# Parametrization and Emperical Simulations of existing Environments and Agents

Bile, Christian
ezanin-christian-prince-carlos.bile@polytechnique.edu
Garcia, Marco
marco.garcia-macias@polytechnique.edu
Laique, Talha
talha.laique@polytechnique.edu

## I. Introduction

With our project we try to analyze two environments of two different complexities and action space in order to catch up the parametrization of the training needed for an overall good result, by doing extensive parameter tunning and generating empirical simulations. The two environments are: Cart Pole and Bipedal Walker. The Cart Pole was analyzed using DQN and DQN-Experience Replay while the BipedalWalker was analyzed with the DDPG algorithm using an Actor-Critic agent and then using Augmented Random Search. For the first, We have found that parametrized implementations of these agents surpass their naive implementations. While for the Bipedal Walker, we found out that more tunnings are needed for the DDPG to achieve a certain amount of steps of the Bipedal, but with the Augmented Random Search achieve much better results are obtained in less time.

For our experiments, we found on the internet the implementation of each agent algorithm analyzed in this project. We also extended these implementations to make them more suitable to our way of working, computational resources and goals. Thus, our project does not aim to implement those algorithms from scratch, but to analyze their training parameters on the Cart Pole and BipedalWalker, for future experiments on equal or similar environments.

## II. Background and Related Work

Our agents interact with our environments in a sequence of actions, observations, and rewards. At each time step, the agent selects an action that will maximize the future rewards. We have made a standard assumption that future rewards are discounted by a factor of $\gamma$ per time-step, and define the future discounted return at time t as $R_t = \sum_{t=t'}^{T} \gamma^{t-t'} r_{t'}$, where T is the time-step at which the simulation terminates. The optimal action-value function $Q^*(s,a)$ is the maximum expected reward achievable by following a policy, after seeing some sequences and then taking some action a, $Q^*(s,a) = \max_\pi \mathbf{E}[R_t|s_t = s, a_t = a, \pi]$, where $\pi$ is a policy mapping sequences to actions (or distributions over actions). The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value $Q^*(s',a')$ of the sequence $s'$ at the next time-step was known for all possible actions $a'$, then the optimal strategy is to select the action $a'$ maximizing the expected value of $r + \gamma Q^*(s',a')$ [1].

$$Q^*(s,a) = \mathbf{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \quad (1)$$

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s,a) = \mathbf{E}[r+\gamma \max_{a'} Q^*(s',a')|s,a]$. Such value iteration algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$. In our neural network we have used function to estimate the action-vale function, $Q(s,a;\theta) \approx Q^*(s,a)$, which is trained by minimizing a sequence of loss function like the Mean Square Error function, $L_i(\theta_i)$ that changes at each iteration $i$ with weights $\theta$ [1][2],

$$L_i(\theta_i) = \mathbf{E}_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2] \quad (2)$$

where $y_i = \mathbf{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration i and $\rho(s,a) \approx \sum_{s',r} \rho(s',r|s,a) = 1$ is a probability distribution that will always be equal to 1 as our environments are deterministic. $s'$ is the new states, $r$ rewards, $s$ previous states, and $a$ actions. The differential of the loss function with respect to the weights is given by:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbf{E}_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \quad (3)$$

Note that Q-learning is a model-free algorithm, so it learns from state transitions and it is also off-policy: it learns about the $\varepsilon$-greedy strategy for selecting the actions, $a = \max_a Q(s,a;\theta)$ by adequately exploring the state space.

## III. The Environment

- Cart Pole:
  - **State space:** 4 vectors corresponding to Cart position and velocity, Pole angle and velocity at tip.
  - **Action space:** Pushing the cart to left or right by applying force of +1 or -1 to the cart.

  - **Reward:** Reward of +1 is given for every time step the pole remain in the air.
  - **Environment:** Deterministic, the cart will either move to right/left depending on the direction of the applied force.
  - **Agent's Perspective:** Simple, agent only has to deal with 4 vectors and it is deterministic.
  - It is only for educational interest
- Bi-Pedal Walker:
  - **State space:** State consists of hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 Lidar rangefinder measurements to help to deal with the hardcore version. There's no coordinates in the state vector.
  - **Action space:** The action space is represented by the speed and the maximum torque for every joint of the robot.
  - **Reward:** Reward is given for moving forward, total 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points, more optimal agent, will get better score.
  - **Environment:** The environment in the normal, with slightly uneven terrain is deterministic, the angle speed and velocity are determined by the force applied to the joints. It is a full observed environment.
  - **Agent's Perspective:** Agent has to deal with many variables in a continuous space, depending on the architecture of the model, the agent may not converge to a optimal solution.
  - This is just a 2D model of a bipedal walker agent made for educational purposes, however a 3D model with more physical constraints can be useful in the robotics field.

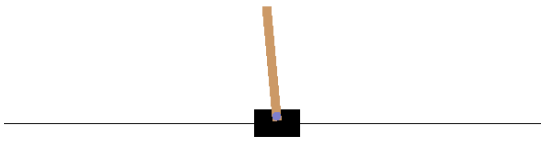


Fig. 1: Cart Pole at 180th episode.

