LUNAR LANDER CONTINUOUS WITH DDPG AND SAC

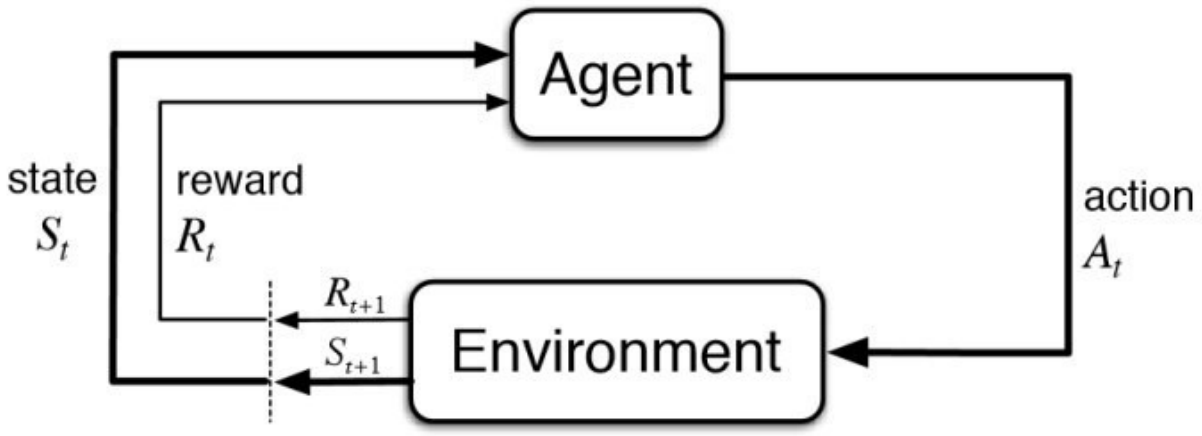
Debduti Sengupta
Student ID:251253766

Introduction

For this project I wanted to explore two specific areas of Reinforcement Learning, namely, environments with continuous action space and agents that are powerful enough to solve them. For this specific project, I chose to work with the environment '**Lunar Lander Continuous -version 2.0'** from Open AI gym. The algorithms that I explored for the task of solving this environment are **Deep Deterministic Policy Gradient (DDPG)** and **Soft Actor Critic (SAC).** In this report, I will give a brief background of continuous state space environments in general and Lunar Lander Continuous in particular. The following sections will explain the algorithm and architecture of DDPG and SAC, with reference to the papers they were published in.

Background

Reinforcement Learning (RL) is a subset of Machine Learning which involves software agents attempting to take actions or make moves in hopes of maximizing some prioritized reward. There are several different forms of feedback which may govern the methods of an RL system. Compared to Supervised Learning algorithms which map functions from input to output, RL algorithms typically do not involve the target outputs (only inputs are given). There are three elements of a basic RL algorithm: the agent (which can choose to commit to actions in its current state), the environment (responds to action and provides new input to agent), and the reward (incentive or cumulative mechanism returned by environment). The basic schema for a RL algorithm is given below:



Flowchart of an RL algorithm (Source: https://i.stack.imgur.com/eoeSq.png)

We define the various component of an RL system below:

- Action(A): All possible moves that the agent can take, for example, in grid, it could me moving east, west, north, or south

- State(S): Current snapshot of the environment
- Reward(R): An immediate return issued to the agent by the environment to evaluate the last action
- Policy (π): The plan of action or the strategy that the agent uses to choose its next action
- Value(V): The expected long-term return with discount applied for later states. $V_\pi(s)$ is the value of state s under policy π
- Q-value or action-value: The expected long-term return with discount applied for later states-action pairs. $Q_\pi(s, a)$ is the value of state s taking action a under policy π
  The optimal action-value function can be computed by solving the Bellman equation:
  $Q^*(s,a) = R(s,a) + \gamma \sum_{s'} \tau(s,a,s') max_{a'} Q^*(s',a')$

  The optimal policy can be defined as the greedy action in each state $\pi^*(a|s)$ = $1/|argmax_a Q^*(s,a)|$ if $a \in argmax_a Q^*(s,a)$, 0 otherwise.

