Git Hash: d7efce1d42362408199fb5901de65705da61efd6

Lunar Lander

Kelly “Scott” Sims

I. Introduction

This article centers on off-policy, continuous episodic reinforcement learning tasks and state-action space generalizations. Given that it would be too computationally and memory expensive to represent every possible state-action enumeration of certain environments, it is appropriate to condense these paradigms in a generalized form. This form is more or less encapsulated in a linear function, or derivative of a linear function, such that a vectorized state input is mapped to an appropriate action-value response. The environment of interest that personifies this situation and serves as the focal point for this article is the LunarLander-v2 problem.

The LunarLander-v2 environment is one in which an agent must land safely on the surface of the moon at a specific location designated by flags. With respect to the environment’s coordinate system, the landing pad is always located at (0,0). As notated directly from OpenAI:

“Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.” - OpenAI

The state of the lander is represented by an 8-tuple $s_t = (x, y, vx, vy, \theta, v\theta, \text{leg } L, \text{leg } R)$ where (x,y) are the coordinate location of the lander at time t, (vx, vy) is the velocity of the lander in the x, y plane, $(\theta, v\theta)$ is the angle and angular velocity of the lander respectively, and (leg L, leg R) is a binary state indicating whether or not the lunar lander’s legs are touching the ground; 0 = not touching, 1 = touching . This is deemed a continuous problem because there are no discrete states.

In model-based reinforcement learning algorithms, such as Dynamic Programming and basic Temporal learning, a model of the environment is utilized as a control system to select actions that produce a policy that maximizes expected rewards. Model in this context is not the same type of “model” that we speak of in traditional supervised learning; e.g. a trained random forest model. Instead, a model as referred in model-based RL is one in which state transition probabilities is known. To know these transitions implies full knowledge of the environment. The lander can enact four different actions; do nothing, fire the left engine, fire the main engine, or fire the right engine. Between the seemingly infinite permutations of the state representation and the 4 actions available to take, solving this problem as a model-based control problem is simply not feasible. Instead, we solve this problem using function approximation.

We will show the performance of generalized state-action responses using neural networks in lieu of a traditional Q-table. More concretely, this is an exploration and analysis to the model-free solution of the Continuous Lunar Lander problem via approximating functions using Deep Q Networks (DQN).

Figure 1 - Lunar Lander attempting to land on the landing pad between the flags

II. The 1.261 Billion Foot View

The governing algorithm which was used to train the agent was Deep Q Networks with memory replay buffer. Given the inconceivability of constructing a Q table to perform simple Q-learning, the neural network is an approximator of the Q-table. A NN was initialized randomly and the agent was reset within the environment. The agent acted ε greedily given its initial state. Acting greedily in this manner simply means, a value is sampled randomly and compared to a predefined value of ε. If the value is less than epsilon, the agent takes a random action. Otherwise, the vectorized state representation is fed forward through the neural network and four values are output, representing the Q values for each action. The agent performs the action corresponding to the max Q value.

Whichever action the agent chooses, the subsequent interaction and response of the environment is committed to the agent’s memory buffer. Once memory has achieved a level of knowledge, a batch of data is sampled from memory and the neural network is trained from this data. This cycle of acting ε greedy, commit to memory, and train is repeated continuously until the agent has learned to pilot the lander to the landing pad successfully. The problem is considered solved once the agent is capable of accumulating an average reward of 200+ points over 100 consecutive episodes.

An important detail of this algorithm is the concept of exploration vs. exploitation with ε.The value of ε starts high so as to promote initial exploration of the state-action interaction with the environment. After each episode, ε is decayed so that the agent eventually, almost always acts in a greedy manner; performing the action that results in the highest expected return.

III. *VANILLA DQN*

Briefly explained here is the architecture of the solution neural network. It is beyond scope of this paper to explain the intricacies of ANNs. We simply make note of the components for continuity and reduction of ambiguity.

Given that the state space for the lander is a continuous one, a three layer neural network was trained to serve as the approximation function for the state-action Q values. This is in contrast to a simplistic discrete environment where a "look up" table is created consisting of state indices and action columns. In this lookup table, for a given state, the action which produces the highest expected future returns is chosen. As stated earlier, this type of lookup table is infeasible due to computational and memory constraints of the continuous and infinitely large state space of the lunar environment.

