Learning an agent to walk using Reinforcement Learning

Viktorija Mijalcheva, Ana Davcheva

December 2021

Abstract

Reinforcement learning offers one of the most general frameworks to take traditional robotics towards true autonomy and versatility. Deep Reinforcement Learning methods have been successfully applied in many environments and used instead of traditional, optimal and adaptive control methods for some complex problems. However, applying deep reinforcement learning to partially observed environments is still a big challenge. When an agent is not informed well of the environment, it must recover information from the past observations. In this paper are analysed different approaches for training a robot to walk in the Gym[12] environment - Bipedal Walker [13]. The main focus will be to implement and compare the results from various algorithms such as: Proximal Policy Optimization (PPO2)[9], Augmented random search (ARS)[17] and Deep Q-Network (DQN)[8].

1 Introduction

Embodied cognition is the theory that an organism's cognitive abilities is shaped by its body. It is even argued that an agent's cognition extends beyond its brain, and is strongly influenced by aspects of its body and also the experiences from its various sensorimotor functions. Evolution plays a vital role in shaping an organism's body to adapt to its environment; the brain and its ability to learn is only one of many body components that is co-evolved together.

While evolution shapes the overall structure of the body of a particular species, an organism can also change and adapt its body to its environment during its life. For instance, professional athletes spend their lives body training while also improving specific mental skills required to master a particular sport [20]. In everyday life, regular exercise not only strengthens the body but also improves mental conditions [4]. We not only learn and improve our skills and abilities during our lives, but also learn to shape our bodies for the lives we want to live.

Reinforcement Learning is one of the three main machine learning paradigms along with Supervised and Unsupervised Learning. It is the closest kind of

learning demonstrated by humans and animals since it is grounded by biological learning systems. It is based on maximizing cumulative reward over time to make agent learn how to act in an environment. In many reinforcement learning tasks, the goal is to learn a policy to manipulate an agent, whose design is fixed, to maximize some notion of cumulative reward. Continuous action spaces impose a serious challenge for reinforcement learning agents. While several off-policy reinforcement learning algorithms provide a universal solution to continuous control problems, the real challenge lies in the fact that different actuators feature different response functions due to wear and tear (in mechanical systems) and fatigue (in biomechanical systems)[7].

Formally, Reinforcement Learning is learning a policy function (strategy of the agent) $\pi : S \rightarrow A$ which maps inputs (states) s ϵ S to outputs (actions) a ϵ A. Learning is achieved by maximization of the value function $V^{\pi}(s)$ (cumulative reward) for all possible states, which depends on policy π. In this sense, it is similar to unsupervised learning. However, the difference is that the value function $V^{\pi}(s)$ is not defined exactly unlike unsupervised learning setting. It is also learned by interacting with the environment by taking all possible actions in all possible states.

In spite of tremendous leaps in computing power as well as major advances in the development of materials, motors, power supplies and sensors, there is still lack of the ability to create a humanoid robotic system that even comes close to a similar level of robustness, versatility and adaptability as biological systems. Among the key missing elements is the ability to create control systems that can deal with a large movement repertoire, variable speeds, constraints and most importantly, uncertainty in the real-world environment in a fast, reactive manner. Most baseline tasks in the RL literature test an algorithm's ability to learn a policy to control the actions of an agent, with a predetermined body design, to accomplish a given task inside an environment. This makes it highly suitable for systems which are difficult to control using conventional control methodologies, such as walking robots. Traditionally, RL has only been applicable to problems with low dimensional state space, but use of Deep Neural Networks as function approximators with RL has shown impressive results for control of high dimensional systems. This approach is known as Deep Reinforcement Learning (DRL). A major drawback of DRL algorithms is that they generally require a large number of samples and training time, which becomes a challenge when working with real robots. Therefore, most applications of DRL methods have been limited to simulation platforms.

In this thesis, Bipedal Locomotion is investigated through BipedalWalker-Hardcorev3 [12] environment of open source GYM library by DRL[2]. In the OpenAI Gym's BipedalWalker-v3 environment, the reward system is defined by incrementally gaining 300 points if the finish line is reached, losing 100 points if robot falls over, and gradually losing points based on the amount of control effort exerted from the motors. Solving the problem is defined by collecting over 300 reward points for 100 consecutive episodes. Various algorithms such as: Proximal Policy Optimization (PPO2)[9], Augmented random search (ARS)[17] and Deep Q-Network (DQN)[8] are tried in order to get the agent walking.

Furthermore, these algorithms will be deeply examined and compared with their outcomes on the agent training.

