ML Major Project: Cartpole Problem

Anirudh Mehra (2016A7PS0033G)
Ishita Mediratta (2017A7PS1013G)
Shubhad Mathur (2017A7PS0129G)

April 21, 2020

Abstract

This report contains our approach towards solving the Cartpole environment with different noise/sensor settings. We begin by explaining the different tasks given in the problem statement, followed by a brief overview of different Reinforcement Learning algorithms that we made use of and the results obtained using their implementation.

1 Introduction

"Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment."

(Excerpt from Sutton and Barto's Reinforcement Learning: An Introduction 2nd edition)

In simpler words, Reinforcement Learning involves training an agent to maximise the reward signal it obtains from the environment over the long term. Within the scope of Machine Learning, Reinforcement Learning forms one of the three basic paradigms, alongside supervised learning and unsupervised learning.

Reinforcement Learning differs from supervised learning in not needed labelled input-output pairs, as well as not needed sub optimal actions taken by the agent to be corrected. Instead, the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). It differs from unsupervised learning in that it does not involve learning the structure of the data.

Reinforcement Learning, while associated with machine learning, is studied with multiple other disciplines, like game theory, control theory, and operations research. The advances in Reinforcement Learning over the last decade, as well as the recent successes of DeepMind's AlphaGo (an agent that plays the game of Go) [10] and AlphaZero (an agent that plays the game of chess) beating the best human players as well as the best computer programs, has brought the field in a spotlight.

2 Problem Statement

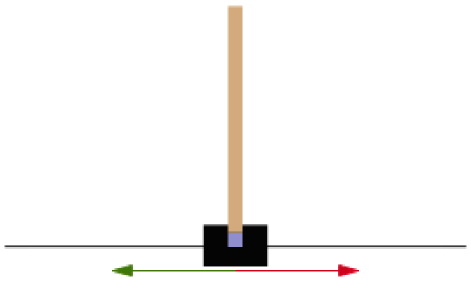


Figure 1: A render of the Cartpole environment

This project revolves around a modified implementation of the celebrated Cartpole environment [1]. In the original OpenAI gym's Cartpole environment, there's a pole which is attached to a cart via an un-actuated joint. The goal is to prevent the pole from falling, i.e. when it's more than 12° from the vertical, and also to keep the cart within 2.4 units from the center for a maximum of 500 timesteps per episode.

The pole can be controlled under the center of mass by moving the cart either left or right, and hence applying a force of either -1 or +1 respectively. (See Figure 1)

Note that the original implementation consists of a friction-less track, and is defined **solved** if one gets an average reward of **475.0** over 100 consecutive episodes.

Tables 1 and 2 show the defined State and Action spaces.

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-24°	24°
3	Pole Velocity At Tip	-Inf	Inf

Table 1: State Space for Cartpole Environment

Num	Action
0	Push cart to the left
1	Push cart to the right

Table 2: Discrete Action Space for Cartpole Environment

The increase or decrease in velocity is dependent on the angle the pole is pointing at, because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it

2.1 Task 1

Here, the values for gravity and friction (for both, cart and pole) vary randomly at each timestep of an episode.

2.2 Task 2

This task contains not only the variations given in Task 1, but also noisy control settings i.e. the cart's force in the desired direction may be less/more than expected.

2.3 Task 3

Includes the modifications in Tasks 1 and 2, as well as, noisy sensors/sensor observations of the pole angle at any moment.

3 Theory and Approach [11]

Consider a state space space S, action space A, reward space R (which is a subset of real numbers) and a space of policy functions Π.
In all reinforcement learning problems, we have two entities constantly interacting with each other in an alternating manner:

1. an Environment
2. an Agent

Any reinforcement learning problem can be seen as a stochastic process where at all times, the agent resides at a state $s \in S$ and chooses an action $a \in A$ in accordance with a policy $\pi(a|s) \in \Pi$ (which is a conditional probability distribution). To this, the environment responds with a reward $r \in R$ and moves the agent to a new state $s' \in S$ with a probability $p(s', r|s, a)$. The whole process is repeated for a certain number of time steps, finishing an "episode" which is a sequence of states, actions and rewards.

