Learning To Run

Leonardo Arcari
Politecnico di Milano
leonardo.arcari@mail.polimi.it

Emiliano Gagliardi
Politecnico di Milano
emiliano.gagliardi@mail.polimi.it

Emanuele Ghelfi
Politecnico di Milano
emanuele.ghelfi@mail.polimi.it

Abstract

In this paper we present our approach for the NIPS 2017 "Learning To Run" challenge. The goal of the challenge is to develop a controller able to run in a complex environment, by training a model with Deep Reinforcement Learning methods. We follow the approach of the team Reason8 (3rd place). We begin from the algorithm that performed better on the task, DDPG. We implement and benchmark several improvements over vanilla DDPG, including parallel sampling, parameter noise, layer normalization and domain specific changes. We were able to reproduce results of the Reason8 team, obtaining a model able to run for more than 30m.

1. Introduction

Running is a very complex task. Understanding the dynamics of the human body while running is important for studying rehabilitation and the development of sophisticated prosthesis. From a control point of view, it involves the actuation and coordination of a large set of muscles and high dimensional perception.
Reinforcement Learning (RL) [1] has been successfully applied to various kinds of control problem and the introduction of deep neural networks augments the application scenario.
The NIPS 2017 Learning To Run [2] competition is focused on developing a controller to enable a physiologically-based human model to navigate a complex obstacle course as quickly as possible. We are provided with a human musculoskeletal model and a physics-based simulation environment where it is possible to synthesize physically and physiologically accurate motion. The score is based on the distance the agent travels in a given amount of time. The scope of our project is to replicate the results of Reason8 team [3] that placed 3rd in the competition.

Following their approach, we addressed the problem using DDPG algorithm. We started from the OpenAI baselines [4] implementation and extended it with:

- Parallel sampling to speed up learning and evaluation processes.
- Changes to improve exploration capabilities.
- Domain-specific improvements to the training step.

Moreover we compared the performance of our algorithm with state spaces of different dimensions.
This paper is structured as following: in section 2 we review the basics of Reinforcement Learning, in section 3 we provide a brief overview of the RL methods which DDPG, the method of our choice, builds upon, in section 4 we illustrate our approach to the problem and the task-specific improvements we introduced over the literature. In section 5 we describe the environment and our modifications. Finally in section 6 we present our experimental results.

2. Background

We consider a standard reinforcement learning setup consisting of an agent interacting with an environment in discrete steps. At each step t the agent receives some representation of the environment's *state* s_t, takes an action a_t and receives a scalar reward r_t. We model the problem as a *Markov decision process* (MDP) [5] which comprises: a *state spaces* $\mathcal{S}$, an *action space* $\mathcal{A}$, an *initial state distribution* with density $\mu(s)$, a stationary *transition dynamics distribution* with conditional density $p(s_{t+1}|s_t, a_t)$ satisfying Markov property, and a *reward function* $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. A *policy* is used to select actions in the MDP, representing the agent behavior. Formally it is denoted by $\pi_\theta : \mathcal{S} \to \Delta(\mathcal{A})$ where $\Delta(\mathcal{A})$ is a set of probability measures on $\mathcal{A}$ and $\theta \in \mathbb{R}^n$ is a vector of n parameters. The return r_t^γ is the total discounted reward from time step t onwards, $r_t^\gamma =$

$\sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ where $0 < \gamma < 1$. The agent's goal is to maximize the *cumulative discounted reward* denoted by $J(\pi) = \mathbb{E}_{s_1 \sim \mu}[r_1^{\gamma} | \pi]$.

3. Related work

Policy gradient methods rely upon optimizing a parameterized policy in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$. A straightforward approach to accomplish this is presented in [6] with REINFORCE algorithm, that uses Monte Carlo sampling to estimate the performance gradient considering a stochastic policy. DPG [7] expands on this by considering *deterministic policies* only, for continuous action spaces. To ensure adequate exploration, an off-policy actor-critic algorithm is introduced to learn a deterministic target policy from an exploratory behavior policy. However, directly using neural networks as function approximators leads to unsatisfactory results due to two problems: a) most optimization algorithms for neural networks assume that samples are independently and identically distributed, which is not true when samples are generated from exploring sequentially in an environment b) since the network $Q(s, a|\theta^Q)$, part of Q-learning algorithms, being updated is also used in calculating the target value, the Q update is prone to divergence [8]. DQN [9] implements Q-learning using a deep neural network to approximate the Q function. To address the problem (a) DQN introduces a *replay buffer* which stores transitions generated by the environment. During training, a batch of transitions is drawn from the buffer to restore the *i.i.d.* property. Although, since a maximization over the action space is required, DQN does not scale with high-dimensional and continuous action spaces. DDPG [8] solves all the these problems by using 1) a deterministic actor $\pi(s) = \arg\max_a Q(s, a)$ to scale with high-dimensional continuous action spaces 2) a *replay buffer* 3) separated target networks with *soft-updates* to improve convergence stability.

