Final Project

Mark Bentivegna, Zichen Wang

April 2022

1 Introduction

For this project we take the techniques we learned in class and apply them to a real-world reinforcement learning problem. Throughout the semester we covered many topics of reinforcement learning and read many papers in the field. We completed several homework assignments applying multi-armed bandits, dynamic programming, and tabular methods to grid world and blackjack. In this project I apply the tabular methods, namely Monte Carlo Exploring Starts and Q-learning, and Zichen implemented DQN, to generating crossword puzzles.

2 Problem Overview

Crossword puzzles are popular puzzle games consisting of a 2-dimensional grid layout, not unlike grid world, in which squares are either "available", meaning they are to be filled in with a letter consisting part of a word, or "unavailable", meaning they are empty spaces and the user can ignore them. The user is provided with a set of clues accompanying each word entry and the user must guess the word based on the clues provided. Sometimes these words may have a unifying theme but this is not necessarily considered a "rule" of crossword puzzles. These puzzles are popular across many different languages and countries but for this project we are interested in American-style crossword puzzles in English. The American-style crossword puzzle is a very clearly defined grid type.

Key traits include minimizing empty spaces, symmetrical layouts, and equal dimensions along both axes. In the layout provided every combination of intersecting word entries along the horizontal and vertical axes must form valid words. It is easy to see how this can be a difficult problem to solve, with many different intersecting squares throughout the grid there are many crossing words that must be validated. For example, 3 down and 29 down each consist of 10 squares and intersect 10 other word entries. Using reinforcement learning to generate this grid is very computationally expensive so we decided to take a slightly simpler grid layout pictured in figure 2.

The crossword puzzle template employed in this project is a "medium" level of difficulty using American rules of symmetry and equal dimensions along axes.

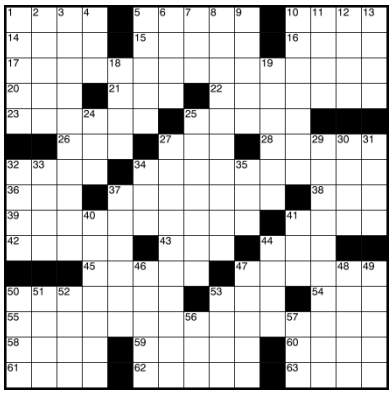


Figure 1: American-style crossword puzzle layout

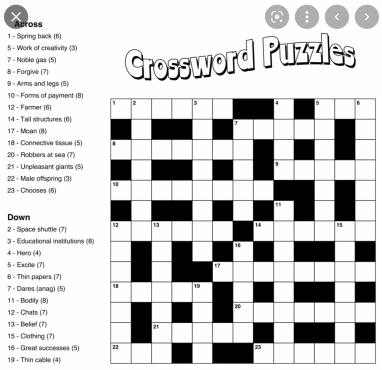


Figure 2: Scaled down crossword puzzle layout used throughout this project

This simpler approach was necessary because training our agent to generate complete and valid puzzles took too long.

We found several different publications of attempts to build crossword puzzles using machine learning but none using reinforcement learning. Because of the lack of research done in this area we thought it presented a very exciting opportunity of exploration. There have been many documented successful attempts at using machine learning to solve crossword puzzles but very little research has been done in generating them.

Fortunately, it was very easy to find excellent data sources for this project. There are many public data sets consisting of word-clue combinations. The New York Times released all of their crossword puzzles dating back to 1997. Provided with 25 years' worth of crossword puzzle templates we have over 6,000,000 individual word-clue combinations to choose from when generating our templates. Each crossword puzzle generated will consist of 26 word entries.

We identified three distinct reinforcement learning sub-problems within the

scope of generating crossword puzzles. They are as follows:

2.1 Word Selection

This problem is quite intuitive. The best fitting words are those consisting of the most common letters in the English language. If the words we select consist of uncommon letters such as "x", "q", or "z" it will be very difficult to build valid word combinations. Because so few words contain these letters the available word pool for a given entry will be very limited to begin with, making it more difficult to build from. For this problem we generated a logarithmic rewards function giving a higher score to words consisting of more vowels and common letters. We used a logarithmic function to provide diminishing returns after a while because after a certain point because otherwise we wanted a more evenly spread rewards distribution. We also penalized words containing uncommon letters.

Figure 3: Sample crossword puzzle output from our agent

Ultimately, due to time constraints, we decided to trim out this portion of the simulation for training our agent. Our final simulations were done using random selections from a list which filtered out words containing uncommon letters. Our target policy is to figure out which word entry to select next given the most recent entry and the current set of available and unavailable slots on the grid.

2.2 Grid Layout

Choosing a valid order of words to lay on the grid is obviously one of the most important components of generating a crossword puzzle. The most constraining words are those with the most intersections. Earlier we highlighted two different

word entries in the American-style grid that intersect ten other words. Clearly, it makes more sense to add words with more intersections first because they start a better foundation for branching off to form other words.

This is the primary objective of our problem. We provided our agent with preexisting knowledge beforehand, giving a boost in preference to several different word entries with the highest number of intersections. We will elaborate on how we did this in a latter section.

2.3 Resolving Conflicts

After randomly several words into the puzzle we will inevitably get to "stalemates" where there are no available valid words to add to a grid and so we need to resolve these by modifying existing words or re-arranging our grid. This problem is especially common in earlier episodes when our agent is just randomly adding in words without any preexisting knowledge.

