Human Motion Simulation Using Reinforcement Learning

Auteur : Adriaens, Jérôme
Promoteur(s) : Sacré, Pierre; Bruls, Olivier
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil électricien, à finalité spécialisée en "signal processing and intelligent robotics"
Année académique : 2022-2023
URI/URL : http://hdl.handle.net/2268.2/17681

University Of Liège
Faculty of Applied Sciences

Academic year 2022-2023

Human Motion Simulation Using Reinforcement Learning

Author
Jérôme Adriaens

Supervisors
Prof. Dr.-Ing. Olivier Brüls
Prof. Dr.-Ing. Pierre Sacré

Jury
Prof. Dr.-Ing. Guillaume Drion
Prof. Dr.-Ing Cédric Schwartz

Master's thesis completed in order to obtain the degree of Master of Science in **Electrical** Engineering by Jérôme Adriaens

Professional focus: Signal Processing and Intelligent Robotics

Abstract

The simulation of realistic human motion is a critical aspect in several fields. Ranging from character animations in video games to medical research, human motion simulation is involved in a lot of domains. In fact, replicating physiologically plausible human motion is essential for creating realistic human motion. However, due to the complexity of modeling a physiologically accurate model, being able to simulate a realistic motion is very challenging. A common approach to tackle this kind of problem includes reinforcement learning. Since reinforcement learning is very popular nowadays and showed to be quite successful on a bunch of tasks, this is the approach chosen for this work. In particular, this thesis aims at controlling a physiologically plausible model in order to make it move forward. This work is segmented into 3 parts. First, the key concepts for this work are exposed in order to make the following as clear as possible. Then, the main components of the reinforcement learning framework are chosen through a comparative analysis of several sets of components. This concerns the algorithm, the neural network architectures along with other training methods. Lastly, a controller is to be trained to make a human model moves forward. The behavior of the human model is then analyzed to assess its gait. This work brings insights into various elements that are important when using reinforcement learning to train an agent to move forward. In particular, it provides a detailed method for training an agent as well as a description of the main components. In addition, this controller succeeds to make a musculo-skeletal model walk forward.

Acknowledgements

This master's thesis was completed with the priceless support of many people, all of whom I would like to express my sincere gratitude.

Firstly, I would like to thank both my supervisors, Oliver Brüls and Pierre Sacré for their support and advice throughout the entire duration of the thesis. In particular, they really helped me organize the thesis' workload and define the methodology without discouraging me at any time. For all of this, I am particularly grateful.

Then, during this thesis, some computations were performed thanks to the NIC5 cluster of Liège University which computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region. Therefore, I would like to thank the NIC5 administrator: David Colignon whose help allowed me to use the cluster for this thesis.

Finally, I would like to thank warmly my family and friends who supported me and helped me during the entire thesis.

Contents

Introduction

Problem Statement

Simulating physiologically plausible human motions is an essential aspect of creating realistic and immersive environments. It involves modeling and replicating the movements of humans in a way that closely mimics their natural patterns and behaviors as well as the underlying mechanisms. By doing so, virtual environments can be designed to be more reliable, which is particularly important in fields linked with biomechanical simulations or virtual reality.

In particular, a wide range of applications involves human interactions and so requires high degrees of reliability from those virtual environments. In order to create plausible human motions, a lot of physics constraints must be respected while considering a muscle-actuated body. Indeed, the dynamics of a human body have constraints and limitations enforced by the human body, such as the range of motion in a joint or the torque developed by a joint. These parameters are intrinsically linked with muscle characteristics and dynamics among other aspects of biomechanics.

It is then important to be able to reproduce plausible human movements from a biomechanical perspective to develop many applications requiring human interactions in simulated environments. For example, in the case of autonomous driving, it is clear that human-car interactions are essential concerns. In such cases, real-life testing can be laborious because it may rise security hazards if humans are involved, and using dummies might not reflect real cases. Using simulated environments is a workaround. However, to be as close as possible to real conditions for the human part, the motion control and simulation should be as plausible as possible. Another example relates to the field of virtual reality gaming and entertainment. Realistic human motion can help create a more immersive and engaging experience for users. Simulating the movements of a character in a game using realistic human motion, might enhance the sense of presence that players feel in the virtual environment, leading to a more satisfying and enjoyable user experience.

Lastly, there are medical applications. In research particularly, by accurately simulating human motion, researchers can study the mechanics of different motions and explore the effects of various factors on human performance and health. More precisely, a prior estimation of the impact of some treatment might be provided by tuning parameters of some accurate motion simulator. Doing so could reduce the quantity of ressourceful experimental data to be recorded from a patient to perform such an analysis for example. In brief, the study of the simulation of human movements is motivated by its broad range of applications ranging from entertainment to medical research.

Actually, creating physiologically accurate movements requires a deep understanding of human anatomy and biomechanics, as well as the ability to create algorithms and models that can reproduce realistic motions. Achieving this level of realism is a challenging but

critical step for all the applications described above, among many more. On top of that, controlling realistically an accurate human model becomes very important. Using reliable models, the task of performing a given motion is challenging and even more when the objective is to provide realistic movements and dynamics.

Presently, Reinforcement Learning (RL) and Neuro-inspired controllers are part of the main approaches that are considered for controlling a reliable human model. The latter is directly linked with the physiological phenomenons that occur within the Central Nervous System (CNS). This approach aims at understanding the underlying neurological interactions in the brain and nervous system that allows control of the human body.

On the other hand, the RL approach is becoming more and more popular these days with the rise of artificial intelligence in many domains. It differs from the previous approach because it tends to learn directly how to control the model in order to perform a certain set of tasks.

Consequently, in some scenarios, the learned controller might miss the pattern and behavior of real human motion. Analyzing what leads to human-like motions when trying to come up with trained controllers in a RL framework is thus critical. Therefore, this is the approach chosen in this thesis. In particular, the 2-dimensional motion of a realistic body will be studied to provide some insights on ways to develop simple controllers with deep RL.

Objectives Description

The main objective of this thesis is to develop a controller able to make a musculo-skeletal model move. Using RL as the main block for developing the controller, this thesis is based on 3 axes:

- Leveraging the computational cost of an accurate model
- Providing an accurate description of the method used
- Training and analyzing a controller

First of all, accurate simulators such as OpenSim are freely available and allow the study of various cases. It comes at the expense of computational cost, especially in a reinforcement learning method. Indeed, when considering a 3-dimensional environment with an incomplete body, state-of-the-art controllers typically require hundreds of CPUs and several GPUs to train neural networks effectively. Reducing the reliance on the simulator to obtain transitions is directly associated with improving sample efficiency. By minimizing the need for extensive simulation-based data generation, the learning process becomes more efficient and resource-friendly. The first objective is thus to see how is it possible to reduce the need for accurate simulators to leverage the computational cost of training a controller. This will be tackled by exploring the possibility of using model-based RL methods to increase sample efficiency. The extent to which a model can be used will be analyzed. In addition, this thesis also aims to determine whether simple networks and methods can be effectively

utilized to learn complex behavior while utilizing reasonable computational resources. In this context, different neural networks and methodologies are explored to see which can be employed to acquire specific behavioral patterns without requiring excessive computational resources. Hence, it involves testing different network architectures and algorithms.

Another objective is to provide a full description of the methods used in order to facilitate the reproducibility of the results presented and improvements of the network architectures and RL algorithms used. Even if only classical algorithms and networks are used throughout this work, it seems important to give a full insight into the methods employed. As such, the training procedure and the parameters impacting the performances will be exposed.

Lastly, this work aims also to be able to train an agent to move. Then, an analysis of the obtained results for the learned behavior will be conducted. Not only focusing on the performance of an agent but also looking qualitatively at the behavior developed to compare it with humans. The ultimate objective is to have an agent that exhibits a human-like gait and to understand the underlying causes.

Related Work

One of the objectives of this study is to try to reduce the computational cost associated with the training procedure. An essential aspect closely related to this objective is the investigation of sample efficiency, which remains a challenging topic in the field of reinforcement learning. To address this challenge, a recent and innovative RL algorithm called Risk Averse Value Expansion (RAVE) [1] has been introduced. This algorithm combines both model-free and model-based RL approaches. Notably, RAVE employs learned models to estimate the transition function of the environment, leading to improved sample efficiency. Within the realm of RL algorithms that utilize an environmental model, two notable approaches are Stochastic Ensemble Value Expansion (STEVE) [2] and Model-Based Value Expansion (MBVE). These preceding algorithms serve as foundational pillars upon which RAVE builds, incorporating their insights and techniques. By studying and comparing these various RL algorithms, with a focus on sample efficiency and reducing computational costs, valuable insights can be gained to develop more efficient and effective approaches to training RL models.

The study of motion is not only restricted to human motion and as such, some have studied the capabilities of neural networks to adapt to morphological changes in the model used [3]. The highlight is thus on the adaptability of the controller to changes in the configuration of the agent controlled, it showed that the underlying architecture of the controller is quite an important factor to take into account. Some work focused also on the application of deep learning for human motion simulation for various sets of tasks [4, 5]. In addition to that, focusing on character animation, some research also studies human motion but the emphasis is put on character animation [6].

Related to controllers, other more classical methods for optimal control are used to tackle similar problems [7, 8]. Differing from RL methods, neurologically inspired controllers were

developed to reproduce human motion in a variety of tasks [9]. This type of controller requires nonetheless a certain amount of fine-tuning. In fact, it results from studying humans and how the CNS operates the human body.

Similarly, Central Pattern Generator (CPG) are neural circuits that generate rhythmic patterns of activity in organisms, including humans and animals. These circuits are often found in the spinal cord or brainstem and are responsible for generating rhythmic movements such as walking, swimming, or flying. In robotics, CPG based controllers are used in order to generate oscillatory patterns for rhythmic motor behaviors such as walking. The study of CPG has provided valuable insights into the generation of rhythmic patterns and locomotion control, both in biological systems and the field of robotics. In fact, in the field of robotics, CPG based controllers may be used as a building block in order to produce walking patterns [10].

As a last related work, some work studied the range of joint torques and angles to provide state-dependent limits on torques. So, optimization problems could be solved in the torque actuation domain instead of the common muscle actuation domain [11]. In fact, they transform an optimal control problem in the muscle actuation space to an equivalent one in the torque actuation space. This methodology can be used for trajectory optimization problems as well as policy learning using RL.

Part I
Key Theoretical Concepts

1 Key Reinforcement Learning Notions

1.1 General concepts

Reinforcement Learning (RL) is a subfield of Machine Learning (ML) that focuses on how an agent can learn to make decisions sequentially in an environment to maximize a notion of cumulative reward. The RL formalization can be seen as an optimization problem that aims at maximizing the total reward denoted J. In fact, the goal is to find an optimal policy π^* that maximizes J such that the problem to be solved is:

$$J^{\pi^*} = \max_{\pi} J^{\pi} \tag{1}$$

Moreover, RL involves an agent interacting with an environment and learning through a trial-and-error process. The agent takes actions in the environment, receives feedback in the form of rewards or penalties, and aims to learn an optimal policy (π^*) that guides its decision-making process. The classical interaction scheme of the agent with the environment is depicted in Figure 1. A key mathematical framework within RL is the definition of the RL problem as a Markov Decision Process (MDP). Using this formalization, the following objects can be defined:

a) State Space (S): A set of possible states that the environment can be in. The states are denoted s such that $s \in S$.

b) Action Space (A): A set of possible actions that the agent can take. The actions are denoted a such that $a \in A$.

c) Transition Probability (T): A function that defines the probability of transitioning from one state to another when a particular action is taken. It is denoted as $T(s, a, s\prime)$ and represents the probability of transitioning from state s to state $s\prime$ after taking action a in the environment.

   Strictly speaking, it can be denoted as

$$T(s, a, s\prime) \equiv p(s\prime \mid s, a) \; \forall s, s\prime \in S \; , a \in A$$

   This transition probability corresponds directly to the interaction with the environment.

d) Reward Function (r): A function that specifies the immediate reward the agent receives upon transitioning from one state to another. It is denoted as $r(s, a, s\prime)$ and represents the reward obtained after taking action a in state s and transitioning to state $s\prime$.

e) Discount Factor (γ): A value between 0 and 1 that determines the importance of future rewards compared to immediate rewards. A higher discount factor values future rewards more. This discount factor is essential for the convergence of some equations when infinite time horizon problems are considered.

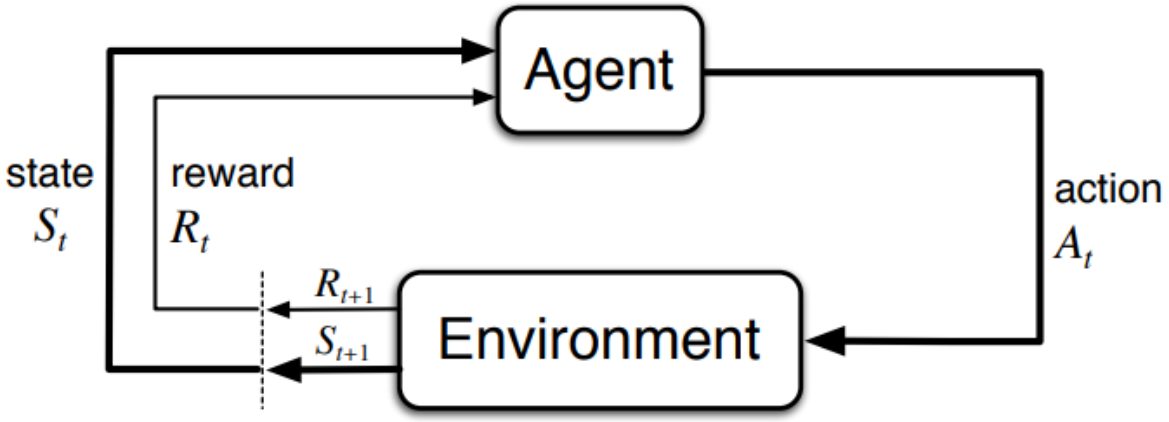


Figure 1: RL Environment Interaction

The deterministic policy that an agent follows is denoted $\pi(s)$ such that:

$$\pi(s) : S \mapsto A$$

In the more general case where π represents a conditional probability distribution, it is denoted as :

$$a \sim \pi(. \mid s) \quad \forall s \in S \ , a \in A$$

For this problem, we want to consider an infinite time horizon problem, for this purpose, the discount factor γ will be very useful. An important function to introduce for further developments is the value function denoted $V_\pi(s)$. This represents the discounted cumulative reward that is expected starting from state s and following policy π. In fact, in this case for an infinite horizon, the value function V is preferred to J since the expression of V is bounded if the reward is bounded. Indeed, by using the Bellman equation for the V function we have that:

$$V(s) = r + \gamma \cdot \sum_{s\prime \in S} P(s\prime \mid s) V(s\prime)$$

Where $P(s\prime \mid s)$ represents the probability to transition from state s to state $s\prime$ thus with γ between 0 and 1 the value function is bounded. By denoting G_t as the cumulative reward starting at time t such that :

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$$

One can rewrite the value function as:

$$V(s) = \mathbb{E}\left[G_t \mid s_t = s\right]$$

In order to show explicitly the policy in this formulation, the expected cumulative return of such a policy for infinite time horizon problems can be stated as:

$$\begin{aligned} V_\pi(s) &= \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \cdot r_{t+1} \mid a_t \sim \pi(. \mid s_t), s_0 = s\right] \\ &= \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t, s_{t+1}) \mid a_t \sim \pi(. \mid s_t), s_{t+1} \sim p(. \mid s_t, a_t), s_0 = s\right] \end{aligned} \tag{2}$$

Where $s_0 = s$ corresponds to the initial state at time-step $t = 0$. The optimization problem that is to be solved can be summarized as:

$$V_{\pi^*}(s) = \max_\pi V_\pi(s)$$

And the objective is to find the optimal policy π^* such that:

$$\pi^*(s) = \arg\max_\pi V_\pi(s)$$

In addition, a very important function is the Q-function, denoted as $Q(s, a)$, which represents the expected discounted cumulative reward that an agent can achieve by taking action a in state s, following a specific policy. It quantifies the desirability of taking a particular action in a given state. The Bellman equation for the Q-function is defined as:

$$Q(s, a) = \mathbb{E}\left[r + \gamma \cdot \max_{a\prime} Q(s\prime, a\prime)\right]$$

Where r is the immediate reward obtained from taking action a in state s, $a\prime$ is the next action to be taken in state $s\prime$, and $\max_{a\prime}$ denotes selecting the action that maximizes the Q-value in the next state $s\prime$. The Q-function is closely related to the value function through the following relationship:

$$V(s) = \max_a Q(s, a)$$

Which states that the value of a state is equal to the maximum Q-value over all possible actions in that state. By iteratively applying the Bellman equation for the value function or the Q-function, RL algorithms can estimate these functions through value iteration, policy iteration, or other iterative methods. These estimations then guide the agent's decision-making process to select actions that maximize the expected cumulative rewards in a given environment.

Overall, the Bellman equation provides a key framework for understanding the interplay between the value function and the Q-function, enabling the development of RL algorithms that can effectively learn and optimize behavior in various environments.

1.2 Classical Approaches

There are many classical approaches to solving the previously described optimization problem. Basically, there are model-free and model-based approaches. The model-based

approaches can either be based on a given model of the environment or learn the model such as Model Based Policy Optimization (MBPO) [12] or Model-Based Value Expansion (MBVE) [13] algorithm that will be used.

In the category of model-free methods, there are direct policy optimization methods that consist of learning directly the policy such as REINFORCE [14].

Another method is the Q-Learning approach, which consists of learning the Q-function corresponding to an environment and then selecting the best action to perform in order to maximize the Q-function on the actions. Such algorithms are often used with discrete action space [15, 16].

Lastly, there are also actor-critic methods that are a mix of Q-learning and policy optimization. A well-known algorithm and the one that will be used in this work is Deep Deterministic Policy Gradient [17]. This approach is mainly based on the Temporal Difference (TD) equation for updating the action-value function (Q-function) corresponding to the critic network. The TD error for the critic network can be written as:

$$\underbrace{r + \gamma \cdot Q(s\prime, a\prime)}_{\text{TD Target}} - Q(s, a) \tag{3}$$

Where r is the immediate reward for transitioning from state s to state $s\prime$ taking action a and $a\prime$ is such that $a\prime \sim \pi(. \mid s)$. The first term is called the TD target. In the following, DDPG and MBVE are detailed in the context of this project.

1.3 DDPG

Deep Deterministic Policy Gradient (DDPG) is an algorithm that simultaneously learns a Q-function and a policy. By utilizing off-policy data and the Bellman equation, it learns the Q-function, which is then employed to learn the policy. The following mathematical developments are inspired by [18]. The motivation behind this approach is similar to Q-learning, as it is driven by the idea that if the optimal action-value function $Q^*(s, a)$ is known, then the optimal action given by the policy $\pi^*(s)$ for any given state can be determined by solving:

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

DDPG employs a unique approach to handle environments with continuous action spaces by interleaving the learning of an approximation to $Q^*(s, a)$ and an approximation to $\pi^*(s)$. This adaptation specifically addresses the challenge posed by the computation of the max over actions in $\max_a Q^*(s, a)$. In scenarios where the number of actions is finite and discrete, evaluating the max is straightforward. Indeed, the Q-values for each action can be computed individually and be directly compared, which also provides the action that maximizes the Q-value.

However, in the case of continuous action spaces, evaluating the entire space is not feasible, and solving the optimization problem becomes quite complex. Thus computing

$\max_a Q^*(s,a)$ with conventional optimization methods would be excessively time-consuming. Moreover, since this computation would be required every time the agent intends to take an action in the environment.

Due to the continuous nature of the action space, the function $Q^*(s,a)$ is assumed to be differentiable with respect to the action. This enables the formulation of an efficient, gradient-based learning rule for a policy $\pi(s)$, which takes advantage of this characteristic. Rather than executing an expensive optimization subroutine whenever $\max_a Q(s,a)$ needs to be computed, it can be approximated as $\max_a Q(s,a) \approx Q(s,\pi(s))$.

For this method, 2 neural networks are needed: one for the Q-function $Q(s,a)$ and one for the function approximating $\max_a Q(s,a)$ that is denoted $\pi(s)$. The parameters of the neural network $Q(s,a)$ will be denoted by ϕ such that $Q_\phi(s,a)$ is called the critic network. For the neural network corresponding to $\pi(s)$, the parameters of the network are denoted by θ such that $\pi_\theta(s)$ is called the actor network.

This algorithm is an off-policy algorithm. It means that the transitions can be collected by a different policy than the one being trained. In particular, in this algorithm, the transitions are collected and inserted into a replay buffer called $\mathcal{D}$ as this has proven to be efficient [19]. The replay buffer $\mathcal{D}$ stores one-step transitions in the form $(s,a,r,s\prime,d)$ where s is the starting state, the action a, r the reward obtained by transition from s to $s\prime$ applying action a, the next state $s\prime$, the terminal flag d. The terminal flag d is provided by the environment and equals 1 if the state $s\prime$ is terminal 0 otherwise. A terminal state terminates an episode.

In order to train the critic network, the TD is a key equation. The equation (3) can be adapted for this algorithm, thus the TD-target is thus:

$$r + \gamma \cdot (1-d) \cdot Q_\phi(s\prime, a\prime) \tag{4}$$

However, to avoid using the same network to compute the target as the one being optimized. A delayed version of each network is used in order to compute the target in equation (4). In fact, 2 more networks are used with parameters ϕ_{tar} and θ_{tar} respectively for the critic and the actor, these are called the target networks. These target parameters are updated during the algorithm such that:

$$\phi_{tar} \leftarrow (1-\rho)\cdot\phi_{tar} + \rho\cdot\phi$$
$$\theta_{tar} \leftarrow (1-\rho)\cdot\theta_{tar} + \rho\cdot\theta$$

Where ρ corresponds to a delay parameter, usually close to 0. With all the above, the definitions for the loss critic can be written as follows:

$$\mathcal{L}(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s\prime,d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - (r+\gamma(1-d)Q_{\phi_{tar}}(s\prime,\pi_{\theta_{tar}}(s\prime))\right)^2\right]$$

The objective is to minimize this loss called the mean-squared Bellman error. For the actor-network, the objective is to maximize the following expression:

$$\mathop{\mathbb{E}}_{s\sim\mathcal{D}}\Big[Q_\phi(s,\pi_\theta(s))\Big]$$

The loss to minimize for the actor can thus be written:

$$\mathcal{L}(\theta, \mathcal{D}) = - \mathop{\mathbb{E}}_{s \sim \mathcal{D}} \Big[Q_\phi(s, \pi_\theta(s)) \Big]$$

Finally, the main steps of the algorithm used in the case of this project can be found in Algorithm 1.

1.4 MBVE

Model-Based Value Expansion (MBVE) is a Reinforcement Learning (RL) approach that combines model-based planning with value function approximation. The main idea behind MBVE is to leverage a learned dynamics model of the environment to generate rollouts and estimate value functions more efficiently.

In MBVE, an RL agent first learns a dynamics model denoted $\hat{f}$ that predicts the next state $s\prime$ and the reward r given the current state s and action a. This model is then used to generate multiple rollouts starting from states collected by interacting with the environment. Each rollout consists of a sequence of states, actions, and rewards. Additionally, in this context, the sequence is also composed of the terminal signal d produced by $\hat{f}$.