$$s_0 \ a_0 \ r_1 \ s_1 \ a_1 \ ... \ r_T \ s_T$$

Our goal is to find a policy that maximizes the expected return G the agent gets after an episode.

$$G = \mathbb{E}_\pi[r_1 + \gamma r_2 + \gamma^2 r_3 + ... \ + \gamma^{T-1} r_T]$$

Let's consider a function $v_\pi(s)$, which is the expected return if the Agent starts from state s under policy π. It is easy to see that

$$v_\pi(s) = \sum_a \sum_{s'} \sum_r \pi(a|s) p(s', r|s, a)(r + \gamma v_\pi(s'))$$

Similarly define $Q_\pi(s, a)$ as the expected return if the agent starts from state s and performs action a under policy π. Then,

$$Q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a)(r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a'))$$

The relation between the two can be shown to be

$$v_\pi(s) = \sum_a \pi(a|s) Q_\pi(s, a)$$

Since our goal is to find an optimal policy π_* that gives the maximum expected $v_*(s)$ return for any state s, it can be shown that

$$v_*(s) = Q_*(s, \pi_*(s))$$

$$v_*(s) = \max_a Q_*(s, a)$$

where Q_* is the expected return value for state value pair under π_*

$$Q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)(r + \gamma \max_{a'} Q_*(s', a'))$$

with our optimal policy being

$$\pi_*(s) = \operatorname*{argmax}_a Q_*(s, a)$$

As we can see, we only require Q to find an optimal policy.
If we know p, we can use methods like value iteration with dynamic programming to find the optimal policy. But in the case of cart pole problem, we don't know what p is. So, we have to use model free approaches, where we sample episodes to estimate return values and update our current policy.
In model free methods, we estimate returns and update our policy together at each iteration. Each iteration end at each state transition (Temporal Difference approach) or only after the episode ends (Monte Carlo approach). The general outline is:

1. Policy evaluation: estimate Q_{π_i} for certain subset of states

2. Policy improvement: updating policy π_i to π_{i+1} so that it behaves greedily with respect to the above updated return estimates Q_{π_i}

The learning algorithms utilized all work on the above principle, but have some significant variations in procedure and show different convergence properties.
Since the state space of the cart pole environment is a 4 dimensional continuous vector, our goal is to construct a function $\hat{Q}_\pi(s, a; \theta)$ with trainable parameters $\theta \in \Theta$ that is an estimate of expected return $Q_\pi(s, a)$ for a given state action pair under a policy π.

$$\hat{Q}_\pi : S \times A \times \Theta \mapsto \mathbb{R}$$

So, with function approximation, each iteration will now have:

1. Policy evaluation: updating $\hat{Q}_\pi(s, a; \theta)$ for certain states action pair(s), by tuning the parameter θ.

2. Policy improvement: updating policy π to behave greedily with respect to the above updated return estimates.

We employed a variety of learning algorithms:

1. SARSA
2. Deep Q Learning
3. Double Deep Q Learning
4. Prioritized Experience Replay
5. A2C

We will now look at each of the above algorithms in greater detail.

3.1 SARSA

SARSA [7] stands for *State s, Action a, Reward r, State s′ and Action a′*, for describing this assignment at each state transition

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma Q(s',a') - Q(s,a))$$

where alpha is called step size. The algorithm can basically be seen as updating the mean value of return function for a state value pair. SARSA is an on policy algorithm, meaning that it updates Q values and policy at the same time as it's gathering experiences. Since we are using function approximation, we need a gradient for our parameter θ to update the Q values. Thus, we take mean squared error between current $Q(s,a)$ and $r + \gamma Q(s',a')$ as our loss and perform gradient descent. So, the algorithm can be modified to be:

Algorithm 1: SARSA with value function approximation

```
initialize model $Q(\theta)$ with arbitrary parameters $\theta$;
for each episode do
    initialise state $s$;
    choose $a$ from S using policy derived from Q (e.g., $\epsilon$ greedy) ;
    for each step in episode do
        take action $a$, observe $r$, $s'$;
        choose $a'$ from $s'$ using policy derived from Q (e.g., $\epsilon$ greedy);
        initialise loss $L \leftarrow (r + \gamma Q(s',a';\theta) - Q(s,a;\theta))^2$;
        tune parameter $\theta$ to minimise $L$;
    end
end
```

