Deep Reinforcement Learning for Simulated Autonomous Vehicle Control

April Yu, Raphael Palefsky-Smith, Rishi Bedi
Stanford University
{aprilyu, rpalefsk, rbedi} @ stanford.edu

Abstract

We investigate the use of Deep Q-Learning to control a simulated car via reinforcement learning. We start by implementing the approach of [5] ourselves, and then experimenting with various possible alterations to improve performance on our selected task. In particular, we experiment with various reward functions to induce specific driving behavior, double Q-learning, gradient update rules, and other hyperparameters.

We find we are successfully able to train an agent to control the simulated car in JavaScript Racer [3] in some respects. Our agent successfully learned the turning operation, progressively gaining the ability to navigate larger sections of the simulated raceway without crashing. In obstacle avoidance, however, our agent faced challenges which we suspect are due to insufficient training time.

1. Introduction

Currently, self-driving cars employ a great deal of expensive and complex hardware to achieve autonomous motion. We wanted to explore the possibility of utilizing a cheap everyday camera to enable a car to drive itself. Our main question was whether we could learn simple driving policies from video alone. Current autonomous driving implementations have shied away from the computer vision techniques because of a lack of robustness. The inaccuracies with vision-based autonomous driving systems lie mostly in the difficulty of compressing the input image into a compact but representative feature vector. There are currently two approaches to this problem: "mediated perception approaches" parse an entire scene (input image) to make a driving decision and "behavior reflex approaches" utilize a regressor to directly map an input image to a driving action. [2]. Neither of these approaches has been resoundingly successful. Because of this, we ultimately turned to autonomous driving through end-to-end Deep Q-Learning.

However, the ability to test these techniques and the various related experiments with an actual car on real-video data was out of the question, given the reinforcement-learning nature of the paradigm. Instead, we turned to JavaScript Racer (a very simple browser-based JavaScript racing game), which allowed us to easily experiment with various modifications to Deep Q-Learning, hyperparameters and reward functions.

2. Related Work

Q-Learning (further explained in the Methods section) was introduced by Chris Watkins in his Ph.D. thesis. [10] Inspired by animal psychology, Watkins sought to develop a method which allows for the efficient learning of an optimal strategy to accomplish arbitrary tasks which can be formulated as Markov Decision Processes. While supervised learning can often learn action policies as well, "sequential prediction" problems are often better served by reinforcement learning approaches, where the model has some level of interaction during the learning process. [7]

In 2005, Riedmiller introduced the idea of using neural network approximators for the Q function in Q-learning. [6] Mnih et al. introduced the idea of image-based Deep Q-Learning in 2015, when the group at DeepMind successfully used a convolutional neural network (DQN) to learn a Q function which successfully plays various Atari 2600 games at or significantly above professional human player ability. The only inputs to their DQN algorithm were images of the game state and reward function values. [5] Most impressively, this paper utilizes a single learning paradigm to successfully learn a wide variety of games - the generalizability of the approach (while obviously not a single set of learned parameters) is powerful, and is what inspired us to attempt to apply their model to learning a policy for JavaScript Racer.

Since that work, DeepMind and others have published numerous extensions to the DQN paradigm. We will summarize some of those extensions here, some of which we later choose to implement variations of to seek to improve our agent's performance.

DQN learning approaches have been successfully leveraged for continuous control in addition to discrete control tasks. While we ultimately modeled our simulated car driving problem as a discrete learning task, the task of simulated

car control has been well-studied as a continuous control problem as well. [4]

Some popular DQN modifications we explore include prioritized experience replay and double DQN, the details of which are explained in Methods [9].

State-of-the-art autonomous vehicle control algorithms are largely orthogonal to DQN approaches, and since the very simplified game we ended up playing bears few actual similarities to real-world autonomous driving, the substantial body of literature that exists in that field was not especially relevant to our work here. DQN-based approaches to simple video games were much more in the vein of the work done in this paper, and thus form the core body of work on which our experiments are based.