Using the generated rollouts, MBVE estimates the value function by considering the immediate rewards and the expected values of the next states. By incorporating the learned dynamics model, MBVE can propagate rewards more accurately and efficiently through the rollouts.

MBVE employs a recursive value estimation process known as the TD-k trick in the original paper [13]. It iteratively updates the value estimates for each generated state $\hat{s}$ by considering the immediate reward and the expected value of the next state.

Once the value function is estimated, MBVE can use it to improve the agent's policy. By selecting actions that maximize the estimated value as explained in previous sections, the policy is refined to make more informed decisions. Additionally, MBVE includes a model refinement step, where the dynamics model is updated using the collected transitions from the environment. This helps improve the accuracy of the model and enhances the quality of future rollouts and value estimates. Overall, MBVE combines model-based planning, value function approximation, and bootstrapping to learn more efficient value estimates and improve the RL agent's policy. It offers a trade-off between exploration and exploitation by leveraging the learned dynamics model to guide the agent's decision-making process.

In fact, this algorithm is well suited to be used with actor-critic methods such as DDPG. The TD target from (3) is changed within this algorithm. In the original paper, a few assumptions are made for the method. It is assumed that the reward function is known and that the learned dynamics model is accurate enough.

For the following, the dynamics model of the environment $\hat{f}$ is assumed to take as input the current state s and current action a and give as output a reward $\hat{r}$, the next imagined state $\hat{s}\prime$ and the terminal signal $\hat{d}$ (representing the probability of being in a terminal state) such that:

$$\hat{f} : S \times A \mapsto S \times \mathbb{R} \times [0, 1]$$

While MBVE can be used on top of an actor-critic method, this method will build upon DDPG. As such, an actor-network π and a critic network Q will be used. Parameters θ/θ_{tar}

and ϕ/ϕ_{tar} are respectively used for the actor and the critic.

In order to compute the TD target, a rollout length of H is fixed and a replay buffer $\mathcal{D}$ is considered to store transitions $(s, a, r, s\prime, d)$ collected from the true environment. The first step is to generate a rollout of length H using $\hat{f}$ starting from a transition $(s, a, r, s\prime, d)$ that will be denoted:

$$\tau_0 = (s_{-1}, a_{-1}, r_{-1}, s_0, d_{-1})$$

Then for $t \in [1, H]$, H transitions are imagined using $\hat{f}$ such that:

$$\begin{pmatrix} \hat{s}_t \\ \hat{r}_{t-1} \\ \hat{d}_t \end{pmatrix} = \hat{f}(\hat{s}_{t-1}, \hat{a}_{t-1})$$

$$\tau_t = (\hat{s}_{t-1}, \hat{a}_{t-1}, \hat{r}_{t-1}, \hat{s}_t, \hat{d}_{t-1})$$

With $\hat{a}_{t-1} = \pi_{\theta_{tar}}(\hat{s}_{t-1})$. The value $\hat{Q}_k$ can be defined as follows:

$$\hat{Q}_k = \sum_{i=k}^{H-1} \gamma^{i-k} \cdot \prod_{j=k}^{i-1} (1 - \hat{d}_j) \cdot \hat{r}_i + \gamma^{H-k} \cdot \prod_{m=k}^{H-1} (1 - \hat{d}_m) \cdot Q_{\phi_{tar}}(\hat{s}_H, \hat{a}_H)$$

From this expression, the loss for the critic can be defined as:

$$\mathcal{L}(\phi, \tau_{0:H}) = \frac{1}{H+1} \sum_{k=-1}^{H-1} \left(Q_\phi(\hat{s_k}, \hat{a_k}) - \hat{Q}_k \right)^2$$

And by denoting the actor's loss by:

$$\mathcal{L}(\theta, \tau)$$

Having the same expression as for DDPG. One may also define $\mathcal{L}(\zeta, \tau)$ as the dynamics model loss of $\hat{f}$ with ζ the parameters of $\hat{f}$ such that $\hat{f}_\zeta$.

Finally, the main steps of the algorithm used in the context of this project are summarized in algorithm 2. The version presented in this work is a bit different than the one presented in the original paper because the terminal signal d needed to be used in the scope of this project as well as a neural network for the dynamics model.

2 Definition of the RL Problem

2.1 Presentation of the environment

The RL environment that will be used in this thesis originates from the *Learn to Move - Walk Around* competition organized by the NeurIPS conference in 2019. The NeurIPS 2019 competition "Learn to Move - Walk Around" focused on advancing locomotion control for simulated characters using mathematical and computational techniques.

In the context of this competition, locomotion control is referring to the ability to control the actions of a musculo-skeletal model in a simulated environment. Participants in the NeurIPS 2019 competition were challenged with the task of creating a locomotion controller capable of following a target velocity map within the given OpenSim-RL simulation environment [20]. This environment is built upon OpenSim [21]. The controllers were evaluated based on their effectiveness in following these velocity targets and on effort minimization. The form of the total reward used is as follows:

$$J^{\pi} = R_{alive} + R_{step} + R_{target}$$

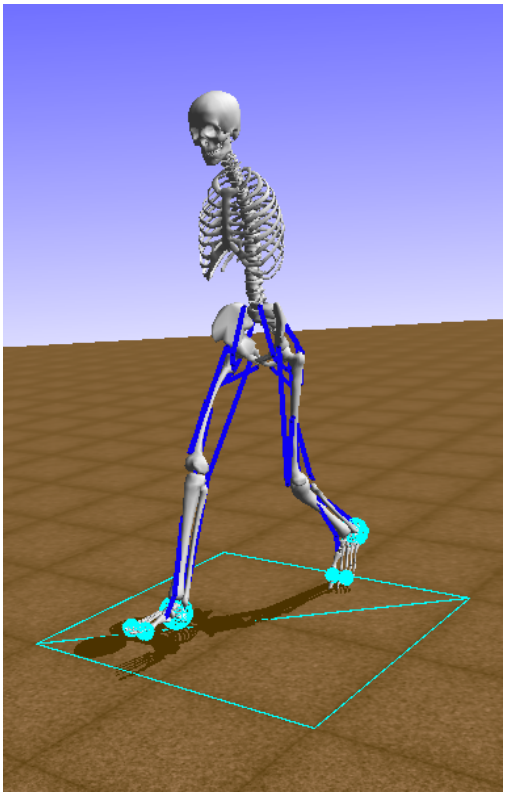
Where R_{alive} corresponds to a bonus for not falling. Then R_{step} is a term that gives a bonus for making a step with minimum effort and following the target velocity. The last term R_{target} is a bonus for reaching the target of the velocity field.

Concerning the control aspect, the controlled agent is a musculo-skeletal model from OpenSim. More precisely, the control is directly performed on the activation of the different muscles of the model. The controlled agent is depicted in Figure 2a. The model consists of a 3D human musculo-skeletal model constituted by seven segments with joints accounting for 8 internal Degrees of Freedom (DOF). The model is actuated by 22 muscles, 11 on each leg.

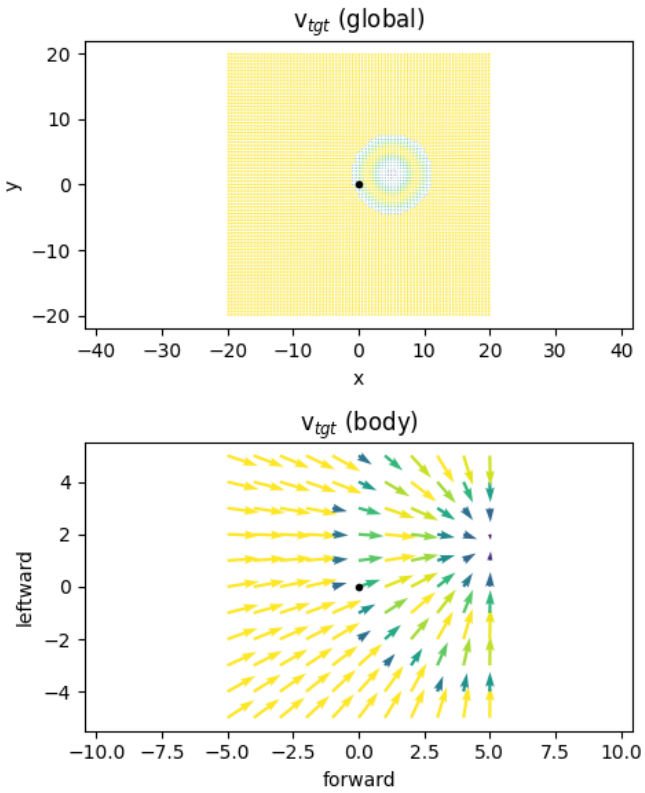
The target velocity vectors to follow in the context of the competition can be found in Figure 2b. There are 2 fields: a global one and a local one. The global represents the full target velocities and corresponds to a 5m $\times$ 5m map. On the hand, the local one (or body in the figure) corresponds to the local targets reporting to the agent's body.

This competition serves as the starting point for this thesis. Since the environment described above is quite complex and computationally costly, the environment will be simplified for this work. The simplifications hold in 2 main components:

Firstly, the agent evolves now in a 2-dimensional environment and as such there is 1 internal DOF less per leg due to the absence of abduction in the hip. The second change concerns the objective function of the environment, the objective will be only to move forward as fast as possible. Next follows a more detailed description of this simplified environment.



(a) Musculo-skeletal model of the simulation environment



(b) Target velocity vector field

Figure 2: NeurIPS Human Motion Problem

2.2 State Space Description

The state space of the simplified environment corresponds to the body state of the agent. The state space denoted S is such that $S \subseteq \mathbb{R}^{97}$. A state s is composed of 97 values describing the body. Some of these values make sense only in the 3-dimensional environment but will be part of a state anyway, for example, the abduction of the hip since the motion is planar. In fact, these 97 are provided by the environment from OpenSim-RL. The first 9 values describe the pelvis state. These are :

- Pelvis height [m]
- Pelvis pitch angle [rad]
- Pelvis roll angle [rad]
- Pelvis forward velocity [$\mathrm{m\,s^{-1}}$]
- Pelvis leftward velocity [$\mathrm{m\,s^{-1}}$]
- Pelvis upward velocity [$\mathrm{m\,s^{-1}}$]
- Pelvis pitch angular velocity [$\mathrm{rad\,s^{-1}}$]
- Pelvis roll angular velocity [$\mathrm{rad\,s^{-1}}$]
- Pelvis yaw angular velocity [$\mathrm{rad\,s^{-1}}$]

Then the state of each leg is successively described, for each leg there are 44 values. These 44 values can be put into 3 categories: the ground reaction forces, the joint state, and the muscles state. The ground reactions account for 3 values, one per direction, and are normalized to the body weight of the model. The joint state is composed of 8 values:

- Hip abduction angle [rad] and angular velocity [$\mathrm{rad\,s^{-1}}$]
- Hip extension angle [rad] and angular velocity [$\mathrm{rad\,s^{-1}}$]
- Knee extension angle [rad] and angular velocity [$\mathrm{rad\,s^{-1}}$]
- Ankle plantar flexion angle [rad] and angular velocity [$\mathrm{rad\,s^{-1}}$]

Finally, for each muscle for each leg, there are 3 values:

- Muscle fiber force, normalized to maximum isometric force
- Muscle fiber length, normalized to the optimal length
- Muscle fiber velocity, normalized to optimal length per second

In total, there are 97 states. However, some of them are supposedly constant in a planar motion but will be part nonetheless of the state.

Once the states are described, an important thing to notice is that a state s is completely agnostic to time and also to its $x - y$ position since only the height ($z - axis$) is given. That means that if the agent is performing optimally close to the initial position it will also perform optimally further away.

2.3 Action Space Description

An important notion to describe is also the action space A. In this case, there are 22 muscles of which the agent controls directly the excitation levels of the model muscles. In fact, A is such that $A = [0, 1]^{22}$ and thus an action a is such that $a \in [0, 1]^{22}$.

As said above, there are 11 muscles per leg and those are summarized in Table 1. The muscles are also listed with their impact on the different joints.

Name	**Role**
Hip Abductor (HAB)	hip abductor
Hip Adductor (HAD)	hip adductor
Hip Flexor (HFL)	hip flexor
Glutei (GLU)	hip extensor
Hamstrings (HAM)	biarticular hip extensor and knee flexor
Rectus Femoris (RF)	biarticular hip flexor and knee extensor
Vastii (VAS)	knee extensor
Biceps Femoris, Short Head (BFSH)	knee flexor
Gastrocnemius (GAS)	biarticular knee flexor and ankle extensor
Soleus (SOL)	ankle extensor
Tibialis Anterior (TA)	ankle flexor

Table 1: List of Muscles

2.4 Reward function

The reward function of the simplified environment should be modified to cope with the new objective i.e. to run as fast as possible. The reward function developed in this section is an adaption of the one used within the competition. By calling r the reward at a given time step, r is composed of 2 main components:

$$r = b_{alive} + b_{step}$$

The first term b_{alive} corresponds to a bonus for not falling such that:

$$b_{alive} = \begin{cases} 0.1 & \text{if standing up} \\ -100 & \text{if agent has fallen} \end{cases}$$

In fact, the criterion to determine if the agent fell or not corresponds to the definition of a terminal state. A state is terminal if the pelvis height is below 0.6m.

Then, the term b_{step} gives a bonus for moving forward fast and making steps with minimum effort. This term equals 0 if no new footstep is made. Otherwise, b_{step} is composed of 3 terms:

$$b_{step} = b_{speed} + b_{new\_step} - p_{muscle}$$

Denoting by i the ith time step corresponding to the last footstep and by j the jth time step corresponding to the new footstep, Δt_{ij} is the time between the two footsteps such that:

$$\Delta t_{ij} = (j - i) \cdot \Delta t$$

With Δt the simulation time step.

Then b_{speed} is the average forward speed of the pelvis between time step i and j. The bonus b_{new_step} is such that:

$$b_{new\_step} = 10 \cdot \Delta_{ij}$$

This form is used to avoid encouraging making very small steps. The last term p_{muscle} is the penalty for muscle activation and aims at minimizing the effort. By noting A_m^k as the activation level of muscle m at time step k and M as the set of all muscles of the model:

$$p_{muscle} = \sum_{k=i}^{j} \sum_{m \in M} (A_m^k)^2 \cdot \Delta t$$

This time integration of the squared muscle activation tends to approximate muscle fatigue and is often minimized in locomotion simulations [20].

3 Bio-Mechanical Modeling

Now that the state space S, action space A, and reward function are completely defined. The type of model used for the musculo-skeletal agent is further detailed with the main considerations.

3.1 Muscle Model

Since the control is done on the muscle excitation levels, it is important to understand the underlying muscle model. For this environment, the well-known Hill-type muscle model is used. The Hill-type muscle model used in OpenSim involves several important equations to describe muscle behavior. Here are the key elements associated with the Hill-type muscle model:

1. The Contractile Element (CE):

   The contractile element models the capability of the muscle fibers to generate force based on their length, velocity, and activation rate. This component is the active element of the model. It combines the force-length and force-velocity relationships to compute the contractile element force (F_{CE}). These 2 relationships are often represented in their normalized form such that the fiber length is normalized by its optimal length, the force by the maximum isometric force, and the velocity by the maximum contraction velocity. An example of such relations is shown in Figure 3 where the active curve corresponds to this element for the force-length relationship.

2. The Parallel Elastic Element (PEE):

   The parallel elastic element is used for modeling the compliance or elasticity of the muscle's passive components that are in parallel with the contractile element. It accounts for the elastic properties of muscle fibers and connective tissues. The force exerted by the parallel elastic element (F_{PEE}) is also a function of the muscle length L^M. Such a relationship is shown in Figure 3 corresponding to the passive curve.

3. The Series Elastic Element (SEE):

   The series elastic element is for modeling basically the tendons and other passive structures in series with the muscle fibers. They are the elements linking the contractile element to the skeletal system in order to transmit the generated forces. The force exerted by the series elastic element (F_{SEE}) is a function of the tendon length L^T.

While the activation levels matter for the CE, it is also important to know how OpenSim computes the activation dynamics. In OpenSim, the activation dynamics of muscles are commonly modeled using a first-order differential equation. This equation describes the relationship between the time derivative of muscle activation $a(t)$ and the muscle excitation level $u(t)$. Basically, the equation is:

$$\dot{a}(t) = \frac{u(t) - a(t)}{\tau(a(t), u(t))}$$

Where $\tau(a(t), u(t))$ is a variable time constant [22]. In practice, a bunch of parameters are needed for the described dynamics, these muscles parameters are given within an *.osim* file describing the OpenSim model. This model is provided in the original NeurIPS competition environment. By combining these components, the muscle model in OpenSim provides a comprehensive representation of muscle behavior, accounting for both contractile properties and passive elasticity.

3.2 Contact Forces Model

Lastly, since a walking motion is simulated, it is crucial to use a model for the contact forces between the body and the ground. This interaction between the ground and the model feet is done with the help of contact spheres, 3 on each foot. This OpenSim model uses the Hunt and Crossley contact force model. The Hunt and Crossley contact force model [23] is often used to describe contact forces with dry friction in multibody dynamics simulations.

It includes both normal and tangential contact forces. It also takes into account the energy being dissipated during impacts. Basically, a coefficient of restitution is a main component of this model, this coefficient represents the ratio of relative velocities before and after the impact of bodies that collides.

Once the muscle forces are computed [24] as the contact forces, the joint torques can be computed and the state of the musculo-skeletal model updated. The way OpenSim computes forward dynamics is detailed in the documentation [25].

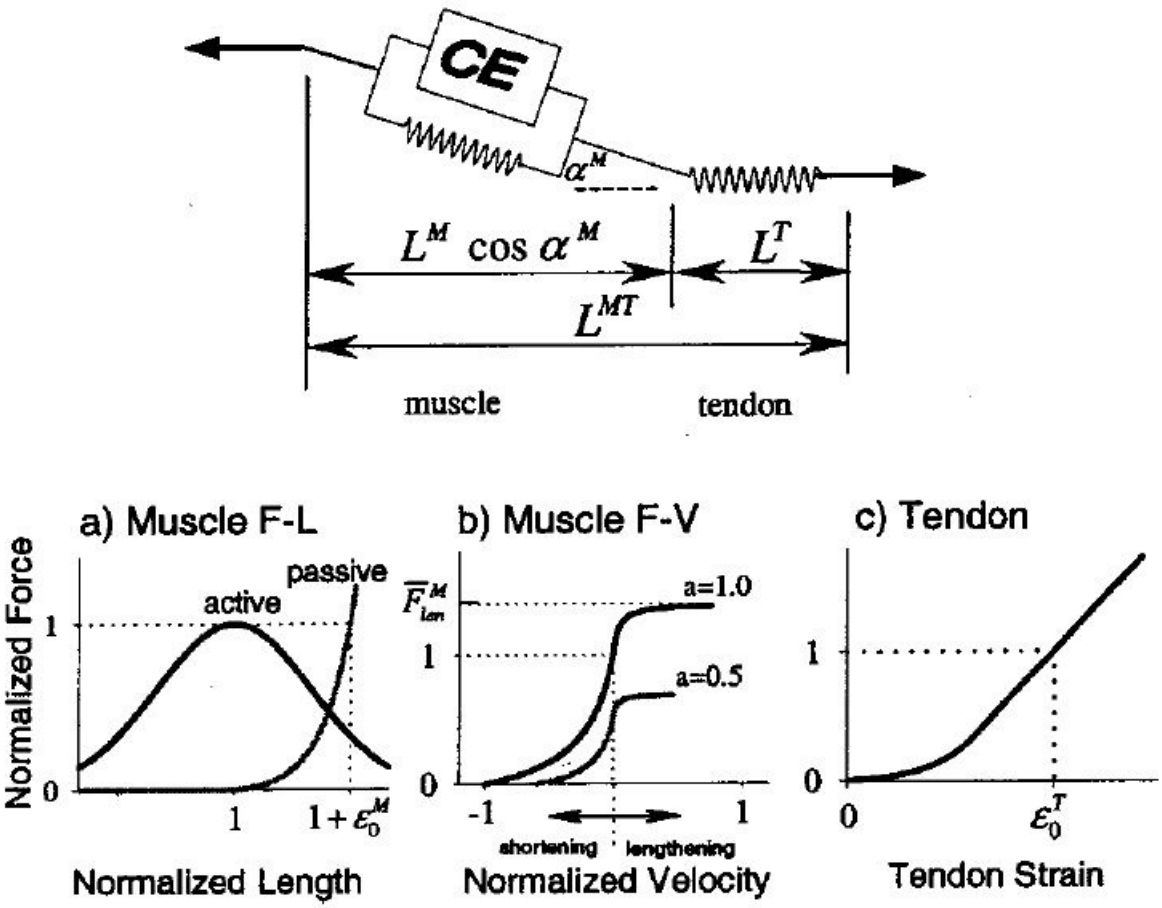


Figure 3: Schematic diagram of the Hill-type muscle model with Pennation Angle of the muscle fibers (α^M). Figure taken from [24].

4 State of the Art Review

State-of-the-art methods for addressing this problem include the top solution of the NeurIPS competition. The first-place solution, Risk Averse Value Expansion (RAVE) [1], drew inspiration from Model-Based Value Expansion (MBVE) and Stochastic Ensemble Value Expansion (STEVE). The key idea behind RAVE was to leverage ensembles of stochastic networks to enhance prediction quality. By incorporating uncertainty estimates into their approach, they aimed to improve the robustness of their model's predictions. The second-place solution [26] also employed an ensemble of networks but adopted a modified version of the TD3 algorithm [27]. They leveraged the ensemble approach to mitigate overfitting and enhance the stability of the learning process. In contrast, the third-place solution [28] utilized the Soft Actor-Critic (SAC) algorithm [29]. They employed a multivariate representation for the reward, enabling them to learn multiple Q functions simultaneously. This approach allowed for a more comprehensive exploration of the action space and improved policy learning.

All three solutions implemented a strategy of reshaping the original reward to facilitate gradual task learning. The first and third-place solutions employed a technique known as curriculum learning. They simplified the environment by initially teaching the agent to move forward before gradually introducing the target velocity. This curriculum-based approach aimed to facilitate a smoother learning process and accelerate convergence. By combining elements from model-based methods, ensemble learning, curriculum learning, and reinforcement learning algorithms, these top-performing solutions in the NeurIPS competition have significantly advanced the field and demonstrated effective strategies for tackling the given problem.

In addition, significant research efforts have been devoted to enhancing the original MBVE algorithm employed in this project, as evidenced by the aforementioned solutions. Several studies have proposed improvements to this approach by incorporating stochastic networks [30]. These extensions aim to introduce randomness and uncertainty into the model's predictions, enabling better adaptability to complex and uncertain environments.

Regarding the network architecture for modeling the environment dynamics, various approaches have demonstrated remarkable success. Some methods [31, 32, 33] have achieved notable results by leveraging deep reinforcement learning techniques to learn accurate and effective dynamics models. By utilizing sophisticated neural network architectures, these approaches have been able to capture intricate relationships and intricate dynamics within the environment.

Furthermore, alternative methodologies have explored the use of memory-based controllers. For instance, certain methods [34, 35, 3] have employed recurrent neural networks (RNNs) to incorporate memory and temporal dependencies into the control process. By leveraging the sequential nature of RNNs, these approaches have demonstrated enhanced capabilities in learning and executing complex control policies that require memory and sequential decision-making.

