Bipedal Walking using Genetic Algorithms and Reinforcement Learning

Aniketh Reddy   Chaitanya Perugu   Ganesh Balakrishnan   Nishant Shah   Samruddhi Kadam

Abstract

For this project, we propose to learn bipedal walking using Artificial Intelligence techniques. We plan to use Walker2d-v2, an OpenAI Gym environment as our testbed. The AI agent will have to learn a policy which allows the simulated humanoid to walk or run as far as it can without falling. We wish to test out two promising approaches to the problem, namely, Genetic Algorithms and Reinforcement Learning. We would then like to compare the two AI agents with a random agent and see how each walker performs.

Introduction

An important learning paradigm for walking of robots in the recent years is Reinforcement Learning (RL). Over the years Genetic Algorithms also proved to be efficient in solving this problem. We have compared these two approaches to train a Walker in Mujoco physics simulation engine. Mujoco is the first full-featured simulator designed from the ground up for the purpose of model-based optimization, and in particular optimization through contacts. The state space and the action space of this environment is 17 and 6 respectively. The image below shows the 'Walker2d-v2' in Mujoco environment.

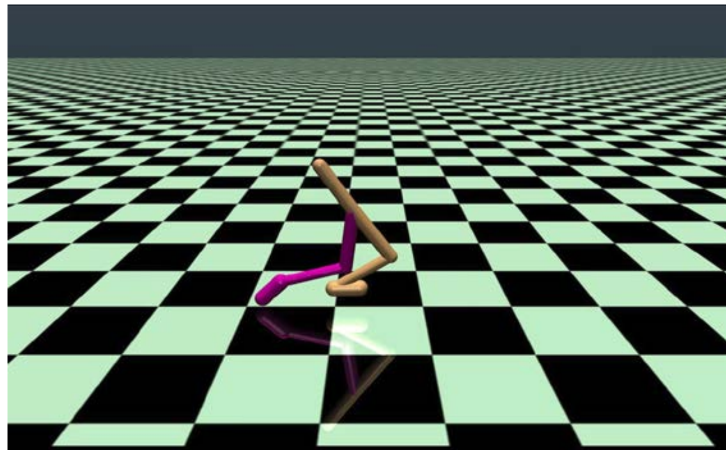


Fig 1: Walker2d-v2

***<u>All the graphs</u>: **X axis** - Number of episodes*
***Y axis** - Total reward received per episode*

We have also used another environment from Mujoco named 'Hopper-v2' whose state space and action space is 11 and 3 respectively. This is was used along with 'Walker2d-v2' to train a Genetic Algorithm. The image below shows the 'Hopper-v2' in Mujoco environment.

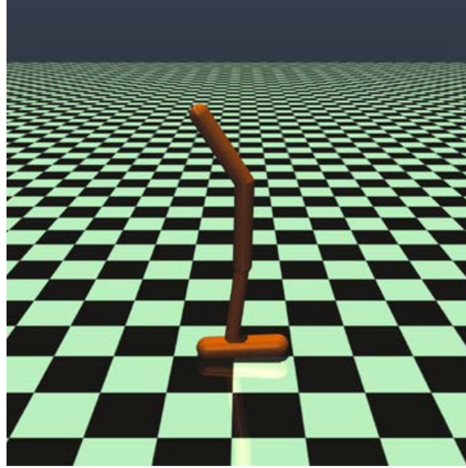
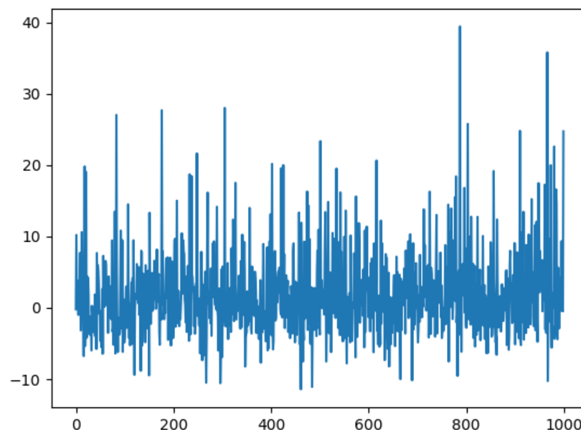


