Reinforcement Learning Approaches for Solving the Language Game Wordle

Alex Guentchev
Department of Computer Science
Colorado State University
Fort Collins, USA
a.guentchev@colostate.edu

***Abstract*—Wordle is a language game where the player has to guess a 5-letter word in 6 guesses. After each guess, the puzzle provides feedback to the user which is used to fine-tune the remaining guesses in an effort to deduce the word [16]. Many people play Wordle daily, and there are strategies which are considered optimal (such as [2]). These approaches do not interest me as much as using reinforcement learning to create a model which figures out its own strategy. I have found a baseline created by Harsh Shah, Anwit Damle, Ujjwal Agarwal [13]. They used Deep Q-Learning [3] and Advantage Actor Critic (A2C) [4] to develop two models that attempt to solve Wordles. I sought to improve on this baseline by implementing Asynchronous Advantage Actor Critic (A3C) [7].**

***Index Terms*—Reinforcement Learning, Deep Q-Learning, Actor Critic, Wordle**

I. Introduction

Wordle is a language game published by the New York Times [16]. The goal of the game is to guess a hidden five letter word in six attempts or fewer. After attempting to guess a valid five letter word, the puzzle returns information to the user in the form of coloring letters in the guess to help attune the user's next guess, such as in Figure1. Wordle uses a word list of 2,315 words for solutions, but well over 10,000 five letter words are valid as guesses. These lists are pulled from the New York Time website [17]. The game has a normal mode and a hard mode. In normal mode, you can guess whatever you would like each round. In hard mode however, if you guess a letter that is in the word, you have to use that letter in subsequent guesses, preventing the player from simply guessing the widest possible variety of letters to deduce the word.

There are many strategies for solving Wordles. The most common strategies revolve around trying to guess which word is most likely to be next based on the frequency at which certain letters appear in the word lists. The best starting word is "salet" [1] according to MIT scientists. Other strategies also exist such as trying to guess all of the vowels first (using a starting word such as "audio"), guessing completely different words while playing easy mode to cover as much of the alphabet as possible (such as "ghost" then "blade"), or even using a tool to narrow the available guesses down to only valid guesses given the feedback [2]. However, in the context of this class and what I would like to study, implementing those approaches is not nearly as interesting or challenging as using reinforcement learning to create a model which has to learn and create its own strategy from scratch.

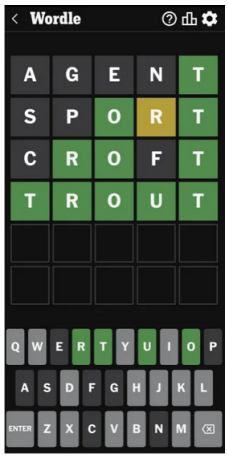

Fig. 1. An Example of the Feedback Provided by the Puzzle

The GitHub project "woRdLe" created by Harsh Shah, Anwit Damle, Ujjwal Agarwal [13] will be used as the baseline for this work. They decided on an RL approach because it is capable of handling the complexity required by the project. Another aspect was that the feedback to indicate correctness from Wordle after a guess resembles the reward signal in RL [13]. They used a 78-dimensional vector to encode a letter's correctness and relative position to the target word [13]. To represent the actions, they selected a word from the provided dictionary, with masking to prevent repeated guesses [13]. The environment is a mockup of the Wordle game which provides feedback in a similar manner to the base game, communicating if a letter is in the word or not, and if the position is correct or not, for a total of three states per letter [13]. The rewards they assign are -10 for failing to guess the hidden word within six attempts. A reward of +10 is given when the hidden word is correctly guessed in six or fewer attempts. There is also a midway reward of +1 per letter correctly placed before the attempts run out or the hidden word is guessed [13]. The terminal condition for the environment is either a correct guess or the agent exhausting all of the attempts without correctly guessing [13].

The variation of Deep Q-Learning used in the "woRdLe"

project is Double DQN. Double DQN has separate Q-network functions for finding the optimal action and for finding the Q-value. Epsilon greedy action selection is used to achieve a balance between exploration and exploitation [3].

The second algorithm used is Advanced Actor Critic (A2C). This uses an actor to select actions based on a policy to maximize rewards. This policy is continually refined to adapt it to the environment. The critic evaluates the actor's actions and provides feedback to guide the actor towards higher returns and improved learning. The Advantage Function, A(s,a), measures the advantage of taking action 'a' in state 's' over the expected value of the state under the current policy [4].

$$A(s,a) = Q(s,a) - V(s)$$

My goal is to beat the baseline set by the aforementioned "woRdLe" GitHub project scores, as well as extending and improving the project. The original scores are shown in Table 1, as found in the original paper [13].