This problem is very computationally expensive to solve so it was quickly decided that we can't feasibly complete this problem within the scope of our project. When our agent has no available word entries it engages in what I call "thrashing" by randomly removing neighboring word entries to those we must fill and trying different words hoping the new letters will allow us to build additional word combinations. Because of the random nature this can sometimes lead to multiple consecutive steps in which our agent is just removing and adding words and hoping for the best. The random nature is obviously very naive but in practice it isn't too much of a problem because our agents understand learn effective patterns to lay words on the grid.

3 Reinforcement Learning Approach

The most time-consuming component of this project has been framing the problem in the context of reinforcement learning as well as generating the Python code needed for the simulator. In previous assignments we designed simulations for grid world, grid world with slippage, and black jack. These simulations are relatively straightforward because there is a small state space and even smaller action space. Generating crossword puzzles is a much less constrained problem, there are over $4 * 10^{26}$ different permutations of adding word entries and over 6,000,000 different words to select from for these entries. It is easy to see how there is a very large number of different combinations of states and actions and this will present computational difficulties and introduce a degree of uncertainty for our agent. To further demonstrate how our agent works, we include figures of the first four steps completed by our agent to illustrate how our agent learns to intelligently lay down word entries. In the figures, we can see how our agent

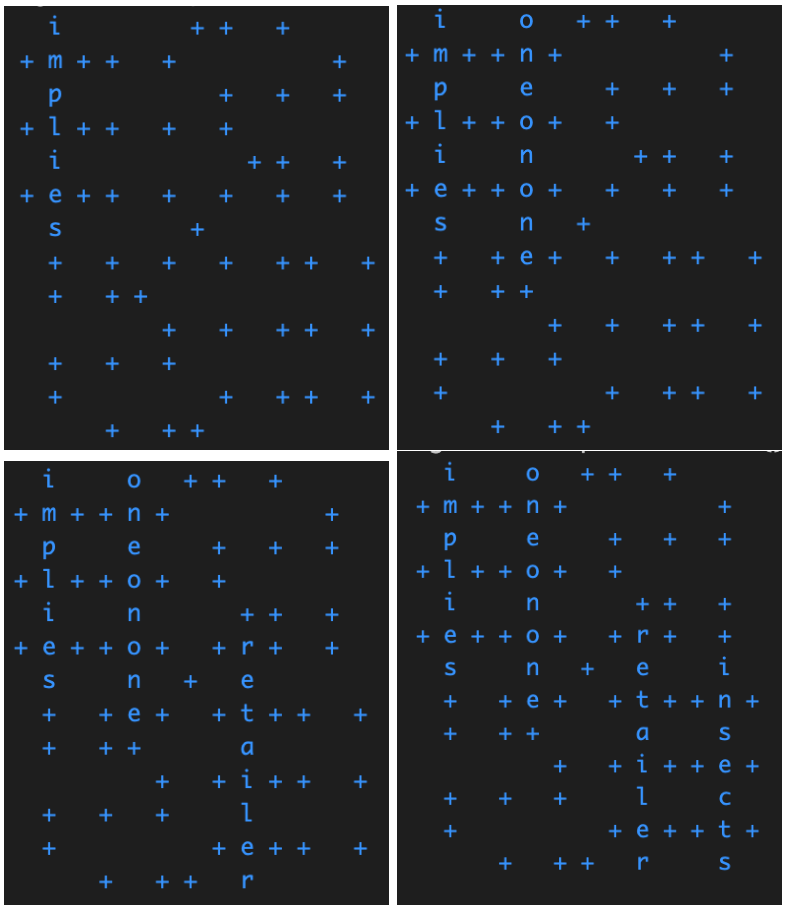


Figure 4: Example of the first four steps of our agent completes using Q-learning. Our agent learns to prioritize word entries containing more intersections with other entries

3.1 Framing the Problem

Intuitively, we found this problem is best represented using reinforcement learning. Supervised and unsupervised learning techniques are appropriate if there are vast quantities of data available for the problem to learn how to draw conclusions and connections but that is not what we set out to do. We are trying to generate crossword puzzles based on nothing more than a game board. Our template has very little data, just the grid and its attributes.

Our input consists of a grid with "0" value for valid coordinates that need letters and "1" for obstacles. Following this grid is one row per word entry containing a unique identifier along with the starting and ending coordinates of the given word.

3.1.1 States

Our state space consists of the list of used word entries, unused word entries which we still need to fill, and our most recently added word entry. It is critical that we keep track of all of the words that we have already filled because we don't want to modify the layout, we want to move on to filling additional open entries as soon as possible in the interest of computational efficiency. We also need to know the most recent entry because our primary objective is to find optimal orderings of word entries. It is not reasonable to expect an agent to be trained on all possible states and actions because of the computational considerations discussed earlier.

3.1.2 Actions

Our action space is confined to selecting every individual word entry to be added to the grid. In our template this consists of 26 different actions to start with one fewer available action following each step because one additional word will be added to the puzzle. We wrote a method in Python to reset the state and game at the start of every episode. To simplify things for training, we always start with 2-down because it has many intersecting words and give very large Q-values to several other entries with a larger number of intersections.

3.1.3 Rewards

Similar to the grid world problem, we took a sparse rewards approach in which every action resulted in a score of -1 except for the final step. The ultimate goal of the agent is to minimize the number of steps in the episodes and find the shortest path to generating a complete crossword puzzle. Our agent reaches a "stalemate" and resorts to "thrashing" by removing and adding new words randomly until our agent can proceed. Each attempt at removing and adding a word results in two separate actions which result in a score of -1.

