Final Report for Computational Economics Project

Reinforcement Learning Applied to Adversarial Manipulations of MDPs

Josh Palmer
Artificial Intelligence and Visually Analogous Systems
2301 Vanderbilt Place PMB4632
Nashville, TN USA
Project Code: https://github.com/Kelym/adversarial-reinforcement-learning/tree/player
joshua.h.palmer@vanderbilt.edu

ABSTRACT

Over the past semester, Liyiming Ke and I have been researching Reinforcement Learning (RL) to see how it can be applied to solve Markov Decision Problems (MDPs) where an adversarial agent manipulates how a player perceives their world. We focused our efforts on a toy game — an n x m grid that a player must navigate to find the goal whilst avoiding traps and falling victim to the adversary's manipulations. At first, we experimented with table-based Q-Learning (an algorithm for RL). However, even for small (n,m), the state space is much too large — such a table with Q-values enumerated over all possible states cannot fit in memory and the time required to visit each state enough times for learning to occur is prohibitive.

Therefore, we switched over to Deep Q-Learning Networks (DQNs) which are able to handle this large state space. In the week leading up to this paper's due date, my job was to train and fine-tune a DQN for the player. Ke's job was to develop a DQN for the adversary, that given a limited budget and the player's network, would perform "bit-flips" on trap squares that would minimize the player's cumulative reward; therefore, this would have been a zero-sum game. Our goal then was to develop opposing DQNs that could learn from one another in a generative adversarial context (a la Goodfellow et al.[1]or Uther and Veloso[2]. However, Ke was overloaded with other schoolwork and I committed myself to a last-minute weeklong fast. Therefore, only the player's DQN was "finished" — after spending an entire weekend (30 hours) of playing around with hyperparameters and network topologies, some learning occurred but certainly not to a satisfactory level. I mention all of this neither to get "pity points" nor attempt to lay blame on my partner; I recognize this project should have been worked on throughout the semester. My purpose is to recognize that these results are underwhelming and share my frustration over not turning in a report with results I can be proud of. The silver lining is that I learned a lot about DQNs in the process and look forward to continue to experiment with this project after the due date.

[1]http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[2]http://reports-archive.adm.cs.cmu.edu/anon/anon/usr0/ftp/usr/ftp/2003/CMU-CS-03-107.pdf

1. INTRODUCTION

RL is a new subject for some; as such it would be useful to define it by outlining the three characteristics associated with it: [3] (1) "being closed-loop in an essential way" (i.e. "[the] learning system's actions influence its later inputs"), (2) "not having direct instructions as to what actions to take", and (3) "the consequences of actions [playing] out over extended time periods".

We are interested in the application of RL to adversarial manipulation of MDPs because, as our world increasingly becomes an "Internet of things", cyber attacks become more and more debilitating. For example, imagine an adversary hacking a self-driving car; the software needs to have learned what to do in such an attack. Recent related literature has included the following:

- An adversary made minor pixel-wise changes to images that are imperceptible to humans and yet "induced misclassification by a CNN"[4] (2014)
- A response to the above paper that critiqued it as being unrealistic because the adversary would require access to the CNN, whose abilities do differ from those of a human. Furthermore, **all ML algorithms** are susceptible to such manipulations. [5] (2015)
- RL has been applied to zero-sum, symmetric adversarial games — Othello/Reversi[6] (2010) and an RTS called Wargus. [7] (2012)

At first, we did not encounter the application of RL the case of an asymmetric adversarial MDP, where the player and adversary are distinct in their goals — the player seeks

[3]Source: https://webdocs.cs.ualberta.ca/ sutton/book /bookdraft2016sep.pdf; page 2

[4]Source: http://cs.nyu.edu/ zaremba/docs/understanding.pdf

[5]Source: http://www.kdnuggets.com/2015/01/deep-learning-flaws-universal-machine-learning.html

[6]Source: www.intelligence.tuc.gr/lib/downloadfile.php?id=433

[7]Source: http://www.aaai.org/ocs/index.php/AIIDE/ AIIDE12/paper/viewFile/5515/5734

to maximize their utility from their environment (as if there were no adversary) while the adversary surreptitiously manipulates this environment and/or the player's perception of it. Thus we felt it was a newer problem worthy of investigation. However, since then we have found a paper[8] titled "Adversarial Attacks on Neural Network Policies" that we intend to draw from as we develop our own adversary.