The ϵ greedy policy is defined to be

$$\pi(s) = \begin{cases} \textit{random } a & \textit{with probability } \epsilon \\ \operatorname{argmax}_a Q(s,a) & \textit{with probability } 1-\epsilon \end{cases}$$

This policy forces our model to be more exploratory, as always acting greedily might make the model miss some state action pairs that might be more rewarding.
We tried a few different models like Linear Regressors and Nueral Networks for estimating $Q(\theta)$.

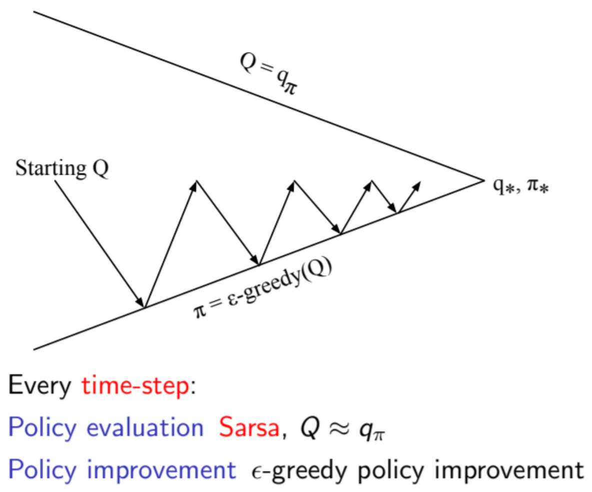

$Q = q_\pi$
Starting Q
q_*, π_*
$\pi = \varepsilon$-greedy(Q)
Every time-step:
Policy evaluation Sarsa, $Q \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

Figure 2: SARSA algorithm [9]

3.2 Deep Q Learning

Deep Q Learning [5] makes use of Neural Networks, which are the universal function approximators, for estimating the Q value function. This is an off policy algorithm, meaning that we update Q values based on past samples of state transitions seen by the agent, that are stored in a buffer. The algorithm utilizes a technique known as experience replay where it store the agent's experiences at each time-step (s, a, r, s') in a buffer D pooled over many episodes into a replay memory. It applies Q learning updates, or minibatch updates, to samples of experience, e, drawn at random from the pool of stored samples. For training, this algorithm makes use of the Bellman Equation and has a loss function defined below:

$$L(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \text{ where}$$
$$y_i = \begin{cases} R_T \text{ for terminal state } s_T \\ R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') \quad \text{ for non-terminal state } s_t \end{cases}$$

The advantages, as described in the article [5], are:

1. Each step of experience is potentially used in many weight updates, which allows for greater data efficiency

2. Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates

3. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters

Algorithm 2: Deep Q Learning with Experienced Replay

```
initialize the model $Q(\theta)$ with random weights, replay buffer $D$;
for each episode do
    initialise state $s$;
    for each step in episode do
        choose $a$ from $s$ using policy derived from Q (e.g., $\epsilon$ greedy);
        observe $r$ and $s'$;
        store $(s, a, r, s')$ in replay buffer $D$;
        sample batch $b = \{(s, a, r, s')\}$ from $D$;
        for each sample in $b$ do
            $Q^*(s, a) \leftarrow r + \gamma Q(s', \text{argmax}_{a'} Q(s', a'; \theta); \theta)$;
        end
        perform gradient descent step on $(Q^*(s, a) - Q(s, a; \theta))^2$ to tune parameter $\theta$;
    end
end
```

3.3 Double Deep Q Learning

This algorithm is an update on Deep Q Algorithm. The original Deep Q algorithm suffers from the problem of overestimations of Q values. These overestimations result from a positive bias that is introduced because Q learning uses the maximum action value as an approximation for the maximum expected action value[2]. To avoid this, the algorithm maintains two models, namely "target" and "primary" models. It uses the primary model for action selection and the target model for action evaluation. The weights of the second model θ are replaced with the weights of the target model θ' for the evaluation of the current greedy policy[12].

