DeepMind's Deep Q-Network Implementation with t-Soft update for Reinforcement Learning

Adarsh Gouda, P.Eng.
Mechanical Engineer - AI Practitioner
adhi.pesit@gmail.com

***Abstract*—This paper describes Deep Q-Network (DQN) implementation to solve OpenAI's Lunar Lander-v2 [1] environment by attempting to replicate the DQN Algorithm proposed by Mnih et al. [2]. The fundamental idea behind DQN is that the Q-Learning is now backed by a Stochastic Gradient Descent Neural Network to *predict* Q-values. The paper also discusses the fundamentals of Q-Learning, Function Approximation and Artificial Neural-Nets (ANN) that are quintessential in implementation of DQN algorithm. Finally, I discuss the results and influences of key hyperparamenters on the agent's learning performance.**

I. Introduction

The objective of any reinforcement learning task is to train an *agent* which interacts with its *environment*. The agents purpose it to maximize total *reward* across all episodes from its interaction with the environment. An *episode* is the event time-frame between the first state and the last or the terminal state the agent encounters. The agents objective is to learn to navigate/operate in the environment from its experience. The strategy that the agent develops through iterative learning is called a *policy*. Q-Learning is a simple yet quite a powerful algorithm to create a *look-up table* to help the agent develop an approximate optimal policy. However, if the environment is high-dimensional or continuous, unlike a grid world, it is extremely challenging and computationally expensive to develop a look-up table. DeepMind developed a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. This paper follows the implementation of DeepMind's algorithm in solving the Lunar Lander environment. The Lunar Lander-v2 problem is considered solved when 100 Simple Moving Average(SMA) reaches 200 reward within 2000 episodes.

II. Temporal Difference Control

For prediction problems, state value estimates are generated using Temporal Difference learning methods. For control problems, instead of estimating state-value function $v(s)$, we estimate action-value function, $q(s, a)$. Unlike $v(s)$, $q(s, a)$ gives us the value of actions without having to use an MDP. Q-value estimates are then used to improve the policies. The learning progresses similar to *Generalized Policy Iteration (GPI)*. There are two approaches to evaluate and improve policies On-Policy methods where agent learns to assess and improve the same policy it uses for generating data. And Off-Policy methods where an agent learns to assess and improve a policy different than the one used for generating the data.

A. SARSA, On-Policy

In this On-Policy method, we must estimate $q_\pi(s, a)$ for the current behaviour policy π and for all states and actions. This can be done using essentially the same TD method used on $V_\pi(s)$. The update equation is given below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \\ - Q(S_t, A_t)]$$

This update rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which brings about the name SARSA for this On-Policy learning method. SARSA selects the next action according to the available approximate optimal policy and updates its Q-values accordingly. This works quiet well on discrete low dimensional state-space problems (like a grid world). However, for high dimensional and/or continuous state-space problems, this approach can be computationally expensive.

B. Q-Learning, Off-Policy

In Q-Learning, the learned action-value function, Q, directly approximates optimal action-value function, independent of the policy being followed. Which is why Q-Learning is an Off-Policy method.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \\ - Q(S_t, A_t)]$$

The key difference between SARSA and Q-Learning is that, Q-Learning estimates rewards for the next-step action from the current Q-values by applying $argmax$ across all actions in the next state. In this paper, we will use Q-Learning backed by Artificial Neural Network (ANN).

III. Off-Policy methods with Approximation

The problem with large state spaces is the memory, time and data needed to accurately fill the look-up Q-tables with the action-state values. The key idea behind *Function Approximation* is generalization i.e., with examples from a desired function attempt to generalize and construct an approximation of the entire function.

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

In the equations above, $\hat{v}(s, \mathbf{w})$ and $\hat{q}(s, a, \mathbf{w})$ are approximate functions where $w \in \mathbb{R}$, is a weight vector updated using MC or TD methods. There are several function approximators but in this paper the focus is on differentiable function approximators and in particular, non-linear function approximator, the artificial neural network.