We began by designing a simple toy grid-based game.[9] The player's goal is to reach the goal square, all the while avoiding hazardous trap squares. Their only perception comes from guessing a move and receiving a reward corresponding to how good that move was. Furthermore, the states to which the player travel to determine the set of actions and decisions it has to make later. The delayed reward of the goal state, combined with the fact that actions have consequences later on down the road lends this problem to be readily solved by RL.

Thus we first implemented Q-Learning to solve an even simpler version of the game that had neither restarts nor adversaries. Next, we restarted the player in a brand new random game upon each successful navigation to the goal. To learn the Q-function and policy required that we extend our state representation to include more than just current position. This however led to an explosion in the size of the state space.

Therefore, we were motivated to switch over to Deep Q-Learning Networks (DQNs) which are able to handle this large state space. My job was to train and fine-tune a DQN for the player. Ke's job was to develop a DQN for the adversary, that given a limited budget and the player's network, would perform "bit-flips" on trap squares that would minimize the player's cumulative reward. Of this report's writing, I only have results concerning the player's DQN. My player network was guided by two variations on the Cart-Pole problem[10][11] as well as a gentle introduction to Deep RL[12]. For this, I fixed the maze size at (4,4) and generated random mazes where on expectation one third of the squares are traps distributed randomly across the maze.

After over 30 hours of experimenting with hyperparameters and network topologies, the best performance I can get is one that hovers around 25% success rate on sets of 100 randomly generated test mazes. I will explain what I experimented with as well as why this success rate seems capped at 25%.

2. PROBLEM DESCRIPTION AND APPROACH

Our project focused on application to a toy game model, as described below, with the inital non-DQN approach in mind and including initial plans for an adversary:

- an n x m grid that contains the player's initial position, the player's goal, and randomly distributed "traps" in which the player will receive a strongly negative reward and have to restart the same game
- at each time step, the player must decide a valid direction to move and then carry out that move. The player is not very perceptive, only "knowing" where the traps are if they have discovered them on a previous attempt
- at each time step, the adversary can manipulate the player's perception of the game state — altering where they believe the holes to be. Adversary details include:
  - Its state includes both the player state and the "remaining budget" it has to wreak havoc
    - **Note:** We have to decide what a fair total budget should be (e.g. can only manipulate three squares in a given game)
  - It has access to the player's Q-function / policy and thus will use it to inform its decisions
  - Its purpose is to maximize a particular heuristic designed to make the player take as long as possible to get to the goal state
  - One possible heuristic is maximizing the difference between the cumulative reward of the optimal path and the path that it suggests the player take vis-a-vis its manipulations
    - In achieving this end, we will look to the aforementioned paper "Adversarial Attacks on Neural Network Policies" for guidance. It details how the adversary uses a DQN of its own to learn what moves should be made to hinder the player as much as possible.
    - **Note:** It knows what changes it needs to make to suggest a path to a player because it knows the player's Q-function / policy

We decided to use Q-learning, one specific flavor of RL, because "it doesn't require knowing the dynamics of the environment" and learns quicker through bootstrapping ("estimating over existing estimates").[13]The first point is especially important because the player is unsure of how exactly the game state is changing.

