Analyzing different algorithms on OpenAI Gym environment

Team Members:
Kshitija Patel (kap676)
Siddharth Choudhary (sc7530)

Github Link: https://github.com/siddharthchd/openai-gym

1 Abstract

Compare and analyze deep reinforcement and evolutionary learning algorithms with the help of the OpenAI Gym Environments, and find out whether for a given environment, one algorithm consistently outperforms the other and if there exists some optimal modeling strategy.
Algorithms used : DQN, NEAT, DDPG
Gym Environments used: Bi-pedal Walker, Lunar Lander

2 Introduction

The OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms on different predefined environments or by creating our own new environments. For the purpose of our project we use the Box2D environments Bipedal Walker and the Lunar Lander.

The Box2D library is a 2D rigid body simulation library for games and 2D simulations which is used to make objects in a simulation or a game move more realistically and the environment more interactive. The Bipedal Walker and Lunar Lander are two such Gym environments based on the python implementation of the Box2D library. The objective of the Bipedal Walker environment is to train the agent to walk up till the very far right end of the map totalling up to 300+ points where hull angle, speed, angular velocity, horizontal and vertical speeds, position, speed and angles of the joints etc. are the input parameters to the algorithm observed at each state. The Lunar Lander environment, on the other hand is a little less complex with the objective to land at the landing spot without crashing into it or going off the map, totalling up to a reward of 200+ points. Unlike Bipedal Walker, the Lunar Lander environment only has left, right and downward throttle values which maybe discrete or continuous depending on the environment.

3 Literature Survey

Reinforcement learning and Evolutionary learning strategies are two major approaches that are widely used to solve complex control problems involving agents interacting with environments. There has been growing research into combining the two approaches with deep learning techniques while at the same time using evolutionary approaches as an alternative to deep reinforcement learning algorithms since evolutionary strategies are easier to implement, scale in a distributed setting, and have fewer parameters compared to the former. Reinforcement learning algorithms define a policy function which takes in the current state of the environment as input and output probability distribution or a discrete action chosen from the action space, depending whether the environment is discrete or continuous. Evolutionary strategies on the other hand follow a 'guess and check' process where starting with some random parameters, each guess is tweaked at each step and moved slightly closer to which update worked best. These strategies mimic the natural principles of reproduction, crossover and mutation.

Current researches, which focus on comparison and analysis of different reinforcement learning algorithms and evolutionary strategies such Ref.[4] and Ref.[5], have used Box2D environments such as the Bipedal Walker and Lunar Lander and MujoCo models since these environments present with an environment as close to the real world as possible in 2D and 3D. These papers used policy optimization and actor critic network based

deep reinforcement learning algorithms like Continuous Asynchronous Advantage Actor-Critic (CA3C), Parallelized Proximal Policy Optimization (P3O), Deep Q-Network (DQN) and Deep Deterministic Policy Gradient (DDPG), along with evolutionary strategies like Neuro-Evolution of Augmented Topologies (NEAT), Natural Evolution Strategy (NES) and Covariance Matrix Adaptation Evolution Strategy (CMAES). Past research into the performance of both these techniques on various Gym environments suggests that the performance of each algorithm is highly task dependent and that a lot of trade-offs exist between the two.

It has been observed that evolutionary strategies, although perform well on Box2D environments, have very high time complexity and can fail at very simple tasks such as the Pendulum whereas, deep reinforcement learning methods worked well with Box2D tasks which suggests that evolutionary strategies are better at exploration while deep reinforcement learning methods handle rich dynamics better. Even though evolutionary strategies are more time complex and require more cores to train faster, they are advantageous to use over RL techniques since they are highly parallelizable and only require a forward pass of the policy, eliminating backpropagation completely, saving time and memory. ES also have lesser hyperparameters compared to RL algorithms and are better at consistent exploration due to the epsilon-greedy policy.