A. Stochastic Gradient Descent (SGD)

SGD methods are among the most widely used of all function approximation methods and are particularly well suited from reinforcement learning. In SGD methods, the objective is to find the parameter vector $\mathbf{w}$, that minimizes the expected Mean-Squared Error (MSE) between approximate action-value function $\hat{q}(S, A, \mathbf{w})$ and true action-value function $q_\pi(S, A)$.

$$J(\mathbf{w}) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

Here, $J(\mathbf{w})$ is a differentiable function of parameter vector $\mathbf{w}$ and the update rule to minimize the expected MSE using SGD will be:

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) \\ &= \alpha\mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_w\hat{q}(S, A, \mathbf{w})] \\ &= \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_w\hat{q}(S, A, \mathbf{w})\end{aligned}$$

$\Delta\mathbf{w}$ is a column vector and the expected update in the above equation is a full gradient update when SGD samples the gradient.

B. Linear Action-Value Function Approximation

Linear methods are considered as special cases of function approximation where $\hat{q}(S, A, \mathbf{w})$ is a linear function of weight vector, $\mathbf{w}$. The state-action is represented as a feature vector:

$$x(S, A) = \begin{bmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{bmatrix}$$

The approximate action-value function can then be represented as linear combination of these features.

$$\hat{q}(S, A, \mathbf{w}) = x(S, A)^T\mathbf{w} = \sum_{j=1}^{n} x_j(S, A)\mathbf{w}_j$$

And the corresponding SGD update for such linear case is:

$$\nabla_w\hat{q}(S, A, \mathbf{w}) = x(S, A)$$
$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))x(S, A)$$

Like prediction problems, we must substitute a *target* for $q_\pi(S, A)$ with MC, TD(0) or TD(λ) incremental control approach. The linear methods have shown convergence guarantees and are computationally efficient. However, the implementation on real-world problem is challenging. Especially so for high dimensional state-space problems where the feature vector has to be constructed to best describe the state-space and feature interactions.

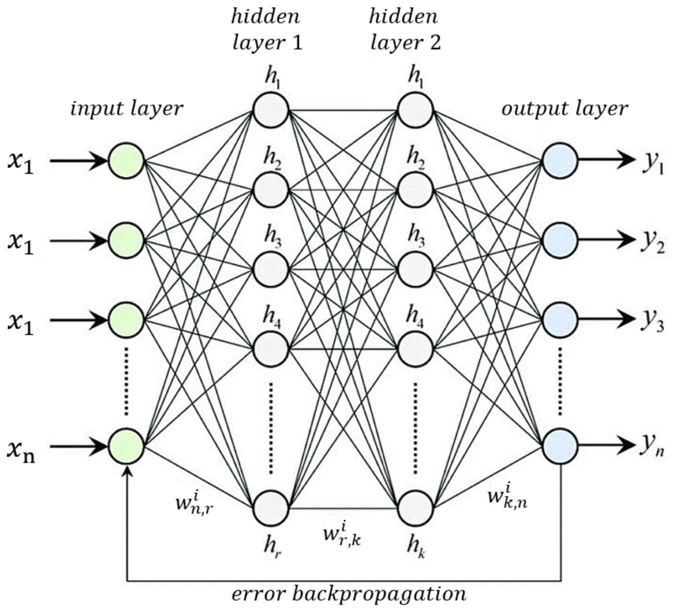

Fig. 1. A generic, fully-connected, feed-forward ANN with two hidden layers.