As mentioned in the introduction, we implemented Q-Learning (sans adversary) on a static game. State was simply current position and the algorithm ran a fixed number of iterations, continuing to learn even after the goal was found. The initial values of the Q-function are randomly generated. Then an agent (i.e. the player) explores the environment. Actions are chosen based on the "curiosity" of the agent: curiosity is the probability that the agent will choose a random action vs. following the action suggested by the current policy. If the move is valid (i.e. inside grid) and does not move the player to a trap, then the action is carried through. Even after achieving the goal, the player continues to move until In the algorithm We then developed a similar game where upon each successful discovery of the goal, we started over with a new random game. Traps are randomly placed based on a binomial with probability selected uniformly from 0 to 0.33 (in the DQN, I fixed this probability at 0.33). The start and goal squares were then randomly placed. With this, we had to rethink the state representation so that with each new game, it could start learning to recognize common

[8]https://arxiv.org/pdf/1702.02284.pdf

[9]https://github.com/Kelym/adversarial-reinforcement-learning/tree/player

[10]http://pytorch.org/tutorials/intermediate/ reinforcement_q_learning.html

[11]https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials

[12]https://www.nervanasys.com/demystifying-deep-reinforcement-learning

[13]Source: http://www.aaai.org/ocs/index.php/AIIDE/ AIIDE12/paper/viewFile/5515/5734

states of knowledge (e.g. if there's a trap to my right where should I go?).

We encoded the state as current position plus a dictionary pairing known squares with known values. However given a game of size = m x n, the number of states is

$$size^2 * 3^{size} \tag{1}$$

This quickly blows up even for the simplest 3 x 3 games. In this case there are 1.5 million states, which means it takes way too many iterations for the value of a state in the Q-function to even be updated, let alone converge to the optimum value.

Again, this is when we turned to Deep Q-Learning Networks (DQNs), which are able to handle this large state space. I focused on training and fine-tuning a DQN for the player (one that is adversary-unaware). I implemented this in the relatively new ML framework, PyTorch.[14] I changed the player's internal state (i.e. the information it "knows" by "observing" its environment that is fed into its neural network) so that it was "pseudo-omniscient"; the player sees the entire maze and all of its values but doesn't know a priori what these values mean. For example, there is a boolean isTrap matrix, identifying which squares are traps; however, the player has to still regularly make the mistake of moving to a trap and receiving a large negative reward for it to know that traps should be avoided at all costs.

I experimented with a variety of hyperparameter settings and network topologies. Across the board, I used a discount rate of 0.9 and the network consisted of (one or two) convolutional layers followed by (one or two) linear (i.e. fully-connected) layers. My intuition was that the initially the most important concept for the player to learn is trap avoidance; there is naturally a spatial correspondence (as there is for navigating mazes in general). Thus, I wanted to capture the information that a neighbor is a trap by using a filter of size 3x3 and calculating this filter at every board position (I used padding of 1). Then once the idea of neighboring traps had been established, the fully connected layer(s) could then give intuition about where the goal was in relation to the player.

Motivated by this paper,[15]I used batch normalization after each convolution; as the paper explains, this avoids the need for dropout and in general it is good practice to use between linear (e.g. convolutional) layers and non-linear (e.g. ReLu) ones. I used ReLu to avoid the vanishing/exploding gradient problem.[16]Admittedly, this should not be a problem given that my network is relatively shallow; it would be an interesting next step to experiment with the activation function. I used Huber loss (i.e. smooth L1 loss) as an arbitrary choice (the code I used as a guide also used Huber loss). I used Stochastic Gradient Descent since it is canonical and I am most familiar with it; in later experiments I started to use momentum=0.9 and weight_decay=1E-4 at the suggestion of a brilliant fellow labmate[17] and saw an increase in performance. Unless where specified, I used a mini-batch size of 50 and nSteps = 1000 (per epoch). After each epoch, I tested the model on a suite of 100 randomly generated mazes. There are myriad parameter permutations to experiment with as a next step; I could use cross-validation to narrow these down.

[14]http://pytorch.org/
[15]https://arxiv.org/abs/1502.03167
[16]http://neuralnetworksanddeeplearning.com/chap6.html
[17]shoutout to Max DeGroot

The precise state representation was mostly the same across the board as well. I used three one-hot encoding channels to represent the board state. These were the position, trap, and goal channels. In the position/goal channels, there was a matrix that was zero everywhere except for the location of the player/goal, where the value was 1. In the trap channel, 1's denoted the locations of traps — 0's otherwise. I could have flattened all this information to one channel but then the matrix elements would have been categorical as opposed to ordinal. My intuition was that the more structured and ordered the input, the better the performance of the network; however, a flattened state could still lend itself to the problem so it would be another interesting next step. For one trial, I did experiment with including âĂIJhistoryâĂİ as a fourth channel (i.e. 1's denoted previously visited squares).