4 Technical details

4.1 Bipedal Walker

The 'Bipedal-Walker-v3' environment is an environment based on the python py-Box2D library which has been designed in order for us to get the 2D bipedal walker to walk through the uneven terrain before it. A total of 24 inputs are used to train the bipedal walker agent, they are:

Index	Observation	Min	Max	Mean
0	hull_angle	0	2 * pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	-inf	-
7	knee_joint_1_speed	-inf	-inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	-inf	-
12	knee_joint_2_speed	-inf	-inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar reading	-inf	+inf	-

Figure 1: Parameters defining state of environment for Bipedal Walker

As, intuitively, humans walk when we apply some torque to rotate our hip and knee joints in order to place one leg in front of the other, similarly, our agent also outputs 4 such values which determine the torque control and velocity control for each of the hip and knee joints as given below:

Index	Action	Min	Max
0	Hip_1 (Torque / Velocity)	-1	+1
1	Knee_1 (Torque / Velocity)	-1	+1
2	Hip_2 (Torque / Velocity)	-1	+1
3	Knee_2 (Torque / Velocity)	-1	+1

Figure 2: Torque and Velocity Controls for hip and knee joints for Bipedal Walker

On generating the state action pairs at each episode, the agent is rewarded for moving forward until it reaches the far end of the environment map which sums up to a total of 300+ points. Each state consists of hull angle and speed, angular velocity of hull, horizontal speed, vertical speed, position of the joints and their angular velocity, contact of legs with the ground and 10 lidar measurements for rangefinding. In order for the agent to learn to walk optimally, the agent is also penalized a small amount for applying motor torque.

Every new episode starts with the agent dropping at a random position on the environment, a little above the ground, with its position mostly being upright with straight legs. An episode is terminated in case of three following scenarios:

- When the agent's body (hull) touches the ground
- When the agent is stuck in a given position (like doing a 'acrobatic split') and no change in position is observed over the given number of time steps
- When the agent solves the environment, in which case the agent falls off the land, where the environment map terminates at the far right edge.

The Bipedal Walker environment, as defined by OpenAI, is considered solved when an average reward of 300 or more is observed over 100 consecutive trials.

4.2 Lunar Lander

The task defined in the 'LunarLander-v2' environment is a very simple and classic problem of landing a space-ship between two flags on the Moon's surface smoothly, without crashing into it. The agent or the space-ship consists of 3 throttles or thrusters facing the left, right, and the downwards direction which help move the agent in the opposite direction, i.e. firing the downwards facing thrusters will push the agent in the upwards direction. The landing pad, which is the area between the two flags designated as the landing position, is always fixed at the coordinates (0, 0) and are the first two values in the state vector.

The agent is rewarded for moving from the top of the screen in the environment to the landing pad, achieving zero speed when landing, and the value for this reward is usually between 100 and 140. A penalty is also given to the agent in case the downward throttle is used too much and the agent ends up moving further away from the landing pad. On landing or crashing , additional +100 or -100 are rewarded or penalized respectively, based on the state. Just like in the bipedal walker, for our lunar lander agent to optimally learn how to land, the agent is also penalized with -0.3 points for every frame that the agent fires the main engine whereas a +10 reward is given for each successful contact of a space-ship leg with the ground. Each state in this environment has 8 components, which are horizontal and vertical position, horizontal and vertical velocity, angle and angular velocity, and left and right leg contact, thus there are 8 inputs to the network or algorithm that will train this agent.

The Lunar Lander environment consists of two different environments, one with a discrete action space and the other with a continuous action space.In case of the discrete Lunar Lander environment, the action space is defined as follows :

In case of the continuous case of the Lunar Lander environment, the action space is given by real valued vectors and are defined as:

Action Space (discrete)	Action Taken
0	Do nothing
1	Fire left engine
2	Fire down engine
3	Fire right engine

Figure 3: Action Space for Discrete Lunar Lander

Action Space (continuous)	Min	Max
Main Engine (off)	-1	0
Main Engine (throttle)	0	+1 (>= 0.5 to throttle)
Fire Left Engine	-1.0	-0.5
Fire Right Engine	+0.5	+1.0
Off	-0.5	0.5

Figure 4: Action Space for Continuous Lunar Lander

For both the discrete and continuous environments, the fuel provided to the spaceship is infinite so that an agent can first learn to fly and then land amongst its first few attempts at the landing spot. Every new episode starts with the agent dropping at a random position on top of the environment screen, free-falling towards the ground till a thruster is deployed. An episode is terminated in case of two following scenarios:

- When the agent's body (spaceship) touches the ground instead of the legs
- When the agent throttles too hard and ends up exiting the environment screen

The Lunar Lander environment, as defined by OpenAI, is considered solved when an average reward of 200 or more is observed over 100 consecutive trials.

