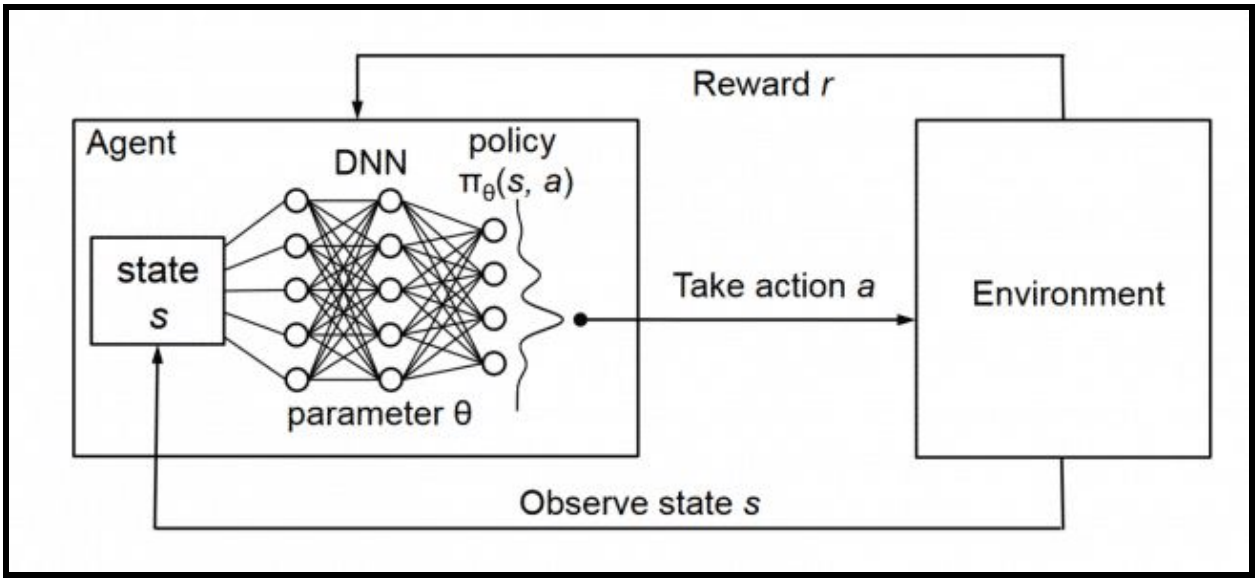
University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

CSE 700TUT Independent Study
Prof. Changyou Chen

Presented by: Vishva Nitin Patel (vishvani), Yash Nitin Mantri (yashniti)

Deep Reinforcement Learning: Analysis of algorithms on discrete and continuous action-spaces



Introduction:

This project aims at developing a hands-on understanding of various deep reinforcement learning (DRL) algorithms by developing them from scratch and exploring their performance on continuous action-spaces while setting the comparison baseline with respect to some well known discrete (non-continuous) action-spaces.

The goal was to understand the underlying implementation logic for all of these DRL algorithms and their performance in diverse environments (action-spaces) provided by OpenAI.

The work-flow can be outlined as follows:

Section - 1 - Understanding the DRL algorithms explored:

1. Deep Q - Network (DQN)
2. Dueling Double DQN (DDQN)
3. Proximal Policy Optimization (PPO)
4. Deep Deterministic Policy Gradient (DDPG) algorithms.

Section - 2 - Analyzing DRL algorithms on discrete action spaces:

1. DQN, DDQN and PPO on LunarLander-v2
2. DQN and PPO on PongDeterministic-v4

Section - 3 - Analyzing DRL algorithms on continuous action spaces:

1. PPO on MountainCarContinuous-v0
2. DDPG on LunarLanderContinuous-v2

Section - 1 - Understanding the DRL algorithms explored:

1. Deep Q - Network (DQN)

DQN by DeepMind was the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. Basically the model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards.

The reason why a convolution neural network (CNN) was chosen in the design of DQN is that CNN can overcome various challenges in DRL such as - learning from a scalar reward signal that is frequently sparse, noisy and delayed, the delay between actions and resulting rewards and large sequences of correlated states. And for DQN, the CNN is trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights.

