Developing and Investigating Reinforcement and Evolutionary Strategies for Bipedal Walking

Group Members:
GOBINDRAM, Tavish
IMRAN, Shayan

Introduction
In this report we present our experiments which involved building an agent which uses various learning methods to perform successfully in Open AI’s “BipdealWalker-v2” environment. We used two different approaches to tackle the problem; reinforcement learning and evolutionary strategies. We also present a comparative analysis on the performance of the agents, comparing the evolutionary learning and reinforcement learning strategies to each other, as well as comparing variants of implementing the two approaches in their own domains.

Task definition
For this task our goal was to get the 2D robot to walk a certain distance without falling over as this results in the episode ending and the robot losing points. To maximize the total reward earned by the robot we had to make it move as far as possible while also minimizing training time. To accomplish this we utilised the 24 dimensional observation space as an input to our algorithms and produced a 4 dimensional output space. The 24 observations and the 4 outputs are all floating point numbers which usually range from -1 to 1. (The observations may be less than -1 or more than 1 in extreme situations).

Observation Space
Hull Angle, Hull Angular Velocity, X Velocity, Y Velocity, Left Hip Angle, Left Hip Velocity, Left Knee Angle, Left Knee Velocity, Left Leg Ground Contact, Right Hip Angle, Right Hip Velocity, Right Knee Angle, Right Knee Velocity, Right Leg Ground Contact, 15 to 24: Readings from lidar, only relevant when there are obstacles present in the path

Output Space:
Left Hip, Left Knee, Right Hip, Right Knee


**Approach**

**Evolutionary Strategies**

**Introduction**

Evolutionary algorithms take heavy inspiration from the natural evolutionary process, incorporating ideas of a population of individuals that mate with each other to produce fitter individuals and also undergo random mutations that change their characteristics, after which they are placed in an environment in which only the fittest individuals survive. Using these ideas we used evolutionary algorithms in a manner similar to hill-climbing in order to optimize individual characteristics and produce better results. The importance of each of the fundamental evolutionary operators mentioned above is stated below followed by the details of our implementation.

**Crossover/Mating operator:** The mating operator is one that combines individuals in a population with the aim of producing a better individual by combining favourable parts of the *genes* in the parents and passing them on to the child to produce an overall better individual.

**Mutation Operator:** This operator is used to produce genetic diversity in a population and can be very helpful when trying to encourage the population to escape a local optimum by changing randomly chosen *genes* in an individual.

**Selection Operator:** The selection operator is used to ensure the *survival of the fittest* by pitting individuals in a population against each other and picking the best individuals from the population that move on to the next generation while the rest of the individuals are discarded. It is important not to discard all bad individuals however, as this would result in a loss of genetic diversity and the population may get trapped in a local optimum.

**Implementation**

For the evolutionary strategies we used the DEAP framework which already has a large number of evolutionary algorithms built-in ([http://deap.readthedocs.io/en/master/](http://deap.readthedocs.io/en/master/)).

**Population:** The size of the initial population was 200 individuals.

**Individual:** Each individual in our population was an ordered array of 56 floating point numbers initialized with random values between -10 and 10. In this implementation we discarded the last 10 observations from the observation space and used only the first 14. The individual was then divided into four equal length subarrays (each of length 14) and each value in the individuals subarray was used as a weight for each observation value and the products of all these values were then summed to produce an output value. This is done for each of the four subarrays to produce 4 outputs.

\[
O_i = \text{Output } i, \quad I_i = \text{Individual } i, \quad B_i = \text{Observation } i \\
O_1 = B_1I_1 + B_2I_2 + B_3I_3 + \ldots + B_{13}I_{13} + B_{14}I_{14} \\
O_2 = B_1I_{15} + B_2I_{16} + B_3I_{17} + \ldots + B_{13}I_{27} + B_{14}I_{28} \\
O_3 = B_1I_{29} + B_2I_{30} + B_3I_{31} + \ldots + B_{13}I_{41} + B_{14}I_{42} \\
O_4 = B_1I_{43} + B_2I_{44} + B_3I_{45} + \ldots + B_{13}I_{55} + B_{14}I_{56} \\
\]

**Crossover Operator:** At first we used a two point crossover operator that randomly selected two indices between 0 and 55 and swapped the floats in two randomly selected individuals between these two indices. Since this did not produce good results we switched to a uniform crossover operator which takes two individuals and exchanges each of the 56 floats between the two individuals with a probability of 0.125.