5 Algorithm: DQN

The Deep Q-Network algorithm is developed by enhancing the classic Q-Learning reinforcement learning algorithm using deep neural networks and experience replay. The Q-Learning algorithm uses the state-action value function (trajectory) known as the Q-function of a policy π as $Q^{\pi}(s, a)$ measures the expected return or discounted sum of rewards that is obtained from state s by taking action a and following the policy π. An optimal Q-function obeys the Bellman optimality equation given as :

$$Q^*(s, a) = E[r + \gamma max_{a'} Q^*(s', a')]$$

The expectation value is computed over the distribution of original rewards as well as the possible next states which will be observed. This optimality equation is used as an iterative update till it converges to an optimal Q-function.

Since Q-function is represented as a table consisting of values for all combinations of state action pairs, it is practically not possible to maintain an exponentially large number of parameters in case of real world problems in a table-like format. Using deep neural networks, a function approximator is trained with parameters θ to estimate the Q-values by minimizing the loss function defined as :

$$L_i(\theta_i) = e_{s,a,r,s' \ \eta(.)}[(y_i - Q(s, a; \theta_i))^2]||where, y_i = r + \gamma max_{a'} Q(s', a'; \theta_{i-1})$$

Known as the temporal difference. η represents the distribution over transitions s, a, r, s' observed and obtained from the environment. DQN uses an epsilon-greedy policy $a = max_a Q(s, a; \theta)$ which chooses the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ to evenly cover the state action space.

5.1 Hyper-parameters

Hyper-parameter	Variable	Values
Number of Episodes	episodes	10000
Learning Rate	lr	0.001
Gamma	gamma	0.98
Epsilon	self.epsilon	0.7
Minimum Epsilon	self.epsilon_	0.01
Weight Decay	self.decay	0.995

Figure 5: Hyper Parameters for Bipedal Walker

Hyper-parameter	Variable	Values
Batch size	self.batch_size	64
Number of Episodes	episodes	1000
Learning Rate	self.lr	0.001
Gamma	self.gamma	0.99
Epsilon	self.epsilon	1.0
Minimum Epsilon	self.epsilon_	0.01
Epsilon Decay	self.epsilon_decay	0.996

Figure 6: Hyper Parameters for Lunar Lander

6 Algorithm: NEAT

In case of evolutionary methods and genetic algorithms, the genotype corresponds to the set of genes representing the chromosome whereas the phenotype corresponds to actual physical representation of the chromosome. NeuroEvolution of Augmenting Topologies is a kind of evolutionary algorithm, designed to evolve and create neural networks, which maintains a population of individual genomes which refer to the set of genes that together code a neural network phenotype. Two sets of genes are used to describe how to build the network, the node genes specifying single neurons and connection genes specifying single connections between two neurons.

The algorithm begins with a feed forward network of input and output neurons and evolves through each time step, increasing or decreasing the model complexity by either inserting a removing a neuron or connection path respectively. The fitness function computes a single real valued score which indicated the quality of individual genome, progressing through each generation through reproduction and mutation of the most fit individuals from previous generations

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3.\overline{W}$$

$$f_i^{'} = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}$$

The first equation deals with distance. **Delta**, or distance, is the measure of how **compatible** two genomes are with each other. c1, c2 and c3 are parameters set to dictate the importance of E, D and W. E represents the total number of **excess** genes. D represents the total number of **disjoint** genes in both genomes, W represents the total **weight difference** between genes that match, and N represents the number of connections / genes the genome with the larger number of connections has.
The second formula is used for adjusting fitness. f_i is the adjusted fitness, which will replace the original fitness, f_i. For every genome j in the **entire population** it calculates the distance between j and i, i being the **genome of fitness** f_i. Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitness values f_i This assigning of number of species offspring is used so that one specie can't take over the entire population.