3. Methods

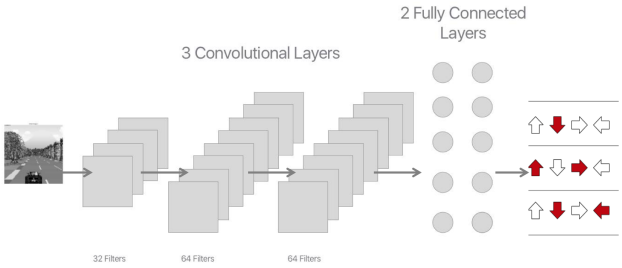
We began our project by re-implementing the Deep Q-Learning algorithm ourselves [5] as presented by the team at Google DeepMind, using TensorFlow [1]. Our implementation can be found on GitHub: `https://github.com/RaphiePS/cs231n-project`. This algorithm extends the general Q-Learning reinforcement learning algorithm to adapt to an infinitely larger state space. In vanilla Q-Learning, the algorithm learns an action-value function that ultimately gives the expected utility of taking a given action in a given state. The implementation of vanilla Q-Learning keeps track of these (state - action - new state) transitions and the respective reward in a table and updates these values in the table as it continues to train. With each training point, the Q function estimates the expected reward for taking a particular action from a particular state to arrive at a new state and picks the action that maximizes this reward. With the knowledge of the actual observed reward r and the next state s', we are able to calculate the relative error for that particular state-action-new state transition (using γ as a discount for future rewards):

$$error = (r + \gamma max_{a'} Q(s', a')) - Q(S, a) \qquad (1)$$

where the minuend is the actual reward and the subtrahend is the reward predicted by our action-value function. With this calculated error, we are able to update the transition table by adding it to the existing Q value for the particular state-action-new state. These incremental updates gradually transform the Q-function from a table of random noise to an effective game-playing agent. However, when the input to the algorithm is an image, the state space becomes prohibitively large and this transition matrix implementation is no longer practical. Instead, a convolutional neural network (CNN) is used as the Q-function approximator and the error is backpropagated to update the network with each minibatch of training examples, such that the parameters of the CNN learn a good non-linear approximation of the Q-function even for states it has not explicitly seen before.

We implemented the same CNN architecture (Figure 1) as proposed by Mnih et al [5], with 3 convolutional layers and 2 fully connected layers.

Figure 1. Convolutional Neural Network Architecture



The naive implementation of Deep Q-Learning is the simple swapping of a table-based Q function for a CNN. However, this setup is unstable and can lead to a sort of "overfitting," where updating the neural network from the most recent experiences hurts the agent's performance in the immediate future. Thus, Mnih et al [5] propose two modifications, both of which are designed to prevent the algorithm from focusing on the most recent frames and help it smooth over irregularities. These can be thought of as a sort of "temporal regularization," and they help the agent converge much faster.

The first of these methods is called experience replay, and it draws inspiration from learning mechanisms in actual neuroscience. The key idea is to update the network using all of its past experiences, not just recent frames. To perform experience replay, we store experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t into the experience replay buffer $B_t = \{e_1, ..., e_t\}$. During training, we apply Q-learning updates on minibatches of experience $(s, a, r, s') \sim U(B)$. Note that these minibatches are drawn uniformly from the buffer. Mnih et al [5] acknowledges that this is a shortcoming - every frame is equally likely to be sampled to update the network, yet it is obvious that some frames (e.g. entering a turn) are far more "influential" than others (say, driving down a straightaway). There has since been work aimed at more carefully sampling the replay buffer, which is called prioritized experience replay.

The second "regularization" technique applied by Mnih et al [5] is the use of not one, but two Convolutional Neural Networks to learn the Q function. In addition to the usual Q-network, they propose a second "target" Q-network, designated $\hat{Q}$. This net is used solely to generate the target values for the Q update - the regular Q-network is still used to choose actions at each step. Thus, the error equation becomes

$$error = (r + \gamma max_{a'} \hat{Q}(s', a')) - Q(s, a) \qquad (2)$$

While error is still backpropagated to Q, $\hat{Q}$ is never updated via backpropagation. Rather, every C frames (10,000 in our experiments), the parameters from Q are simply copied over to $\hat{Q}$. This temporal delay similarly prevents temporal "overfitting".

