INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Alexandre Ferreira Velho Muzio

DEEP REINFORCEMENT LEARNING APPLIED TO HUMANOID ROBOTS

Bachelor's Thesis
2017

Course of Computer Engineering

CDU 681.3:16

Alexandre Ferreira Velho Muzio

DEEP REINFORCEMENT LEARNING APPLIED TO HUMANOID ROBOTS

Advisor

Prof. Dr. Takashi Yoneyama (ITA)

Co-advisor

Prof. Ms. Marcos Ricardo Omena de Albuquerque Máximo (ITA)

COMPUTER ENGINEERING

São José dos Campos
Instituto Tecnológico de Aeronáutica

2017

Cataloging-in Publication Data
Documentation and Information Division

Muzio, Alexandre Ferreira Velho
Deep Reinforcement Learning Applied to Humanoid Robots / Alexandre Ferreira Velho Muzio.
São José dos Campos, 2017.
61f.

Bachelor's Thesis (Undergraduation study) – Course of Computer Engineering– Instituto Tecnológico de Aeronáutica, 2017. Advisor: Prof. Dr. Takashi Yoneyama. Co-advisor: Prof. Ms. Marcos Ricardo Omena de Albuquerque Máximo.

1. Dinâmica de robôs. 2. Robôs humanoides. 3. Locomoção por pernas. 4. Controle de robôs. 5. Inteligência artificial. 6. Robótica. 7. Controle. I. Instituto Tecnológico de Aeronáutica. II. Deep Reinforcement Learning Applied to Humanoid Robots.

BIBLIOGRAPHIC REFERENCE

MUZIO, Alexandre Ferreira Velho. **Deep Reinforcement Learning Applied to Humanoid Robots**. 2017. 61f. Bachelor's Thesis (Undergraduation study) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Alexandre Ferreira Velho Muzio
PUBLICATION TITLE: Deep Reinforcement Learning Applied to Humanoid Robots.
PUBLICATION KIND/YEAR: Bachelor's Thesis (Undergraduation study) / 2017

Alexandre Ferreira Velho Muzio
Rua H8 A, 139
12228-460 – São José dos Campos–SP

DEEP REINFORCEMENT LEARNING APPLIED TO HUMANOID ROBOTS

This publication was accepted like Final Work of Undergraduation Study

Alexandre Ferreira Velho Muzio

Author

Takashi Yoneyama (ITA)

Advisor

Marcos Ricardo Omena de Albuquerque Máximo (ITA)

Co-advisor

Prof. Dr. Cecília de Azevedo Castro César
Course Coordinator of Computer Engineering

São José dos Campos: November 21. 2017.

To Ana Carla, partner in crime, partner in life.

Acknowledgments

First and foremost, I would like to thank God for all the blessings in my life.

A special thanks to my family (Mom, Dad and Aline) for all their support and unconditional love for me. They have always stood by my side since I can recall and have made me who I am. They are my heros.

I would also like to thank my best friend and love, Ana Carla. Her kindness and will to stay by side helped me believe I could do it.

Thanks to all my friends from COMP17. I learned so much in such a short period of time. You guys made the days more cheerful.

Thanks to all my colleagues at ITAndroids. In special, thanks to Marcos Máximo, for all his insight and guidance. He's willpower does not cease to inspire me.

"This page intentionally left blank."
— Unknown Author

Resumo

Futebol de robôs humanoides é uma tarefa muito competitiva que desafia os limites da robótica de ponta. Um dos diversos desafios de futebol de rôbos é caminhar e correr sem perder o equilíbrio. *Deep Reinforcement Learning* (DRL), que mistura algoritmos de *Deep Learning* com aprendizado por reforço, recentemente tem sido uma ferramenta muito utilizada em robótica para atacar problemas complexos de controle em espaços de ações contínuos. *Model-free* DRL é muito eficiente para o problema de locomoção de robôs, já que possibilita evitar modelar as dinâmicas complexas de um robô humanoide. O foco desse trabalho é o problema de corrida de robôs humanoides. Usando algoritmos de DRL *model-free* baseados no cálculo de gradientes da política (*Proximal Policy Optimization*, *Trust Region Policy Optimization* e *Deep Deterministic Policy Gradients*), desenvolveu-se um controlador *Follow-Line* robusto para corrida de rôbos humanoides. Finalmente, o controlador foi avaliado em um robô Nao simulado da RoboCup 3D Soccer Simulation League usando métricas clássicas de aprendizado por reforço.

Abstract

Humanoid robot soccer is a very traditional competitive task that aims to push the boundaries of state of the art robotics. One of the many challenges of playing soccer is walking and running while not losing balance. Modern Deep Reinforcement Learning (DRL), that combines Deep Learning algorithms to Reinforcement Learning (RL), has been frequently used to solve complex continuous control problems such as those in robotics. Model-free DRL is very effective for robot locomotion since it avoids directly dealing with the complex dynamics of a humanoid robot. In this work, we focus on the problem of humanoid racing. Using DRL Policy Gradients model-free algorithms (Proximal Policy Optimization, Trust Region Policy Optimization and Deep Deterministic Policy Gradients) we effectively develop a robust and stable Follow-Line controller that allows humanoid robots to race. Finally, the controller was evaluated on a simulated Nao robot from the RoboCup 3D Soccer Simulation League using classic performance metrics from Reinforcement Learning.

List of Figures

List of Tables

List of Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DDPG	Deep Deterministic Policy Gradients
DPG	Deterministic Policy Gradients
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GPU	Graphics Processing Unit
MC	Monte-Carlo
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilayer Perceptron
NN	Neural Network
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SGD	Stocastic Gradient Descent
TD	Temporal-Difference
ZMP	Zero Moment Point

List of Symbols

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
a_i	Element i of vector a, with indexing starting at 1
A^T	Transpose of matrix A
$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
∇y	Gradient of y
$\int_a^b f(x)dx$	Definite integral with respect to x over [a, b]
$f : \mathbb{A}^2 \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f(x, \theta)$	A function of x parameterized by θ
$\mathbb{P}(x)$	Probability distribution over a continuous variable
$\mathbb{E}[X]$	Expectation of random variable X
$\mathrm{KL}[P, Q]$	KL divergence between P and Q
$\log x$	Natural logarithm of x

Reinforcement Learning

$s, s\prime$	States
a	Action
r	Reward
t	Discrete timestep
A_t	Action at timestep t
S_t	State at timestep t
R_t	Reward at timestep t
G_t	Return (cumulative discounted reward) following time t
π	Policy, agent behavior rule
$\pi(s)$	Action taken in state s under policy π

$v_\pi(s)$	Value of state s under policy π (expected return)
$v_*(s)$	Value of state s under the optimal policy
$q_\pi(s, a)$	Value of taking action a in state s under policy π
$q_\pi(s, a)$	Value of taking action a state s under policy π
V, V_t	Estimate of state-value function v_π or v_*
Q, Q_t	Estimate of action-value function q_π or q_*

Contents

1 Introduction

1.1 Motivation

Artificial Intelligence (AI) has never been more part of our civilization as it is today. It is even considered a threat to humans given it's power.

In 2015, DeepMind created an AI that learned how to play 50 Atari games using the same learning algorithm (including the same hyper parameters). The algorithm is called Deep Q-Learning (DQN) and the only inputs are the pixel values from the screen (MNIH *et al.*, 2015).

Also, in 2015, DeepMind built AlphaGo (Figure 1.1): the first computer program to defeat a professional human player in Go. Go is a very complex board game, believed to be the oldest game (created over 2500 years ago) still being played today. It has over 10^{170} states and was considered an incredible feat.

But Artificial Intelligence is not only learning how to play games. It is being employed in the most diverse areas: a recent artificial system based on Deep Neural Networks capable of artistic visual perception (GATYS *et al.*, 2015) shows off just how diverse AI can be. Figure 1.2 shows the algorithm in action.

AI also has multiple applications in Robotics. The latter is a field of engineering that deals with multiple areas like Control Theory, Eletrical Engineering and Computer Science.

FIGURE 1.1 – DeepMind's AlphaGo logo.

FIGURE 1.2 – Example execution of artistic style combination algorithm.

Robotics has been studied exhaustively as a domain for applying Machine Learning (ML). For example, Google's Self Driving Car (Figure 1.3) extensively uses Artificial Intelligence to spot pedestrians and to navigate between lanes.

RoboCup is an international robot soccer competition with the goal of advancing state of the arts robotics and AI. It is composed of several leagues that range from simulated robots to real robots, but what they have in common is that the robots play soccer autonomously. This is yet another example of AI applied to Robotics.

1.2 Problem Statement

Inspired by the RoboCup Soccer Simulation 3D league, our objective is to learn a high level racing behavior for simulated humanoid robots. We are not interested in the problem of robot locomotion using the joint angles as inputs, and therefore, we use a ZMP-based walk engine algorithm that outputs the joint angles from commanded velocities.

FIGURE 1.3 – Google's Self Driving Car.

1.3 Approach

Instead of explicitly defining the behavior of the simulated humanoid robot through explicit source code and heuristics we use a model-free deep reinforcement learning based approach that tries to learn the desired behavior by interacting with an environment and receiving rewards depending on which actions it chooses. This way, we do not deal with the complex dynamics of a humanoid robot.

1.4 Literature Review on Deep Reinforcement Learning

Reinforcement Learning algorithms have been studied for over 30 years. Value function estimation algorithms such as TD-Learning (BARTO *et al.*, 1983) lay the groundwork for multiple RL control algorithms. Q-Learning (WATKINS, 1989), for example, estimates the action-value function to implicitly learn an optimal policy. It is a central algorithm for modern DRL.