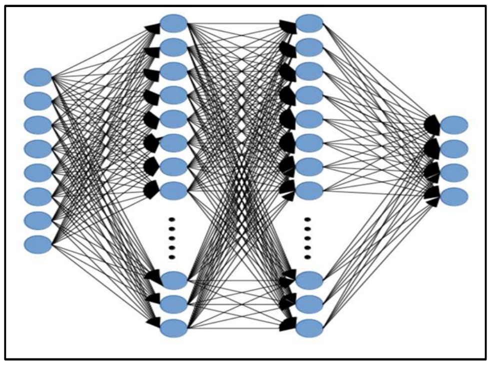

Figure 2 – Solution NN that had 8 input features mapped to two hidden layers of 256 nodes each. The final layer had 4 outputs which represented the Q-value for each available action to the lander

The neural network acts as a proxy for a traditional Q table. The 8-tuple state representation is the input to the NN, which then produces 4 output values. These values are essentially the Q values for each available action to the Lunar Lander for that given state. The agent then takes the action which gives the highest expected returns as produced by the NN. For our solution network, the input layer was an 8 feature input that mapped to the first hidden layer consisting of 256 nodes. The second hidden layer also consisted of 256 nodes which fed forward to the final output layer of 4 nodes – a node for each action. The activation function for the first two hidden layers was ReLU, rectified linear units which bounds the output of each node in a hidden layer to max(0, value). The last layer simply output the logits of the linear function. The loss function aimed to minimize mean squared error (1) using Adam optimization.

$$£ = [r + \gamma \max_a Q(s', a;\theta) - Q(s,a;\theta)]^2 \qquad (1)$$

Where $r + \gamma \max_a Q(s', a;\theta)$ is the target reward and $Q(s,a;\theta)$ is the prediction from the neural network. This is synonymous with traditional supervised learning, where mean squared error is

$$\mathrm{MSE} = \frac{1}{N}\Sigma_1^N (y_i - y_i')^2 \qquad (2)$$

With traditional supervised learning, the network is trained to generalize to a training set. This set consists of stationary independent and identically distributed samples that are fed through the network repeatedly. This poses a problem in Reinforcement Learning. The data in the context of a RL problem consists of actual state action observations the environment reveals as the agent acts upon it. This means that the data is not i.i.d nor is it stationary. The current state action observation is highly dependent on the previous. These observations are also always changing due to the different actions the agent is taking. This is where the memory buffer comes into play.

IV. *BUFFERING……*

Since there's no initial set of training data to train the neural network on, the agent creates the training data through real environment interactions. The agent takes an action and records the subsequent information: previous state S_{t-1}, reward, transition state S_t, action took, and flag that signaled if state S_t was terminal or not. Each environment reaction, as noted, was recorded to the memory buffer - a numpy array. The memory buffer was provisioned with a maximum storage limit such that once the agent filled it up, the next environment interaction would overwrite a previous entry committed to memory.

The actual memory sized used during the experiment will be discussed in full detail later. It is only important to note the consequences of a restricted memory. The neural network was trained from observations sampled from the memory buffer. Depending on how short term the memory was, it would be possible for the network to always be trained on the most recent observations. This bears significance on network performance in that it would only generalize to the most recent agent actions. If the agent was mostly exploring in the most recent actions, this could prove problematic for performance as the network would more than likely promote an exploratory response, always.

Conversely, a long term memory buffer would maintain earlier interactions when the agent was both exploring and acting greedily to the best possible action. The issue here is that initially, due to random initialization, the best possible action (the action with the highest Q value) is probably still the wrong action due to the ignorance of the agent to the environment. It hasn't had enough time to learn from anything. This bears, potentially, negative weight on the network in training. The samples from memory would have constituents of poor actions from earlier training stages combined with the more informed actions from later training steps. In this paradigm, this could create an agent that excels in mediocrity.