Upon feedback we received at the Poster Session, we implemented the Double Q-Learning variant proposed by van Hasselt et al. [9] Double DQN is meant to alleviate the problem of DQNs overestimating the value of a given action in some circumstances. Double DQN attempts to correct for this by separating the selection and evaluation of the max function employed in the calculation of y_j. More precisely, Double DQN replaces the original target y_j evaluation function (in the non-trivial case where step j is non-terminal, otherwise $y_j = r_j$ in both DQN and Double DQN algorithms), which is

$$y_j = r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) \tag{3}$$

with the following:

$$y_j = r_j + \gamma \hat{Q}(\phi_{j+1}, \arg\max_a Q(\phi_{j+1}, a; \theta), \theta^-) \tag{4}$$

Also upon feedback we received, we replaced our unclipped loss function with a simplified variant of Huber loss, instead of the original unclipped ℓ_2 loss function, with the goal of improving the DQN's stability.

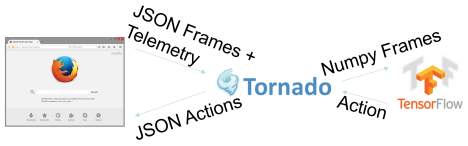
$$loss = min(|\Delta Y|, |\Delta Y|^2) \tag{5}$$

As mentioned previously, the original DQN implementation in [5] sampled uniformly from transition memory. The authors noted this as a potential area for improvement, which Schaul et al. attempted to correct. [8] This alteration extends the experience replay model by not simply uniformly sampling transitions from memory, but instead, weighting individual transitions in memory with their "TD-error", an attempt to quantify the "unexpectedness" of a given transition, with the ultimate goal of allowing the DQN to replay "important" transitions more frequently and therefore learn more efficiently. The formula for TD-error (also using the Double Q-learning equation) is given as follows - essentially, it computes the difference between y_j as computed by the Double DQN update and the Q value as computed by the other (non-target) network.

$$\delta_j = r_j + \gamma \hat{Q}(\phi_{j+1}, \arg\max_a Q(\phi_{j+1}, a; \theta); \theta^-) - Q(\phi_j, a; \theta) \tag{6}$$

We implemented the binary-heap rank-based prioritization approach described in appendix B2 of [8], where each transition is inserted into a binary heap. This heap is then used as an approximation for a sorted array, and the array is divided up into 32 "pieces" (the size of the minibatch) different-sized pieces. On every update, then, one element of the minibatch is selected from each of the "pieces," such that elements with larger TD-error are more highly valued and thus in a smaller "piece," meaning they'll get sampled more frequently unless they are displaced to a lower-priority "piece" by new higher-TD-error transitions. [8] determined that the imprecise ordering conferred by abusing the binary heap is not a roadblock to successful priority sampling, and that the performance overhead of maintaining a perfectly sorted array at all times is not worth it.

Figure 2. Training architecture Browser - Webserver - TensorFlow



4. Dataset and Features

We did not use a pre-existing dataset. Rather, our images came from real-time play of the game we chose, JavaScript Racer. [3] We chose this particular game for several reasons: it is open-source, its code is well-documented and easily-modifiable, its actions are discrete (binary key-presses for steering instead of a float for steering wheel position like other simulators), and it had all the simulated challenges we were looking for - road boundaries, lanes, and traffic on the road. Since JavaScript Racer is a browser game, our Python-based agent could not collect frames directly. Instead, we modified the source of JavaScript Racer to send JSON-encoded frames and telemetry data to our Python web server, which would run the data through our TensorFlow network and return actions back to the browser (Figure 2).

Following the example of the DeepMind paper, our browser code implemented Action Repeat. With this model, each action chosen by the agent is repeated for four frames, which we called "sub-frames." Since the four sub-frames are so similar, only the last sub-frame is sent along with the reward accumulated over all the sub-frames. The simulation steps forward 50 milliseconds each sub-frame, so with an action repeat of four, the agent observes a frame every 200 milliseconds of simulation time.