These advancements in the field highlight the continuous efforts to refine and extend the MBVE algorithm, improve the modeling of environment dynamics, and explore innovative controller architectures. By incorporating these developments, researchers aim to tackle the challenges posed by complex and dynamic real-world tasks more effectively.

Part II
Training Pipeline Design

5 Methodology

5.1 Main Steps

The design methodology is a crucial aspect of this thesis, as it outlines the systematic approach employed to create and develop the final solution employed to address the initial problem.

The design procedure is divided into three main steps:

1. The *learning algorithm*: two well-known algorithms namely Deep Deterministic Policy Gradient (DDPG) and Model-Based Value Expansion (MBVE) will be compared and discussed to choose the more suitable one for the OpenSim environment.

2. The *networks architectures*: three simple implementations of the different components needed within a RL framework such as the actor and critic will also be studied.

3. The *training procedure*: this part aims at highlighting the impact of different parameters in the training procedure. Mainly, the impact of pre-training and curriculum learning on performance. As this approach was used among the top solutions [36, 28] of the NeurIPS competition. Also, the impact of frame skipping and reward shaping is analyzed.

Throughout the analysis and comparison in each part, the choice is made to arbitrarily fix all the other parameters and not to test and compare every combination of the three main steps. This would be time-consuming in terms of computation time just to record all the necessary data. On top of that, the idea is also to have an insight of what is the individual impact and gains of the three different parts cited above. Hence, when comparing algorithms, the network architectures are fixed such as the training procedure. Following the idea to compare these different steps independently.

Lastly, all choices made in the different steps will be incorporated into the method that will be used to train the complex agent inside the OpenSim environment.

5.2 Test Environments

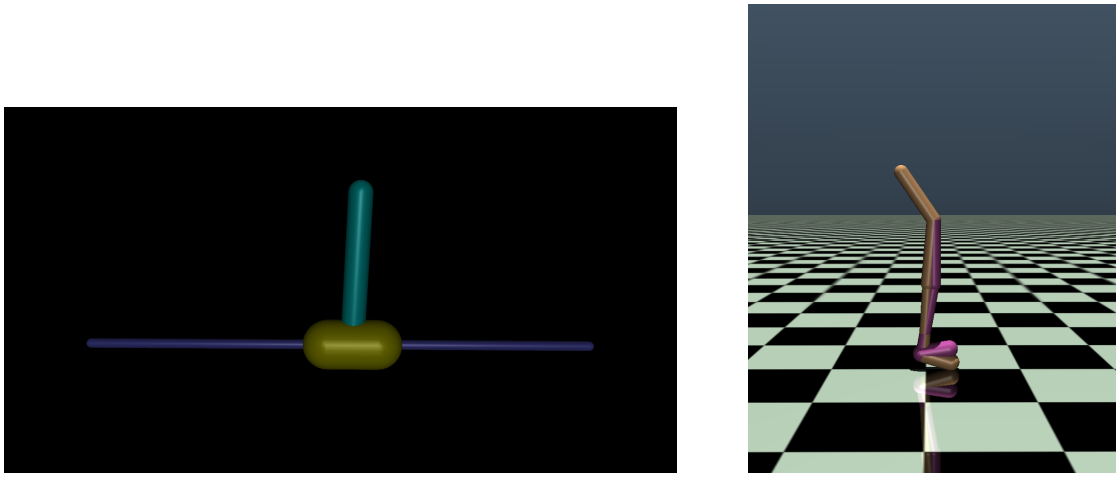
Even if the main goal is to train an agent for controlling a musculo-skeletal model, the choice of the different components is performed using much more simpler problems.

Since the environment of interest described in section 2 is computationally very heavy, the different tests for choosing the design won't be pursued in this environment but rather in two well-known environments often used for benchmarking. These environments will be

used only for design purposes.

These are the **Inverted Pendulum** problem and the **Walker2D** problem from the Gym [37] python library that uses MuJoCo [38] which is a physics engine. These libraries are very popular and are used notably in robotics and biomechanics.

These 2 environments will be only used for choosing the learning algorithm and the network architectures. Also, for the training procedure, only the frame-skipping part will be done in these 2 environments. The reason for that is that the pre-training and curriculum learning makes sense only for the initial OpenSim environment. The two above-mentioned environments are depicted in Figure 4.



(a) Inverted Pendulum (b) Walker2D

Figure 4: MuJoCo Environments

The inverted pendulum environment is mainly used to debug and test the different implementations. However, it will be used also in the discussion for the choice of the different components. In addition, the Walker2D environment is chosen because it is quite similar to the initial RL environment of interest. Indeed, it is composed of a torso, 2 legs, and 2 feet. It has in total 9 DOF just like the simplified OpenSim model. However, the states are different and the controller acts directly on the individual torque of each joint and no control on any muscle is present. Thus this environment can be seen as a highly simplified version of the initial problem but cannot replace the initial environment for the considered problem without further caution.

5.3 Comparison Procedure

The last important aspect of the design procedure is now to define explicitly what will be the criteria to differentiate the multiple approaches.
Firstly, as the main criterion, the direct performance of the learned policy π will be evaluated in the corresponding environment to obtain a measure of how well it is performing in the environment. The goal is to optimize the cumulative reward, this is the measure of

performance that will be used in this context.

Furthermore, the training duration will be considered, taking into account the time required for training. As one of the objectives of the thesis is to minimize computational expenses, the training time holds significant importance. Although the explicit computation time depends on the machine and set-up on which the different methods are run when comparing methods, the consistent use of the same machine allows for a meaningful comparison of computation times. In particular, the focus will be on comparing the training time required to achieve good performance across the methods. One could want to compare the quality of the estimated Q-Values but the problem is that it is quite complicated to collect the necessary data to provide a thoughtful analysis. If the agent performs well in the environment, it might suggest that the Q-Values are estimated correctly. Thus, it is assumed the performance will be enough as a criterion for the comparison.

Following the idea of being efficient in terms of computation, the inference time is also to be considered. The inference time is included in the training time as there are forward passes through the different networks. However, the inference time is important to have an idea of the computation cost of the trained controller when exploiting it.

The comparison criteria being defined, the next sections will focus on the use of these criteria to choose the best methods to apply to the initial environment.

5.4 Transition Collection

Since the environment that will be used is expensive in terms of computation, the procedure to collect the data is adapted. In the implementation, multiple environments run in parallel to collect each a certain number of transitions. These transitions are then gathered and added to a replay buffer. Then, one epoch of training is performed and the updated policy is sent back to the multiple environments in parallel to collect again a set of transitions.

Focusing on the replay buffer, the idea is to store all the transitions in a buffer. During training, batches of transitions are sampled from this buffer (with replacement) and these batches are used in the training algorithm. Such an approach has been shown to improve the training process [19]. The underlying idea is to consume a transition more than one single time. This method can be safely used with the learning algorithms that will be utilized since these algorithms are off-policy methods. This means that the current policy being trained can use transitions generated by another policy.

For the collection of the data, an additional method is used to tackle the problem of exploitation versus exploration trade-off. The famous ϵ-greedy algorithm [15] is implemented with a small variation. The original ϵ-greedy takes with a probability ϵ a random action from a uniform distribution. In this work, the random action taken with probability ϵ is not sampled from a uniform distribution anymore. Instead, a noise is added to the original action a and this noise is sampled from a normal distribution. As such the exploration method is

as follows:

$$a = \begin{cases} \pi_\theta(s) & \text{with probability } 1 - \epsilon \\ \text{clip}(\pi_\theta(s) + n, 0, 1) & \text{with probability } \epsilon \end{cases}$$

Where a is the final taken action, π_θ is the policy with parameter θ and and n is such that:

$$n \sim \mathcal{N}(0, \sigma^2)$$

In the context of this project, σ is chosen to be equal to 0.1 and ϵ to 0.2. Also, the clip function clamps the value to be within 0 and 1.

The motivation behind this variation of the ϵ-greedy method is that a completely random action is much more likely to yield a bad combination of the muscles' activation. In addition, the activation of a muscle is between 0 and 1, as such sampling uniformly an action from this interval leads to an expectancy of 0.5. Then, it means activating each muscle at 50% on average which is much more likely not to lead to gait-like motion. So, the idea is to explore actions near the ones that are chosen by the policy π.

6 RL Algorithms

The first design step is to choose which algorithms will be used to train the agent in the complex environment. As explained before, there are 2 algorithms to be considered:

- Model-Based Value Expansion (MBVE)
- Deep Deterministic Policy Gradient (DDPG)

The usage of the model-free approach (DDPG) is a state-of-art algorithm when it comes to RL. It is expected to have a reduced sample efficiency compared to MBVE but doesn't need the use of a model nor to learn a model. On the other hand, the MBVE algorithm is a hybrid approach and requires the use of a model of the environment. In this case, the model is to be learned during the training process and the model of the environment corresponds to a neural network.

6.1 Numerical Training Parameters

Since this part focuses on comparing 2 algorithms, the neural networks used and the training parameters have to be fixed.

For both algorithms there are an actor and a critic network as described in the DDPG algorithm in section 1.3. Additionally, for the model-based approach, there is another neural network that corresponds to the model of the environment as described in section 1.4. These networks can be denoted as :

$$\begin{aligned} \text{Actor} &\rightarrow \pi_\theta(s) \\ \text{Critic} &\rightarrow Q_\phi(s,\ a) \\ \text{Environment Model} &\rightarrow f_\zeta(s,\ a) \end{aligned}$$

Where ϕ, θ, and ζ represent the parameters of the different neural networks. Both the actor and the critic have an architecture corresponding to the variant Multi-Layer Perceptron (MLP) architecture that will be described more thoroughly in section 7.

For the model f_ζ, the same base blocks are used but combined in a parallel way. The estimated next state $s\prime$, estimated done state d, and the estimated reward r are computed in completely parallel blocks. They correspond to 3 distinct neural networks in parallel.

Now that the network architectures are fixed, the training procedure and parameters need to be fixed as well. The main parameters are summarized in Table 2 and they are chosen from common values used. The optimizer used to update the different networks is the Adam optimizer [39] which is also a classical choice to optimize the neural networks. The rollout length is only used for the model-based approach as described in the MBVE algorithm. It is not used for DDPG.

Concerning the frame skipping that is applied for both algorithms in both environments: The number of frames skipped is the same for the 2 environments. However, the simulation

Name	Symbol	Value
Discount Factor	γ	0.99
Learning Rate	α	$1e-4$
Soft Update Factor	τ	$1e-3$
Frame Skipping	-	10
Number of Epochs	-	1000
Updates per Epoch	-	256
Rollout Length	-	3

Table 2: Training Parameters

time-step Δt is different for the inverted pendulum and the walker:

$$\begin{aligned} \text{Inverted Pendulum} &\rightarrow \Delta t = 0.02\text{s} \\ \text{Walker2D} &\rightarrow \Delta t = 0.002\text{s} \end{aligned}$$

So the resulting equivalent simulation time-step is different in both environments. The in-depth analysis of frame skipping is done in section 8 and gives more information on frame skipping.

6.2 Simulation Results

The results are presented in a way to show the evolution of the performance of the agent in the environment throughout the learning process. The parameters of the neural networks are recorded every 10 epochs. For every recorded neural network, the performance on 20 runs is used to compute a mean and standard deviation to show the performance. The resulting performance for the pendulum environment is shown in Figure 5a and for the walker environment in Figure 5b.

Interestingly, the DDPG algorithm seems to outperform the MBVE algorithm in terms of performance for both environments. The model-based approach seems to never find a good policy to follow. On the contrary, the model-free approach seems to learn something as its performance gets better at some point. Looking at Figure 5a, DDPG has maximum performance on the pendulum environment since in this configuration the maximum cumulative reward possible is 100.

For the walker (Figure 5b), DDPG is also clearly better, the difference being that the agent can still learn after 1000 epochs but it is sufficient to compare both algorithms. Indeed, although the performance of the model-free approach seems to fluctuate, it is way better than the model-based one.

In terms of computational cost, the time needed to train for 1000 epochs is reported for both methods for both environments in Table 3. The format is hh:mm:ss for the times. From the training times, it is clear that the MBVE takes more time to train than DDPG for the

	MBVE	DDPG
Pendulum	6:20:56	3:12:13
Walker	14:37:43	2:19:48

Table 3: Training Times

same number of epochs. The inference time is not considered in this section as this will be the same for both algorithms since the neural networks are the same in both cases.

6.3 Discussion

The results presented previously tends to show that the DDPG algorithm is better than the MBVE. However, it should be the contrary if one refers to this paper [13] introducing the MBVE algorithm. The problem is that in this context, the model that is learned from the environment is not good enough to allow for the algorithm to be efficient. To confirm this intuition, the graphs in Figure 6 show the different errors for the 3 estimated quantities. In Figure 6a, the relative error of the estimated reward $\hat{r}$ and the estimated next state $\hat{s\prime}$ are shown. The relative error is computed as such:

$$e = \left\| \frac{x \ - \ \hat{x}}{x \ + \ \epsilon} \right\|$$

Where e is the relative error, x is the true quantity and ϵ is a constant equal to $1e-5$ for numerical stability.

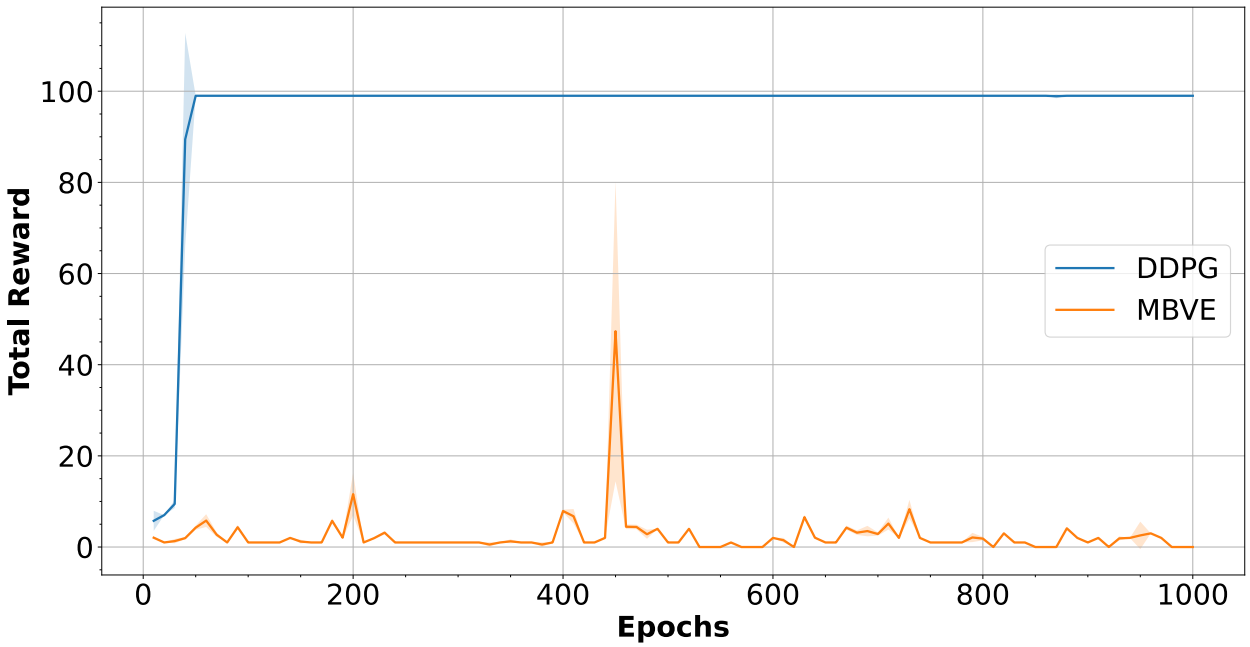
It is clear that the model of the environment is not accurate for the states especially since the relative error climbs up high sometimes. The order of magnitude between the predicted and the true state is quite important and that leads to poor performance. On the other hand, the reward is approximated quite correctly since the error is very low. Focusing on the done state, the error is defined differently as the estimated done state $\hat{d}$ represents the probability of being in a terminal state. The binary cross entropy loss is used as an error measure such that:

$$e_{bce} \ = \ -\left(x \cdot \log \hat{x} + (1-x) \cdot \log(1-\hat{x})\right)$$

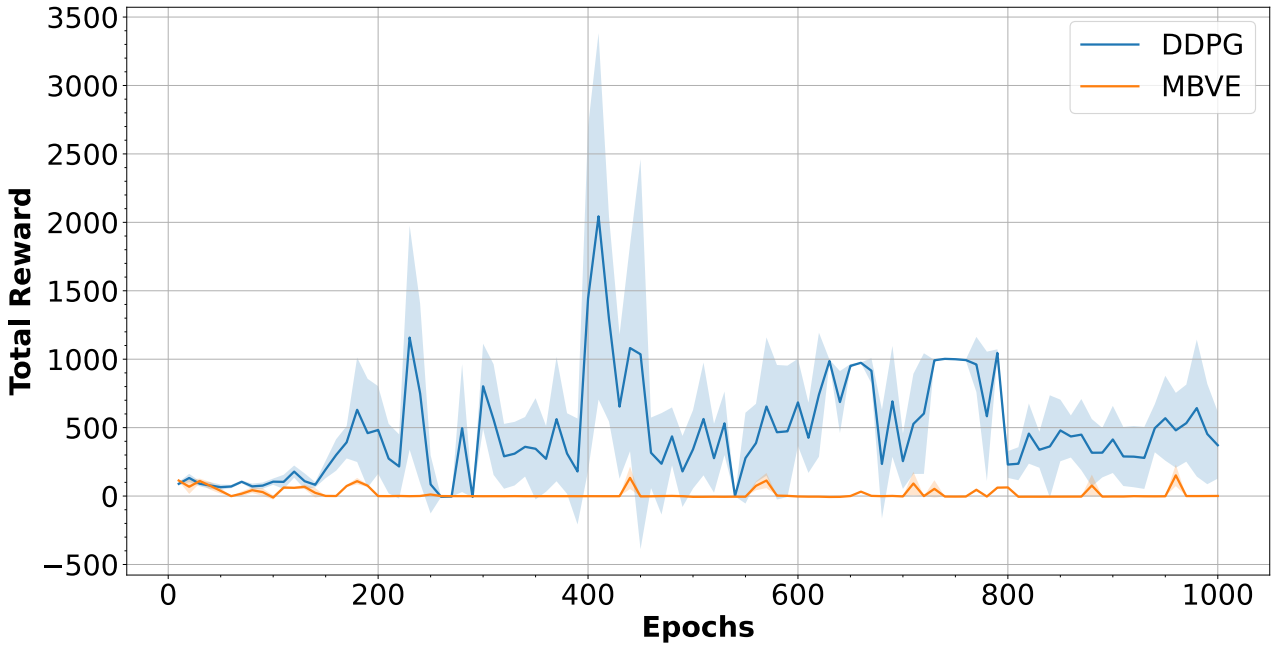
Where e_{bce} is the binary cross entropy. The limit cases when $x \to 0$ or $x \to 1$ are taken care of and the error is clamped at 100.

In Figure 6b, the cross entropy of the estimated done state with the true done state is represented. Again, it struggles to classify between terminal states and non-terminal states which leads to the failure of the algorithm. Due to lack of time, the cause for the poor performance of the model is not analyzed in depth but is most likely due to the way the model is trained since the model to approximate, in the case of the pendulum, is quite simple: only 4 state values to estimate, a simple reward function (= 1 if alive), and a simple definition of a terminal state (terminal if deviation angle from vertical is greater than a constant).

The objective of using a model-based method to increase the sample efficiency is thus compromised since the use of a model itself impairs the learning process. To conclude this part, the choice is made to use DDPG in the following since it is faster epoch-wise and does not rely on the quality of a model which could lead to bad performance if not accurate.

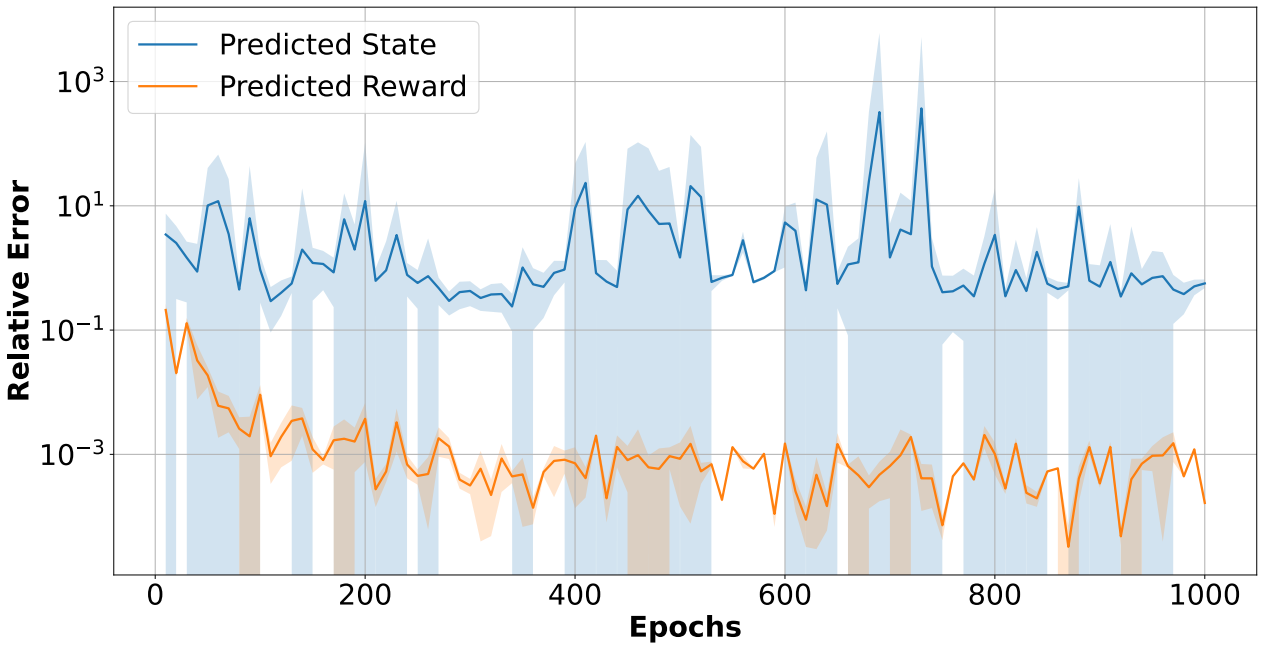


(a) Inverted Pendulum

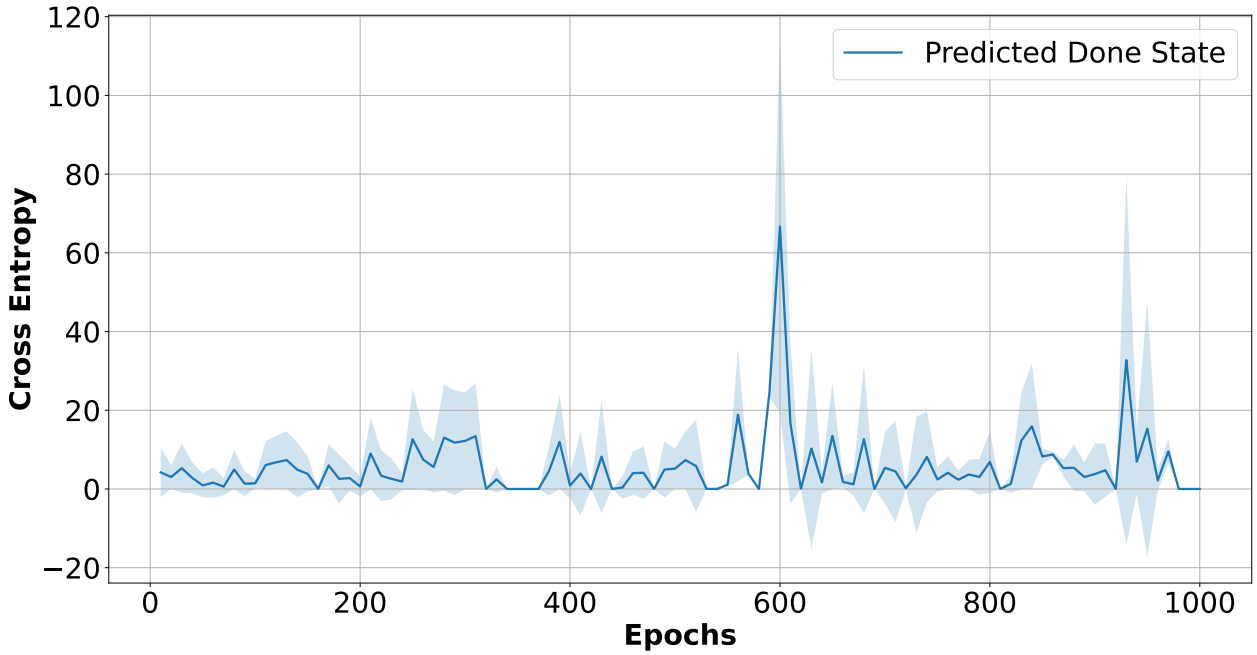


(b) Walker2D

Figure 5: Cumulative Reward



(a) State and Reward relative error



(b) Cross Entropy error of done state

Figure 6: Model Error

7 Network Architectures

The next step is now to choose an architecture for the neural networks that will be used for the actor and critic networks. Since the DDPG has been chosen previously as the training algorithm for the following steps, there are only these two networks.

