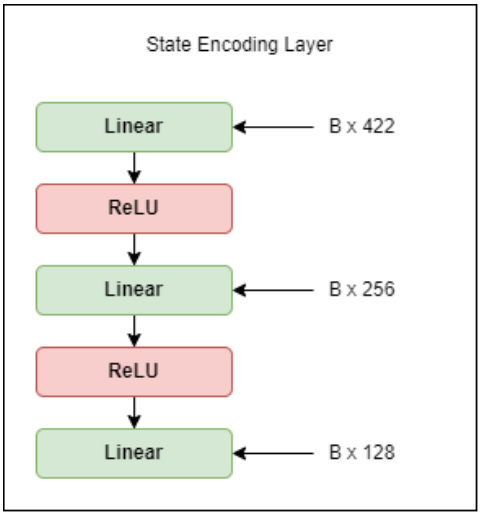
#### 4.1.2 Reward Function

The reward function for our agent is as follows: For each step until termination, a reward of -1 was given. If the agent was able to solve the Wordle before running out of guesses, a fixed reward of 10 was given at the step of termination. Otherwise, if the agent was unable to guess the correct word, a reward of +2 was awarded per green tile, and a reward of -2 was awarded per grey tile. Yellow tiles are ignored and given 0 points.

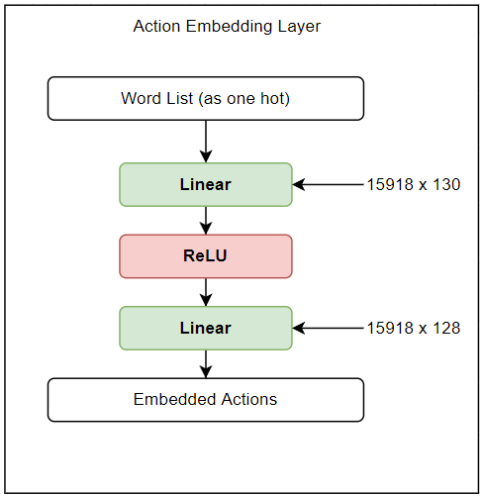
In order to arrive at the reward function above, several iterations of mixing different rewards were tested. Out first reward function rewarded the agent for every step regardless of its termination. On each step, the most recent guess was used to give rewards depending on the number of green tiles. However, when trying this agent, we found that the agent had frozen in progress after getting a few yellow tiles. We believe that this had occurred due to the fact that the different in winning and losing were not divided well. Another reward function attempted was to solely give points simply on winning or losing, with 10 points on winning and -10 points for losing. We found that this reward function was too sparse in giving a win, and is often reached randomly, due to our unique environment. The agent would either stick to the first word it guessed correctly, or not learn at all.

### 4.2 Network Architecture

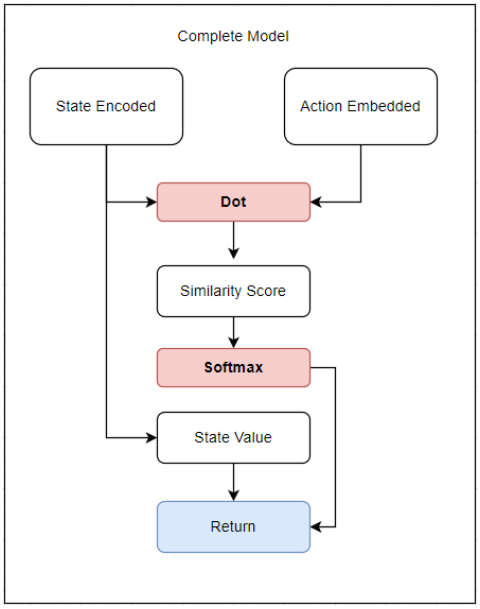
We tried to select an architecture that was as simple as possible for our agent but still able to learn the environment. Prior to the actor and critic layers, the state and possible actions are fed through word embedding layers, which consisted of only Linear and ReLU layers. Both the critic and actor network utilized a single linear layer which worked within this embedding. Although this does not effect the functionality of the critic network, some changes were made to the actor network. Prior to outputting the probability distribution, the embedded action produced by the actor is projected into each of the embeddings for the possible actions. Based of these projections, we utilized a log softmax layer to produce a probability distribution. A diagram showing the network architecture of our agent can be found below.