Fig 2: Hopper-v2

Random Agent

A random agent takes a purely random action at every time step, thus not learning from its experience to improve its performance. We use a random agent to build a baseline for our AI algorithms to beat. The results of a random agent in the Walker2d-v2 environment is shown below.



<u>All the graphs</u>*: *X axis* - *Number of episodes
Y axis *- Total reward received per episode*

The x-axis is the number of episodes while the y-axis shows the score. These results are not surprising because it is an unintelligent agent. It can be seen that the average score is very close to zero. In fact, it is around 1.5; to put it in perspective, a score of about 5000 is required to call the environment solved.

Deep Deterministic Policy Gradients

Real world problems are high dimensional and learning function approximations that map a continuous high dimensional state space to a continuous high dimensional action space is a hard task. The model(transition model $T(s_{t+t}|s_t,a)$) is of magnitude s x s x a and for real world problems it would be infinitely large. One approach is to have a notion of the direction in which the function approximation for policy has to be changed in order to maximize the expected reward for taking actions according to the policy the agent is learning. One such approach is the policy gradient approach. Policy gradient methods in reinforcement learning have become increasingly prevalent for state-of-the-art performance in continuous control tasks. The deterministic policy gradient has a particularly appealing form: it is the expected gradient of the action-value function. This simple form means that the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient.

The basic idea is to represent the policy by a parametric probability distribution $\pi_\theta(a|s) = P[a|s; \theta^\mu]$ that stochastically selects action a in state s according to parameter vector θ^μ. Policy gradient algorithms typically proceed by sampling this stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward. Function approximations for real world problems are very complex and cannot be engineered easily. With the advent of Deep Neural Networks to learn highly complex function approximations, a very active field of research is the use of such techniques to make reinforcement learning agents learn a policy function that maps a state to action. Deep Deterministic Policy Gradient is one such technique where there are two networks that work in symphony to learn a Q value function and a policy function, parametrized by the network parameters θ^Q and θ^μ respectively. The network that learns the policy function is called the actor network and the network that learns the Q value function is called the critic network. The policy that the actor network has learnt is evaluated by the critic network and there is a TD error that tells it how the actor and critic have to change their parameters (weights) as seen in figure 3.

***<u>All the graphs</u>: **X axis** - Number of episodes*
***Y axis** - Total reward received per episode*

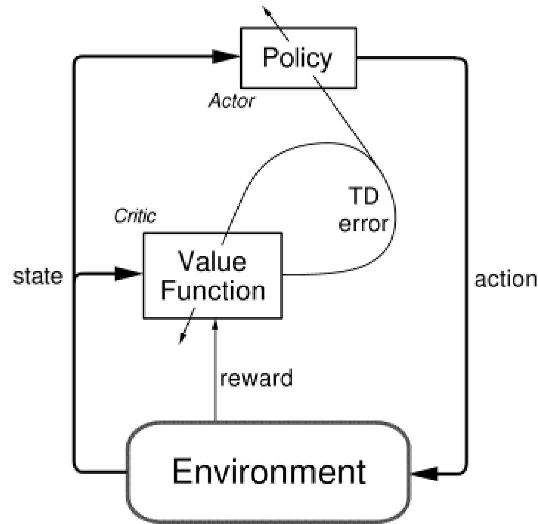


Fig 3: Actor-Critic Architecture [6]

DDPG is an off-policy learning algorithm where the agent updates its policy using another policy. This is very similar to Q-learning where the policy of the agent is learnt using a greedy policy to calculate the TD error.