4. Proposed approach

In this section we describe our solution to the task of the "Learning to run" challenge. The purpose of our project was to replicate the results obtained by the Reason8 team. According to [3] the method that best performed on the challenge task was DDPG, therefore that was our choice. There are several reasons why DDPG is a good choice for this problem:

- The environment simulator, OpenSim [10], is very slow and generating samples can take seconds. Therefore having a sampling-efficient method is a must. Being an off-policy method, DDPG allows re-usage of all the samples, whereas on-policy approaches would require to throw them away at each update.
- The algorithm from the literature is given in a sequential formulation, but the samples generation part is easily parallelizable.

4.1. DDPG

DDPG [8] is an off-policy actor critic method. The actor network implements the agent's policy, as a deterministic function $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The critic network approximates $Q(s, a)$. Critic is trained using off-policy data coming from the replay buffer, that is a FIFO queue containing tuples (s_t, a_t, r_t, s_{t+1}). Critic's task is to minimize the Bellman error (notice that the policy is deterministic, so we can avoid the expectation over actions):

$$Q(s_t, a_t \mid \theta^Q) = r(s_t, a_t) + \gamma Q(s_{t+1}, \pi(s_{t+1} \mid \theta^\pi) \mid \theta^Q) \tag{1}$$

where θ^Q are parameters of the critic network and θ^π are actor's parameters. It is evident in 1 that the update step of the weights θ^Q comprises $Q(s_t, a_t \mid \theta^Q)$ in the target, resulting in an iterative update prone to divergence. DDPG solves this problem using target networks. Target networks are copies of the actor and critic networks that are only used for computing target values, and softly updated for improving stability:

$$\theta' = \tau\theta + (1 - \tau)\theta' \qquad \tau \ll 1 \tag{2}$$

where θ are the weights of the original network and θ' the weights of the target networks.
The resulting error for the critic network is:

$$L = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(s_i, a_i \mid \theta^Q))^2 \tag{3}$$

with target

$$y_i = r_i + \gamma Q(s_i, \pi(s_i \mid \theta'^\pi) \mid \theta'^Q) \tag{4}$$

The actor is updated using the estimated deterministic policy gradient, here reported for the sake of completeness:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi} \pi(s \mid \theta^\pi) \mid_{s_i} \tag{5}$$

4.2. DDPG improvements

In this section we describe the techniques we used to increase the vanilla DDPG performance.

Parallel sampling A key step in any reinforcement learning algorithm is the generation of (s_t, a_t, r_t, s_{t+1}) transitions to gather information from the environment. Since our simulator is very slow, making this step as fast as possible is

desirable. Parallelization in our algorithm is implemented through three types of threads: sampling threads, training thread and evaluation threads. Each sampling thread is tasked with collecting trajectories using the provided policy, pushing them in a common queue and waiting for a new policy. The training thread waits samples from m sampling threads, stores them in the replay buffer and trains the actor and critic networks for a fixed number of training steps. The new actor network is then sent to the waiting sampling threads that can now restart the sampling process. Evaluation threads are spawned every fixed number of training steps.
Having multiple sampling threads running policies with different weights improves exploration and results in a substantial performance improvement. We used 20 sampling threads, 1 training thread and 5 evaluation threads in our implementation. We found out in our experiments that $m = 1$ is a good trade-off between sampling and training.

Exploration To explore we used action noise and parameter noise [11] in an alternated way. At the beginning of an episode we selected between action noise and parameter noise with 0.7 and 0.3 probability respectively.
Action noise is directly applied to the action selected by the actor network. We used an Ornstein-Uhlenbeck (OU) [12] process to generate correlated noise for efficient exploration in physics based environments.
Parameter noise perturbs actor network weights to obtain a state dependent exploration, thus more coherent with respect to action noise. The noise used in parameter noise is sampled at the beginning of an episode and it's kept fixed for all the episode. Parameter noise works well with layer normalization [13]. Layer normalization, as the name says, normalizes the output of a selected layer. This technique, besides stabilizing training, makes possible to use the same perturbation scale across all network layers. We used layer normalization both for actor and critic networks applying it to all layer except the last one before the non linearity.

