Learning to walk with multi-agent systems

Nikita Burov
nikita.burov@studio.unibo.it

September 2022

The project aims to create a multi-agent system to control the musculoskeletal model of the human body in the way it will be able to walk. Agents can control the human body via muscles attached to a skeleton in a biomedical simulation. Training of agents is done with reinforcement learning.

1 Introduction

Control tasks are quite common in a real life. Successful solutions to these tasks might find their applications in robotics, intelligent systems, autonomous driving, etc. The tasks of developing controllers for 3D human models are also control tasks. Even though the problem statements of such tasks are easy to understand, solving them might require knowledge from neuroscience, biomechanics, robotics, and machine learning. One of these tasks - Learning to Walk around from NIPS 2019 [6]. Machine learning engineers here had to develop a controller for the plausible 3D human model, so it would be able to walk around.

1.1 Goal/Requirements

The goal of the project is to obtain a multi-agent system that will be able to control a model of the human skeleton so it will be able to walk.

One of the natural ways to solve such type of challenges is reinforcement learning. So, in this project, the multi-agent system is supposed to be obtained as a result of reinforcement learning algorithms.

To achieve the goal it is necessary to:

- Design multi-agent system
- Implement reinforcement learning approach. DreamerV1 [5] was chosen for a baseline.
- Implement training pipeline, including supplementary modules, such as replay buffers, multi-processing utilities, and environment wrappers.

- Launch training experiments with different hyperparameters to obtain the best models.

As a result of training experiments, it is expected to obtain trained weights for neural networks according to the Dreamer approach, training metrics, and losses charts, and replay buffer with training experience.

1.2 Environment description

The environment for the task is designed in a classical reinforcement learning way. The controller takes a step in the environment and the environment returns a new observation and immediate reward for the step. These actions are repeated while the terminate state is not reached.

Each observation in the environment for the challenge consists of two parts - the local velocity map represented as a vector field and the vector of the body state. A local velocity map contains information about the direction to the point to which the skeleton should walk. The body state vector contains information about the velocity, angles, and other parameters of the skeleton model.

Detailed observation description is given in the repository of the challenge [6].

Multi-agent system has access to 22 muscles (11 per leg) attached to a simplified human skeleton model. The system has to control these muscles to maximize the reward i. e. learn how to walk around. Actions in the environment are continuous in [0..1]

The controller is rewarded for moving in the right direction, keeping the skeleton figure standing, and maintaining the right speed in accordance with a target velocity map.

1.3 Self-assessment policy

Self-assessment policy contains two groups of criteria - necessity and sufficiency criteria. Sufficiency criteria could not be met without fulfilling necessity ones, but fulfilling necessity criteria will not guarantee that sufficiency criteria are going to be met.

The necessary criteria for success are a correctly working RL approach and an implemented multi-agent system. Validation of implementation of the approach might be done in several ways. One of them - is to solve some simple RL challenge (inverted pendulum problem from gym environment [1]). The other is to monitor training losses and compare agents' current performance with what is seen on the charts.

There are two sufficiency criteria of success of the given work. The first one is informal - during rendering the simulation we can evaluate whether agents control the skeleton in the right way (so that it is walking). The second one (formal) is to rely on the environment's reward function, which agents should learn to maximize. Actually, maximizing rewards should imply that agents learned how to walk.

2 Design

This section describes the design of the multi-agent system and how the training pipeline was implemented.

2.1 Design of the multi-agent system

As the baseline for the multi-agent system DreamerV1 approach was chosen. The architecture of the system consists of three main parts: environment encoder, recurrent state-space model (RSSM) [2] and actions decoders from the latent space Z. The illustration is given in the figure **??**. Each of the parts is, in fact, a separate neural network. These networks are trained jointly according to [5].

The encoder receives two-parted observations, performed action, and outputs latent representation for the RSSM model. RSSM model gets encoded embedding and combines it with information from the previous steps for the following action decoding.

Action decoders might be considered as agents which are represented by the neural networks. Each agent predicts actions for its own leg, and of course, they have to do it cooperatively. The interaction between agents is implicit and done via a recurrent state-space model and environment encoder.