The long-term goal of most RL system is to strike a balance between exploration and exploitation to maximise rewards and/or reach the goal and finish the task. This is known as Reward Hypothesis. By exploration, we mean training on new data points and exploitation means to use previously captured data. RL agents can learn either in a On Policy method or Off Policy method

On-Policy vs. Off-Policy

On-policy learning:

- Uses the deterministic outcomes or samples from the target policy to train the algorithm
- Has low sample efficiency
- Require new samples to be collected for nearly every update to the policy
- Becomes extremely expensive when the task is complex

Off-policy learning:

- Training on a distribution of transitions or episodes produced by a different behavior policy rather than that produced by the target policy
- Does not require full trajectories and can reuse any past episodes(experience replay) for much better sample efficiency
- Relatively straightforward for Q-learning based methods

Actor-Critic Algorithms

For this project we are using the environment **LunarLanderContinuous-v2**. Continuous tasks are where there is no clear terminal state, the agent must learn how to choose the best actions and simultaneously interact with the environment. With small, finite Markov Decision Processes (MDPs) that have a terminal state and are episodic, it is possible to represent the action-value

functions with a table, dictionary, or other finite structure. Small continuous spaces can be discretized, but for large continuous spaces, we need to use function approximation methods as it is computationally impractical to store the action-value function in a tabular form.

Besides having the capability of selecting any real-valued action, RL algorithms for continuous action problems should be able to efficiently find the greedy action, i.e., the action associated to the highest estimated value. Differently from the finite MDP case, a full search in a continuous action space to find the optimal action is often unfeasible. To overcome these difficulties, several approaches have adopted the actor- critic architecture [2, 3]. The key idea of actor-critic methods is to explicitly represent the policy (stored by the actor) with a memory structure independent of the one used for the value function (stored by the critic). In a given state, the policy followed by the agent is a probability distribution over the action space, usually represented by parametric functions (e.g., Gaussians [4], neural networks [5], fuzzy systems [6]). The role of the critic is, based on the estimated value function, to criticize the actions taken by the actor, which consequently modifies its policy through a stochastic or deterministic gradient on its parameter space. In this way, starting from a fully exploratory policy, the actor progressively changes its policy so that actions that yield higher utility values are more frequently selected, until the learning process converges to the optimal policy. By explicitly representing the policy, actor-critic approaches can efficiently implement the action selection step even in problems with continuous action spaces. In regular policy gradient methods with Baseline, learned state-value function estimates the value of only the first state of each state transition. This estimate sets a baseline for the subsequent return but is made prior to the transition's action and thus cannot be used to assess that action. In actor–critic methods, the state-value function is applied also to the second state of the transition. The estimated value of the second state, when discounted and added to the reward, constitutes the one-step return, $G_{t:t+1}$, which is a useful estimate of the actual return and thus is a way of assessing the action. When the state-value function is used to assess actions in this way it is called a critic, and the overall policy-gradient method is termed an actor–critic method.

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T}\nabla_\theta \underbrace{\log \pi_\theta(a_t|s_t)}_{\text{Actor}}\underbrace{(Q(s_t,a_t) - V_\phi(s_t))}_{\text{Critic}}$$

For this project, we are exploring DDPG and SAC algorithms. DDPG or Deep Deterministic Policy Gradient is an off-policy actor-critic deep RL algorithm that is an extension of the Policy Gradient RL algorithm and applies a Deep Q-Network to do random exploration with a soft updates strategy.

SAC or Soft Actor Critic is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework. In this framework, the actor aims to maximize

expected reward while also maximizing entropy. The goal is to succeed at the task while acting as randomly as possible. SAC combines off-policy updates with a stable stochastic actor-critic formulation[7].

The code has been based on the tutorial from the Youtube channel:
https://www.youtube.com/channel/UC58v9cLitc8VaCjrcKyAbrw

2. Method

2.1 Environment

In our project, we use the Lunar Lander Continuous environment from Open AI gym. The state space, action apace and rewards of this environment are as follows:

State Space

The state space consists of 8 values that are:

* x coordinate of the lander
* y coordinate of the lander
* v_x, the horizontal velocity
* v_y, the vertical velocity
* θ, the orientation in space
* v_θ, the angular velocity
* Left leg touching the ground (Boolean)
* Right leg touching the ground (Boolean)

Action Space

The action is a two values array from -1 to +1 for both dimensions. The first one controls the main engine, -1.0 is off, and from 0 to 1.0, the engine's power goes from 50% to 100% power.

The engine can't work with less than 50% of the power. The second value controls the left and right engines. From -1.0 to -0.5, it fires the left engine; from 0.5 to 1.0, it fires the right engine; from -0.5 to 0.5, the engines are off.

