Training Deep Q Networks with Time Hopping for Classic Control Reinforcement Learning

Nalini Singh
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
nmsingh@mit.edu

Andrew Ilyas
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
ailyas@mit.edu

Abstract

Reinforcement learning, and particularly deep Q-learning, despite recent popularity in game-playing and control agents, is still quite slow to train. In this work, we attempt to apply time hopping, a previously proposed method for accelerating traditional Q-learning algorithms to the domain of deep Q-learning. We obtain mixed results, showing a slight and potentially insignificant improvement in performance in episodes, and an increase in human-time taken. Here, we outline the background behind each of the implementation components, our specific implementation of them, and the results obtained. Then, we analyze the implications and limitations of our results and use them to discuss potential issues with the originally proposed time hopping method.

1 Introduction

Reinforcement learning has recently been a relatively oft-used strategy in teaching artificial agents to play games. Q-learning in particular seems to be a very common strategy for learning games, and has matured at an astounding pace in recent years. The introduction of the "deep-Q" network by DeepMind [1], for example, showed a highly effective way of combining deep neural nets with traditional Q-learning. However, the accessibility problem of often-infeasible long training times continues to plague reinforcement learning. In [2], Kormushev et. al describe a technique known as "time hopping" that is claimed to significantly accelerate traditional Q-learning methods. Kormushev et al. also demonstrate the application of time hopping to the "biped crawling robot" control problem.

In this work, we attempt to apply an implementation of the time hopping algorithm described by [2] to a Deep-Q network. In particular, we use a DQN solution [4] to the Acrobot Problem [3], a classic control problem hosted by OpenAI described in detail in Section 3. We begin by reviewing the background work done related to our experiment, including a cursory look at traditional Q-learning, the advancements introduced in the Deep-Q framework, as well as the method proposed to speed up traditional Q-learning through time-hopping. We also review the authors' proposed implementation of the components of time hopping, and how they serve to achieve the goal of speeding up Q-learning. Finally, we apply these techniques to a standard deep-Q network for classical control problems, and report the results.

2 Background and Related Work

2.1 Q Learning

In general, Q-learning is a model-free RL process that attempts to find the best process ("policy") for selecting actions in an arbitrary Markovian Decision Process (known as an MDP). As in any RL algorithm, the external environment is the same; at any point, the process is given an input state $1 \ldots s$ and a set of admissible actions $1 \ldots a$; upon taking an action, the algorithm is rewarded or penalized accordingly through a *reward*, R_i. Now, because of this problem structure, one can imagine a matrix $R \in \mathbb{R}^{s \times a}$ such that R_{ij} represents the reward obtained by taking action j from input state i. Given this conception of a reward matrix, Q-learning models the action-selection problem through its own representation $Q \in \mathbb{R}^{s \times a}$, initially set to all zeros. Then, at each step in the learning algorithm, when we are given an input state s, we can look at the Q matrix, pick an arbitrary action a, and look at the values of the Q matrix that correspond to the resulting state (s_{res}). Then, given a tuned hyper-parameter γ, representing the weight given to future rewards,

$$Q(s, a) = R(s, a) + \gamma \max_{a_n \in A} \left[Q(s_r es, a_n)\right]. \tag{1}$$

When the matrix is sufficiently determined, we can use it to make policy decisions, by simply selecting the action that maximizes the Q-value for the given input state.

2.2 Deep-Q Network (DQN) and Experienced Replay

The paper "Playing Atari with Deep Reinforcement Learning" by Mnih et. al discusses the usage of Q-learning to playing common Atari games such as Pong or Space Invaders. The main idea involves learning the Q-policy by using an artificial neural network (ANN) rather than through the simple update step described in the last section. In particular, a loss function is defined as the expected value of the squared difference between a target Q-value y_i, and the estimated $Q(s, a)$. For a state s and an action a, the target Q-value is determined by the conditional expected value over all possible "next states" s' of $r + \max_{a'} \gamma Q(s', a')$ given a and s. More formally, for a current state s and an action a that would result in a new state s', we define:

$$y_i = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q(s', a') | s, a\right]$$

$$L = \mathbb{E}_{s,a \in \rho(\cdot)} \left[(y_i - Q(s, a))^2\right]$$

Note that the expected value for loss is taken across a distribution ρ instead of uniformly—this is denoted as the "behavioural distribution" across states and actions. Now, in addition this loss function, Mnih et. al also introduce a technique called "experienced replay," where the agent's experiences are stored in a dataset and sampled from during training. Given these two new ideas, a simplified version of the deep-Q network can then be defined. Essentially, it follows the same iterative process that is described in traditional Q-learning, but with a few differences in the actual training in each iteration. Namely, we now just sample a random action with probability ϵ, otherwise selecting the policy-maximizing action. We then execute this action and observe the reward and output, and store everthing in our experience dataset. Then, we sample a mini-batch from the experience dataset and for each element j, set y_j to be r_j if j is the step before termination of the episode, otherwise setting $r_j + \gamma \max_{a'}(Q(s_{j+1}, a))$ as described above. Finally, we calculate the loss across the mini-batch, and use SGD on the gradient of this loss function to minimize it with respect to a parametrization variable θ. [1]

2.3 Time Hopping

Time hopping, proposed by Kormushev et. al, describes a potential method by which one could speed up traditional Q-learning. It relies on the idea that normally, RL algorithms spend too much time exploring unnecessary paths, and given the size of the search space, most algorithms will spend too much time learning about useless high-probability states. The goal behind time hopping is to provide what Kormushev calls "shortcuts" to low-probability states from possibly distant high-probability states, thus making the visitation distribution across states nearly uniform despite a potentially skewed expected distribution. In order to do this, it is proposed that three components are required in addition to a standard Q-learning implementation. The first of these is a trigger which, when activated, causes the "hopping" to happen; the work suggests a method called "Gamma Pruning" as a good method by which to implement this trigger. The second component is the selection component which, when a trigger is activated, must choose *which* state to "hop" to—the work once again suggests an implementation for this, which is called "Lasso". Lastly, an actual hopping mechanism must be implemented, which will allow the environment to hop between states that may not be connected by a single action. In the following three sections, we examine the components presented in the work and how they may serve to speed up Q-learning.

2.3.1 Trigger: Gamma Pruning

The first component as described by [2] is the "trigger" functionality, which the work claims is inspired by α/β-pruning in search. In particular, it is noted that when Q-learning is examining some alternative best-path, after a certain point if even the best possible reward gleaned from the path is worse than what we got from the main path, we should "prune" this alternative path and "hop" to a better state. In order to quantify this trigger, the authors set a "best projection" threshold, given that maximum reward is acheived from every future action:

$$Q_{Best} = \frac{R_{max}}{1-\gamma}. \tag{2}$$

Then, $T_{s,t}$ is defined as the threshold for the best policy at state s to be modified to pass through t, according to the following recursive formula:

$$T_{1,1} = Q_1 T_{1,n} = \frac{T_{1,n-1} - R_{n-1,n}}{\gamma}. \tag{3}$$

2.3.2 Selection: Lasso Selection

Next, a method for selecting a state to hop to is required. The method described by the authors in [2] is a simple one; starting from an initial state, continue following the Q-maximizing actions until a state is repeated thus forming a cycle; this is dubbed "lasso" selection since a state diagram showing these transitions has a lasso-like appearance. Once the "lasso" is formed, Kormuchev et. al. select the state to hop to by conducting a weighted sampling, with each state being weighted inversely with the number of visits that state has received. The lasso part ensures that we select a "relevant state," while the weighting of the random sample ensures that we equalize what would be a skewed distribution of states.

2.3.3 Mechanism: Basic Hopping

The last implementation component mentioned by Kormushev et. al is a simple hopping mechanism; that is, the environment must provide a tool by which it can jump from state to state. In the case of simple visualizeable control problems, this is near-trivial, since all that is involved is updating the underlying problem representation with the new state, and then updating the visuals based on the new problem representation.