3.1.4 Policies

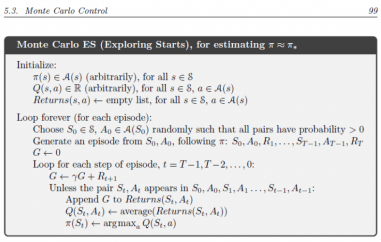
The policy that we are learning is a function of the state. Because of the vastness of the state space combinations, we only take into account a small subset of features when making a decision. Based on the most recently added word entry our agent will try to learn which word entry to add next. Our q-values keep track of all pairings of word entries. There are 26 different word entry and we keep track of the q-values of all pairings so there are 676 total (26 multiplied by 26) but not all are valid at any time. For example, if we already selected a word for 2-down then no other word entries can select this next state, regardless of how high the expected reward of the action may be. So our policy is not quite tabular in that it is a simple look-up in a table, there is the added validation step as well.

3.2 Tabular Methods

Because of the computational complexity of our problem we were fairly limited in how many different approaches we can attempt. Framing our problem and designing the simulators for our crossword puzzle generator we had limited time to explore different reinforcement learning techniques so we chose some of the simpler methods we learned in class. Similar to our tabular methods homework assignment, we designed a Monte Carlo approach similar to Exploring Starts and Q-learning.

3.2.1 Monte Carlo

The very first Monte Carlo method we built for our tabular methods homework assignment was Monte Carlo with Exploring Starts.


5.3. Monte Carlo Control 99

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

Figure 5: Monte Carlo Exploring Starts algorithm

The Monte Carlo simulations differ from temporal difference methods in that they generate an entire episode before iterating through to update policies. The Monte Carlo ES algorithm starts by randomly initializing states, policies, and q-values. It then loops through the desired number of episodes and the first time a state-action pair appear it will update the q-value.

Our approach differed slightly in that we always start by adding 2-down, so our initial state is fixed. Ultimately, this will not explore any starting points so the algorithm name is a bit of a misnomer for our approach. We then iterate through every state-action pair and update the Q-value the first time a pair is observed so that it is set to the averaged returns across all episodes. We then update our policy to greedily be the optimal available action.

Compared to Q-learning, Monte Carlo ES learns very slowly. Because it only takes into account the first time a state-action pair appear in an episode, no learning takes place during "thrashing" when the agent is struggling to break stalemates. This happens when our agent has taken a poor action so the penalty incurred does not get taken into account for future episodes. Monte Carlo ES takes much longer to converge but when it finally shows signs of convergence it outperforms Q-learning.

3.2.2 Q-learning

Q-learning is a temporal difference technique we discussed in lecture and applied to grid world and blackjack for the homework assignments. Temporal difference methods are distinct from Monte Carlo in that q-values are updated at every step. Q-learning finds the optimal based on the maximal expected reward over all future steps starting at the current state.

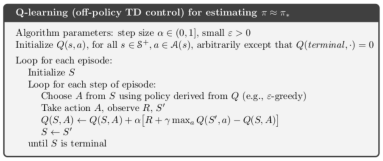


Figure 6: Q-learning algorithm

Our approach to Q-learning was very similar to what we did in class but with the added caveat that only a small subset of actions are available to our state at a given moment. We had to make a decision for what to do when our current state knows a particular action provides high reward but is unavailable. As we can see, the algorithm takes the difference between the best action and the action taken. When taking this difference we decided to greedily compare the best known action, regardless of whether or not it is available. If there are no known good actions available then we would not be learning very much at a given state because the difference between two low Q-values will be very small. However, if an action is bad we would prefer to amplify this difference which is why we compare to the optimal action.

Compared to Monte Carlo ES, our model converges very quickly. Because we are updating policies at every step and we take into account all state-action pairs, including repetitive pairs, we learn much more quickly. State-action pairs that appear repeatedly are most likely to be poor actions and Q-learning picks up on this while Monte Carlo ES does not. After several thousand episodes Monte Carlo ES eclipses Q-learning after the former has had the opportunity to more sufficiently explore different actions and states.

3.3 Deep Q-Networks

Deep Q-Networks(DQN) is another approach we tried. It is a combination of Q-learning with a deep convolution neural network. Similar to Q-learning, we update the q-value in state-action pairs to find a better learning result. For each state-action pair, the q-value is the sum of the current reward and the q-value of the next state, which can be expressed as:

$$Q(s, a) = r(s, a) + Q(s_{next}, a_{next})$$

Take both sides of the equation, our training goal is to minimize the difference of both sides. Thus, the loss function of our network is the mean square error of the

both sides. In this project, the implemtation of DQN is done through pytorch, so we just need to input the loss function as pytorch does the derivation for us. One feature of DQNs is the usage of a replay buffer. The replay buffer is used to memorize the transition after each action between our agent and environment, where the transition is:

$$transition = (state, action, next\_state, reward)$$

When the training happens, we randomly sample from the replay buffer to involve the Q-network training process. This allowed the network to accumulate experiences from past games.

An interesting point to mention is that the DQN uses ϵ-greedy in exploration. To start with, ϵ value is set to be 1, which is pure random. There is a decay rate of 0.001 for every step, and a minimum value of $\epsilon = 0.001$. This means after 1000 steps of decay, the agent has a probability of 0.1% of choosing random action, and 99.9% chance of choosing the output action of our Q-network.