Lastly, across the board, I maintained that the trap probability (in generating random mazes) was 0.33 and that the rewards for moving to goal, trap, blank squares were +100, -100, and -0.5 respectively. The cost for moving to a blank square was negative to encourage the player to make its way to the goal in as few moves as possible. To allow for more experiences on the same maze, if the player in training fell into a trap, it would âĂIJrespawnâĂİ at its most recent state with the knowledge gained from the negative reward. In testing, a player falling in a trap restarted the maze and a failure is recorded. A similar policy was in place for players who took âĂIJtoo longâĂİ (i.e. took more steps than there are squares) to complete the maze.

The parameters I focused on manipulating were nEpochs, curiosity, learning rate, and specificity of SGD parameters (i.e. momentum and weight_decay). For the uninitiated, curiosity refers to the probability that the player will choose a random next state to explore as opposed to the one suggested by its policy (i.e. the max output of the network). A general rule of thumb is that early on it's good to maximally explore the state space (i.e. curiosity = 1). Then, as the Q-function begins to converge to its optimal value (a la the Bellman Equation), it's outputs are more accurate and thus should be considered when making moves (i.e. curiosity < 1). This is explored in the results section. In addition to different numbers of convolutional and linear layers, I also experimented with softmax. At one point, I made traps and goals terminal states w.r.t. to how targets are calculated (details of how targets should handle terminal states in an aforementioned article[18]).

All reported trials were run on Intel Xeon(R) CPU E5-1620 v2 @ 3.70GHz x 8. Thus, running times correspond to running on this CPU.

3. RESULTS

The experiments with the Q-value table for static games as well as with changing games were for our benefit to get more comfortable with Q-learning. Thus, the results are not important and as such are left out.

Figures 1 and 2 show the results of training for 5000 epochs using a constant curiosity of 1.0 and learning rate of 0.01. The topology of the network is one Conv2d(input=3 channels, output=4 channels) layer and one Linear (Fully-