**Mutation Operator:** For the mutation operator we chose a simple bit flip mutation operator, which flipped each of the bits in every float inside an individual with a probability of 0.05.

**Selection Operator:** In order to select the individuals that will move on to the next generation we used tournament selection which randomly selected three individuals from the population (while also ensuring that each individual gets
selected at least once), applied the evaluation function on the individuals and selected the best individual from these to move forward to the next generation. We also made sure we maintained a population of more than 100 individuals in each generation.

**Evaluation Function:** The evaluation function used the individuals to produce outputs in the way shown above and gave them as inputs to the gym environment. The gym environment then outputted new observations and rewards.

**Optimization:** We used multithreading to parallelize the evaluation of each individual since each individual is independent of the others. This resulted in a very large speedup in the evaluation of each generation.

**Results**

After running the algorithm using the parameters described above for 1000 generations, which took around 3 to 4 hours on an 8 core machine, we obtained the results shown in the graph below (x axis represents generation number). The graph shows that the population was able to evolve from the fittest individual having a max reward of -100 to one having a max reward of 150, all in the span of 60 generations. This trial was particularly fast due to favourable random numbers being produced, the problem is however susceptible to a much slower start if the initial random numbers are not favourable. The max value then increases slowly until it reaches its highest value of 200 after a total of around 230 generations. Subsequent generations do not result in significant increases in this reward which implies that the approach used was able to obtain a locally optimum array of individuals very quickly and then did not try to explore further for a potentially different global optimum. The algorithm also tries to maintain diversity in the population as can be seen by the minimum value which is always low, meaning that the population contains good and bad individuals.
Reinforcement Learning

First Approach: Naive Q-Learning

Introduction:
Q-Learning is a model-free algorithm that learns optimal actions for a given state by maximizing its Q-values i.e. optimal expected sum of discounted rewards after taking a particular action. The Q-value update equation can be seen below.

\[ Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a_{t+1})) \]

Implementation

Parameters
Learning rate: \( \alpha = 1 \) due to the deterministic nature of the environment.
Discount factor: none was employed as early rewards are not prioritized

Exploration Policy
We used an epsilon-greedy policy with a decaying epsilon given by \( \min(0.995^{\text{episode}}, 0.01) \)

Discretization
Q-Learning traditionally is only employed in environments with discrete state and action spaces, due to the tabular nature of the algorithm. In order to use it for the 'BipedalWalker-v2', a continuous control environment, we had to apply some discretization. To prevent a tremendous loss of information, discretized the state space by rounding the value of the state variables to the nearest 0.15. Additionally, we were able to circumvent discretization of the action space by applying a learning rate \( \alpha = 1 \) which allowed us to only store optimal actions for each state.

Programmatic Optimization
This aggressive learning rate applies to our problem as it is a deterministic environment. Another optimization employed to prevent high memory usage was storing states dynamically into a dictionary data structure, reducing the memory overhead of keeping a very large sparse array in memory.

Results

Naive Q-Learner

![Graph showing reward over episodes for Naive Q-Learner](image)
After training for up to 25000 episodes (~5 hours), values remain highly volatile and no obvious positive trend of rewards can be seen, this implies that a much larger number of episodes must take place to see any results, as despite discretization there may be a large number of states still unexplored. Additionally, taking random actions at every new state without taking into account similarity with other states would have slowed down learning speed.

**Second Approach: Deep Deterministic Policy Gradient**

**Introduction:**
Due to the unsuccessful nature of our value-based Q-Learning approach, a natural progression was to experiment with policy gradient methods. Particularly, we employed a Deep Deterministic Policy Gradient (DDPG) which was proposed in the following paper (https://arxiv.org/abs/1509.02971).