6.1 Programming Details

The python version of the NEAT algorithm consists of a 'configparser' that makes it easy to set the hyperparameters directly from the config file. The config file has been broadly divided into four sections. The 'NEAT' section consists of the parameters generic to the NEAT algorithm such as fitness criterion, fitness threshold, population size, on fitness termination, and reset on extinction. The 'Default Stagnation' section is for the built in DefaultStagnation class used to deal with species that haven't improved over a given number of generations and thus are considered stagnant. When preserving the most fit individuals from each generation to the next, the DefaultReproduction section hyperparameters may be adjusted to define elitism and survival threshold. The 'Default Genome' section consists of hyperparameters such as mutation rate, initialization, statistics, addition and deletion probabilities etc. for nodes, connections, responses, weights and biases.

6.2 Hyper-parameters

Hyper-parameter	Variable	Bipedal Wallker	Lunar Lander
Fitness Threshold	fitness_threshold	300	200
Population Size	pop_size	200	100
Number of inputs	num_inputs	24	8
Number of outputs	num_outputs	4	4
Connection Addition Probability	conn_add_prob	0.5	0.15
Connection Deletion Probability	conn_delete_prob	0.5	0.1
Mutation Rate	mutate_rate	0.01	0.02
Node Addition Probability	node_add_prob	0.2	0.15
Node Deletion Probability	node_delete_prob	0.2	0.1
Weight Mutate Rate	weight_mutate_rate	0.8	0.8
Bias Mutate Rate	bias_mutate_rate	0.7	0.7
Max Stagnation	max_stagnation	20	15
Elitism	elitism	10	2
Survival Threshold	survival_threshold	0.2	0.2

Figure 7: Hyper Parameters for NEAT

7 Algorithm: DDPG

Deep Deterministic Policy Gradient (DDPG) lies under the class of Actor-Critic Methods. The actor produces a deterministic policy and the critic evaluates the deterministic policy. One limitation of DQN is getting the value for continuous action. The critic in DDPG is used to approximate the maximizer over the Q-values of the next state, thus solving the problem that occurs with DQN.

7.1 Network Architecture

The actor and the critic consist of two and four fully connected hidden layers respectively. Earlier we used the ReLu activation function, but the critic-network now uses Leaky ReLu as it is more 'balanced' and speeds up the training process. Tanh activation was used on the output layer for the actor-network as it ensures that every entry in the action vector is a number between -1 and 1.

Given a state, the actor is basically learning the argmax of Q(s,a) which is the best action. The critic learns to evaluate the optimal action-value function, i.e. a new target value for training, by using the actors' best-believed action

Method	Description	Estimate used	Tend to have
Policy-based methods	Agent learns the policy directly from the states	Monte-Carlo estimate	high variance and low bias
Value-based methods.	Agent learns the policy via the action valued function	TD estimates	low variance but high bias
Actor-Critic methods	Actor is a neural network which updates the policy and the critic is another neural network which evaluates the policy being learned which is, in turn, used to train the actor.	The critic is updated using the TD-error and the actor is trained using the deterministic policy gradient algorithm.	Solves bias-variance problem by combining both the methods

Figure 8: Methods used by agent to learn a policy

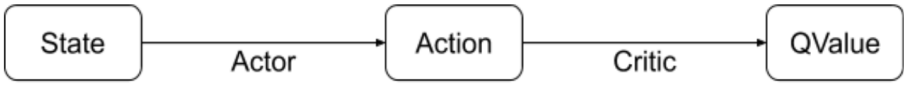


Figure 9: DDPG Actor-Critic flow

7.1.1 Replay Buffer

During each roll-out, we save all the experience tuples (state, action, reward, next-state) and store them in a finite-sized cache — a "replay buffer." Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

7.2 Hyper-parameters

Hyper-parameter	Variable	Values
Batch Size	BATCH_SIZE	128
Replay Buffer Size	BUFFER_SIZE	int(1e6)
ɣ (gamma - discount factor)	GAMMA	0.99
Learning Rate (Actor)	LR_ACTOR	1e-4
Learning Rate (Critic)	LR_CRITIC	3e-4
τ (soft update of target parameters)	TAU	1e-3
Number of Episodes	n_episodes	2000
Number of Time steps per episode	max_t	700
Weight Decay	WEIGHT_DECAY	0.0001

Figure 10: Hyper Parameters for DDPG

8 Conclusion

The initial idea was to compare and analyze various deep reinforcement and evolutionary learning algorithms with the help of the OpenAI Gym Environment, but preliminary results shed a light to the computational complexities of some tasks. Through this project we conclude that e performance of each algorithm is highly task dependent and that a lot of trade-offs exist between the two.