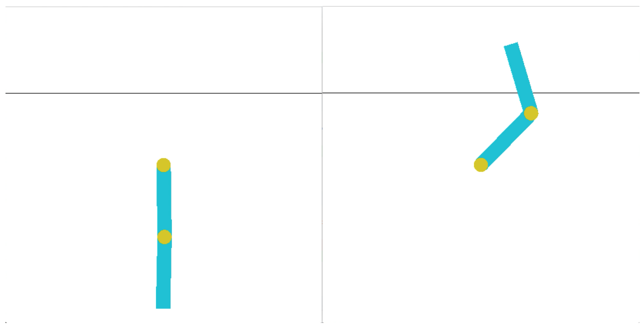
3 Methods

We augmented the training procedure for a Deep Q Network used to solve classic control problems to use the time hopping strategy described by Kormushev et al. Then, we compared the training times required by a plain Deep Q Network with those required by a Deep Q Network trained with time hopping. Each component of this project is described in more detail in subsections 3.1, 3.2, 3.3, and 3.4.

3.1 Acrobot Test Environment

We used the open source reinforcement learning environments available through the OpenAI Gym to analyze the effect of time-hopping on the training time of a Deep Q Network. The OpenAI Gym offers two classic control environments. In the Cartpole environment, the reinforcement learning algorithm being tested learns to apply a force of either +1 or -1 is to a cart in order to keep an attached, un-actuated pole upright. In the Acrobot environment, the algorithm learns to apply a torque of either -1, 0, or 1 to a joint connecting two links, which are initially hanging vertically; the ultimate goal is for the tip of the bottom link to swing above above a height of one link above the bar.

We initially planned to implement the time-hopping strategy for the Cartpole control problem. However, after running a pre-existing Deep Q Network without time-hopping to solve the cartpole problem, we were able to achieve success within only a few minutes and a few hundred iterations. Since this network did not spend significant time switching between different exploration paths, it became clear that it would be difficult to observe speed-ups from time-hopping in this context, even though the time-hopping strategy might be beneficial in more complex contexts. Thus, for this project, we decided to test the time hopping strategy on the Acrobot control program. Specifically, we used Acrobot-v1 as provided by OpenAI. A visual depiction of the problem to be solved is shown in Fig. 1. The state of the environment is characterized by a vector with four elements, two representing the angular positions of each link and two representing the corresponding angular velocities.



(a) Initial Acrobot state. (b) Goal Acrobot state.

Figure 1: Two possible states within the Acrobot control environment. The links begin in the position shown in Fig. 1a. Reinforcement learning is used to guide the links to a position like the one in Fig. 1b, where the tip of the second link has swung above the reference bar.

3.2 Deep Q Network

We used Li Bin's open source Classic Control DQN repository (`https://github.com/lbbc1117/ClassicControlDQN`) as a baseline Deep Q Network against which to test the time hopping strategy. The Classic Control DQN repository interfaces directly with the OpenAI gym classic control environments and includes a python script that can be run directly to train a QAgent to solve the Acrobot control problem.

3.3 Time Hopping Implementation Strategy

Implementing the time hopping strategy proposed by Kormushev et al. required three major components: (1) modification of the environment simulator to allow time hopping, (2) implementation of the 'hopping trigger' to determine when hopping should be performed, and (3) implementation of the 'lasso selection' procedure by which a new state to hop to is selected. Details of each of these implementation steps are provided in subsections 3.3.1, 3.3.2,and 3.3.3.

3.3.1 Modification of the Simulator

We modified the OpenAI gym implementation to allow hopping to arbitrary states within any environment. We edited the OpenAI gym infrastructure to include a straightforward method allowing an environment's state to be set to a state specified by the user within their high-level training procedure.

3.3.2 Hopping Trigger

Next, we modified the Deep Q training regime in order to determine when hopping was required. In particular, we implemented the gamma pruning regime described by Kormushev et al. During each transition of every exploration, we updated the Q-value threshold required for hopping between states based on the reward achieved as described in Equation 3.