Policy search methods, on the other hand, do not require a model of the value function to take actions, but directly search for an optimal policy π_* in the policy space. (WILIAMS, 1992) introduces the REINFORCE algorithm that provides an update rule for a parameterized policy. A more recent approach based on (WILIAMS, 1992) is Deterministic Policy Gradients (DDPG) (LILLICRAP *et al.*, 2015) that works very well for high dimensional continuous control problems.

Policy search methods have been used in DRL for problems with continuous state and action space, such as those of robotics.

The recent success of DRL have been focused on scaling the prior work of RL to high-dimensional problems.

The revolution to DRL kicked off with DQN (MNIH *et al.*, 2015). The algorithm learns successful policies directly from high-dimensional sensory inputs (raw pixels) in the domain of classic Atari 2600 games. DQN it the first RL algorithm demonstrated to work directly from raw visual inputs on a variety of environments.

DQN works by approximating the Q-function as a deep neural network composed of convolutional layers followed by dense ones. The output of the network is the Q-function for each of the possible action (joystick commands) and therefore only works for discrete action spaces. Figure 1.4 illustrates the DQN network.

DQN uses two techniques that address the instability problem of using function approximation of RL:

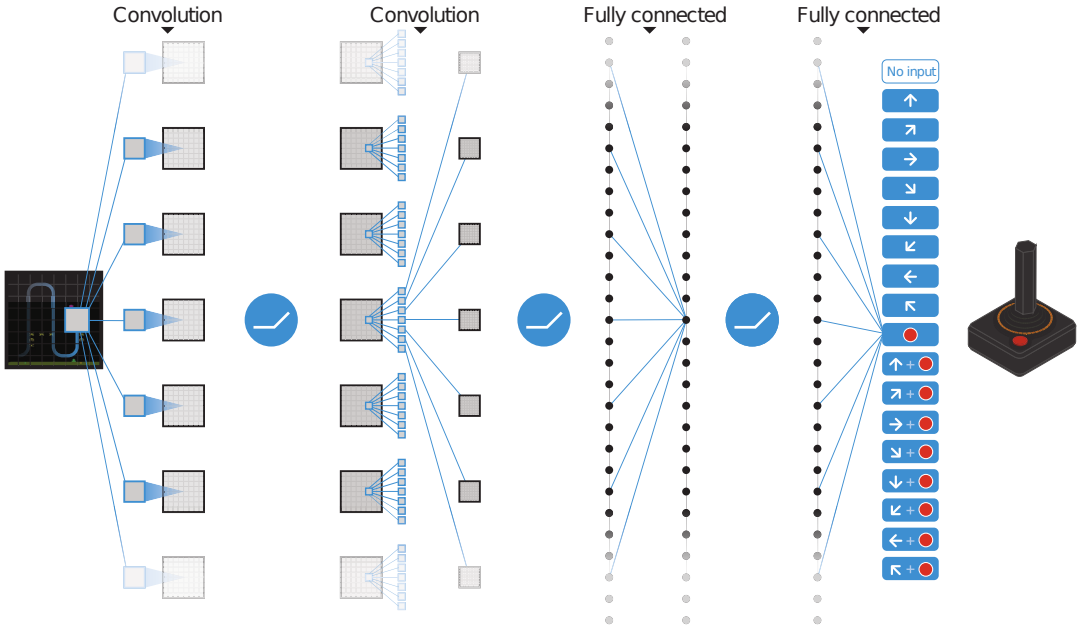


FIGURE 1.4 – Convolutional NNs used in DQN algorithm. Image from (MNIH *et al.*, 2015).

- **Experience replay** (LIN, 1992) that consists of saving transitions of the form $(S_t, A_t, S_{t+1}, R_{t+1})$ in a buffer that can be sampled offline.
- **Target Networks** introduced by (MNIH *et al.*, 2015) that consists of using a target network that is updated at a much slower with the weights of the Q-network.

The DDPG algorithm incorporates the techniques from DQN into DPG (SILVER *et al.*, 2014) making it capable of solving continuous control problems with high dimensional visual inputs (LILLICRAP *et al.*, 2015). It is an actor-critic algorithm that uses deep neural networks as estimators for the parameterized policy and parameterized value function.

Robotics has a range of problems that can be modeled using RL: an agent (robot) interacts with the environment obtaining rewards. In the simulated robot domain, specifically RoboCup Soccer Simulation 2D, (GABEL *et al.*, 2009) learns soccer defensive behaviors using a Multilayer Perceptron (MLP). Also, in the same soccer domain, (HAUSKNECHT; STONE, 2016) extends DDPG into a parameterized action space by bounding action space gradients.

1.5 Outline of this Dissertation

- **Chapter 1:** motivation and the objective of this work.
- **Chapter 2:** background about reinforcement learning.

- **Chapter 3:** background about deep learning.
- **Chapter 4:** describes the deep reinforcement learning algorithms used in this work.
- **Chapter 5:** methodology of this work and implementation.
- **Chapter 6:** description of results and experiments.
- **Chapter 7:** conclusions and future works.

2 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning that studies how an agent interacts with an environment receiving instant rewards (SUTTON; BARTO, 1998). Figure 2.1 illustrates this notion: an agent in state S_t at timestep t takes action A_t and ends up in state S_{t+1}, receiving an instant reward R_{t+1}.

RL is a simple yet powerful framework that defines the interaction between the agent and the environment. The agent learns from the environment in order to achieve long-term goals.

When modelling a problem with RL, there are 3 main components that must be carefully modeled.

- **State space**: Set of values that define the state of the agent in the environment. They are the information available to the target.
- **Action space**: Set of values that define the possible actions the agent can take in the environment.
- **Reward signal**: Defines the agent's goal.

The agent's goal is to maximize the accumulative reward it gets when interacting with the environment. Note that normally these interactions are episodic, which means that the task restarts after some number of steps.

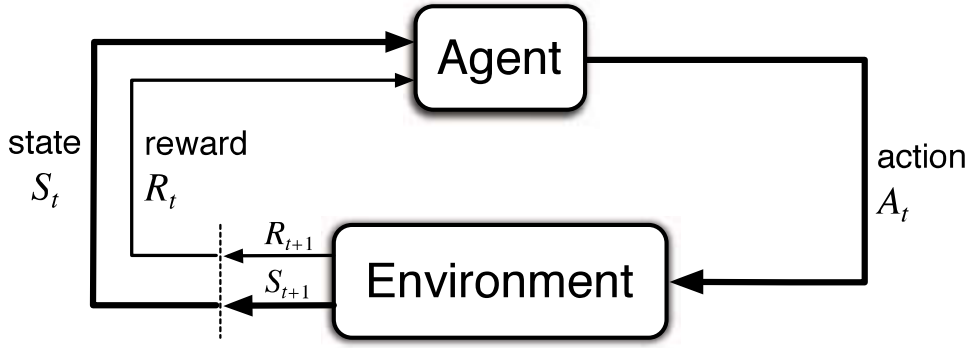


FIGURE 2.1 – Agent interacting with environment.

The following topics will describe the theory behind RL.

2.1 Markov Decision Process

Markov Decision Processes (MDP) defines a mathematical framework that formally describes this interaction with the environment. It is said that if the state of the agent contains all the necessary information from the past, it has the Markov Property and can be defined formally as:

$$\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_1, S_2, ...S_t) \tag{2.1}$$

Therefore, RL task is an MDP if it has the Markov Property.

A particular MDP is defined by

- State set
- Action set
- One-step dynamics of the environment, denoted as $\mathbb{P}(S_{t+1}|S_t, A_t)$.

2.2 Return

Cumulative reward that the agent receives starting at timestep t:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

Notice that $\gamma \in [0, 1]$ introduces the concept of discounting. The agent prefers to gain instant rewards instead of long term rewards.

2.3 Policies

A policy π is a distribution over actions given states. It is represented by Equation (2.3).

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.3}$$

The policy fully describes the behavior of the agent and can be deterministic or stochastic.

2.4 Value functions

The value functions are estimates of the agent's current return.

- State-value function

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \tag{2.4}$$

- Action-value function

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{2.5}$$

The value function estimates how good the agent's current state is (or how good it is for it to take a given action at that state).

2.5 Goal of RL

The goal of RL is to learn the optimal behavior (described by the policy) of the agent.

We will now define optimality for the value function and policy.

2.5.1 Optimal Value Function

The value functions are optimal if they are the highest value for all states (or state-action pairs).

- Optimal State-value function: $v_*(s) = \max_\pi v_\pi(s)$
- Optimal Action-value function: $q_*(s, a) = \max_\pi q_\pi(s, a)$

Therefore, solving an MDP means finding the optimal value function (or optimal policy).

2.5.2 Optimal Policy

Let the partial ordering of policies be

$$\pi \geq \pi\prime \text{ if } v_\pi(s) \geq v_{\pi\prime}(s)$$

The optimal policy is better than any other, $\pi_* \geq \pi, \forall\pi$ and achieves the optimal value functions.

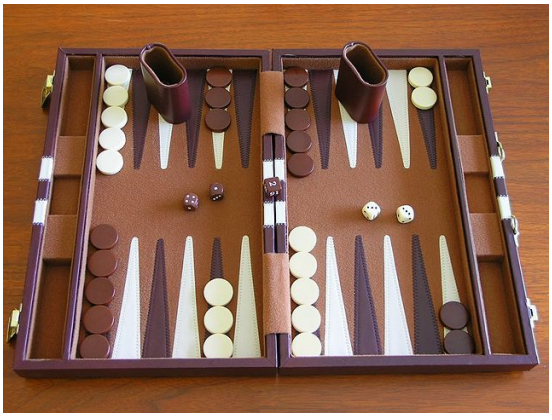
An interesting property of the optimal policy is that it can be implicitly defined with q_*.