C. Non-Linear Function Approximators - ANNs

An alternative to linear function approximators is to use a much richer function approximator that is able to go from just the states without requiring an explicit specification of features. ANNs are widely used for nonlinear function approximation. Figure 1 shows the generic architecture of a feed-forward ANN. ANNs are built like human brain, with neurons or nodes interconnected like a web. A set of input variables are passed through different neurons, whose output is a non-linear function of contents of the previous layer. Shown in

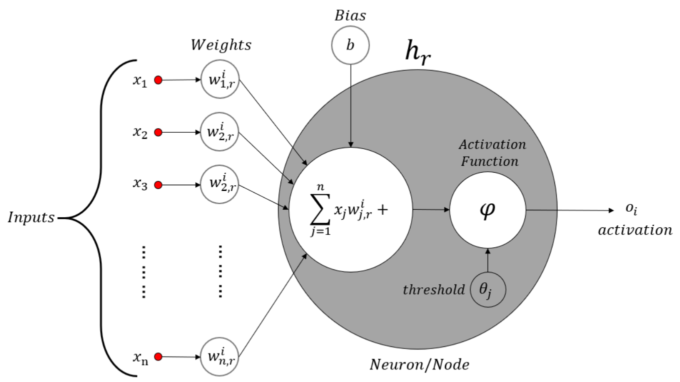

Fig. 2. Functioning of a single neuron (or, node) in 1st hidden layer at i^{th} iteration in an ANN

Figure 2 is a single neuron (represented as circles in hidden layers in Figure 1) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function* to produce the neuron's output. Each successive layers compute increasingly abstract representations of the network's "raw" input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network. Training the hidden layers is a way to automatically create approximate features for a given problem without relying exclusively on hand-crafted features.

IV. Deep Q-Network (DQN)

DQN combines Q-learning with ANN. Essentially, the Q-network is learned by minimizing the following mean squared error:

$$J(\mathbf{w}) = \mathbb{E}_{(S_t,A_t,R_t,S_{t+1})}[(y_t^{DQN} - \hat{q}(S_t, A_t, \mathbf{w}))^2]$$

where y_t^{DQN} is the one-step ahead learning target,

$$y_t^{DQN} = R_t + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w})$$

and $\mathbf{w}$ represents parameters(weights) of the Q-network.

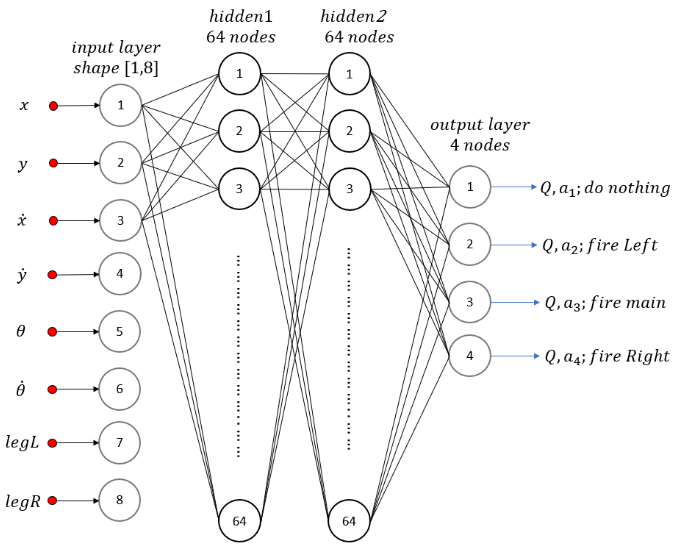

Fig. 3. Q-network setup for the Lunar Lander-v2 problem. Shown only first 3 nodes connected for simplicity. However, the ANN is a fully-connected network.

Mnih's paper has 3 primary contributions (1) ANN as a nonlinear approximator that maps state to action-value; (2) using mini-batches of random training data rather than single-step updates on the last experience, famously called *experience replay*; and (3) using a separate *target network*, $\tilde{q}$ with aged network parameters to predict the Q-values of the next state. Let's use the above 3 pointers to setup the premise for solving this environment.

In this paper, LunarLander-v2 [1] is used as the environment with which our agent interacts and learns to land the Lunar Lander on a fixed landing pad.