The initial OpenAI implementation of the Acrobot environment assigns binary rewards based on the state of the two links. In particular, the QAgent receives a reward of -1 if the tip of the second link is below the horizontal bar and a reward of 0 if the tip of the second link is above the horizontal bar. However, this reward scheme is problematic for performing the update described in Equation 3. Since the environment is initialized with the links below the bar and remains that way for many steps, the repeated receipt of rewards of -1 drives up the threshold value at a constant rate, causing gamma pruning to occur at every step. To prevent this, we implemented a continuous reward scheme in which the QAgent received a reward of 0 if the tip of the second link was at or above the horizontal line and a reward of -1 if both links pointed vertically downward. For every height between the bottommost point and the horizontal line, the reward achieved was linearly interpolated between -1 and 0.

At each of these transitions, we also checked whether this threshold, the minimum Q-value for a state required to make the best policy pass through that state, surpassed the best-possible reward attainable by passing through that state. As described in Equation 2, this occurs when the threshold T exceeds the quantity $R_{max}/(1-\gamma)$. Under our reward scheme, $R_{max} = 0$. Thus, whenever $T > 0$, the hopping trigger is activated, the current line of exploration is terminated, and the QAgent begins performing lasso selection to determine which state to hop to.

3.3.3 Lasso Selection

Finally, we implemented a procedure for selecting a new state to hop to when the hopping trigger suggested a hop was necessary. In particular, we implemented the lasso selection procedure as described in [2].

We store a hash list of the states previously visited during the current instance of lasso-selection. This hash list is initialized with the initial state. Then, the action that yielded the maximum Q-value for the initial state is taken. The new state is added to the list of visited states, and the action yielding the maximum Q-value from this new state is taken. This process is repeated until the procedure reaches a state that was already included in the hash list. At this point, the target state is randomly selected from the lasso, the states of the QAgent and the environment are set to that state, and the higher-level training algorithm resumes.

In order for the lasso search to terminate within a reasonable time, we needed to limit the search space of the algorithm. To do this, we discretized the search space such that each link's position and velocity were only described to one decimal point. This change significantly decreased the total number of possible states and thus decreased the expected amount of time before the lasso selection procedure revisited a state it had previously visited.

3.4 Evaluation Strategy

In order to evaluate the effectiveness of the training strategy described in this paper, we first trained a Deep Q Network without time hopping and calculated both the number of episodes required for learning to finish and the number of seconds required for each episode. We defined learning as being finished when the QAgent achieved an average score higher than -100 over the course of 100 consecutive episodes; the resulting model was then tested to ensure that it produced scores above -100 over the course of 1000 steps. The score computed during each episode was defined to be the sum of the rewards achieved over 1,000 consecutive timesteps, where the rewards were continuous values ranging from -1 to 0. Though the time required to train the network prevented us from doing an exhaustive optimization of the hyperparameters of the model, we informally experimented with different values of γ (0.5,0.7,and 0.9) and ϵ decay rates (0.2,0.5,0.8) over 1,000 iterations to determine the parameters that came closest to the desired Acrobot behavior. Based off of those experiments, we trained the DQN over 5,000 iterations using $\gamma = 0.5$ and an ϵ decay rate of 0.5 for our baseline DQN analysis.

In order to perform a strict evaluation of the impact of the time-hopping strategy on training time, we re-trained the Deep Q network using the time hopping strategy with the same γ and ϵ decay rate. We again recorded both the number of episodes and the number of seconds required for each episode, and then we compared these results to those from the Deep Q Network trained without hopping.

We then performed additional informal experimentation with other values of γ and ϵ to determine whether different values of these parameters affected the success of the time hopping strategy. This training method is referred to as Adjusted Time Hopping throughout the remainder of this paper.

4 Results

The scores resulting from training the baseline Deep Q Network along with the results of the time hopping experiments are shown in Fig. 2. As the figure demonstrates, the baseline Deep Q Network finished learning after roughly 3,400 episodes. In contrast, the Deep Q Network trained using the same hyperparameters as the baseline network was unable to finish learning over the course of the 5,000 iterations for which we ran the simulation.