2 Related work

Bipedal walking is one of the most challenging and intriguing controls problem in recent years. It poses a variety of challenges which make it difficult to apply classical control methodologies to it.In [15], Rastogi explained the four main reasons which makes these environments difficult, such as non-linear and changing dynamics, multi-variable system and uncertainty in the model. Continuous action spaces impose a serious challenge for reinforcement learning agents. While several off-policy reinforcement learning algorithms provide a universal solution to continuous control problems, the real challenge lies in the fact that different actuators feature different response functions due to wear and tear (in mechanical systems) and fatigue (in biomechanical systems). In [7], there is a soultion proposed enhancing the actor-critic reinforcement learning agents by parameterising the final layer in the actor network. This layer produces the actions to accommodate the behaviour discrepancy of different actuators under different load conditions during interaction with the environment. To achieve this, the actor is trained to learn the tuning parameter controlling the activation layer (e.g., Tanh and Sigmoid). The learned parameters are then used to create tailored activation functions for each actuator. Here, Deep Deterministic Policy Gradient (DDPG)[18] is used as a method in solving continuous state problems and its use is shown as a helpful tool for dealing with this issue. All of the models tried in this paper are also used in [13]. Later, it will be made a comparison between the both papers. The results in this paper are not enough to conclude on a superior neural network for all RL problems, because there are other factors such as DRL algorithm, number of episodes, network size etc. However, networks are designed to have similar sizes and a good model requires to converge in less episodes. In addition, it is possible to conclude that Transformers can be an option for partially observed RL problems. Today, all subfields of Deep Learning suffers from lack of analytical methods to design neural networks. It is mostly based on mathematical and logical intuition. Until such methods are developed, it seems that we need to try out several neural networks to get better models, which is the case in our work. In [1], a comparison between the performance of two well known reinforcement learning (RL) algorithms is presented. CACLA (Continuous Actor-Critic Learning Automaton)[21] and SPG (Sampled Policy Gradient)[22] are two RL algorithms designed to work with continuous input and action spaces. In [1], CALCA performs better than SPG, finishing the line around the episode 1600-1700. [14] is actually a book that has all of the explanation of the joints, ankles, body of the bipedal robot. Bipeds are multi-input, multi-output systems that are both continuous and discrete. While in single support, the system operates in a continuous fashion, as soon as the support leg switches, there is discreteness, as well. There have been robots such that their control is based on their certain joints and/or certain points on

their structure track pre-specified trajectories. One of the advantages to this approach is that the controller is relatively simple since all the trajectories are known, but if there is a slight change in the shape of the robot or the terrain on which the robot walks, the controller may not work any longer and it will usually require supplemental control in addition to trajectory tracking. The major problem we are currently dealing with is the noisiness of the velocity signals, which limits us to use low damping parameters and makes the roll control more challenging. A powerful filter can be quite helpful. Robustness measurement of a biped control algorithm can be carried out in more extensive ways such as applying different forces in different directions at different times while the robot is in different states. In [5] is shown that allowing a simple population-based policy gradient method to learn not only the policy, but also a small set of parameters describing the environment, such as its body, offers many benefits. By allowing the agent's body to adapt to its task within some constraints, it can learn policies that are not only better for its task, but also learn them more quickly.

3 Methods

The reinforcement learning environment poses a variety of obstacles that need to be addressed and potentially to make trade-offs among them. In RL, an agent is supposed to maximize rewards (exploitation of knowledge) by observing the environment (exploration of environment). This gives rise to the exploration-exploitation dilemma which is an inevitable trade-off. Exploration means taking a range of acts to benefit from the consequences, which typically results in low immediate rewards but high rewards for the future.

The first idea to work with a robot movement came from NeurIPS: AI for prosthetics. In this competition, the task is developing a controller to enable a physiologically based human model with a prosthetic leg to walk in requested directions with varying speeds. The provided material is a human musculoskeletal model and a physics-based simulation environment where you can synthesize physically and physiologically accurate motion. Recent advancements in material science and device technology have increased interest in creating prosthetics for improving human movement. Designing these devices, however, is difficult as it is costly and time-consuming to iterate through many designs. This is further complicated by the large variability in response among many individuals. One key reason for this is that our understanding of the interactions between humans and prostheses is not well-understood, which limits our ability to predict how a human will adapt his or her movement. Physics-based, biomechanical simulations are well-positioned to advance this field as it allows for many experiments to be run at low cost. Recent developments in using reinforcement learning techniques to train realistic, biomechanical models will be key to better understand the human-prosthesis interaction, which will help to accelerate development of this field. We tried all the similar environments here, but in the end they were closed because the competitions were over, but in the continuation it will be

explained what we researched and what we tried.

Formally, we tried to build a function a: S→A from the state space S to the action space A. Each element s of the state space is represented as a dictionary structure that includes current positions, velocities, accelerations of joints and body parts, muscles activity, etc. The action space represents muscle activations. The objective was to find such a function that reward throughout the episode is maximized.