States and actions flip The model has a symmetric body, thus it's easy to increase the sample size by flipping states and actions. Flipping a state means to swap left and right parts of the body, flipping an action means to swap actuations of left and right legs. States and actions flip is implemented in this way: we sample transitions (s_t, a_t, r_t, s_{t+1}) from the replay buffer, flip state components of s_t and s_{t+1}, flip the action a_t and add to the batch original as well as flipped transition. This has an easy interpretation: we know that if action a_t in state s_t , results in state s_{t+1} and in a reward signal r_t, doing the symmetric action with respect to a_t in the symmetric state with respect to s_t results in the symmetric state with respect to s_{t+1} and in the same reward signal r_t.
Flipping transitions helps in obtaining symmetric policies, that is desirable since running is a symmetric task.

5. Environment

The agent is a musculoskeletal model including information about muscles, joints and links moving in a 2D environment (no movement is possible along Z axis). Kinematic quantities are provided for body parts and joints, while activation, fiber length and fiber velocity are provided for muscles.
The total amount of variables describing the state of the agent is 146. The agent can actuate 9 muscles for each leg, thus $a \in [0, 1]^{18}$. In our version of the environment we removed obstacles, obtaining a slightly easier version of the task. Reward is defined as the change in the x coordinate of the pelvis for each step plus a small penalty for using ligament forces. An episode terminates when the agent falls (pelvis $y < 0.65$) or after 1000 steps. Simulation is based on the OpenSim library that relies on the SimBody physics engine. Due to a complex physics engine the environment is quite slow compared to standard RL environments (Roboschool, Mojoco, etc.) and a step can take seconds, thus it is crucial to use the most sample-efficient method.

5.1. Environment modifications

We used several modifications of the environment during training to improve efficiency and to help the learning algorithm.

Reward We added a small bonus to the reward for each time-step survived. We did not study the contribution of this change thoroughly, but we expect it to add some greediness to the initial training steps to favor policies that keeps the model standing.

Environment accuracy We used a lower integrator accuracy with respect to the standard one of the simulator obtaining 5x speedup. After some episodes the environment becomes slower with respect to initial episodes, probably for a memory leak. We solved this problem doing a reset of the environment after 100 episodes.

Relative positions As mentioned above, position vectors of the model body parts are exposed by the simulator as absolute. For the purpose of learning, keeping an absolute reference frame is undesirable. In fact, being running an approximately periodic task, having the skeleton in some position at a given distance from the origin, should make no difference from having it in the same position at another distance. Therefore, we modified the observation vector by centering the x coordinates of the body parts around the *pelvis* x. Exploiting such symmetry of the model enables

shorter learning time and, most importantly, higher generalization.

State variables OpenSim exposes a number of variables for a musculoskeletal model. Even though to preserve the Markovian property we should consider them all, many of them are in practice useless for the task to learn. In training our model, two subsets of them were considered and we refer to them as *full-state* and *reduced-state*.
Reduced-state comprises the x, y position of body parts, the rotation and rotational speed of joints, the speed and position of the center of mass, resulting in $s \in \mathbb{R}^{34}$. Namely body parts are *head, torso, right and left toes and talus* and joints are *ground pelvis and left and right ankles, hips and knees*.
Full-state takes into account all the variables from *reduced-state*, together with the speed and acceleration of body parts and acceleration of joints, resulting in $s \in \mathbb{R}^{67}$.

6. Experiments

In this section we describe our experimental results. For all the experiments we ran DDPG algorithm with the modifications we describe in sections 4 and 5. We performed an *ablation study* to test the relevance of our main changes to vanilla implementation. We compared the performance of a model trained on the *reduced-state* space with respect to the *full-state* space. Moreover we compared the quality of the models learned with or without state-action-flip and parameter noise, in terms of performance and required training steps. All the models running on the *reduced-state* configuration share the same architecture for actor and critic networks, with Xavier initialization [14] for the neurons. Best performing hyper-parameters are listed in table 1.

6.1. State-action flip and parameter noise

In this experiment we investigated on the importance of *state-action flip* and *parameter noise* (PN) for the learning process. We trained four models for approximately 10^6 training steps with all the combinations of the two improvements, i.e. with and without state-action flip and parameter noise. Performance statistics are reported in figure 1a. From our experimental results, introducing both modifications leads to both better performance, in terms of longer run distance, and a significant speed-up in terms of training steps to reach same distance. It is also worth highlighting that the learned model with state-action flip achieved higher performance than the one with PN only. This possibly remarks the importance of domain-specific additions in the context of RL which outperformed an uninformed exploration.