The same structure will be applied for both actor and critic networks, this is an arbitrary choice and one could test multiple combinations of architectures for the actor and critic respectively. In this work, there are only 3 architectures that are investigated. These architectures will be tested on both test environments and the architectures will be nearly identical for the critic and actor on a single test. The criterion for the comparison will be the same as the previous section. The only difference is that in this case, the inference time of the actor will be considered as it will represent the computational cost to run the controller if one wanted to use it in a running environment.

The training parameters that will be used are the same as for the comparison of the algorithms specified in Table 2.

7.1 Neural Networks

The main ideas of the architecture are developed hereafter. There are 3 architectures:

- Multi-Layer Perceptron (MLP)
- Variant Multi-Layer Perceptron (MLP)
- Self-Attention

Firstly, the MLP architecture is one the most basic neural network architectures and is just composed of multiple sequential linear layers with activation functions in-between layers. The architecture implemented in this context is just a sequential combination of a linear layer and an activation function. The representation of this architecture can be found in Figure 16. The activation function is chosen to be the ReLU function which is a common choice for the activation function. It is defined as:

$$ReLU(x) \;=\; \max(x, 0)$$

Actually, this type of architecture was used by the top solution of the NeurIPS competition [36]. This network represents a very naive network without taking any other considerations into account.

Secondly, the next architecture is a variant of the first which differs from the addition of small features. A form of residual connection is added to the network by concatenating the input of a block with its output. This idea was used by [28] and has shown its interest in deep neural networks[40]. Then a normalization layer is added at the entry of each residual block. More specifically, batch normalization is used as described here [41] and it is common practice to use this kind of layer in neural networks. A dropout layer [42] is also added before

every activation function because it is also good practice and helps to increase the robustness of the network. It might however increase the training time needed. A representation of the architecture of this network is depicted in Figure 17 alongside with the different blocks involved.

The last architecture that will be tested is inspired by the recent success of transformers and more particularly attention layers [43]. Actually, the idea of incorporating transformer architecture inside RL framework has already been studied [44, 45, 46]. In this work, only self-attention layers will be used. This architecture consists of multiple blocks assembled sequentially. Each block consists of a multi-head self-attention layer followed by a linear layer and a residual connection is added to the whole block. Dropout and layer normalization are also incorporated. This implementation is inspired by an architecture presented in the context of the INFO8010 course given by Professor Gilles Louppe at the ULiège University [47]. The representation of the self-attention architecture is found in the Appendix in Figure 18. Lastly, all actor networks have a scaled version of a hyperbolic tangent activation function in order to project the action in the range of the different environments.

7.2 Simulation Results

The different results on both environments are shown in Figure 7. The first thing to notice is that for the pendulum environment, all networks find a policy that maximizes the reward as can be seen in Figure 7a. However, the MLP seems not to be very stable in its training in the sense that its performance sometimes drops greatly. A similar observation can be made for the self-attention network even if it is a bit more stable than the MLP. On the other hand, the variant MLP seems to be more stable than the two other networks in this case.

Then, when looking at the performances of the networks on the walker environment in Figure 7b, it is quite obvious that the variant MLP has the best performances among all tested networks. The MLP never performs greatly, it seems to never learn something useful. The self-attention network seems to have learned something at the start but then its performance decreases without improving again. There is a small plateau between the 300^{th} and 350^{th} epochs. This plateau is a case where the agent has learned not to fall and so the agent stabilizes itself but does not move forward therefore its cumulative reward is bounded by 1000. The variant MLP has the best performances in general even if the standard deviation is also higher. However, the agent has learned not only to stand without falling but at some point, it can move a bit forward.

The next thing to look at is the training time needed for the different architectures. These training times are shown in Table 4. From this table, it can be seen that the MLP is the fastest to train followed by the variant MLP and the most time-consuming is the self-attention network. Then, the same order is observed when comparing the inference time for the 3 networks.

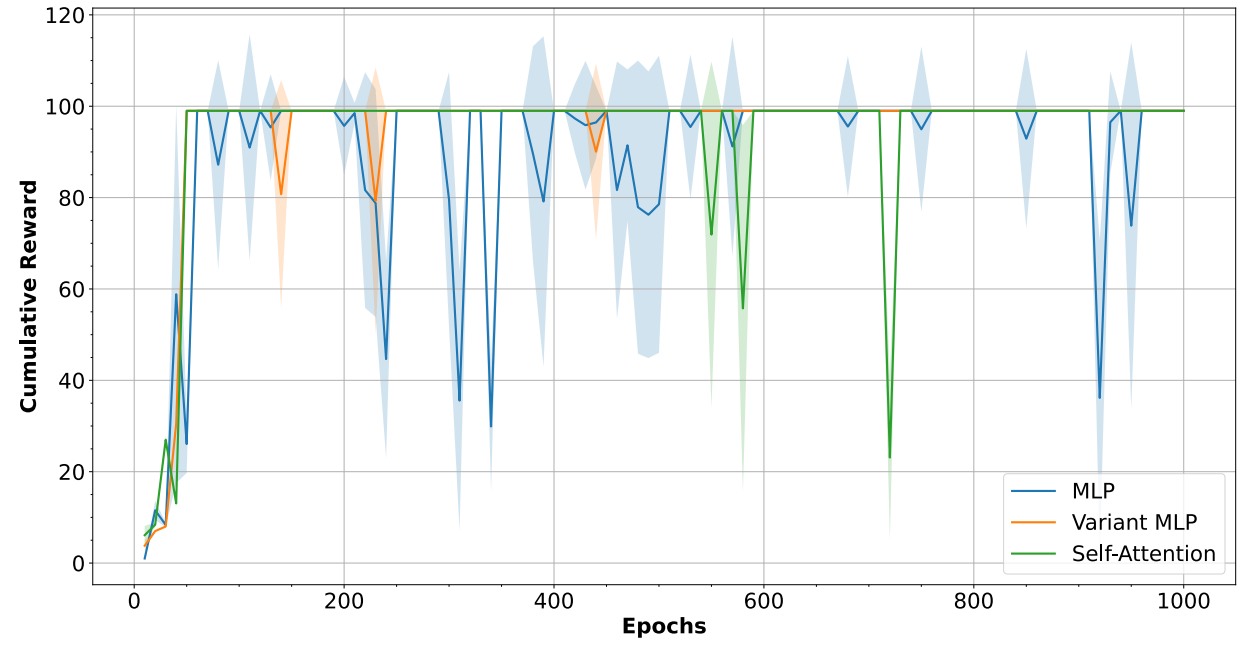
	MLP	Variant MLP	Self-Attention
Pendulum	01:40:00	02:20:00	04:20:00
Walker	02:15:00	03:30:00	05:10:00

Table 4: Training Times

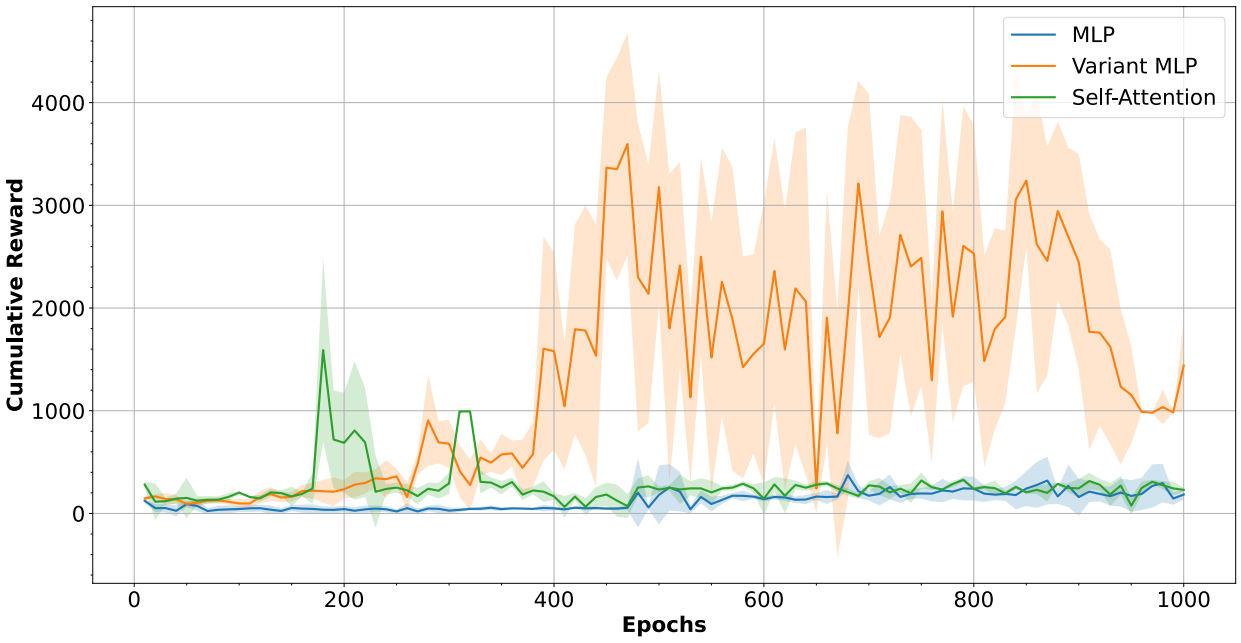
7.3 Discussion

From the results explained previously, the choice of network is quite clear. Indeed, the variant MLP will be used in the next steps since it's the best compromise between performances and computational cost. It is the best network in terms of performance. The fact that the variant MLP is better than the MLP could be explained by the fact that it incorporates common good practice in terms of architecture and thus is better than the simple and naive version.

On the other hand, the reason for the self-attention to perform poorly compared to the variant MLP cannot be explained straightforwardly. It might be that such an architecture isn't appropriate in this context for the different functions it needs to approximate.



(a) Inverted Pendulum



(b) Walker2D

Figure 7: Cumulative Reward

8 Frame Skipping

The last experiment to conduct on the 2 test environments is the effect of frame skipping. For this part, the algorithm is fixed to be DDPG and the architecture used is the variant MLP. Except for the frame skipping all the other parameters are identical to the ones specified in Table 2.

8.1 Concept

First of all, the idea of frame skipping is to make an agent apply the same action over a specified amount of time steps. For example, denoting by fs the numbers of skipped frames, an action a is taken by the policy π every fs time steps in the environment. It is such that:

$$a_t = \begin{cases} \pi(s_t) & \text{if } \ t \bmod fs = 0 \\ a_{t-[t \bmod fs]} & \text{Otherwise} \end{cases}$$

Where a_t represents the action at time step t, s_t is the state at time step t and mod is the modulo operator. In fact, this technique has shown its effectiveness for performing in Atari games [48] and even in other environments [49]. The underlying idea is to force the agent to act the same for a longer period of time. In a way, the actions that the agent takes have a higher impact when it comes to the interaction with the environment.

Alternatively, it can be interpreted as if the number of possible sequences of action is reduced. For a finite number of time steps, the total number of actions to be taken is lower when performing frame skipping. Hence, it may be easier to find a sequence of actions leading to a good reward.

8.2 Results and Discussion

The performance for comparing the impact of frame skipping can be found in Figure 8. An important thing to notice is that the scale of the cumulative reward is different. Indeed, for the test environments, the reward obtained time step by time step is not scaled according to the frame skipping. So if there is a frame skipping of 10 frames, the agent will collect a reward 10 times less than in an environment without frame skipping for the same total duration time of an episode. In order to provide comparable curves, the rewards are scaled accordingly to the frame-skipping factor. In that way the cumulative rewards for both approaches are comparable.

Starting by looking at Figure 8a, it can be seen that frame skipping helps to learn faster and more robustly. Indeed, once the agent has learned to perform optimally, the performances are quite stable. On the other hand, the agent that didn't use frame skipping took more epochs to achieve the best performances. However, these performances are not stable as can be seen in the figure. There are huge drops in performance at some epochs.

For the walker environment, the performances are shown in Figure 8b. In this environment, the benefit of using frame skipping is even clearer, without frame skipping it seems

that the agent never learns to perform correctly in the environment since the performances are very low. When using frame skipping, the agent performs way better even though the performances are fluctuating.

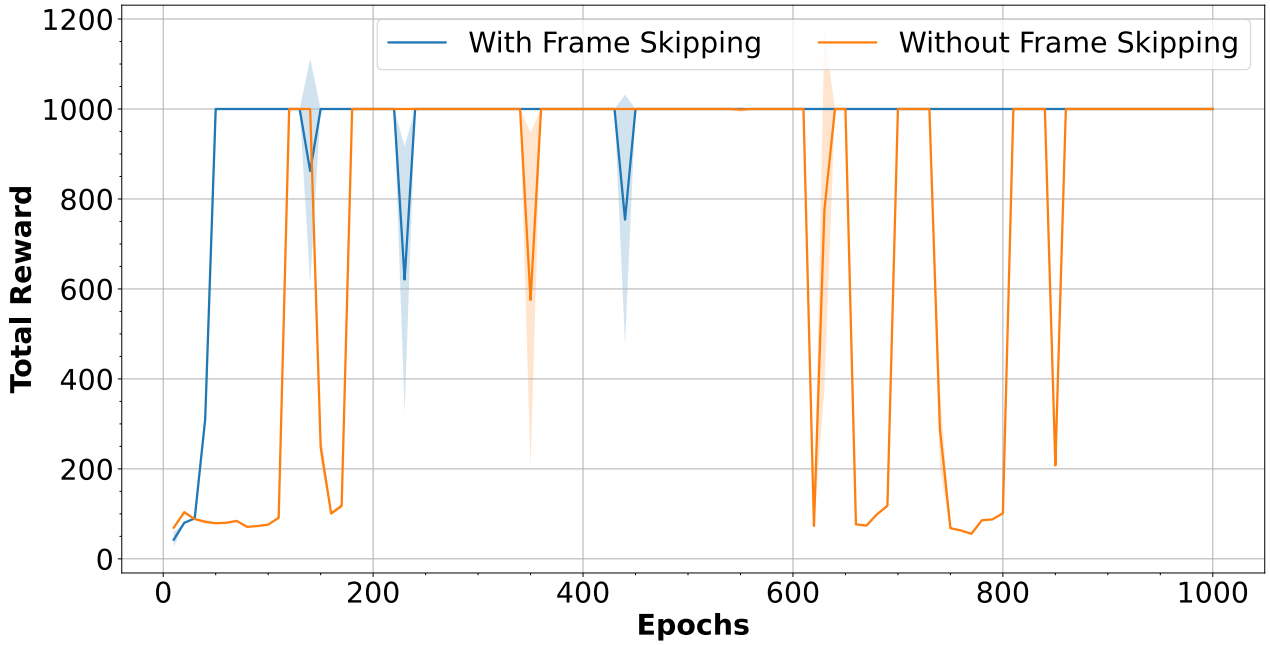
These results confirm the intuition that frame skipping helps in the process of learning. As discussed earlier, the number of sequences of actions is reduced when performing frame skipping and thus it might be easier to find a good sequence of actions.

Concerning the training times, they are quite similar as shown in Table 5. Using a frame skipping of 10 takes a little longer than without any frame skipping. However, the difference in performances between the two approaches clearly makes us choose to go with the approach of using frame skipping. It is expected that using frame skipping may extend the training time since the simulation time steps are also greater so the environment simulates for a longer time at each time step when using frame skipping.

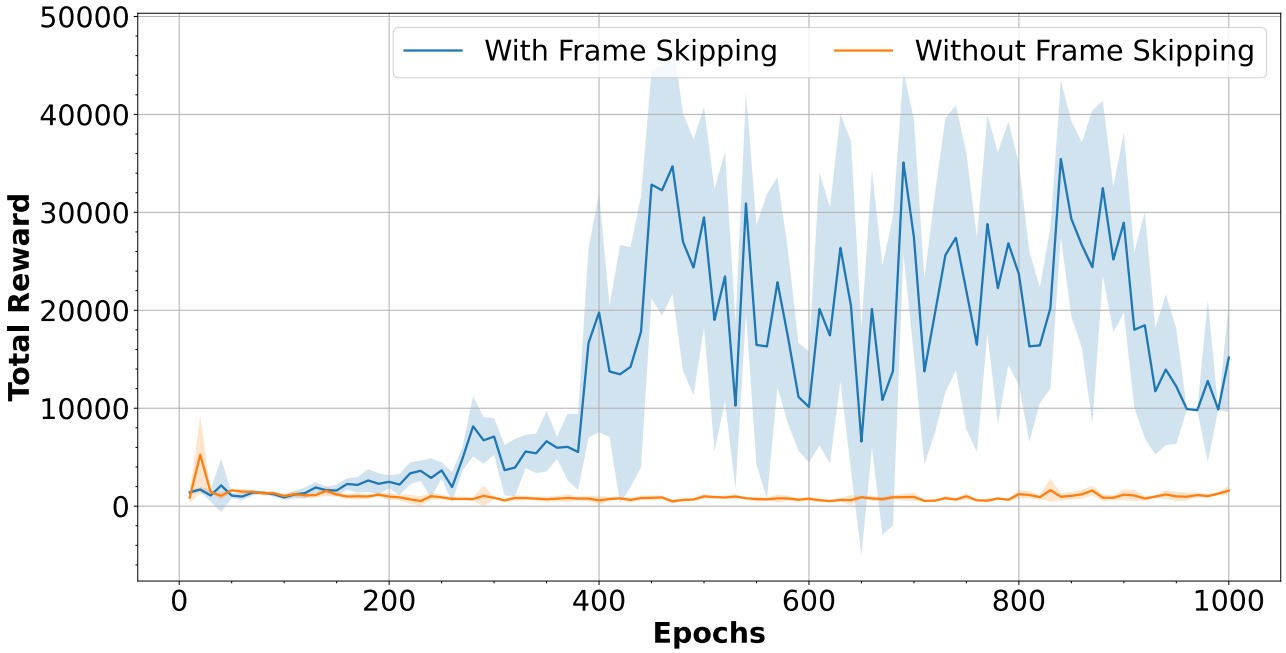
In conclusion, considering the initial environment in this work, frame skipping emerges as a favorable choice due to its greater performance in test environments.

	Frame Skipping	**No Frame Skipping**
Pendulum	02:20:00	02:05:17
Walker	03:30:00	03:11:07

Table 5: Training Times



(a) Inverted Pendulum



(b) Walker2D

Figure 8: Cumulative Reward

Part III
Training the Musculo-skeletal Agent

9 Training Procedure

In the previous part, a short analysis was done to choose which algorithm and networks were to be used for training the initial agent in the physiologically accurate environment.

The algorithm that will be used for training is DDPG as it performed better than MBVE and does not rely on the quality of a learned model of the environment. Next, the architecture chosen will be the variant MLP as it leads to the best performances on the test environments with reasonable training times. The exact parameter for the structure of the network can be found in Table 6 following the architecture in Figure 17. Lastly, use of frame skipping within learning will be used as it was demonstrated to be very effective in the 2 test environments. So this was a summary of the choices made so far for the training of the musculo-skeletal agent.

This part will only focus on the musculo-skeletal agent and the final adjustments to the method developed throughout this thesis. As a first adjustment, the initial reward function of the environment has been modified during this work and a small discussion about the shape of the reward will also be present in the following. Then, the idea is also to assess the impact of pre-training the agent to imitate another controller. Thus, a comparison between learning from scratch and a pre-trained agent will be conducted.

Finally, the results of the training process will be analyzed in terms of performances over time as well as the quality of the gait and motion developed by the agent.

9.1 Reward Shaping

Shaping a reward in a RL framework is something really important and might be difficult [50]. In fact, an ill-shaped reward can lead to undesired behavior in a given environment. The policy could act optimally in concordance with the reward function but still not behave as it is intended to [51]. A badly shaped reward could also lead a policy to be stuck in a local minimum.

For this problem, the original reward function is the one specified in section 2. A few attempts of modifications were performed to make the agent learn faster. A first attempt was to add a bonus for swinging the legs. Denoting $\dot{\theta}_r$ the angular velocity of the right hip joint and $\dot{\theta}_l$ the angular velocity of the left hip joint, the term in the reward r_{swing} was as follows:

$$r_{swing} = w_s \cdot \|\dot{\theta}_r - \dot{\theta}_l\|$$

The idea was to encourage the agent to swing its legs in opposite directions and so to make it walk. However, observing some intermediate results, showed that the agent had learned

to throw one of its legs into the air which is not a desired behavior. This idea was thus abandoned.

Then, another idea was to penalize the agent if the height of its pelvis was too low. The threshold is chosen to be 0.8m since 0.6m corresponds to the threshold for a terminal state as described previously. Basically, the idea was to encourage the agent to keep its pelvis above the terminal threshold. However, it was observed that sometimes the agent chose to terminate the episode as fast as it could when its pelvis was below 0.8m. In fact, this was because the agent would have less penalty by terminating the episode fast than staying with a pelvis height between 0.6 and 0.8m.

Hence, this penalty was a bit modified in order to only give a large penalty when the agent is in a terminal state as described in section 2. So that the penalty is equal to -100 when the pelvis height is lower than 0.6m. Finally, the form of the reward function consists of multiple terms that are detailed in section 2.4.