(a) State Encoding




(b) Action Embeddings




(c) Full Model

Figure 2: Diagrams detailing the network architecture of our agent

### 4.3 Hyperparameter Selection

As for hyperparameter tuning, we tried many different sets of hyperparameters before settling on our final set of hyperparameters. The hyperparameters we tuned the most were the learning rate and the discount factor. We initially tried higher discount factors, but we found that this caused the agent to have a lower win rate overall. We found that this was because the agent was not taking as much risk as necessary to succeed at the game. With so few steps before episode termination, we needed our agent to take more risks. We also found that with higher learning rates, the agent would often end up learning nothing. Another issue we ran into was the fact that the network would suffer from catastrophic forgetting if we trained the network for too long (we found this to occur generally after 10000 epochs of training). The final set of hyperparameters can be found below.

| Hyperparameter | Value |
|---|---|
| Learning Rate ($\alpha$) | 3e-4 |
| Discount Factor ($\gamma$) | .9 |
| Entropy Regularization Weight ($\beta_{entropy}$) | .05 |
| Batch Size | 64 |
| Number of Epochs | 4000 (100-word Wordle)<br>20000 (1000-word Wordle)<br>60000 (Wordle) |

The optimizer utilized was ADAM with $\beta_1$ of .99 and $\beta_2$ of .9. Weight Regularization was also used with $\lambda$ set to .01.

### 4.4 Additional Training Tricks

In addition to the techniques listed above, two tricks were utilized to help train the neural network. One of the tricks we utilized was shifting the distribution of words such that solutions that the agent was not able to reach were weighted higher, and thus, replayed more times. Without this trick, we found that our agent was sometimes unable to solve certain Wordle's no matter how long we trained due to the fact that some combination of letters appeared significantly less often than other combinations. Another trick that we utilized was that with a 5% change, we would allow the agent to "cheat" by inputting an expert action into the agent. In our case, the expert solution was simply the solution to the word itself. We found that this technique also helped the agent win more often. We believe that this is due to the fact that occasionally inputting expert actions helps the agent learn to make conclusions faster than without these expert actions.

### 4.5 Results

To analyze the performance of our agent, we utilized a variety of other agents that could solve Wordle. To be exact, we utilized the entropy-based agents built by 3Blue1Brown, as outlined above. To ensure our agent actually learns to play the game, the performance of our agent will be compared against an agent that simply select a random action each turn. Finally, we will compare the performance of our agent to human baselines established in Anderson and Meyer [8].

Prior to attempting the full problem, we trained our agent on smaller problems. Namely, we tried learning how to play the 100-word and 1000-word Wordle variants. In both cases, our agents were able to learn how to play with win percentages of around 99%. However, upon scaling up to the full problem, various issues occurred which prevented our agent from reaching optimal performance. These issues are detailed in a subsection below Firstly, we will report the win rate of the agents across 10 trials. In each trial, each of the possible hidden words is set as the goal. This results in around 20000 games per agent.

Although our agent was not able to match the performance of the entropy-based agents, it does do better than the human baseline and the random action baseline. Therefore, we can conclude that our agent learns to play Wordle at a level exceeding humans. Although our agent cannot match the performance of the entropy-based agents, it is to be noted that the entropy-based agents utilize knowledge on the solution set in order to make decisions, something that our agent does not do.

Next, we will look at the number of turn it look each agent to arrive at the solution if the agent won the game. Once again, our agent is able to exceed the performance of humans, but is unable to match the performance of the entropy-based agents.