The game outputs frames that are 480 pixels wide by 360 pixels high (Figure 3). As part of our preprocessing, the browser-side code downsized each frame to 84 by 84 pixels, as recommended by the DeepMind paper. In early trials we extracted the luminance (Figure 4) and used just one channel per image, but in later ones we used three-channel RGB images. Since we didn't have the training examples in advance, we didn't perform any mean image subtraction or

similar normalization.

Figure 3. Full-color, full-sized frame.

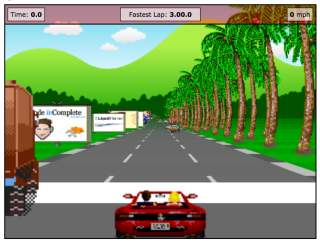


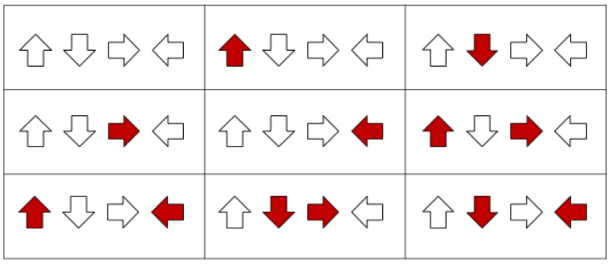
Figure 4. After resizing and grayscaling.

In addition to raw frames, the browser also sends over telemetry data, namely the car's speed, its position on the road, and whether a collision had occurred. The agent never directly uses this data as input - it learned to make decisions based solely on the pixels in each frame. But JavaScript Racer doesn't have a game score, so this telemetry data was used as input for our reward function, as described in the Experiments section below.

Telemetry Data Field	Example Value
Speed	35
Max Speed Attainable	120
X Position	-0.2
Collision Occurred	False
Progress around track	23%

After a forward pass through the Q-network, our algorithm returned an action to send to the browser. This action consisted of four booleans, whether each key - left, right, faster, slower - would be pressed or not. Once we removed illegal actions (for instance, pressing left and right at the same time), we were left with 9 discrete actions, as visualized in Figure 5. After experimentation, we removed many of these actions, leaving us with only three: faster, faster-plus-left, and faster-plus-right. As expected, this led to a much higher average speed, as the car was unable to decelerate.

Figure 5. Full action space



As to the size of our dataset, our experiments utilized between 200,000 and 600,000 frames. Since the game plays the first 50,000 frames completely randomly and uses these frames as fodder for the experience replay buffer, we were able to reuse these frames across trials. Otherwise, each trial generated novel frames for its own use.

5. Experiments

Our initial set of experiments used grayscale images with an agent history length of 4 (i.e., four consecutive frames fed as input to the DQN). Upon discovering a not-particularly-clear saliency map and less-than-satisfactory performance characterized by intermittent stopping of the agent along the racetrack, we suspected our network lacked the ability to consistently differentiate important features on the road. To help ameliorate this, we switched to start using three-channel RGB images. To compensate for the increased computational cost of this larger input (most significantly, network traffic time), we reduced agent history length to 1, so only one frame at a time was being evaluated by the DQN. This was also at Andrej's suggestion, who suggested debugging our implementation would be easier with the net viewing single frames at a time.

We utilized multiple evaluation metrics to qualitatively and quantitatively assess the relative success of a given experiment. One consistent challenge was striking a balance between confirming experiment reproducibility and trying new experiments, given limited time and computational resources. Some of our experiments may have resulted in outlying results that may have been overturned given enough repeats. For the most part, however, we erred on the side of more, diverse experiments instead of confirming initial results. In this sense, some of our "quantitative" results have a more anecdotal flavor. On a related note, we chose to run a greater number of shorter-term experiments instead of training a single model for a very extended duration. Most of our experiments terminated after 200,000 frames, with some continuing to run until 600,000 frames. For context, Mnih et al. trained on 50 million frames for each Atari game.

For quantitative evaluation of our agent's performance,

we analyzed the average Q-value for a minibatch, the Huber loss associated with each gradient update, and the rolling average reward over some number of frames (say, 1000). These continuous metrics, while helpful for debugging and selecting hyperparameters, are often less enlightening than a simple video of the agent controlling the car, the most salient qualitative summary of our best model's driving performance. While the increase in average reward may appear relatively smooth, that belies the discrete jump learning which often happens (as, for example, the car learns how to round a turn it previously consistently crashed on).

