UCLA CS275 18W Final Report

A Reinforcement Learning Approach for Locomotion

Wandi Cui
University of California, Los Angeles
wandicui@ucla.edu

Ziqi Yang
University of California, Los Angeles
larittic@ucla.edu

Zeyu Zhang
University of California, Los Angeles
zeyu.zhang@cs.ucla.edu

Yunchu Zhang
University of California, Los Angeles
yunchuzhang@ucla.edu

ABSTRACT

In this project, we explored a reinforcement learning approach and an evolution strategy for physics-based character locomotion skills training. The locomotion skills developed for our physics-based character will not only target flat terrain but also more complex environment where the character needs to navigate a sequence of challenging terrain with rough ground, stumps, pitfalls, and stairs. We implemented an evolution strategy and the Asynchronous Advantage Actor-Critic (A3C) algorithm on the BipedalWalker-v2 physical environment provided by OpenAI Gym. Both of two algorithms led to good results with satisfying accumulated rewards. Comparing these two algorithms, we decided that the agent trained by the A3C algorithm is more stable and has more natural and dynamic gaits. We then conducted further experiments on the advanced BipedalWalkerHardcore-v2 environment where obstacles, pitfalls, and stairs are randomly generated. We also explored deeper into the underlying explanation and analysis for the experiment results.

1 INTRODUCTION

Bipedal robots have attracted more and more researchers' attention since 1990s due to the low energy consumption and their human-like walking characteristics. Compared to wheeled and multi-legged robots, bipedal robots can be more adaptive to complex terrains (e.g. uneven terrain, up/down stairs, stumps, pitfalls and etc.); and have more flexible locomotion directions as well as speed range. It is not difficult to imagine that with so many advantages, humanoid robots could be widely used in various fields in the real world. For instance, they can conduct rescue operations in dangerous circumstances like fire, toxic gases or chemicals and explosives as a substitute for human beings. Furthermore, they can also play an important role in the service industry because of its advantages of human-computer interaction.

However, bipedal locomotion is one of the most challenging tasks in robotics. The control problems of walking with dynamic balance are convoluted. Most scholars proposes deterministic and analytic engineering approaches to solve bipedal walking control problems, while some of researchers apply machine learning techniques, such as reinforcement learning to this problem and make similar achievements.

2 RELATED WORK

Both evolutionary strategy and reinforcement learning approaches succeed in addressing complicated control problems even though they hold totally different initial inspirations. Moreover, since deep learning techniques perform excellent in so many fields, some approaches are proposed to combine deep learning methods with reinforcement learning as deep reinforcement learning methods.

Evolutionary strategy can be easily understand as a process of taking samples from groups of individuals and making successful ones produce future generations, which must come from biological evolution. Thus, evolutionary methods solve control problems by evolving the control policy in this way. In [1], the structure of neural network is predefined, and each generation consisting of candidate parameters is evaluated by environment. During the process, the weights of the network are sampled from a multivariate Gaussian distribution, and some candidates are selected to update the distribution to gain the next generation. Paper [10] sticks to different idea with CMAES, which has accomplished numbers of difficult control tasks by evolve both the structure and the weights of the network. A genetic encoding method is introduced to represent the network efficiently and historical markings are used to avoid expensive topological analysis. [11], is an improvement of CMAES assuming that the population of network parameters are drawn from a certain distribution. Its aim is to maximize the expected fitness.

Deep reinforcement learning (DRL) methods create a reinforcement learning framework merging reinforcement learning and deep neural networks. Article [3] summarizes that this kind of measure utilizes deep neural networks to approximate value function, policy, state transition and reward. A large amount of progress has been made in DRL since it was first proposed in [5]. Quite a few methods based on DRL are designed to tackle bipedal locomotion. The approach in [6] describes a successful demonstration of training a bipedal humanoid character to traverse rugged terrain in a two dimensional simulation environment.Then in their paper [7], they expand their work to train a bipedal humanoid character to achieve complex locomotion tasks in a three dimensional environment by building a hierarchical DRL framework. In the meantime, [2] has also presents a method: Distributed Proximal Policy Optimization (DPPO) to complete training a humanoid character navigating through different terrains in a three dimensional environment. In the latest paper [9], the authors come up with a learning algorithm based on the Recurrent Network-based Deterministic Policy Gradient (RDPG). They apply the Long-Short Term Memory (LSTM) to realizing a partially observable Markov Decision Process (POMDP) framework and provide a wonderful solution to the Bipedal-Walker challenge in virtual environments of OpenAI.