The algorithm:

A Q-Value function, a policy and an exploration scheme are characteristics of any reinforcement learning problem.

Q-Value function and Policy function:

The agent interacts with the environment at every discrete time step t, receives an observation s_t , takes an action a_t, ends up in a state s_{t+1} and receives a scalar reward r_t. The Q-Value is the expected accumulated reward that the agent would get taking actions according to the policy it has learnt till the trial ends. The agent tries to learn a policy that maximizes the Q-Value. DDPG being an off-policy learning algorithm uses a greedy policy to update its current policy, very similar to Q-learning. To scale such simple off-policy algorithms to solve problems of such high dimensional spaces using Neural networks, two concepts were introduced : replay buffer and a target network. The target network computes the true Q-value for that state action pair that serves as the true value for computing the error in the critic network (will be discussed below).

Neural Networks can learn good general function approximation when the data used to train it is independent and identically distributed (i.i.d). The problem in a reinforcement learning scenario is that the state where the agent would end up in and the action it would take from there are dependent on the previous state and the action the agent had taken. Thus, the data is not independent. One solution to this problem is to store all the experiences (s_t , a_t , s_{t+1} , r_t) in a

All the graphs*: ***X axis *- Number of episodes*
Y axis *- Total reward received per episode*

replay buffer and sample randomly a batch of fixed size. At each time step an action is taken, a state is observed and a reward is obtained and these are added to Replay, a batch is sampled and the actor and the critic networks are trained. NNs are basically matrix multiplications and additions (linear operations) that map the inputs to the outputs. To increase their ability to generalize on large dimensional data there must be non-linear activations that makes the function learn better. In reinforcement learning scenarios adding non-linearities might lead to instability sometimes also divergence. This is because of the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values and the target values.

To solve this problem a variant of a method to update target networks proposed by Mnih et al.[7] was used. Two target networks (for policy and Q-value) are necessary to have stable y_t (used to calculate loss function) to train the critic network without divergence. So, the parameters of the actor and the critic are copied into the target actor and critic. The weights of the target networks are updated using a weighted sum between the weights of the trained and the target network giving the weights of the trained network a very small weight. Thus the target values change very slowly thereby improving stability. So, basically the y_t obtained using the target networks (mentioned below) are used as true values to train the critic network. The problem of training the reinforcement learning agent now reduces to a supervised learning problem which is stable.

Exploration Noise:

Exploration-exploitation trade-off is a key concept in reinforcement learning. Knowing when to stop exploring and when to start exploiting the policy learnt is of utmost importance. Exploration is necessary in a stochastic world to find out the best possible policy that the agent should follow to maximize the expected reward for the agent. The exploration strategy used is an Ornstein-Uhlenbeck noise [8] which is a mean-reverting noise,i.e., as time passes the noise slowly zeros out towards its mean value. The OU Noise is given by:

$$dx_t = \theta(\mu - x_t)\, dt + \sigma\, dW_t$$

Where, W_t is a Wiener process [9] for which dW_t is a Gaussian with mean = 0 and variance = 1.

***<u>All the graphs</u>: **X axis** - Number of episodes*
***Y axis** - Total reward received per episode*

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

Fig 4: DDPG algorithm

The actor policy is updated using the sampled policy gradient given in the algorithm. This is obtained by a simple chain rule that splits it into two vector terms being multiplied. The first term $\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)}$ is obtained from the critic network and the second term from the gradient of actor net's outputs with respect to its parameters.