The optimal policy can be defined as the following deterministic policy

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \underset{a}{\text{argmax}}\ q_*(s,a) \\ 0, & \text{otherwise} \end{cases}$$

2.6 Function Approximation

Many classic tabular RL algorithms work by estimating the value functions for each state, for example, Value Iteration methods. A problem arises when the state/action spaces are extremely large, or if the state/action spaces are continuous. Some examples of large discrete state spaces are board games. This is known as the *curse of dimensionality*. Backgammon has over 10^{20} (LEVNER, 1976) board states and the game of Go (with board size of 19x19) has over 10^{170} states (TROMP; FARNEBäCK, 2016). Figure 2.2 illustrates these games.



(a) Game of Backgammon.

(b) Game of Go.

FIGURE 2.2 – Example of board games that have large discrete state spaces.

There are 2 problems of having so many states in realistic situations:

- It will be impossible to store all the states in memory.

- It will be impossible to visit all the states in training.

For continuous domains, we have an analogous situation since it is possible to discretize the state or action space and then apply tabular methods.

Figure 2.3 illustrates 3 continuous control problems and the number of states resulting from one possible discretization.

FIGURE 2.3 – Game of Go.

If the discretized space is large, we still have the problem described above.

An approach to deal with this problem is to use a function approximation that estimates the value function. The approximate value function, $\hat{v}(s, \theta)$ will be defined by a parameter weight vector $\theta \in \mathbb{R}^n$ and

$$\hat{v}(s, \theta) \approx v_\pi(s)$$

The interesting part of function approximation is that this function estimator can be any function, ranging from a simple linear function of the states to a complex neural network. Normally the number of weights is much less than the number of states.

In this work, we will focus on using neural networks as function approximators, Chapter 3 will lay the groundwork for the theory regarding Neural Networks.

2.7 RL algorithms

There are 3 classes of RL algorithms: methods based on value functions, methods based on policy search and a hybrid approach which combines value functions and policy

search (SUTTON; BARTO, 1998).

2.7.1 Value Function Methods

Value function methods are based on estimating v or q functions.

Monte-Carlo methods and Temporal-Difference Learning are the two methods generally used for predicting the value function.

2.7.1.1 Monte-Carlo

MC prediction are model-free (no knowledge of the dynamics) methods that learn directly from complete episodes of experience. They estimate the value functions by averaging sample returns and therefore can only be applied to episodic MDPs.

2.7.1.2 Temporal-Difference Learning

TD Learning are model-free and can learn from incomplete episodes. TD Learning updates the estimate by *bootstrapping*, which means it updates the value function using an existing estimate.

The simplest TD method, known as TD(0), updates the value function as

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.6}$$

TD has some advantages in comparison to MC: TD can learn before the end of that episode (or in non episodic MDPS) and have low variance and some bias. MC on the other hand has high variance, zero bias, having good convergence properties (even with function approximation).

2.7.2 Policy Search Methods

Policy search methods do not have an estimate of the value function. These algorithms typically maintain a parameterized policy π_θ whose parameters are updated to maximize the expected return.

The Policy Gradient Theorem defines that for any differentiable policy $\pi_\theta(s, a)$ the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_{\pi_\theta} \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)] \tag{2.7}$$

An example usage of this theorem is seen in the REINFORCE algorithm that uses Equation (2.7) to update the policy's weight using gradient ascent (WILIAMS, 1992).

2.7.3 Actor-Critic Methods

Actor-critic methods also work by updating a parameterized policy π_θ but instead use a *critic* to estimate the action-value function Q^{π_θ},

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$$

where w is the weight vector that parameterizes Q. Therefore actor-critic algorithms maintain two sets of parameters: the *critic* that updates that action-value function parameters w and the *actor* that updates the policy parameters θ in the direction suggested by the *critic*.

Figure 2.4 illustrates the architecture of actor-critic algorithms.

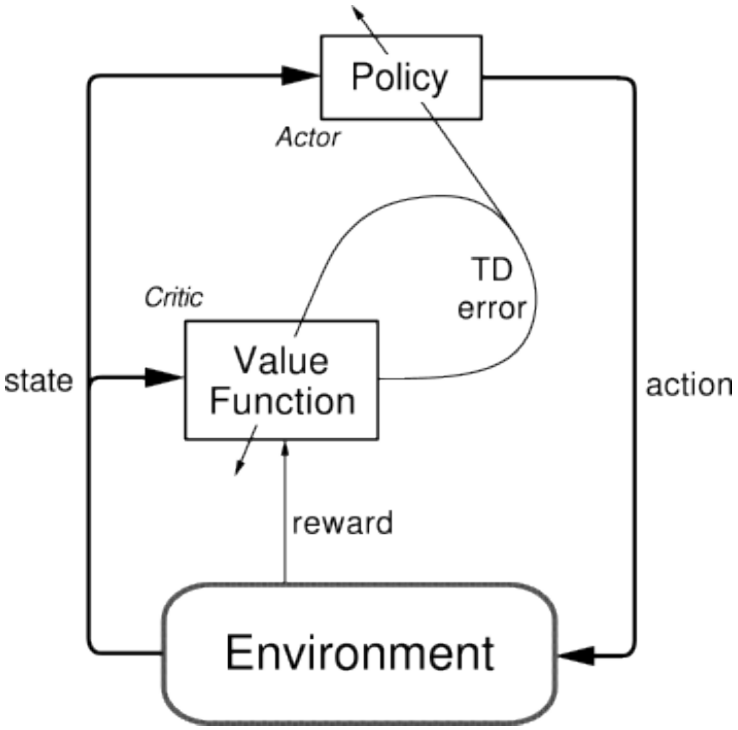


FIGURE 2.4 – Actor-critic architecture from (SUTTON; BARTO, 1998).

3 Deep Learning

Deep Learning is a part of Machine Learning that focus on methods of representation learning (how data is represented).

3.1 History

Today, Deep learning is a very hot topic for the scientific community however its history goes back since the 1940s (GOODFELLOW; COURVILLE, 2016).

There has been a major resurgence of deep learning mainly because computers today became faster. Training large deep learning models are computationally very expensive and could be sped up with the use of Graphics processing units (GPUs).

Artificial Neural Networks (ANNs) are one of the earliest learning algorithms. The novel concept behind it was to create mathematical models that tried to mimic the brain. ANNs were based on the human brain's neurological structure, more specifically neurons and how they connect with other neurons. Figure 3.1 illustrates a single neuron cell.

A neuron cell is capable of transmitting information by sending electrical and chemical signals through axons and they connect to other neurons forming neural networks. The human brain, for example, has billions of connected neurons. ANNs, therefore, try to be a simplified mathematical representation of neurons: basically a neuron can be viewed as a cell that receives one or more inputs and produces one or more outputs.

3.2 Neural Networks

3.2.1 Representation

In simple terms, a neural network is a mathematical structure that receives an input and calculates an output. We will describe a simple NN known as feedforward neural network or MLP.

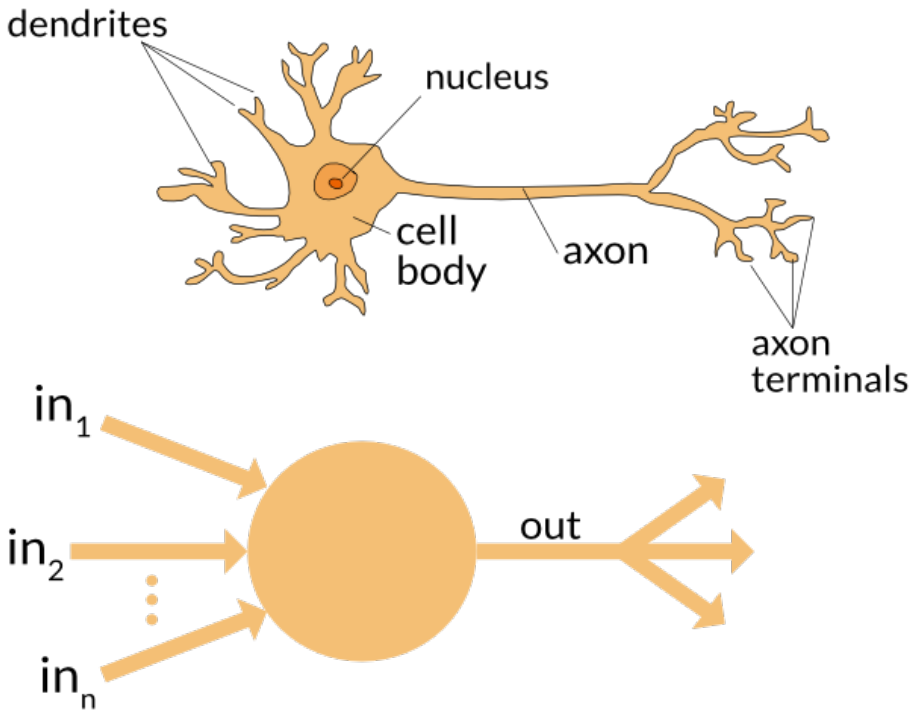


FIGURE 3.1 – Neuron unit cell structure.

A neural network is composed of multiple layers where each layer has one or more nodes (neurons). The NN receive an input and computes an output using its parameters θ according to it's architecture. Each input, called sample or example, is a n-dimensional feature vector x, known as the input layer, and each coordinate represents the sample's feature. The output is a k-dimensional vector y, known as the output layer. Every other layer that is not the input or output layer is known as a hidden layer.

Note that every time we refer to an element $z^{[l]}$ we are referring to an element of the l-th layer.

Figure 3.2 illustrates a very simple NN with a 3-dimensional feature vector, 1-dimensional output vector and only one hidden layer.