The benefits of using DQN is that it involves deep artificial neural networks, which allowed us to handle some real complex problems. In our text book, it mentioned that DQNs can be used to play a variety of Atari 2600 games, this illustrates the flexibility of this algorithm.

In this project, the DQN used has two hidden layers, each layer has 64 hidden units, and activated by tanh as the activation function. The last layer that gives the Q output is activated with Relu.

4 Future Considerations

Because of time constraints we were not able to explore some of the more complex techniques we discussed in class. Some techniques we initially discussed exploring include actor-critic method, neuralevolution.

The actor-critic method was considered because it separates policy and value logic. We did this in a very limited scope but it would have been very interesting to see how neural networks, capable of exploiting non-linearities and more complicated patterns, would have performed. The value and policy functions we employed were fairly naive for such a complex use case so any kind of neural network-based approach would be very interesting to explore.

Another technique we wanted to explore was neuralevolution. One key takeaway that I had from the guest lecture on this subject is that genetic algorithms are more capable of generalizing in certain use cases then deep learning. A CNN trained for image processing on a particular group of photos will not perform well on unseen data whereas genetic algorithms are able to evolve slightly over time and explore different permutations. This could potentially yield very positive results when generating crossword puzzles if our agent can learn more complex behavior to work around complicated grid arrangements.

5 Final Results

As mentioned previously, our benchmark is completely random traversal through our environment until a valid crossword puzzle is generated. We will present several figures comparing performance of our agent against the benchmark.

5.1 Benchmark Performance

Here, we discuss the performance of the completely random steps through the environment.

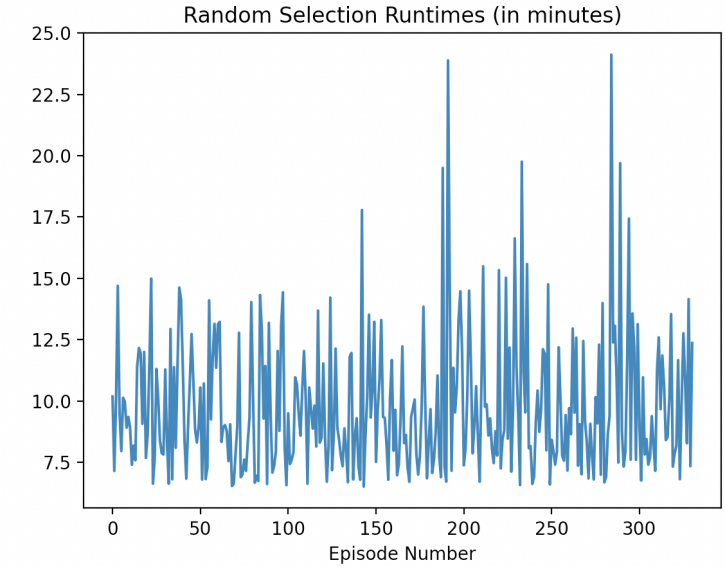


Figure 7: Random Runtimes (Over 300 episodes)

As can easily be understood, randomly adding word entries into a crossword puzzle and randomly removing words when at a stalemate is a very poor and inefficient matter of solving our problem. Because of this inefficiency, we were only able to run about 300 episodes using this random policy. As we can see from the figure, this takes at best 7 minutes per episode with median value just over 10 minutes. There are two major outliers where the agent using the random policy struggled for 25 minutes before finally solving the problem. It is easily understood that this is a very poor performance, next we will consider performance of our reinforcement learning approaches.

5.2 Monte Carlo Performance

Here, we consider the performance of our Monte Carlo method.

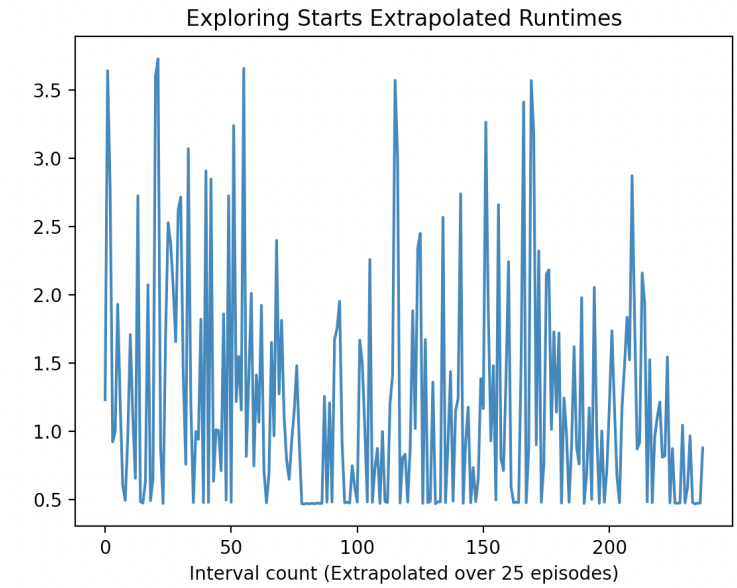


Figure 8: Monte Carlo Runtimes (Nearly 6,000 episodes, extrapolated over 25 episodes)

We extrapolated the average runtime over intervals of 25 episodes to smooth results so we can better see trends. The best case runtime values we see is about 28 seconds per episode. For runtime we removed outliers outside the 99th percentile because there are several extreme outliers, instances where our agent struggled for extended periods of times, which skewed our results. This approach has an average of about one minute per episode, a very significant improvement over the mean value of 10 minutes for our benchmark. We also see that while there are occasional spikes we do see a slow downward trend appearing over time. In our Monte Carlo approach it is slower to learn over time.