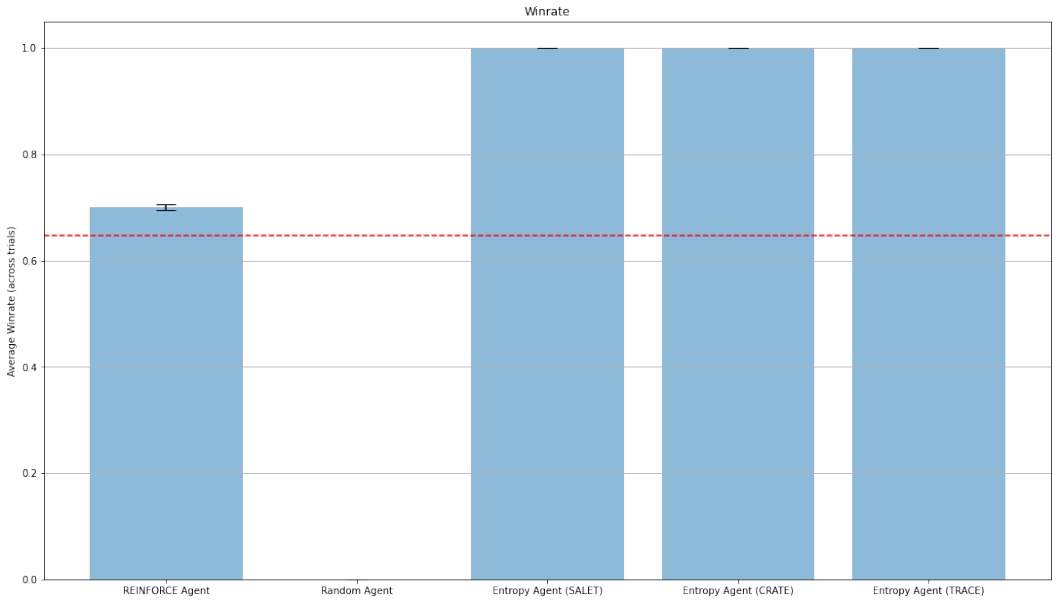



Figure 3: A comparision of Wordle win-rates accross different types of agents. The red-line indicates the human baseline performance

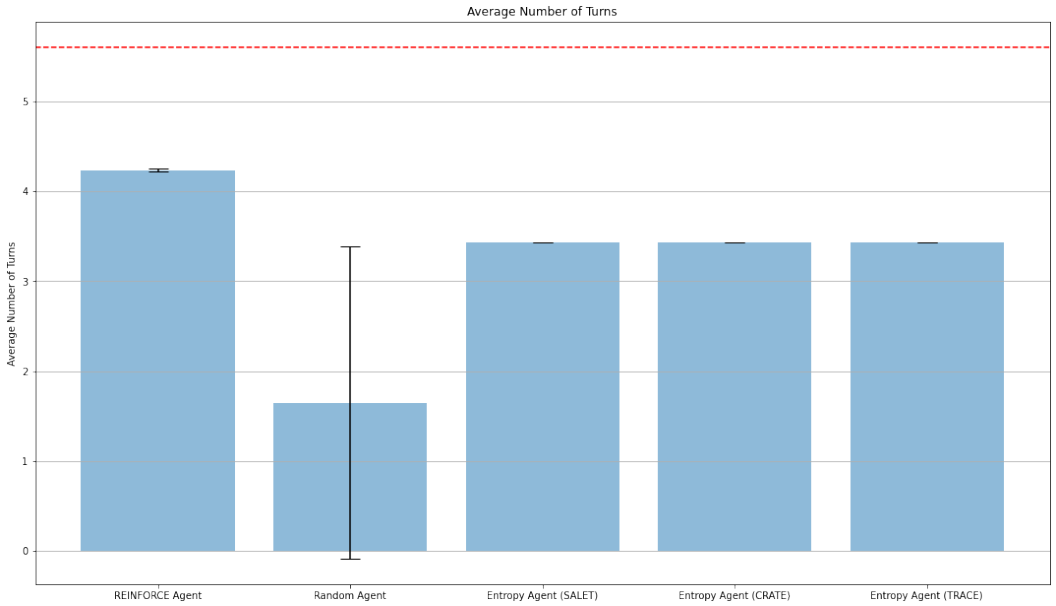



Figure 4: A comparision of Wordle win-rates accross different types of agents. The red-line indicates the human baseline performance