We validate the generalizability of our learned model by taking some fixed number of random actions before turning the policy over to the model. This means that the world the DQN agent sees is stochastic and the frames are unlikely to be identical to those it saw during training. In almost all cases, the agent is able to recover from whatever state it finds itself in and complete the number of turns it was able to when started deterministically.

We had greater success using a track with no cars on it (only turns and fixed obstacles), than a car with cars on it. While the agent learned to navigate turns successfully in both, it exhibited only very sporadic ability to avoid cars, and those successes may have been the result of chance rather than true obstacle avoidance. Given the model's ability to learn turning successfully, we ascribe this inability to insufficient training time.

We also had much quicker success by limiting the allowable actions. By eliminating slowing-down (i.e., only allowing forward, forward+right, forward+left), the agent was much more quickly able to learn a sensible policy, as the initial random exploration of the action space contained no frames where the car simply wasn't moving. When the car was allowed to choose to slow down / not move at all, it was still able to make turns, but after an equivalent number of training frames (300k), had mastered only one turn while its action-limited counterpart was consistently rounding four or five.

Initial bugs in our DQN implementation led us to experiment with alternative gradient update rules as a possible solution. We tried Adagrad, RMSProp, and Adam, to find that our original choice (the same as in the Mnih et al. DeepMind paper) of RMSProp with momentum = 0.95 worked best.

We tested multiple reward functions intended to induce different driving behavior. The variables we considered include: staying in a single driving lane (INLANE), progress around the track (PROG), speed (S) as a fraction of max speed, and going off-road (O) or colliding with other vehicles / stationary objects on the course (C). Formally, the two very different reward functions we tested were as follows:

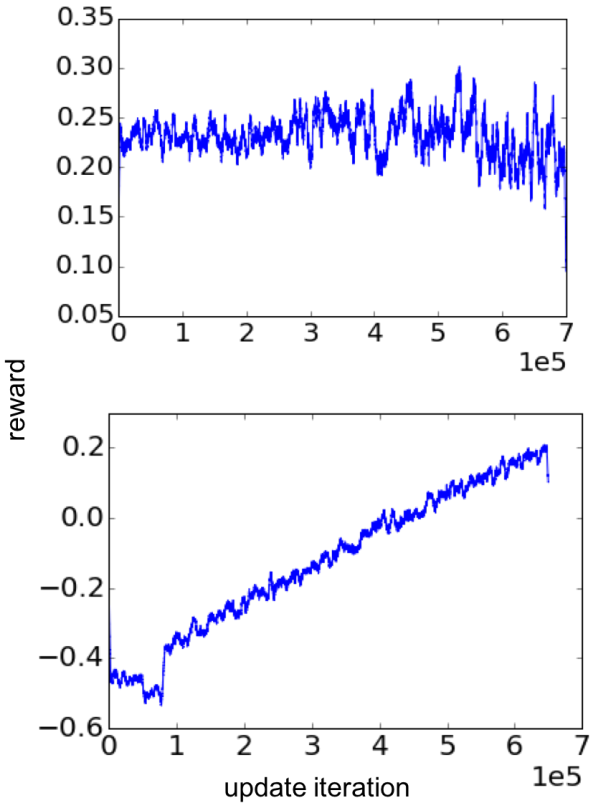
$$S = \frac{\text{current speed}}{\text{maximum attainable speed}} \tag{7}$$

$$R_1 = \begin{cases} -5 & C \text{ or } O \\ -10 & S = 0 \\ \min(10, 0.2 + 5S) & \text{otherwise} \end{cases} \tag{8}$$

$$R_2 = \begin{cases} -1 & C \\ -0.8 & O \\ -1 & S = 0 \\ S & \text{INLANE} \\ 0.8S & \text{not INLANE} \end{cases} \tag{9}$$

At first glance, both reward functions would seem similar - they both penalize collisions and standing still, and reward acceleration proportional to the car's speed. However, we found that the choice of reward function dramatically affected our learning performance. When we plotted the average reward per episode vs the update iteration, we found that the reward function R_2 performed significantly better, as seen in Figure 6 with R_1 above R_2. In fact, using R_1 our agent didn't seem to learn anything at all.