For training the agent, we tried the DDPG[18] algorithm from baselines[6] source code. We imported the Actor and Critic and created the models. Then for the training we used the train environment (imported from baselines.ddpg.training):

- training.train(env=env, evalenv=evalenv, paramnoise=paramnoise, actionnoise=actionnoise, actor=actor, critic=critic, memory=memory)

We also tried to train RL with Deep Learning and made some neural networks, but the environment was very complicated and required too much processing power. Then we did a research of the solutions from the competition, where we also found that the most used algorithms are DDPG[18] and PPO[9]. All of the participants used servers on which they trained the data and also in the early phase of training, participants reduced the search space or modified the environment to speed up exploration using frameskip, exploration noise, etc.. A common statistical technique for increasing the accuracy of models is to output a weighted sum of multiple predictions. This technique also applies to policies in reinforcement learning, and many teams used some variation of this approach: ensemble of different checkpoints of models, training multiple agents simultaneously, or training agents with different seeds. For this environment we lost about 3 months of research, of which 1 month was needed to be able to set the environment, trying on all operating systems. Then we waited a long time to train ordinary models that lasted a long time and were often unsuccessful due to insufficient memory. So from here, and from talking to other students who are interested in this field, we searched for similar environments where we can use the models we have created.

The most similar environment that we found and which will be further used is the OpenAI Gym[12] enviornment called BipedalWalker-v3[13]. The Bipedal Walker series of environments is based on the Box2D [3] physics engine. The environment provides a model of a five-link bipedal robot, depicted in Figure 1. The robot state is a vector with 24 elements: $\theta, \cdot x, \cdot y, \omega$ of the hull center of mass (white), θ, ω of each joint (two green, two orange), contacts with the ground (red), and 10 lidar scans of the ground (red line). The action space, then, is a four element vector representing the torque command to the motor at each of the four joints. This robot has to navigate through the environment, which is a randomly seeded ground with an uneven surface. The lidar scans can help detect this unevenness, though they are most useful in the hardcore version of the environment where obstacles and stairs are present. To find the optimal policy, which maps the robot states to the robot actions yielding the highest reward, various algorithms are tried such as: PPO2, ARS and DQN.

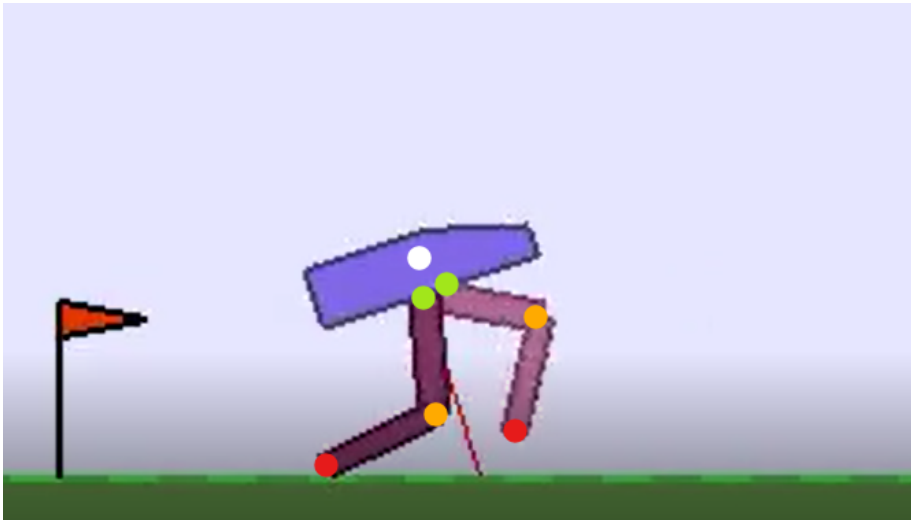

Figure 1: The robot in the BipedalWalker-v3 environment. Red flag indicates starting position.

The robot's stimulus/environment is a grassy field. The robot's decision will be how to align its legs to stand up and walk. Depending on the robot's decision, the outcome will either be a reward (a few points for walking forward) or a punishment (-100 points for falling down). The robot knows:

- How fast it's going
- How its feet touch the ground
- How far away the ground is from its head (using a sensor of the kind found in self-driving cars)
- Whether it's been rewarded or punished for its actions

The first algorithm that will be tried and explored is Augmented Random Search[17]. ARS explores the policy parameter space, as opposed to the action and state space, by taking a set of parameter samples with zero mean Gaussian noise and executing rollouts for each of those samples. The parameters of the model θ take the form of an $n \times m$ matrix where n is the number of action variables and m is the number of state variables. A state observation is matrix multiplied to produce a desired action to take. In ARS, several parameter noise matrices δ are sampled, resulting in $\theta + v\delta$ and $\theta - v\delta$ pairs where θ is the current parameterization, v is a step size scalar that determines how much noise is introduced to the model, and δ a random noise matrix of the same dimensions as θ. If in total there are h number of random noise matrices δ then there are 2h matrices, each of which are used to conduct rollouts. These noisy parameterizations all lie within a hypersphere of a specified radius around the current parameterization θ.