## IV. Agents

- **Naive Deep Q Learning (DQN)**
  The basic strategy of Q learning is to compose a Q-Table with Q values corresponding to all combinations of state-action which are then updated using the Bellman equation, $Q(s,a) := r + \gamma \max_a Q(s',a)$, and when it reaches the terminal state $Q(s,a) := r$. The downside of this basic approach is that we can get stuck with local minima, e.g. if going left is the best move, our agent will always return that and we might skip more rewarding actions. DQNs replace Q-Table with neural network that approximates Q values, and for convergence it tries to minimize the relation given in eq. (2) [1]. The agent learns from previous states and selects the best action using $\varepsilon$-greedy approach with network. It converges when the $\varepsilon$ value is higher but the drawback is with lower $\varepsilon$-values it operates with restricted exploration of state space hence gets stuck at local minima. Moreover, the max function computing the expected Q values introduces potential bias. We have parametrized our network with $\epsilon$-min, $\epsilon$-decaying value, learning rate, no. of episodes, and no. of neurons.

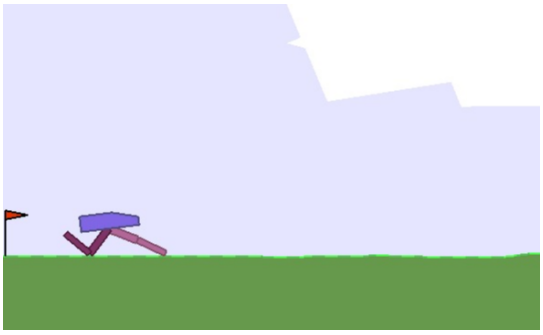


Fig. 2: Bi-Pedal Walker at 220th episode.

- **Deep Q Learning with Experience Replay**
  We have taken the implementation of a Naive DQN and optimized it with experience replay. The agent stores the experiences at each time-step $e_t = s_t, a_t, r_t, s_t + 1$ into a replay buffer. After the replay buffer counter, which keeps track of how many memories have been stored, has exceeded the provided batch size, then it learns from randomly sampled memories, selects and executes an action according to an $\varepsilon$-greedy approach. We have used fixed batch and memory sizes as training network, as with arbitrary length of inputs the task can be difficult. This approach has several advantages over naive Q-learning. First, each step of experience is potentially used in many weight updates, allows 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 policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically.

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 [1]. The algorithm is as follow;

**Algorithm 1:** Deep Q Learning with Experience replay

Initialize replay memory $D$ to capacity $N$;
Initialize action-value function $Q$ with random weights;
**for** episode = 1, $M$ **do**
  Initialise state $s_1 = x_1$ and Pre-State
  **for** $t = 1, T$ **do**
  With probability $\varepsilon$ select a random action $a_t$
  otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
  Execute action $a_t$ and observe reward $r_t$
  Set $s_t + 1 = s_t, a_t$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
  Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
  Sample random transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
  Set $y_j = r_j$, terminal $\phi_{j+1}$
  Set $y_j = r_j + \gamma \max'_a Q(\phi_{j+1}, a'; \theta)$, non-terminal $\phi_{j+1}$
  Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \theta))^2$
  **end for**
  **end for**

**Pre-State:** is an optimization technique inserted in our implementation consisting of letting the agent explores the state space as long as our accumulated memories (Agent's experiences) are less than the batch size. Our decision is based on the following main reasons: Firstly, Q-learning is a temporal difference learning technique, so we will be doing learning at each step after we filled up our memories; Secondly, if we don't accumulate these memories as per the batch size before learning we could end up sampling the same states that is going to take same actions over and over again. For instance, if we have a batch-size of 64 and we don't do the memory accumulation before learning, we could end up getting 64 times same states.
In our algorithm, we have included the pre-state as an initialization before learning because firstly, it is an optimization technique, secondly, we need to accumulate agent's experiences as per the batch size so we can start learning process and accumulate more memories, then select batch of random memories based on the given batch size, do the training again and repeat until it converges.

- **Deep Deterministic Policy Gradient (DDPG)**
The action space is continuous, we can't evaluate the space completely using DQN, it would require an expensive computation and not accurate estimation of $\max_a Q^*(s, a)$, at each step. The DDPG algorithm is suitable for the Bipedal Walker as it allows to get a good generalized predictor of actions and Q-learnings in the continuous action space of the agent. It is based on the Actor-Critic interaction in which the actor decides to take an action given the current state and the critic evaluates this action by estimating the action-value $Q(s, a)$. Thus, the critic judges the action by calculating the approximate future reward. The Actor and Critic are represented by neural network models, and the estimate $Q^*(s, a)$ is learned using the Bellman equation and the actor is updated by following the applying the chain rule to the expected return from the start distribution $J$ with respect to the actor parameters:

$$\nabla_{\theta^\mu} J \approx \mathbf{E}_{s_t}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]$$

This allows to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s, a)$, we can approximate it with $\max_a Q(s, a) \approx Q(s, \mu(s))$, as shown in the following algorithm.