Every node of the network is computed using the values from the previous layer. For the MLP, the node computation model is done through a linear model $f(x;\theta)$, with θ consisting of w (weights) and b (bias). The model is defined as

$$z = w^T x + b \tag{3.1}$$

The output of the node, activation, is a function σ, called activation function, applied

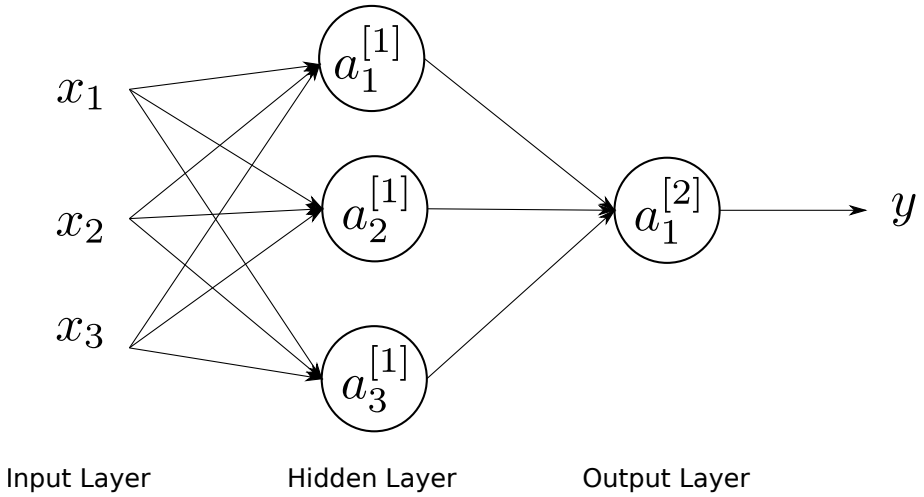


FIGURE 3.2 – Neuron unit cell structure.

to the model output (z). The activation is defined as

$$a = \sigma(z) \tag{3.2}$$

The activation function should be a non-linear function to the z and the default recommendation for the activation function today is the **rectified linear unit** (NAIR; HINTON, 2010) or ReLU and is defined as $\sigma(z) = max\{z, 0\}$ and shown in 3.3.

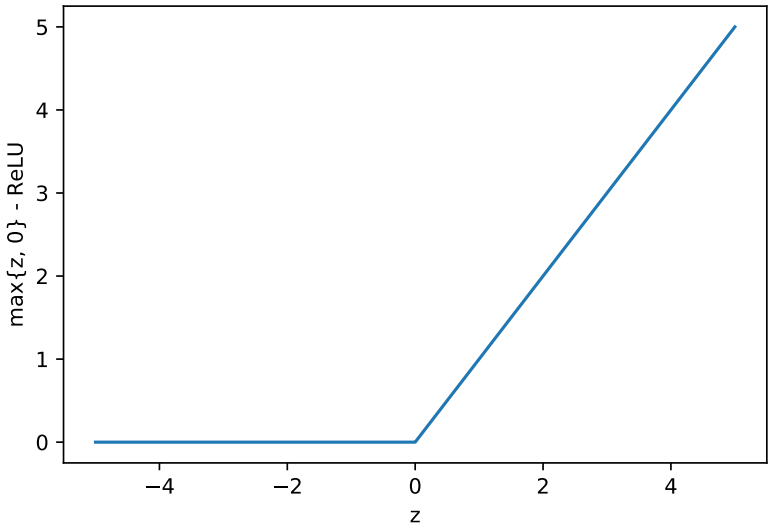


FIGURE 3.3 – Linear rectified unit plot.

Other examples of popular activation functions are $\sigma(z) = \dfrac{1}{1+e^{-z}}$, known as the sigmoid function and $tanh(z)$ (XU; LI, 2016).

Figure 3.4 illustrates the output of a neuron.

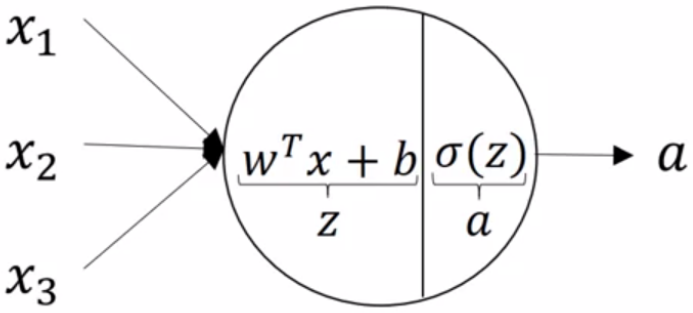


FIGURE 3.4 – Node computation.

For the network depicted in Figure 3.2, we are now capable of calculating the output y.

For the first layer:

$$z^{[1]} = w^{[1]T} x + b^{[1]} \tag{3.3}$$

$$a^{[1]} = \sigma(z^{[1]}) \tag{3.4}$$

And for the second layer:

$$z^{[2]} = w^{[2]T} a^{[1]} + b^{[2]} \tag{3.5}$$

$$y = a^{[2]} = \sigma(z^{[2]}) \tag{3.6}$$

We can think of the neural network as a recursive structure where each activation is calculated with the past's layers activation. $a^{[l]}$ can be computed as:

$$z^{[l]} = w^{[l]T} a^{[l-1]} + b^{[l]} \tag{3.7}$$

$$a^{[l]} = \sigma(z^{[l]}) \tag{3.8}$$

Note also that $a^{[1]} = x$.

3.2.2 Vectorization

In the previous section, we focused on computing the output of the neural network given a single sample x. In modern days, however, we are normally interested in calculating the output of a neural network for thousands or even millions of samples.

For m samples, we could naively compute the output of the neural network for each example $x^{(i)}$ with Algorithm 1.

Algorithm 1: Naive algorithm for computing NN output of m samples.

Result: Output of NN for m samples.

for *i = 1 to m* **do**

  $a^{[0]} = x^{(i)}$

  for *j = 1 to l* **do**

    $z^{[j](i)} = w^{T[j]} a^{[j-1](i)} + b^{[j]}$

    $a^{[j](i)} = \sigma(z^{[j]}(i))$

  end

  $y^{(i)} = a^{[l](i)}$

end

In practice this algorithm runs relatively slow since it computes the output of the network sequentially for each sample and we will apply vectorization to achieve a much faster algorithm.

Firstly, Vectorization is a technique that transforms a set of computations done sequentially in a for loop into matrix operations. We will now vectorize our initial problem.

Let us define the following matrices

- X is a matrix where column i is the i-th sample $x^{(i)}$ and $W^{[l]}$. We can analogously define matrices A and Z. $X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}$.
- $W^{[l]}$ is a matrix where the i-th row is $w^{T[l]}$. $W^{[l]} = \begin{bmatrix} w^{T[l]} \\ w^{T[l]} \\ \vdots \\ w^{T[l]} \end{bmatrix}$.
- $B^{[l]}$ is a vector where it's i-th coordinate is $b^{[l]}$. $B^{[l]} = \begin{bmatrix} b^{[l]} \\ b^{[l]} \\ \vdots \\ b^{[l]} \end{bmatrix}$.

For the vectorized version of Algorithm 2.

Algorithm 2: Vectorized algorithm for computing NN output of m samples.

Result: Output of NN for m samples.

for $j = 1$ *to* l **do**

$Z^{[j]} = W^{[j]}A^{[j-1]} + B^{[j]}$

$A^{[j]} = \sigma(Z^{[j]})$

end

The vectorized version is much better since these matrices operations can be greatly sped up on GPUs or hardware that have support for Streaming SIMD Extensions (SSE). The great majority of popular deep learning libraries use as much vectorization as they can.

3.3 Learning

As described in the previous Section, feedforward networks defines a mapping $y = f(x;\theta)$, or, equivalently, serve as general function approximators.

This section describes deep learning algorithms that learn the value of the parameters θ that best approximates f.

Most deep learning algorithms need to describe a cost function and optimization algorithm.

3.3.1 Cost Function

The cost function $J(\theta)$ is a metric of how good our estimator is to our dataset: the lower the cost function, the less error the model has when predicting y. It describes the function that we wish to minimize and normally it envolves an average of the errors between the the target value of a sample and the predicted value of the estimator for the sample.

For example, for the problem of linear regression where X are the sample values and y are the target values for our dataset, we wish to learn the parameters θ from our function approximator f. The most used cost function for this problem is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y_i - f(x_i,\theta))^2 \tag{3.9}$$

Another example is the logistic regression problem. The most common cost function

for this problem is

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log f(x_i, \theta + (1 - y_i) \log 1 - f(x_i, \theta)] \tag{3.10}$$

Choosing a good cost function greatly impacts how good the design of the deep neural network is.

3.3.2 Optimization Algorithms

After defining a cost function, finding the parameters that effectively minimize the cost function is not trivial. The non linearity of NN causes most cost functions to be non-convex and therefore, neural networks are usually trained by iterative, gradient-based optimizers. In practice, however, this is very unlikely (SWIRSZCZ *et al.*, 2017; GOODFELLOW *et al.*, 2014; LIN *et al.*, 2017).

Section 3.4 describes how neural networks gradients can be calculated.

3.3.2.1 Gradient Descent

One of the most common optimization algorithm is Gradient Descent. It is an iterative first-order method that tries to find a local minimum for a function f by moving in the negative direction of it's gradient. Specifically for neural networks, our goal is to minimize the cost function $J(\theta)$. The algorithm's update step is Equation (3.11) done for each θ_i

$$\theta_i = \theta_i - \alpha\frac{\partial J}{\partial \theta_i} \tag{3.11}$$

The learning rate, α measures how big the update step in the direction of the gradient will be.

For example, let $g : \mathbb{R}^2 \to \mathbb{R}$ given by $g(x, y) = y\sin(x) - x\cos(y)$. Figure 3.5 illustrates 5 steps of the gradient descent algorithm. Notice how the red circle is the starting point and after the last iteration, the algorithm arrives approximately at a local minimum.