Rewards

The Reward for moving from the top of the screen to the landing pad and zero speed is from 100 to 140 points. If the lander moves away from the landing pad, it loses the reward. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing the main engine is -0.3 points for each frame.
The agent wins the game when the average reward per episode is 200 for the last 100 episodes.

2.2 DDPG (Deep Deterministic Policy Gradient)

Algorithm

The algorithm for DDPG is [8]

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

The DDPG algorithm, much like DQN(Deep Q Network), uses a Base Network to estimate the Q-values using the current state. This is the network that does all the 'learning'. The Target network on the other hand estimates the target Q value using the next state and updates the Q values. We construct two network classes, one for Actor and another for Critic. The architecture follows the original paper [8].

Replay Buffer

All the actions and observations that the agent has taken from the beginning, limited by the capacity of the memory, are stored. Then a batch of samples is randomly selected from this memory. This ensures that the batch is 'shuffled' and contains enough diversity from older and newer samples to allow the network to learn weights that generalize to all the scenarios that it will be required to handle.

Actor

The Actor neural network comprises of two fully connected layers, the dimensions of which are 400 and 300 respectively, each followed by a batch normalization layer. Each layer has ReLu activation function. The final layer has a tanh activation function. In case of Lunar lander, we have two discrete actions -1 and +1, so we do not need any multiplication with the action's upper bound in the final layer. The input is the state space, and the final output is the optimum policy or the probability of choosing an action. The actor, therefore, approximates the policy by using a policy gradient approach. The actor output is deterministic, meaning, given the same input (state), we will have the same output vector(action).

The Actor loss is the mean of the Critic network's value given the actions taken by the Actor network. We want to maximize this value (we want a higher Q value), so we use the minus sign to transform it into a loss function. From the paper, actor loss or policy gradient:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

Critic

The Critic neural network comprises of two fully connected layers, the dimensions of which are 400 and 300 respectively, each followed by a batch normalization layer. Each layer has ReLu as activation function. The fully connected layers are followed by another layer that takes all the actions as inputs. ADAM optimizer is used in the output layer. The final output is a single output that returns the state-action value. The base and target critics use this architecture but use different inputs, as described above. The base critic network has a learning rate beta. The input vector of actions comes from the actor network.

The loss between the target Q values and estimated Q values (Mean Squared Error loss) is backpropagated along the Base Critic Network to adjust the weights. The Target network is not used for learning, however, at a very gradual rate, determined by parameter tau, usually a very small number, the Base Network weights are copied over to the Target Network. This is a way to strike a balance between not chasing a moving target but still having high prediction accuracy. Critic loss, from the paper:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right]$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q).$$

While y_t is also dependent on θ^Q, this is typically ignored.

Exploration vs Exploitation

As we see above, the Actor network gives a deterministic output of action vectors so there is a possibility that exploration of other possible actions not chosen by the greedy algorithm will be sacrificed. To overcome this, some noise is introduced in the Actor networks. The original paper uses Ornstein-Uhlenbeck Noise for this purpose. From the paper:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

2.3 **SAC (Soft Actor Critic)**

SAC models the RL problem not just for the expected reward maximization but also the expected entropy at the same time. It is an off-policy algorithm that uses the actor-value-critic framework and gives a stochastic policy by maximising rewards as well as the entropy.

Algorithm The algorithm for SAC is as follows [7]

Algorithm 1 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ.
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

The SAC algorithm uses one Actor, one Value Network, one Target Value Network and two Critic Networks in its architecture. The Actor network takes the states and the environments as input and gives a stochastic policy as output. The Value network uses samples from the buffer to estimate the state values. The target Value network is soft updated with the weight of the base Value network. <> Two Critic networks are used to calculate the state action values, and the minimum of these outputs is used to compute the critic loss. Like DDPG, SAC also uses a replay buffer. The details are as follows:

Replay Buffer

All the actions and observations that the agent has taken from the beginning, limited by the capacity of the memory, are stored. Then a batch of samples is randomly selected from this memory. This ensures that the batch is 'shuffled' and contains enough diversity from older and newer samples to allow the network to learn weights that generalize to all the scenarios that it will be required to handle.

Actor Network

This is the policy network. The actor neural network comprises of two fully connected layers. The state space and the action space vectors are used as input to the network. The dimensions of the fully connected layers is 256(as per the original paper [7]). Each layer uses a ReLu activation function. It takes as input the state and actions and outputs the Mu(mean) and Sigma (standard deviation) values to create a normal distribution from which we sample our actions. Final activation function is Tanh. We do add a multiplication factor equal to the upper bound of action here, although for Lunar Lander it is not necessary because the upper bound is 1. Output is the action and log probabilities of choosing that action.