TABLE 1
WORDLE BASELINE

Model	Start Type	Win Rate	Ave Guesses	Training Time
DDQN	Exploration	0.66	5.09	3 Hours
DDQN	"SALET"	0.71	4.95	3 Hours
A2C	Exploration	0.7	4.8	1 Hour
A2C	"SALET"	0.76	4.98	1 Hour

II. Related Work

I had a few ideas for how to best implement my improvements, and I was inspired from several different sources.

Andrew Ho has a publicly available Advantage Actor Critic (A2C) implementation for solving Wordle as well as a detailed writeup talking about his design choices and process [6]. He was able to achieve 99% winrate on a 1000 word case, and 95% on the full-size problem. He trained his implementation on over 20 million matches, which is an unreasonable amount for my timeframe, but his work still provides important insight into how I can improve on the baseline I have chosen. One interesting thing to potentially try is keeping a FIFO queue of "recent losses". Ho's model had issues with words containing many scarce letters such as "pizza", so he kept a queue of challenging words which the model would occasionally access if it was not making progress. This allowed the model to cut the number of losses in half after the updated training was completed. Another interesting optimization Ho used was first training the implementation on a smaller problem (1000 words), then scaling up to the full-size problem. This allowed the training for the full-size problem to be more streamlined and effective.

Santiago Tettamanti sought to improve upon Andrew Ho's implementation, by implementing Asynchrous Advantage Actor Critic (A3C) [15]. He was able to achieve the same scores as Andrew Ho (99% winrate on a 1000 word case, and 95% on the full-size problem), but with significantly shorter training times (2 million games). Tettamanti tried multiple ways to preprocess the data, but found that the one-hot encoding that Ho's solution used was the best solution he could find. He also heavily reiterated that pretraining the model on a smaller sample size helped significantly with training for the full-size problem. That alongside implementing A3C allowed the model to converge significantly faster than Ho's implementation.

I found more information on A3C in a paper that builds on the Advantage Actor Critic (A2C) algorithm [7]. It introduces the algorithm Asynchronous Advantage Actor Critic for the first time. This relies on parallel actors learning and accumulated updates for further improved training stability. This also expedites the training and improves exploration capability. Adding entropy to the policy also further improves exploration by discouraging premature convergence [18].

One of the driving forces in the development of reinforcement learning was the base Deep Q-Networks algorithm that came out in 2013 [8]. Since then, there have been a wide array of optimizations or additions to the formula. The most exciting one I found was called Rainbow DQN [5]. This combines six extensions of the DQN algorithm into one integrated agent that outperformed all of the individual optimizations on their own. The six optimizations that were combined were Double DQN, Prioritized Double DQN, Dueling Double DQN, Distributional DQN, Noisy DQN, and Asynchronous Advantage Actor Critic (A3c). In Andrew Ho's A2C implementation he claims that he also tried a basic DQN, but was unable to make it work properly for the full-size problem [6]. I would be curious to see how Rainbow DQN would deal with the problem, and how much it improves the baseline set in the "woRdLe" project.

III. Method

First, I began to mess with the baseline implementation to figure it out more. I ran it on my home machine, but I do not have an NVIDIA card, so the models performed worse than advertised in their report. It took around five times longer to run, and each only scored around 20% win rate after training for 300,000 epochs each. I had never run a project this large for this many epochs, so I was not quite aware of the immense impact that having an NVIDIA GPU with access to CUDA would have on training at this scale. Not only did it take longer, but since the architecture is so optimized and specialized, it also improves the scores when run on the NVIDIA hardware. I took a bit of time to do some extra research and I came across an implementation of A3C which was designed to run in parallel on CPU cores [14]. This would hopefully solve the issue of longer runtime. Since it was intended to be run on CPU cores, the negative impact of not having NVIDIA's parallel processing would hopefully be dampened. I would be taking inspiration from the A3C implementation across CPU cores, but I would mostly be trying to extend the current baseline implementation, and basing my code off of it stylistically as well to a certain extent.