Hyperparameters:

These hyperparameters were borrowed from Lillicrap et al.'s paper [3]

Hyperparameters	Actor	Critic
Number of hidden layers	2	2
Number of units/layer	1st layer 400, 2nd layer 300	1st layer 400, 2nd layer 300
Activation	Both hidden layers - softplus Final output - tanh	Both hidden layers - softplus Final output - tanh
Learning Rate	1e-4	1e-3
τ	1e-2	1e-2

***All the graphs: **X axis** - Number of episodes*
*   **Y axis** - Total reward received per episode*

Other non-learnable parameters:
$\gamma = 0.99$
Batch size = 64 for both actor and critic
Length of replay buffer = 100000 , remove initial entries if full
Noise:
$\mu = 0$
$\Theta = 0.15$
$\sigma = 0.2$

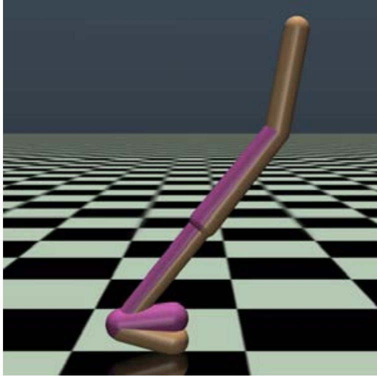
The weights of the network are initialized from a normal distribution with zero mean and standard deviation equal to $1/\sqrt{n}$, where n is the number of inputs to the layer (fan-in). This is done because the tanh activation function used in the output layer is very sensitive(discussed below).

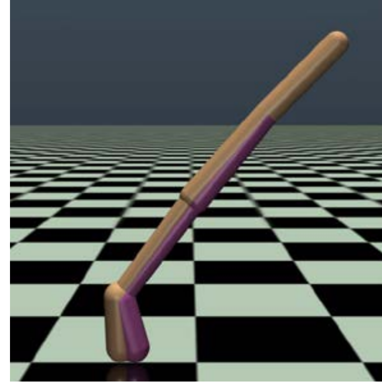
The reward structure for the environment is as follows:
The environment has a position handle that gives position of Center of Mass (horizontal), height of Centre of Mass (vertical) and the angle of the robot. The robot is only allowed to tilt upto a certain angle. If it moves more than that, the episode terminates. The episode also terminates when the robot's Center of Mass goes above or below a height of 2 and 0.8 respectively. There is a reward for the robot moving forward given by reward = ((posafter - posbefore) / dt), a +1 reward for being alive and a negative reward given by 0.001*(norm of action taken). The actions are continuous ranging from [-1.0,1.0]. This explains the tanh activation being used.

Results

Trial 1:
With the above mentioned parameters, agent after training for 2000 episodes, had learnt to jump in the air within the height limit of the environment mentioned above by rotating just its angle joints. The policy had saturated. This is because the living reward is pretty high, compared to other rewards and it is good to live longer without taking too many actions. We thought it might have been due to a low value of sigma of the OU Noise and tried tweaking that.

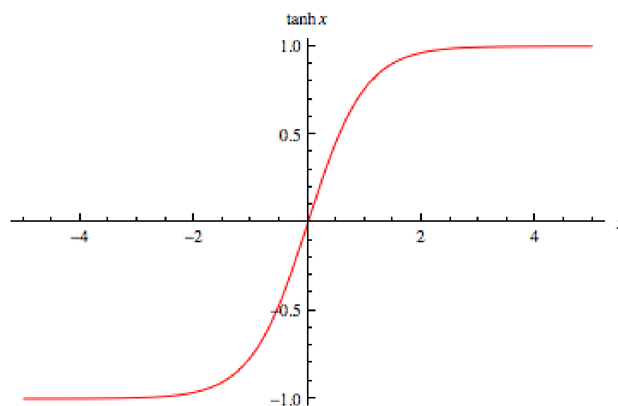
***All the graphs: **X axis** - Number of episodes*
***Y axis** - Total reward received per episode*