Parameters	Value
Actor network architecture	[64, 64], *elu* activation
Critic network architecture	[64, 32], *tanh* activation
Actor learning rate	Linear decay from $1e{-3}$ to $5e{-3}$ in 10^6 steps with Adam optimizer
Critic learning rate	Linear decay from $2e{-3}$ to $5e{-5}$ in 10^6 steps with Adam optimizer
Training steps	300
Batch size	400
Sampling processes	20
γ	0.9
τ	0.0001
Replay buffer size	$5 \cdot 10^6$
Reward scaling	10
Action repeat	5
Parameter noise probability	$\sigma = 0.2$
OU exploration parameters	$\theta = 0.1$, $\mu = 0$, $\sigma = 0.2$

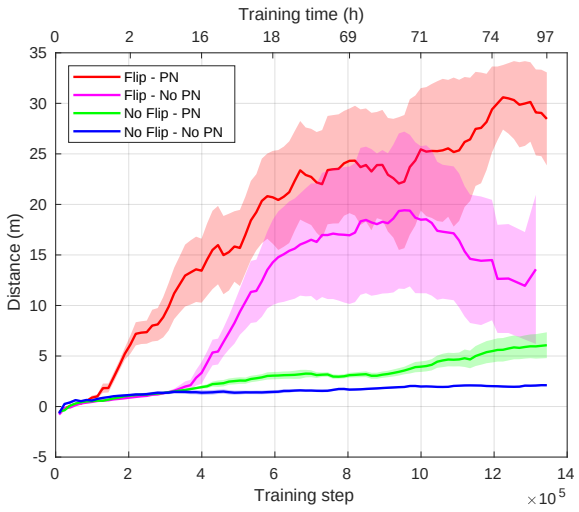
Table 1: Hyper-parameters used in our best experiment with *reduced-state*

6.2. Sampling threads

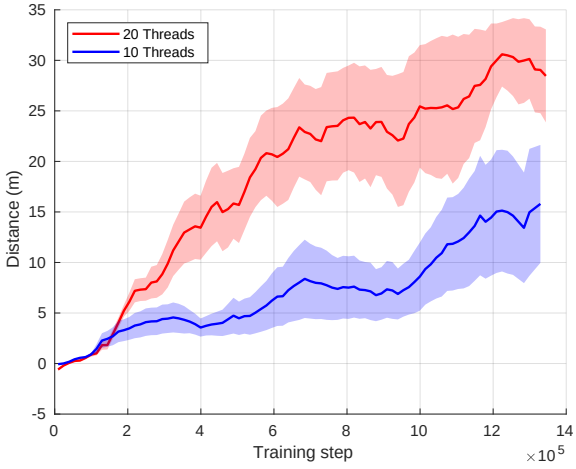
In this experiment we analyzed the impact of the number of sampling threads. We trained two models with 10 and 20 sampling threads respectively. We used the same sampling-training strategy: wait for samples from 1 thread, check the state of the other threads (collecting samples if available), train for 300 steps, send the updated policies to waiting threads that restart the sampling process. Figure 1b shows that, as expected, the experiment with 20 sampling threads outperformed the experiment with 10 sampling threads. This shows the importance of exploration in this task, as well as the importance of parallelization.

6.3. Full-state vs. reduced-state spaces

In this experiment we compared the performance of two learned models, the first trained over *full-state* space and the second over *reduced-state* space. The former was trained using a $[150, 64]$ *elu* actor network and a $[150, 50]$ *tanh* critic network. The latter was trained with a $[64, 64]$ *elu* actor network and a $[64, 32]$ *tanh* critic network. Performance statistics are reported in figure 2. From our experiments, models trained with a *reduce-state* space outperformed those trained with the bigger state space. *Full-state* space introduces several variables that could help in learning a controller for our task, but they also increase the complexity of the model. We did not test thoroughly the networks architecture for the *full-state* space and incrementing the number of neurons might lead to better performance.



(a) Performance of models trained over reduced-state space. Training time is only relative to the Flip - PN experiment.