3 SIMULATION ENVIRONMENT

In our project, we utilized both evolutionary strategy and reinforcement learning approaches to give solutions to BipedalWalker challenge provided by OpenAI Gym. Gym is a toolkit for developing and comparing reinforcement learning algorithms. In addition, Gym library has designed a variety of environments that we can use to evaluate our algorithms.

First, we will introduce the BipedalWalker task of OpenAI Gym. In this task, the robot agent perceives 24-dimensional observation from the environment. The information includes 10-dimensional Lidar (visual) with range limit, 4-dimensional translational/rotational displacement and velocity of hull, 8-dimensional rotational displacement and velocity of bipeds, and 2-dimensional binary haptic on foot. The environment runs episodically; and episodes terminate when the hull of the agent hits the ground in environment, the agent reaches the goal, or the maximum time limit exceeds. In fact, there are two versions of the task, one is the normal version with slightly uneven terrain; and the other is the hardcore version with different challenging terrains like stairs, stumps and pitfalls. We tried both versions and reported our results in the following parts.

To comprehensively understand our challenge, we summarized the following difficulties that need overcoming:

- The bipedal locomotion task is partially observable
- Due to the shape of the agent's body, its center of gravity is back-swept, which makes it more difficult to keep balance without falling backwards
- Stochastic environment

Since the environment is partially observable, the agent can only perceive the environment in front it. Therefore, the agent has no idea of what happened in the past and what is the whole picture of the environment. For example, in Figure 1 (c), the agent does not know it is over a pitfall since it can only perceive the world in front of it, due to the limitation of the Lidar. Intuitively, the agent will assume all the environment behind it is flat terrain. Therefore, it will make some wrong perceptions as shown in Figure 1 (c) and (d).

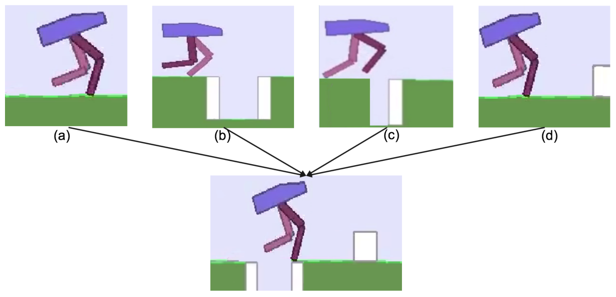

Figure 1: Short-Term Memory with Queue Size 4

In order to address the problem of partial observation, we utilized a short-term memory trick. As shown in Figure 1, we maintain a fixed-size queue to record previous observations, we then combine all the previous observations stored in the queue to form a new observation. The intuition behind this is that the combined observation will help the character to reconstruct the whole picture.

4 METHODOLOGY

In this project, we explored two approaches to solve the BipedalWalker-v2 problem—Evolution Strategy and Reinforcement Learning based on A3C algorithm.

4.1 Evolution Strategy

Evolution Strategy (ES) is an optimization technique that has been known for decades. It rivals the performance of standard reinforcement learning (RL) techniques on modern RL benchmarks (e.g. Atari/MuJoCo), while overcoming many of RL's inconveniences. Briefly speaking, ES is a scalable alternative to Reinforcement Learning with the following advantages:

- It is simpler to implement because it has no need for back-propagation.
- It is easier to scale in a distributed setting.
- It does not suffer in settings with sparse rewards.
- It has fewer hyperparameters.

We came across this method during exploring the possible approaches for BipedalWalker-v2. We found it extraordinarily convenient to implement and simple to understand, specifically regarding this problem. Intuitively, the ES algorithm can be best summarized by "guess. check and optimize". Facing a learning problem, we start with random parameters (a parameter vector w) as the initial weights for the model. For each step, we take the current w and generate a population of n slightly different parameter vector $w1, w2, \ldots, wn$ by jittering w with gaussian noise, in which the probability density function p of the Gaussian random variable z is given by:

$$p_G(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

where z represents the grey level, μ the mean value and σ the standard deviation.