In Deep Reinforcement learning, we implemented DDPG and DQN. DQN discritizes the action space. It has worked well in complex environments (like Atari games), but when the action space is continuous (esp. Tasks involving physical control), discretization results in a very large action space. DDPG solves this issue, and we could use it for 'LunarLanderContinous-v2'. As the critic approximates the maximizer over the Q-values, DDPG can be seen as an approximate DQN method. Hence a comparatively better result was found with DDPG.

The biggest challenge that we have faced so far is computational cost especially in the NEAT algorithm which lacks gpu support. In theory NEAT should perform better, given the condition it has enough cores available for parallelization, but since at every generation, a new population of the same size is generated, run for same number of time steps, the number of parameters keep multiplying thus increasing time complexity on basic computing machines. In complex tasks where evolutionary algorithm such as NEAT took 3 days and still did not solve the environment, DDPG with the support of CUDA toolkit, trained on GeForce GTX 1660 Ti, with 6GB memory and a compute compatibility of 7.5 was able to solve it in approximately an hour. Due to the complexity of these algorithms in terms of number of hyper-parameters, we could not perform a grid search in order to find optimal hyper-parameters. Values that were found to be most well-suited from the literature survey we had conducted were used.

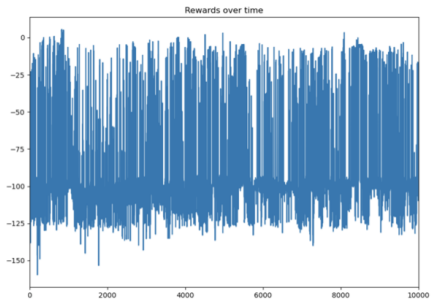
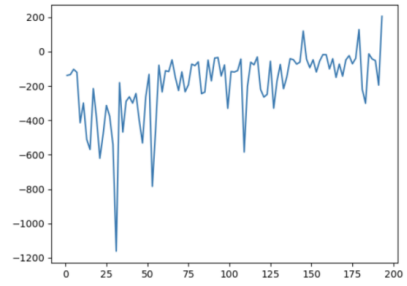
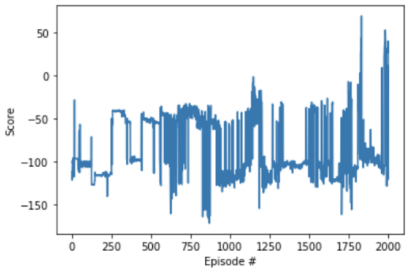
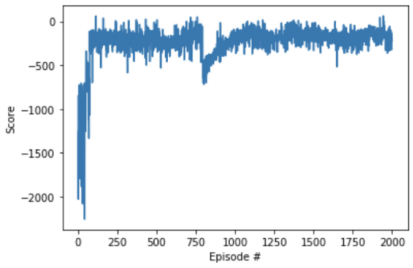
Algorithms	Bipedal Walker	Lunar Lander
DQN		
DDPG		

Figure 11: Score v/s Epochs graphs

9 References

1. Hands-On Intelligent Agents with OpenAI Gym: Your Guide to Developing AI Agents Using Deep Reinforcement Learning Book by Praveen Palanisamy
2. Documentation of OpenAI https://gym.openai.com/docs/
3. https://openai.com/blog/evolution-strategies/
4. https://arxiv.org/pdf/1509.02971.pdf
5. https://arxiv.org/pdf/1712.00006.pdf
6. https://project-archive.inf.ed.ac.uk/ug4/20181201/ug4_proj.pdf
7. https://www.tensorflow.org/agents/tutorials/0_intro_rl
8. https://box2d.org/documentation/
9. https://neat-python.readthedocs.io/en/latest/index.html
10. http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pd