Algorithm 3: Double Deep Q Learning

```
initialize primary model $Q(\theta)$, target model $Q(\theta')$, replay buffer $D$;
for each episode do
    initialise state $s$;
    for each step in episode do
        choose $a$ from $s$ using policy derived from Q (e.g., $\epsilon$ greedy);
        observe $r$ and $s'$;
        store $(s, a, r, s')$ in replay buffer $D$;
        sample batch $b = \{(s, a, r, s')\}$ from $D$;
        for each sample in $b$ do
            $Q^*(s, a) \leftarrow r + \gamma Q(s', \text{argmax}_{a'} Q(s', a'; \theta); \theta')$;
        end
        perform gradient descent step on $(Q^*(s, a) - Q(s, a; \theta))^2$ to tune parameter $\theta$;
        update target model parameters $\theta' \leftarrow \theta$;
    end
end
```

3.4 Prioritized Experience Replay

PER [8] is built on-top of the Experienced Replay, with the difference being that instead of randomly drawing out samples from the memory buffer, here we take out samples having high priority more frequently. Priority for each sample is decided on the basis of the TD error, if the error is large then that sample is assigned more priority (probability for selection) because it has more information to train our model better.

$$p_t = |\delta_t| + e$$

Here, δ_t is the TD-error and e is a constant that makes sure no sample has 0 probability.

However, this might lead to overfitting with samples having high-priority always being selected. To prevent this, "stochastic prioritization" is performed wherein some randomness, α is added to the probability for every sample. If α=0, then pure uniform randomness, and if α=1, then it'll select the highest probability samples.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Since this priority sampling introduces certain bias to the model, "importance sampling" of weights is performed which reduces the weights of the frequently occurring samples.

$$\left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\underline{b}}$$

N is the size of replay buffer, $P(i)$ is the sampling probability and b controls the effect of importance sampling.

3.5 A2C

A2C is one of the benchmark Actor-Critic algorithms [6]. It's an off-policy Policy Gradient method having multiple synchronous agents working in parallel, with each agent having two models:

1. Critic estimates the value function (either action-value $Q(a, s)$ or state-value $v(s)$).
2. Actor updates the policy $\pi(a|s)$ as per the evaluation by critic.

However, vanilla Actor-Critic carries certain limitations to itself, one of them being an expensive computation cost. This is where parallel processing helps, working well enough even on CPUs, as justified in [4].

Policy gradient methods are better suited for continuous state-action spaces, which is not the case with the Cartpole environment. Hence, we did not explore more of such algorithms, apart from the A2C.

4 Results

Note that all models were trained and tested on a single machine for the sake of consistency and due to the limited computational resources available at our disposal.

4.1 SARSA vs DQN vs DDQN vs DDQN with PER vs Actor Critic

We tried the algorithms explained in the previous section for all the three tasks to see which one is more suitable for the given problem statement. Following criterion were used to evaluate the models: Episodes to solve, Mean Test Score Over 100 episodes and Model size.

Additionally, we also tried Policy-based Actor Critic (A2C) [3] as well as DDQN with Prioritized Experienced Replay [8], thereby exploring other State-of-the-Art methods available.

Algorithm	Episodes to solve			Mean test score over 100 episodes			Model Size
	Task 1	Task 2	Task 3	Task 1	Task 2	Task 3	
SARSA	599	2430	-	500	500	-	43 Kb
DQN	521	478	289	500	500	500	43 Kb
DDQN	269	456	398	500	500	500	43 Kb
A2C	Didn't converge within 4000 episodes						
DDQN PER	367	296	299	488.05	487.93	481.93	53 Kb