The n candidates will then be evaluated independently by running the corresponding policy network on the target environment. After the evaluation, we'll get the accumulated rewards in each case, which will serve as the weights for them to get the weighted sum of $w1, w2, \ldots, wn$ as the new w. Normally, the weight for each wi is proportional to the total reward. In this way, it is ensured that the parameter vector keeps moving towards the better results as the more successful candidates have higher weights to contribute to the next update. Mathematically, the procedure can be viewed as equivalent to estimating the gradient of the expected reward in the parameter space using finite differences in n random directions.

The simplicity of ES can be best demonstrated by the following example of optimizing a quadratic function using ES:

```
import numpy as np

def f(w):
  return -np.sum((w - solution)**2)

alpha = 0.001        # learning rate
sigma = 0.1          # noise standard deviation
pop_size = 50        # population size
iteration = 500      # iteration times
```

```
w = np.random.randn(3)   # initial guess

for i in range(iterartion):
  N = np.random.randn(pop_size, 3)
  R = np.zeros(pop_size)
  for j in range(pop_size):
    w_try = w + sigma*N[j]
    R[j] = f(w_try)
  A = (R - np.mean(R)) / np.std(R)
  w = w + alpha / (pop_size*sigma) *
     np.dot(N.T, A)
```

To further illustrate the ES algorithm, there are two points to clarify. First, it is important to note that the term "evolution" here means the abstract process of tweaking the parameter vector, which can be seen as sampling a population of individuals and allowing the more successful individuals to dictate the distribution of future generations. Another point is the black-box optimization. In ES, we forget entirely the existence of the agent, the environment, the neural networks, or the interactions being taken. It approaches the problem through a pure mathematical way, in which it tries to find the best parameter vector to get the highest total rewards by repeatedly adjusting and revising its current values. With no assumptions about the structure of the relationship between the parameter vector and the reward (except that there is a definite way to evaluate it), the optimization process can be completely decoupled from the specific target problem.

Actually, ES can be seen as a modified version of Reinforcement Learning where the agent's actions are to emit entire parameter vectors using a gaussian policy. Compared to traditional Reinforcement Learning, ES is mainly different in the four aspects:

- ES injects noise directly in the parameter space while RL normally injects noise in the action space and uses backpropagation to compute the parameter updates.
- ES does not need backpropagation but only requires the forward pass of the policy, which makes both the code and the network simpler.
- ES is highly parallelizable because it only requires workers to communicate a few scalars between each other instead of synchronizing entire parameter vectors.
- ES provides structured exploration by, similar to Q-learning, using deterministic policies to achieve consistent exploration.

For BipedalWalker-v2, we implemented the ES model based on evostra, a Python package for Evolution Strategy.

4.2 Reinforcement Learning Algorithm

In this project, we also utilized a reinforcement learning algorithm to learn the walking controller for our BipedalWalker. The first part of this section will introduce how we formalize the motion control problem into a reinforcement learning problem, and the rest of this section will explain the details of our algorithm and implementation.

The goal of reinforcement learning is to find out an optimal policy $\pi_{\theta}(\boldsymbol{a}|\boldsymbol{s})$ which can maximize the total rewards. A policy $\pi_{\theta}(\boldsymbol{a}|\boldsymbol{s})$ is a function of state $\boldsymbol{s}$ with a set of parameters $\boldsymbol{\theta}$. Given a state, the policy π_{θ} will determine what action should be taken given current state. Since both policy and motion controller perform similar function (i.e., given a state determines what action should be taken in the next step), instead of learning a motion controller, we can learn an optimal policy by utilizing reinforcement algorithm. Therefore, we can learn the walking controller for the BipedalWalker by utilizing a reinforcement learning algorithm.

In Figure 2, a whole picture of reinforcement learning architecture is given[8]. There are two main modules: the Environment module and the Agent module. These two modules interact with each other. Specifically, the agent can perform an action based on the observation of current environment state, which in turn will change the environment to another state.

Technically, the observation not only contains the perception from the environment but also contains the information of the agent itself. The information of a agent, for example, may contain position and dynamic information of the agent. The environment will also return a *Reward*, which is a real number, to the agent. The *Reward* is a function of state and action, which evaluate the behavior of the agent on a specific state. Given a state $\boldsymbol{s}_t$ and an action $\boldsymbol{a}_t$ (action $\boldsymbol{a}_t$ is taken at state $\boldsymbol{s}_t$), the reward could be represented as $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$.