3.3.2.2 Stocastic Gradient Descent

Another popular algorithm is Stocastic Gradient Descent (SGD) that is a stochastic approximation of Gradient Descent. Instead of calculating $J(\theta)$ as an average between all the samples, the gradient update step is done once per sample. It calculates an approximation of the gradient and updates the parameters θ_i with every sample. Algorithm 3

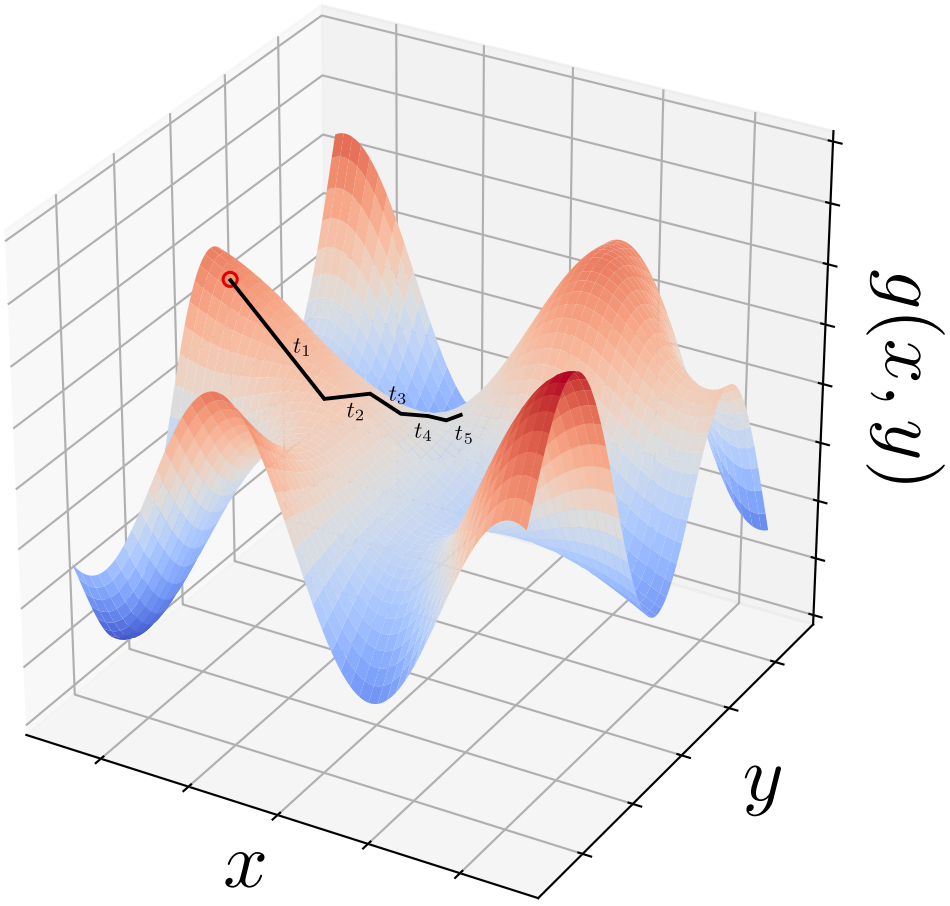


FIGURE 3.5 – Gradient descent on function g.

describes SGD

Algorithm 3: SGD algorithm.

Result: Function minimum

while *An approximate minimum is not obtained* **do**

  Randomly shuffle examples in the training set

  for $i = 1,2,\ldots, m$ **do**

    $\theta_i = \theta_i - \alpha \nabla J_i(\theta)$

  end

end

The gradient approximation for SGD is much faster to compute, however, since it is a noisy estimate of the gradient, SGD may need more steps to converge to a local minimum.

3.3.2.3 Mini-Batch Gradient Descent

Finally, Mini-Batch Gradient Descent combines the idea from Gradient Descent and SGD. It computes the gradient using more than one training example, called a *mini-batch*, at each step. It may result in smoother convergence and the code can be accelerated by

making use of vectorization.

3.4 Back Propagation

We have seen how a few gradient-based optimization algorithms work. In this Section we will briefly describe how to actually compute the gradients of a neural network.

Backpropagation is an algorithm that computes the gradients of the loss function w.r.t. each input from the network.

4 Deep Reinforcement Learning

In this chapter, we describe the reinforcement learning algorithms we use to learn the tasks.

4.1 Deep Deterministic Policy Gradients (DDPG)

DDPG (LILLICRAP *et al.*, 2015) is an off-policy actor-critic algorithm that simultaneously estimates the policy and the action-value function.

It has two components:

- Actor $\mu(s|\theta^{\mu})$: Policy Function approximation using deep neural networks.
- Critic $Q(s, a|\theta^{Q})$: Value Function approximation using deep neural networks.

It is an extension of the Deterministic Policy Gradient Algorithm (DPG) (SILVER *et al.*, 2014). DPG introduces a very important theorem for policy gradient methods that is:

$$\begin{aligned} \nabla_{\theta^{\mu}} \mu &\approx \mathbb{E}_{s_t \sim \mu'}[\nabla_{\theta^{\mu}} Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}] \\ &= \mathbb{E}_{s_t \sim \mu'}[\nabla_a Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_{\mu}} \mu(s|\theta^{\mu})|_{s=s^t}] \end{aligned} \tag{4.1}$$

Equation (4.1) basically shows that the policy's gradient should be updated in the direction of the estimated action-value function's gradient.

DDPG is based on Equation (4.1) but adapts 3 key ideas from (MNIH *et al.*, 2015) for the continuous domain. These ideas make it work very well in practice:

- Use deep-neural networks as non-linear estimators for the actor and the critic.
- Use a finite sized cache known as a replay buffer.
- Use a copy of the actor and critic as target networks. These are called target networks and have slower learning rate.

4.2 Trust Region Policy Optimization (TRPO)

TRPO is an iterative procedure for optimizing policies (SCHULMAN *et al.*, 2015). The algorithm alternates between sampling data by interacting with the environment and optimizing an alternate ("surrogate") objective function subject to a constraint on the size of the policy update. The policy is represented as a fully-connected neural network.

To understand TRPO, we must first understand KL divergence. KL divergence is a measure of how one probability distribution diverges from a second. It is defined as the relative entropy between two continuous random variables P, Q (and similarly for discrete probability distributions)

$$KL[P,Q] = \int_{-\infty}^{+\infty} p(x) log \frac{p(x)}{q(x)} dx$$

TRPO tries to optimize the following problem

$$\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{4.2}$$

$$\text{subject to}\ \hat{\mathbb{E}}_t[KL[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \tag{4.3}$$

Here $\hat{E}_t[...]$ is the empirical averaged over a finite batch of samples and $\hat{A}_t$ is an estimator for the advantage function at timestep t.

One of the novel ideas introduced by TRPO is limiting the step size of the policy (using the KL divergence) to guarantee policy improvement.

4.3 Proximal Policy Optimization (PPO)

PPO is very similar to TRPO but tries to optimize a different surrogate objective (SCHULMAN *et al.*, 2017).

Let $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Note that when there is no update to the policy, $r_t(\theta_{\text{old}}) = 1$.

The main surrogate objective for PPO is

$$L(\theta) = \hat{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right] \tag{4.4}$$

The policy is represented as a fully-connected MLP that outputs the mean of a Gaussian distribution.

This algorithm allows parallel implementations and is data efficient.

5 Methodology

5.1 Tools

This Section describes the tools that made this work possible.

5.1.1 C++

C++ is a general purpose programming language that is highly efficient and flexible. It is greatly focused on performance allowing low-level memory manipulation (STROUSTRUP, 2013).

The ITAndroids Soccer3D base code was already in C++ and therefore, anything that interacts with the soccer simulation server was also in C++ to integrate seamlessly with the codebase.

For this work, we use version 14 which is the most recent standardized (and stable) version.

5.1.2 Python

Python is an interpreted general purpose high-level programming language (PYTHON, 2017). It has an easy-to-read syntax closely resembling pseudocode therefore it allows faster development time which makes it easy to prototype new ideas.

For these reasons, the scientific community's usage of Python has greatly increased in recent years, even more so in the deep learning community.

For this work, we chose version 3.5 since it is the most recent and stable version when starting development.

5.1.3 Protocol Buffers

Protocol Buffers is a multiplatform library for serializing structured data that supports multiple programming languages (PROTOCOL..., 2017). It is very useful for developing applications that have many moving parts that communicate with each other.

For these reasons, we chose Protocol Buffer as the communication protocol for this work.

5.1.4 OpenAI Gym

OpenAI Gym is a toolkit for running reinforcement learning algorithms in different environments (BROCKMAN *et al.*, 2016). It allows testing and benchmarking RL algorithms by providing a standardized set of environments. This is very important for RL since makes it accessible for the research community to compare different algorithms (NADA, a).

Figure 5.1 illustrates a few example environments from OpenAI Gym.

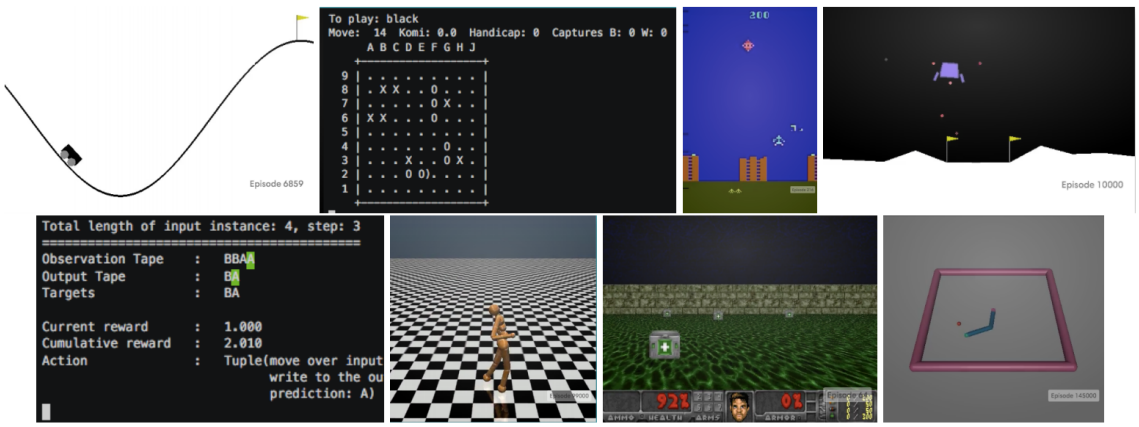


FIGURE 5.1 – OpenAI Gym environments.