<u>Trial 2:</u> OU Noise: sigma = 0.3

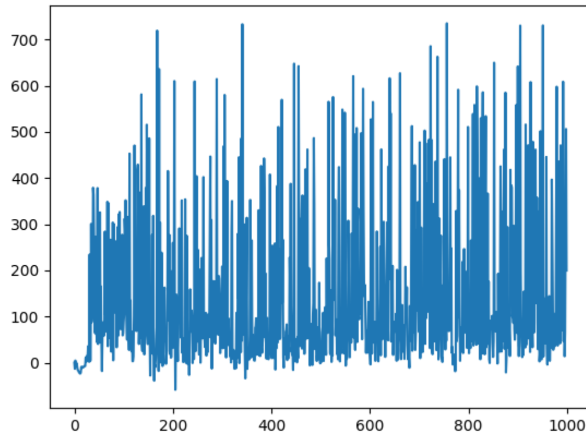
With the other parameters remaining the same we ran the trials again. The agent had explored slightly better but one thing we realised was that the actions it was performing was either 1 or -1 and had saturated at that point. We realised that the weights had dropped to zero.



This is because we can see that around the value of (-2,2) the function has a gradient and otherwise it is a straight line parallel to the X-axis. This means that in the regions greater than 2 and less than -2 the gradient term for the last layer drops to zero. Back propagation is nothing but a simple gradient based update and since the gradient of the last layer is zero, gradient of all the weights wrt the output becomes zero. Thus none of the weights are updated further. We realized that using a function such as tanh that squishes the output but results in saturation is a bad decision. Instead we decided to manual engineer our gradient terms.

***<u>All the graphs</u>: **X axis** - Number of episodes*

***Y axis** - Total reward received per episode*

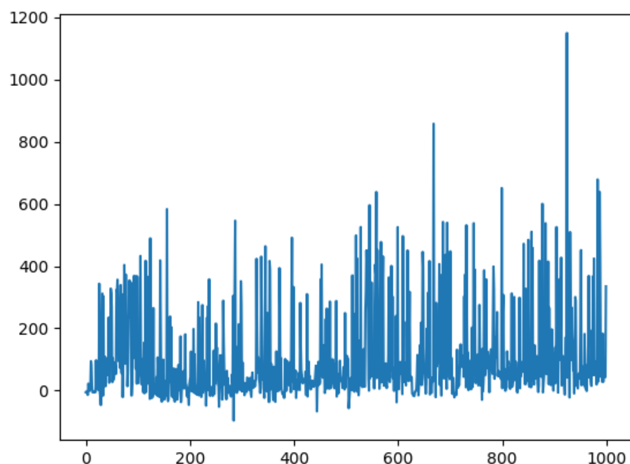


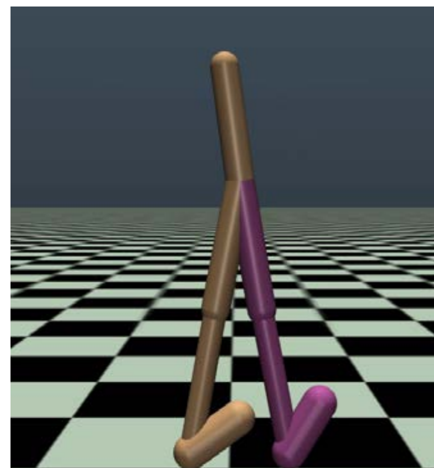
Trial 3: Inverting the Gradients [10]

In this method, the gradients are decreased as the agent's actions start going closer to the boundaries of the action space and are inverted (sign changes) if parameters cross the bounds. We can see below that as p gets closer to pmax or pmin the gradient term becomes closer to zero and as it crosses the bounds the gradient term changes sign.

$$\nabla_p = \nabla_p \cdot \begin{cases} (p_{\max} - p)/(p_{\max} - p_{\min}) & \text{if } \nabla_p \text{ suggests increasing } p \\ (p - p_{\min})/(p_{\max} - p_{\min}) & \text{otherwise} \end{cases}$$

This is better than using a tanh activation function because it does not lead to saturation but also keeps the action between the bounds. When we tried this out the agent had learnt two things. One was to move one of its legs backward if it is about to fall behind to balance itself and tried to stay still doing minimal actions as it received a positive living reward. Another thing it had learnt was to rotate its ankles whenever it fell forward as this also pushed it forward and kept it living longer with minimal actions.