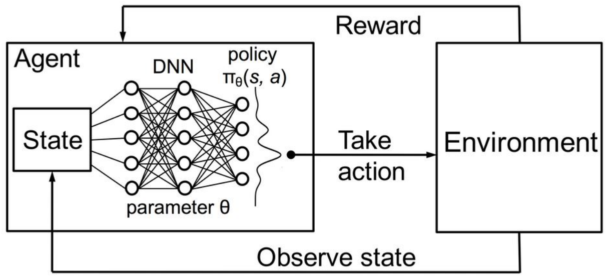

Figure 2: Reinforcement Learning Architecture

As shown in Figure 2, the core part of the Agent module is a deep neural network which takes observation as input and outputs an action. In other words, this neural network represents the policy $\pi_{\theta}(\boldsymbol{a}|\boldsymbol{s})$ which determines what action should be taken in the next step given current state. Therefore, in order to find the optimal policy, we need to find out a set of appropriate parameters $\boldsymbol{\theta}$ for the deep neural network.

The technique that we improve the policy (i.e., the neural network parameters) is called *Policy Gradient.* A generic policy gradient workflow is shown in Figure 3. Initially, we start with a random policy π_{θ_0} (i.e., random parameters in the deep neural network). Then we apply such random policy π_{θ_0} to the agent and generate a bunch of state-action pairs $(\boldsymbol{s}_i, \boldsymbol{a}_i)$. We then fit the learning model with these state-action pairs and evaluate the behavior of the agent based on the reward function $r(\boldsymbol{s}_i, \boldsymbol{a}_i)$. Once we have the evaluation of the agent running policy π_{θ_0}, we can improve the policy based on the evaluation and update the policy to π_{θ_1}. Then we replace π_{θ_0} with π_{θ_1} and run the iteration again. In every iteration, we improve the policy and then apply the new updated policy in the next iteration. The intuition behind this is that, hopefully, the policy will converge to an optimal state.

The goal of reinforcement learning is to find an optimal policy that can maximize overall rewards. Since we use a deep neural network to represent the policy, we want to find a set of parameters $\boldsymbol{\theta}$ that can maximize the total rewards. Therefore, we have following

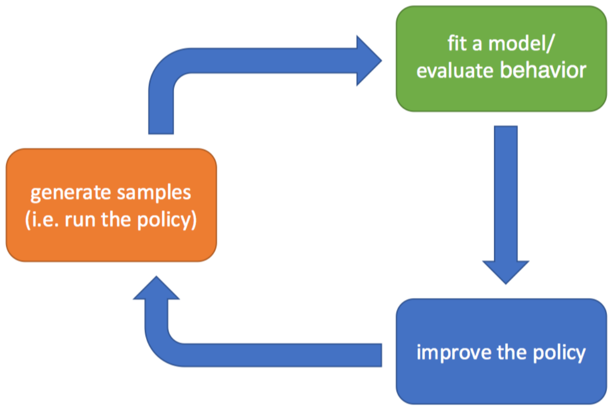
fit a model/ evaluate behavior
improve the policy
generate samples (i.e. run the policy)

Figure 3: Policy Improvement Workflow

equation,

$$\theta^* = \arg\max_{\theta} J(\theta)$$

where θ^* denote the optimal policy parameter and $J(\theta)$ represents the expectation of total rewards under policy π_θ which is defined by the following equation,

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\sum_t^T r(s_t, a_t)\right]$$

in which τ denotes a series of state-action pairs (s_i, a_i) generated by policy π_θ, and T denotes the number of state-action pairs.

Since the problem we want to solve is in a continues space (i.e., both of the state space and the action space are continues), we cannot calculate the expectation directly if we do not know the underlaying distribution. In order to calculate the expectation, Monte Carlo approximation is adopted. Thereby, we can rewrite $J(\theta)$ in the following form,

$$J(\theta) \approx \frac{1}{N}\sum_i^N \sum_t^{T_i} r(s_{i,t}, a_{i,t})$$

where N is number of times sampling from policy π_θ, and T_i denotes the number of state-action pairs in sample i. We then take the derivative of $J(\theta)$, so that we can have the following equation,

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^N \left(\sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta \left(a_{i,t}|s_{i,t}\right)\right)\hat{Q}_{i,t}$$

where $\hat{Q}_{i,t}$ is the estimate of expected reward if action $a_{i,t}$ is taken at state $s_{i,t}$, and it could be calculated based on following formula,

$$\hat{Q}_{i,t} = \left(\sum_{t'=t}^{T_i} r(s_{i,t'}, a_{i,t'})\right)$$

Once we have the derivative of $J(\theta)$, policy could be updated based on the following formula,

$$\theta' \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

where α is the learning rate, θ is the current policy parameters, and θ' is the updated policy parameters.