Because of these reasons, OpenAI Gym was used in this work.

5.1.5 TensorFlow

TensorFlow is an open source software library for numerical computation using data flow graph (TENSORFLOW, 2017). It is greatly used for machine learning applications since it provides efficient implementation of gradient computations and optimization algorithms that can be parallelized by making use of specialized hardware instructions like SSE and

GPU. The library also exposes an API for multiple languages such as Python, C++, Go and Java.

TensorFlow was chosen because of its big developer community and its various tools available such as Tensorboard - visualization dashboard that is very useful for debugging.

5.2 Simulation Environment

The chosen simulation environment is the *SimSpark* simulator (SIMSPARK, 2004). It is a simulation system for multi-agent simulations and uses Open Dynamics Engine (ODE, 2004) for rigid body dynamics and collision detecting. The system allows two team of up to eleven humanoid robots to play against each other.

Unfortunately, the implementation of SimSpark does not guarantee that events are reproducible and adds noise to the problem. This means that different instances of the same simulation may have different outcomes and makes it harder to learn. This makes it important to run the RL algorithms multiple times.

The server also exposes a network interface that allows external processes to be notified of the simulation's state for purposes of visualization or logging.

The server also exposes an interface for querying the ground truth state of many state features that are not available during game play, e.g. the global position of the robot. The entity that reveals the ground truth is known as the Wizard in the ITAndroids code.

For this work, we use *RoboViz*, a publicly available visualization tool created by Justin Stoecker (STOECKER, 2011) that allows a few improvements over the default visualization tool.

Figure 5.2 shows a typical game scene between two teams inside *RoboViz*.

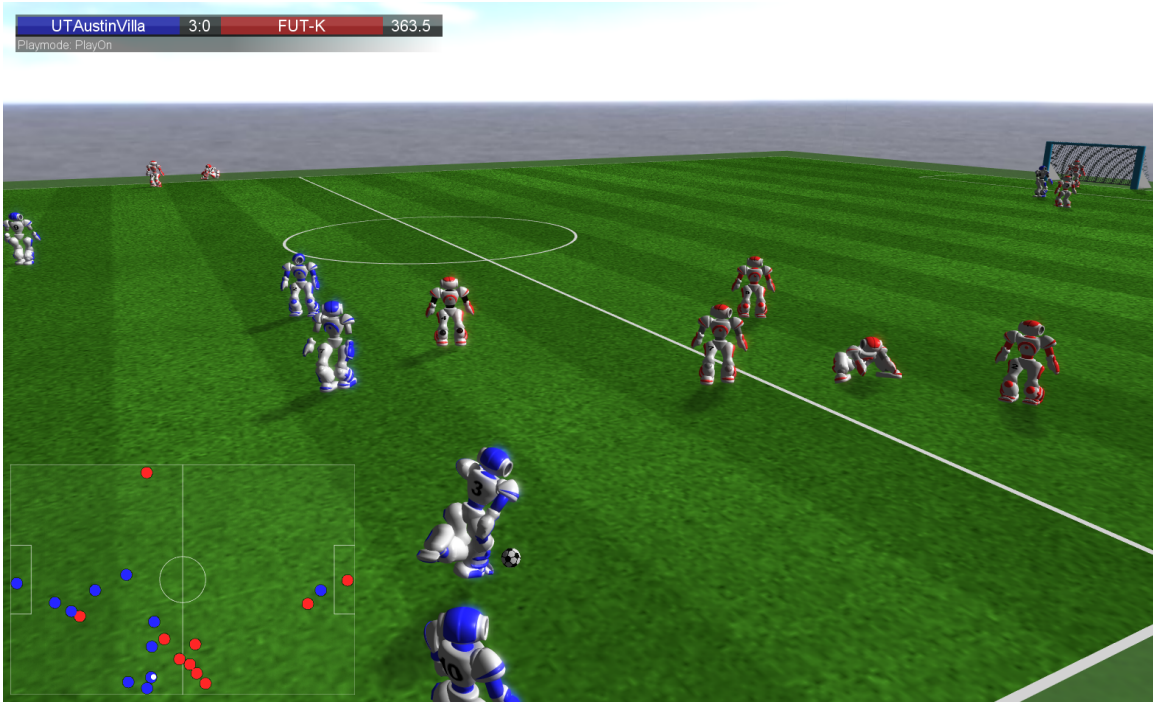


FIGURE 5.2 – *SimSpark* Simulation Environment running.

Agents communicate with the simulator via TCP connections. The agent sends the action to the server and the server executes a simulation step of $\Delta t = 0.016$s.

The main reason this simulation environment was chosen was that it is the official simulator used in RoboCup Soccer Simulation 3D competition since 2004.

5.3 Walking Engine

The walking engine is an algorithm that calculates how the humanoid robot should walk. As input, it receives a desired velocity $v = [v_x, x_y, x_\phi]^T$, where v_x, v_y and v_ϕ are speeds in the forward, lateral and rotational directions, respectively and calculates the robot's joint angles as output. These angles are then fed to PID controllers at each joint to compute the joint velocities actually sent to the server.

In this work, we are not interested in the walking engine per se, however it is important to understand some low-level aspects of the walking engine since it implies some design restrictions on how the robot walks and influences the learned behavior.

The walking engine we use is based on (MAXIMO, 2015; MAXIMO; RIBEIRO, 2016).

It is based on the Zero Moment Point (ZMP) concept and approximates the robot dynamics using the Linear Inverted Pendulum Model.

The walk cycle, shown in Figure 5.3 has constant duration T and the support foot

always alternates between the right and left, having two double support phases in each step

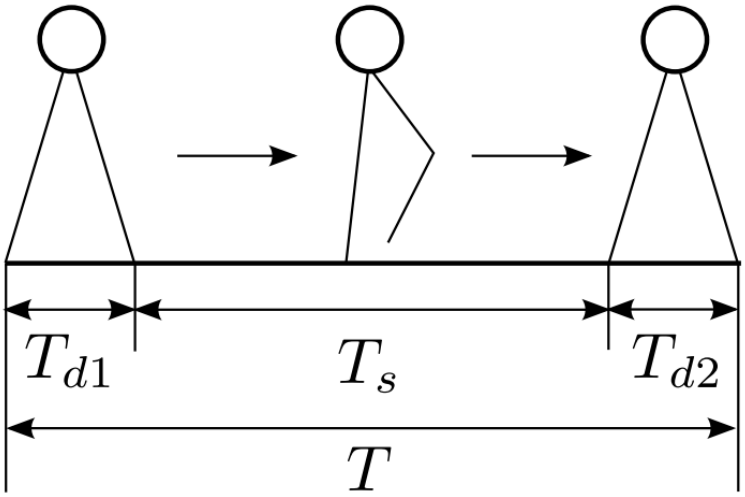


FIGURE 5.3 – Walk cycle (a step).

s

Also, for stability purposes, the acceleration is bounded, so it may take multiple walking steps to reach the desired speed.

5.4 Implementation

In the early stages of the project, we implemented a working version of DDPG that could successfully learn some classical continuous control problems from OpenAI Gym. One example of a toy domain it could learn appropriately was the inverted pendulum swingup problem that consists of trying to swing up a pendulum so it stays upright. This process was very time consuming because Deep RL algorithms are hard to validate.

However, since we would experiment with multiple algorithms, we decided to use readily available RL algorithms implementations. The implementation of PPO, TRPO and DDPG algorithms we used are from the OpenAI Baselines repository (DHARIWAL *et al.*, 2017). OpenAI Baselines is a repository from GitHub with a set of high-quality implementations of reinforcement learning algorithms. Not implementing all the DRL algorithms from scratch made it much easier to experiment with new algorithms.

The project is composed of two modules (each of them being a single process):

- **Learning Client:** Process that runs the reinforcement learning algorithms and makes remote procedure calls to the server exchanging state/action information

soccer agent. This module was implemented in Python 3.5 and TensorFlow using the OpenAI Baselines implementations.

- **Soccer Agent:** Process that interacts with SimSpark and that interacts with the learning client. This module was implemented in C++ and uses the ITAndroids Soccer Simulation 3D code base.

The API between the the server and the client is defined using Protocol Buffers.

A general diagram is depicted in Figure 5.4.