V. *LEARNING TO FLY*

Given the basic premise of the algorithm from section II, there's some lingering ambiguity – how to decay epsilon such that the agent explores sufficiently, yet is exploitative enough to land successfully? Another point of challenge comes from section III. Since the expected reward is dependent on Gamma, how far should the agent be concerned with maximizing future reward? Does the immediate present carry more weight? And finally, as addressed in section IV, how great should the agent's memory be? The consequences of a short term vs long term memory bank can bear significant impact on overall success. To address these key performance areas, a hyper-parameter grid search was performed:

	Value 1	**Value 2**	**Value 3**
Gamma (γ)	0	0.5	0.99
Memory Size	10,000	100,000	1,000,000
ε - Decay	1e-5	1e-4	1e-3

During training, nine different hyper-parameters were utilized, three parameters per the three different points of interest noted above. Gamma was investigated across a range of [0.0, 0.5, 0.99], revealing to us agent responses when only the immediate present is of concern, an equal weight between the present and distant future is accounted for, or maximizing future reward is key, respectively. Second, memory size was altered to reflect short, medium, and long term memory across [10000, 100000, 1000000] entries of capacity. And the last tuning metric centered on attenuating the agent's exploratory tendencies. In all scenarios, epsilon was initialized to be 1. After each action the agent took, epsilon was decayed across the search of [1e-5, 1e-4, 1e-3]. E.g. if decay rate was 1e-3, after the first action step an agent performed, epsilon was decayed 1 – (1e-3) = 0.999.

A. *Tunnel Vision is Blinding*

An interesting consequence observed by varying gamma is that acting with the future in mind is always better than caring about the immediate. There was no case in which being more concerned with immediate rewards was fruitful.

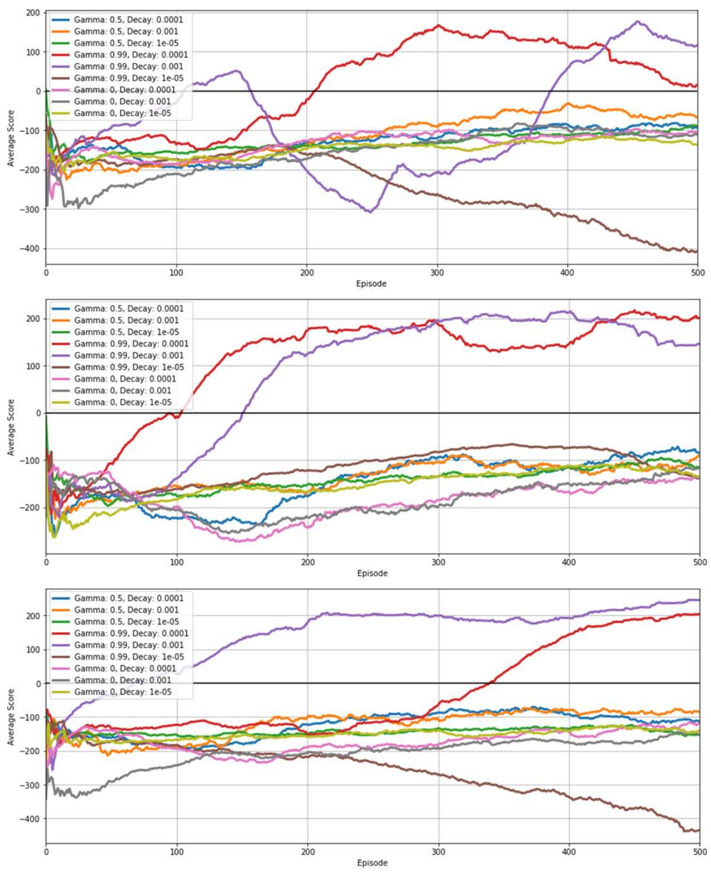

Figure 3: Average score shown over multiple hyper-parameters. (Top) Memory buffer of 10,000. (Middle) Memory buffer of 100,000. (Bottom) Memory buffer of 1,000,000.