The intuition behind the policy gradient is similar to the idea of gradient descent/ascent, i.e., take the derivative of the objective function and approach to the local optimal closer and closer during every iteration. A problem in the policy gradient is that we cannot calculate the objective function $J(\theta)$ accurately. The way we calculate the objective function is based on Monte Carlo approximation. Specifically, we use $\hat{Q}_{i,t}$ as the estimation of expected reward if action $a_{i,t}$ is taken at state $s_{i,t}$. Since $\hat{Q}_{i,t}$ calculates the expected reward only based on a single sample under policy π_θ (i.e., the summation of all rewards after $(s_{i,t}, a_{i,t})$ in sample i), it may cause high variance in the estimation. Therefore, in our project we applied Actor-Critic algorithm, which is an advanced variant of policy gradient, for a better policy evaluation.

In Actor-Critic algorithm, instead of estimating the expected reward based on the $\hat{Q}_{i,t}$ function, it utilizes a deep neural network to estimate the expected reward. The neural network that estimates the expected reward is called *Critic*, whereas the neural network that performs policy gradient is call *Actor*. The Actor updates its parameters (i.e., updates the policy) based on the estimation of expected reward from the Critic.

The Critic neural network performs supervised regression learning algorithm, the objective function is described as following,

$$\mathcal{L}(\phi) = \frac{1}{2}\sum_i \left\|\hat{V}_\phi^\pi(s_i) - y_i\right\|^2$$

where ϕ represents the parameters in the Critic neural network, $\hat{V}_\phi^\pi(s_i)$ is the value function which indicates the expectation of the total reward at state s_i, and y_i represents the target value.

More specifically, y_i is calculated based on the following formula,

$$y_{i,t} \approx r(s_{i,t}, a_{i,t}) + \gamma\hat{V}_\phi^\pi(s_{t+1})$$

where t is the timestamp and time t and γ is the discount factor for the value function. The intuition behind the discount factor is that better to get rewards sooner than later.

Algorithm 1: Online Actor-Critic Algorithm

```
repeat
    1. Take action $a \sim \pi_\theta(a|s)$, and get $(s, a, s', r)$
    2. Update value function: $\hat{V}_\phi^\pi \leftarrow r + \gamma\hat{V}_\phi^\pi(s')$
    3. Evaluation: $\hat{A}^\pi(s,a) = r(s,a) + \gamma\hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$
    4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s)\hat{A}^\pi(s,a)$
    5. Update policy: $\theta' \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
until policy converged;
```

The whole Actor-Critic algorithm is shown in Algorithm 1, where $\hat{A}^\pi(s,a)$ indicates the advantage of the action a (i.e., how much better taking action a at state s than average). What's more, in the Actor-Critic algorithm we replace the $\hat{Q}_{i,t}$ function with the $\hat{A}^\pi(s,a)$.

In order to accelerate the training process, we apply an asynchronous advantage actor-critic (A3C) [4] method to accelerate the Actor-Critic algorithm. The idea behind the A3C algorithm is that we can run multiple copy of Actor-Critic algorithm simultaneously, and each copy of Actor-Critic will help each other by sharing global parameters. A simple example of A3C architecture is shown in Figure 4. There are 4 workers (i.e., W_0, W_1, W_2, W_3), each worker runs a copy of Actor-Critic algorithm. Workers share their knowledge by utilizing the *Golbal_Net* where workers share their learned parameters. If a worker finish one round of its iteration, it will upload its parameters to the global net. Once the global net receives the parameters from the worker, it will update its own parameters. The worker then will fetch the latest parameters from the global net to its local networks and start a next iteration.

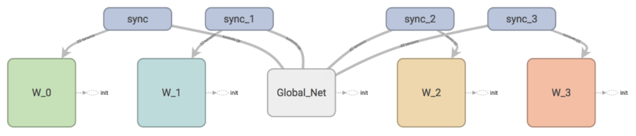

Figure 4: A Simple Example of A3C Architecture

5 EXPERIMENTS

In our project, we train a simple model "bipedalwalker" and a complicated one named bipedalwalker-hardcore. Also, in bipedalwalker, we use evolution strategy and A3C algorithm to train that model and use A3C to train bipedalwalker-hardcore.