**Algorithm 1** DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ 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**

This algorithm has some parameters that can be tunned:

- **Learning rate for the actor (LRA)** : this is the learning rate of the neural network model representing the actor;
- **Learning rate for the critic (LRC)**: The learning rate of the neural network model representing the critic;
- **Buffer size (BuffS)**: This parameter represents the minimum amount of data collected (experiences) necessary in order for the agent to start learning;
- **Batch size of the replay buffer (BS)**: size of the memory containing experiences;
- **Discount factor($\gamma$)**: a value between 0 and 1 defining the importance of the next reward;
- **Soft target parameter ($\tau$)**: for a regularization of the target weights;
- **L2 weight decay (L2)**: for a weights regularization of the critics neural network model;
- **Configuration of the Neural Network models of the Actor and the Critic**: This is the number of hidden layers, units per hidden layers and type of activation function.
In order to test the different parameters we set all in optimal values and then change them one by one

evaluating the results after training.

- **Augmented Random Search**
  It is commonly believed that model-free methods that explore the action space based on policy gradient like DDPG are more efficient for environments just like the Bi-Pedal Walker. However, in 2018 Horia et al. demonstrated that using methods that explore the parameter space of policies using a simple Random Search with augmented features could provide a competitive approach to this task.
  The Augmented Random Search consists in choosing a direction uniformly at random in the parameter space and optimize the reward function by updating the policy parameters $\theta$ along that direction computing a finite difference approximation.

$$\frac{r(\pi_{\theta+\nu\delta}, \xi_1) - r(\pi_{\theta-\nu\delta}, \xi_2)}{\nu}$$

  where $\xi_1, \xi_2$ are two i.i.d random variables, $\nu$ the standard deviation of the exploration noise and $\delta$ a zero mean Gaussian vector. Sometimes the evaluations may be noisy so mini batches of directions can be used to reduce the variance error of the gradient estimate.
  This random search can be improved if we add three main augmented features that will give the algorithm the name of Augmented Random Search: Scale the step update by the standard deviation of the rewards collected from that updated step, normalize the inputs using estimates of their mean and standard deviation and finally discard the steps whose directions yield the least improvement of the reward. The importance of these added features can be justified: in basic random search the training can lead to extreme variations in the rewards, affecting the learning, by scaling using the standard deviation this extreme variation can be reduced. The normalization ensures that policies put equal weights on the different components of the states and the selection of the top directions improves the update of the parameters because it uses the average of the directions that obtained the best rewards.

**Algorithm 2:** Augmented Random Search

Initialize $\theta$ to 0 , ndeltas: number of deltas , b = number of top deltas, $\alpha$= learning rate, $\nu$=standar deviation of the noise ;
**for** episode = 1, $N$ **do**
  sample $\delta_1, \delta_2 ...$ ndeltas with i.i.d values
  **for** $k = 1$, ndeltas **do**
  get the rewards using the positive and negative observations $r(\pi_{\theta_j+\nu\delta_k})$ and $r(\pi_{\theta_j-\nu\delta_k})$
  Calculate the standard deviation of the rewards
  Sort the directions $\delta_k$ by max($r(\pi_{\theta_j+\nu\delta_k}, r(\pi_{\theta_j-\nu\delta_k})$
  Update the parameters

$$\theta_{j+1} = \theta_j + \frac{\alpha}{b\sigma}\sum_{k=1}^{b}[r(\pi_{\theta_j+\nu\delta_k}) - r(\pi_{\theta_j-\nu\delta_k})]\delta_k$$

  **end for**
**end for**

## V. Results and Discussion

### A. Cart Pole

The results showed in Table I, Table II, Figure 3 and Figure 4, have been obtained by tuning the following parameters of the DQN Naive and DQL with Experience memory:

**Tuning strategy**: We have implemented a 3 fully connected layered architecture with mean square error as our objective function, and Adam as adaptive learning rate optimizer, on top of that we have used the following parameters and aimed to get maximum rewarding actions by optimizing them;

- $\epsilon$**-decay:** As in our implementation, this parameter reduces the $\epsilon$ linearly and as long as $\epsilon$ is high the agent will explore the state space with probability ($\epsilon$) in the context of taking random actions or else it will choose greedy actions with probability (1-$\epsilon$). Our objective is to reduce the bias in our neural network introduced by the max() in Q-equation and maximize our reward at the same time, we tuned this parameter to get an optimal value to achieve our objective.
- **Learning Rate** $\alpha$ **and Episodes** $ep$**:** Both of these parameters are related; the general rule of thumb is while training your neural network if you keep the learning rate low you need to set the no. of iteration (episodes) to an appropriately high value and vice versa. The first approach will require more computational power and second will cause overfitting so to find an optimal value for both of the parameters we tuned these parameters in the aforementioned order aiming to reduce error and get reward maximizing actions.
- $\epsilon$**-min:** In our implementation, $\epsilon$ value becomes $\epsilon$-min if it's less than $\epsilon$-min, now keeping $\epsilon$ high will result in agent learning a randomized policy hence random actions and it won't converge and by keeping it very low we could end up biasing our neural network. For instance, in our cartpole environment, if the maximizing action is moving left most of the time, our neural network will be trained with this action and will always favour the outcome of that action or vice versa. To handle that, we tuned this parameter to find an optimal value that could reduce the bias and favour reward maximizing action.
- **Batch:** For experience replay, batch size and the quality (in context of the maximum reward) of the extracted batch of memories determine the overall learning and performance of our neural network, again to reduce the bias introduced by choosing only left maximizing/not action or right, and as the selection of batch of experience tuples from the memory is purely random so by finding the appropriate batchsize of memories(to be selected), we can optimize our neural network, for instance, let's suppose we have a batch size of 64 and we ended up with the majority of bad experiences (in context of low

reward) then our agent will perform badly as these64 memories will be used to start the learning process. On the other hand, if we have a large batchsize let's say512 we could up end with an appropriate proportion of good experiences that would let the agent perform better.