A. Building the ANN Q-Network for LunarLander-v2

In DQN, a ANN model serves the purpose of a Q-function. For the LunarLander-v2 environment, the model I have setup consists of a fully-connected, feed-forward network with 2 hidden layers each containing 64 nodes as shown in Figure 3. The input layer takes a 8-length vector for state values and the output layer produces 4-length vector of Q-values for each action, given the state. The *Loss Function* chosen is Mean Squared Error, *Optimizer* is ADAM and *Activation Function* is ReLU.

B. Catastrophic Forgetting and Experience Replay

Catastrophic Forgetting [5] is a phenomenon suffered by ANNs where a model can drastically loose its generalization abilities on a task after being trained on a new task. This is detrimental to the performance of a RL agent that tends to "memorize" rather than learn. Experience Replay basically gives us batch updating in an online learning scheme. Python's *deque* is used to store agent's experience memory.

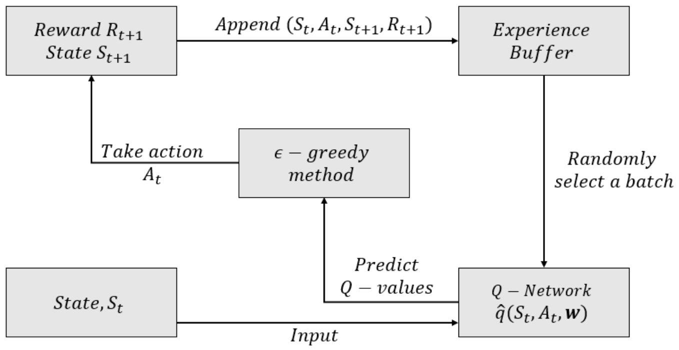

Fig. 4. General overview of Experience Replay, a method to mitigate Catastrophic Forgetting.

C. Using a Target Q-Network

One potential problem that DeepMind identified in their paper [2] is that, if we keep updating the Q-network parameters after each move, we might cause instabilities to arise. This instability has several reasons including correlations present in the sequence of observations and the fact that small updates to $\hat{q}$ may significantly change the policy and therefore change the data distribution. To address this, the paper proposes using another Q-network $\tilde{q}$, a copy of the online network $\hat{q}$, to generate target Q-values. At $t = 0; \tilde{q} = \hat{q}$. As the time steps, the update of $\tilde{q}$ lags behind $\hat{q}$. The target y_t^{DQN} will now become:

$$y_t^{DQN} = R_t + \gamma \max_{a'} \tilde{q}(S_{t+1}, a', \mathbf{w}^-)$$

where $\mathbf{w}^-$ represents the parameters of the target network, and the parameters $\mathbf{w}$ of the online network are updated by sampling gradients from the minibatches of past transition tuples (S_t, A_t, R_t, S_{t+1}). Figure 5 provides an overview of DQN with a target Q-network.

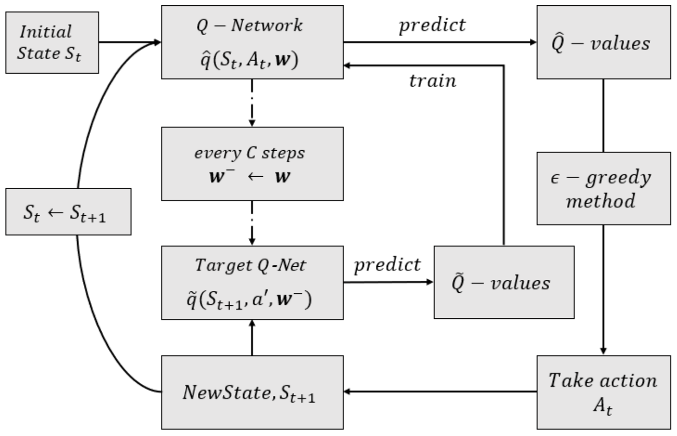

Fig. 5. Q-Learning with a target network.