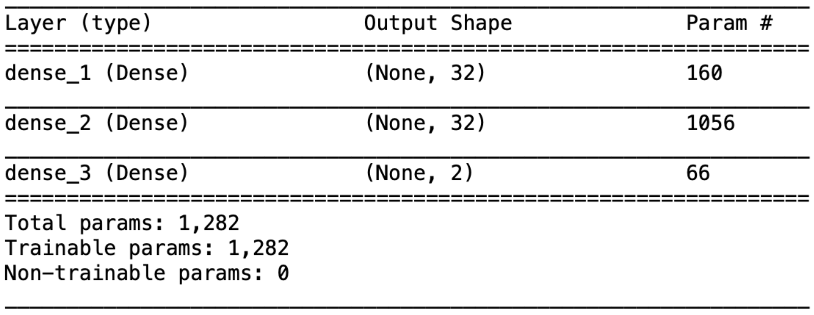
Table 3: Comparing various approaches. ("-" indicates that the environment couldn't be solved in 4000 episodes)

Looking at the table, we found DDQN to be the best suited for us, and therefore, we performed further hyperparameter tuning to the it. This included playing with the number of hidden layers (1, 2 or 3), number of neurons, learning rate (0.001, 0.0005, 0.0001, and 0.00005) as well the batch size (32 and 64).

Refer to the Appendix for detailed training curves on all these models.

4.2 Final DDQN model

Based on the various approaches tried and the hyperparameter tuning process, we found out that utilising a **Double Deep Q Network** gave the best results, for all the three tasks, an hence was robust to various noises in the environment. Refer to Figure 3 and Table 4 for the model parameters and architecture used.