Similar to the A2C implementation in the baseline, my model has an Actor and a Critic which both feature two linear layers using a ReLU activation function and 64 units per hidden layer. I decided to continue using ReLU to avoid the vanishing gradient problem, since I knew I would be training

for a very long time. This was motivated in part by this paper's [9] arguments that ReLU is a good solution to this issue. The Critic outputs a scalar that estimates the value of the current state. The Actor returns a vector of logits (raw confidence scores) 1, as opposed to the original A2C implementation that directly returns the Categorical distribution of probabilities after applying softmax. This allows me to use the logits as a mask for the actions before converting it into a distribution. I can then use "dist.sample()" to select an action based on probability, with higher rated moves being selected more often, but not discouraging exploration since low confidence moves have a small chance of being chosen as well [11].

Listing 1. A3C Model Architecture - Python

```python
class Actor(nn.Module):

    def __init__(self, state_size, action_size):
        super(Actor, self).__init__()
        self.layer1 = nn.Linear(state_size, 64)
        self.layer2 = nn.Linear(64, 64)
        self.layer3 = nn.Linear(64, action_size)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.layer3(x)
        return x

class Critic(nn.Module):

    def __init__(self, state_size, action_size):
        super(Critic, self).__init__()
        self.layer1 = nn.Linear(state_size, 64)
        self.layer2 = nn.Linear(64, 64)
        self.layer3 = nn.Linear(64, 1)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        value = self.layer3(x)
        return value
```

The main advantage, in my opinion, of A3C over A2C is the asynchronous aspect, and this gave me by far the most difficulty. Luckily, Torch has its own multiprocessing package [12]. I originally wanted to have my Actor and Critic architecture in a separate python file, which is imported into the main Jupyter Notebook, similar to how the baseline was normally set up, so that it was easy to follow along with the code and to view the results. However, I was unable to get the multiprocessing to work on separate cores unless I launched a separate process for each agent ("mp.set_start_method('spawn')"). Unfortunately, doing this requires restarting the kernel every time I want to run the multiprocessing portion of the notebook, and it worked inconsistently in the notebook as well due to a reason I was unable to figure out. After wrestling with that for a

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t;\theta'))$$

Fig. 2. A3C Gradient Function with Entropy [7]

considerable amount of time, I opted to just have my code for the multiprocessing run in a separate Python file which is in the same folder as the original "main.ipynb". As a result, I will be adding an example output from a randomly selected run of "a3c.py" in Appendix A, Section VIII for the reader's reference.

The code makes a global Actor and global Critic that get their gradients updated by local Actors and Critics created by the newly spawned processes. I used Torch's Adam optimizer, with a learning rate of '0.001', similar to the baseline A2C implementation [13]. The optimizer is global, but needs to be updated by one process at a time so I use a Lock to prevent race conditions. Other than the things listed so far, the code works in a pretty similar manner to the original implementation from A2C from the baseline. The final major difference is in the rewards.

In the baseline GitHub project there is a PowerPoint presentation that justifies some of the team's design choices, and explains how they wanted certain aspects of the code to work [13]. To reiterate from earlier, the reward function gives +10 for guessing the hidden word correctly within six guesses, -10 for failing to guess the hidden word in six guesses, and +1 for each correctly placed letter in a guess. I originally interpreted this to be continuous, but in practice this means that at the end of an epoch there can only be scores of 10 or -10. While the environment is stepping, the rewards for correct letters get pushed onto the rewards list to be used for calculating the advantage. That means an award of at most +4 gets added to that list. Then, when the puzzle is solved or failed, -10 or +10 is pushed onto the list without taking into account how close the model got to guessing the word before that occurred. This is a reasonable idea, but since I originally assumed the reward would be continuous until the end of the update, I decided to implement that to see how it would affect the training. In my implementation, the reward increases a small amount with each additional letter guessed, then when the individual puzzle concludes, 10 is either added or subtracted from that running total based on success or failure. I was hoping this would reward the model more for getting close to the hidden word even if it could not guess it. This would also reward the model more for guessing a higher number of correct letters earlier on, which I hoped would incentivize the model further to find a more effective starting strategy.

The gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the form in Figure2 where H is the entropy. The hyperparameter B controls the strength of the entropy regularization term. log(Pi(a t—S t;Theta')) represents the log_probs and V(S t;Theta v) is the single value output by the critic. Rt are the returns [7]. So in my code we have "-(log_probs * ad-

vantage.detach()).mean() - 0.01 * entropies.mean()". Entropy needs to be subtracted so that it can penalize certainty. This encourages exploration [18].