D. t-Soft weight updates

Kobayashi Y, et al. (2018) [3] proposed a new robust update rule of parameters of the target network called t-soft update rule inspired by student-t distribution. Since SGDs are unstable in the optimization of nonlinear function approximations, instead of $\mathbf{w}^- \leftarrow \mathbf{w}$ for every C steps, a soft update is implemented:

$$\mathbf{w}^- \leftarrow (1 - \tau)\mathbf{w}^- + \tau\mathbf{w}$$

where τ denotes the smoothness, and if $\tau = 1$, this update rule becomes the hard update as implemented in [2].

TABLE I
DESIGN OF EXPERIMENTS

Exp	ϵ	γ	τ	α	Con.	Eps	Score
1	0.995	0.99	0.010	0.0001	No	2000	0.2 %
2	0.995	0.99	0.010	0.001	Yes	562	85.2 %
3	0.995	0.99	0.010	0.01	No	2000	0.2 %
4	0.995	0.99	0.100	0.0001	No	2000	0.0 %
5	0.995	0.99	0.100	0.001	Yes	755	85.0 %
6	0.995	0.99	0.100	0.01	No	2000	0.0 %
7	0.995	0.99	1.000	0.0005	Yes	567	88.2 %
8	0.995	0.99	1.000	0.0001	No	2000	0.6 %
9	0.995	0.99	1.000	0.001	Yes	571	88.8 %
10	0.995	0.99	1.000	0.01	No	2000	0.0 %
11	0.010	0.99	0.001	0.0005	No	2000	0.0 %
12	0.950	0.99	0.001	0.0005	No	2000	0.6 %
13	0.995	0.99	0.001	0.0005	Yes	1776	89.0 %
14	1.000	0.99	0.001	0.0005	No	2000	0.0 %
15	0.995	0.95	0.001	0.0001	No	2000	0.0 %
16	0.995	0.95	0.001	0.0005	No	2000	0.0 %
17	0.995	0.98	0.001	0.0001	No	2000	5.4 %
18	0.995	0.98	0.001	0.0005	Yes	1424	81.4 %
19[a]	0.995	0.99	0.001	0.0005	No	2000	0.0 %
20[b]	0.995	0.99	0.001	0.0005	No	2000	0.0 %

[a] 16 nodes, [b] 32 nodes, rest all Exp. 64 nodes in hidden layer.

V. EXPERIMENTS AND ANALYSIS

A. Train

To understand the effects of each hyper-parameter - ϵ, decay rate for ϵ-greedy method; γ, the discount factor; τ, the t-Soft update; α, the learning rate for ANN optimizer - 20 experiments were devised. Table I shows all the values of the hyper-parameters tested as part of these 18 experiments. Column *Con.* shows if the Experiment converged or not and *Eps.* column shows the number of episodes the test was run before converging or the maximum episodes before termination, which is 2000 episodes.