In the next figure we plot our episode length extrapolated over 25 episodes. Because there are 25 word entries in our template the best possible value is 25. Our total reward per episode is the negative value of steps because our goal is to minimize the episode length. We see some spikes but, similar to runtime, we see a slow downward trend over time with fewer spikes.

5.3 Q-learning Performance

Finally, we consider the results of our performance for Q-learning.

We see that, compared to Monte Carlo, the Q-learning approach learns very quickly and achieves near-optimal performance after several hundred episodes. Despite the initial learning, performance then drops over time and we see higher

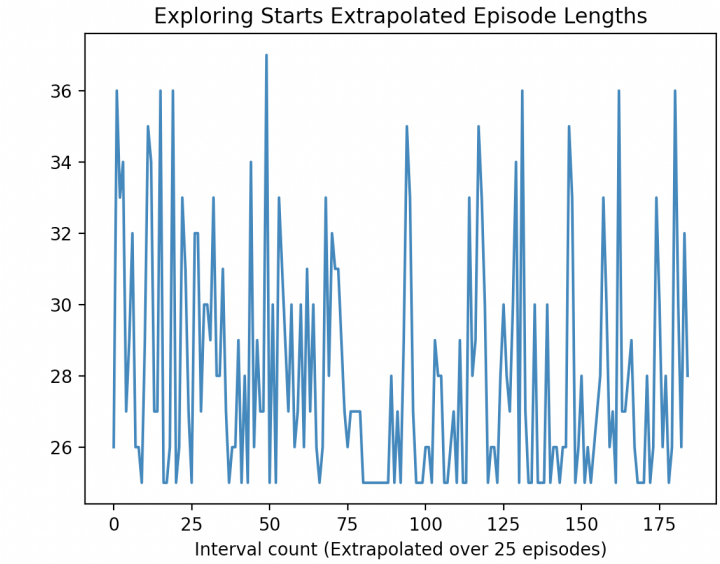


Figure 9: Monte Carlo Episode Lengths (Nearly 6,000 episodes, extrapolated over 25 episodes)

spikes in runtime after several thousand episodes. We are not sure what causes this drop in performance, we assume this is caused by "over-fitting" where our agent makes too many assumptions based on previous results. This is ironic given our earlier concerns about not having enough episodes completed for our agent to converge to optimal values.

The number of steps per episode is nearly perfectly correlated with the runtime chart we discussed previously. As we can see it quickly settles to near perfect performance before dropping in performance compared to our Monte Carlo approach.

5.4 DQN Performance

In terms of DQN results, we experimented with 100 episodes and 10 episodes, both with 25 steps within an episode.

As we can see, both cases result in an increasing reward, while in losses, 10 episodes are too less to converge.

The results are shown in Figure 14, as our episodes goes higher, the game board seems to be more complete. Fewer episodes tend to leave more blanks on the board. This is interesting that, although the loss did not converge, the game itself is playable, with the unsolved part an increase of difficulties for our human player.

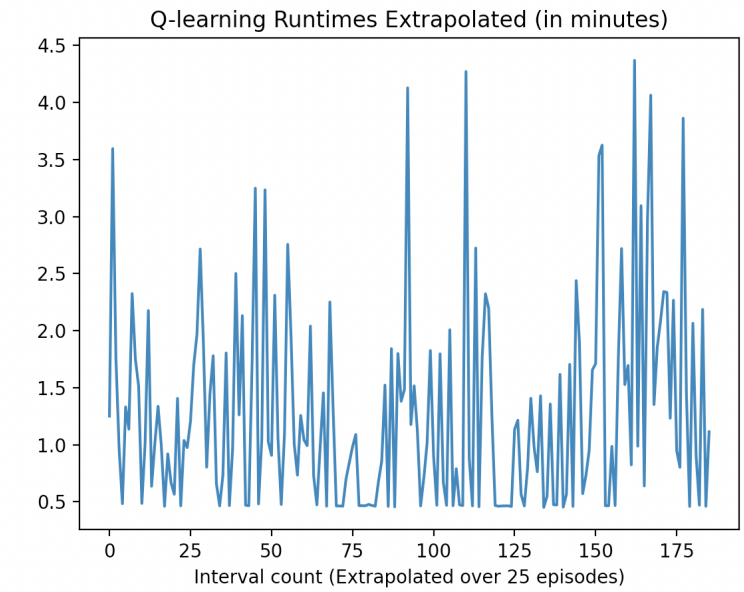


Figure 10: Q-learning Runtimes (Nearly 6,000 episodes, extrapolated over 25 episodes)

5.5 Comparison of Results

As we can see, our reinforcement learning approaches vastly outperform the completely random benchmark. There are several improvements that tabular methods offer over random walks. First and foremost, our agents are able to learn which actions lead to good results and which actions lead to bad results. Second, we also give our agents a slight boost by artificially inflating Q-values so they understand that word entries with a higher number of intersections should be prioritized. Despite these advantages we were very surprised to see our agents post a 90 percent improvement over our benchmarks. Trimming our training times down from 10 minutes to 60 seconds vastly outperformed our expectations and we have a difficult time fully justifying this astounding improvement. In the field of machine learning it is very common for researchers to struggle to justify results or performance. My personal area of interest is natural language processing and one of the biggest innovations our field has seen in recent years has been BERT, bi-directional encoder representations from transformers. Researchers were astounded by the performance improvement this technique showed in common problems and to this day we still can't quite understand how it works so well. What is interesting about our agents is that the tabular methods are, compared to BERT, very straightforward.