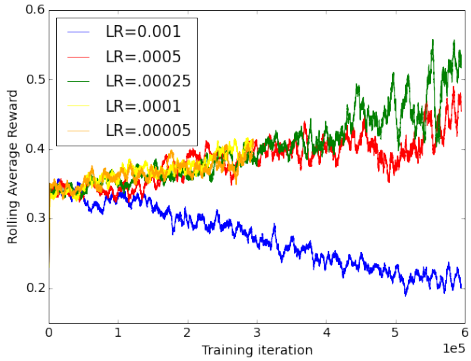
Figure 6. Average episode reward, with R_1 on top and R_2 below. These experiments were conducted with all nine actions allowable (including no-ops and slowing down).



In addition to various reward functions, we tested different learning rates. Our first model which learned unambiguously and convincingly was an RMSProp implementation

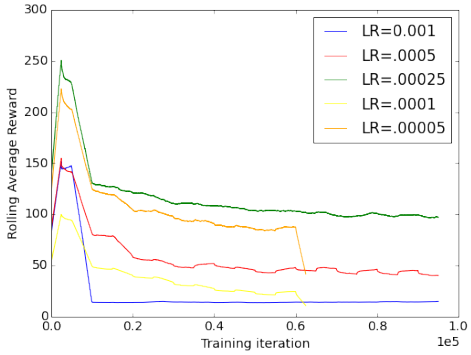
with a reward function optimized for staying in the lane on a track with no other cars on it. Video of one sample run of this agent can be found here: `https://www.youtube.com/watch?v=zly8kuaNvsk`. We then used this as the baseline against which to compare alternative models. The performance of various learning rates is visualized in Figure 7. Upon selecting the reward function and gradient update rule, we held these as fixed for the remainder of our experiments.

Figure 7. Reward function over varying learning rates.



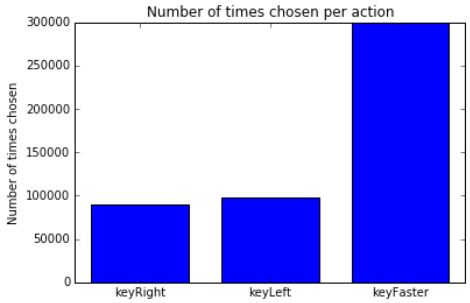
It was immediately clear that a learning rate of 0.001 was too steep, but differentiating between the remaining learning rates was less obvious, so we turned to a visualization of the loss function, which made it clearer that .0001 was the preferred option. This choice was validated by viewing the car driving - the LR=.0001 agent made it around more turns with greater consistency than any of the other agents. Varying the learning rate was the next most significant determinant of qualitative success after reward functions and update rules (Figure 8).

Figure 8. Loss function over varying learning rates.



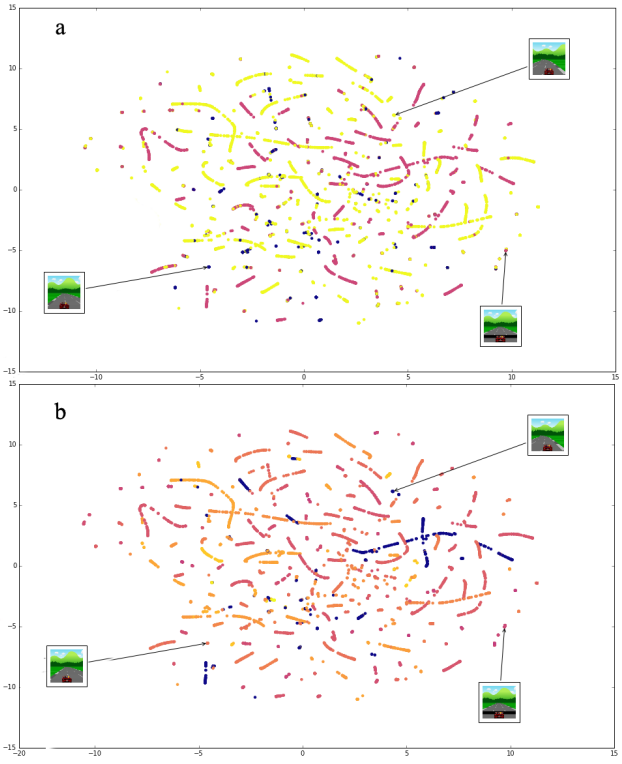
We examined the percentage of actions selected as a sanity check of the policy the agent learned (Figure 9). Reassuringly, the selection of left to right turns was almost identical, and the car consistently accelerated.