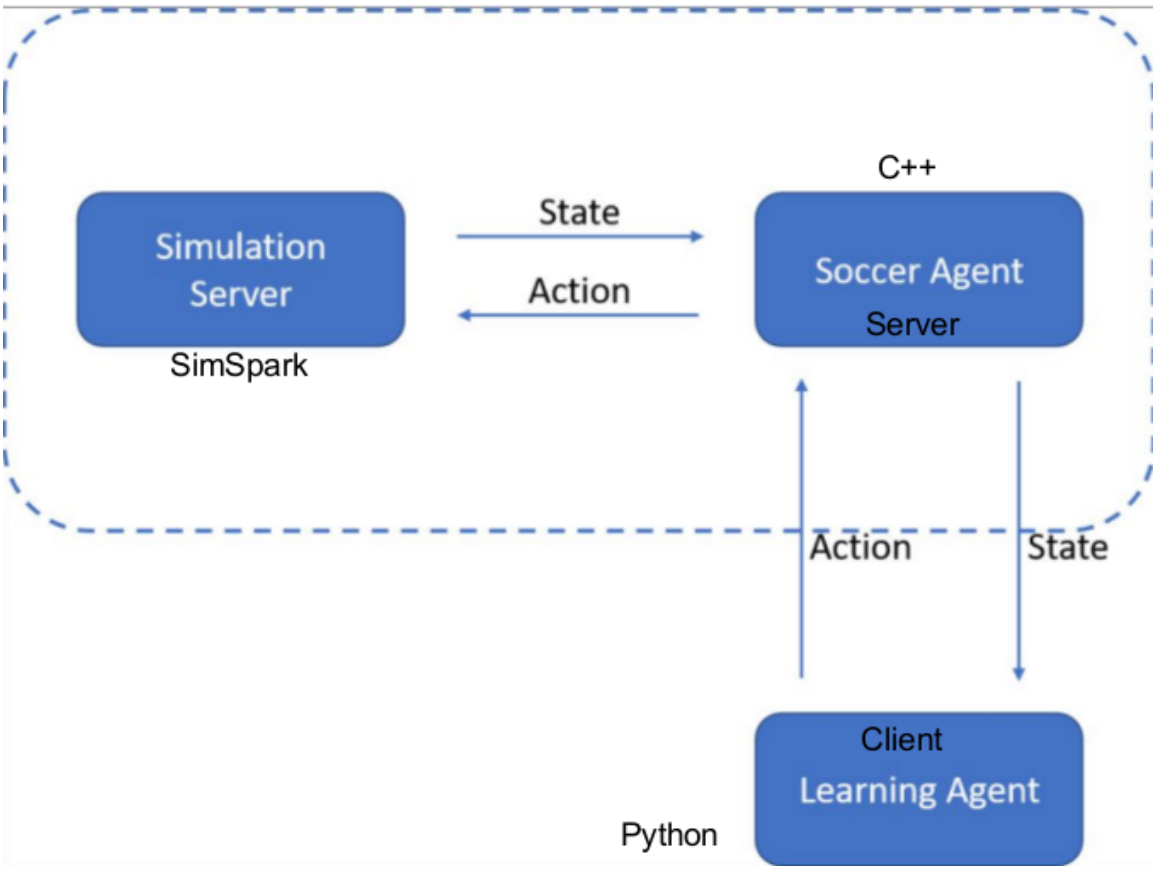


FIGURE 5.4 – Learning Architecture diagram.

This uncoupled design between the server and the client enabled us to easily change the learning algorithms in the client while not influencing the server. Similarly, this approach allowed us to switch tasks without influencing the client. This project lays the groundwork for the ITAndroids Soccer Simulation 3D team to use DRL for learning.

The source code for the client is available at:

`https://github.com/alexandremuzio/ddpg-humanoid`.

5.5 Tasks Specifications

The main task we are trying to solve is the humanoid racing problem. We also experiment on a simpler domain that we call warm-up task. In this Section we fully describe the two tasks: the state space, action space and the reward signal of each task.

Note that pose variables (x, y, z and θ) are retrieved from the server using the Wizard (ground truth) and each speed variable is calculated by

$$v_x = \frac{x_t - x_{t-1}}{\Delta t}$$

and analogously for v_y and v_θ. Here, v_x is the frontal speed at step t, x_t is the x coordinate at step t and x_{t-1} is the coordinate at step t. Since calculating speed by deriving position is very noisy, we use a low-pass filter to smooth out the signal.

5.5.1 Constant Speed Walker (Warm-Up) Task

This task is basically for the simulated robot agent to learn how to to walk forward with constant reference speed v_{ref}. It is a simple problem to get a better intuition of how the simulator works.

State space: The 1D observations consists only on the frontal robot velocity v_x.

Action space: The 1D action space consists of the commanded frontal speed u_x to be executed by the walk engine.

Reward signal: The reward is given by

$$r(s,a) = \mathbb{I}^{\text{close to ref}} - 100\mathbb{I}^{\text{robot fell}}$$

where $\mathbb{I}^{\text{close to ref}}$ equals 1 if $|v_{ref} - v_x| < \epsilon$ and 0 otherwise.

The episode terminates after $T = 2000$ steps or if the robot falls: $z_{CoM} < 0.2$

5.5.2 Racing Task

This is the task we are truly interested in. It describes a race track of 18x4m in which the robot must reach the finish line while remaining between the two lanes.

Figure 5.5 illustrates the task as shown in *RoboViz*. The simulated Nao robot starts the race on the center of the green line and tries to reach the red finish lines without

crossing the black border lines.

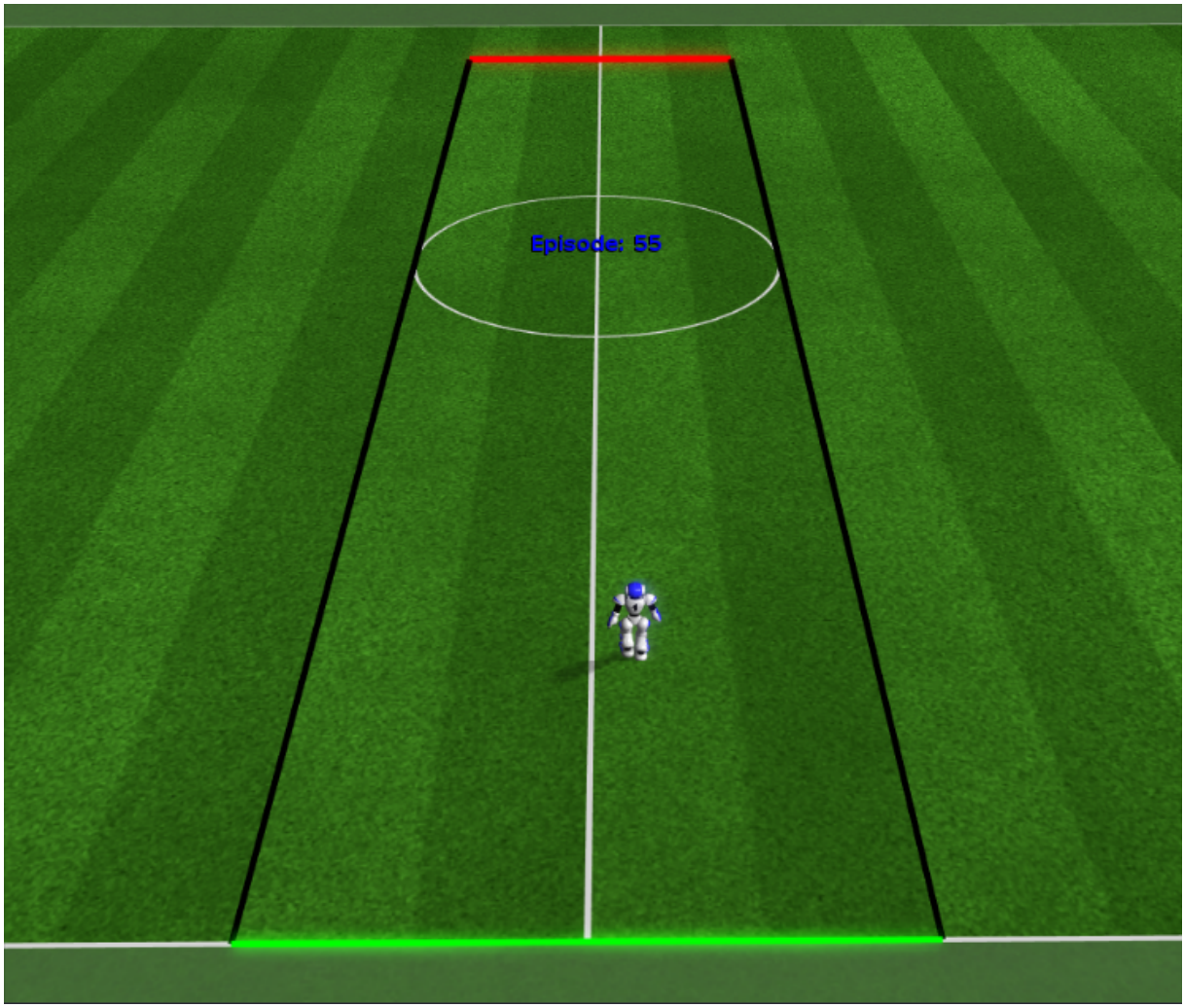


FIGURE 5.5 – Humanoid Racing Domain.

State space: The observations consists of the x, y, z coordinates of the torso, the yaw angle of the torso, the forward speed v_x, sideways speed v_y and rotational speed v_θ.

Action space: The 3D action space consists of u_x, u_y and u_θ, speed control signal in forward, sideways and rotational directions, respectively.

Reward signal: The reward is defined by

$$r(s,a) = v_x - 0.005(v_\theta^2 + v_y^2) - 0.05(u_x^2 + u_y^2 + u_\theta^2) - 0.05y^2 \\ +50\mathbb{I}^{\text{finish line}} - 10\mathbb{I}^{\text{leave track}} - 10\mathbb{I}^{\text{robot fell}}$$

The term v_x rewards the robot for moving in the direction of the race track and the terms $-0.05(v_\theta^2 + v_y^2)$ and $-0.05y^2$ penalize deviation from the forward direction. This was inspired by (DUAN *et al.*, 2016).

The episode duration is 2000 steps and the episode terminates if the robot falls: $z_{CoM} <$

0.2 or if the robot leaves the race track.

5.6 Metrics

When trying to solve some reinforcement learning task, it is very important to define a few metrics that effectively provides information during the training stage. With this information we try to understand how well the agent is doing in it's task.