Each graph in Figure (3) shows average reward per 100 episodes for the agent during training. The top graph shows training with a memory buffer of 10,000. The middle and bottom graph have 100,000 and 1,000,000 respectively for memory buffer. In all instances, the agent performed the best when it was acting in the interest of maximizing long term rewards. The purple and red curves are representative of an agent acting with gamma of 0.99 and epsilon decay of 1e-3 and 1e-4 respectively.

Interestingly enough, the above analysis does show that the best trained agent is one that attempts to maximize long term rewards. They also show that the worst trained agent is one that attempts to maximize long term rewards, yet is almost always in an exploratory state. The brown curve in each graph is an agent with gamma of 0.99 and an epsilon decay of 1e-5. At this decay rate, even across all other training sessions, epsilon remained significantly large throughout the entire training process. After 500 episodes, in only a couple of training sessions, did epsilon drop below 0.3. This constitutes high exploration throughout the entire training process.

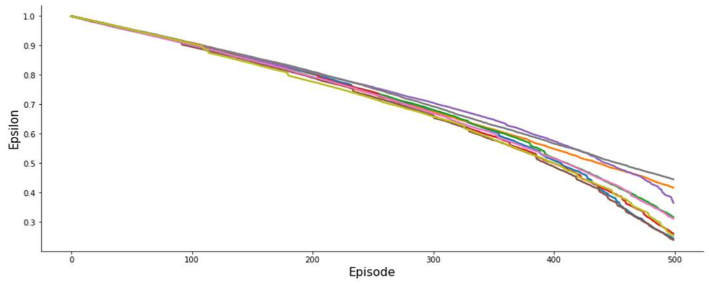

Figure 4 - Value of epsilon after each episode for all grid search hyperparameters with decay rate of 1e-5

Given that any value of epsilon other than 0.99, or an epsilon decay not greater than 1e-5, produces sub optimal agents, the analysis' from this point will not make mention to the results from those training sessions which do not meet these thresholds.

B. *Memento*

In searching over optimal memory capacity, it was seen that having short term memory is not ideal. Intuitively, this makes sense. Because training data was sampled from memory, the neural network was always being trained from more recent environment action-observations. What the most recent information comprised of was a function of epsilon. Recall that all training sessions were initialized with an epsilon of 1, so the agent was almost always exploring. We will show in a later section that a lot of exploration in this environment was not very beneficial. This result on short term memory, however, already alludes to this conclusion. From the top graph in Figure 3 and Figure 5, we can see that it took significantly more episodes for the agent to net a positive average score in comparison to the long term memory capacity in any training session. It is also clear that it never achieved the maximum average reward that either the mid or long term memory achieved as well. Figure 5 is a consolidated view of the average rewards where gamma equals 0.99.

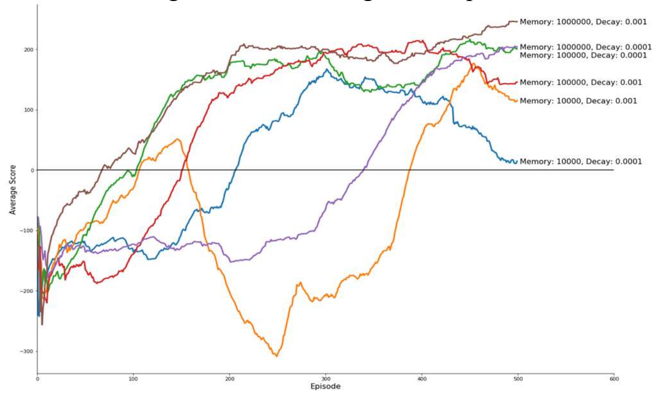

Figure 5 - Average rewards (Gamma = 0.99)

Analyzing mid memory versus long term memory, it is easy to see that long term memory outperformed all variants ultimately. But the interesting thing to note is that midterm memory did "solve" the environment, but it also suffered from the same complication as the short term memory. It's just not as prevalent since the buffer is larger.