Our strategy revolves around, minimizing the error, reducing the bias introduced by max() and selecting reward maximizing actions.

TABLE I: **DQN (Naive) - Cart Pole**

| $\epsilon$**-decay** | $\alpha$ | $ep$ | $\epsilon$ **-Min)** | **Fig** |
|---|---|---|---|---|
| 1e-6 | 0.001 | 5000 | 0.01 | 3a |
| 1e-5 | 0.0001 | 10000 | 0.01 | 3b |
| 1e-5 | 0.001 | 5000 | 0.001 | 3c |
| 1e-5 | 0.001 | 5000 | 0.01 | 3d |

TABLE II: **DQN (Experience memory) - Cart Pole**

| **Batch** | $\epsilon$**-decay** | $\alpha$ | $ep$ | $\epsilon$**-min** | **Fig** |
|---|---|---|---|---|---|
| 512 | 1e-5 | 0.001 | 2000 | 0.001 | 4a |
| 256 | 1e-5 | 0.001 | 2000 | 0.001 | 4b |
| 128 | 1e-5 | 0.0001 | 3000 | 0.001 | 4c |
| 64 | 1e-5 | 0.001 | 2000 | 0.01 | 4d |

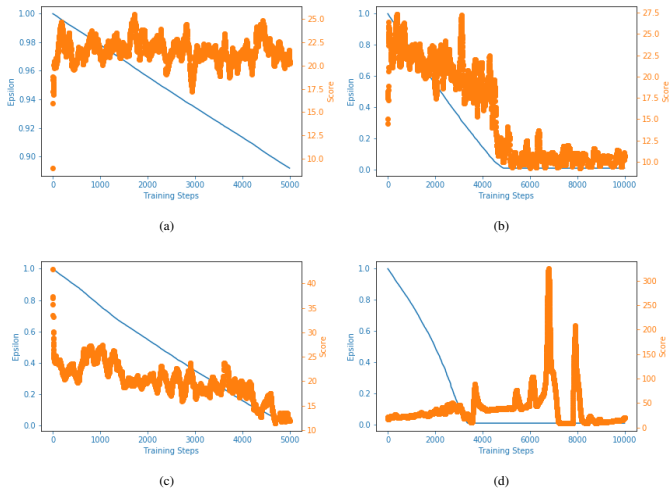



Fig. 3: **DQN (Naive) - Cart Pole**

**Fig 3a:** As we reduce $\epsilon$ at a low rate (1e-6) our network keeps searching for optimal Policy for maximizing the total reward and we observe the fluctuations in our total score. We don't observe a sudden drop in our graph due to the slow rate of decay of $\epsilon$ which enables the agent to go greedy actions.
**Fig 3b:** If we keep the decay of $\epsilon$ at higher rate (1e-5) and provide higher no. of iterations (10000) we observe sudden drop in our scores w.r.t to $\epsilon$ the reason being when $\epsilon$ value becomes the $\epsilon$ minimum which in our case is 0.01 the agent's jumps through state space become more restricted which results in getting caught in a local minimum.
**Fig 3c:** The overall trend of graph is decreasing in nature but as the minimum value of epsilon is set to 0.001 so we don't observe sudden drops as the agent keeps searching for optimal policy and based on the value of epsilon it either goes for greedy action or random one.

**Fig 3d:** Most of the reasons are same as mentioned above but one more thing to add as we are using the same network for evaluating and then choosing the max actions, this part $\max_{a'} Q^*(s', a')$ in our Q-function approximator (eq.1) introduces bias which only favors max actions that our Agent explores through space and with decreasing $\epsilon$ it gets caught in local minimum and our score drops. After running many empirical tests, we have found these set of values for our parameters the most optimum ones. We also observed increasing trend in our score by increasing the no. of iterations but it is computationally very expensive.