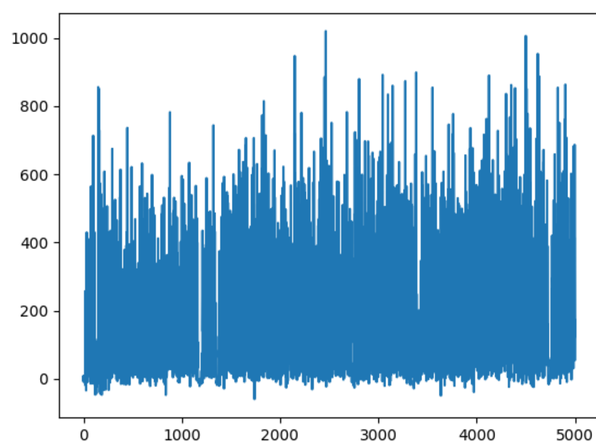


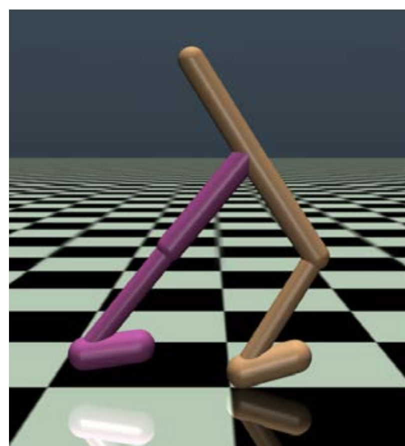
*******All the graphs*: ***X axis*** *- Number of episodes*

Y axis *- Total reward received per episode*

Trial 4: Cyclic action space

One more thing we realised was that the agent does not initially explore that much. This is attributed to the fact that the initial exploration noise is high and when added to the action it leads to the action being above 1 or below -1 (beyond the bounds). This sets the joints to their maximum actions thereby not exploring much initially and as time passes exploration reduces and it start to exploit the policy that it has learnt which is to hop by moving its ankle. Thus we made a loop enclosure for the action space. In other words say if the action was 1.2 then the action would loop 0.2 (1.2-1 = 0.2) from -1 giving an action of -0.8. This lead to an exploration behavior that was intended in the paper [3]. We saw the agent performing more sets of actions and learning running gaits. Since there is more exploration due to cyclic in this case we ran it for a larger number of episodes.





Trial 5: Batch Normalization

We realized that the input data (the state space) to the neural networks had a large variance (as it ranges from -inf to inf and is high dimensional) in certain directions and the training was very slow and susceptible to local minima. So, we had to whiten the data to basically condition the data in such a way that the agent does not reach a local minima. So, we tried to implement the batch normalization method which is an approximate whitening transformation. We were not able to implement it within the limit and decided to continue on this in the near future.

Deep GA

Inspired by recent work by Salimans et.al., Uber AI Labs have released a paper that seeks to apply gradient-free evolutionary algorithms for evolving the weights of deep neural networks, a novel and fascinating combination that reportedly worked. They were able to train a neural network to perform well in RL tasks solely through basic genetic algorithms and demonstrated

***All the graphs: **X axis** - Number of episodes*
*__**Y axis**__ - Total reward received per episode*

that the trained network is competitive with most DRL trained networks (like DDPG and TRPO). This paper is titled "*Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*" and the most recent draft was updated on April 20, 2018 (a mere 10 days ago, if you are reading this on May 2).

We found this paper very interesting and decided it would make for a good comparison with standard DRL algorithms like the DDPG we are implementing. The core idea is simple enough. Instead of using traditional gradient-based updates like backpropagation to update the weights of the neural network, the authors suggest using a genetic algorithm to intelligently search for a combination of weights that result in a high accuracy or reward, depending on how the fitness function is defined. This idea is not exactly new, researchers have attempted it earlier but, given the high dimensionality of the weights of a deep neural network, genetic algorithms were not very computationally feasible for this problem.