Policy Gradients in general optimizes a policy by computing an estimate of the reward of a policy and updating the policy in the direction of the gradient. DDPG is a variation of this algorithm. DDPG uses a stochastic policy to ensure exploration but determines a target deterministic policy. It takes on an actor-critic structure where the actor receives states as inputs and outputs actions, while the critic takes states and actions as inputs and outputs Q-values.

The following equation acts as the gradient for the policy’s performance

\[
\nabla_{\theta_\mu} J \approx \mathbb{E}_{a_t \sim \rho^a} \left[ \nabla_{\theta_\mu} Q(s_t, a_t|\theta^Q)|s=s_t, a=\mu(s_t|\theta_\mu) \right] \\
= \mathbb{E}_{s_t \sim \rho^s} \left[ \nabla_{\theta_\mu} Q(s_t, a|\theta^Q)|s=s_t, a=\mu(s_t) \right] \nabla_{\theta_\mu} \mu(s_t|\theta_\mu)
\]

Continuous Control with Deep Reinforcement Learning - Silver et al.

Further optimizations as mentioned in the algorithm’s publication include utilizing experience replay and target networks which allows for a more stable learning process.

**Implementation**
The learning algorithm was implemented with the aid of Keras and Keras-rl libraries which provided a high-level interface for deep reinforcement learning techniques.

**Networks:**

*Actor (Dense) Network:* Input 14 units, 2 hidden layers with 400 and 300 units, output with 4 units.

*Critic (Dense) Network:* Input 14 units, 3 hidden layers with 400, 404 (injected with 4 action-units) and 300 units, output with 1 unit.

Hidden layers for both networks utilized the relu activation function, which is cheap to compute compared to sigmoid and has experimentally shown better results. And the output layer of the actor network utilized a tanh activation (as actions need to be bounded between -1 and 1) and the output of the critic network uses a linear activation. 

*Optimizer:* Adam Optimization (generally successful and efficient according to literature)

**Noise:**

Added Ornstein-Uhlenbeck process for exploratory noise (Suggested in the aforementioned DDPG paper)

**Experience Replay:**

Experience buffer of size 1e5 with batches randomly injected during the training process.
Results:

Despite having been shown to succeed at several complex environments, we were not able to see results from the DDPG model, which has seemingly reached a highly unfavorable solution or just has much more room for exploration. This is clearly due to the lack of training and limited computational resources which are not able to quickly optimize the model’s $\approx 127k$ different parameters. Additionally, there may be more room for hyperparameter optimization and preprocessing approaches to speed up learning and reduce the probability of getting stuck at a suboptimal solution.

Comparative Analysis

Ease of Implementation: Both approaches mainly utilized existing frameworks with some modifications and optimizations for the specific problem. The DEAP framework is comprehensive in its implementation of evolutionary tools which were straightforward to use with aid from the documentation. The Keras-rl library used for DDPG also provided us with customizable agents which eased the implementation of sophisticated algorithms, and Q-Learning was fairly straightforward to implement in plain python.

Speed: The evolutionary approach was very well suited to the problem domain as we treated it as a continuous control optimization problem. The evolutionary algorithm was able to test a very large part of the domain fairly quickly as it was initialized randomly and in each generation it improved upon its predecessors to whittle down the constraints to a close to optimum value. The usage of multithreading resulted in an even larger speedup so this algorithm can very effectively take advantage of parallel computing resources with no side effects. The reinforcement learning strategies on the other hand took a very long time to train. In terms of DDPG stable updates by its very nature has a consequence that training would take a long period, particularly given the limited computing infrastructure utilized (an ASUS X450J laptop). Similarly Q-Learning suffers from requiring a large training time and with it’s naive approach would require a much larger set of episodes.

Results: In out short period of testing with limited resources it can be seen that there is a large gap in the results achieved by the two methods. The evolutionary strategies outperformed reinforcement learning strategies by a large margin and we believe this is because the evolutionary approach was very well suited to this problem and was able to take advantage of standard computing resources. We believe that reinforcement learning strategies should also be able to tackle this problem, but due to the limited computing resources we were unable to run them long enough to obtain a result similar to evolutionary strategies.

References

http://ai.berkeley.edu
https://keras.io
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html
https://github.com/matthiasplappert/keras-rl
https://github.com/DEAP/deap
https://arxiv.org/abs/1509.02971