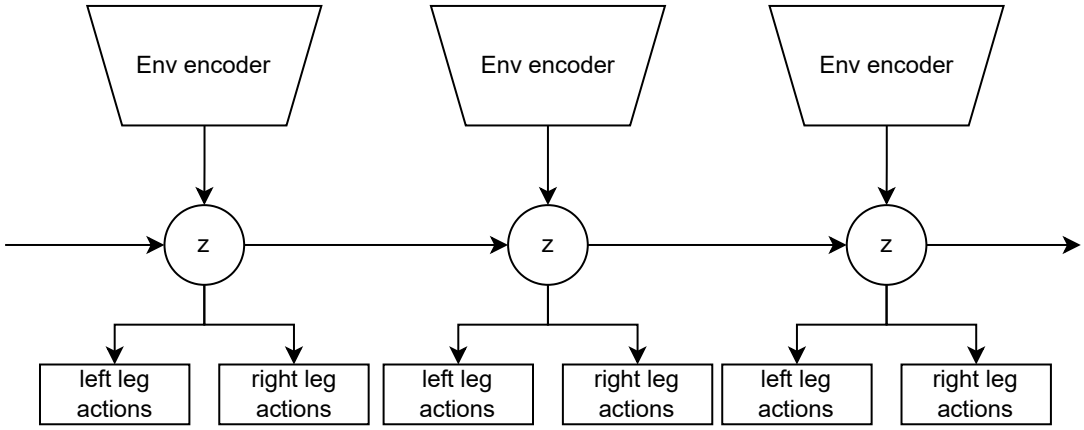


Figure 1: Multi-agent system design

2.2 Design of the collecting experience pipeline and training procedure

It is crucial for reinforcement learning to perform as many steps in the environment as it is possible. To speed up the collecting experience process multiprocessing with MPI was used. Training scripts execute in several processes. One of them is the main process, which manages training procedures and collecting experience. Child processes have running environments. They receive actions for execution from the main process, execute them in their environments and send back observations and rewards. Pseudocode of train pipeline is given in 1

Training of models based on collected experience is done according to [5].

3 Implementation Details and instructions

The code for the implementation is represented in the gitlab repository [3].

Algorithm 1 Training algorithm

```
Initialize environments in child processes
for i ← 1 to number of epochs do
    for j ← 1 to number of episodes do
        Send reset environments signal to child processes
        Send initial observations to the main process
        while some environment is alive do
            Predict actions in the main process
            Send actions to the environments in child processes' environments
            Execute actions in child processes in parallel
            Send new observations and rewards back to the main process
            Append observations, actions, and rewards to the replay buffer
        end while
    end for
    Perform training and update models
end for
```

It was interesting to measure the performance benchmarks of the resulting code. Noted details:

- Performance of the replay buffer strangely does not depend on whether to store observations in the numpy arrays or in python lists.
- With MPI it is crucial to specify datatypes with send and receive operations. Python with dynamic typing cannot prevent sending of a float number with a single precision instead of a float64.
- Performance of blocking operations of send and receive with provided training algorithm is almost equal to performance with non-blocking communication. But Sendrecv operations slow down the algorithm significantly.

  Deployment and usage instructions are also given in the repository [3].

4 Self-assessment / Validation

The necessity criteria of work were met. Working RL approach was implemented and validated on an easy challenge from gym environment [1]. Multi-agent system was designed for the task and implemented in python.

The sufficiency criteria of the work were not met. The reward achieved by models is still quite poor and the model of the human body can not walk.

On the figure, 2 presented loss and rewards plots collected during the last training experiment. Training run started with pre-trained models, so initial loss values for freshly initialized models are different. It might be seen that losses are decreasing and everything seems plausible, but values for rewards obtained during training do not grow.

Neural networks might be considered general-purpose function approximators. Noisy plots with a big number of outliers might be an indicator of a very sharp nature of solutions.

5 Encountered issues

Multi-processing and Reinforcement learning code is very difficult to debug. Usually, it is very difficult to understand whether the approach itself doesn't work, hyperparameters are wrong or there is an error in used libraries and frameworks. Moreover, reinforcement learning requires a lot of computational resources, which is why some issues occurred during the deployment stage of the training environment to the remote HPC server.

5.1 Deployment issues

The main issue here was combining all python packages into a hardware-independent wrapper. Local testing and development were done on the Windows machine, while training was performed on the HPC Linux server. That's why some compatibility issues occurred. A workaround for this problem was using of conda virtual environment.

The other problem was OpenMPI, which is hardware dependent. To solve the problem building pipeline for OpenMPI was implemented in a Dockerfile.

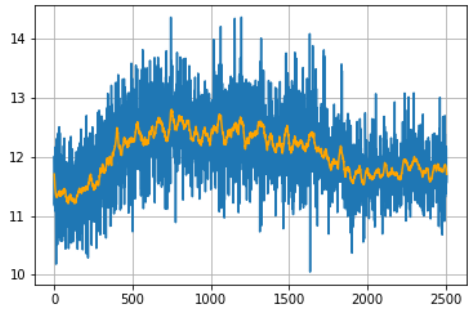
5.2 Hyper parameters fine-tuning

DreamerV1 approach uses probabilistic losses to train neural networks. The main difference of the current project and the environment used in the original paper [5] is that observations from biomedical simulation come from an unclear range of values. That caused problems, with losses convergence and imbalance. For example, the loss component for the reward model from [5] with initial parameters differs only after the fifth floating point character (10^{-5}), while reconstruction losses were approximately 10^6. The workaround for this problem was fine-tuning loss parameters and adding weights to loss components.