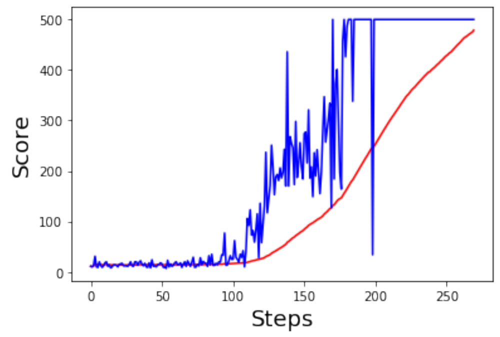


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                160
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,282
Trainable params: 1,282
Non-trainable params: 0
_________________________________________________________________
```

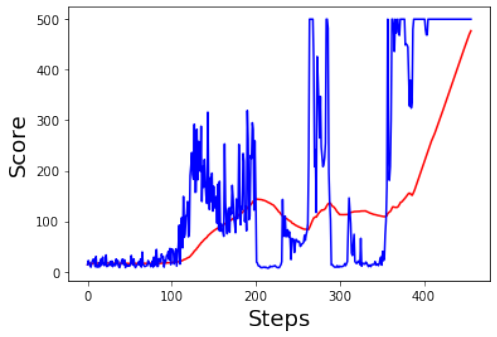
Figure 3: Final DDQN model summary

Parameter	Value
γ	1.0
Learning Rate	0.001
ϵ (max)	1.0
ϵ (min)	0.01
ϵ - decay	0.999
Batch Size	32
Replay Buffer Size	4000

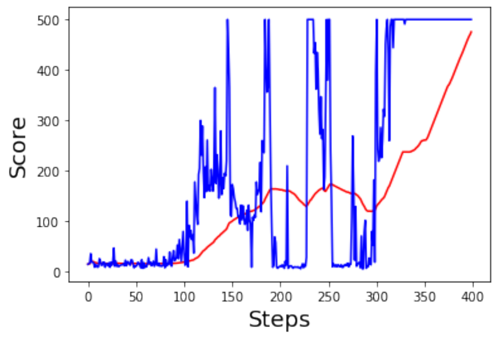
Table 4: Hyperparameters Parameters for the final DDQN model



(a) Task 1



(b) Task 2



(c) Task 3

Figure 4: **Training Curves for DDQN**. Red line indicates the progress of mean-over-100-trials until that episode, whereas Blue line indicates the total reward of that episode

4.3 Training Log

On using the same model architecture, we were able to obtain convergence to a mean training score of ≥ 475.0 over 100 consecutive episodes, in all three tasks.

Task 1 took 269 episodes, Task 2 took 456 episodes and Task 3 took 398 episodes to train. Refer to Figure 4 for complete logs.

Testing: Our saved models perform to give a perfect score of 500 over all the 100 episodes for all the three tasks. Model files and testing script have been attached to validate the same.

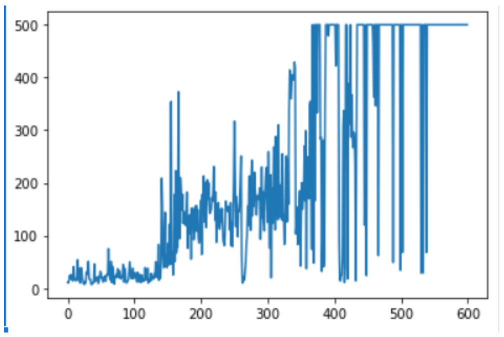
5 Contributions

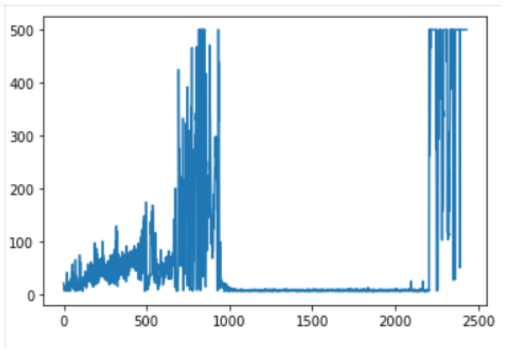
1. Anirudh: Introduction, Evaluating A2C, PER Model
2. Shubhad: Section 3, 3.1, 3.2, 3.3, 4, Appendix; Implementing SARSA, DQN, DDQN (together)
3. Ishita: Section 2, 3.4, 3.5, 4, Appendix; Implementing code for DDQN (together), PER, Hyperparameter tuning for DDQN, generating plots

6 Appendix

Hidden layers	BS	Optimizer	Best Training Eps	Mean Test Score
1 (32 units)	32	Adam	1186	500
2 (32, 32 units)	32	Adam	256	500
2 (32, 32 units)	64	Adam	264	500
1 (32, 32 units)	32	RMSProp	Didn't Converge	-
1 (64, 64 units)	32	Adam	Didn't Converge in 2000	-

Table 5: Performance with different Hyperparameters for the final DDQN model


(a) Task 1


(b) Task 2

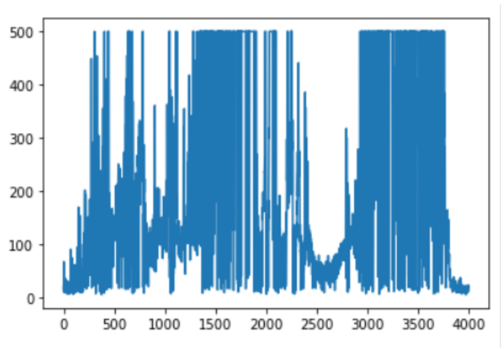