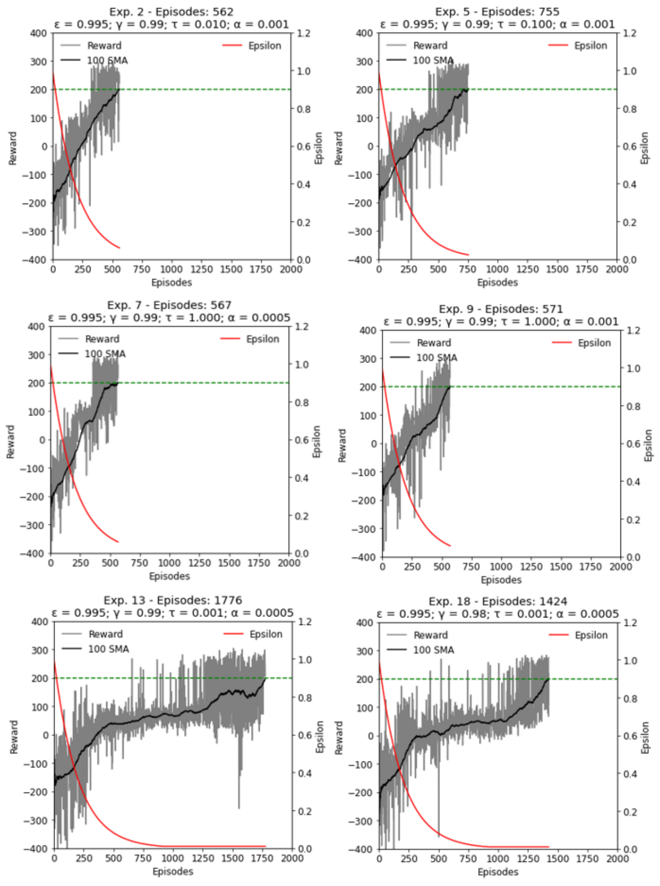

Fig. 6. Experiments that converged within 2000 episodes.

For each episode during training, 500 steps were chosen and at every C=4 steps, the target network parameters were t-Soft updated. The buffer memory was set to 1,000,000 so that no observations were dropped off the buffer memory. A batch size of 64 was chosen to train the Q-Network at each step of a given episode until 500 steps or a terminal state is reached

B. Test

The trained model was saved for each Experiment, a total of 18 models. Each model was tested in the Lunar Lander-v2 environment for 500 episodes with 1000 steps each. The reward, average reward were plotted. An episode with reward equal to or exceeding 200 was considered a win and a Win Rate was determined averaging over 500 episodes.

VI. ANALYSIS AND CONCLUSION

As seen in Table I, only 6 out of 18 experiments converged within 2000 episodes and showed promising results during the tests. Figure 6 shows training plot of Experiments that converged. As seen in Figure 6, Exp 2, 5, 7, 9, 13 and 18 converged within 800 episodes. The corresponding test case performances are shown in Figure 7.

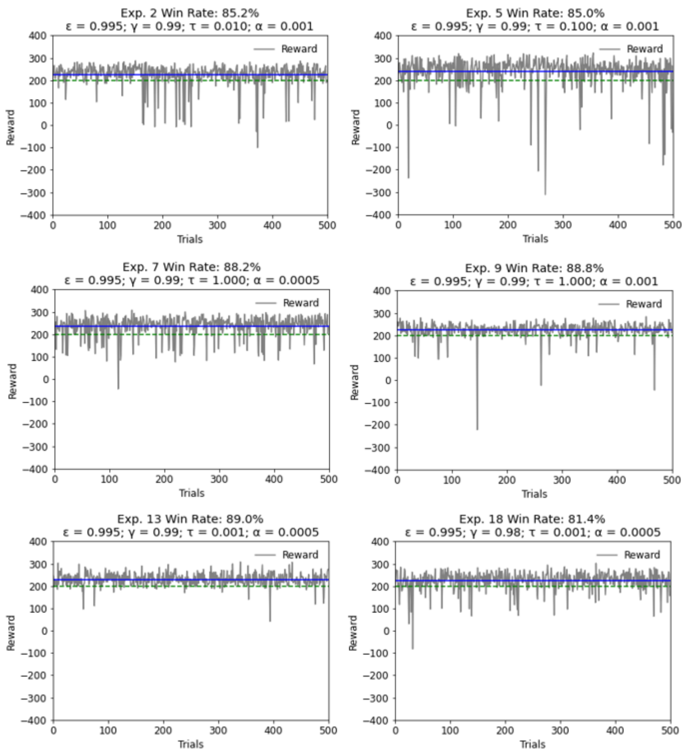

Fig. 7. Test results of Experiments that converged. Exp13 has the best Win Rate of 89.0 %

1) Effect of Learning Rate, α: The learning rate for the Adam optimizer plays a very important role in training the agent. Figure 8 shows the soft and hard update cases with varying α. Clearly, $0.0005 \geq \alpha \leq 0.001$ is an ideal range for the ANN model setup.