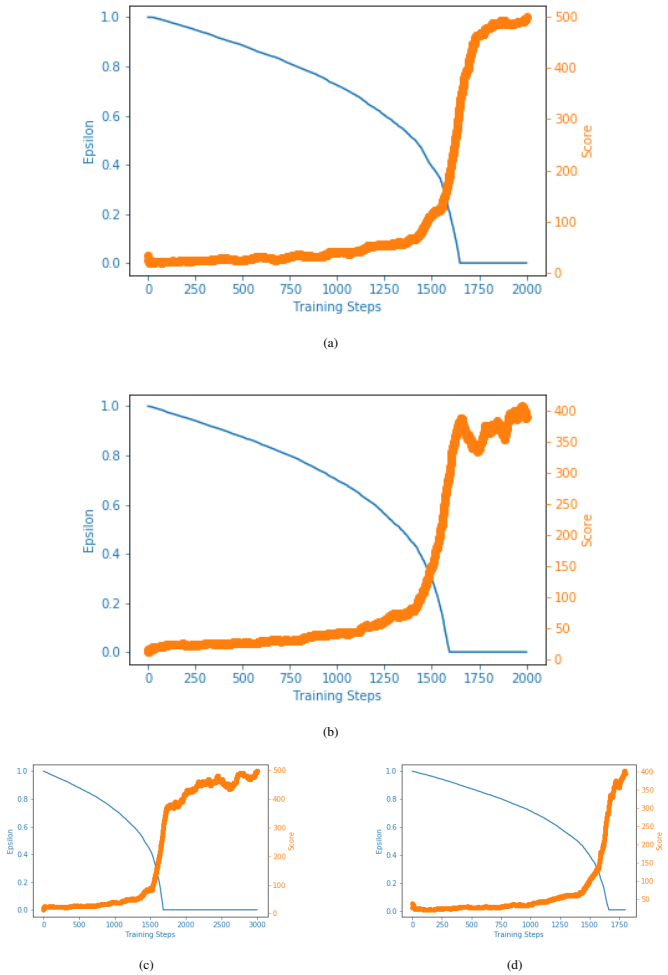



Fig. 4: **DQN (Experience replay) - Cart Pole**

**Fig 4a**, **Fig 4b**, **Fig 4c:** As our agent stores the memories so by increasing the batch size which corresponds to the batch of memories to be used for training the neural network; our agent starts with a low score but soon achieve a high score. The reason being the agent will explore the state space for the first 512 iterations and will choose the maximizing reward actions and store the respective experience tuples $(\phi_t, a_t, r_t, \phi_{t+1})$ after that it will randomly choose a batch of size=512 experience tuples for training the network. This approach reduces the bias and breaks the correlations between consecutive samples and we don't observe unwanted

oscillations as the network learning is averaged over many previous states. In the case of Fig 4b by decreasing the batchsize, we can observe slight drops and oscillations in the score after the $\epsilon$-min has reached the reason being the batch of memories used for training the agent might have majority of left or right actions as with $\epsilon$=$\epsilon$-min, the agent will mostly go for greedy approach and will start to favor either left or right actions but as we accumulate more memories and randomly select them the agent gets back on the increasing trend as this time our agent would have batch of reward maximizing diverse actions to trained with and this behaviour goes on. In the case of Fig 4c, The reason is same as the Fig 4b but now we have more iterations so our agent accumulates more memories.

**Fig 4d:** 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 which is why we observe the increasing trend. After running many empirical tests, we have found these set of values for our parameters the most optimum ones as we get consistently increasing trend and no abrupt oscillations.

### *B. Bipedal Walker*

#### *1) DDPG*

For the DDPG algorithm, after a lot of experiments, we decided to expose in Figure 5 five of them that seem to us interesting to discuss. Each of those experiments required a minimum time of three hours on our laptops, we thus decided to fix a small upper bound value of parameters episodes, maximum of steps, batch and hidden layers, in order to reduce the time computation and have acceptable results. The scores over the episodes of training were obtained by tunning the following parameters:

- $\gamma$: Discount factor 0.99 for all experiments as we want the agent to consider the total rewards and not only immediate rewards;
- Maximum number of steps per episode, which was fixed at 300 to reduce the time computation. This parameter has no influence on the convergence of the algorithm as it represents the maximum steps that the walker can make.
- **ep:** Number of episodes representing the number of iterations. The value of this parameter may influence a lot on the goodness of final results.
- **b:** batch size, which value may affect the convergence as it represents the minimum number of observations to collect before learning.
- **ALR:** Neural network learning rate of the Actor in order to confirm the importance of the actions prediction accuracy obtained through gradient descent algorithm with the value of this learning rate.
- **CLR:** Neural network learning rate of the Critic in order to confirm the importance of the rewards prediction accuracy obtained through gradient descent algorithm with the value of this learning rate.
- **L2r:** L2 regularization for the critic.
- **HLA:** Hidden layers of the actor.
- **HLC:** Hidden layers of the critic.

TABLE III: **DDPG Bipedal Walker**

| ep | b | ALR | CLR | L2r | HLA | HLC | Fig |
|---|---|---|---|---|---|---|---|
| 3000 | 128 | 1e-4 | 3e-4 | 1e-4 | (256) | (256,256,128) | 5a |
| 1500 | 512 | 1e-4 | 3e-4 | 1e-4 | (256) | (256,256,128) | 5b |
| 3000 | 128 | 1e-3 | 3e-4 | 1e-4 | (256) | (256,256,128) | 5c |
| 1500 | 256 | 1e-4 | 3e-4 | 1e-4 | (128) | (128,64,128) | 5d |
| 1500 | 256 | 1e-4 | 3e-4 | 1e-4 | (256) | (256,128,64) | 5e |