Rewards for each rollout are collected, and their standard deviation ω is computed and used to normalize the final update step. The $hv\delta$ matrices are then sorted in descending order of maximum reward using the criteria $max(r_{\theta+v\delta_i}, r_{\theta-v\delta_i})$ for each parameterization pair i, where $r_{\theta+v\delta_i}$ denotes the reward received from the rollout using a model parameterization $\theta + v\delta_i$, and $r_{\theta-v\delta_i}$ denotes the reward received from the rollout using a model parameterization $\theta - v\delta_i$. The top m pairs are then used to update θ using the following update rule:

$$\theta \leftarrow \theta + \frac{\alpha}{m\sigma} \sum_{i=1}^{m} [(r_{\theta+v\delta_i} - r_{\theta-v\delta_i}) * \delta_i]$$

where α represents the learning rate.

As the agent learns and the average reward size increases, the summation over $r_{\theta+v\delta_i} - r_{\theta-v\delta_i}$ will increase as well, resulting in proportionately larger step sizes as the policy improves overtime. To mitigate this inconsistency in step size, the learning rate α is normalized by the standard deviation of the rollout rewards ω. This normalization is the primary difference between basic random search (BRS) and augmented random search (ARS).

The next algorithm that will be tried to learn the agent to walk is proximal policy optimization version 2 called PPO2 [9]. Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games, to 3D locomotion, to Go. With supervised learning, cost function can be easily implemented, gradient descent can be additionally run on it, and excellent results will be the outcome with relatively little hyperparameter tuning. Proximal policy optimization[9] strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. The variant of PPO uses a novel objective function not typically found in other algorithms:

$$L^{CLIP}(\theta) = \hat{\mathrm{E}}_t[min(r_t(\theta)\hat{\mathrm{A}}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{\mathrm{A}}_t)]$$

- θ is the policy parameter
- $\hat{\mathrm{E}}_t$ denotes the empirical expectation over timestamps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{\mathrm{A}}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

This objective implements a way to do a Trust Region update which is compatible with Stochastic Gradient Descent, and simplifies the algorithm by removing the KL penalty and need to make adaptive updates. In tests, this algorithm has displayed the best performance on continuous control tasks. This

paper uses the PPO2 implementation by baselines[6]. The release of baselines includes scalable, parallel implementations of PPO, PPO2 and TRPO which both use MPI for data passing. Both use Python3 and TensorFlow. PPO2 represents a GPU-enabled implementation of PPO. This runs approximately 3X faster than the current PPO baseline. In addition, baselines released an implementation of Actor Critic with Experience Replay (ACER), a sample-efficient policy gradient algorithm. ACER makes use of a replay buffer, enabling it to perform more than one gradient update using each piece of sampled experience, as well as a Q-Function approximate trained with the Retrace algorithm.

The last algorithm that will be used is a Deep Q-Network also called DQN[16]. This algorithm is a variation of the classic Q-Learning algorithm. Markov Decision Process (MDP)[19] is a sequential decision making process with Markov property. It is represented as a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$. Markov property means that the conditional probability distribution of the future state depends only on the instant state and action instead of the entire state/action history, so it is regarded as memoryless. In MDP setting, the system is fully observable which means that the states can be derived from instant observations; i.e., $s_t = f(o_t)$. Therefore, the agent can decide an action based on only instant observation o_t instead of what happened at previous times [11]. MDP consists of the following:

- **State Space** $\mathcal{S}$ - A set of all possible configurations of the system.
- **Action Space** $\mathcal{A}$ - A set of all possible actions of the agent.
- **Model** $T : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ - A function of how environment evolves through time, representing transition probabilities as $T(s^{'}|s, a) = p(s^{'}|s, a)$ where $s^{'} \in \mathcal{S}$ is the next state, $s \in \mathcal{S}$ is the instant state and $a \in \mathcal{A}$ is the action taken.
- **Reward Function** $R : \mathcal{S} \times \mathcal{A} \to R$ - A function of rewards obtained from the environment. At each state transition $s_t \to s_{t+1}$, a reward r_t is given to the agent. Rewards may be either deterministic or stochastic. Reward function is the expected value of reward given the state s and the action taken a, defined by: $R(s, a) = E[r_t|s_t = s, a_t = a]$
- **Discount Factor** $\gamma \in [0, 1]$ - A measure of the importance of rewards in the future for the value function.

A key concept related to MDPs is the Q-function, $Q^\pi : S \times A \to R$, that defines the expected future discounted reward for taking action a in state s and then following policy π thereafter. According to the Bellman equation, the Q-function for the optimal policy (denoted Q*) can be recursively expressed as:

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma max_{a'} Q^*(s', a')]$$

where $0 \leq \gamma \leq 1$ is the discount factor that defines how valuable near-term rewards are compared to long-term rewards. Given Q$*$, the optimal policy, $\pi*$

, can be trivially recovered by greedily selecting the action in the current state with the highest Q-value: $\pi* = argmax_a Q*(s,a)$.This property has led to a variety of learning algorithms that seek to directly estimate Q , and recover the optimal policy from it. Of particular note is Q-Learning.