(c) Task 3

Figure 5: **Training Curves for SARSA**

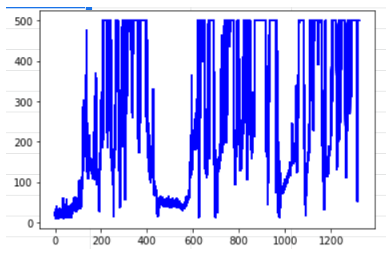
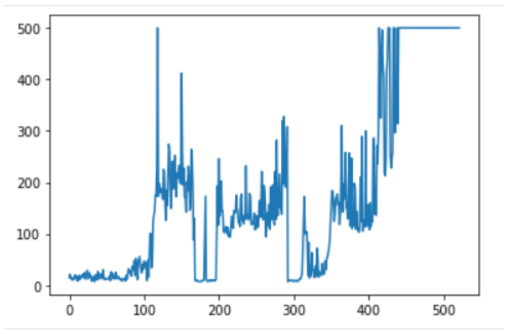
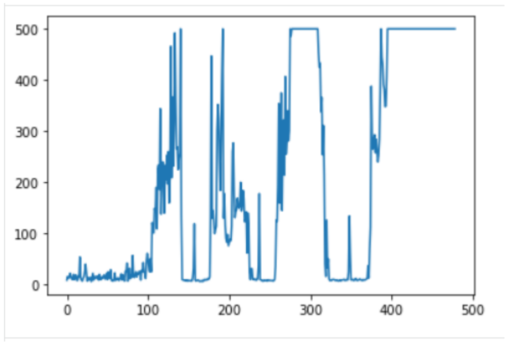


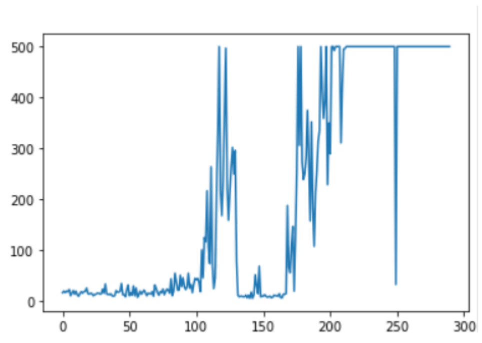
Figure 6: **Training Curves for Action Critic**



(a) Task 1

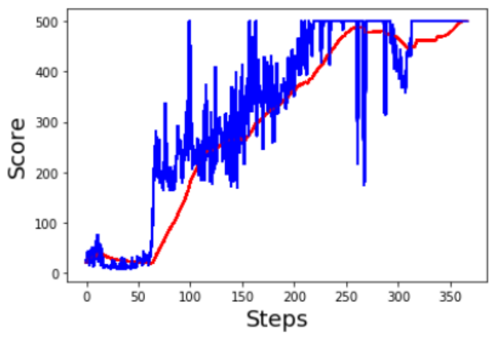


(b) Task 2

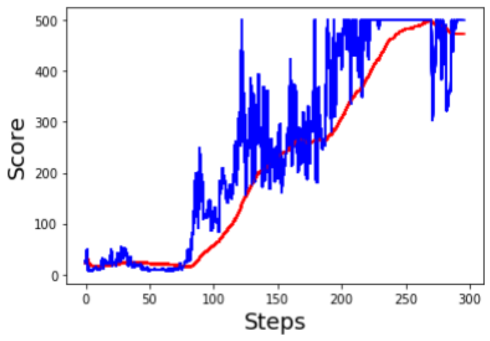


(c) Task 3

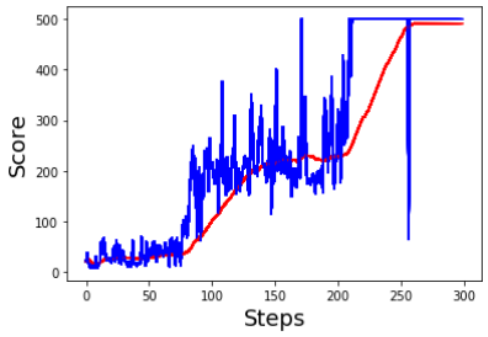
Figure 7: **Training Curves for DQN**



(a) Task 1



(b) Task 2



(c) Task 3

Figure 8: **Training Curves for PER**

References

[1] Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE transactions on systems, man, and cybernetics (5), 834–846 (1983)

[2] Hasselt, H.V.: Double q-learning. In: Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A. (eds.) Advances in Neural Information Processing Systems 23, pp. 2613–2621. Curran Associates, Inc. (2010), `http://papers.nips.cc/paper/3964-double-q-learning.pdf`

[3] Konda, V.R., Tsitsiklis, J.N.: Actor-critic algorithms. In: Advances in neural information processing systems. pp. 1008–1014 (2000)

[4] Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. pp. 1928–1937 (2016)

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing atari with deep reinforcement learning. CoRR **abs/1312.5602** (2013), `http://arxiv.org/abs/1312.5602`

[6] OpenAI: Advantage actor-critic (a2c), https://openai.com/blog/baselines-acktr-a2c/

[7] Rummery, G.A., Niranjan, M.: On-line Q-learning using connectionist systems, vol. 37. University of Cambridge, Department of Engineering Cambridge, UK (1994)

[8] Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. arXiv preprint arXiv:1511.05952 (2015)

[9] Silver, D.: David silver lecture 5 model free control, https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf

[10] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484 (2016)

[11] Sutton, R.S., Barto, A.G., et al.: Introduction to reinforcement learning, vol. 135. MIT press Cambridge (1998)

[12] Van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. In: Thirtieth AAAI conference on artificial intelligence (2016)