As part of this project, we apply the Deep GA algorithm to the Walker2d-v2 environment and attempt to learn a simple walking policy on flat ground. Since it is the action policy that needs to be learnt, we represent the policy with a deep neural network and try to learn the best combination of network weights. We opt for a simple 3-hidden-layer (64, 128, 64) network with tanh activation function at each layer. The network structure was motivated by the fact that the input and output layers have 17 and 6 nodes each. The activation function was chosen because the actions were all supposed to be in the range [-1, 1] for the Walker2d-v2 environment. The weights of this neural network act as the parameter state of the genetic algorithm. Since the entire network is just (17, 64, 128, 64, 6), the total parameter count comes up to about 18K parameters for a single set of weights. Although it might sound like a lot, this is a very small parameter space in deep learning standards. Yet, for a genetic algorithm, it is a very high dimensional space to effectively search.

The paper uses a basic variant of genetic algorithms with just elitism and mutation, but no crossover and justifies this choice by saying the authors wanted to try to set a baseline for genetic algorithms. We follow a similar model except we also include crossover. Considering the hyperparameters used in the paper (population size of 12,500, mutation strength of 0.0022 and 1,500 generations) were computationally huge and were chosen for the more complex and high-dimensional Humanoid-v2 environment, we went with a more reasonable set of hyperparameters for our Walker2d-v2 environment. Given below is a summary of our parameters:

***<u>All the graphs</u>: **X axis** - Number of episodes*

*   **Y axis** - Total reward received per episode*

Hyperparameter Name	Hyperparameter Value
Neural Network	(17, 64, 128, 64, 6)
Activation Function	tanh
Population Size	1000
Mutation Strength	0.01
Number of Generations	500

Keras, the package we use to create our neural network in Python, initializes the networks using Xavier initialization (random weights, zero biases). We mutate this initializes weights with a mutation strength of 0.5 to generate the initial population for the genetic algorithm. The mutation strength was set much higher than usual to encourage initial exploration. Mutation of the weights was implemented in a very straight forward way; we randomly perturb each element of the weights with a maximum deviation equal to the mutation strength. This slightly moves the entire network in a random direction when visualized in the parameter space and gives rise to most of the exploration that is key to genetic algorithms. Elitism simply retains a certain percentage of the top performing (according to fitness) individual states in the population.

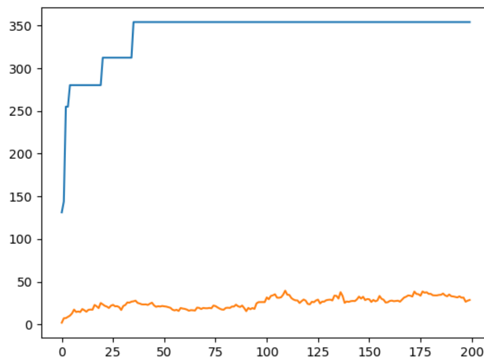
Our implementation of crossover is quite straight forward too but needs some justification. We simply swap the weights cut at a random layer in the neural network. For example, the child might get the weights of the first two layers of the mother network and the weights of the last layer of the father network. The idea behind this implementation is that the initial representation learnt by one network might work well with the final representation of another network. Since, the original paper uses no crossover at all, we too keep our crossover rates low and mutation rates high. By default, we use 10% elitism, 60% mutation and 30% crossover.

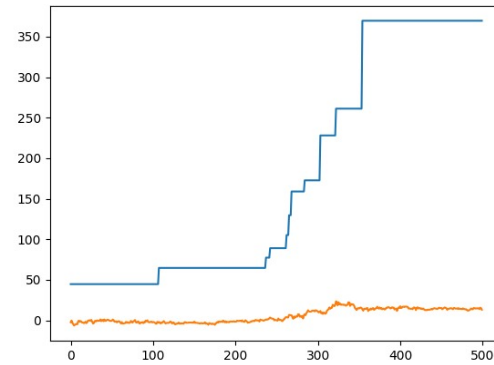
Results

Overall, the results of Deep GA are not very promising.