### 4.6 Shortcomings of our approach

Although our agents was able to exceed the performance of humans, we had lots of trouble with the hyperparameter tuning. We believe that with better hyperparameter tuning, our agent would be able to produce better results. Furthermore, computational resources were a large issue. An issue with the algorithm listed above is the fact that these algorithms are generally considered to be very sample inefficient. Therefore, for any significant learning to occur, hundreds of thousands of games were needed to be played. When increasing the total number of possible valid words, we found that this number balloons into the millions. The model itself was also a large issue. Although overfitting

to the problem is actually fine due to the fact that the solution set is fixed, we wanted our model to actually learn how to play the game regardless of the solution set. Therefore, we decided on a smaller model and attempted to avoid overfitting. Although this leads to worse performance than optimal, our agent still does quite well. We believe with increased computation power and a larger model, we could match the performance of the entropy agents.

Additionally, we tried implementing Actor-Critic methods and utilize bootstrapping to avoid the need to complete a full episode in order to get the necessary values to update the neural networks. However, we had trouble implementing these methods, as our implementation could not even learn an easier reinforcement learning tasks. We eventually decided to forego bootstrapping, as we felt that the maximum length of episodes in Wordle was so short that this would not be an issue. Given more time, we would experiment with our implementation of bootstrapping and see how well that implementation could learn Wordle.

## 5 Conclusion

The combination of reinforcement learning techniques and natural language processing techniques can lead to interesting applications when dealing with words or speech. We utilize the combination of these techniques to learn to play Wordle, a trending word game. Utilizing REINFORCE in conjunction with Word Embeddings, we were able to train an agent to learn to play Wordle at a level that not only exceeds human levels, but is able to win almost 100% of the time.

## 6 Acknowledgements

Although the code written was written by us, there were some code bases that we either utilized or found inspiration in.

One of the code bases that we utilized for inspiration can be found here https://andrewkho.github.io/wordle-solver/. The write-up found at this web page gave us the confidence to know that solving this problem was possible and provided some guidance to how we approached the problem. However, our approach utilizes a significantly different reward function and state representation. Furthermore, the approach given here utilizes libraries that we thought to be unnecessary and we believed that their approach was very clunky. Additionally, there are some training tricks that we utilize that are not utilized in their write-up, and they utilize some training tricks not present in our write-up. Many of the hyper-parameters suggested by the write-up were also changed by our group, as we found better hyperparameters.

The original code for the OpenAI gym that we modified can be found here: https://github.com/zach-lawless/gym-wordle. Although we utilized the code here as a basis for the gym utilized to train our agent, we made significant modification to the gym in order to follow the state and action specifications found above. Furthermore, the original OpenAI gym contained a major bug that we resolved on our own.

The code for 3Blue1Brown's Entropy-based agent can be found here: https://github.com/3b1b/videos/tree/master/$_2022/wordle$. However, we did not utilize the code itself, just simply the results of this experiment as a baseline.

## References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

[2] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Confer-*

*ence on Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[3] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

[4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization, 2015. URL https://arxiv.org/abs/1502.05477.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

[7] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao. A survey of deep reinforcement learning in video games. *CoRR*, abs/1912.10944, 2019. URL http://arxiv.org/abs/1912.10944.

[8] B. J. Anderson and J. G. Meyer. Finding the optimal human strategy for wordle using maximum correct letter probabilities and reinforcement learning. *CoRR*, abs/2202.00557, 2022. URL https://arxiv.org/abs/2202.00557.

[9] 3Blue1Brown. Solving wordle using information theory. URL https://www.youtube.com/watch?v=v68zYyaEmEA.

[10] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, mar 2003. ISSN 1532-4435.

[11] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.