After that change, I ran a few more experiments to try to repair the accuracy in the model. I tried running with fewer than the max cores to avoid having the processes wait on each other for control of the lock. I tried some runs with the old reward structure as well to compare the values. The results from all of the runs can be found in the following Section IV.

IV. Results

The results from running the baseline on my machine can be found in Table 2. The results from my model and experiments can be seen in Table 3 featuring the new reward function and Table 4 featuring the old reward function.

TABLE 2
"woRdLe" Models Run on CPU Instead of GPU

Model	Start Type	Win Rate	Ave Guesses	Training Time
DDQN	Exploration	0.19	5.82	13 Hours
DDQN	"SALET"	0.22	5.85	13 Hours
A2C	Exploration	0.23	5.63	12 Hour
A2C	"SALET"	0.22	5.58	12 Hour

TABLE 3
A3C Model Results (New Rewards)

Start Type	Win Rate	Ave Guesses	Episodes	Cores
Exploration	0.19	5.87	1000	24
"SALET"	0.20	5.82	1000	24
Exploration	0.16	5.87	20000	24
"SALET"	0.20	5.82	20000	24

TABLE 4
A3C Model Results (Old Rewards)

Start Type	Win Rate	Ave Guesses	Episodes	Cores
Exploration	0.18	5.86	13000	24
"SALET"	0.23	5.80	13000	24
Exploration	0.18	5.88	20000	16
"SALET"	0.21	5.79	20000	16

All of the scores were around 20% win rate, which is very bad. However, it is essentially the same score the baselines provided when I ran them for 300,000 epochs on my hardware as well. Running the maximum number of processes seemed to run very slowly. Lowering the number of processes below max gave a slight performance boost, but it was not enough to justify running significantly more episodes with fewer processes. The A3C models took about an hour to run on 1000 episodes, and about 10 hours to run 20,000 episodes. 20,000 episodes with 24 versions of the actor should be roughly equivalent to running 480,000 epochs for an individual A2C actor.

V. Discussion

Although I was unable to beat the baseline, I started this project with the intention to learn, and that goal was achieved. I had decide whether I would stop at nothing to beat the baselines, or learn from where I succeeded and failed. I opted for the latter, since in research even failure can represent progress. Here's what I learned from my results.

Despite being unable to beat the baseline, my A3C implementation ran noticably faster on my hardware than the baseline did, which was a personal success. However, as I increased the number of processes, the speed of the training decreased. This was likely due to the length of the individual episodes being shorter than the amount of time taken for each process to update the optimizer. This led to many processes waiting for the lock instead of continuing to train. When I lowered the number of processes to 16 from 24, the training was faster, despite not having a chance to train as many actors in parallel.

The result I was most interested in seeing was how the new reward structure would compare to the old one. This seemed comparable when run for 1000 episodes, but when I ran it for 20,000 episodes, the model achieved the lowest scores I had seen the entire time with 0.16 win rate. As the training was commencing, I began to notice that the rewards were steadily increasing early on before dipping to become consistently negative after around 3,000 episodes. I believe what happened was that the higher rewards early on for guessing correct letters were effectively training the model to only guess words that contained a large number of the correct letters, since that was a more consistent way of maintaining a higher score than guessing the word correctly. For instance, if the model guesses the word mostly correct six times then fails to guess the hidden word, it can get a reward of around 14 for that episode. If the model guesses the word second try (which is more aligned with the goal of the project), the maximum possible score it could achieve is 14 (4/5 letters correct first guess, +10 for correct word second guess). I think the old method of not cumulatively adding the reward is a better strategy. An improvement I can think of that I did not have an opportunity to try would be using the old architecture. However, instead of only +1 for each correct letter in the correct space, the model could award +0.5 for a correct letter in the incorrect space. Currently the model gives no reward whatsoever for a correct letter unless it is in the correct space, which seems counterintuitive to me.

All of the scores that were achieved during this project were disappointing and highlight a glaring issue with the work that I did. Even running the exact code provided by the baseline, I was unable to get a score higher than one third of the score the baseline project claims to have achieved. I was unable to figure out exactly why this happened, but my best guess is that it is hardware related. As mentioned in Section VII, I was running everything on my computer's CPU and the best practice for running torch-related models is generally to use an NVIDIA GPU. I was surprised to learn how much of an

impact and improvement using specialized hardware would provide in this context since in all of my projects up until this point I had yet to notice a difference. That being said, this is the first project I have tried that has such a large scale to it.