Deep Q-Learning (DQN) makes 3 primary contributions to the classic Q-Learning algorithm: (1) a deep convolutional neural net architecture for Q-function approximation; (2) using mini-batches of random training data rather than single-step updates on the last experience; and (3) using older network parameters to estimate the Q-values of the next state. The pseudocode for DQN is shown on Figure 2. The deep convolutional architecture provides a general purpose mechanism to estimate Q-function values from a short history of image frames (in particular, the last 4 frames of experience). The latter two contributions concern how to keep the iterative Q-function estimation stable.

Algorithm 1 Deep Q-learning with experience replay

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

Figure 2: Pseudo code for Deep Q-Network (DQN) from [11]

DQN keeps a large history of the most recent experiences, where each experience is a five-tuple (s, a, s', r, T), corresponding to an agent taking action a in state s, arriving in state s' and receiving reward r.T is a boolean indicating if s' is a terminal state. After each step in the environment, the agent adds the experience to its memory. After some small number of steps ([11] used 4), the agent randomly samples a mini-batch from its memory on which to perform its Q-function updates. Reusing previous experiences in updating a Q-function is known as *experience replay* [10]. However, while experience replay in RL was typically used to accelerate the backup of rewards, DQN's approach of taking fully random samples from its memory to use in mini-batch updates helps the samples from the environment that otherwise can cause bias in the function approximation estimate.

4 Results

The trained ARS model in this paper uses seperate classes for Hyperparameters and Normalization. Hyperparameters are variables that define how fast our robot's brain will work. These parameters require a lot of tuning in order to get a good model that will enable the robot to learn efficently. In our case multiple hyperparametras changes were done in order to optimize the outcome. The HP that were selected as best for this training are:

- number of steps: 1000
- episode length: 2000
- learning rate: 0.2
- number of deltas: 16
- number of best deltas: 16
- noise: 0.03
- seed: 1

An uncertainty in the agent movements in this algorithm is present due to the random search for actions. For that particular reason, we want to normalize all of the robot's behaviours (the size of its step, speed of walking, etc.) so that we can directly compare parameters with different units. To do this, we converted the robot's movement parameters to z-scores using the known formula:

$$(currentposition - meanposition)/standarddeviation$$

After training the model, a policy evaluation is required in order to see how much did the agent learn and to update the behavioural model accordingly. The update is done using the following formula:

$$\theta+ = \frac{\alpha}{(\delta^* \times \sigma_r) \times s}$$

Each of the arguments are explained bellow:

- θ is the weight assigned to a particular behaviour. From a neuroscience perspective, it is something like a long-term potentiation or inhibition (strengthening or weakening) of a neural connection. If the behaviour is beneficial, then we want to strengthen the neurons driving that behaviour (and vice versa).
- *alpha* is the learning rate, the speed at which the model learns. If the model learns too fast, it may overshoot the optimal setup for making a good decision (i.e. making too large of a step and falling over).

- δ^* represents the robot's past experiences. Here, the robot is 'thinking' about the decisions it made in the past, and the optimal deltas - the optimal shifts in strategy it has made.
- σ_r is the sum of the rewards the robot has gained for making specific behaviours of that type before. If the action resulted in the robot making progress towards the goal in the past, it's assigned a positive reward. If the action resulted in the robot falling, it's assigned a punishment (negative reward).
- s represents the step that the agent has made in that moment

Our agent is further represented by a class ARSTrainer which uses the above mentioned hyper parameters, normalizer and policy. This class has two methods: explore and train. The explore function is called after each episode of training. Here we're telling the robot to go forth into this brave new world, try out various strategies according to its policies, and note down the rewards and punishments it gets. The train function consists of the actual learning for the agent. It uses the rewards and punishments that the agent has faced during exploration to update its behaviour for the next strategy that it will try.