By feeding sufficient data into deep neural networks, it is often possible to learn better representations than handcrafted features. The goal is to connect a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates.

DQN starts by utilizing a technique known as experience replay where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set D = e1...eN, pooled over many episodes into a replay memory. During the inner loop of the algorithm, we apply Q-learning updates, or mini batch updates, to samples of experience, e ~ D, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an -greedy policy. Since using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on fixed length representation of histories produced by a function "∅".

This approach has several advantages over standard online Q-learning. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Second, 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. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on.

The full algorithm is presented in Algorithm 1 below:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

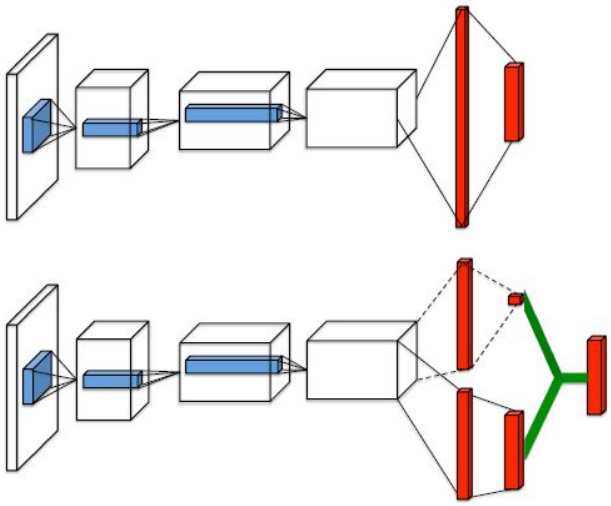
By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

2. Dueling Double DQN (DDQN)

DDQN is a newer neural network architecture (after DQN) for model-free reinforcement learning. The dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. This architecture leads to better policy evaluation in the presence of many similar-valued actions.

DQN is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values. DDQN reduces overestimations by decomposing the max operation in the target into action selection and action evaluation.

The proposed network architecture i.e. the dueling architecture, explicitly separates the representation of state values and (state-dependent) action advantages. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q as shown in Figure below:



A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action

Both networks output Q-values for each action.

This dueling network should be understood as a single Q network with two streams that replaces the popular single-stream Q network in existing algorithms such as DQN. The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision.

Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

The key point is that dueling architecture can more quickly identify the correct action during policy evaluation as redundant or similar actions are added to the learning problem.

Prioritized Replay built on top of DDQN, further improved the state-of-the-art; the key idea behind prioritized replay was to increase the replay probability of experience tuples that have a high expected learning progress (as measured via the proxy of absolute TD-error).

With every update of the Q values in the dueling architecture, the value stream V is updated – this contrasts with the updates in a single-stream architecture where only the value of one action is updated, the values for all other actions remain untouched. This more frequent updating of the value stream in our approach allocates more resources to V , and thus allows for better approximation of the state values, which need to be accurate for temporal-difference-based methods like Q-learning to work.

3. Proximal Policy Optimization (PPO)

PPO alternates between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent, whereas standard policy gradient methods perform one gradient update per data sample. The need for developing a method that is scalable (to large models and parallel implementations), data efficient, and robust (i.e., successful on a variety of problems without hyperparameter tuning) led to the development of PPO.

PPO is on-Policy (unlike DQN & Q Learning) i.e. it doesn't use a replay buffer to store experiences, rather it learns directly from whatever its agent encounters directly in the environment. Once a batch of experience has been used to do a gradient update the experience is then discarded and policy moves on.

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Advantage Function tells us whether the action that our agent took was better than what we expected.

$$Q_\pi(s_t,a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\ldots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l})\right]$$

Discounted sum of rewards (Return) gives us a weighted sum of all the rewards the agent got during each time-step in the current episode. The discount factor γ is the present value of future rewards.