Actor(policy) loss can be calculated using the following equation(policy gradient)

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) + (\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t))\nabla_\phi f_\phi(\epsilon_t; \mathbf{s}_t), \tag{13}$$

Value Network

It takes the states as input and gives the state value as output. It comprises of two fully connected layers each with a dimension if 256. We use one base Value network and one Target Value network for an off-policy update of Values. The target is initialised with similar weights as that of the base, but after that it is very slowly updated with the base weights based on the update parameter tau.

Value loss is calculated as follows (value gradient) [7]

$$\hat{\nabla}_{\psi} J_V(\psi) = \nabla_{\psi} V_{\psi}(\mathbf{s}_t) \left(V_{\psi}(\mathbf{s}_t) - Q_{\theta}(\mathbf{s}_t, \mathbf{a}_t) + \log \pi_{\phi}(\mathbf{a}_t | \mathbf{s}_t) \right), \tag{6}$$

Critic Network

Our Critic Network has two fully connected layers that use ReLu as activation function. Each fully connected layer has dimension 256. We are using ADAM as the activation function in the output layer. The output of the critic network is the q value. The critic loss function is given below. The critic networks are trained to minimise the loss.

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_{\theta}(\mathbf{s}_t, \mathbf{a}_t) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right)^2 \right]$$

with

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\bar{\psi}}(\mathbf{s}_{t+1}) \right],$$

Where $[V_{\bar{\psi}}(\mathbf{s}_{t+1})]$ comes from the Target Value network.

Two critic networks are used and the min q values from the output of the two networks is chosen to calculate the policy gradient as well the value gradient. This is done to mitigate positive bias in the policy improvement step that is known to degrade performance of value-based methods [11].

Reward Scaling and Entropy Maximisation

The magnitude of reward scaling determines the degree of entropy maximisation. Entropy appears in policy and value functions. When entropy is not maximised, the stochasticity of the policy is decreased, thereby leading to less exploration. With the right reward scaling, the model balances exploration and exploitation, leading to faster learning and better asymptotic performance.

3. Experiments and Results:

DDPG:

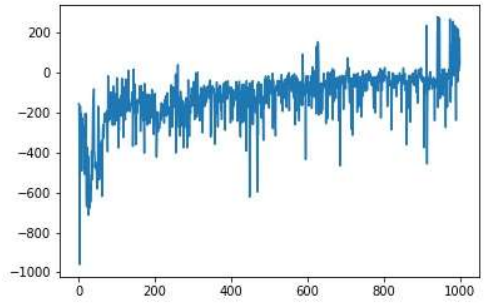
I ran the algorithm with default hyperparameters as per the paper. The agent reached 100 episode average of 200 points after about 2000 episodes of training. Model checkpoints were periodically saved to help resume training.

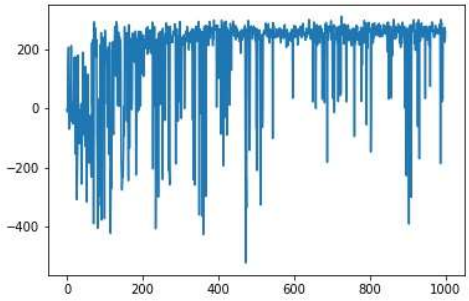
Scores from the last 10 episodes of training(1990 to2000):

Episode	Score	100 Episode Average
1990	290.11	201.36
1991	22.94	201.16
1992	55.56	201
1993	33.04	200.81
1994	268.79	200.88
1995	261.59	200.95
1996	272.56	201.03
1997	247.35	201.08
1998	274.05	201.17
1999	224.55	201.19
2000	259.2	201.26

Rewards per Episode:

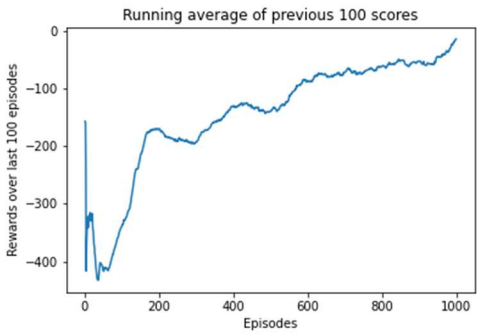
The left snapshot shows the per episode reward for the first 1000 episodes and the right snapshot is for the next 1000 episodes.