One issue we faced was the definition of the fitness function. We initially defined the fitness function to equal the reward achieved during one episode of the environment using the policy defined by neural network policy loaded with the weights. It lead to reasonably good graphs initially such as the ones below.

***<u>All the graphs</u>: **X axis** - Number of episodes*
***Y axis** - Total reward received per episode*





The graphs show the best individual reaching scores of over 350 while the average score of the generation creeps up to around 25-30. The x-axis shows the number of generations.

But, we believe this fitness function is not a true representation of the individual's performance. This is because some experiments show that saving these network weights and reloading them does not lead to the policy network achieving scores around 350. This suggests that the OpenAI environment incorporates noise into its simulation and achieving a good score once does not translate well to other episodes.

Hence, we designed a new fitness function that runs 3 episodes with the same network and averages these three scores to build a better estimate of the network's fitness. A larger set of simulations would be even better, but we are pressed for training time and computational resources and hence settled for three. This new formulation of the fitness function changed the entire performance of the algorithm and also made it much slower to test, about 3 times slower. The best individual in the first 200 generations now barely breaks above the score of 30-40 while the average score of the generation stays below 0 for most generations.

We believe the second fitness function is a much more representative one than the first. The disappointing performance of Deep GA in our experiments compared to its performance reported by Uber AI has only one sensible explanation: computational time and resources. The Deep GA paper ran 1500 episodes for a population size of 12,500. We ran it for a much shorter population size and much less generations and observed run times of up to 12 hrs at a stretch. This makes sense because an 18K dimensional space is extremely huge and difficult to search. The genetic algorithm would require a lot of time to find the direction which leads to a better fitness. In spite of running it on multiple cores of our CPUs, for 500 generations, we were not able to find one which consistently performs a score of over 100. It would be interesting to see how Deep GA would perform with much longer training times.

<u>All the graphs</u>: **X axis - Number of episodes*

***Y axis** - Total reward received per episode*

Conclusion

We implemented two AI algorithms in a bipedal walking setting and compared their performance against a random agent. The random agent performs extremely horribly as expected. The DDPG algorithm is the most promising of them all. The agent learns to balance itself and not fall down quickly. The agent also tries to hop forward because moving ahead would give a more positive reward. It has not learnt to walk consistently yet but it does realise that it has to try to stay up and hop forward. With more training and perhaps better parameter tuning, it might very well learn to walk and run. We also tried an extremely new approach suggested by Uber AI Labs called Deep GA. In spite of faithfully implementing their algorithm, we were able to achieve some success but nowhere close to what they report. We believe this is because of the computational time and resources we could spare for the task.

References

1. http://gym.openai.com/envs/Walker2d-v2/
2. Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2017). Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567*.
3. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.
4. Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.
5. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
6. Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction.
7. Mnih, et al. Human-level control through deep reinforcement learning
8. Bibbona, E.; Panfilo, G.; Tavella, P. (2008). "The Ornstein-Uhlenbeck process as a model of a low pass filtered white noise". *Metrologia*. **45** (6): S117–S126. doi:10.1088/0026-1394/45/6/S17
9. Doob, J.L. (April 1942). "The Brownian Movement and Stochastic Equations". *Annals of Mathematics*. **43** (2): 351–369. doi:10.2307/1968873. JSTOR 1968873
10. Matthew Hausknecht, Peter Stone. Deep Reinforcement Learning in Parameterized Action Space, Available at: https://arxiv.org/pdf/1511.04143v4.pdf

*******All the graphs*: ***X axis*** *- Number of episodes*
Y axis *- Total reward received per episode*