[18]https://www.nervanasys.com/demystifying-deep-reinforcement-learning

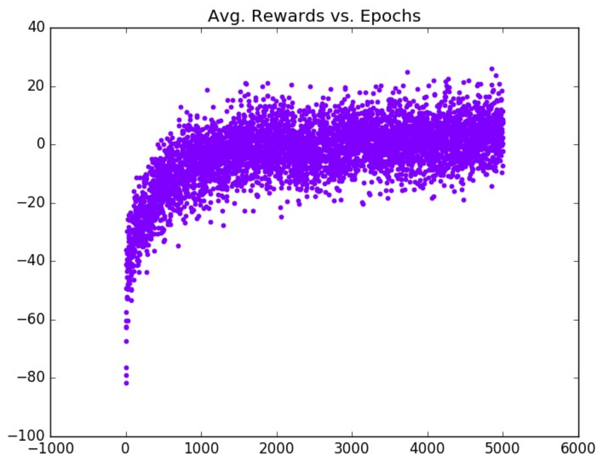

Figure 1: Average Cumulative Rewards, Curiosity = 1

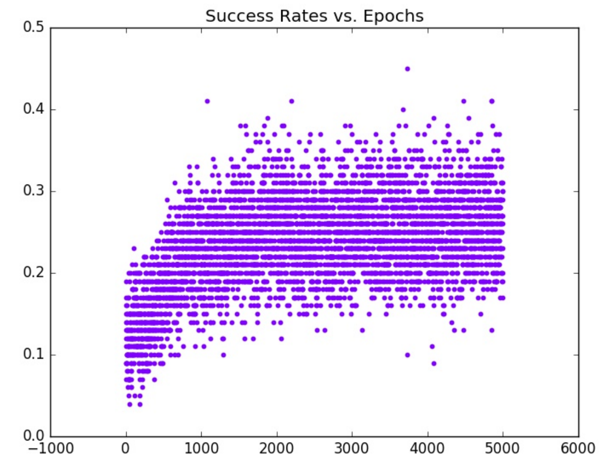

Figure 2: Maze Success Rates, Curiosity = 1

Connected) layer. (This is the topology used in the majority of trials.) The total training time was non-trivial: 14682 seconds. (As an aside, I leveraged PyTorch's user-friendly GPU framework to try running my code on the GPU in my lab. There was no noticeable speedup (i.e. total GPU running time within 1% of CPU running time). This is likely because my network is not terribly deep so most of the processing occurs on the CPU no matter what, outside the network feed-forwards and backprops. As a next step, I could try transforming these Numpy operations to Tensor ones.)

As the number of epochs increases, the average cumulative rewards vary widely about a stagnant slightly positive equilibrium. The maze success rates vary even more widely about 0.25. Given that on average there are only 25 successful mazes out of 100 and the average cumulative reward is slightly positive, this suggests that the 75 failures are mostly due to timeout (the player taking too many steps to get to the goal). If the failures were mostly due to traps, then the average cumulative reward would be more skewed towards -100. Thus, these graphs indicate that the player readily learns to avoid traps, but then is not encouraged to move towards the goal; it seems to move about blank squares indefinitely.

This hypothesis was confirmed when I ran tests on the model in debug mode, watching what the player does on a move-by-move basis. More often than not, the player first moves to a non-trap space and then back to its starting space, back to the same non-trap space, ad infinitum. Why the 25% success rate then? Well when randomly generating a maze, there is roughly a 25% chance that the player will spawn right next to the goal; the network is at least good enough to learn to move to the goal when it is the player's immediate neighbor.

My first immediate thought was that the network was not deep enough to learn the full path. I changed the network to include two Fully-Connected layers. However, the performance was noticeably worse — peak success rate was 0.15. Two Convolutional layers with one or two Fully-Connected layers similarly performed worse (and not to mention took significantly more time to run). For interest of space, I have left such plots out of the paper.

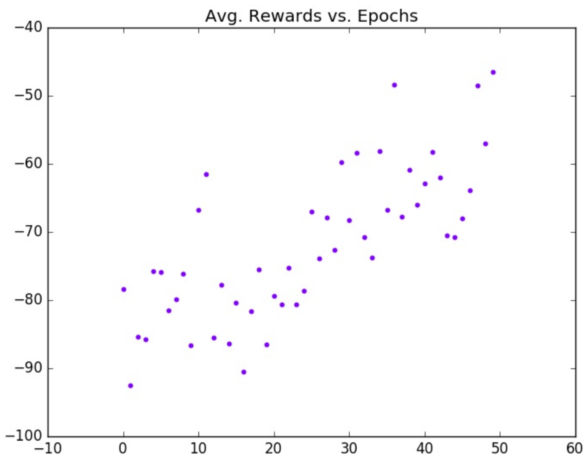

Figure 3: Average Cumulative Rewards, cur = 1, lr = 0.01

My next instinct was to run a suite of trials varying learning rates and curiosities — in interest of time I only trained for 50 epochs (each trial took 200 seconds). Fixing curiosity at 1, Figures 3 and 4 show the average cumulative rewards (success rates not shown for space reasons) for learning rates of 0.01 and 3 respectively. Notice that the rewards start to level off when lr=3, whereas they continue to grow for the lr=0.01 case. Next steps could improve this analysis by having more gradations, but from just these two data points it's clear that a lower learning rate is more a propos in this network. Note: although the lr=3 case had higher average reward, it started at a higher point, which can be chalked up to the randomization of weight initialization.