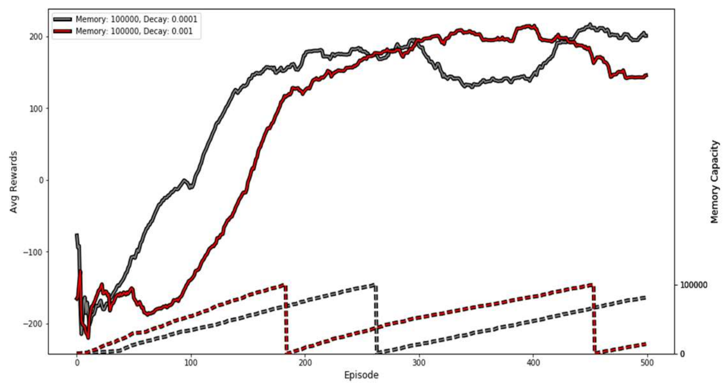

Figure 6 - Average rewards compared to buffer fill capacity for midterm memory and gamma of 0.99

Figure 6 shows average reward over how full the memory buffer is for mid memory capacity. In the bottom two dashed curves, when they reset to 0, this is indicative of old memories being overwritten by new memories. The gray curve is an agent that explores more often than the red curve. It just so happens to be the case that the exploration here starts off netting a lot of positive reward. But after the buffer fills and has been overwriting memories, the performance drops. This same type of trend happens with the red agent between episodes 200 to 500. As the buffer fills, the agent's performance increases. However, as the buffer gets full, the performance degrades. This is deemed to be a consequence of the agent being trained on the more recent action-observation sequences and no recall of earlier experience (due to them being overwritten). For the Gray agent, the drop in performance happened after memories were being overwritten the first time. For the red agent, the drop happened after the earliest memories were fully overwritten by the most recent observations.

In contrast to this paradigm, we don't see this roller coaster ride of performance with the long term memory (Fig 7). After substantial amount of memory has been recorded, the performance trend of the agent stabilizes in both cases. There's also never any crossing in performance between the exploratory agent vs. the greedy agent.

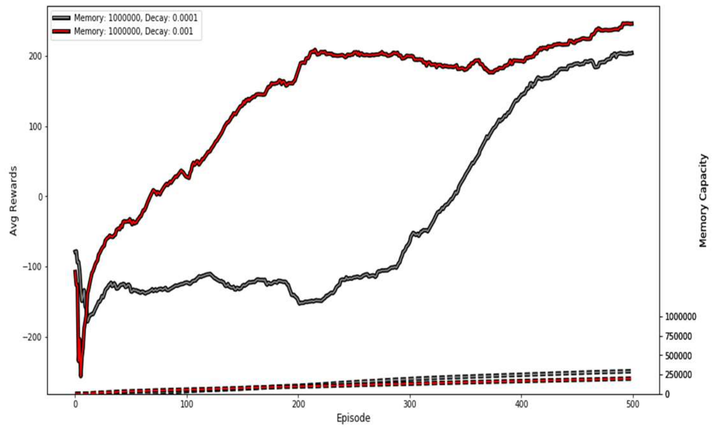

Figure 7 - Average rewards compared to buffer fill capacity for midterm memory and gamma of 0.99

C. Lunar Greed

We have shown that a long term memory agent seeking to maximize future rewards performs better than all other agents in our training. With that comes only the matter of how much exploration should the agent do. Given that its memory is almost infinite in comparison to the amount of memories it stores within the 500 episodes of training, the issue of forgetfulness is not a concern here. But as stated in earlier sections, exploring didn't seem to be in the agent's best interest.

Figure 7 shows the agent that did the least amount of exploring, performed the best, and accumulated more positive reward in fewer episodes. Figure 8 shows just how quickly the agent stopped exploring compared to the more exploratory agent.

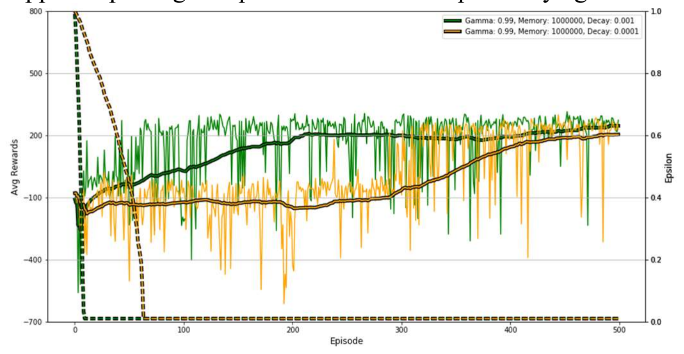

Figure 8 - Effects of exploration vs Exploitation on score achieved. The agent that explores less performs better. After the exploratory agent stops exploring, its performance takes off

After nine episodes, the agent with a decay rate of 1e-3 was acting greedily for the duration of its training with only a 1% chance of acting randomly. The agent with 1e-4 decay, however, took 64 episodes. And in those 64 episodes, approximately 10,000 entries of memory had been stored of a random actions.