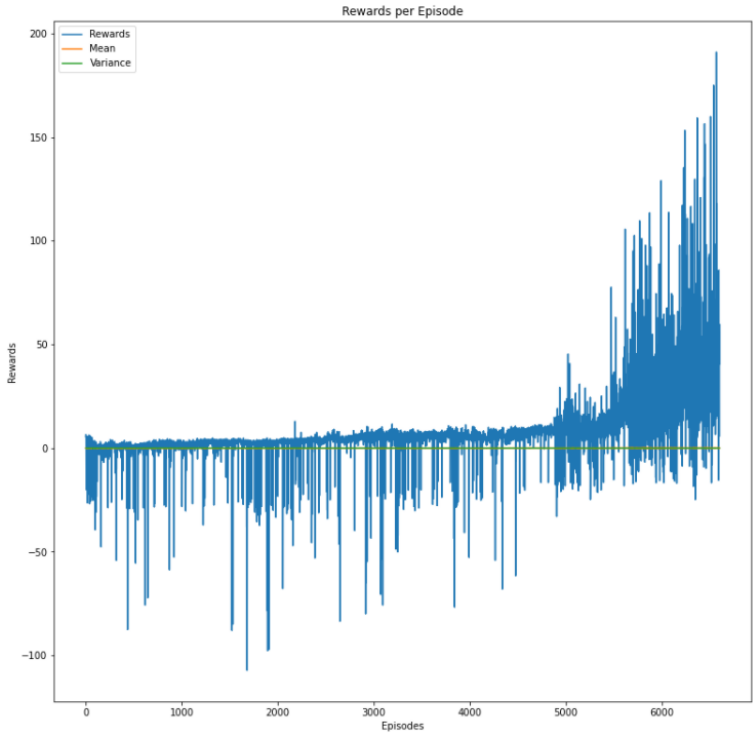

Figure 3: Rewards per episode with the ARS model

Figure 3 represents the rewards the agent has gathered during a training session of 2000 episodes with number of steps set to 200.

From the figure, it can be concluded that the agent using ARSTrainer is learning arbitrarily. The rewards are moving on approximately the same range of rewards between [5, -40] in the episodes until reaching 5000. After that, an drastic improvement is detected showing that the agent got reward bigger than 20. The agent trained with ARS in our paper used 1000 number of steps and 2000 number of episodes. It resulted in having the agent walk with no failing in the last episode, but having the agent fail before reaching the end in the episode before that.

The second trained model is DQN model. The architecture of the model used is almost identical to the architecture we used in the exercises, with several new features. The HP that were selected as best for this training are:

- episode length: 200
- learning rate: 0.01
- gamma: 0.98

The network used for this algorithm is simple Neural Network with 2 hidden layers:

- model.add(Dense(400, inputdim=self.nx, activation='relu'))
- model.add(Dense(300, activation='relu'))
- model.add(Dense(self.ny, activation='linear'))
- model.compile(loss='mse', optimizer=Adam(lr=self.lr))

The results from the rewards are shown in Figure 4.

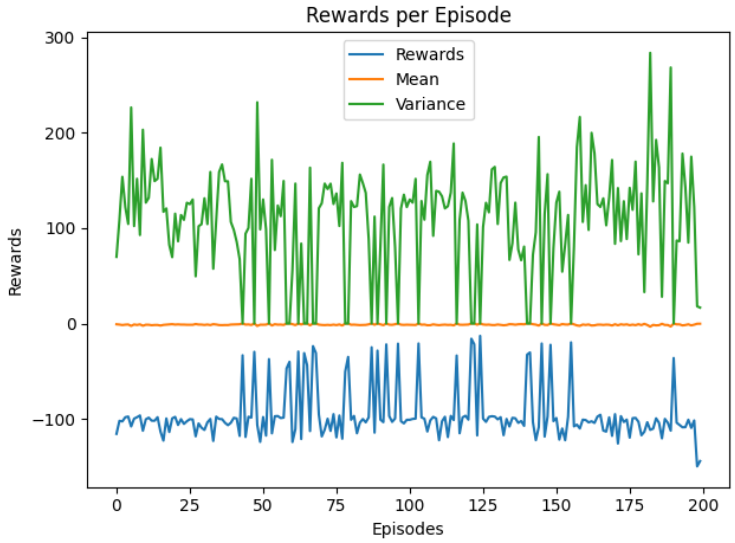

Figure 4: Rewards per episode with the DQN model

From the results it can be concluded that DQN, trained with the given parameters and neural network, is not the best choice. The reason for these results is that DQN needs to be trained with a more complex network, multiple episodes and trained hyper parameters, but at the moment we are not able to because we are limited due to the processing power of computers. However, it is worth mentioning that DQN is not guaranteed to converge because of the instability caused by bootstrapping, sampling and value functional approximation. In order for the policy to converge, we need to ensure the policy is near greedy (epsilon is close to 0) once the value is converge, which could be achieved by playing a large enough number of episodes.

The third trained model is PPO model. Actor Critic approach was used for our PPO agent. It uses two models, both Deep Neural Nets, one called the Actor and other called the Critic. The architecture is shown in Figure 5.

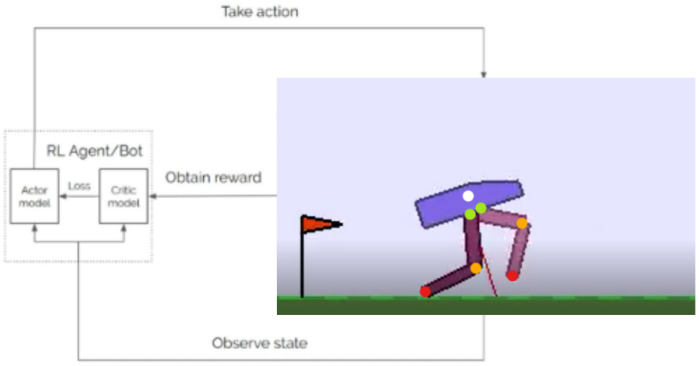

Figure 5: PPO model architecture

The key contribution of PPO is ensuring that a new update of the policy does not change it too much from the previous policy. This leads to less variance in training at the cost of some bias, but ensures smoother training and also makes sure the agent does not go down an unrecoverable path of taking senseless actions.