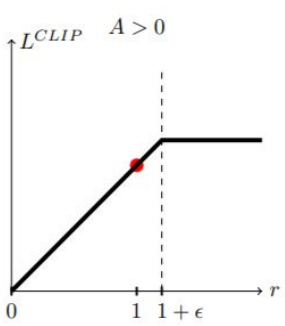
$$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},\ldots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l})\right]$$

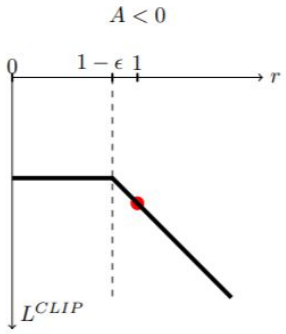
Value Function gives an estimate of discounted sum of rewards from the current state.

PPO deals with clipped probability ratios, which forms a pessimistic estimate (i.e., lower bound) of the performance of the policy. To optimize policies, PPO alternates between sampling data from the policy and performing several epochs of optimization on the sampled data.

Clipping has different graphs based on Advantage function value:

If action taken is good then it is more likely to happen in new policy compared to the old policy. In this case, we do not want to overdo action updates. Therefore, the objective function gets clipped here to limit the effect of gradient update.





If action taken was bad then it is less likely to happen in new policy compared but we do not want to reduce the likelihood of that action happening too much.

Algorithm PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

4. Deep Deterministic Policy Gradient (DDPG)

DDPG was implemented by DeepMind to provide continuous control. DDPG is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces.

While DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. DQN cannot be straightforwardly applied to continuous domains since it relies on finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

DDPG, as the name suggests is dependent on Deep Policy Gradient algorithm, and it combines actor-critic approach with insights from DQN. DQN is able to learn value functions using such function approximators in a stable and robust way due to two innovations: 1. the network is trained off-policy with samples from a replay buffer to minimize correlations between samples; 2. the network is trained with a target Q network to give consistent targets during temporal difference backups. DDPGuses the same ideas, along with batch normalization to achieve continuous control.

As with most reinforcement learning algorithms, the use of non-linear function approximators nullifies any convergence guarantees and such approximators appear essential in order to learn and generalize on large state spaces. DDPG combines DPG's actor critic algorithmic model with DQN's replay buffer to achieve robustness. A major challenge of learning in continuous action spaces is exploration. An advantage of off-policy algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. The approach followed by DDPG is described below:

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:

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

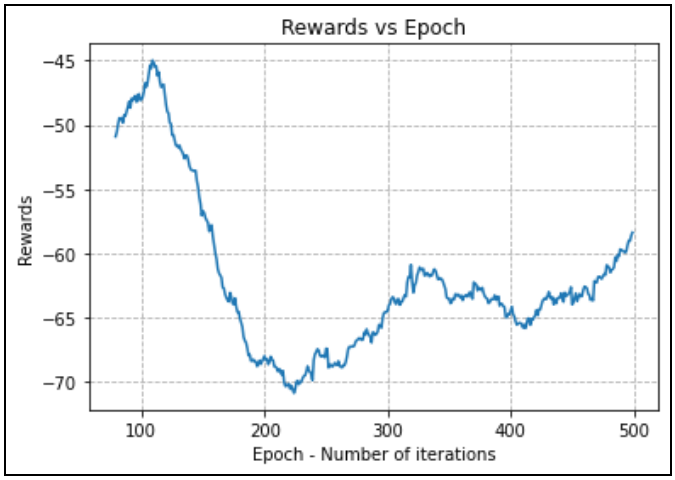
Section - 2 - Analyzing DRL algorithms on discrete action spaces:

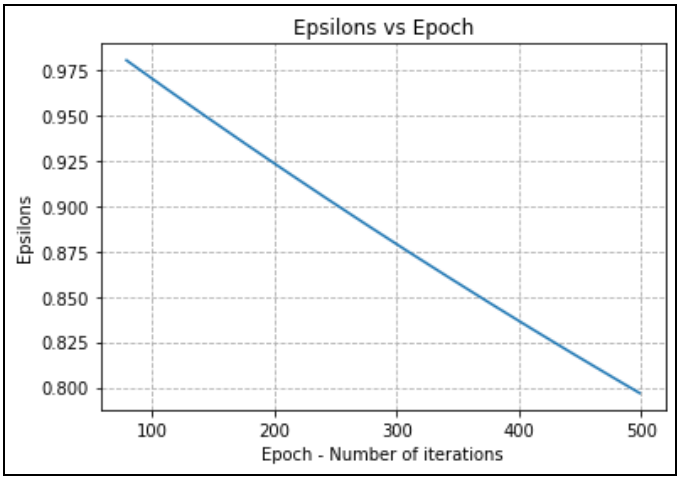
1. DQN, DDQN and PPO on LunarLander-v2

LunarLanderContinuous-v2: Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in the state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If the lander moves away from the 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 = 200 points. Landing outside the 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.