Figure 9. Actions selected by best model so far



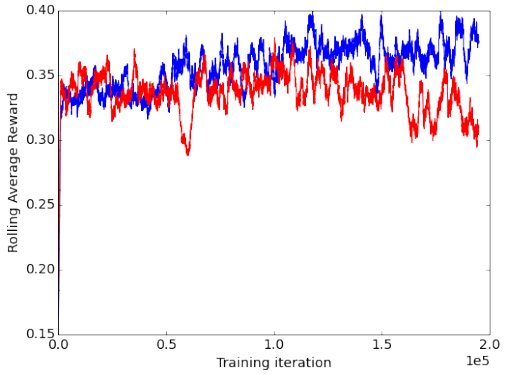
As another qualitative evaluation, we generated a t-SNE embedding of points in our final connected layer. Clear clustering is evident in both action coloring and Q-value coloring. Again inspired by [5], we visualized a sampling of some of the states in these distinct clusters. In Figure 10A, yellow corresponds to move left, blue corresponds to move right, and purple corresponds to move straight in the current direction. The frames we visualized justify these labels: in the highlighted point which is blue, for example, the car is clearly veering to the left side of the road and is best followed by a move to the right to correct course. Similar rational labeling is observed in Figure 10B.

Figure 10. t-SNE embedding of 10,000 randomly selected points in final connected layer (a) - colored by action selected by agent in that state, (b) - colored by predicted Q-value of state.



Finally, we experimented with rank-based prioritized experience replay and Double DQN. It is unclear whether these methods helped - given the increased performance overhead of prioritized experience replay, we were only able to train this version of the network to 200,000 iterations on a single trial, over which the qualitative and quantitative performance of the agent was indistinguishable from the naive agent without prioritized experience replay or Double DQN:

Figure 11. Rolling average reward from "naive" implementation, and PER+Double DQN implementation



6. Future Work

We see several paths for improvement. For one, we'd like to experiment more with prioritized replay - it seems like low-hanging fruit with its ease of implementation and its potential impact on convergence times. It wasn't clear to us if it was improving our learning or not, so trying metrics other than TD-error would be interesting. We'd also like to investigate more improvements to Q-learning, techniques in the same vein as Double Q-Learning that are relatively simple to implement and can potentially help us learn faster. Beyond changes to our algorithm, we'd like to run more experiments, and run them for a lot longer. We observed that hyperparameters can make or break a trial, so it would make sense to explore the hyperparameter space much more thoroughly than we did with out time constraints (this was also a comment made in [5]). Furthermore, we'd like to run experiments multiple times to confirm a negative result wasn't simply the consequence of bad initialization. Finally, with more time we would run our trials for far longer. Most of our experiments ran to approximately 500 thousand frames. For comparison, DeepMind was able to train on 50 *million* frames for each game they learned. Reinforcement learning performance is absolutely a function of training time, so with a couple more orders of magnitude, we would expect significantly improved results.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving, 2015.

[3] J. Gordon. JavaScript Racer. JavaScript racing game we modified.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *ICLR*, 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Human-level control through deep reinforcement learning, 2015.

[6] M. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. *ECML*, 2005.

[7] S. Ross. Interactive learning for sequential decisions and predictions, 2013. Ph.D. thesis.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay, 2016.

[9] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. arXiv:1509.06461v3 [cs.LG].

[10] C. Watkins. Learning from Delayed Rewards, 1989. Ph.D. Thesis.