6.231: Dynamic Programming and Reinforcement Learning

SophieZero: A Wordle-Solving Bot

Boris Velasevic

Massachusetts Institute of Technology

May 25, 2022

Abstract

"Wordle" is a game of guessing a five-letter word using six tries. On each attempt, the game asks us to input a five-letter word and then give us feedback if the solution contains those letters and, if yes, whether we placed them right or wrong. If the letter is at the right place, it is marked green; if it is at the wrong place, it is marked yellow; if the solution does not contain a particular letter, it is marked black. One can input the words from the list of 12972 words, of which 2315 are solutions.

In this paper, we will investigate a reinforcement-learning approach to the problem where we will attempt to solve it with no domain knowledge. The motivation behind this is that we want to make a unified general framework for reinforcement learning rather than using domain-specific algorithms. We will compare and contrast the different representations of state spaces, feature selection methods, and different algorithms and parameters. We will achieve human-like results with less information available than to humans with the indication of super-human play. Furthermore, we will beat an important benchmark set in the paper with our bot.

1 Introduction and Past Results

Although there is no concrete academic research on this problem, it is very popular as a recreational project among computer scientists. A general opinion is that entropy maximizing bots, as well as language models, do very well. Both are very intuitive heuristics. Entropy maximization reduces the uncertainty in the final word and, therefore, makes our beliefs more justified, while language models derive from human intuitions such as the similarity between words, arrangement of letters, etc. We also distinguish between bots that are aware of the final list of answers and those that are not, as out of 12972 possible guesses, only 2315 are possible answers. The entropy maximization bot that does not know final answers achieves the score of 100% correctness, with the average number of tries being 4.12, while the same bot with such knowledge achieves almost 100% correctness, with the average number of attempts being 3.4-3.6[1], depending on the implementation. The awareness of final answers can be implied and does not need to be specified as a list of answers since it is common knowledge that the answer in the game of "Wordle" will always be a commonly used word. Therefore, introducing metrics such as different frequencies of words in the English language can make a bot implicitly aware of the answers. Notice that this, at least in principle, makes humans aware of the list of final answers, although that is hard to quantify. With the information available online[2], we will estimate that the average number of human guesses is slightly larger than four, although this estimate has high variance. Nevertheless, we will focus only on making bots that are not aware of the final list. Hence, we will have less information than humans.

2 Benchmarks

The upper benchmark that we are trying to achieve is the performance of the entropy-maximizing bot without the knowledge of final words. We expect any learning algorithm to be worse than these results, as this is a hard-coded optimal policy. On the other hand, an algorithm that picks randomly has a 0% success rate. We have, therefore, set a benchmark somewhere in between. The benchmark strategy is to select a word randomly on each turn, only from the words fitting the current board pattern if they are answers. The simulations show that this strategy has an 88% success rate, with the aver-

age number of guesses being around 4.8.

Although we refer to this strategy as a "dummy" strategy throughout the paper, this is far from a dummy strategy. It is rather powerful. Random guessing usually discloses plenty of information since uniform random distributions carry the most entropy. Furthermore, since all 5-letter words in English have some similarities (for example, they all have vowels), generating a good guess is relatively easy. On the other hand, as we have six guesses, it is very forgiving if we make an incorrect guess. This benchmark is what makes this optimization hard as "Wordle" is not that difficult of a game, with trivial strategies giving good results. Therefore, even marginal improvements are noteworthy, and every performance worse than this is not justifiable as we need to make a case for the additional resources that we use for training models.