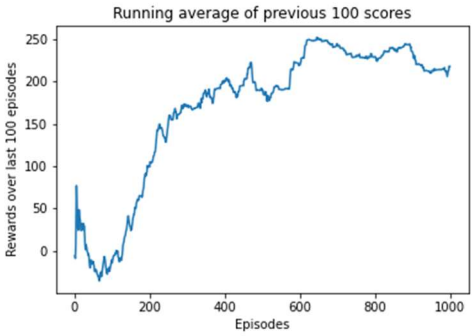




Hundred Episode Average Reward:

The left snapshot shows the 100 episode average reward for the first 1000 episodes and the right snapshot is for the next 1000 episodes.





The results show that there is a big dip in the reward in the earlier episodes , for about the first 100 episodes or so. During the first 100 episodes or so, the agent is not sampling from the replay buffer and not learning, which can explain the big up and down jumps. It starts to steadily increase the cumulative rewards from about 200 episodes onwards. However, it seems to be quite jittery after it hits the 100 episode of 200 points , and we can also see that in the per episode reward chart.

Performance wise, it is fairly time consuming, considering we are training two networks and the learning rate is also quite low. Training 2000 episodes took us about 3 hours.

SAC

This is a much more resource intensive algorithm and each episode is worth many more timesteps compared to DDPG. The Agent was trained for 3500 episodes, which took close to 10 hours of training before it could hit the 200 points per 100-episode average. All hyperparameters were chosen as mentioned in the main paper. I trained the Agent for a total of 3500 episodes, but it did not reach the winning score of a 100 episode average of 200.

Model checkpoints were periodically saved to help resume training.

The training was broken up into 3 chunks- 1000 episodes, 1000 episodes and then 1500 episodes.

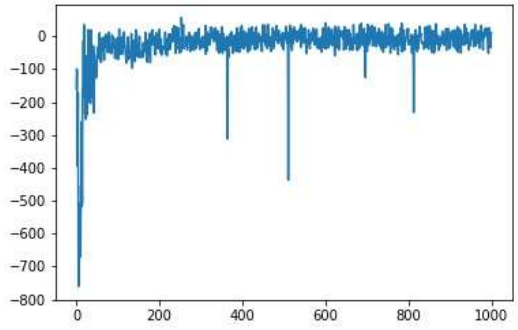
Scores from the Episodes 1990 to 2000:

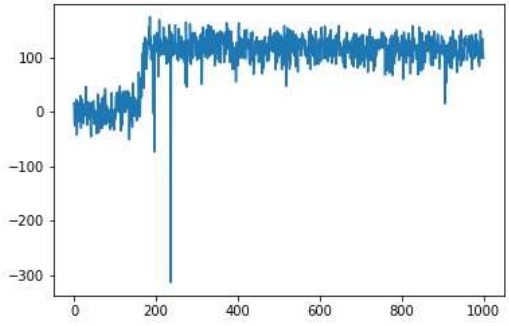
Episode	Score	100 Episode Average
1990	83.94	109.31
1991	117.49	109.32
1992	132.48	109.34
1993	149.4	109.39
1994	127.93	109.41

1995	109.74	109.41
1996	98.48	109.4
1997	134.72	109.42
1998	107.16	109.42
1999	108.67	109.42
2000	99.09	109.41

Rewards per Episode:

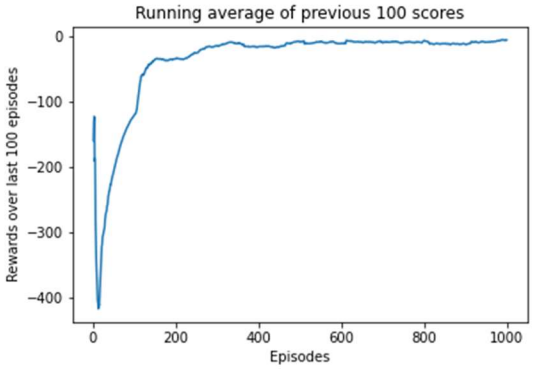
The left snapshot shows the per episode reward for the first 1000 episodes and the right snapshot is for the next 1000 episodes.