While training using the time hopping procedure, we observed that the value ϵ was decreasing too quickly for time hopping to be valuable; ϵ became very small very quickly, and as a result, there were very few periods of exploration during which the training procedure was able to actually perform the hopping. Increasing the ϵ decay rate from 0.5 to 0.8 allowed more hopping to occur, and the results of that training procedure are also shown in Fig. 2. The Deep Q Network trained with the adjusted hyperparameters and with time hopping finished learning after roughly 3,000 episodes, about 300 episodes sooner than the baseline training method.

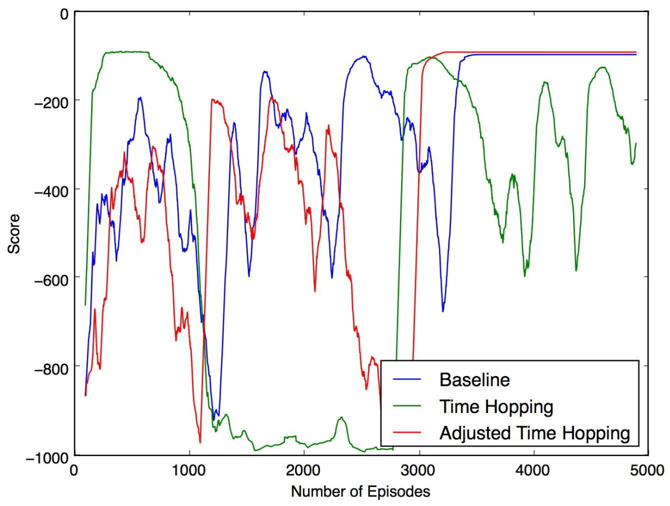


Figure 2: Training scores achieved over 5,000 iterations. The model trained using time hopping with adjusted parameter values achieved success roughly 200 episodes before the baseline DQN. The model trained using time hopping with the same parameter values as the original DQN did not achieve success with 5,000 iterations. The scores reported here are smoothed with a median filter of length 100 to improve readability of the plot.

In addition to evaluating training time in terms of episodes elapsed, we also computed the distributions of the time spent per training episode for each of the three training procedures. The results of those calculations are shown in Fig. 3. Fig. 3 shows that every step within the baseline training procedure took under 10 seconds to complete. In contrast, a sizable fraction of the episodes involved in both training procedures incorporating time hopping took over 10 seconds. The training process for the Deep Q Network trained with the same parameters as the baseline network and with time hopping included an increase in the proportion of the episodes that lasted between 10 and 15 seconds, and the training process for the Deep Q network trained with adjusted parameters and time hopping included an additional increase in the proportion of episodes lasting between 15 and 20 seconds.

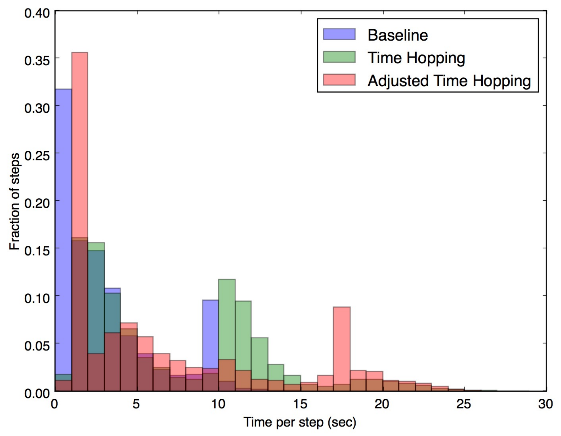


Figure 3: Distributions of times spent per step. Nearly every step spent training the baseline model took under 10 seconds, while both models trained with time hopping spent longer on a significant fraction of steps.

We also calculated the total amount of time spent training each of the three methods analyzed here. These times are reported in Table 1.

Table 1: Training times for the three training methods analyzed in this paper.