For the latter agent, performance remained low until enough memory had been stored so that the amount of greedy actions dominated the amount of random actions. After some time, we see reward begin to climb rapidly. This is only speculation of the root cause for increase in performance of the exploratory agent. No stats were collected on which memory points were sampled for the batch training. This same trend, however, did reveal itself in a lot of the other training sessions with a decay rate of 1e-4.

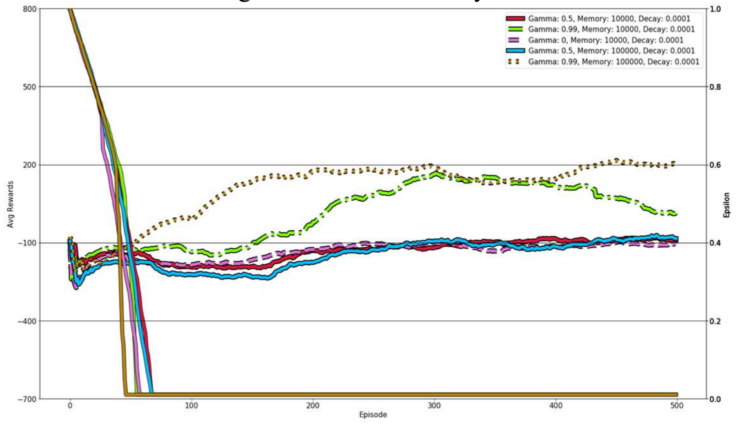

Figure 9 - Epsilon decay of 1e-4 for different Gamma and Memory size

From the figure (9), the trend is clear in that epsilon completely decays, then after some amount of time of greedy actions being written to memory, performance begins to increase. Given this response, and the performance of the non-exploratory agent, these paradigms create a strong argument for generating an agent that is interested only in maximizing future rewards with minimal exploration. This agent should also have long term memory for complete synthetic recall.

VI. THE MAVERICK

A complete and isolated training summary can be seen for the best performing agent below in Figure 10.

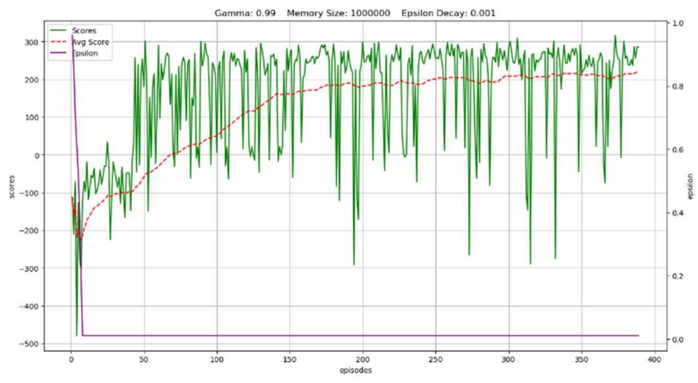

Figure 10 - Best performing agent. Gamma: 0.99, Memory Size: 1000000; Decay Rate: 0.001; Maximum achieved Average Score: 219

As stated earlier, epsilon decays almost immediately, and within a few episodes of that, the agent excels. A happy byproduct of a quick learner, is the time to train is actually faster. When plotting out time per episode (Figure 11), it is very apparent when the agent is exploring, it takes longer time to fail. In fact, there's a direct correlation between success and time per episode. Comparing the raw inter-episode scores as well as the raw times per episode, anywhere there is a massive drop off in score, there is a huge spike in time. Conversely, as score improves, time spent on an episode is minimal; around 2 to 3 seconds on average.