9.2 Pre-training

Pre-training in Reinforcement Learning (RL) refers to the process of training an agent on a related task or a large dataset before fine-tuning it on the specific target task. This approach has proven to be effective for some complex tasks [52].

In particular, for this project, a tool available is a controller adapted from this paper [9]. The controller is neuro-inspired but the version provided only works for particular initial conditions. It is however able to provide a dataset of a walking agent.

The inspiration for using such an approach comes from the top solutions of the NeurIPS competition [36, 28] where they used curriculum learning in the training procedure. Curriculum learning consists of changing the objective progressively in an environment to learn intermediate behavior and accelerate the learning of an agent for a particular task. In their work, they first learn to move forward as fast as possible in the 2d simplified environment used in this thesis. Then, the agent learns to follow a target velocity in a forward direction. Finally, they used this agent to generalize in 3 dimensions.

In this work, the agent will be pre-trained with data collected from the neuro-inspired controller cited above. Then, the agent will continue to train on the environment to learn to move forward. The interest of pre-training, in this case, is that the learned policy is trying to imitate the neuro-inspired controller. Then, when training with DDPG, the policy will generate transitions that are better than completely random actions. Hence, it is expected that the use of imitation learning before starting the real training process accelerates the training.

9.3 Initial State

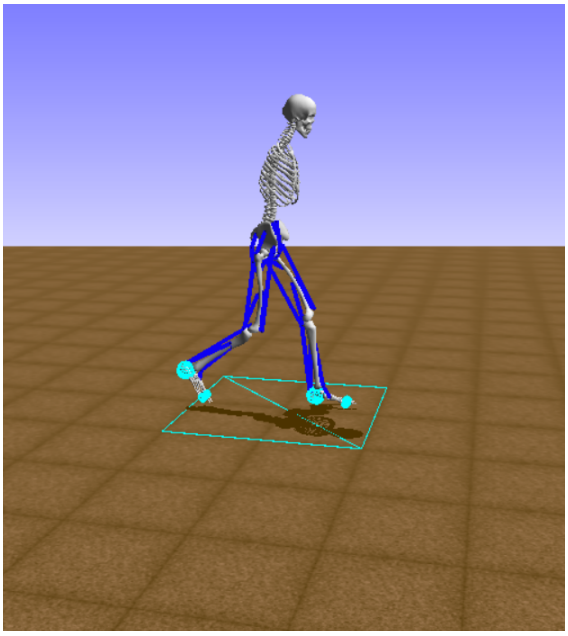
A last improvement that is performed related to the original environment concerns the initial states. Initially, the agent starts by being straight up with its legs and body aligned (Figure 9c). However, when performing imitation learning, the initial state needed to be different because the neuro-inspired controller needs to start in a particular state (Figure 9a).

The following idea was to design 3 main initial states. Basically, it consists of:

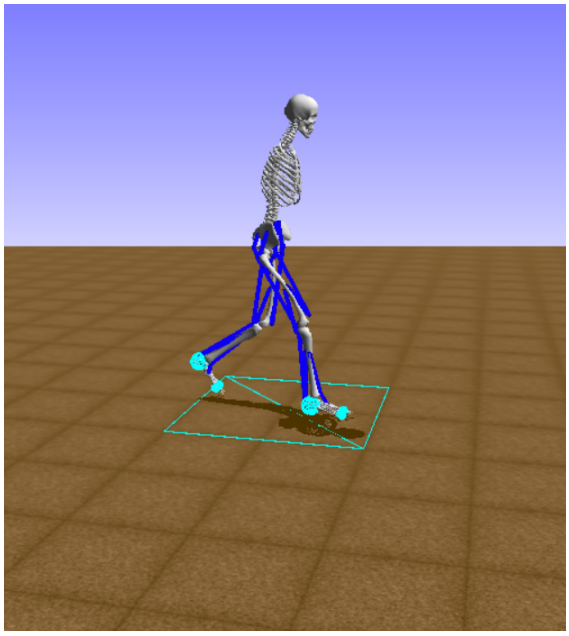
1. The upright position (Figure 9c)
2. A position with the right leg in front and the left leg in the back (Figure 9b)
3. A position with the left leg in front and the right leg in the back (Figure 9a)

Then some noise is added to the position in order to randomize as much as possible the initial states. The noise is a Gaussian noise around the position with a very small standard deviation. This set of initial states is used in order to provide a lot of different starting points in an episode. Hence, this would allow us to explore a large variety of different states.

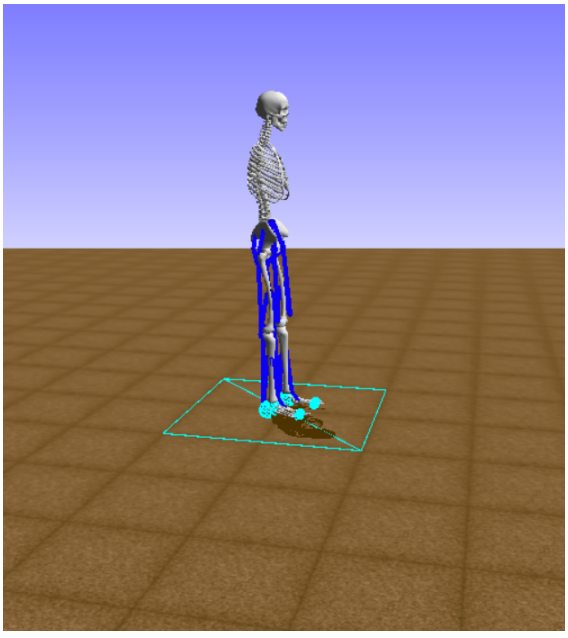
In addition, thanks to the state's locality as explained in section 2.2, if the agent is able to perform optimally initially then it should perform the same way further from the initial point. Hence, this justifies this idea of taking these multiple initial positions such that more positions are covered at the initial state of an episode.



(a) Left Leg in front



(b) Right Leg in front



(c) Upright

Figure 9: Initial positions of agent

10 Analysis of final controller

After an extensive training process, the final controller is a policy that has undergone rigorous training for a total of 500 epochs. During each epoch, the policy was updated 1024 times with a batch of size 4000 for each update. The learning rate was equal to $1e-4$ for both the actor and the critic network. Frame skipping is also applied for the training process, the initial time step is $dt = 0.01s$ and it is dilated by a factor of 10 so that the dt used for training is $dt = 0.1s$.

The training procedure, spanning numerous iterations, consumed approximately 33 hours to complete, showing the great number of computational resources that are needed for such a task. In the following, an analysis of the performances of the final controller as well as the gait exhibited by the agent with the learned policy is conducted.

10.1 Performances

In terms of performances, it will be decomposed reporting to the 3 initial states described in section 9.3. For each initial state, 20 episodes are played in order to record the average performances of the agent starting from each different state. Additionally, it has been observed that evaluating the agent trained with frame skipping ($dt = 0.1$) with the original time step $dt = 0.01$ leads to better performances. In fact, evaluating with $dt = 0.1s$, the agent falls from time to time. On the contrary, evaluating with $dt = 0.01$, the agent falls much less and is also able to move forward.

This result is interesting in the sense that it shows the agent can interpolate the action to take. It means that if the control frequency increases, the agent is still able to work. In particular, in this case, it worked even better. So, a time step of $dt = 0.01$ will be used in order to evaluate and analyze the agent's performance and gait.

The results are shown for each of the initial states including variation around these positions in Figure 10. It can be seen that except for some episodes, the performances are quite high for the 3 starting positions. In fact, if the agent does not make any steps, the total reward would be 100. In addition, for all initial positions, the majority of episodes are above this value. Hence, it shows that the agent is able to move forward. It is indeed confirmed by visualizing multiple episodes in the environment. By looking at Figure 11, it can be seen that the velocity of the agent varies but does not get really high. Indeed, it barely goes up to $2.2\mathrm{m\,s^{-1}}$ and is most of the time below $1.5\mathrm{m\,s^{-1}}$.

Next, an analysis of the gait is performed. Indeed, since previous results show that the agent is able to move forward then a gait analysis makes sense.

10.2 Gait analysis

This section will focus on the behavior of the agent, especially the gait of this agent. For the analysis, the agent starts in the position with the left leg in front (Figure 9a). As part

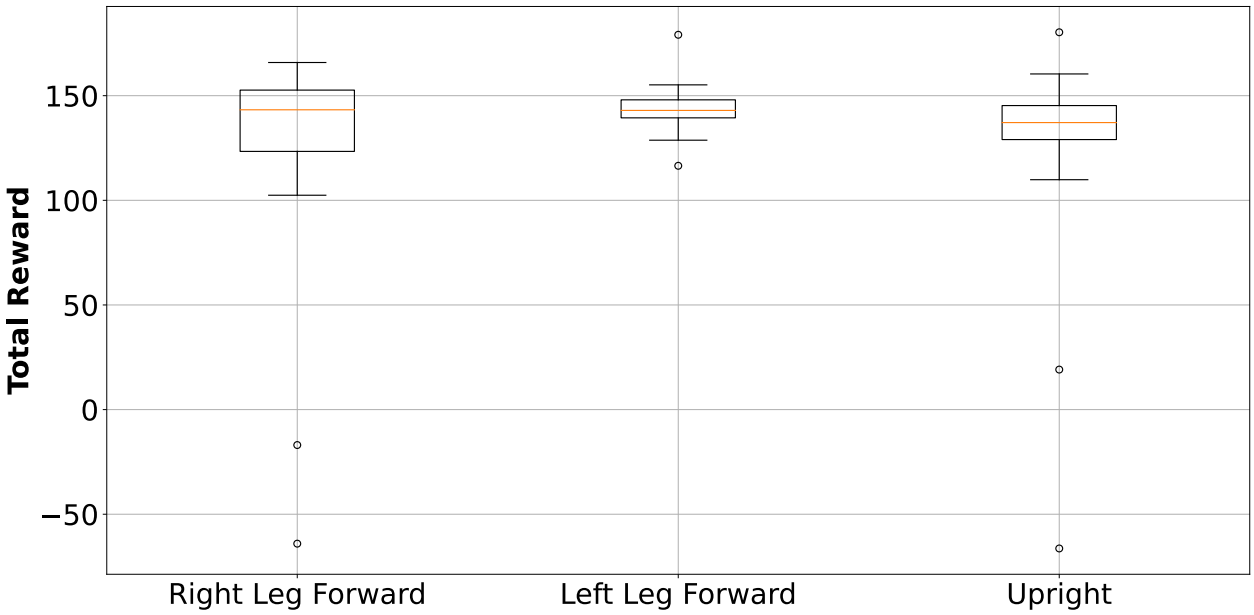


Figure 10: Total Reward of the agent in comparison to the 3 initial poses

of this analysis, all excitation signals generated by the agent are shown in Figure 15. Each figure represents the excitation level of a muscle on both legs. Also, the upward ground reaction force is shown for both feet in Figure 12. Similarly, the angles and angular velocity of each joint are shown in Figure 13 and 14 respectively. As for the excitation levels of the muscles, the different quantities are shown for the right and the left leg. This choice of representation is to enhance the visibility of symmetric behavior in the agent. Lastly, the global motion of the agent is summarized by the motion of the pelvis, Figure 11 shows the height of the pelvis as well as its velocity forward and upward.

Starting by looking at the evolution of the state of the pelvis through time (Figure 11), it can be seen that a certain pattern is repeating itself. There are small variations but the agent moves forward with a kind of cycle. However, the agent does not behave symmetrically, indeed by looking at the joint values (Figure 13), it can be seen clearly that the curves for the left leg and the right leg are different. Similarly, the same kind of observation can be made by looking at the angular velocities in Figure 14.

The source of this non-symmetric behavior can be seen with the excitation levels (Figure 15) provided by the controller. Indeed, for some muscles like the BFSH, there is not any symmetry between the excitation levels of the two legs. The excitation for the right leg is often near 0 while it is the opposite for the left leg. However, the agent seems to have learned complementary values for the excitation of the muscle on both legs. When the value is low for one leg it is higher for the other leg. So, the agent may have learned a dependency

between the two muscles. The same relations can be observed for different groups of muscles for example the HAM in Figure 15f.

Some muscles seem to exhibit some symmetry nonetheless, for example, the GAS in Figure 15b, the excitation signal is high on one leg then the other is high in a symmetric way. This corroborates the previous observation with the complementary excitations.

All muscles won't be analyzed in detail because overall the figures, the reason for the asymmetry in the behavior of the agent is due to some muscles. The agent does not provide symmetry in its excitation signals pattern for some muscles. Hence, the gait is not symmetric.

Lastly, this analysis showed that the agent is not symmetric in his gait. This does not mean that it performs badly in the environment but from a physiological point of view, this type of gait is closer to a disturbed gait than a typical human gait.

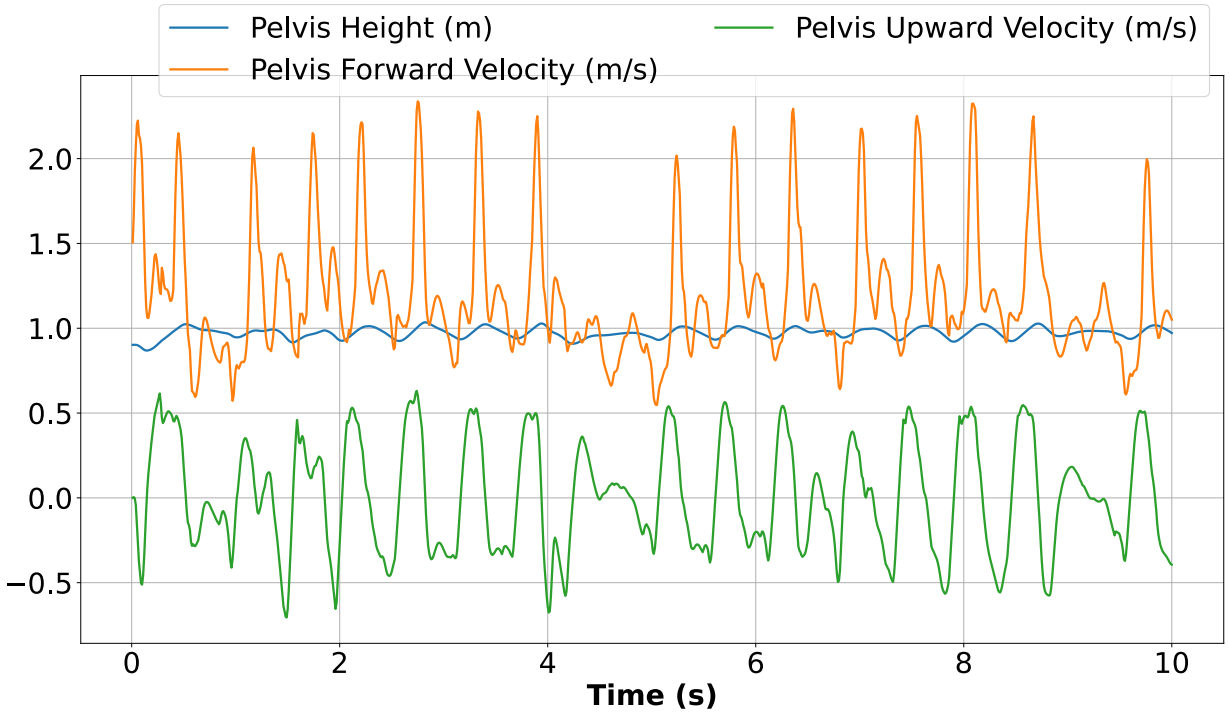


Figure 11: Pelvis State including forward velocity, upward velocity, and its height

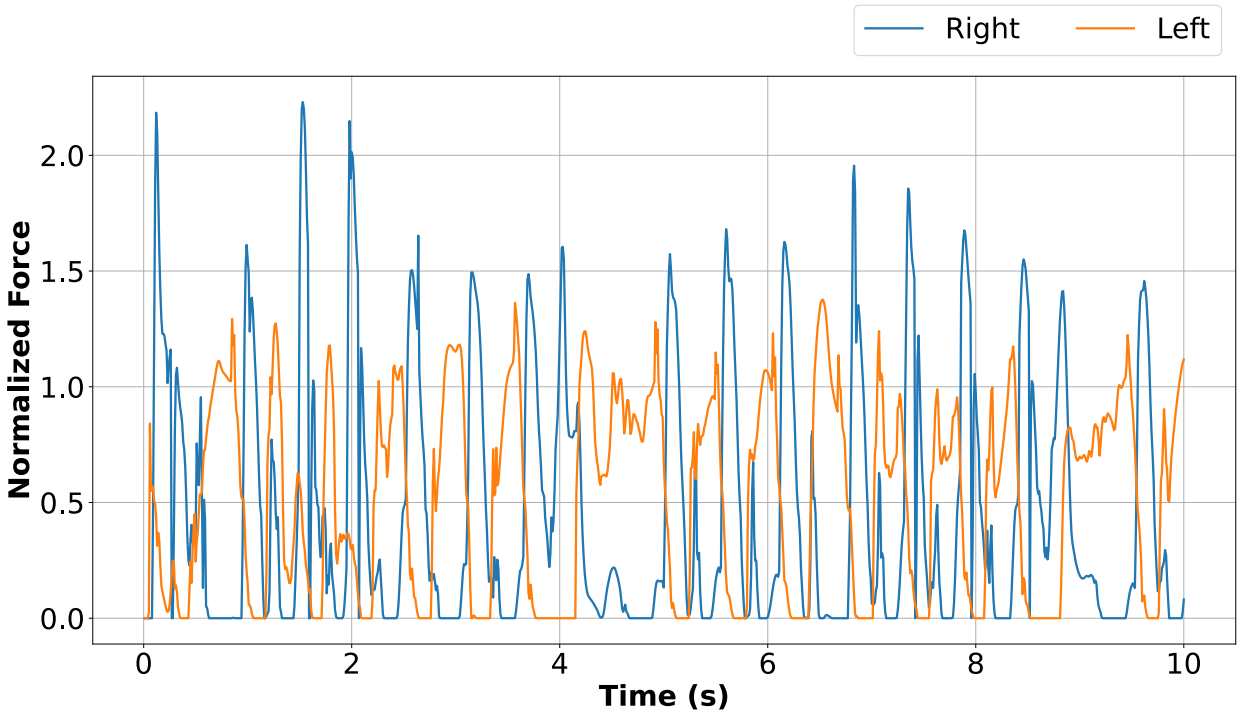
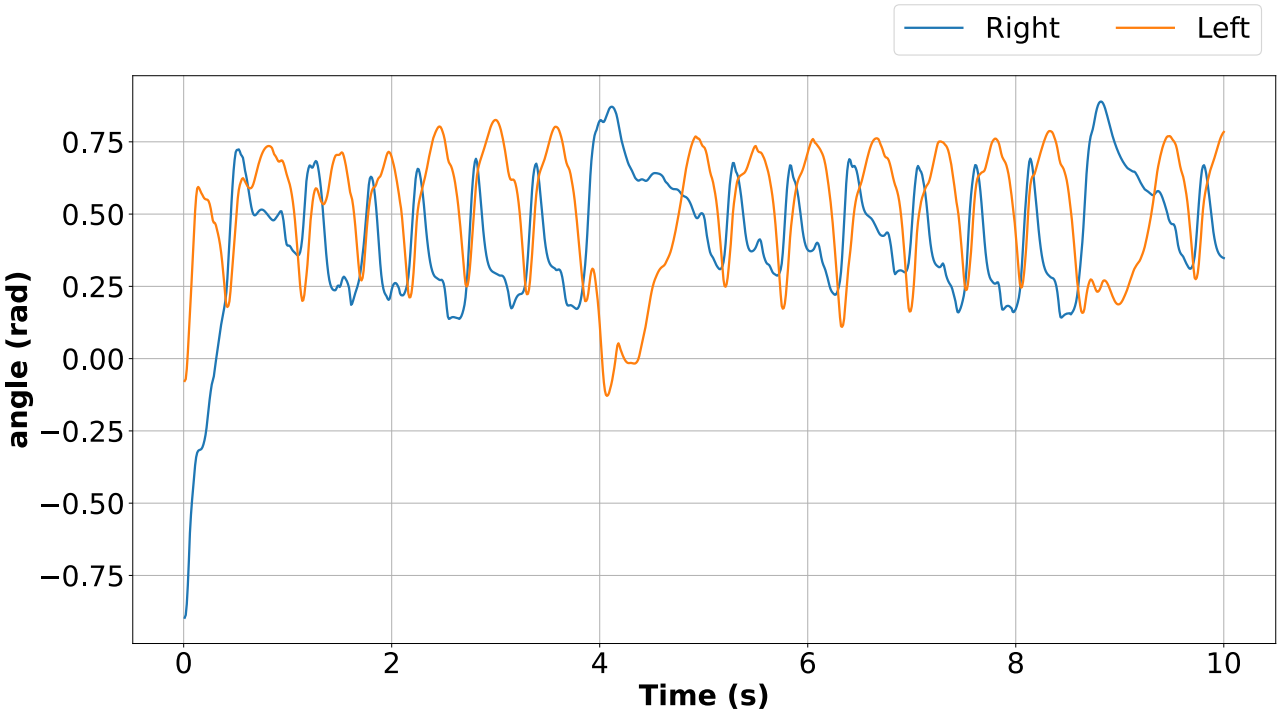
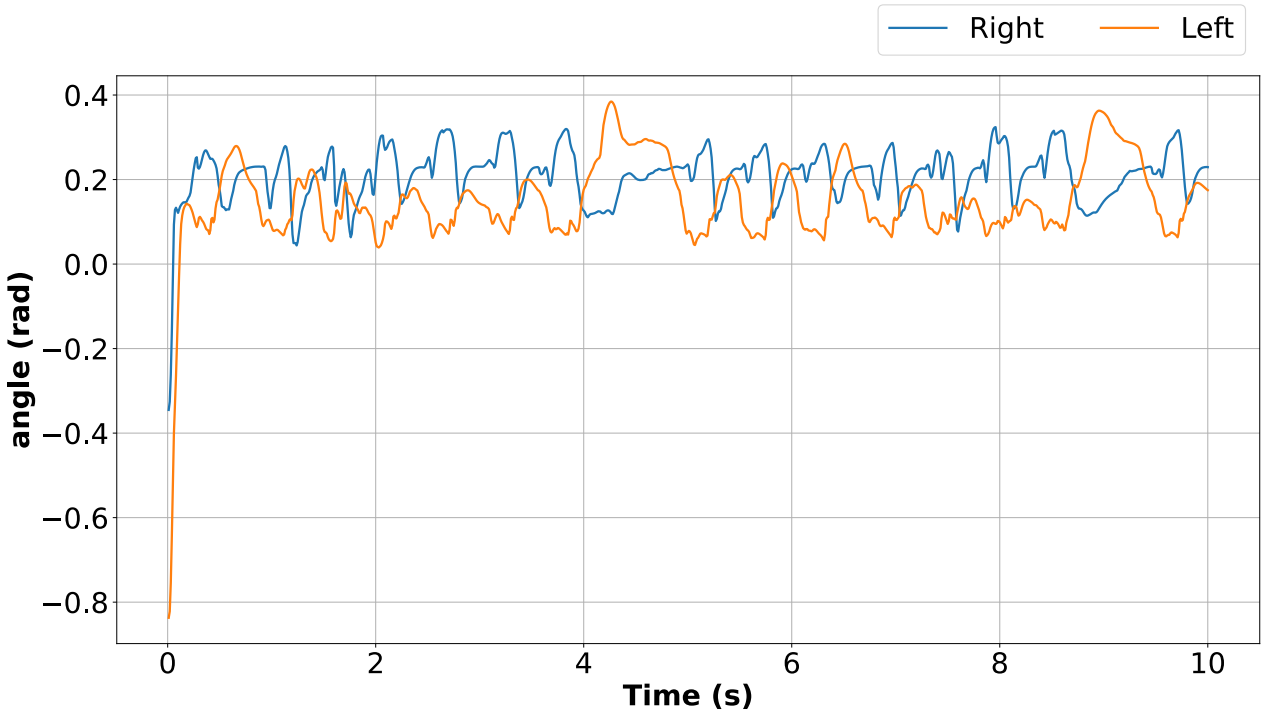