The results we present are divided into two part. In first part, we show the result in simple mode–"bipidalwalker" with evolution strategy, and then show the result in both simple and complicated mode with A3C algorithm. In bipedalwalker-hardcore, we train the "walker" to learn from 5 different classes of terrain randomly. In A3C, all networks are built and trained using pytorch. Source code is available at https://github.com/TooSchoolForCool/CS275-Parkour-Go.

5.1 Evolution Strategy Experiments

In order to estimate the Evolution Strategy model, we record every episode's reward to see the trend of this model.

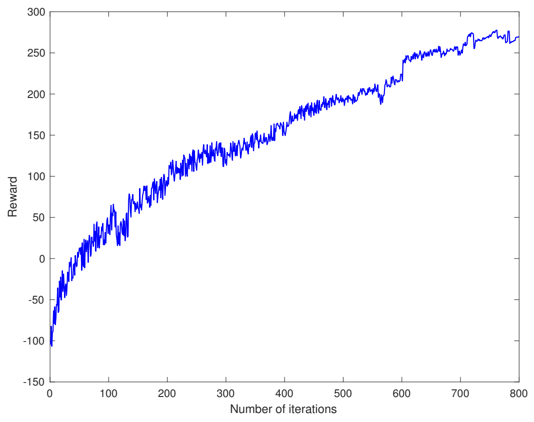

Figure 5: Evolution Strategy reward

In Figure 5, we could see that the curve is unstable and has a little vibration but increases gradually. Within limited episode, this algorithm could converge finally.

The instability exposed in the experiments is mainly due to the randomized noise injection mechanism of Evolution Strategy. Because it is by directly injects Guassian noise in the parameter space, the noise has more influence on the overall optimization process, which may lead to various levels of performance. As we conducted more experiments, we found that the ES algorithm may not always converge to a good results. In few cases, the agent might fall into a self-locked state where it keeps a steady but locked stance with no further advancement. We suspect that it is also due to the accidentally unwanted noise injection. It requires more experiments to understand the exact reason. All in all, we found Evolution Strategy simple and intuitive to implement, but not so stable or reliable for this problem. Therefore, we focused more on the Reinforcement Learning method based on A3C algorithm.

5.2 A3C Experiments

In order to estimate our model, we use the logging package to record every episode's reward and mean reward. Also, we record test result in video.

The training time dominated by the complexity of environment that "walker" need to learn, which means that the more complicated the environment is, the more exploration work that the model need. Because in hardcore mode, we need the policy that could deal with all these terrain classes,thus, the model's training time was very long. In contrary, the simple mode "bipedalwalker" only need about 7 hours' training and converge quickly. The plot below is the reward in each episode: In Figure 6, remove the noise point, we

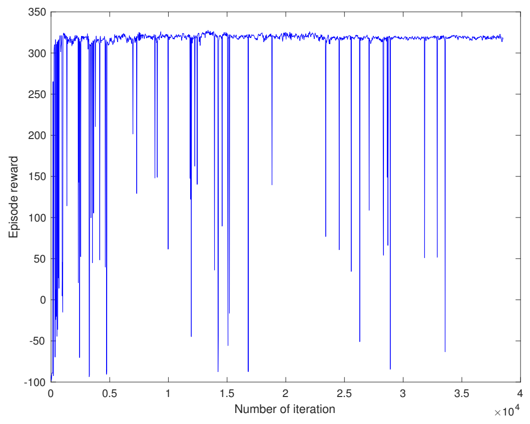

Figure 6: Simple mode Episode reward

could conclude that nearly every epoch, the "walker" could learn and run to the end and learning result is very good. In Figure 7, we could conclude that A3C algorithm have a good performance in this simple mode. It converge quickly and steady.

After a long-time training, we could see the "Hardcore" plot below:

In Figure 8, we could see that although episode reward has a concussion pattern, through depth blue point, episode reward has a increasing tendency totally.

In Figure 9, the mean reward confirm our assumption above, the reward increases continuously, which shows that our model could learn how to finish that task step by step. Since this "hardcore" mode is very difficult, maybe we need more time to train it to converge. After longer training, it could have a better performance in this difficult mode.