The HP that were selected as best for this training are:

- episode length: 200
- learning rate: 0.0001
- gamma: 0.99
- alpha: 0.1

The Actor was trained with a Neural Network with 2 hidden layers using RELU as an activation function, and the output layer is using tanh. The used optimizer is Adam. The Critic is using the same hidden layers, and his output layer is trained with linear activation function.

The rewards from PPO model are shown in figure 6.

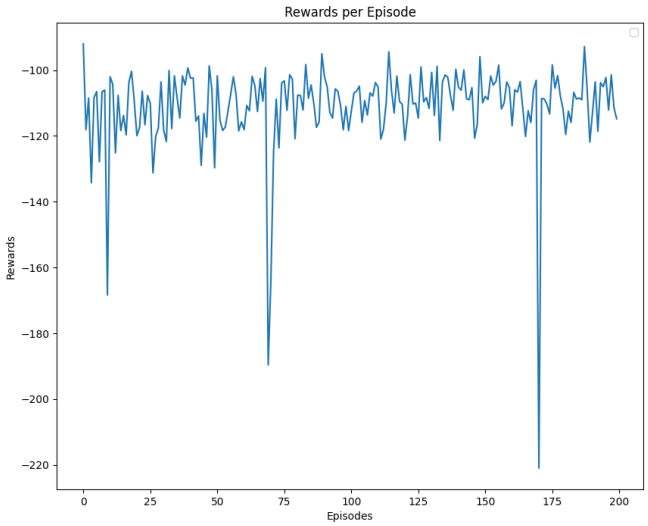

Figure 6: Rewards per episode with the PPO model

After PPO, we trained PPO2 model from baselines, using MlpPolicy. Stable-baselines provides a set of default policies, that can be used with most action spaces. MLP Policy implements actor critic, using a Multi-layer Perceptron (2 layers of 64).

The HP that were selected as best for this training are:

- episode length: 100
- learning rate: 3e-4
- gamma: 0.99

The model was trained with 4000 total timestamps. The results from PPO2 are shown on Figure 7.

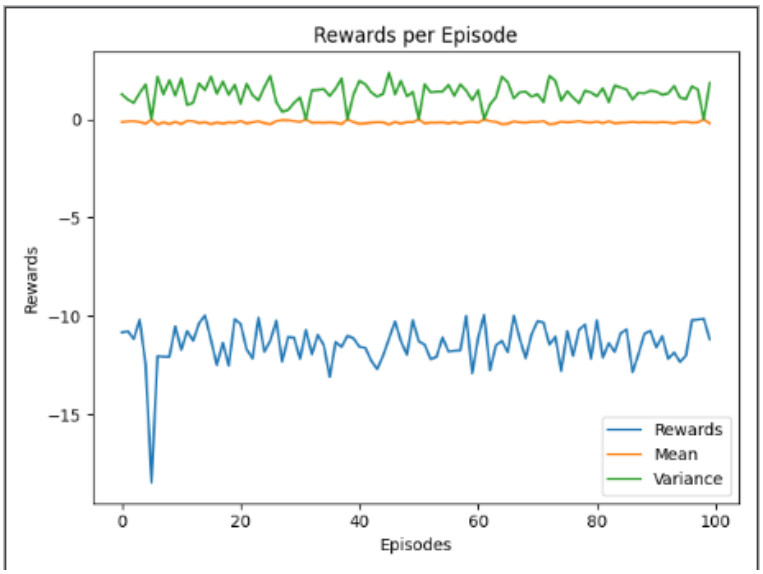

Figure 7: Rewards per episode with the PPO2 model

The last thing that we want to share from results is the hard-coded model. This model is specially built for this environment and that is the obvious reason why this model has the best results. Figure 8 shows the rewards that were gathered per each episode training this hard-coded model.