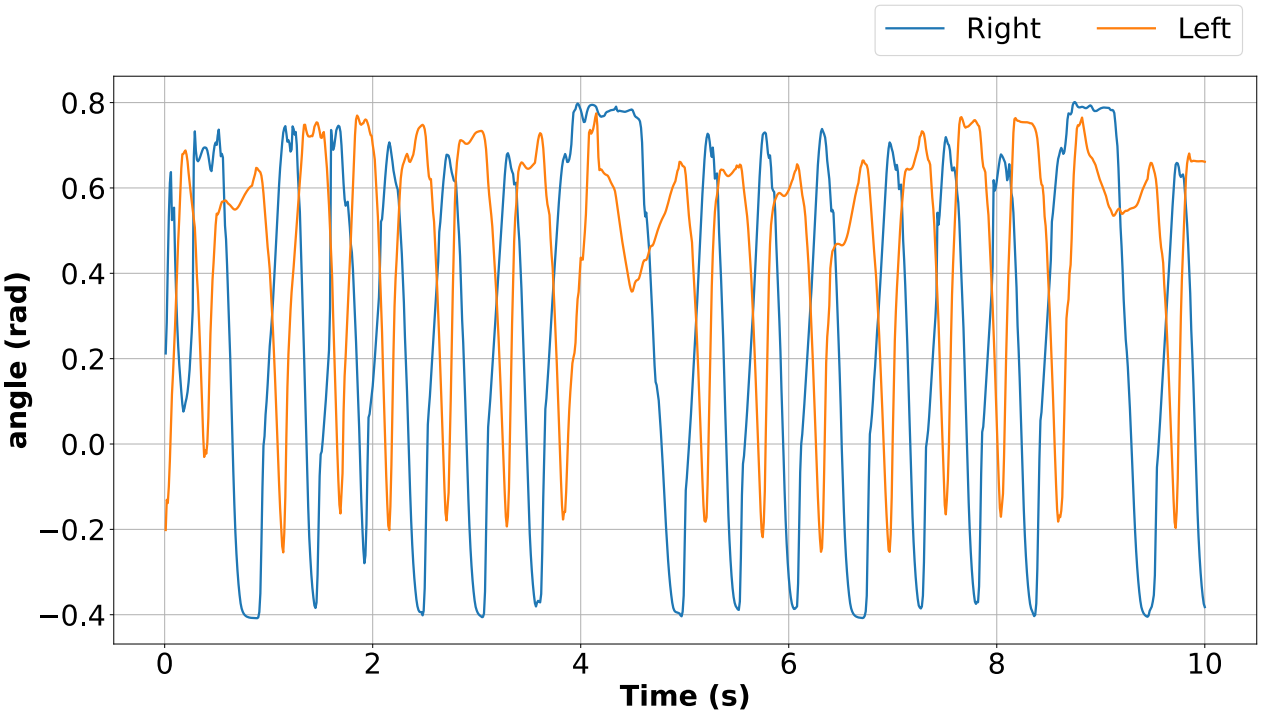
Figure 12: Ground reaction forces pointing upward for both feet



(a) Hip Joint Angles

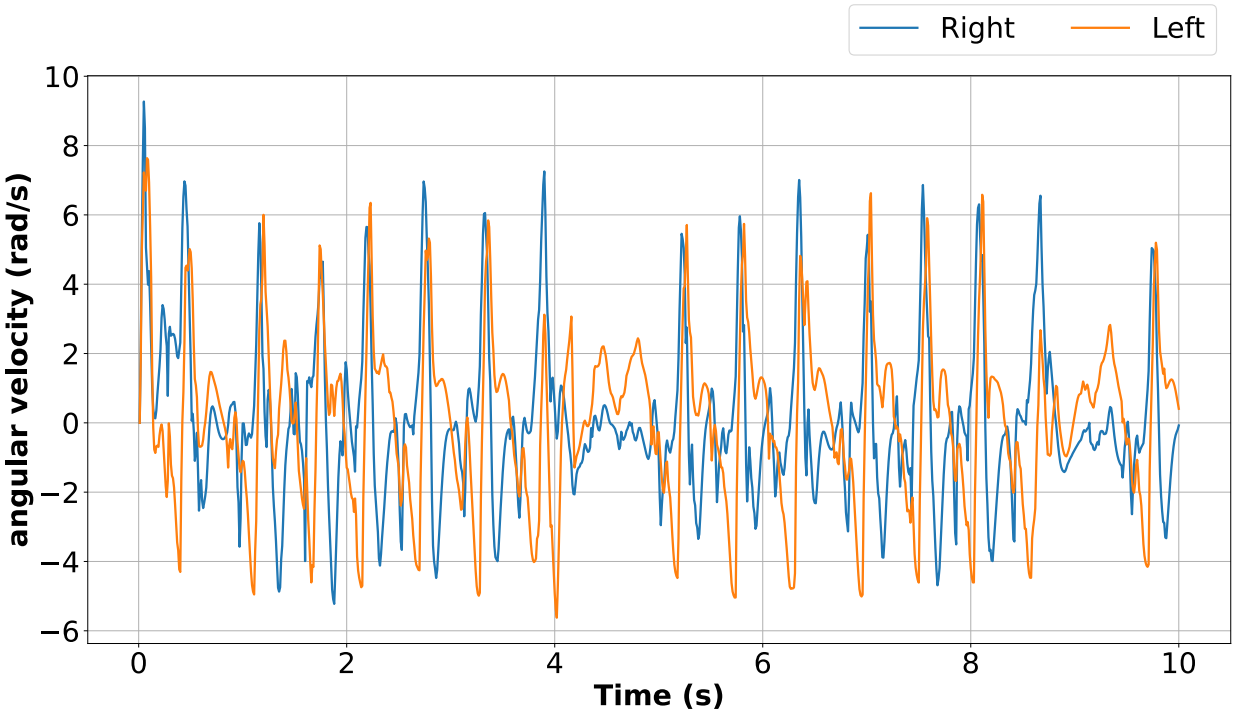


(b) Knee Joint Angles

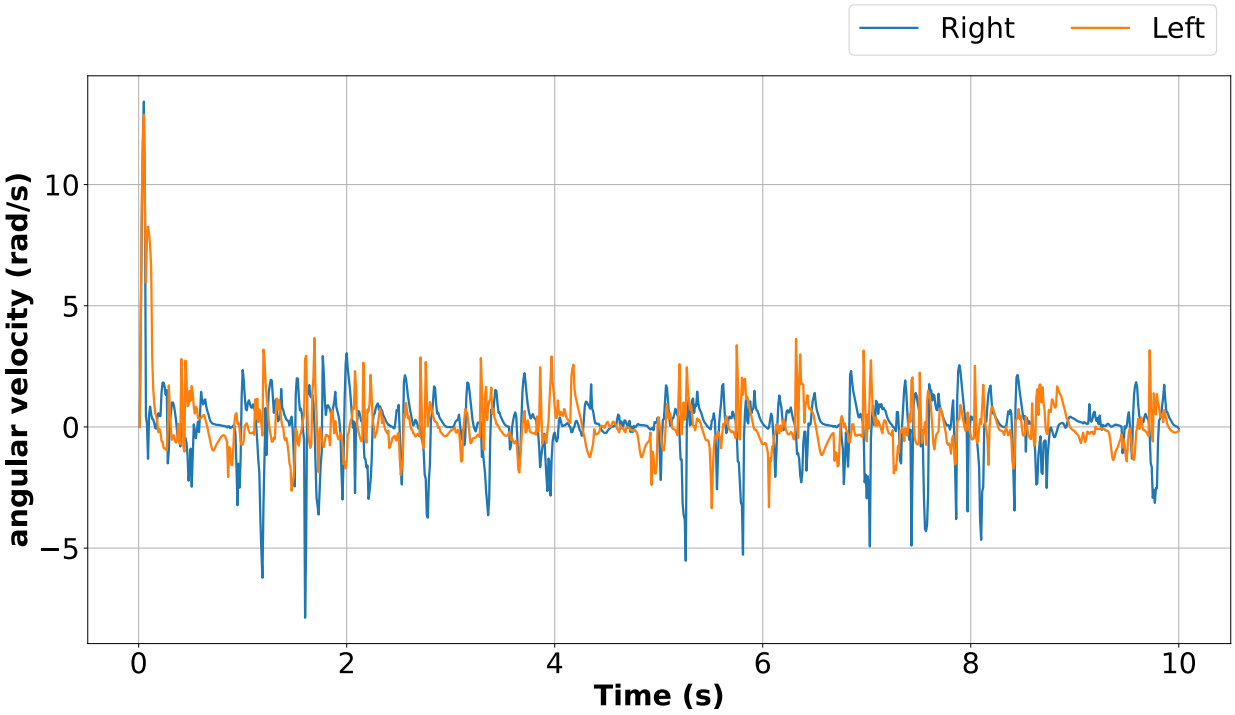


(c) Ankle Joint Angles

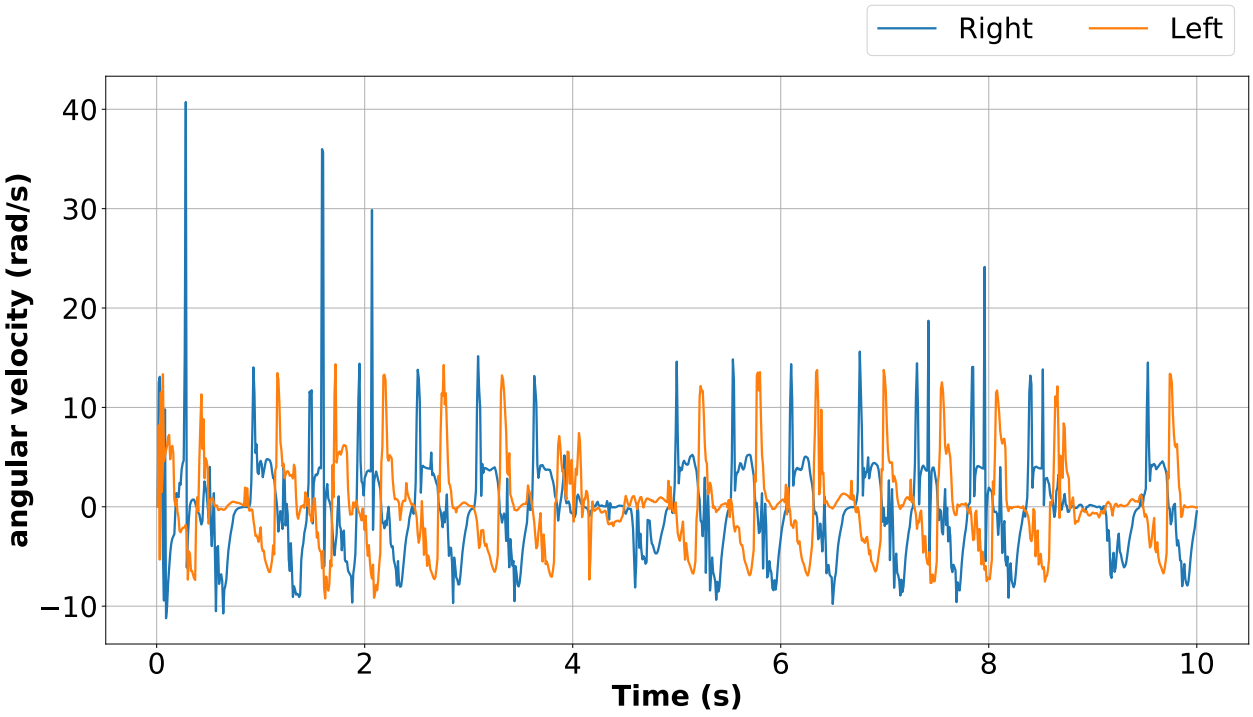
Figure 13: Joint Angle Values for both leg.



(a) Hip Joint Angular Velocity

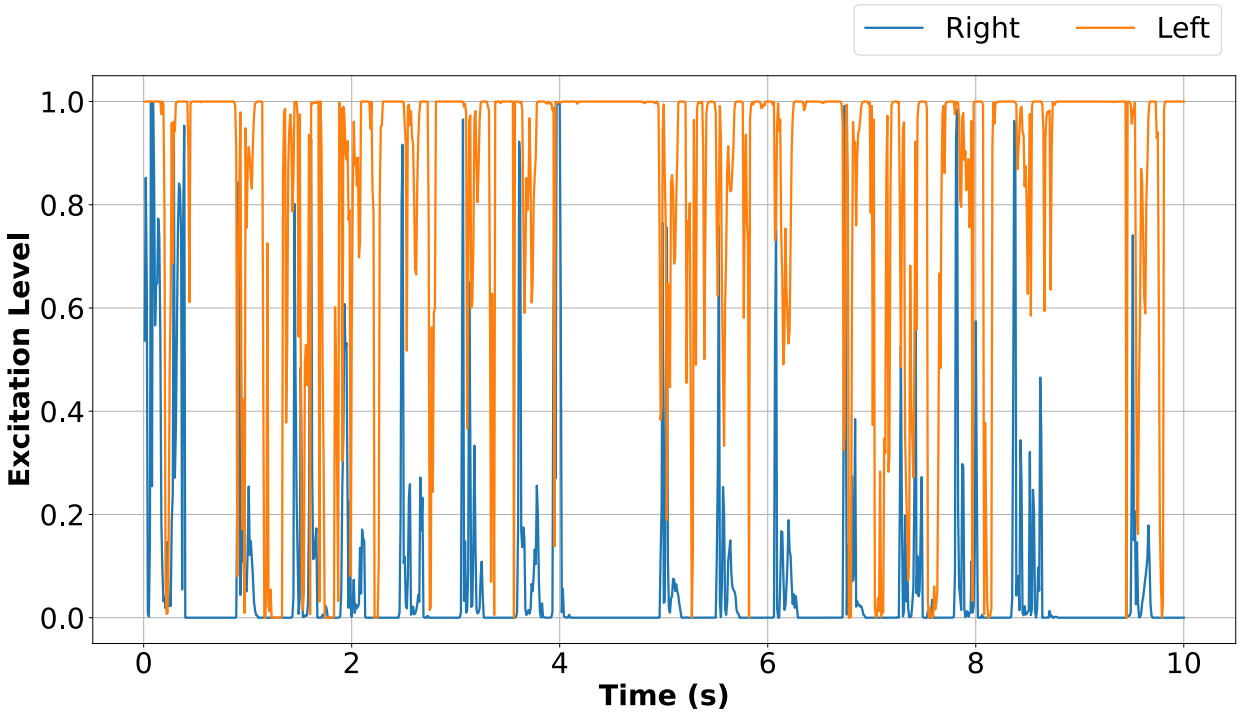


(b) Knee Joint Angular Velocity

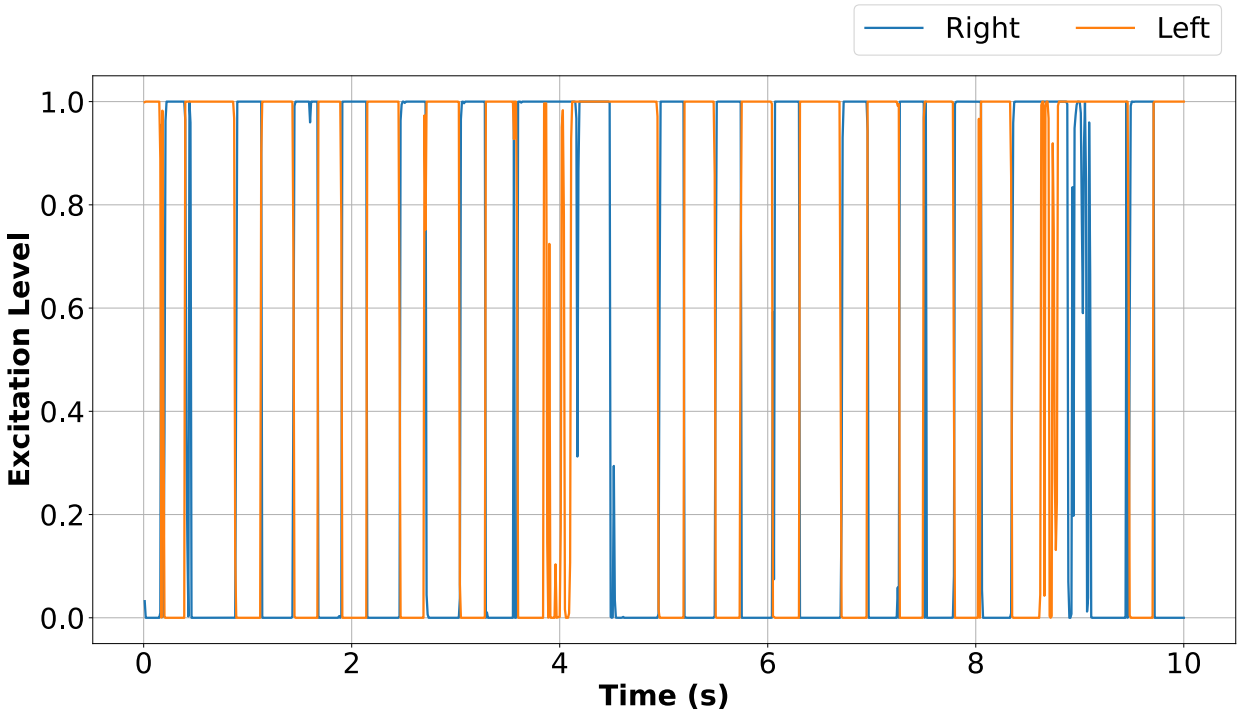


(c) Ankle Joint Angular Velocity

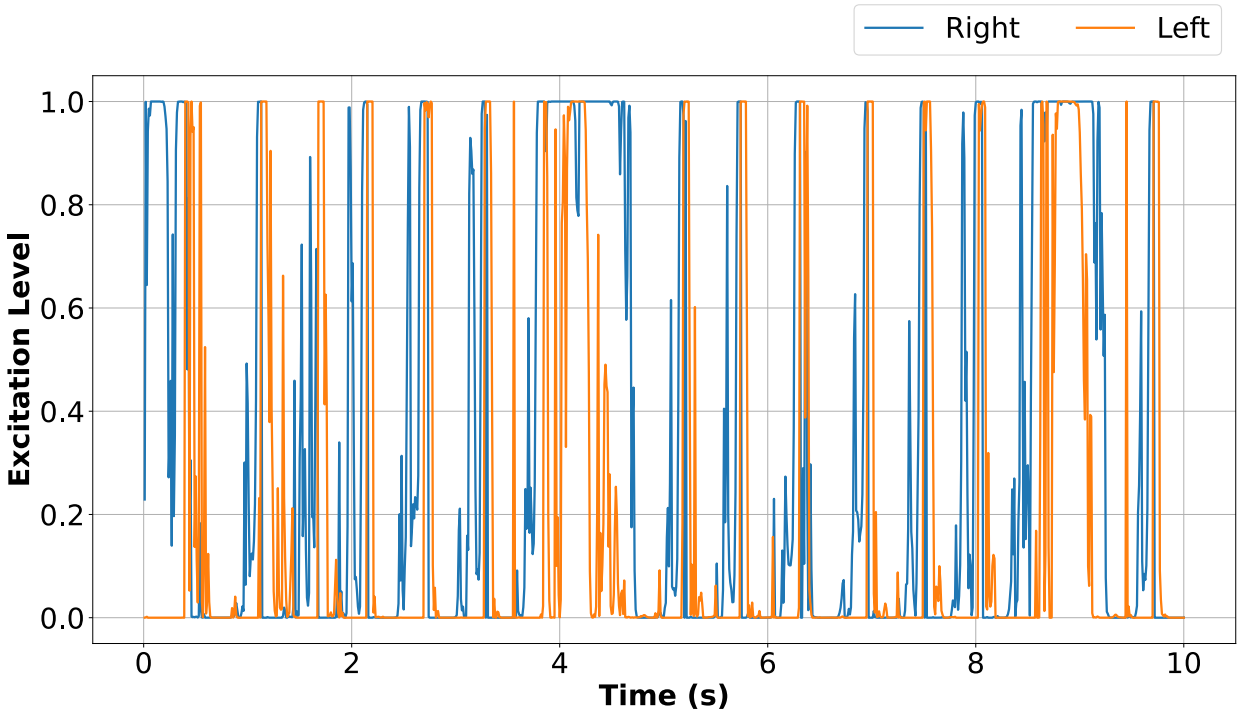
Figure 14: Joint Angular Velocity Values for both leg.