	Training Time (hrs)
Baseline DQN	4.32
Time Hopping	9.98 (Did not finish learning)
Adjusted Time Hopping	9.50

5 Discussion

5.1 Analysis of Results

The results presented in Fig. 2 do not present a convincing case for the use of time hopping in training DQNs to solve control reinforcement learning problems. Fig. 2 does show a few sharper jumps in the achieved score for both time hopping procedures compared to the baseline procedure, indicating that the random exploration proposed in [2] did occur. Regardless, the Q network trained using the same hyperparameters as the baseline case ultimately performed worse than the baseline case. Further, it is not clear that the difference of roughly 300 episodes in training time we observed between the baseline training procedure and the adjusted time hopping procedures is significant. That observed speed up of roughly 10% is very small compared to the 7x speedup reported by Kormushev et al. One potential reason for the difference in relative performance of time hopping between this work and that observed in [2] is that, while hopping introduces sudden deviations from the exploratory training paths already implemented by the Q-learning algorithm, these deviations are not guaranteed to be better than the original exploratory paths; they are simply guaranteed not to be worse. In this case, it is possible that the hopping procedure resulted in explorations that were roughly equally as uninformative (or even less informative) compared to the original explorations.

The results presented in Fig. 3 provide a different view on the training scenarios tested in this paper. The higher frequency of longer training episodes in the two hopping distributions suggests that hopping can cause individual episodes to take longer to complete. This result is likely due to the fact that the lasso selection procedure can get 'stuck' visiting many states before returning to a previously visited one if the state space representation is large. Despite discretizing our state space such that all link positions and velocities were only reported to one decimal point, this effect was clearly present in these results. As shown in Table 1, both training regimes that used time hopping took significantly longer to complete in terms of human time compared to the baseline DQN, regardless of the number of episodes elapsed.

5.2 Limitations of Results

The results presented in this paper suffer from several limitations that remain to be explored.

First, due to high training times, we were unable to replicate the experiments described in this paper several times. As a result, we do not know the confidence bounds surrounding the results presented in this paper. Further, since Deep Q networks incorporate randomness by nature, this makes it difficult to discern how much of the difference in performance between different trials was due to randomness and how much was due to concrete differences due to time hopping.

Further, due to high training times, we were unable to exhaustively optimize the hyperparameters of the models described in this paper. Though we did do preliminary experimentation to roughly tune the parameters, this experimentation was based on results over the course of a relatively low number of episodes (roughly 1,000), and might not have reflected optimal trends over a longer training period. Since the models we trained were not perfectly optimized, it is possible that the baseline and time hopping results in this paper are not representative of the results from models trained to their fullest capacity.

Third, we made several decisions in transforming the Acrobot training environment to be suitable for experimenting with the time hopping procedure. In particular, we imposed our custom reward scheme with rewards ranging continuously from -1 to 0 instead of being a binary choice between those values, and we imposed a discretization of the state space to be reported to only one decimal point. It is unclear how these decisions might have affected the ultimate scores, training times, and relative performances of the models.

5.3 Critique of Original Time-Hopping Paper

In theory, time-hopping seems to be an effective way to equalize a skewed state distribution, and seems to have the potential to reduce episodes spent on a reinforcement learning problem. However, in implementing the algorithm, many limitations appear that render time hopping much less useful than one would imagine.

The first of these is the fact that in practical applications, we tend to care more about “speed” in terms of human seconds more than in terms of episodes trained, however even the source material in [2] fails to provide a time-based comparison. In this experiment, it was empirically observed that episodes when using time-hopping were significantly slower than without time-hopping, something that is perhaps to be expected, given that the lasso selection policy must be run several times in each epoch, something that would not have to be done in a traditional Q-learning context.

The other main problem observed with the source paper is its incompatibility with large state spaces, which is unfortunately a feature that seems to be found in many RL problems, even one as simple as Acrobot, where millions of states are possible. Thus, it would take an immense number of iterations of following best-Q actions in order to find a repeated state. This has extremely high cost in terms of speed (since the algorithm is completely blocked until the lasso selection finds a cycle), and in terms of memory, since the lasso selection function must keep all visited states in memory in order to be able to detect cycles. This effect can be slightly mitigated by discretization as was done in our experiment, but even with a quite generic binning discretization of the data, lasso selection caused the algorithm to freeze and use up a significant amount of memory compared to standard Q-learning.