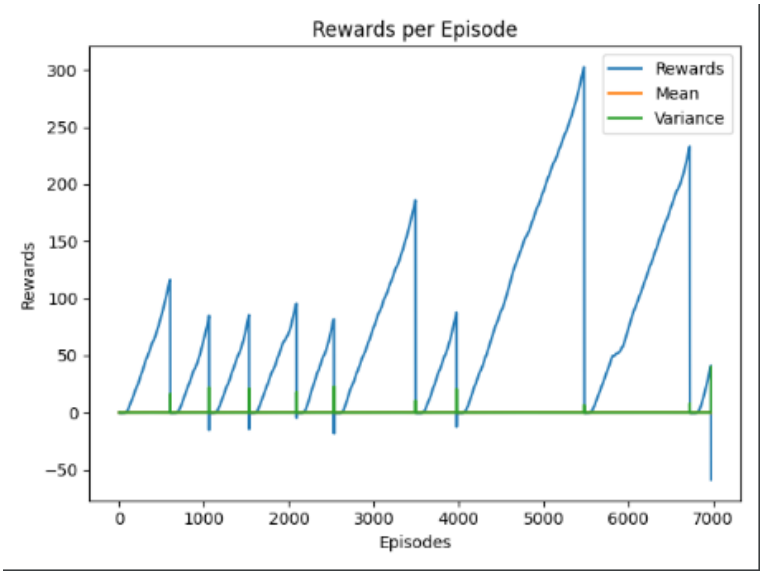

Figure 8: Rewards per episode with the hard-coded model

5 Conclusion

With the results examined from all methods there are conclusions that can be made for each one. The ARS method for training an agent to walk is making the agent learn at random and not that much from past experience. That results in having the curve of rewards going up and down and not being sure if the agent is learning or not. This agent should be trained using this RL method on a bigger number set for the number of episodes, as well as for the number of steps.

From the results, the PPO and PPO2 method are not showing us a significant reward curve from which we can conclude if the agent is learning or not. It gives us uncertainty of the outcome because the rewards curve is going well and in one particular moment it is making a significant mistake. This means that a long hyper-parameter tuning process is required in order to find when does the agent make this mistake and how he should overcome it.

Although these models were not trained on big number of episodes, it can be concluded that the agent trained using the DQN algorithm is showing remarkable and consistent increasing of the reward curve compared to others. On the other hand, the variance of this model is certainly showing a varying curve which is making this algorithm uncertain that it will learn better each time we start learning.

All of the methods are acting particularly exceptional for this environment and should be truly examined in future works with longer episodes numbers. What can be of significant importance is that the last episodes of each method gave high rewards of the agent walking which is making us believe that by the end of each method tried the agent has learned to walk.

There might be a few possible ways to improve this work. First of all, longer observation history can be used to handle partial observably for LSTM and Transformer models. However, this makes learning slower and difficult and requires more stable RL algorithms and fine tuning on hyper-parameters. Secondly, different exploration strategies might be followed. Especially parameter space exploration may perform better since it works better for environments with sparse reward like BipedalWalker. Lastly, advanced neural networks and RL algorithms (specifically designed for environment) may be designed by incorporating domain knowledge.

References

[1] Neda Ahmadi, Diego Cabo, Sudhakaran Jain, and Ruben Kip. Comparing continuous reinforcement learning algorithms using openai.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[3] Erin Catto. Box2d: A 2d physics engine for games. *URL: http://www. box2d. org*, 2011.

[4] Andréa Deslandes, Helena Moraes, Camila Ferreira, Heloisa Veiga, Heitor Silveira, Raphael Mouta, Fernando AMS Pompeu, Evandro Silva Freire Coutinho, and Jerson Laks. Exercise and mental health: many reasons to move. *Neuropsychobiology*, 59(4):191–198, 2009.

[5] David Ha. Reinforcement learning for improving agent design. *Artificial life*, 25(4):352–365, 2019.

[6] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[7] Mohammed Hossny, Julie Iskander, Mohamed Attia, Khaled Saleh, and Ahmed Abobakr. Refined continuous control of ddpg actors via parametrised activation. *AI*, 2(4):464–476, 2021.

[8] Jonathan Hui. RL — DQN Deep Q-network.

[9] Filip Wolski Prafulla Dhariwal Alec Radford John Schulman, Oleg Klimov. Proximal Policy Optimization.

[10] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[12] OpenAI. BipedalWalker-v2 enviornment OpenAI Gym.

[13] Uğurcan Özalp. Bipedal robot walking by reinforcement learning in partially observed environment. Master's thesis, Middle East Technical University, 2021.

[14] Allen S Parseghian. *Control of a simulated, three-dimensional bipedal robot to initiate walking, continue walking, rock side-to-side, and balance*. PhD thesis, Massachusetts Institute of Technology, 2000.

[15] Divyam Rastogi. Deep reinforcement learning for bipedal robots. 2017.

[16] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*, 2017.

[17] Ziad SALLOUM. Introduction to Augmented Random Search.

[18] Hemant Singh. Deep Deterministic Policy Gradient (DDPG).

[19] Wikipedia source. Markov decision process.

[20] Valmor Tricoli, Leonardo Lamas, Roberto Carnevale, and Carlos Ugrinowitsch. Short-term effects on lower-body functional power development: weightlifting vs. vertical jump training programs. *The Journal of Strength & Conditioning Research*, 19(2):433–437, 2005.

[21] Chang Wang, Chao Yan, Xiaojia Xiang, and Han Zhou. A continuous actor-critic reinforcement learning approach to flocking with fixed-wing uavs. 101:64–79, 17–19 Nov 2019.

[22] Anton Orell Wiehe, Nil Stolt Ansó, Madalina M Drugan, and Marco A Wiering. Sampled policy gradient for learning to play the game agar. io. *arXiv preprint arXiv:1809.05763*, 2018.

Here you can see the Github repo.