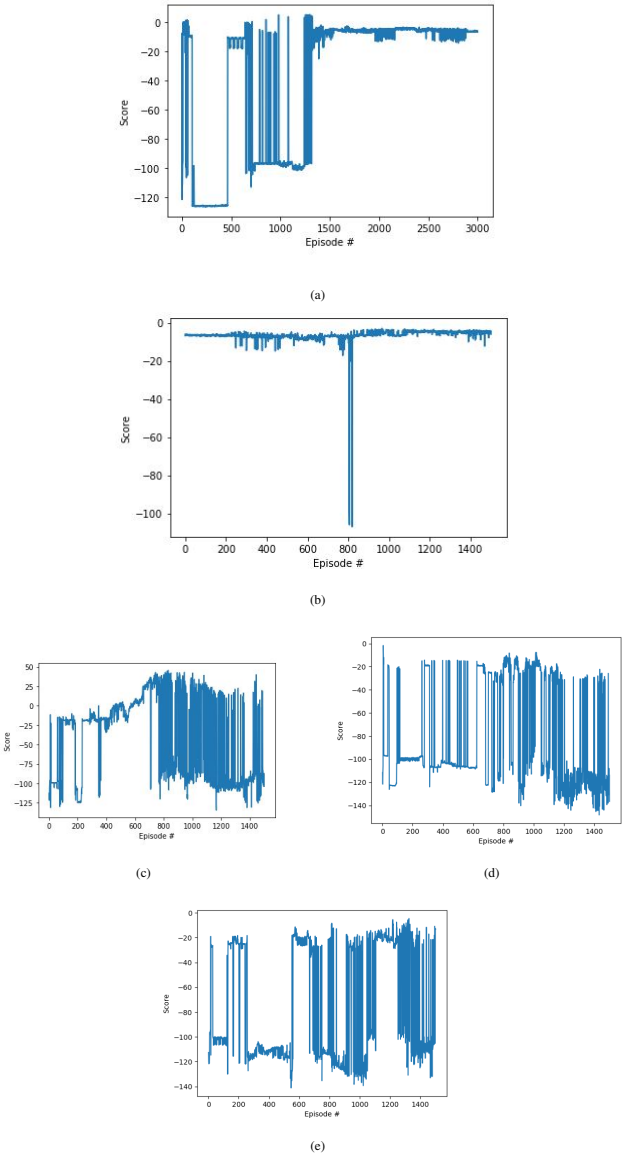



Fig. 5: **DDPG BipedalWalker**

As we can see, the results obtained are for now not satisfying. Some parameters affect more the final results of the model. One of the key parameters that showed a significant difference in the results is the batch size of the data from the replay buffer, the stability of the scores increases when the batch size increases and the other parameters are fixed.

We also investigated the effects of the actor and critic learning rates and the complexity of their models. We found that the actor and critic learning rates have less effect on the environment, while the complexity of the model given by the number of hidden layers and neurones in those layers,

influence a lot the training of the agent.

**Fig 5a:** The walker starts exploring new different actions but it get stuck after a certain number of episodes.
**Fig 5b:** In this graph we can see that the bipedal, in order to maximize its reward decides to remain stationary since falling would represent a high cost, but it is not capable of exploring new actions that may let it explore other optimal solutions.
**Fig 5c:** After changing the learning rate of the actor, we can see that the Walker start explore new actions faster in comparison to the other trainings, however it is unstable and it fails to the optimal goal which is a score of 300.
**Fig 5d:** In this experiment, just like in experiment for Fig 5e, we tried to detect any influence that the complexity of the Neural Network models of the Actor and the Critic may have on the training. We set a less complexity by setting one single hidden layer with 128 neurons for the Actor, and 3 hidden layers for the Critic with respectively 128, 64, 128 units. We can observe an overall worsening of the score when at each episode, meaning that the Actor and the Critic are too much generalized to catch the real complexity of the action space in order to make a good action-value prediction.
**Fig 5e:** In this experiment we set a complexity of the models slightly higher than the one set in experiment of Fig 5d. We set one single hidden layer of 256 neurons for the Actor, and 3 hidden layers for the Critic with respectively 256, 128, 64 units. We observe an overall score at each episode, better than the one observed in Figure 5d, meaning that the Actor and the Critic catch a little bit more the real complexity of the action space in order to make a good action-value prediction.

*2) Augmented Random Search*

For this algorithm we decided to observe the training behavior when we increase or decrease the following parameters:

- The number of steps, fixed at 600, for each experiment as we don't want more steps of the walker for computational resources issues;
- The number of episodes, fixed at 2000, as it was part of the default and suggested value for this environment;
- Learning rate $\alpha$ of the parameters in order to check any influence of such parameter on the convergence of the expected score;
- Noise factor $\nu$ noise added to the parameters to explore different reward outcomes and calculate the finite differences ;
- Deltas $\delta$ number of variations of random noise that generates each training step, each one of these will be run two times;
- Best deltas $\delta^*$ is the number of best paths to consider (the paths that give the best results). these deltas are used to update the weights

The best experiment found so far, is the experiment which the expected score over the number of episodes converge to the maximum expectation as showed in Figure 6. This experiment was done by the people that furnished the implementation, and tuned the parameters $(\alpha, \nu, \delta, \delta^*) = (0.2, 0.03, 16, 16)$