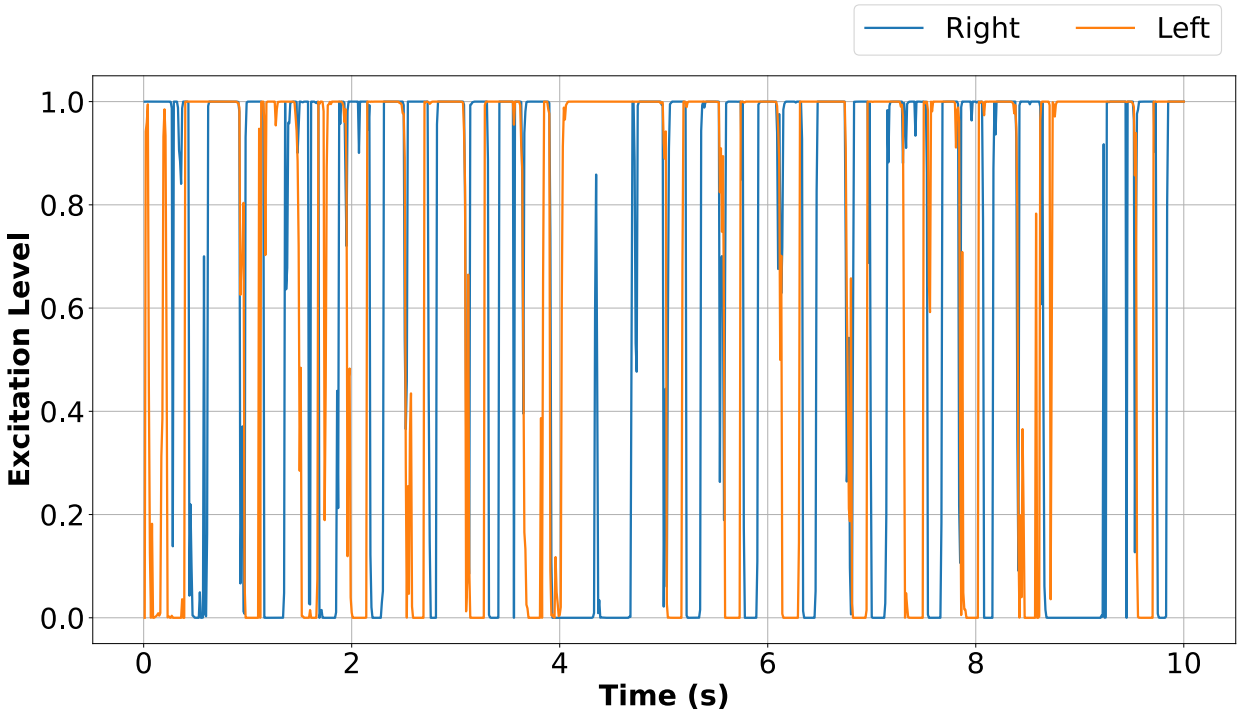
(a) Biceps Femoris, Short Head (BFSH)



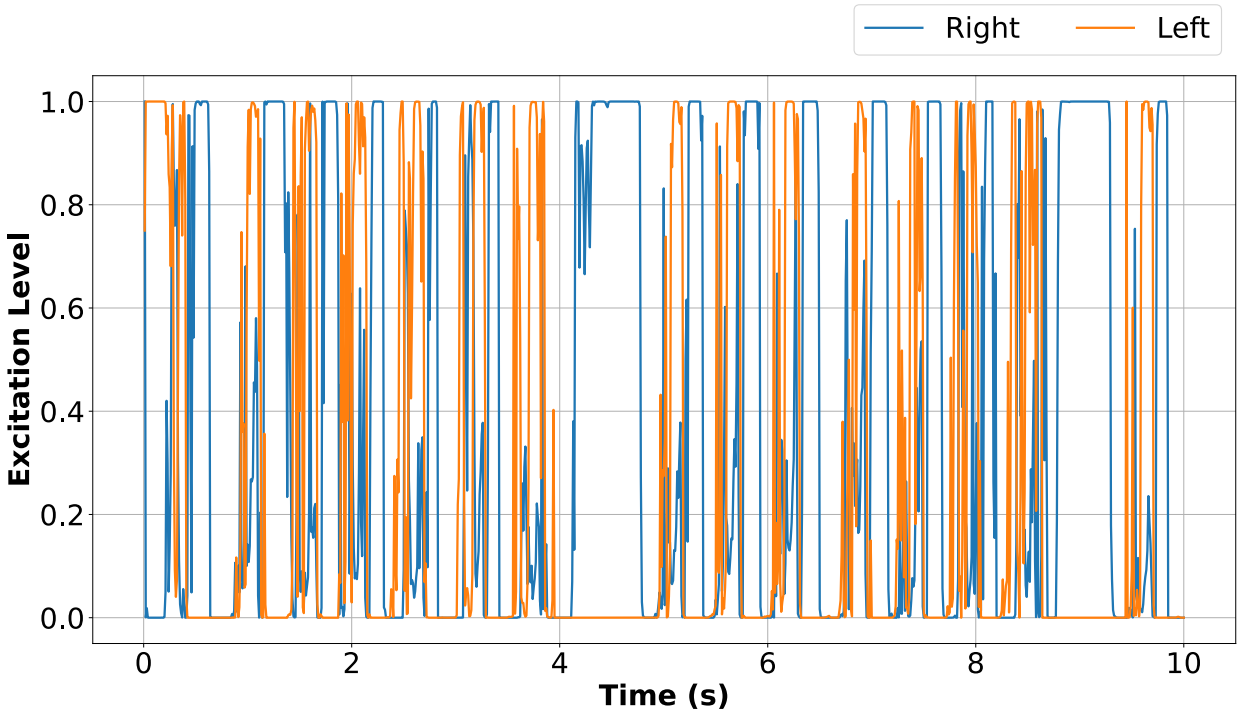
(b) Gastrocnemius (GAS)



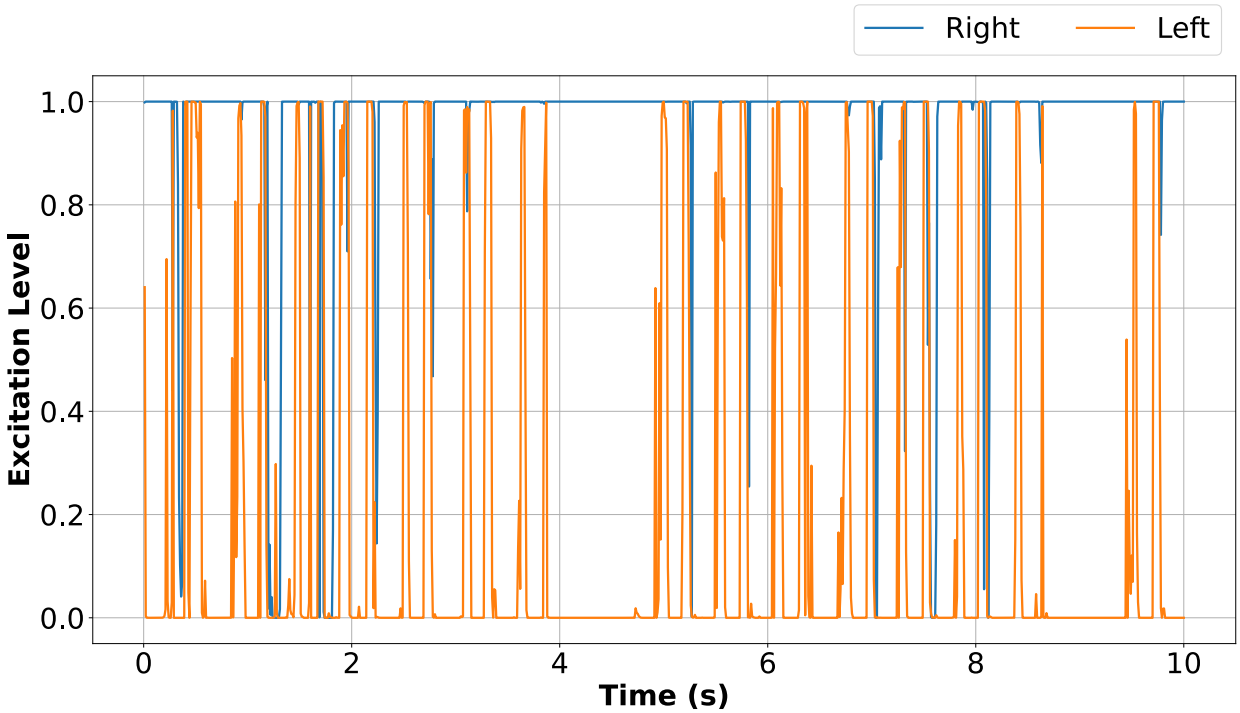
(c) Glutei (GLU)



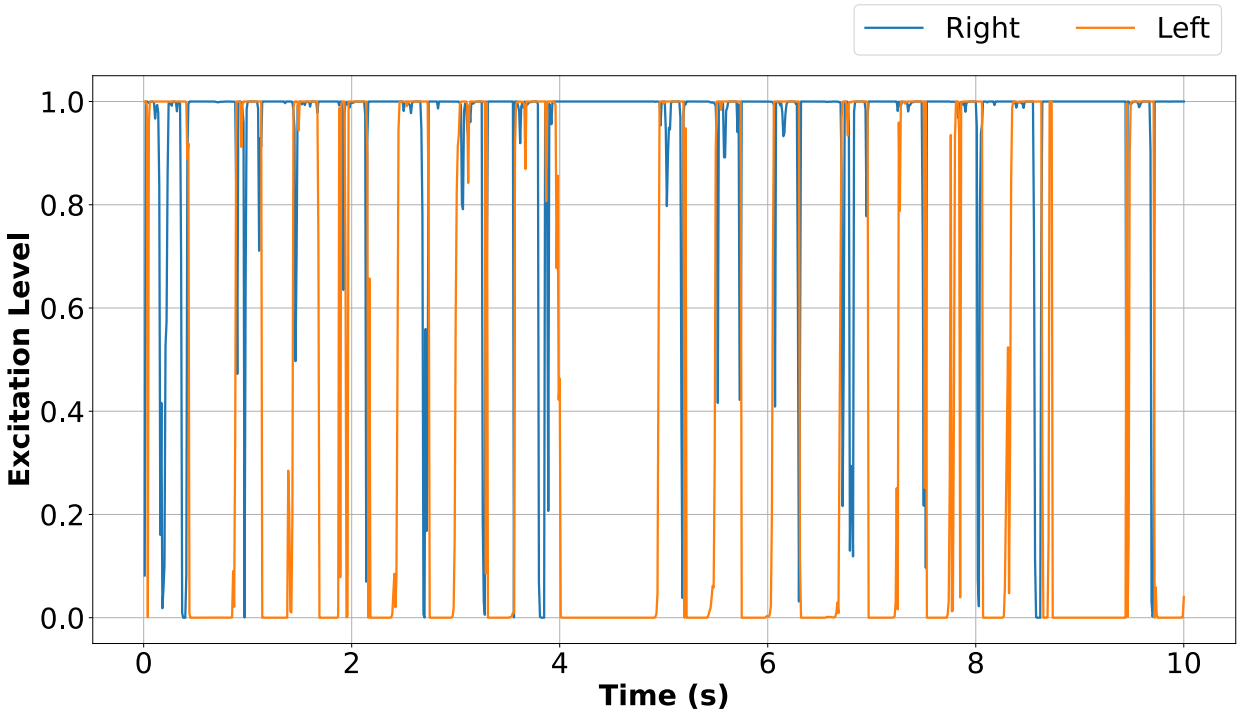
(d) Hip Abductor (HAB)



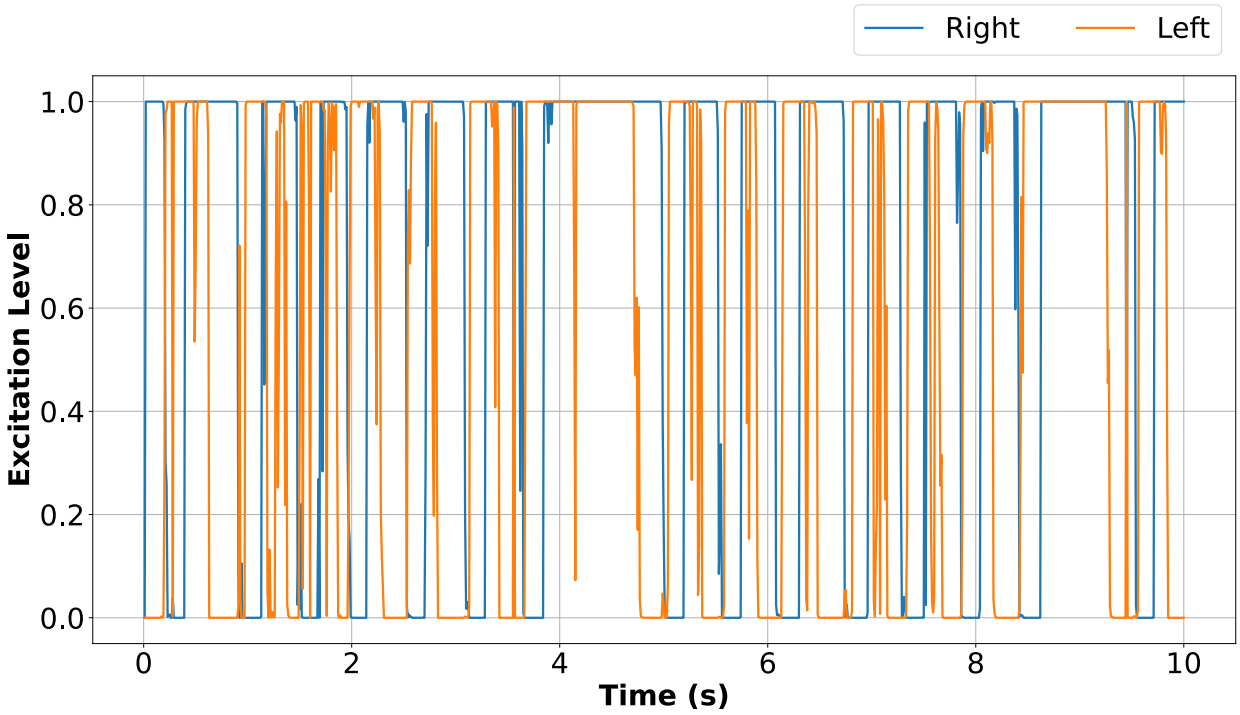
(e) Hip Adductor (HAD)



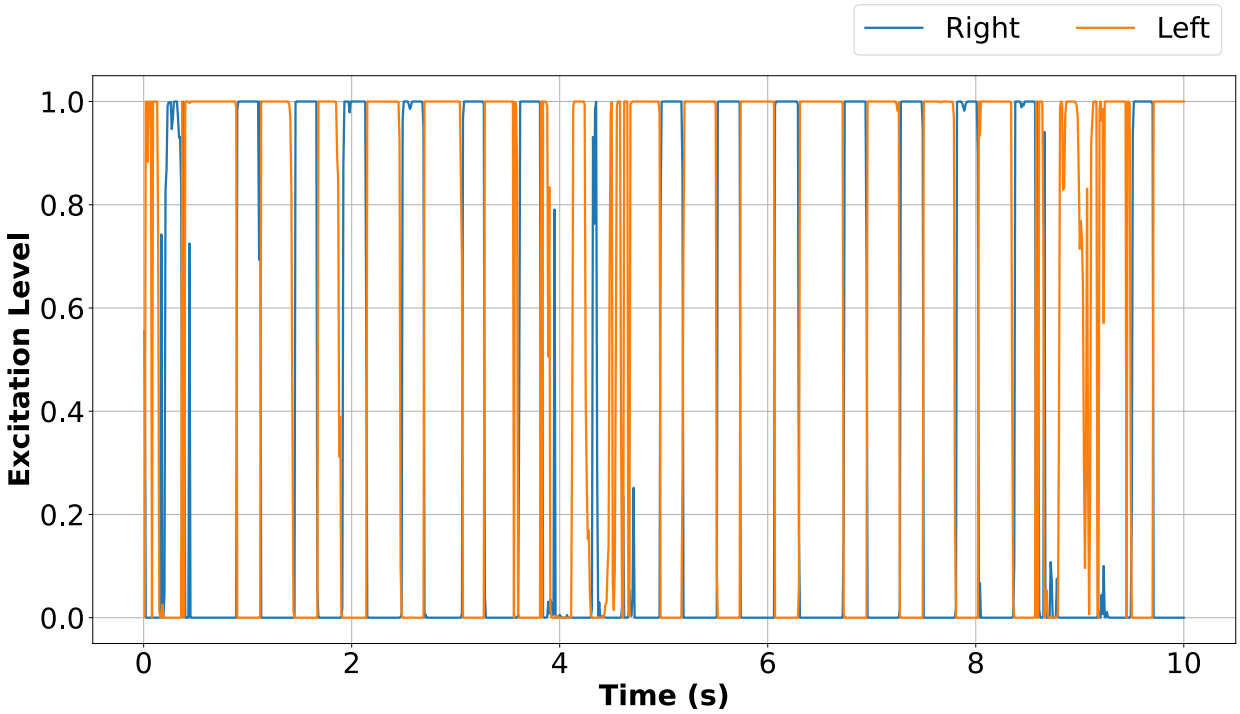
(f) Hamstrings (HAM)



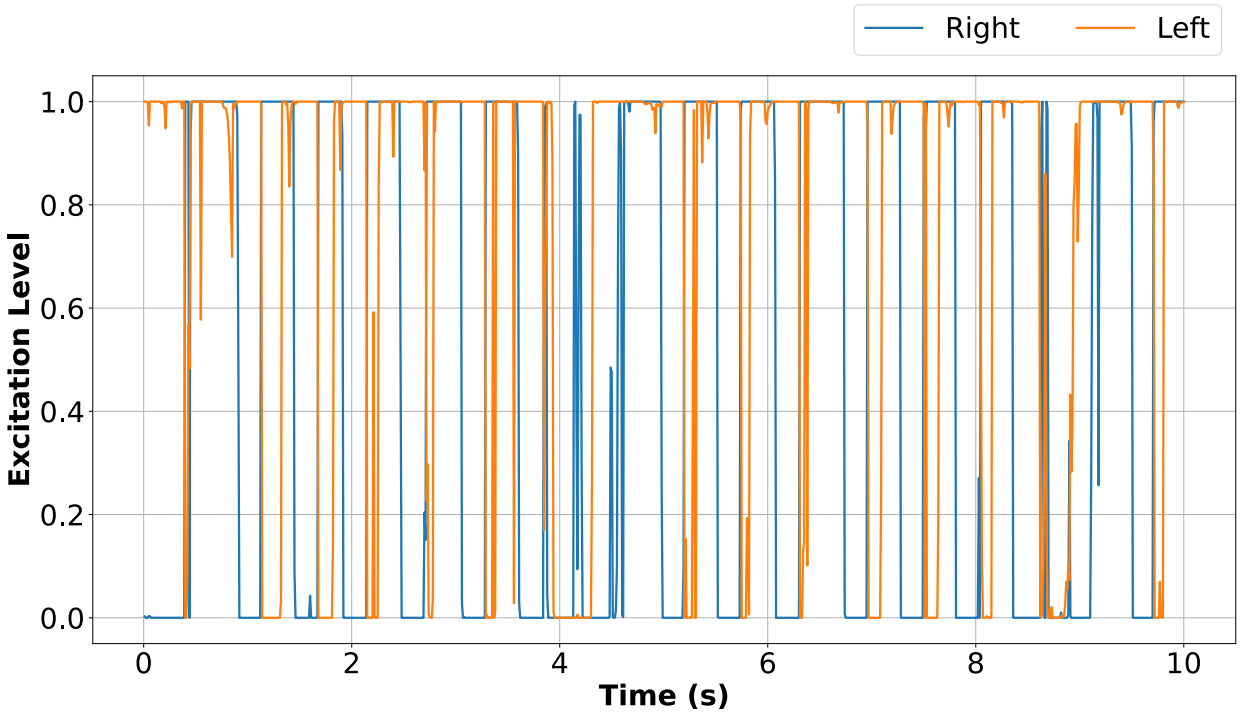
(g) Hip Flexor (HFL)



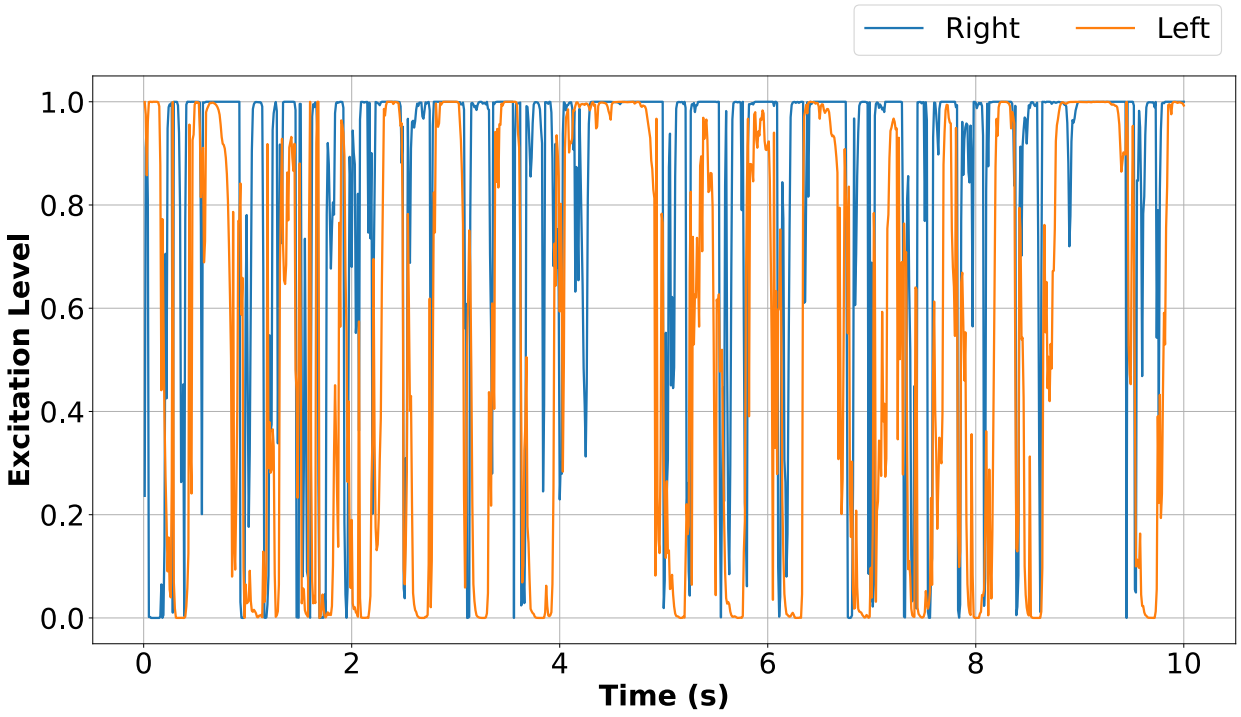
(h) Rectus Femoris (RF)



(i) Soleus (SOL)



(j) Tibialis Anterior (TA)



(k) Vastii (VAS)

Figure 15: Excitation levels for each muscle for the right and the left leg.
Right leg is in blue and left leg is in orange

11 Conclusion

A brief summary of the work done so far is provided as a starter for the conclusion. It will be followed by a posterior view of the initial objectives and then some tracks are given to improve this project further.

11.1 Summary

First, the RL algorithm was chosen by comparing the implementation of MBVE and DDPG in the scope of this thesis. These tests were performed on simple problems to highlight the best choice and to limit the computation time needed. It turns out that in this context, DDPG was the fastest and most efficient algorithm.

Then, a comparative analysis was performed to select a neural network architecture to use in the training of the musculo-skeletal agent. 3 distinct simple architectures were developed and tested. After testing, the variant Multi-Layer Perceptron was chosen for its performance on the test environments.

Thereafter, other considerations in the training process were analyzed. In particular, frame skipping showed a great interest in training. The importance of the shape of the reward and pre-training an agent with imitation learning was investigated. All those considerations helped to improve the final training process.

Finally, the quality of the developed controller was analyzed from a pure RL perspective by looking at the total reward of the agent in the environment. In addition, a more physiological analysis was provided focusing on the gait of the agent. This showed that even if the agent can move forward, the gait is not symmetrical as opposed to the common human gait.

11.2 Fulfilments of Objectives

Back to the 3 main objectives of this thesis. As a reminder the 3 objectives were:

1. Reduce the computational cost of the usage of an accurate model
2. Give a detailed description of the procedure used
3. Provide an analysis of the resulting controller

Concerning the first objective, it would be presumptuous to say that the method developed in this work significantly reduces the computational cost. Indeed, despite the use of imitation learning at first for pre-training and frame-skipping that reduces the total training time needed. Their impact on the computational cost is limited and additional work may be needed to truly improve the computational limitations.

Then, the second objective consisted of a detailed procedure description. The whole process and decisions for choosing the different elements of the training process were explained.

Hence, this objective is fulfilled.