In Figure 10, with the comparison of two algorithms in same mode(upper is A3C,lower is Evolution), we could see that two

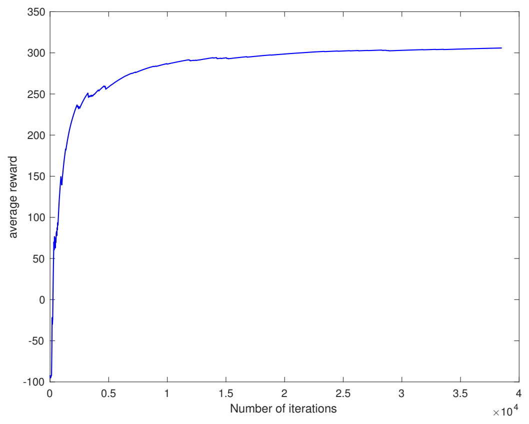

Figure 7: Simple mode Average reward

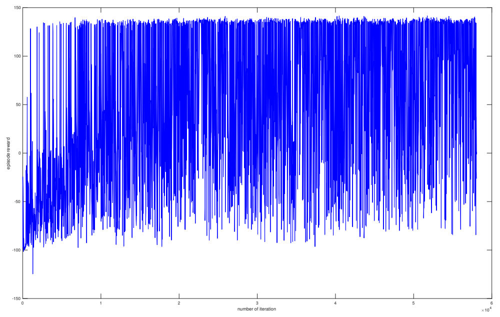

Figure 8: Hardcore mode Episode reward

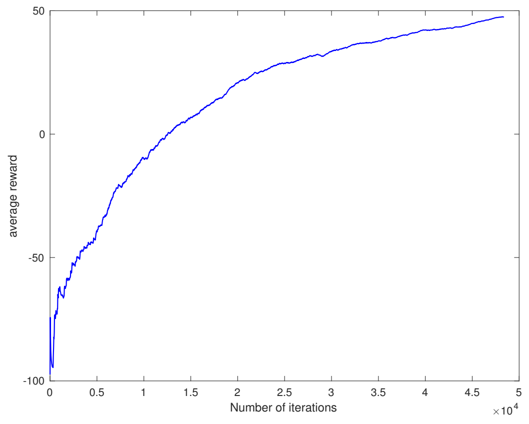

Figure 9: Hardcore mode Average reward

algorithms use different way to balance the "walker" when running in same simple terrain. The evolution strategy use front-balance to balance their heavy head and make itself a slightly backwards. But with A3C algorithm, it shows a good double-balance position and make itself lean forward. In Figure 11, it is very tough for "walker" to pass the stair. And the gesture is not nature and seems to failed down immediately. In Figure 12, when the "walker" meets the block task, he gets the best gesture to pass it, but maybe he needs more time of training to make a balance when pass it.