I then fixed the learning rate at 0.01 and varied the curiosity: 1 (Figure 3), 0.95 (Figure 5), 0.9 (Figure 6), 0.4, and 0.1. In terms of average rewards, cur=0.95 trended positively until turning around at epoch 30, cur=0.9 behaved erratically (no positive trend), cur=0.4 and cur=0.1 behaved

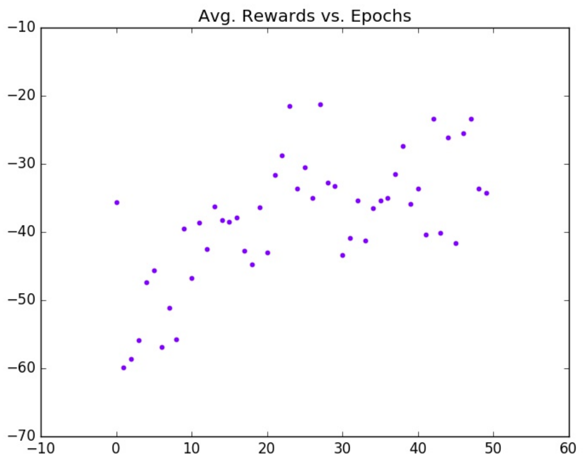

Figure 4: Average Cumulative Rewards, cur = 1, lr = 3

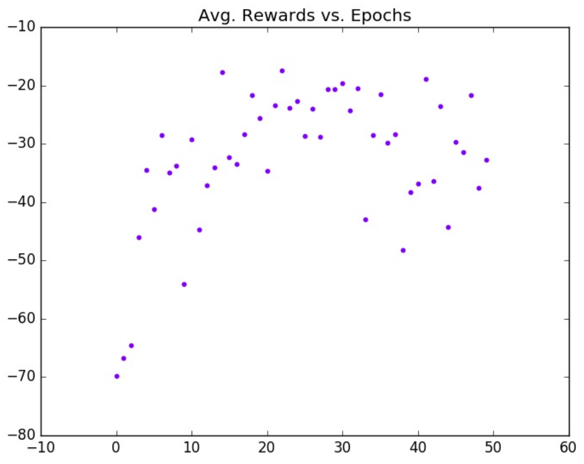

Figure 5: Average Cumulative Rewards, cur = 0.95, lr = 0.01

similarly erratically. Therefore, cur=1 performed the best (as expected given that maximal exploration is favorable in earlier stages).

From all these initial trials, I concluded that lr=0.01 and cur=1 with a network topology of 1 conv2d and 1 fully-connected layer were good initial guesses (at least better than the other options I tested out in these trials). However, this meant I still wasn't improving performance.

One final idea I had was in a sense "cheating" — updating the state representation to also keep track of previously explored squares. My intuition was that if the player knew where it had been previously (and I made it so that moving to said squares incurred a hefty penalty), then it would stop continuously looping back and forth between two squares and instead venture out, hopefully in the direction of the goal. This is "cheating" because ideally the Bellman Equation (assuming large enough discount factor) would propagate back goal rewards as to allure the player towards a goal.

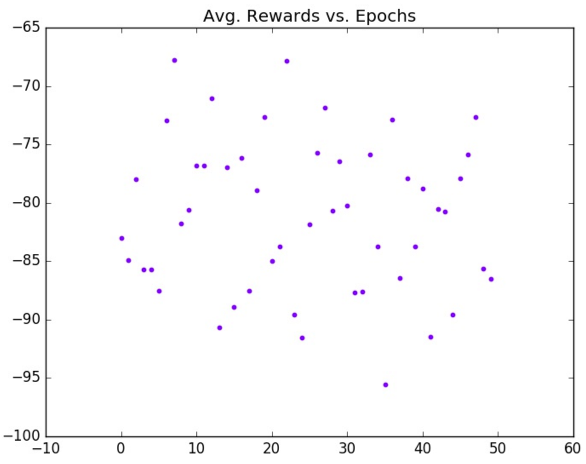

Figure 6: Average Cumulative Rewards, cur = 0.9, lr = 0.01

Thus, by penalizing revisiting squares, I was in a sense artificially trying to help the update rule (in the DQN case loss function) do the job it was failing to do. However, the point is moot. Performance actually suffered (a la Figure 6). My best guess is that the addition of another channel necessitated a change to the network topology; thus, if I played around more with the network, I might have been able to get better performance. But I wanted to focus my attentions on achieving success through the original, "fairer" state representation. On a similar note, I also tried removing the softmax layer, thinking that perhaps the rewards were dominating the next state network outputs in the target calculations. However, once again my rewards mirrored that of Figure 6.