3 Problem Formulation

3.1 Formulation as POMDP

"Wordle" is a classic example of a partially observed Markov decision process. The current state is an ordered pair (B, f) where B is our current board information and f is the final word we are trying to guess. f is not observable, and we can only form beliefs around it. Our action space is W, the set of all 12972 available words. At each step, we choose a word $w \in W$. The word is then entered in the first free row on the board. If $w[i] = f[i]$, then that letter is lit up as green. If $w[i] \notin f$, the letter is lit up as black. If $w[i] \in f$, but $w[i] \neq f[i]$, the letter can either lit up as black or yellow. Let g be the number of letters in w equal to $w[i]$ that are lit up as green. Let y_i be the number of letters in w equal to $w[i]$ up to the i^{th} spot that lit up as yellow already, and let n be the number of letters equal to $w[i]$ in f. Then, if $n > y_i + g$, the letter lights up as yellow, otherwise as black. All the cells on the board that are still without letters are black. We say that we will get a reward $r(B, a)$ for this action, where we tune this function in our different models, which is not a trivial task and might be one of our key decisions. The discount factor, in this case, is 1.

3.2 Different State Representations

We need to represent our state such that it makes our model able to mimic the entropy-maximization policy. Therefore, this is a crucial decision we need to make. Because of this, we propose three different state representations:

1. The full representation. This representation takes in the board as described in the previous subsection and does one-hot encoding of the board. The action is also encoded using one-hot encoding as a 26x5 vector. Since this representation is quite large, the training in more complex models is slow. However, not everything about it is inadequate since, with such representation, we do not miss any information mathematically. Our models only need to be complex enough to capture all the interactions. Furthermore, it captures our goal of not using any outside information.

2. The small-information representation. This representation generates a 26x5 matrix. Each row represents a letter, and $M[i][j] = 1$ if there can be a letter with index i at j^{th} position, and 0 otherwise. However, this representation has a fatal flaw: it is ambiguous around double letters. The way to represent that there is a letter somewhere for certain (the green light-up) is by setting everything to zero except one. However, we hit an ambiguity for a double-letter word: do we want to light up two letters to signal that the letter might be in both places, or do we go for certainty and light up only one letter? We have opted to continue with the latter because of later design choices. The fact that there are not many double-letter words also supports such a decision. The action is one-hot encoded, and we also add the number of guesses left to our representation. Even though this description does not entirely capture our state, it is faster to train and works well in practice, albeit it is not as robust.

3. The information representation. It is a mix of the two above. We take the idea from the small-information representation and use the matrix. However, each entry is now a triplet. One hot encodes three possible scenarios for a letter placement: it is in that place with certainty, or it is in that place with uncertainty, or it is not in that place. The action is one-hot encoded, and we also add the number of guesses left.

4 Methodology and Results

4.1 Common Decisions Across Experiments

For each of our experiments, we run a version of the SARSA algorithm for E episodes, where

each episode is defined to be as playing through the game.

The way we choose actions is decaying ε-greedy. That means our exploration parameter in episode e is set to $\varepsilon_e = \frac{0.5}{\sqrt{\frac{24*e}{E}+1}}$. Hence, $\varepsilon_0 = 0.5$ and $\varepsilon_E = 0.1$, with the exploration parameter decaying. This is done so that as training goes on, and we start getting the idea of what the correct policy is, we explore less and more fit to that policy. The function $f(x) = \frac{1}{\sqrt{x}}$ is chosen as our model because the shape fits the idea that we are trying to capture: the rate of the decline slows down.

Next, we discuss our action choices. We build on top of the dummy policy, as described in section two, where we let our bot choose an action from the list of words fitting the board pattern if they are a final answer. Technically, this is avoidable if, in the beginning, we allow our bot to explore radically. However, it would take away much time and computational resources.

Finally, we discuss rollouts. Notice that we can only simulate a transition on the board of which answer matches the seen pattern. Hence, if we simulate according to some policy and get the answer right, we unlock one possible answer from our position. Therefore, if we run rollouts with a Monte Carlo Tree Search flavor where we use our dummy policy as an expansion policy and backpropagate the information we receive at the leaf nodes, we can almost certainly reconstruct the list of possible answers. However, as we have limited computational power, we will use the same method as above, just choosing a response from the list of potential fits. It is important to note that this is achievable without it with an MCTS.