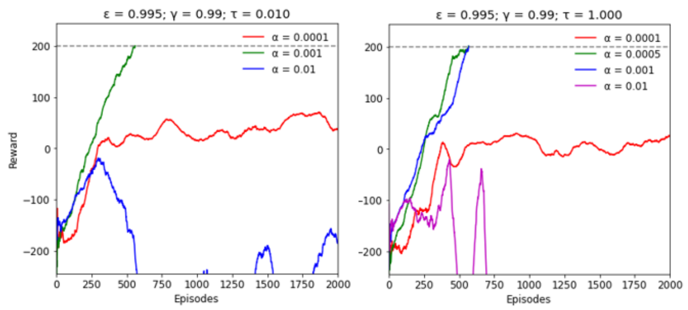

Fig. 8. Reward 100 SMA Soft-update(L) and Hard-update(R) and varying α

2) Effect of t-Soft Update, τ: Referring to Figure 9(L) the soft-update did not yield better results than hard-update of the target network parameters. It is likely that the choice of other hyper-parameters overshadowed the significance of t-Soft update. More experiments are needed to understand the effects of τ on number of nodes in the ANN hidden layer.

3) Effect of discount-factor, γ: Referring to Figure 9(R), it is clear that the discount factor is one of the important hyper-parameter. When $\gamma < 0.98$, it is seen that the Lander just learns to hover and not attempt to actually land. Intuitively, the agent does not value the future rewards as much to see the greater reward in performing the landing.

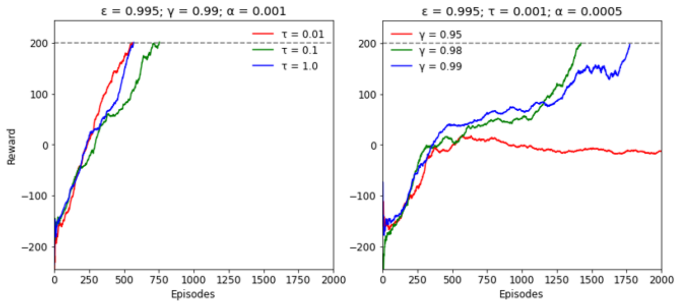

Fig. 9. Reward 100 SMA plot - varying τ (L) and γ (R) values.

4) Effect of ϵ decay for ϵ-greedy policy: Referring to Figure 10(L), as expected, the ϵ value has significant influence on the learning behaviour of the agent. As seen, $\epsilon = 0.995$ gives the best result while $\epsilon = 1.00$, a full greedy policy, shows no signs of learning.

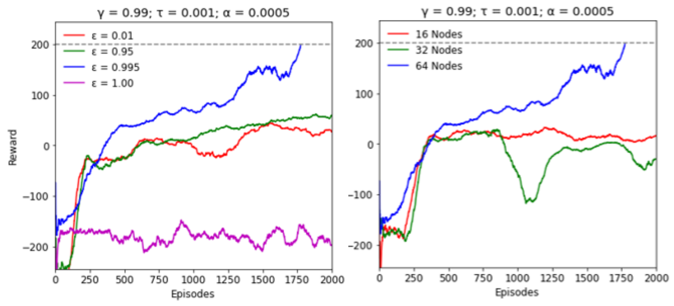

Fig. 10. 100 SMA of reward with varying ϵ(L) and hidden layer nodes(R)

5) Effect of number of nodes in hidden layers: Referring to Figure 10(R), the number of nodes in the ANN model is another key factor in convergence. 2 hidden layers with 16, 32 and 64 nodes trained with same hyper-parameters produce significantly different results. Clearly, 64 nodes gives the best result.

REFERENCES

[1] Lunar Lander v2 - https://gym.openai.com/envs/LunarLander-v2/

[2] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[3] t-Soft Update of Target Network for Deep Reinforcement Learning arXiv:2008.10861

[4] Playing Atari with Deep Reinforcement Learning arXiv:1312.5602

[5] French, R. M. Catastrophic forgetting in connectionist networks. Trends in cognitive sciences, 3(4):128–135, 1999