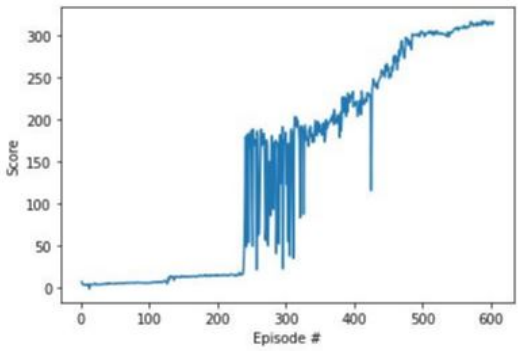



Fig. 6: Best result for Augmented Random Walk

Thus our tunning parameters experiments consisted in assigning deviation from these values and analyze the behaviour through plotting of the convergence of the scores over the episodes. The figures below expose the experiments we did with the values of the parameters assigned.
First of all is important to notice that the ARS is more efficient than the DDGP reaching a score of 200 at 300 episodes when tuned properly and with less hyper parameters which make things easier and more efficient.
The first parameter we tested was the learning rate, the effect of increasing and decreasing this value is the same as in the other models, as we can see in figure 7a , when the learning rate is low the scores are stable but very low because it needs more iterations to arrive to a optimal solution. On the other hand, when the learning rate is high as in 7b , the scores become unstable since it cannot converge to a optimal value.

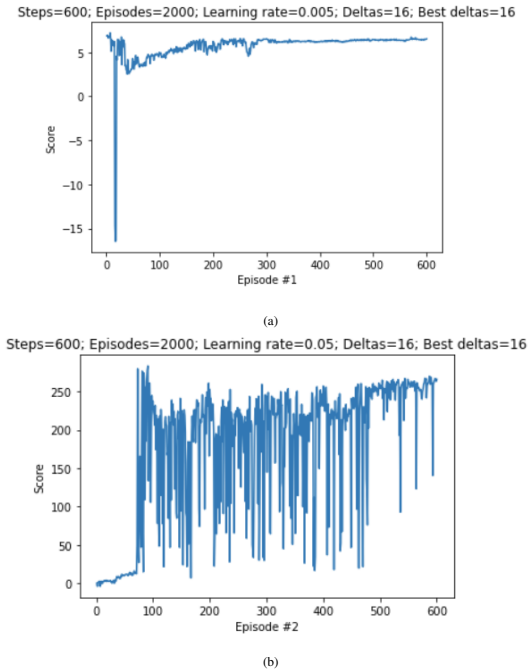
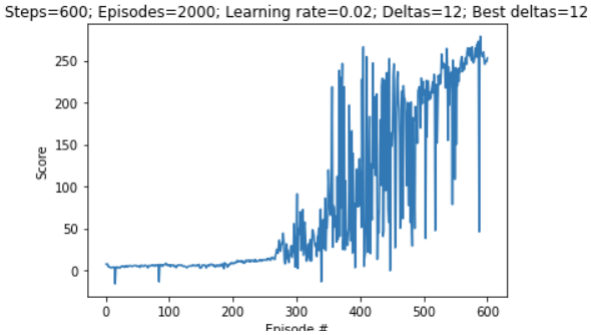
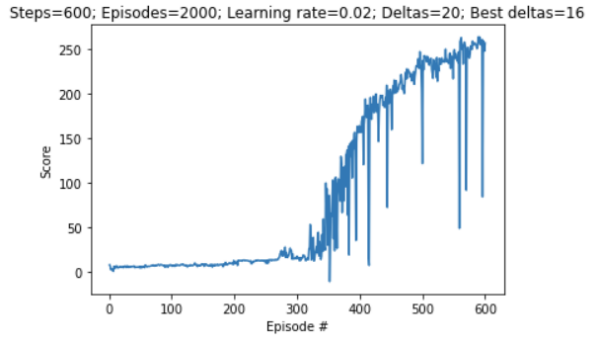



Fig. 7: **Augmented Random Search BipedalWalker**

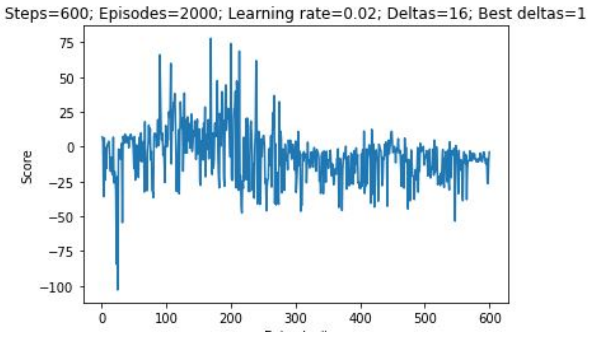



(a)

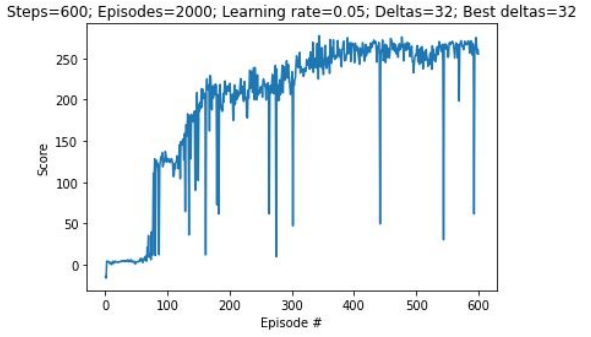



(b)