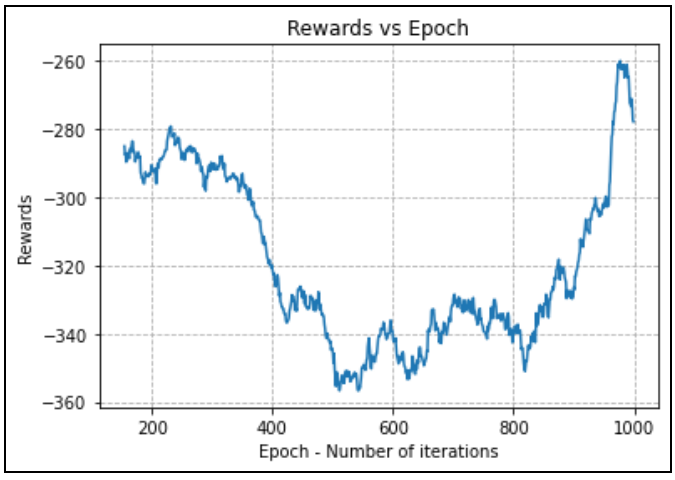
Applying DQN, DDQN and PPO on LunarLanderContinuous-v2, we observe the following results:

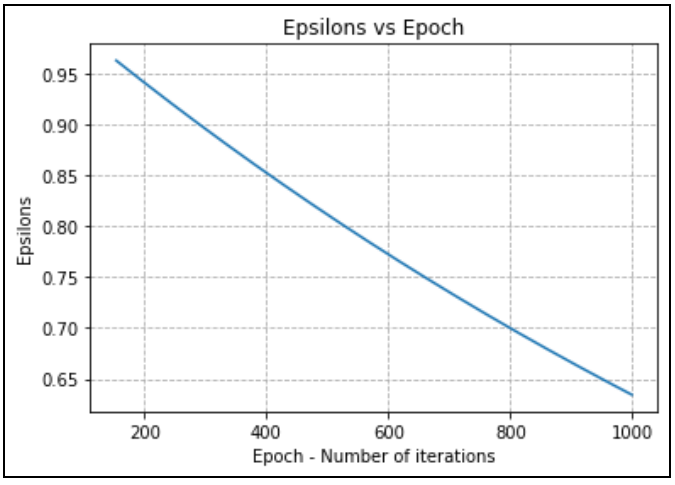
DQN gives the following rewards over iterations and epsilon values over iterations:

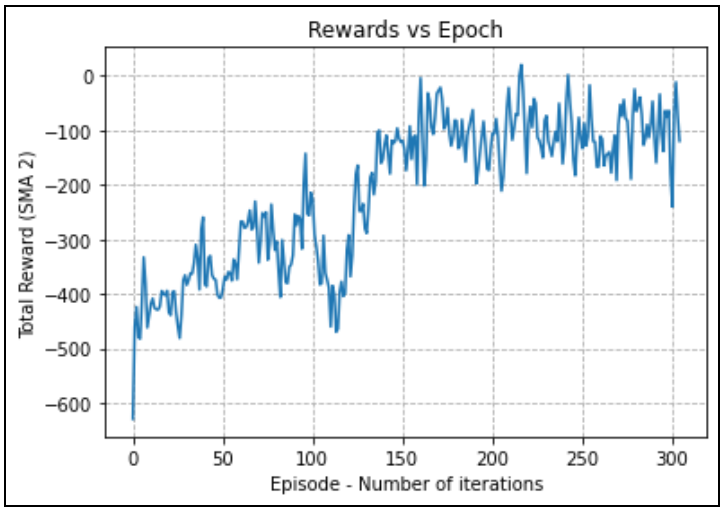




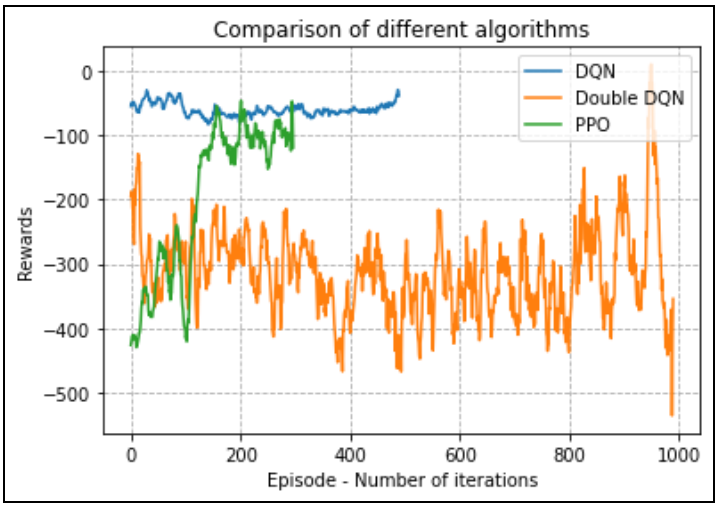
DDQN gives the following rewards over iterations and epsilon values over iterations:







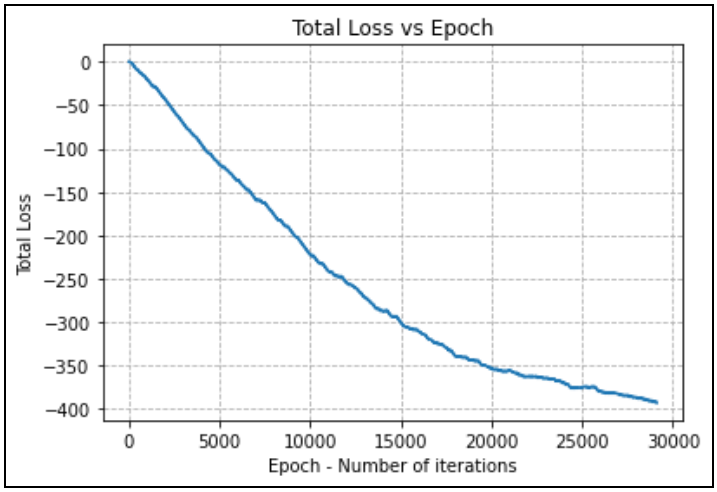
PPO's rewards over iterations



Comparison of all algorithm's rewards

2. DQN, PPO on PongDeterministic-v4

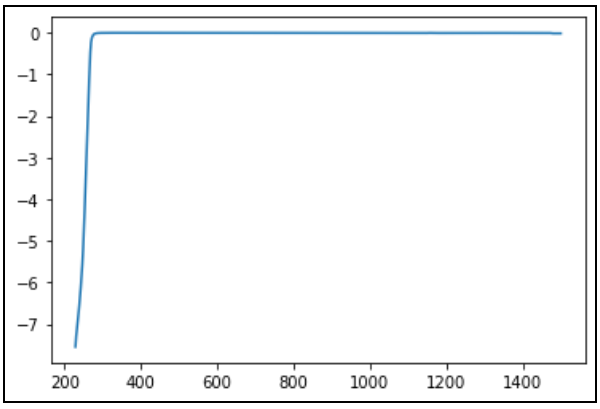
PongDeterministic-v4: In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3) Each action is repeatedly performed for a duration of k frames. In OpenAI Gym, each game has a few variants, distinguished by their suffixes. Through these variants, you can configure frame skipping and sticky actions. Frame skipping is a technique of using k-th frames. In other words, the agent only makes action every k frames, and the same action is performed for k frames. Sticky actions is a technique of setting some nonzero probability p of action being repeated without the agent's control. This adds stochasticity to the deterministic Atari 2600 environments. There are six variants for the Pong environment. In PongDeterministic-v4 - Frame Skip, k = 4^2, and Repeat action probability, p = 0.