VI. Conclusion

I was able to create a model which meet the score I got by running the baseline code on my hardware. I did not anticipate the scores being so low, or how difficult it would be to notice a change in the scores when attempting to tune the parameters. The score is also nowhere near my personal Wordle completion rate, which was mentioned in the original proposal for the project. I would be curious to see how the scores compare when run on better or more appropriate hardware. Regardless, this was a very interesting and difficult challenge. I definitely learned a lot throughout the entire process.

VII. Future Work

One of the things I found most interesting while playing around with the baseline code was how differently the models performed when run on CPU and GPU. I was unable to get my hands on an NVIDIA GPU, but if I had an opportunity to continue this work in the future, I would definitely seek one out. Another alternative is to use a GPU cluster. I know CSU has GPU clusters, but I did not learn about it in time to go through the application process to use them for this project. They seem like a very powerful tool for anything AI or ML related. After figuring out a way to run multiprocessing on GPUs, I could use PyTorch's GPU functionality for multiprocessing, instead of just running every process on a separate CPU core. Oftentimes the overhead of cross-thread communication on CPU negates the performance benefits, leading to GPU parallelism being far more effective due to its architecture [10].

Another thing I would have liked to have looked into if given the time would have been writing my own optimizer for A3C. The original implementation has a custom optimizer for DDQN, but uses torch.optim.Adam for A2C. I also ended up using Adam, but writing a more suited optimizer for this specific problem could potentially increase performance in the model. Optimizers are also interesting to me because I would say it is one of the things I understood the least while working on this assignment. Luckily Torch has a ton of functionality to automate whatever I need, but in the future it would be nice to learn enough about optimizers to write my own.

If I had an unlimited amount of time to train the model, I would have loved an opportunity to mess with the hyperparameters more, such as learning rate, discount factor, entropy function, and activation function. I am confident in the hyperparameters I chose, but would have been curious to see how altering them impacted accuracy.

One more interesting approach to take would have been an implementation of the Rainbow DQN mentioned in Section II. It is an optimization over the DDQN that was implemented in the baseline, and it incorporates A3C into its model. This leads me to believe that it had the potential to score the highest out of the models considered in this paper.

The last thing I wish I had time for was to train the model over a large number of epochs on a smaller version of the problem, such as training over 1000 words like Ho's project [6]. Then, I could use the pretrained model to scale up to the full-size problem.

References

[1] Siddhant Bhambri, Amrita Bhattacharjee, and Dimitri Bertsekas. Reinforcement learning methods for wordle: A pomdp/adaptive control approach, Nov 2022.

[2] Jason Chao. Jason-chao/wordle-solver, Spring 2022.

[3] PyTorch Foundation. Reinforcement learning (dqn) tutorial - pytorch tutorials 2.7.0+cu126 documentation.

[4] GeeksforGeeks. Actor-critic algorithm in reinforcement learning, Feb 2025.

[5] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow dqn, Oct 2017.

[6] Andrew Ho. Wordle solving with deep reinforcement learning, Jan 2022.

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, Jun 2016.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and et al. Pytorch: An imperative style, high-performance deep learning library, 2019.

[11] PyTorch. Probability distributions - torch.distributions.

[12] pytorch mp. Multiprocessing package - torch.multiprocessing.

[13] Harsh Shah, Anwit Damle, and Ujjwal Agarwal. Harsh788/wordle: A deep reinforcement learning approach for solving the popular game of wordle. used algorithms such as double dqn and advantage actor critic., Spring 2024.

[14] Phil Tabor. Asynchronous advantage actor critic (a3c) tutorial.

[15] Santiago Tettamanti. How to solve wordle using machine learning, May 2023.

[16] The New York Times. Wordle game.

[17] KC White. List of valid wordle guess words pulled directly from the ny times website.