Next, we show a box plot of both of number of steps per episode (absolute value of total reward per episode) to see how they compare to one another. In

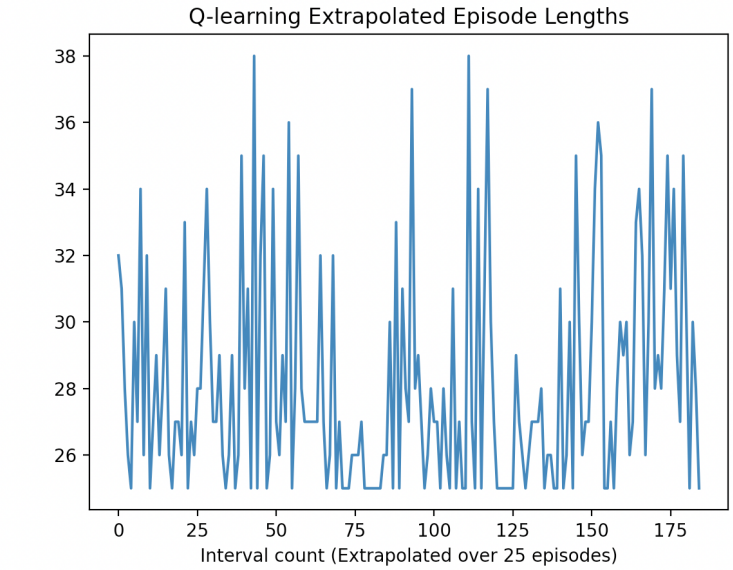


Figure 11: Q-learning Episode Length (Nearly 5,000 episodes, extrapolated over 25 episodes)

the box plot presented we filter out episode lengths above the 99th percentile because several extreme outliers skew the results to the extent that the figures are barely legible. As we can see from the rewards distribution, our Monte Carlo ES approach slightly outperforms the Q-learning approach. I think this happens because after awhile Q-learning starts to overfit based on past experience which is why we see more outliers in the plot. Most of those outliers from Q-learning come from latter episodes. If we look at the ES box plot we see more evenly distributed quartiles and slightly lower values across all depicted percentiles. Overall, it appears as if Q-learning learns quicker but after awhile performance recedes and ES eventually outperforms.

6 Practical Considerations

Through our experience with this project we encountered many practical issues that we did not expect. In previous homework assignments we applied the theory of reinforcement learning to relatively straightforward use cases but this project involved a more complex and practical use case. Because of the difference in scale of the underlying problem we encountered many obstacles that we did not expect and so we searched through academic literature to learn more about how these methods work in practice.

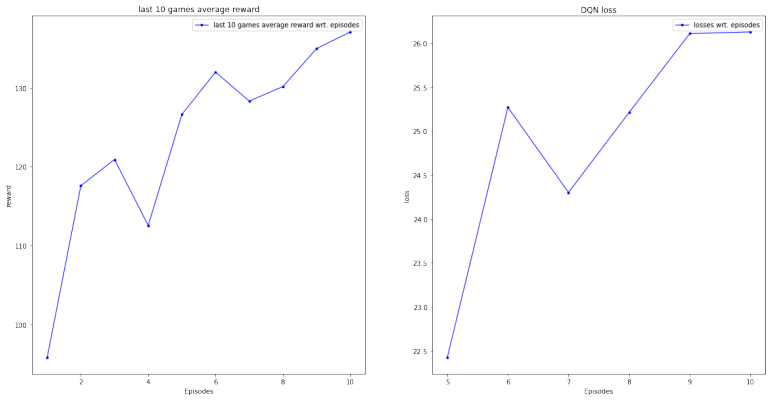


Figure 12: DQN average reward and loss(10 episode)

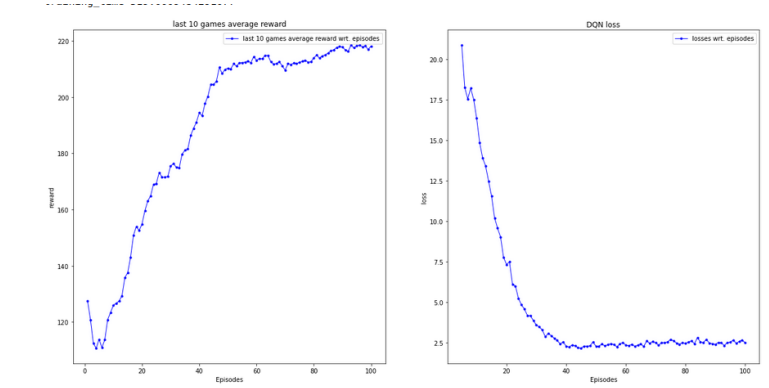


Figure 13: DQN average reward and loss(100 episode)

6.1 Computational Complexity

The time needed to train an agent is very, very long. In the homework assignments I was able to complete 1,000,000 episodes of blackjack and grid world for all of the tabular methods in several minutes using Google Colab. Unfortunately, it is rarely the case that a problem will be able to solved in such a short time. For my research I have been exploring some natural language processing models and training time can take days across large amounts of text data. To solve this, I generally scale down my input significantly and only training over several epochs. We took a similar approach by scaling down the size of our word bank and only training over several thousand episodes.