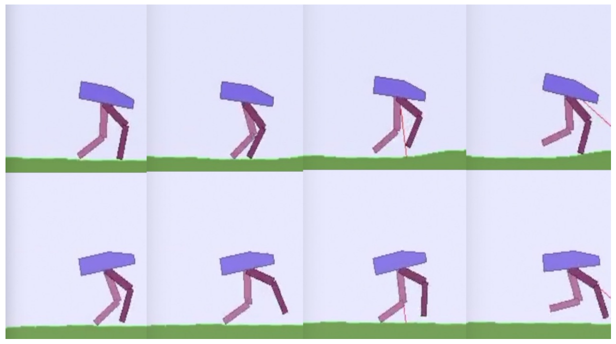

Figure 10: Evolution and A3C in simple mode

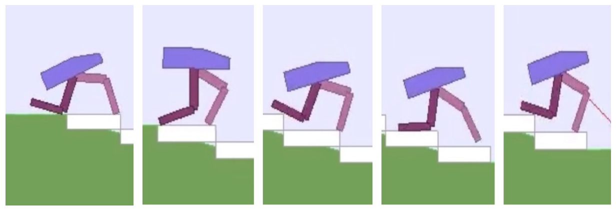

Figure 11: Pass stair successful

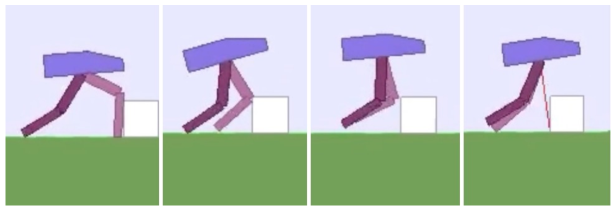

Figure 12: Pass block failed

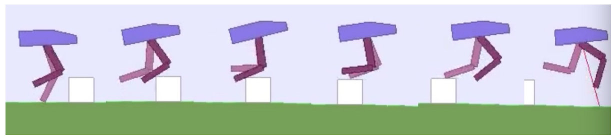

Figure 13: Pass big block

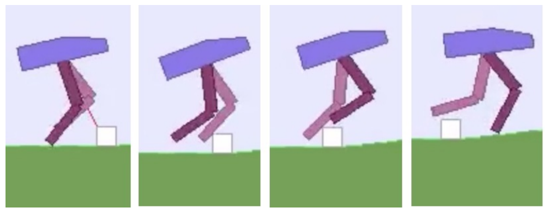

Figure 14: Pass small block

In Figure 13and Figure 14, these figures show the successful pattern which we crop from another person's 80 hours' training result. The "walker" in this figure performance as a jumper,which is natural and stable. Although we have add a short-term memory to help "walker" to combined all the previous observations stored in the queue and reconstruct the whole picture, it need a large quantity of time to train it. And in our result, it has a good trend to pass the blocks, maybe more training time could make a better performance in future.

6 CONCLUSION

In this project, we have presented a deep reinforcement learning algorithm A3C and evolution strategy, and based on those algorithm we add LSTM to actor-critic experts to model for helping "walker" to see whole picture and improve performance. The architecture generates the control policies which can work for complicated terrain adaptive locomotion. With the result in Figure 10, after giving the model a "think" net like actor-critic, the "walker" could have the correct way to exploration. However, for evolution strategy,it does not give a clear direction for the model to explore "new" gesture to pass different terrain. Also, without LSTM in model, evolution strategy may be limited into local minimum easier. That's why it's gesture is a slight bit backwards(only focus on passing and do not focus on better balance position).

However, there is also some problems in A3C. Since it's actor part and critic part are connected, in some cases, the network will go into a endless loop. Fortunately, there is a updated algorithm called Deep Deterministic Policy Gradient, which could solve the endless loop problem.

In our future work, we will reduce deviation from proven good policies and trust region policy optimization based on Deep Deterministic Policy Gradient algorithm. Most importantly, we will try a longer training time in our new model.

We believe that RL method with deep neural network will have a good performance in physics-based character control and be applied wildly in the future.

ACKNOWLEDGMENTS

The authors would like to thank their professor Demetri Terzopoulos and TA for their valuable instructions and helpful mentoring.

REFERENCES

[1] Nikolaus Hansen and Andreas Ostermeier. 1996. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*. IEEE, 312–317.

[2] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. 2017. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286* (2017).

[3] Yuxi Li. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).

[4] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *CoRR* abs/1602.01783 (2016). arXiv:1602.01783 http://arxiv.org/abs/1602.01783

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[6] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2015. Dynamic Terrain Traversal Skills Using Reinforcement Learning. *ACM Trans. Graph.* 34, 4, Article 80 (July 2015), 11 pages. https://doi.org/10.1145/2766910

[7] Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. 2017. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Trans. Graph.* 36, 4, Article 41 (July 2017), 13 pages. https://doi.org/10.1145/3072959.3073602

[8] PVoodoo. 2017. Deep Reinforcement Learning for Trading, NOPE! (2017). https://2.bp.blogspot.com/-bZERYUNyjao/Wa98yt7GjhI/AAAAAAAACt8/SYQjUNrbe1YDtKTMKR6LPt68C0pPqkoowCLcBGAs/s1600/DRL.JPG [Online; accessed March 26, 2018].

[9] Doo Re Song, Chuanyu Yang, Christopher McGreavy, and Zhibin Li. 2017. Recurrent Network-based Deterministic Policy Gradient for Solving Bipedal Walking Challenge on Rugged Terrains. *arXiv preprint arXiv:1710.02896* (2017).

[10] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[11] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. 2014. Natural Evolution Strategies. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 949–980. http://dl.acm.org/citation.cfm?id=2627435.2638566