[18] RONALD J. WILLIAMS and JING PENG and. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

VIII. Appendix A

Listing 2. A3C Example Output

```
<Process name='Process-1' pid=8616 parent=19900 started>
<Process name='Process-2' pid=19760 parent=19900 started>
<Process name='Process-3' pid=8848 parent=19900 started>
<Process name='Process-4' pid=3456 parent=19900 started>
<Process name='Process-5' pid=592 parent=19900 started>
<Process name='Process-6' pid=18176 parent=19900 started>
<Process name='Process-7' pid=12192 parent=19900 started>
<Process name='Process-8' pid=16632 parent=19900 started>
<Process name='Process-9' pid=8208 parent=19900 started>
<Process name='Process-10' pid=24492 parent=19900 started>
<Process name='Process-11' pid=22252 parent=19900 started>
<Process name='Process-12' pid=21144 parent=19900 started>
<Process name='Process-13' pid=23292 parent=19900 started>
<Process name='Process-14' pid=18156 parent=19900 started>
<Process name='Process-15' pid=5248 parent=19900 started>
<Process name='Process-16' pid=19468 parent=19900 started>
<Process name='Process-17' pid=8872 parent=19900 started>
<Process name='Process-18' pid=20688 parent=19900 started>
<Process name='Process-19' pid=5188 parent=19900 started>
<Process name='Process-20' pid=23168 parent=19900 started>
<Process name='Process-21' pid=16904 parent=19900 started>
<Process name='Process-22' pid=9680 parent=19900 started>
<Process name='Process-23' pid=25492 parent=19900 started>
<Process name='Process-24' pid=14004 parent=19900 started>
[Thread 0] Started
[Thread 1] Started
[Thread 2] Started
[Thread 3] Started
[Thread 4] Started
[Thread 5] Started
[Thread 6] Started
[Thread 7] Started
[Thread 8] Started
[Thread 10] Started
[Thread 9] Started
[Thread 11] Started
[Thread 12] Started
[Thread 14] Started
[Thread 13] Started
[Thread 18] Started
[Thread 15] Started
[Thread 17] Started
[Thread 16] Started
[Thread 20] Started
[Thread 19] Started
[Thread 21] Started
[Thread 23] Started
[Thread 22] Started
[Thread 0] Episode: 0/20000, Attempts: 6, Reward: -4
[Thread 0] Episode: 500/20000, Attempts: 5, Reward: 17
[Thread 0] Episode: 1000/20000, Attempts: 5, Reward: 16
[Thread 0] Episode: 1500/20000, Attempts: 6, Reward: 21
[Thread 0] Episode: 2000/20000, Attempts: 4, Reward: 13
[Thread 0] Episode: 2500/20000, Attempts: 6, Reward: -7
[Thread 0] Episode: 3000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 3500/20000, Attempts: 6, Reward: -7
[Thread 0] Episode: 4000/20000, Attempts: 6, Reward: -1
[Thread 0] Episode: 4500/20000, Attempts: 6, Reward: -8
[Thread 0] Episode: 5000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 5500/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 6000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 6500/20000, Attempts: 6, Reward: -5
[Thread 0] Episode: 7000/20000, Attempts: 6, Reward: -4
[Thread 0] Episode: 7500/20000, Attempts: 6, Reward: -3
[Thread 0] Episode: 8000/20000, Attempts: 6, Reward: -3
[Thread 0] Episode: 8500/20000, Attempts: 6, Reward: -5
[Thread 0] Episode: 9000/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 9500/20000, Attempts: 6, Reward: -7
[Thread 0] Episode: 10000/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 10500/20000, Attempts: 6, Reward: -4
[Thread 0] Episode: 11000/20000, Attempts: 6, Reward: -3
[Thread 0] Episode: 11500/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 12000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 12500/20000, Attempts: 6, Reward: -5
[Thread 0] Episode: 13000/20000, Attempts: 6, Reward: -7
[Thread 0] Episode: 13500/20000, Attempts: 6, Reward: -1
[Thread 0] Episode: 14000/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 14500/20000, Attempts: 6, Reward: 1
[Thread 0] Episode: 15000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 15500/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 16000/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 16500/20000, Attempts: 6, Reward: -4
[Thread 0] Episode: 17000/20000, Attempts: 6, Reward: -5
[Thread 0] Episode: 17500/20000, Attempts: 6, Reward: -6
[Thread 0] Episode: 18000/20000, Attempts: 6, Reward: 1
[Thread 0] Episode: 18500/20000, Attempts: 6, Reward: -1
[Thread 0] Episode: 19000/20000, Attempts: 6, Reward: -10
[Thread 0] Episode: 19500/20000, Attempts: 6, Reward: -10
~~~ TRAINING COMPLETE ~~~
~~~ BEGINNING TESTING ~~~
Trials: 1000, Success rate: 0.16,
Average number of attempts: 5.87
~~~ BEGINNING TESTING 2 ~~~
Trials with SALET start: 1000, Success rate: 0.20,
Average number of attempts: 5.82
```