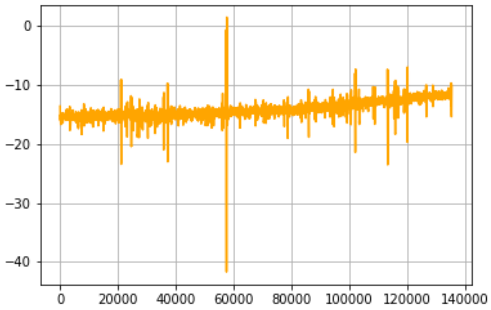
5.3 Environment drawbacks

The environment for simulation is quite complex and slow. After passing some threshold with MPI execution processes (¿20) program falls into intermittent hangs for up to 40 minutes. Even though to speed up the environment accuracy of the built-in integrator was decreased the problem with these hangs remains unsolved.

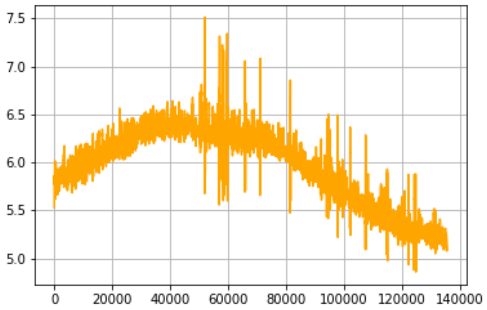
The other problem lies in the reward function. It is not very accurate and is not discriminative enough for efficient training. During training experiments, it was revealed that if agents decide to do nothing (zero action vector) then the reward will be $13 - 15$ reward points, while in early stages with random actions the model is able to achieve $8 - 10$ reward points.



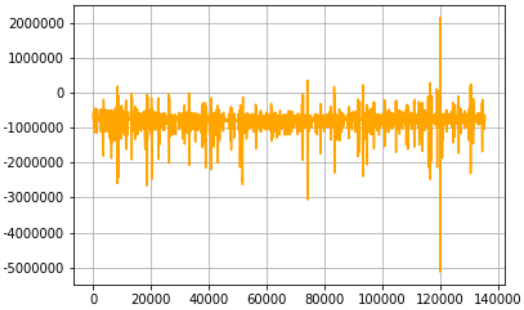
(a) Mean rewards plot



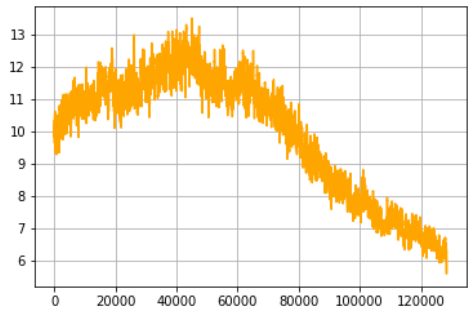
(b) Reward loss plot



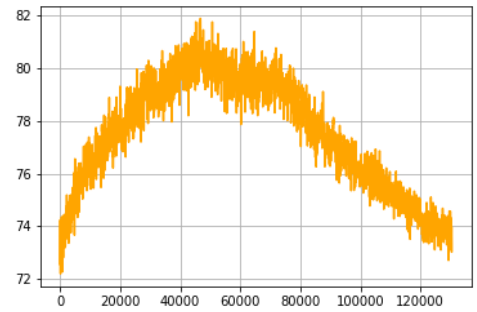
(c) Divergence loss plot



(d) Reconstruction loss plot



(e) Value loss plot



(f) Actor loss plot

Figure 2: Plots

5.4 Algorithm bootstapping

Authors of the challenge provide links to the recorded simulation of real human gaits in OpenSim [4] environment. It might've been high-quality data to fill the replay buffer, but musculoskeletal models used in the challenge and for gait recordings are different. Solvers for muscle activation cannot find a solution. As muscle activations are in fact outputs of the multi-agent system data from recordings without them cannot be used.

6 Conclusions

To summarise, the multi-process MPI-based pipeline for reinforcement learning with DreamerV1 approach for the chosen task was implemented. The reinforcement learning approach was modified for a multi-agent system setup. To make the process of deployment easier pre-built docker image was created.

Other useful results of the project are:

- Knowledge of general reinforcement learning was significantly improved.
- DreamerV1 approach was studied extensively.
- Skills of working with docker containers and MPI were acquired.

The system was not trained to finally walk around and this is the main thing to do. Also, there is an iterative improvement of the used reinforcement learning approach which also might be implemented.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Lars Buesing, Theophane Weber, Sébastien Racanière, S. M. Ali Eslami, Danilo Jimenez Rezende, David P. Reichert, Fabio Viola, Frederic Besse, Karol Gregor, Demis Hassabis, and Daan Wierstra. Learning and querying fast generative models for reinforcement learning. *CoRR*, abs/1802.03006, 2018.

[3] Nikita Burov. Learning to walk around repository, 2022.

[4] Scott Delp, Jennifer Hicks, Ayman Habib, Thomas Uchida, Ajay Seth, and Carmichael Ong. Simtk: Opensim: Project home, 2022.

[5] Danijar Hafner, Timothy P. Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *CoRR*, abs/1912.01603, 2019.

[6] Łukasz Kidziński Seungmoon Song. Reinforcement learning with musculoskeletal models, 2020.