PPO (on left) and DQN (on right) - Loss over iterations graph for PongDeterministic-v4

Section - 3 - Analyzing DRL algorithms on continuous action spaces:

1. PPO on MountainCarContinuous-v0

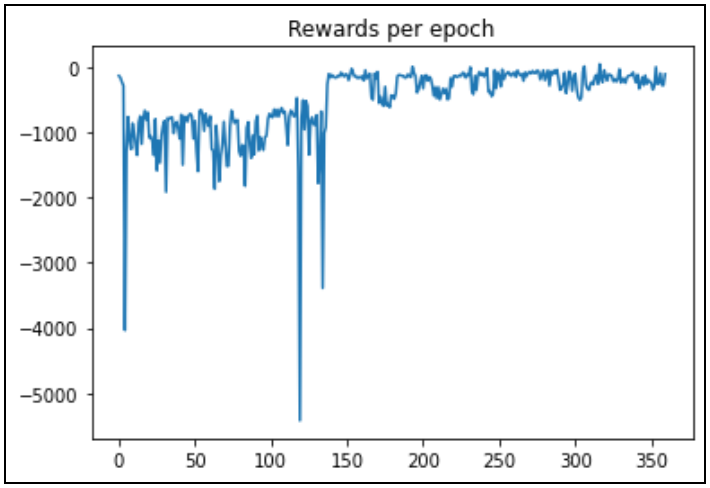


MountainCarContinuous-v0:

A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Here, the reward is greater if you spend less energy to reach the goal.

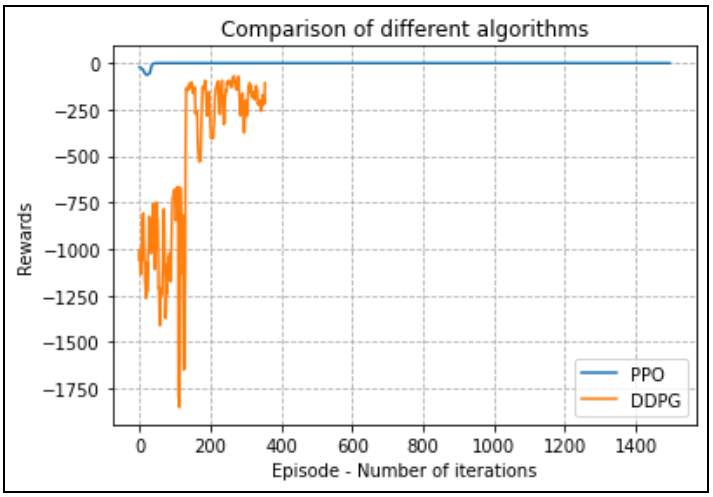
The adjoining graph shows the rewards over iteration for PPO on MountainCarContinuous-v0

2. DDPG on LunarLanderContinuous-v2



LunarLanderContinuous-v2: Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in the state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If the lander moves away from the 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 = 200 points. Landing outside the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real value vectors from -1 to +1. First controls the main engine, -1..0 off, 0..+1 throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value -1.0..-0.5 fire left engine, +0.5..+1.0 fire right engine, -0.5..0.5 off.

The comparison between PPO and DDPG over LunarLanderContinuous-v2 can be shown by the following rewards per iteration graph:



References:

1. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
2. http://cs231n.stanford.edu/reports/2017/pdfs/616.pdf
3. http://proceedings.mlr.press/v48/wangf16.pdf
4. https://arxiv.org/pdf/1707.06347.pdf
5. https://arxiv.org/pdf/1509.02971.pdf
6. http://www.eurecom.fr/~berthet/files/drl.pdf
7. https://www.novatec-gmbh.de/en/blog/deep-q-networks/
8. http://ceur-ws.org/Vol-2540/FAIR2019_paper_47.pdf
9. https://gym.openai.com/envs/LunarLanderContinuous-v2/
10. https://www.endtoend.ai/envs/gym/atari/