Our first simulations were taking about 90 seconds per episode so it quickly became obvious that it was infeasible to run as many episodes for this problem as we did with blackjack and grid world. Through optimizing the code I was

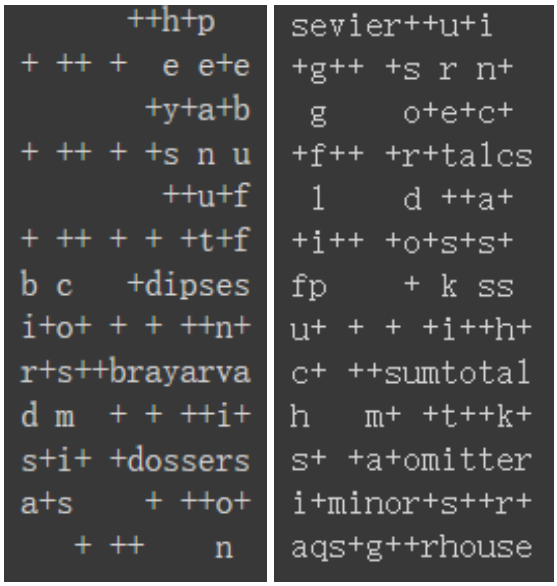


Figure 14: DQN result

able to get our best-case training time down to about 30 seconds but in general they average about 60 seconds. Because of the complexity of the problem we are only able to generate about 1,500 episodes per day if there are no underlying hardware problems or outages, but these outages frequently occur and require manually restarting. At the end of every episode I write the crossword puzzle that was generated to a text file and I record the q-values, episode list, and some other attributes to a JSON file. In the event of failure, because our previous episode's results are recorded we can easily pick up where the agent left off.

Long training times is a very common problem in the field of machine learning. In class we discussed a paper, "Asynchronous Methods for Deep Reinforcement Learning", in which the authors leverage four CPU's to speed up training. Their approach was very successful and they were able to cut down training time across an array of benchmarks from 8 days to 4 days. This speedup factor of two is tremendous but waiting 4 days for results is very costly. Our goal of 10,000 episodes would require about two weeks worth of training time, taking into account network failures and process timeouts.

6.2 Hierarchical Reinforcement Learning

As discussed previously, our reinforcement learning problem was not actually one problem but three distinct sub-problems. Our approach to designing crossword puzzles using reinforcement learning involved effective selection of words, efficient selection of grid locations, and capably resolving conflicts. In summary, the problem we tried to solve was actually composed of three smaller problems.

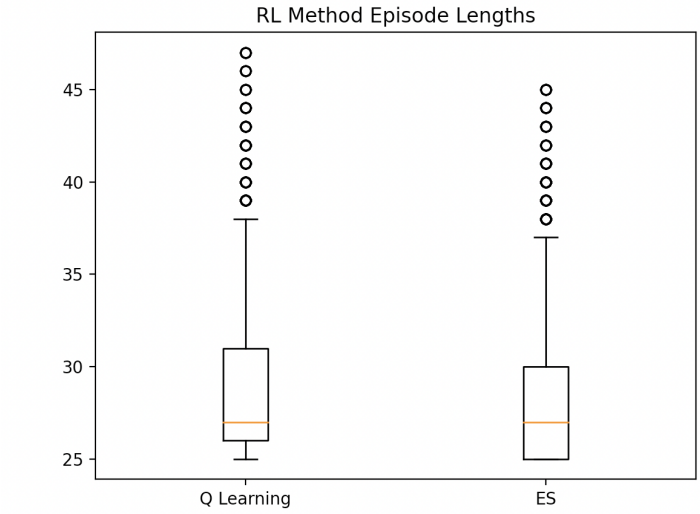


Figure 15: Number of steps per episode for our agents

This is not uncommon, authors in "Hierarchical Reinforcement Learning for Self-Driving Decision-Making without Reliance on Labeled Driving Data" designed a self-driving car that they decomposed into three smaller problems: driving in lane, right lane change, and left lane change [4]. Hierarchical reinforcement learning is a very interesting concept and is growing in popularity, I think this applies to more reinforcement learning problems than I initially thought.

6.3 Exploration-Exploitation Trade-off

In attempting to calculate how many different permutations of state-action combinations our agent would need to explore it quickly became apparent that this is not possible. This was not a problem we encountered in previous assignments. There are a very small number of possible states in blackjack and episode lengths are generally pretty short, while I didn't attempt to solve it using dynamic programming to thoroughly explore all possibilities I imagine it wouldn't take too long. Using brute force in grid world with a 10x10 board took several minutes but ultimately we were able to solve it using dynamic programming. Not only is exploring the full state and action space infeasible, it can be quite dangerous in the case of physical systems. We read "Safe Model Based Reinforcement Learning With Stability Guarantees" which goes as far as to propose safety boundaries for which states and actions can be explored. It is easy to understand how this is necessary for use-cases like self-driving cars. It would be very unsafe to explore what would happen if a car was in a state of driving at a speed of one hundred miles per hour and then to take the action of jerking the steering

wheel sharply to the right. In practice only a very small subset of state-action pairs in reinforcement learning problems can be explored.