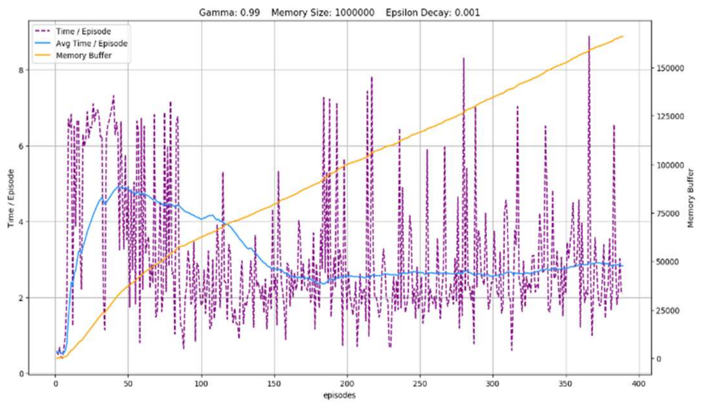

Figure 11 - Time per episode, average time per episode and memory capacity per episode. The obvious association from this is that more data is written to memory when an episode takes longer. This is very trivial, but as expected.

Another apparent, but trivial aspect seen in training the best agent is the amount of memory being stored is directly related to time spent on an episode. The memory capacity has the steepest rate of change early on when the time per episode is the largest. By reducing the amount of time needed to complete an episode, the amount of computational memory is reduced. This is nothing ground breaking. It only serves to highlight an often overlooked aspect in training; it isn't enough motivation to only train a high performing agent, it is in ones best interest to facilitate fast learning as well.

VII. *THE LANDING*

It came as big surprise that the simple vanilla DQN algorithm was enough to solve this problem. Initially the attempt to solve the environment was by using Double DQN (DDQN). This attempt was unsuccessful. The model kept diverging greatly, and the agent's response was to simply blast off to mars as opposed to landing. It is believed that this issue stemmed from poor implementation. Network responses alluded to an improper mixing of weight updates and target Q values between the two networks. This is what is believed to have caused the divergence. After about two days of tweaking and debugging, the decision was made to implement the vanilla DQN to attenuate model complexity. This decision also helped all steps of debugging, tremendously.

The challenges didn't end there however. There is no hard and fast rule about neural network architecture. A separate hyper-parameter search could have been performed just for the neural network. Number of hidden layers, how many nodes in each layer, which activation functions to use, batch normalization or not, these are some of the major touchpoints in NN architecture. The architecture settled on in this experiment was birthed from discussions with fellow students on current failures vs slight successes. There were many iterations of failures before this one however. Each architectural tweak was monitored for results, and then tweaked again in the direction that provided the desired effect; increase in agent performance.

If more time was granted, it would be an area of interest to study the effects of specific or controlled batch sampling for training. Some of the analyses performed in this paper makes assumptions due to trends in the data with no factual backing on memory training. However correlation is not causation. Maybe performance could be improved more rapidly if the agent was trained on a uniform set of batch memories. By this, it is meant that each batch would have equal parts very early memory to mid memory to most recent memory. After thousands of memory updates, it's not infeasible to believe that the sampling of memory could become very imbalanced. It would be interesting to see just how much memory from each time step the agent was using, and use this knowledge to enhance memory training.

REFERENCES

[1] Sutton, R. & Barto, Andrew. Reinforcement Learning 2nd Edition. MIT Press, Cambridge, MA, 2018

[2] Silver, D. 2015 Lecture 6: Value Function Approximation, University College London, delivered May 13, 2015

[3] Silver, D. 2015 Lecture 6: Value Function Approximation, University College London, delivered May 13, 2015

[4] Sentdex. "Deep Q Learning and Deep Q Networks (DQN) Intro and Agent - Reinforcement Learning w/ Python tutorial p.5. https://pythonprogramming.net/deep-q-learning-dqn-reinforcement-learning-python-tutorial/

[5] Choudhary, Ankit. "A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python. April 18, 2019. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[6] Zychlinski, Shaked. "Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes. Jan 9, 2019. https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677