Note that we have made certain sacrifices where we have substituted a method with a probabilistically equivalent result. We have done this because testing and training become very expensive once we include smart move selection and rollouts, and we want to save as much time as possible.

We test the models on two metrics. The first is the percentage of correct answers after e episodes, while the second is the expected number of guesses given our answer is correct. We sample 55 random words from the answer sheet for every test and select actions using the usual Q-learning selection test with rollouts as described.

4.2 Linear Parameterizations

We start with linear parameterizations. The features that we have chosen are as follows: a constant; guesses left; the number of letters for which we know the word contains them but we do not know their position; the number of letters for which we know the word contains them, and we also know their position; the number of letters in our chosen action for which we know are at the correct spots; and the number of letters in the chosen action for which we do not know their correct spots. These are very intuitive language-inspired features. The cost function that we choose for this model is equal to the negative number of letters for which we know the correct place, with a bonus of the negative number of tries left if we get it right. However, if we fill the board and do not have the answer yet, the cost is six. We run incremental SARSA with updates being:

$$\theta \longleftarrow \theta + \alpha\Phi(c + \widetilde{Q}(s', a'; \theta), -\widetilde{Q}(s, a; \theta))$$

where α is the learning rate. One can argue that this introduces certain domain knowledge, which we want to avoid. On the other hand, the good thing is that this model results in tractable coefficients, which are the following:

$$[-1.16427791e-04, -3.63262047e-04,$$
$$-7.78223438e-05, -2.59573111e-04,$$
$$-2.59573111e-04, -7.86967558e-05]$$

Graphs of the accuracy is given below:

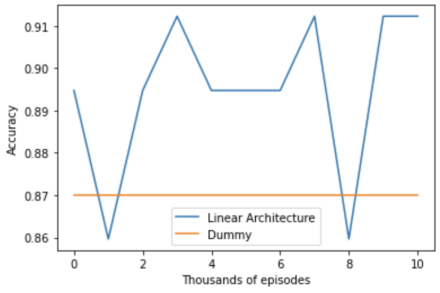

Figure 1: Linear Architectures, Accuracy

And below is the graph of average number of tries, given that we got the answer correctly:

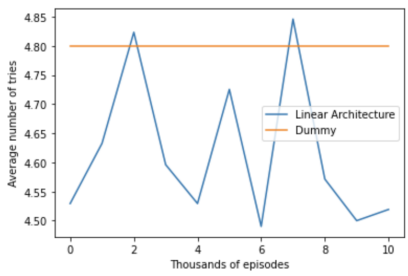

Figure 2: Linear Architectures, Tries

4.3 Deep Neural Network

We can then move onward to using a deep neural network for our approximation architecture. We use a linear network with four layers and ReLU activation functions. The cost, in this case, is set to the number of tries taken if we finish and get the correct answer; it is equal to seven if we finish but do not get the correct answer; otherwise, it is equal to zero. As inspired by AlphaZero[3], this neural network tries to minimize the function $l = (Q - c)^2 + \lambda\|\theta\|^2$. The parameters are frozen during action generation, and they are updated at the end of every episode, using backpropagation from the loss function. Therefore, this neural network tries to estimate $\mathbf{E}[c|(s, a)]$ with regularization. We run first run this for full representations and get the following result for accuracy:

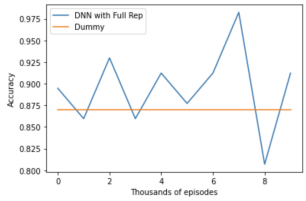

Figure 3: Deep Neural Network, Full Representation, Accuracy

The following is the result for average number of tries given that we guessed correctly:

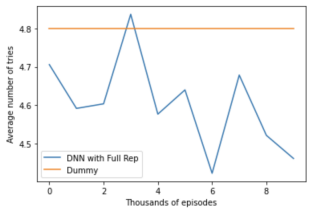

Figure 4: Deep Neural Network, Full Representation, Tries

The following are results for small information representations accuracy:

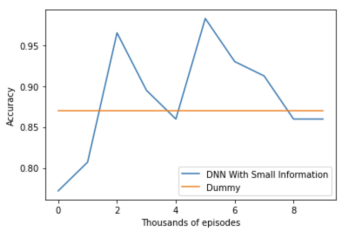

Figure 5: Deep Neural Network, Full Representation, Tries

The following are results for small information representations average number of tries:

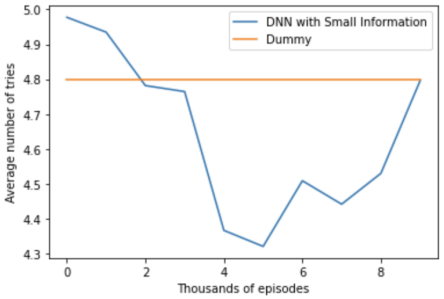

Figure 6: Deep Neural Network, Full Representation, Tries

We leave out large information representation results from this paper. They did not end up great, and the divergence is larger. One possible reason is that it is unclear how to quantify the difference between that a letter could potentially be in a position or certainly be there.

5 Discussion

First, in all our methods, we see the shortcomings of approximate value iteration, which are the basis of our algorithms. Namely, the convergence is not guaranteed, and the graphs can zigzag as other policies are discovered. However, we found some quality policies in our training.

For linear representations, we can track what the coefficients mean. Namely, we see that they are all negative as expected since we expect all the mentioned features are good for us and reduce the cost. Secondly, we can see the relative weight of different things. Therefore, we can infer that it is important to use the letters for which we already know their place, which many humans do during play. Furthermore, we see that having letters for which we know their spot reduces the cost more than having ambiguous letters when we weigh in the fact that we usually have more ambiguous letters on the board than the correct ones. As for the metrics, we can improve on our dummy policy consistently, but with deep neural networks, we will do even better.

For the deep neural network that takes in a full representation, we can see that we are generally able to improve the accuracy of our bot, with the best policy having a 98% accuracy. However, the real improvement comes in the form of average tries, where we consistently outperform our dummy strategy, having it as low as 4.42. It is a good improvement as it gets us more than halfway close to our upper bar, which is 4.12, from our lower bar being 4.8. Notice

that the policies that have high accuracy and do not match one-to-one with the policies that have low average guess count. This might suggest two possible things. One is that our network is trading accuracy for speed by playing words that carry higher variance but more information. The other possibility is that we are learning a part of a state-space well, but when we wander outside that part, we do not know what is going on and start playing random moves.

For the deep neural network that takes in a small information representation, we see that we have a unifying policy that achieves 96% and average number of tries of 4.31. Similarly, this is a very good result. However, one can see that the results are not as robust for this one, with accuracy being over our benchmark on only 50% of our tries.

6 Further Work

For any deep neural network representation, as our states and input features are well-defined, we used a usual deep neural network with linear fully connected layers and ReLU activation functions. However, it is tempting to substitute this for a deep convolutional neural network. This makes sense as we know which cells and entries in our representation are supposed to mix, so we can design kernels around this. This substitution is one possible way of doing further work.

Next, in our rollouts, we only used information gathered by MCTS, but not the value functions that were estimated along the way. Therefore, this is a possible area of improvement, implementing a fully functional MCTS that backpropagates value functions and information, with the expansion policy being our dummy policy.

Finally, we can investigate why different policies optimize for the average number of tries and accuracy and identify in which instances any given policy fails. It is possible that a policy fails on a portion of state space that has not been explored yet, or that it is too biased towards the currently explored state space. In that case, we can keep some visited states and try to combine our policies to get a better policy that achieves both accuracy and a low number of tries.

7 Appendix

Please find the code to the GitHub repository in the references[4].

8 References

[1]-https://www.3blue1brown.com/lessons/wordle
[2]-https://twitter.com/WordleStats?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor
[3]-arXiv:1712.01815
[4]-https://github.com/borisvel/sophiezero