(b) Performance of models trained using 20 vs 10 sampling threads on reduced-state space

6.4. Comparison with Reason8 results

The ultimate goal of our project was to replicate the results obtained in the NIPS 2017 competition by one of the best teams, Reason8. In figure 3b [3] shows the maximal rewards achieved in the given time, with a peak at 38.46 m for their best model. Our best model, with the same DDPG improvements, ran 32.97 m, as reported in table 2. The performance we achieved are comparable to those of Reason8 team and, from our perspective, satisfactory.

6.5. Equipment

Experimental results reported in sections above are obtained by running our implementation on the following machines:

- Intel Xeon (32 cores, 64 threads), RAM 256 GB, Ubuntu 16.04 - owned
- Intel Xeon (16 cores, 32 threads), RAM 128 GB, Ubuntu 16.04 - owned
- Intel Xeon (16 cores, 32 threads), RAM 64 GB, Ubuntu 16.04 - Microsoft Azure

Method	State space	Distance (m)
Flip - PN	Reduced	**33.4074**
Flip - No PN	Reduced	25.8489
Flip - PN	Full	15.4193
No Flip - PN	Reduced	8.6108
Flip - No PN	Reduced	2.535

Table 2: Run distances by different configurations of the DDPG algorithm in our experiments

6.6. Implementation

We made our code available in our github repository: https://github.com/MultiBeerBandits/learning-to-run.

7. Conclusions

In our project we replicated the results obtained by one of the best teams in the NIPS 2017 - Learning to Run competition. We implemented several improvements on top of vanilla DDPG algorithm, either general or domain-specific.

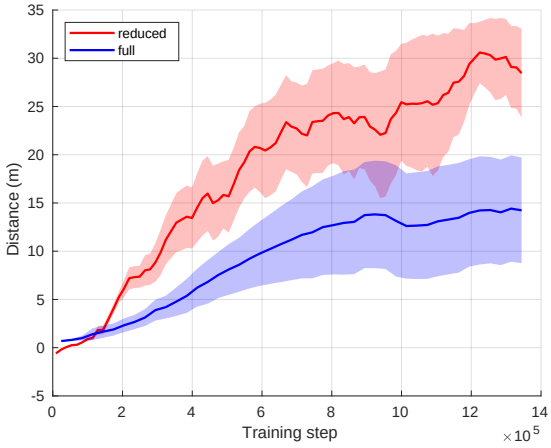


Figure 2: Performance of models trained over full vs reduced state space

We compared the performance of the algorithm over two state spaces with significantly different dimensionality and investigated the impact of our changes on the quality of the learned models. Future work might expand from this by researching more on the architecture of actor and critic networks for large state spaces like *full-state*. It would also be interesting to spend more time in tuning other RL algorithms that performed worse in their baseline implementation and compare them with our achieved results. Moreover the algorithm could be extended to compete in the NIPS 2018 - AI for Prosthetics challenge [15].

8. Acknowledgments

We would like to thank our friend Edoardo Varani for the performance analysis of our machines and our algorithms. We also thank professor Marcello Restelli and his PhD students Alberto Maria Metelli, Andrea Tirinzoni and Matteo Papini for having proposed this work and for the support they provided to us.
A special mention is for the team Reason8 that made our work possible.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. 1

[2] crowdAI, "NIPS 2017: Learning to Run," 2017. https://www.crowdai.org/challenges/nips-2017-learning-to-run. 1

[3] M. Pavlov, S. Kolesnikov, and S. M. Plis, "Run, skeleton, run: skeletal model in a physics-based simulation," *arXiv:1711.06922 [cs, stat]*, Nov. 2017. arXiv: 1711.06922. 1, 2, 5

[4] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines." https://github.com/openai/baselines, 2017. 1

[5] R. Bellman, "A markovian decision process," vol. 6, p. 15, 04 1957. 1

[6] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992. 2

[7] D. Silver et al., "Deterministic Policy Gradient Algorithms," *ICML*, 2014. 2

[8] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2016. 2

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs]*, Dec. 2013. arXiv: 1312.5602. 2

[10] Standford University, "OpenSim Simulation Toolkit," 2007. http://opensim.stanford.edu/. 2

[11] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, "Parameter space noise for exploration," *CoRR*, vol. abs/1706.01905, 2017. 3

[12] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930. 3

[13] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv:1607.06450 [cs, stat]*, July 2016. arXiv: 1607.06450. 3

[14] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010. 4

[15] crowdAI, "NIPS 2018: AI for Prosthetics," 2018. https://www.crowdai.org/challenges/nips-2018-ai-for-prosthetics-challenge. 6