Fig. 8: **Augmented Random Search BipedalWalker**

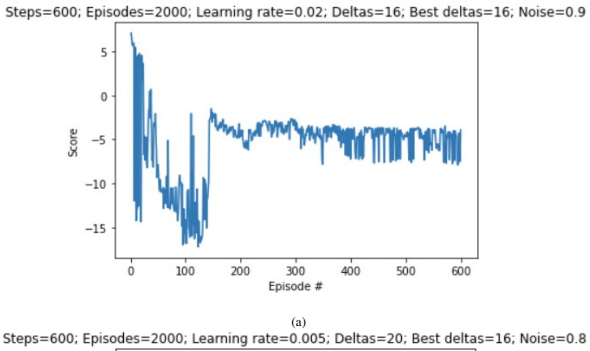



(a)

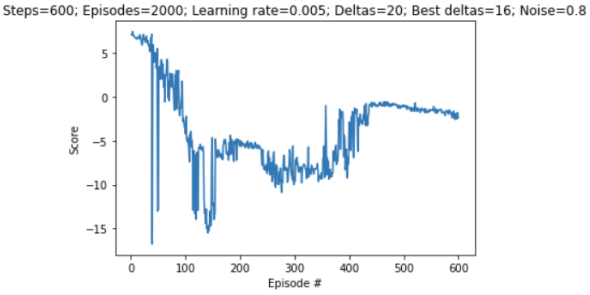



(b)

Fig. 9: **Augmented Random Search BipedalWalker**




(a)




(b)

Fig. 10: **Augmented Random Search BipedalWalker**

Then we explored the effect of the number of deltas and the number of best deltas to use, One would think that with more deltas the results would be better, however we can see in 9b that trying with 32 deltas, it starts getting good scores earlier, but it cannot reach the 350 score, probably because the learning rate is smaller than the reference one, so it takes longer to reach to the same score, in 8a with 12 deltas and 8b the score convergence is the same, reaching a score of 250 after 600 episodes. As well, we can see in 9a that choosing only one best delta is a bad idea, even if it explored 16 different variations, updating the weights with only the best variation, it doesn't allow the model to reach a good score, this shows one of the added values of the augmented random search in comparison with the basic random search

Finally, in 10a and 10b we changed the values of the noise, we see that when having a high value, even with two different number of deltas, the value of the parameters don't allow the model to have a high score.

## VI. Conclusion

DQN (Naive) learns from a single example (cart pole) where our agent iterates through thousands of steps and these iterations are discarded each time because we don't store them in our memory buffer. Our neural network is a function approximator in a high dimensional space for our low dimensional input vectors, each time agent iterates through the state space it starts from a random point in that high dimension and as long as we keep the $\epsilon$ high it explores and we get good results until we reach the $\epsilon$-min where we experience sudden drop in our agent's performance. By doing parametrization we observed that by decaying the $\epsilon$ at a low rate we were able to get good longterm results. On the other

hand, DQN (Experience Replay), is based on training our network on randomly selected experiences from the memory buffer hence each experience is used in the weights updates which allow greater data efficiency, breaks the correlations between consecutive experience samples hence reduces the variance of updates (we don't see any sudden drops in our agent's performance). By doing parametrization we observed an overall increasing trend of our agent's performance for all the parameters. After trying with this simple environment in a discrete space, we wanted to extend it on more complex environments in a continuous space.

Using DDPG on the Bipedal Walker with many parameters and a continuous space, makes it a difficult optimization problem because of the training time to test each one of them while having others fixed to see the different behavior of the agent. The results we obtained remain non optimal since the agent should complete a 300 score in less than 1500 steps while our best trained agent is getting a max score of 43 using the configuration in the third row of Table III. However we can see that there is a decay of score after 700 episodes, where the bipedal learns how to start walking but fails to keep the movement after the first steps. Further tunning needs to be done in order to get other possible results.
DDPG with many hyper parameters to tune and with a high computation time cannot achieve the same results as the augmented random search, we can say that a basic random search can be used to train policies and achieve better performance than complex neural network policies with a simple algorithm with no more than 3 parameters to tune to have a good result. Finally we can say that hyper-parameter tuning is a very important task to do in order to understand the right behaviour of a model, the more stable the model is to small variations of the parameters, the better it is, since algorithms with this lack of robustness cannot be integrated in more realistic applications.

## REFERENCES


[1] Dr. Volodymyr Mnihl. Playing Atari With Deep Reinforcement Learning. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf, 2013.

[2] Dr. Tamis Achilles van der Laan. Consolidated Deep Actor Critic Networks, 2015: https://pdfs.semanticscholar.org/234ba0b5e18d6505136e15bb845ad20e3677904b.pdf

[3] CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING, Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra. Google deepmind 2016: https://arxiv.org/pdf/1509.02971.pdf

[4] BipedalWalker Repository on Udacity github: https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-bipedal

[5] OpenAI Gym Environments and Documentation https://gym.openai.com/

[6] OpenAI SpinningUp documentation https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[7] Simple random search provides a competitive approach to reinforcement learning, Mania .H, Guy .A, Recht B. https://arxiv.org/pdf/1803.07055.pdf