6.4 Measuring Success

Based on the above results, our ultimate measure of success is the number of steps per episode. For reasons described earlier, runtime is subject to fluctuation because I run CPU-intensive applications on the same hardware as our agent so if compute resources are being consumed heavily by something like Docker then the Python process running our simulation will slow down. Total reward is the number of steps per episode which is why we chose to represent this as a sparse rewards problem. Our benchmark of comparison, the "completely random" trajectory, yields very long episode lengths compared to our reinforcement learning approach because our agent is trained to more methodically add and remove words. We look at various percentiles of episode length across all methods because there are extreme outliers which can skew results. Overall, we found that our methods were much more successful than we would have expected. While we do offer some explanations for this we find that the difference between the benchmark and our reinforcement learning agents is much larger than we can reasonably explain.

7 Key Takeaways

Throughout this project we have gained a lot of experience of practical applications of reinforcement learning. Some of our key takeaways include the difficulty of framing a reinforcement learning problem, the importance of preexisting knowledge, and the lack of bias compared to supervised and unsupervised learning.

7.1 Problem Statement Complexities

One of the biggest challenges we faced was framing our problem in the context of reinforcement learning. Our definitions of "state", "action", "reward", and even what we were training our agent to learn, was constantly changing throughout the semester. For example, we initially tried encapsulating multiple dimensions of "state" into our Q-value functions but this quickly became way too difficult. As mentioned earlier, there are many trillions of possible permutations for our state-action pairs and this is simply too large to include in a tabular data structure for look-up. We then reduced it to all possible word entry permutations and this proved very effective.

I am particularly interested in reinforcement learning applications in self-driving cars and I read several papers on this subject to try and learn how researchers frame their problems. Unfortunately, authors generally don't divulge much information on this subject so I had limited success in understanding how complex reinforcement learning problems are framed.

7.2 Pre-training Agents

Initial attempts at solving this problem with randomly generated Q-values yielded poor performance with regards to convergence. It took very long and we weren't able to successfully train an agent in a reasonable time frame. I then decided to artificially inflate Q-values for a few "good" actions so our agent was able to learn more quickly. We discussed whether or not we considered it "cheating" to give an agent pre-existing knowledge and there are some within the reinforcement learning community who do believe this is the case. After reading opinions on both sides of this debate we arrived at the conclusion that it is very impractical, and sometimes dangerous, to not give an agent information beforehand. For example, if researchers were to try to train an agent to drive a car with no prior knowledge and were to let it randomly explore the state-action space it can lead to very dangerous, and possibly fatal, results. We chose to leverage pre-existing information in this project to speed up training time and we think that this is not "cheating" and should in fact be encouraged.

7.3 Reinforcement Learning vs. Alternative Learning

A significant problem plaguing the field of machine learning is bias in data sources. While the models generated by supervised and unsupervised learning are not prone to any form of bias the underlying data sources are. This presents problems because machine learning is applied to very sensitive use cases such as approvals for loans and prison sentences. Minorities in the United States are disproportionately likely to be denied for a loan or sentenced to a harsher prison sentences and alternative learning approaches that rely on input data will pick up on this and "learn" to associate protected attributes (race, gender, etc.) with outcomes. Reinforcement learning is much less prone to this form of bias because agents are learning to explore an environment as opposed to mapping inputs to outcomes. In our use case, our agent was fed a crossword puzzle template and a very long list of input words. Obviously the list of words present a minor form of bias but we believe that the sheer size of the word bank effectively removes any potential human bias we may have introduced.

8 Future Plans

This was a very enjoyable project and we learned a lot from it. As previously mentioned, throughout the semester we had to scale down the scope of our project to meet deadlines. I chose this topic because it is aligned with my main area of interest which is natural language processing. Some future plans of mine include training an adversarial agent to solve crossword puzzles using deep bi-directional encoder representations from transformers (BERT) to gauge context from word clues. While we didn't get as many episodes completed to compare results we found that our agents converged relatively quickly and we were able to gain sufficient performance results so we don't see a need to continue running

more episodes. I look forward to publicly releasing my findings from this project and releasing our code to my public GitHub profile.

9 Member Contributions

For this project both group members, Mark Bentivegna and Zichen Wang, framed the problem statement and identified the dataset used for this project. Because Mark has a stronger Python background he wrote the crossword puzzle simulator and code to analyze results and generate figures. Mark also generated the tabular learning methods while Zichen generated the DQN code. Both members contributed to the generation of this report.

10 Citations

[1] Berkenkamp, F., Turchetta, M., Schoellig, A. and Krause, A., 2017. Safe Model-based Reinforcement Learning with Stability Guarantees. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1705.08551¿ [Accessed 2 May 2022].

[2] Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, H., Van Overveldt, T., Petrou, D., Ramage, D. and Roselander, J., 2019. Towards Federated Learning at Scale: System Design. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1902.01046¿ [Accessed 2 May 2022].

[3] Emuna, R., Borowsky, A. and Biess, A., 2020. Deep Reinforcement Learning for Human-Like Driving Policies in Collision Avoidance Tasks of Self-Driving Cars. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/2006.04218¿ [Accessed 2 May 2022].

[4] Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016. Asynchronous Methods for Deep Reinforcement Learning. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1602.01783¿ [Accessed 2 May 2022].