I then made a noticeable performance leap (at least in terms of rewards converging faster to its slightly positive equilibrium value seen earlier — I still wasn't getting success rates consistently above 0.25). This "leap" arose by making two changes:

- Updating the target calculation to reflect goals and traps as terminal states (explained more thoroughly in previous section
- Updating the SGD to include momentum=0.9 and weight_decay= 4

Momentum addresses the issue in SGD where the updates fly down steep "canyons" but then crawl on nearly level "valleys". Weight decay (a la L1 regularization) prevents the weights from getting too large (large weights make the network more sensitive to individual training examples which is unproductive when trying to generalize to future instances). Again these values were suggested to me by my labmate Max DeGroot, who has significantly more experience with deep learning than I do; I tried them and they "magically" improved performance. I'm sure that I could find more optimal values by experimenting more.

With this performance increase, my final strategy was to dynamically change the curiosity during runtime; I would start at cur=1 and lower it gradually so that the player

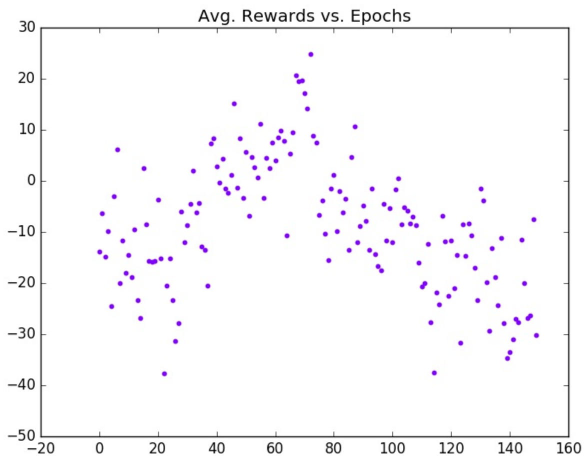

Figure 7: Average Cumulative Rewards, cur = dynamic, lr = 0.01

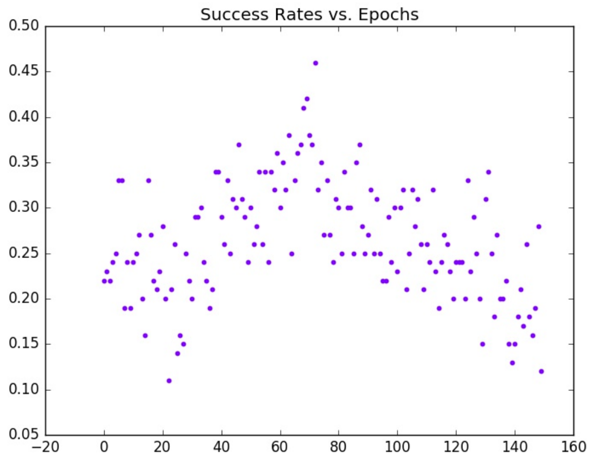

Figure 8: Maze Success Rates, cur = dynamic, lr = 0.01

would rely on the model more and more. My most successful (and dramatic) multi-trial occurred when I first trained for 150 epochs (reducing the curiosity to 0.9 at epoch 50) and saved the model weights. I then started training again (starting at cur=0.9 and reducing it to 0.8 after 50 more epochs (i.e. 200 total epochs). Figures 7 and 8 show the results during this second training session. I reach new (consistent) success rate records and relatively higher rewards halfway through. I assumed my stepwise curiosity reduction training was working and that I'd finally achieve better and better performance. Alas these hopes were diminished when both metrics took a sudden turn. One possibility is that training for so long leads to overfitting. Another is that I need to play around more with how I decrease the curiosity. The most likely scenario is that the way the network is designed, it's unlikely for the player to learn to move towards the goal; it seemed more promising with this training session, given the consistent performance above a 25% success rate. While deepening the network didn't help (at least how I tried), it'd be good to try to widen the network. If I had more time, I'd play around more with the topology.