Human-time speed and large search space were the two main high-level problems encountered with the idea described in [2] during implementation, but there were far more technical challenges involved with actually applying the idea into a development-scale environment; these challenges are outlined in the next section.

6 Technical Challenges

In this section, we outline some of the lower-level implementation challenges involved with applying time-hopping to the DQN algorithm for classic control, such as the need for disambiguation,

6.1 Ambiguous Implementation Details

As with any project involving the implementation of abstract ideas into a concrete model, many of the technical challenges encountered were as a result of ambiguity in implementation. In this case, there were actually two sources of ambiguity during the implementation of the application. The first of these originates from within the time hopping paper itself, which justifiably omits some minutia of implementation; for example, the exact procedure and constraints for sampling a state from the "lasso" is not really specified in the work. Unfortunately, the algorithm does not appear to have been publicly implemented before, and therefore these ambiguities had to be disambiguated arbitrarily by the authors of this work.

There was also a unique second source of ambiguity involved with implementing an algorithm devised for use with traditional Q-learning into a deep Q-learning framework, since though the underlying problem representation is the same, the training process itself is quite different and is much more parameterized than a traditional Q-learning model (which takes only γ). Thus, strategies had to be devised for how to deal with extra hyperparameters (such as ϵ or its decay), and how to deal with a slightly different training structure than in traditional Q-learning.

6.2 Long Training Times

As is to be expected with (deep) reinforcement learning problems, training times even for a model as simple as acrobot were quite long compared to other types of learning, which made it more difficult to iterate on the implementation and collect results efficiently. Fortunately, we had access to several different machines, which let us somewhat parallelize the experimentation and result-collecting process.

7 Conclusion

In this work, we explored the possibility of speeding up a historically slow process of Deep Q learning, specifically in the context of the OpenAI-hosted Acrobot 2D control problem. A method claimed to reduce training time for traditional Q-learning methods was implemented in the context of the DQN algorithm, and experiments were run tracking the score achieved by the agent using DQN, as well as both the adjusted and unadjusted variants of the time hopping algorithm described in [2]. Overall, we found that though the time hopping seemed to slightly improve speed of convergence in terms of number of episodes, this may or may not have been significant, and flaws in the state selection methods described in the underlying time hopping model causes each episode to run significantly slower and consume exponentially more memory than a standard DQN algorithm would use.

Distribution of Work

The initial plan for distribution of work was for the brainstorming and disambiguation of implementation details to be done together, and then for Andrew to implement Gamma Pruning and Lasso Selection while Nalini edited the OpenAI gym environment to be able to perform basic hopping from state to state, as this was anticipated to be the most difficult of the three tasks. However, due to the clean, user-friendly, and open-source implementation of the OpenAI gym, implementing the basic hopping functionality was found to be just a one-line function. Meanwhile, Gamma Pruning and Lasso Selection proved very difficult to implement in a DQN environment, and so the work was redivided, with Andrew implementing Gamma Pruning and Nalini implementing Lasso Selection. Once the implementation was done, the experiments were designed, run, and reported on by Nalini (on her personal machine).

In terms of writing, Nalini wrote Sections 3, 4, and 5.1-2, and Andrew wrote the Abstract as well as Sections 1, 2, 5.3, 6, and 7.

Acknowledgments

We would like to thank Prof. Tomas Lozano-Perez for his guidance in selecting our subject, and Anish Athalye for lending us computing resources used to train our models.

References

[1] Mnih, Volodymyr, et al. ”Playing atari with deep reinforcement learning.” arXiv preprint arXiv:1312.5602 (2013).

[2] Kormushev, Petar, et al. ”Time hopping technique for faster reinforcement learning in simulations.” CYBERNETICS AND INFORMATION TECHNOLOGIES 11.3 (2011).

[3] Brockman, Greg, et al. ”OpenAI Gym.” arXiv preprint arXiv:1606.01540 (2016).

[4] Bin, Li. Classic Control DQN Repository. `https://github.com/lbbc1117/ClassicControlDQN`.