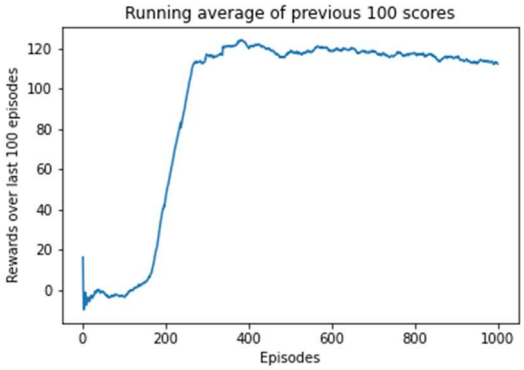




There is a higher degree of exploration in the earlier episodes, but then it gets fairly steady at around 0 reward points till the end of 1000 episodes. For the next 1000 episodes, the agent starts to explore a bit around episode 200, and quickly reaches a steady reward of around 100 henceforth.

Hundred Episode Average Reward:





Compared to DDPG, SAC is much more stable once it finds an optimal policy. Point to note though, even after 2000 episodes that took around 6 hours to train, the agent had not 'won' or reached a 100 episode of average of 200. We then trained the agent for an additional 1500 episodes.

Inference: SAC is much more resource intensive than DDPG, but it appears to be more stable as training time progresses. The difference in result probably isn't too striking in small and comparatively low complexity environments. It would be interesting to observe the difference in continuous spaces where even the actions are continuous.

4. Challenges and Learnings

These are some of the challenges and struggles that I faced

1. Selection of topic took me a long time. RL is a hard subject, mastering it takes very long, and I was not confident in picking a good topic.
2. Figuring out how to render non text-based environments in Google Colab notebooks. Admittedly, it was not crucial to the project, but I still wanted to cross that frontier because, well, it seemed liked a challenge. A lot of Internet searching and trial and error later, I was able to accomplish it. I have included it in the DDPG notebook.
3. I was having trouble choosing an environment which wasn't too trivial, due to challenges in working with Colab. I tried installing several environments, a few from Atari as well, but somehow always ended up with errors with some or the other dependency not being satisfied. In the end I went with Lunar Lander Continuous. For future work, I will work with a few different environments.

Some of the learnings I had

1. To have a vision and solid plan in place for projects
2. To read, understand and implement scholarly papers. It is still a challenge still, and I will need a lot of practice to feel comfortable, but it is a step in the right direction
3. I have a much better understanding of a few of the concepts of Deep Reinforcement Learning algorithms and approaches now.

Future work

1. To conduct more tests, with different hyperparameters.
2. To create videos of trained agents playing
3. To test on other environments.

Appendix

Colab Notebooks:

DDPG

https://colab.research.google.com/drive/12wE2J4eeqx09OEa-NBdgZRidToNhTocu#scrollTo=X1wGxemGhUrt

SAC

https://colab.research.google.com/drive/1mimgqpU5qY01hQx7II3P1Po8tao5eVLb#scrollTo=X1wGxemGhUrt

References

1. https://proceedings.neurips.cc/paper/2007/file/0f840be9b8db4d3fbd5ba2ce59211f55-Paper.pdf
2. V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. SIAM Journal on Control and Optimization, 42(4):1143–1166, 2003
3. Jan Peters and Stefen Schaal. Policy gradient methods for robotics. In Proceedings of the IEEE International Conference on Intelligent Robotics Systems (IROS), pages 2219–2225, 2006.
4. H. Kimura and S. Kobayashi. Reinforcement learning for continuous action using stochastic gradient ascent. In 5th Intl. Conf. on Intelligent Autonomous Systems, pages 288–295, 1998
5. Hado van Hasselt and Marco Wiering. Reinforcement learning in continuous action spaces. In 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, pages 272–279, 2007
6. L. Jouffe. Fuzzy inference system learning by reinforcement methods. IEEE Trans. on Systems, Man, and Cybernetics-PART C, 28(3):338–355, 1998.
7. Haarnoja.Tuomas et al. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.ICML,2018
8. Lillicrap.Timothy et al.CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING,Google Deepmind,2016
9. Antonio. Reinforcement Learning in Continuous Action Spaces: DDPG. 2016. https://antonai.blog/reinforcement-learning-in-continuous-action-spaces-part-1-ddpg/
10. https://spinningup.openai.com/en/latest/algorithms/sac.html#:~:text=Soft%20Actor%20Critic%20%28SAC%29%20is%20an%20algorithm%20that,bridge%20between%20stochastic%20policy%20optimization%20and%20DDPG-style%20approaches.
11. Hasselt, H. V. Double Q-learning. In Advances in Neural Information Processing Systems (NIPS), pp. 2613–2621,2010.
12. https://www.youtube.com/channel/UC58v9cLitc8VaCjrcKyAbrw