Fourth-wall break: There are some more results to share here but I have 30 minutes to turn this in! I've shared the most important of these results however.

4. RELATED WORK

As mentioned in the introduction, there were several papers we looked at whilst formulating our problem. One outlined an adversary that slightly changed pixels to induce errors in image classification. Others have applied RL to zero-sum, symmetric adverarial games (Othello/Reversi and Wargus). Again, our approach is novel because of its asymmetric nature (the goals of the player's and adversary's policies are distinct).

As aforementioned, after starting the project, we encountered a paper entitled "Adversarial Attacks on Neural Network Policies"[19] that concludes that even small changes (imperceptible to the average human) to policy inputs can have devastating impacts on the efficacy of a network learned via RL. One difference we noted is that these perturbations are introduced as the policy is learning. On the other hand, we planned on only introducing the adversary once the policy is done learning.

The article is particularly useful because it outlines two possible adversarial implementations — the Fast Gradient Sign Method (FGSM) and the choosing of a norm constraint so that adversaries don't change too many variables by too great of an amount as to make the task impossible. It also characterizes how vulnerable a network is to both black-box attacks (adversary doesn't know underlying network and thus can't utilize it to fully maximize damage) and white-box ones (omniscient adversary). Since our adversary will have access to the player's Q-function, a white-box attack more characterizes our situation. We will definitely take this under advisement when we develop our adversary.

Just a few days ago, I discovered this project on GitHub[20] that is nearly identical to what I worked on in this paper, except that it works. It teaches a player to navigate a maze to achieve a goal. However, it doesn't take into account adversarial manipulation. And on a subtler note, it implements Value Iteration instead of the Bellman Equation (which gets reformulated as loss in our DQN context). Thus, if we later successfully implement the adversary and have the player and it learn from one another, we'll have accomplished something novel (as far as I am aware).

5. CONCLUSION

In summation, we decided to tackle Reinforcement Learning as it related to Markov Decision Problems and their manipulation by adversarial agents. We were motivated due to the susceptibility of technology to attack in a world in which we increasingly depend on said technology in multifaceted, intersecting ways. We also identified a gap in the research when it comes to the application of RL to asymmetric games. Thus we wanted to contribute to the field by examining a simple toy grid-based game to see how a player

[19]https://arxiv.org/pdf/1702.02284.pdf
[20]https://github.com/kentsommer/pytorch-value-iteration-networks

and adversary would interact (and possibly learn from one another).

Admittedly, we didn't accomplish anything new because we failed to develop the adversary in time for this paper's due date. However, we have some solid next steps. First and foremost, I can spend more time examining the topology of the player's network — there's no sense in having a adversary if our player can't even find its way without interference. After all of the tests I performed, I'm confident that the topology is to blame. I know maze navigation is possible in an RL context (the GitHub project mentioned in Related Work is proof of this). As I get more well-versed in deep learning, I know I'll get a better feel on where certain layers should go to accomplish certain tasks. Another possible venture is to ensure that generated mazes are valid (it's not helpful to give a player a maze that is unsolvable). Proving validity might be computationally expensive but one can use heuristics to eliminate many invalid ones (i.e. goal and/or player is completely surrounded by traps). Once I have the player DQN working, it would be interesting to compare performance (in terms of CPU time) to how a BFS search would perform.

And of course, we will implement an adversary based on the suggestions found in related literature (noted in above section). Another possibility is to change the agent so that it has some perception of the environment without needing to get stuck in a trap to find out where the traps are (e.g. agent "feel" a breeze if neighboring a trap). Furthermore, the complexity of the game can be increased by adding additional types of squares that yield different rewards.