Similarly, the third and last objective is also completed. Once the controller is trained, a short analysis of its behavior was performed to criticize the knowledge it has learned.

Finally, the overall objectives presented in the context of this are partially accomplished. Even if they were far from ambitious, they tend to give insights for further developments in the field of human motion simulation with reinforcement learning.

11.3 Going Further

To improve further this project, there is always the possibility to train the agent until the complete convergence of the policy. This would of course take a lot of time since this approach is computationally costful.

One of the results was that the learned dynamics model was not accurate enough to be able to bring the model-based approach above the model-free approach. Thus, it would be interesting to study in more detail how to provide a good dynamics model and to what extent a neural network can estimate an environment.

Directly following this, improving the algorithms using a model-based approach can also be very interesting. The improvements can be to lower the dependency of the algorithm performances on the learned dynamics such as using an ensemble of networks [2]. Since basic algorithms were used in this project, one could also use more advanced algorithms to solve this problem. Such algorithms could however be more costly in terms of computation. These advanced methods may be linked with the usage of stochastic neural networks to account for the stochasticity of the environment and thus provide better results.

An aspect that could be improved is the architecture of the neural networks. Indeed, memory can be included in the neural networks to improve further the performance of the agent in the environment. Learning how to walk or move with such memory-based controllers is shown to work with a certain efficiency [3, 34, 35]. This type of architecture could then be used to generalize a controller to work for multiple morphologies. Such a property for a controller is crucial for simulating the motions of different entities. Prior work on simple problems showed promising results [3].

A method that was not used in this thesis, called transfer learning, can also be studied to train an agent in a simpler environment. This simpler environment being cheaper in terms of computation could be used extensively. The idea is then to transfer the knowledge to the complex environment.

Appendix

A Detailed Architectures

Here follows the different neural network architectures developed in the context of this thesis.

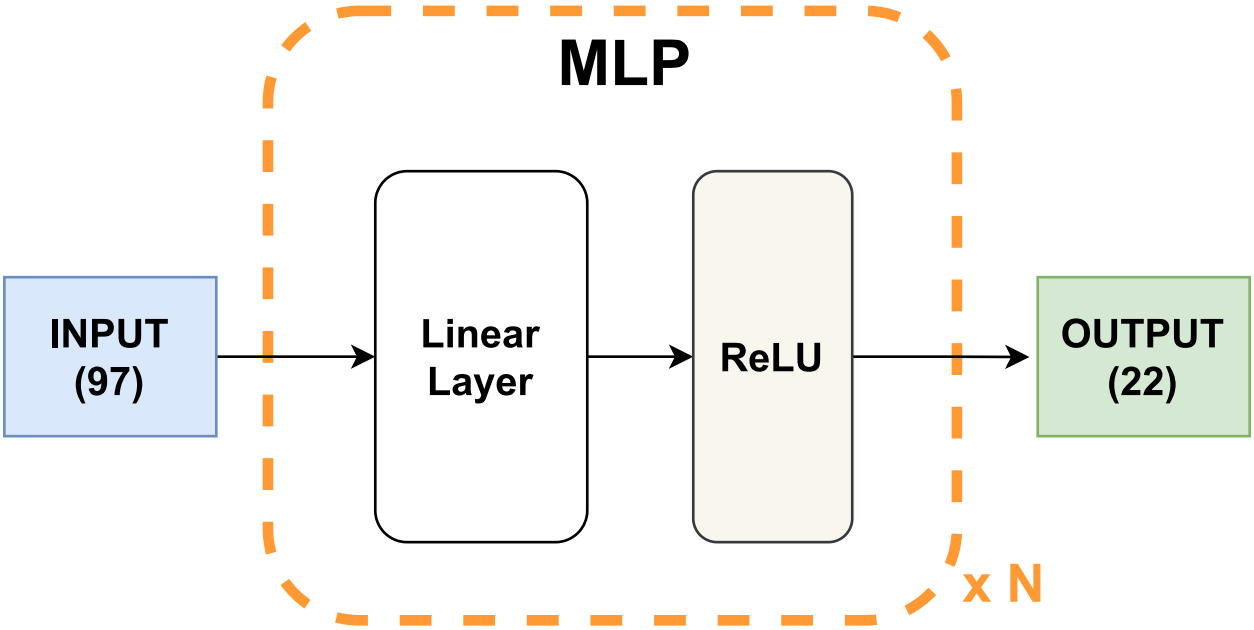


Figure 16: Multi-Layer Perceptron (MLP) Architecture

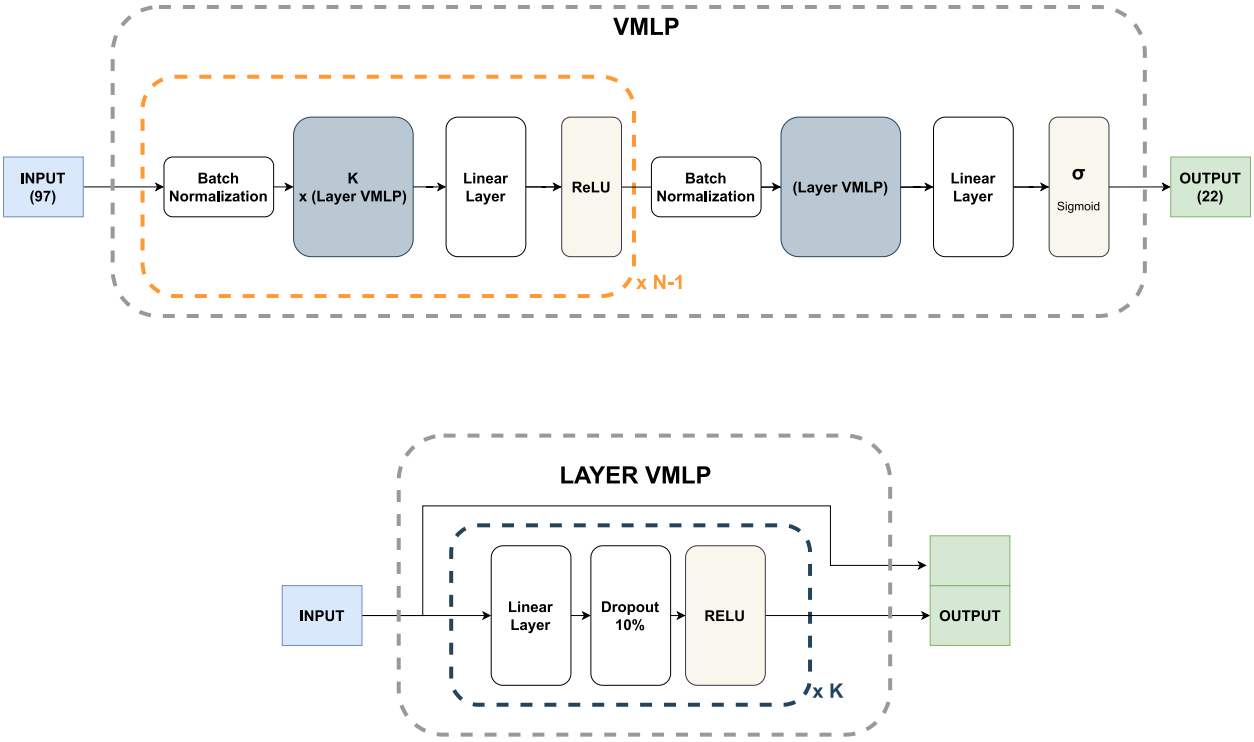


Figure 17: Variant Multi-Layer Perceptron Architecture

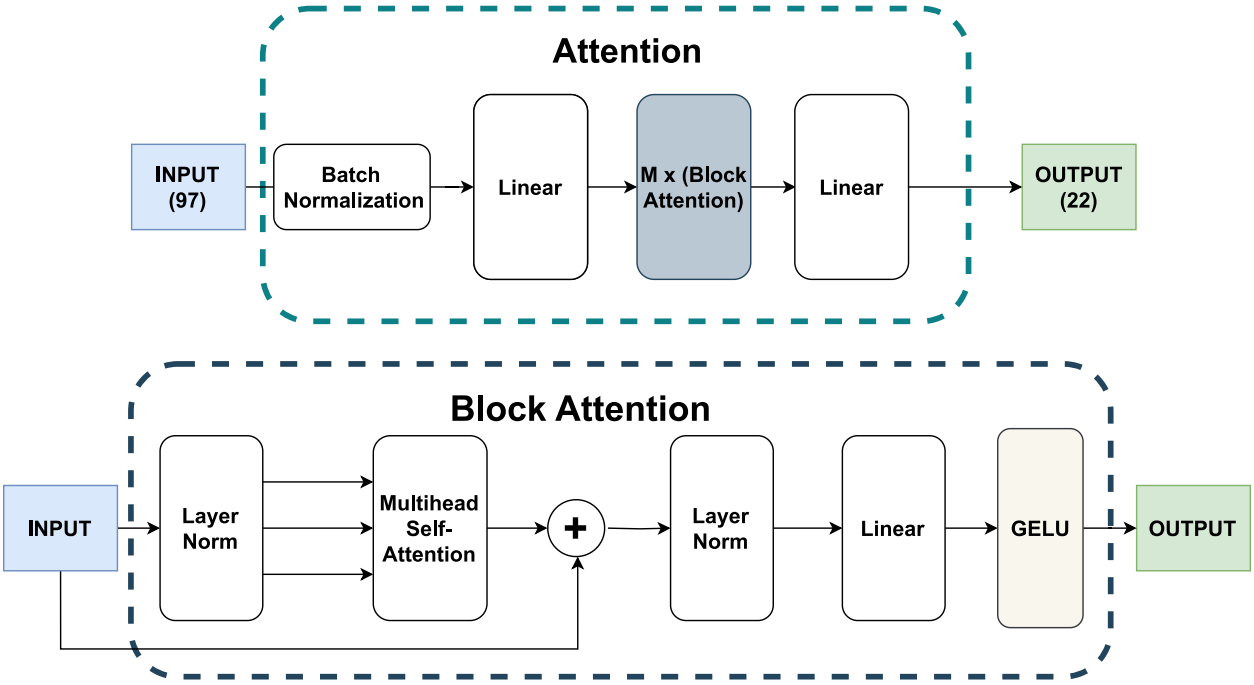


Figure 18: Self-Attention Architecture

The following table summarizes the network parameters for the final controller:

N	4
K	4
Layer Width	512

Table 6: Variant MLP parameters for final controller

B Algorithms

Algorithm 1 Deep Deterministic Policy Gradient (DDPG)

Initialization: Initialize actor network parameters θ and critic network parameters ϕ randomly. Initialize target network parameters $\theta_{tar} \leftarrow \theta$ and $\phi_{tar} \leftarrow \phi$. Empty replay buffer $\mathcal{D}$.
Sample initial state: Observe initial state s_0.
repeat
    Select action a_t according to the current policy $a_t = \pi_\phi(s_t)$ and the exploration method (see section 5.4).
    Take action a_t, observe next state s_{t+1}, receive reward r_t, and terminal signal d_t.
    Store the transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{D}$.
    If d_t is 1, then reset the environment.
    if it is time to update **then**
        for a fixed number of updates **do**
            Sample randomly a batch of transitions, $B = \{(s, a, r, s\prime, d)\}$ from replay buffer $\mathcal{D}$
            Compute the target Q-value:

$$y(r, s\prime, d) = r + \gamma(1 - d)Q_{\phi_{tar}}(s\prime, \pi_{\theta_{tar}}(s\prime))$$

            Update the critic network by one step of gradient descent using:

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s\prime,d) \in B} \Big(y(r, s\prime, d) - Q_\phi(s, a) \Big)^2$$

            Update the actor-network by one step of gradient descent using:

$$-\nabla_\theta \frac{1}{|B|} \sum_{(s,a,r,s\prime,d) \in B} Q_\phi(s, a)$$

            Update target networks with soft updates:

$$\theta_{tar} \leftarrow \rho \cdot \theta + (1 - \rho) \cdot \theta_{tar}$$

$$\phi_{tar} \leftarrow \rho \cdot \phi + (1 - \rho) \cdot \phi tar$$

        end for
    end if
until convergence

Algorithm 2 Model-Based Value Expansion (MBVE)

Initialization: Initialize actor network parameters θ and critic network parameters ϕ randomly. Initialize target network parameters $\theta_{tar} \leftarrow \theta$ and $\phi_{tar} \leftarrow \phi$. Initialize dynamics model parameters ζ Empty replay buffer $\mathcal{D}$.
Sample initial state: Observe initial state s_0.
repeat
    Select action a_t according to the current policy $a_t = \pi_\phi(s_t)$ and the exploration method (see section 5.4).
    Take action a_t, observe next state s_{t+1}, receive reward r_t, and terminal signal d_t.
    Store the transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{D}$.
    If d_t is 1, then reset the environment.
    if it is time to update **then**
        for a fixed number of updates **do**
            Sample randomly a batch of transitions, $B_0 = \{(s, a, r, s\prime, d)\}$ from replay buffer $\mathcal{D}$
            Update the dynamics model network by one step of gradient descent using:

$$\nabla_\zeta \mathcal{L}(\zeta, B_0)$$

            Update the actor-network by one step of gradient descent using:

$$-\nabla_\theta \frac{1}{|B_0|} \sum_{(s,a,r,s\prime,d) \in B} Q_\phi(s, a)$$

            Generate rollouts of length H from B_0 using $\hat{f}_\zeta$:

$$B_H = \{(s_{-1}, a_{-1}, r_{-1}, \hat{s}_0, d_{-1}, \hat{a}_0, \hat{r}_0, \hat{s}_1, \hat{d}_0, \ldots, \hat{a}_{H-1}, \hat{r}_{H-1}, \hat{s}_H, \hat{d}_{H-1}, \hat{a}_H)\}$$

            Update the critic network by one step of gradient descent using:

$$\nabla_\phi \frac{1}{|B_H|} \sum_{\tau \in B_H} \mathcal{L}(\phi, \tau)$$

            Update target networks with soft updates:

$$\theta_{tar} \leftarrow \rho \cdot \theta + (1 - \rho) \cdot \theta_{tar}$$

$$\phi_{tar} \leftarrow \rho \cdot \phi + (1 - \rho) \cdot \phi tar$$

        end for
    end if
until convergence

C Tools

In the context of this thesis, the PyTorch library was mainly used. All the algorithms were implemented by hand. There was a try to use the library PyTorch-RL but it wasn't possible to adapt the algorithm as it was intended to.

Some parts of the collection of the data were unclear. As such, all RL algorithms were re-implemented.

Also, the library and codes used by the top solutions [36, 26, 28] could not be reproduced even after numerous trials.

It was thus abandoned since it took too much time and did not work.

The OpenSim and OpenSim-RL python libraries were also used to use the OpenSim environment. As for the environments, the gym library with the MuJoCo environments is also used.

This was the list of all the different tools used during this thesis.

References

[1] Bo Zhou, Fan Wang, Hongsheng Zeng, and Hao Tian. Risk averse value expansion for sample efficient and robust policy learning, 2020.

[2] Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. *CoRR*, abs/1807.01675, 2018.

[3] Alberto Silvio Chiappa, Alessandro Marin Vargas, and Alexander Mathis. Dmap: a distributed morphological attention policy for learning to locomote with a changing body. *arXiv preprint arXiv:2209.14218*, 2022.

[4] Zijie Ye, Haozhe Wu, and Jia Jia. Human motion modeling with deep learning: A survey. *AI Open*, 3:35–39, 2022.

[5] Seung-Hee Lee, Moon Seok Park, Kyoungmi Lee, and Jehee Lee. Scalable muscle-actuated human simulation and control. *ACM Transactions on Graphics (TOG)*, 38:1 – 13, 2019.

[6] Ariel Kwiatkowski, Eduardo Alvarado, Vicky Kalogeiton, C. Karen Liu, Julien Pettré , Michiel van de Panne, and Marie-Paule Cani. A survey on reinforcement learning methods in character animation. *Computer Graphics Forum*, 41(2):613–639, may 2022.

[7] Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite-horizon model predictive control for periodic tasks with contacts. 06 2011.

[8] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. *Advances in Neural Information Processing Systems*, 06 2015.

[9] Seungmoon Song and Hartmut Geyer. A neural circuitry that emphasizes spinal feedback generates diverse behaviours of human locomotion. *The Journal of physiology*, 593(16):3493–3511, 2015.

[10] Florin Dzeladini, Nadine Ait-Bouziad, and Auke Ijspeert. *CPG-Based Control of Humanoid Robot Locomotion*, pages 1099–1133. Springer Netherlands, Dordrecht, 2019.

[11] Yifeng Jiang, Tom Van Wouwe, Friedl De Groote, and C. Karen Liu. Synthesis of biologically realistic human motion using joint torque actuation. *CoRR*, abs/1904.13041, 2019.

[12] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. *CoRR*, abs/1906.08253, 2019.

[13] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *CoRR*, abs/1803.00101, 2018.

[14] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[15] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[16] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, PP:1–1, 09 2019.

[17] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[18] Deep Deterministic Policy Gradient — Spinning Up documentation. https://spinningup.openai.com/en/latest/algorithms/ddpg.html. Accessed 2023-06-01.

[19] Ruishan Liu and James Zou. The effects of memory replay in reinforcement learning. *CoRR*, abs/1710.06574, 2017.

[20] Seungmoon Song, Łukasz Kidziński, Xue Peng, Carmichael Ong, Jennifer Hicks, Sergey Levine, Christopher Atkeson, and Scott Delp. Deep reinforcement learning for modeling human locomotion control in neuromechanical simulation. *Journal of NeuroEngineering and Rehabilitation*, 18, 08 2021.

[21] Ajay Seth, Jennifer L. Hicks, Thomas K. Uchida, Ayman Habib, Christopher L. Dembia, James J. Dunne, Carmichael F. Ong, Matthew S. DeMers, Apoorva Rajagopal, Matthew Millard, Samuel R. Hamner, Edith M. Arnold, Jennifer R. Yong, Shrinidhi K. Lakshmikanth, Michael A. Sherman, Joy P. Ku, and Scott L. Delp. Opensim: Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement. *PLOS Computational Biology*, 14(7):1–20, 07 2018.

[22] First-Order Activation Dynamics - OpenSim Documentation - Site global. https://simtk-confluence.stanford.edu:8443/display/OpenSim/First-Order+Activation+Dynamics. Accessed 2023-06-03.

[23] K. H. Hunt and F. R. E. Crossley. Coefficient of restitution interpreted as damping in vibroimpact. 42(2):440–445.

[24] Thelen 2003 Muscle Model - OpenSim Documentation - Site global. https://simtk-confluence.stanford.edu:8443/display/OpenSim/Thelen+2003+Muscle+Model. Accessed 2023-06-03.

[25] How forward dynamics works - OpenSim documentation - site global. https://simtk-confluence.stanford.edu:8443/display/OpenSim/How+Forward+Dynamics+Works. Accessed 2023-06-03.

[26] Sergey Kolesnikov and Valentin Khrulkov. Sample efficient ensemble learning with catalyst.rl. *CoRR*, abs/2003.14210, 2020.

[27] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods, October 2018. arXiv:1802.09477 [cs, stat].

[28] Dmitry Akimov. Distributed soft actor-critic with multivariate reward representation and knowledge distillation. *CoRR*, abs/1911.13056, 2019.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[30] Daniel Palenicek, Michael Lutter, and Jan Peters. Revisiting model-based value expansion, 2022.

[31] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *CoRR*, abs/1805.12114, 2018.

[32] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *CoRR*, abs/1708.02596, 2017.

[33] Guoqing Ma, Zhifu Wang, Xianfeng Yuan, and Fengyu Zhou. Improving model-based deep reinforcement learning with learning degree networks and its application in robot control. *Journal of Robotics*, 2022:1–14, 03 2022.

[34] Jonah Siekmann, Srikar Valluri, Jeremy Dao, Lorenzo Bermillo, Helei Duan, Alan Fern, and Jonathan W. Hurst. Learning memory-based control for human-scale bipedal locomotion. *CoRR*, abs/2006.02402, 2020.

[35] C.L.P. Chen. Deep q-learning with recurrent neural networks. 2016.

[36] Bo Zhou, Hongsheng Zeng, Fan Wang, Yunxiang Li, and Hao Tian. Efficient and robust reinforcement learning with uncertainty-based value expansion. 12 2019.

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

[38] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[41] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[42] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[44] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *CoRR*, abs/2106.01345, 2021.

[45] Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Çaglar Gülçehre, Siddhant M. Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, Matthew M. Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning. *CoRR*, abs/1910.06764, 2019.

[46] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019.

[47] GPT implementation by Gilles Louppe in the Lecture 8 of INFO8010. https://github.com/glouppe/info8010-deep-learning/tree/master/code/gpt. Accessed: 2021-05-17.

[48] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. 01 2015.

[49] Shivaram Kalyanakrishnan, Siddharth Aravindan, Vishwajeet Bagdawat, Varun Bhatt, Harshith Goka, Archit Gupta, Kalpesh Krishna, and Vihari Piratla. An analysis of frame-skipping in reinforcement learning. 02 2021.

[50] Abhishek Gupta, Aldo Pacchiano, Yuexiang Zhai, Sham M. Kakade, and Sergey Levine. Unpacking reward shaping: Understanding the benefits of reward engineering on sample complexity, 2022.

[51] Jack Clark and Dario Amodei. Faulty reward functions in the wild. https://openai.com/research/faulty-reward-functions, 2016. Accessed on May 31st, 2023.

[52] Zhihui Xie, Zichuan Lin, Junyou Li, Shuai Li, and Deheng Ye. Pretraining in deep reinforcement learning: A survey, 2022.

[53] Samarth Sinha, Homanga Bharadhwaj, Aravind Srinivas, and Animesh Garg. D2RL: deep dense architectures in reinforcement learning. *CoRR*, abs/2010.09163, 2020.

Acronyms

BFSH Biceps Femoris, Short Head. 20, 48, 55

CE Contractile Element. 21, 22

CNS Central Nervous System. 6, 8

CPG Central Pattern Generator. 8

DDPG Deep Deterministic Policy Gradient. 12, 15, 16, 26, 30–33, 36, 40, 43, 44, 61, 67

DOF Degrees of Freedom. 17, 27

GAS Gastrocnemius. 20, 49, 55

GLU Glutei. 20, 56

HAB Hip Abductor. 20, 56

HAD Hip Adductor. 20, 57

HAM Hamstrings. 20, 49, 57

HFL Hip Flexor. 20, 58

MBPO Model Based Policy Optimization. 12

MBVE Model-Based Value Expansion. 7, 12, 15, 24–26, 30–32, 43, 61, 68

MDP Markov Decision Process. 9

ML Machine Learning. 9

MLP Multi-Layer Perceptron. 30, 36–38, 40, 43, 61, 63, 64

PEE Parallel Elastic Element. 22

RAVE Risk Averse Value Expansion. 7, 24

ReLU Rectified Linear Unit. 36

RF Rectus Femoris. 20, 58

RL Reinforcement Learning. 3, 6–9, 11, 15, 17, 26, 27, 30, 37, 43, 44, 61

SEE Series Elastic Element. 22

SOL Soleus. 20, 59

STEVE Stochastic Ensemble Value Expansion. 7, 24

TA Tibialis Anterior. 20, 59

TD Temporal Difference. 12, 13, 15, 16

VAS Vastii. 20, 60