In this work, we focus on 2 main metrics

- **Episode accumulated reward:** Maximizing the episode return is the goal of RL therefore it is the most obvious performance indicator and gives a very good sense of how well the agent's actions are doing.
- **Episode duration:** Episode reward is not enough to describe the agent's performance. For example, for the *Racing Domain*, if the episode is finishing early, it is a good indicator that the robot is falling or leaving the race track early.

We also logged all the state and action information from the agent for each episode on the server side. This was very helpful for reviewing past runs to extract non-obvious conclusions from the data.

6 Experiments and Results

Based on the task specifications from Chapter 5, we run a series of experiments to assess the learning results.

6.1 Constant Speed Walker (Warm-Up) Task

In this domain we are only interested in familiarizing with the environment. We ran PPO once for 10^6 timesteps with the parameters described in Appendix A. Figure 6.1 shows how the episode returns change over time. Notice how it significantly increases showing clear signs the agent is learning how to walk.

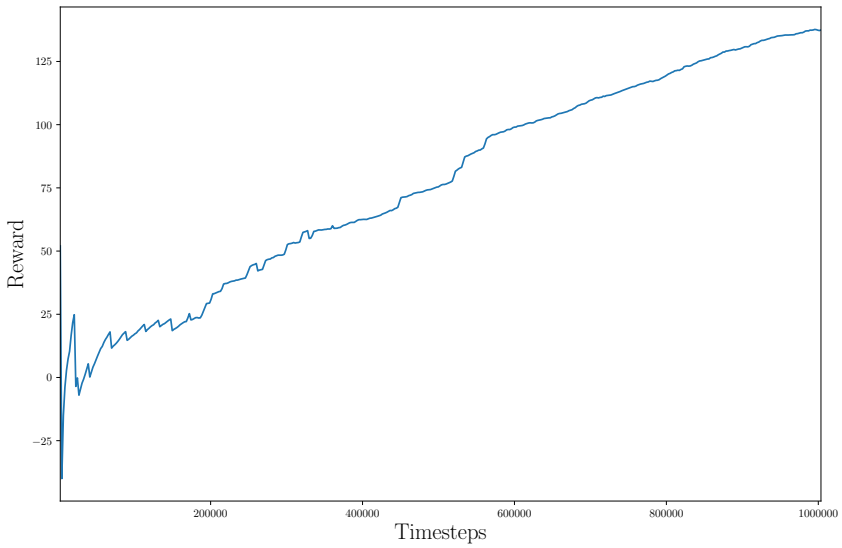


FIGURE 6.1 – Episode Reward for the Constant Speed Walker task trained with PPO.

Figure 6.2 shows how the episode duration changes during training. We see that the episode length also increases, which means that the agent is falling less.

Based on the results from Figure 6.1 and 6.2 we conclude that the agent is effectively learning.

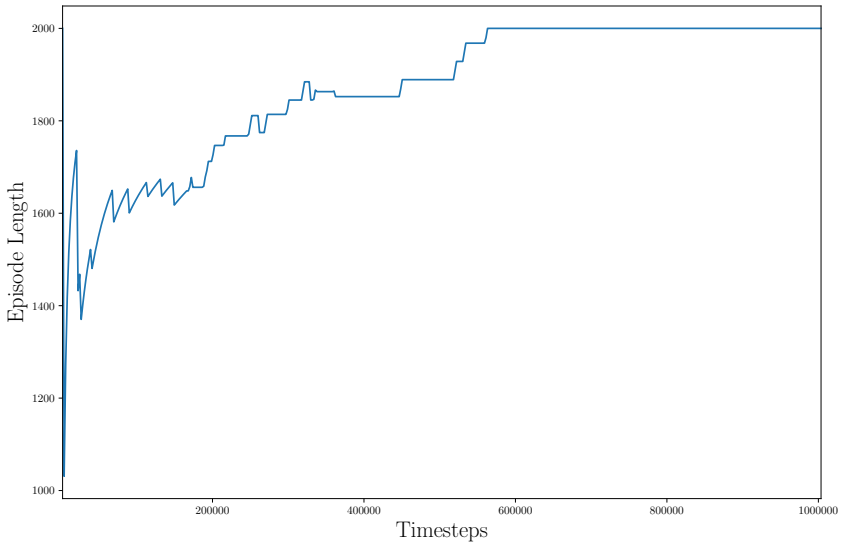


FIGURE 6.2 – Episode length for the Constant Speed Walker task with PPO.

6.2 Racing Task

In this domain, we run PPO, TRPO and DDPG during one million timesteps with the parameters described in Appendix A. We are interested in comparing the performance of these algorithms in this domain to see which algorithms work best for this task.

Figure 6.3 depicts the episode reward mean ± the standard deviation over five different runs.

FIGURE 6.3 – Reward for PPO, TRPO and DDPG in the Racer task

Similarly, Figure 6.4 illustrates episode reward length mean ± the standard deviation over the same five runs.

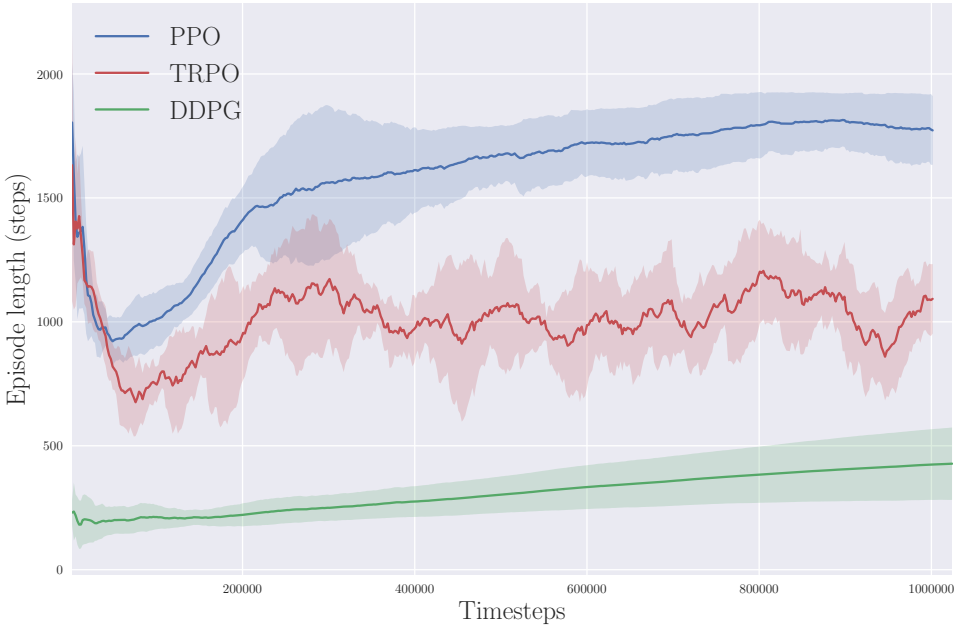


FIGURE 6.4 – Episode length for PPO, TRPO and DDPG in the Racer task.

It seems that performance of PPO is significantly better than DDPG an TRPO for this problem.

It is also interesting to note that in the first 200000 timesteps, the episode length for PPO and TRPO are below the episode length at the end of training. This could mean that, initially, the robot is constantly falling down or leaving the race track until it starts to realize that a better policy is to avoid falling and staying on track and starts taking longer for the episode to terminate.

Since PPO gave the best results for this problem, we analyse 20 trajectories that execute the policy learned by the agent. We compare these trajectories with the open loop "go-straight" controller. The results are show in Figure 6.5.

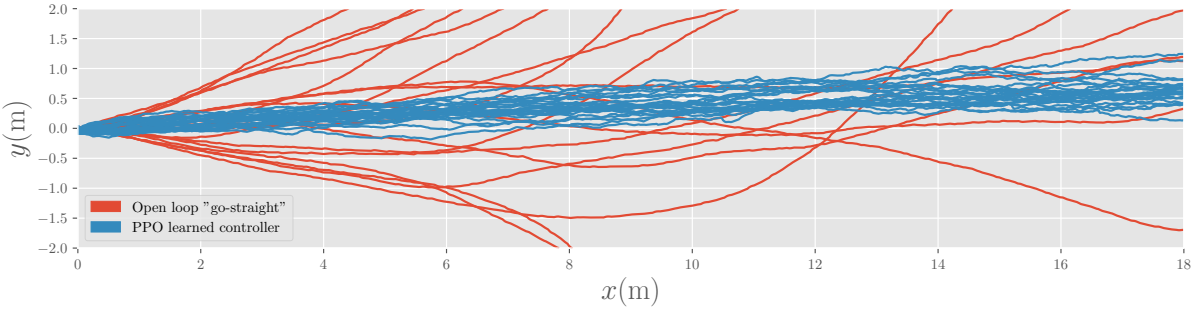


FIGURE 6.5 – Trajectories comparison between the PPO controller and the open loop "go-straight" controller.

We conclude that the learned policy effectively solves the racing domain.

For these trajectories, the empirical results: $v_{\text{mean}} \approx 0.59\text{m/s}$ and the $y_{\text{mean}} \approx 0.47$.

This represents a working but biased soccer agent behavior. This means that there is still room for improvement.

7 Conclusions and Future Work

7.1 Conclusions

The goal of this work was to learn a racing behavior for a simulated robot. We tackled this problem using model-free Deep Reinforcement Learning since it avoids dealing with the complex dynamics of a humanoid robot.

We use a Zero Moment Point (ZMP) walking engine that abstracts the need to deal with joint angles. The walking engine receives the desired speeds in forward, sideways and rotational directions and outputs the joint angles.

A client-server implementation allows us to use DRL algorithms in different tasks using SimSpark. This made it easy to switch between tasks and between learning algorithms.

We experimented with three different learning algorithms, PPO, TRPO and DDPG in the racing task.

Based on our training results, PPO performs the best being able to effectively learn a humanoid soccer policy. Despite working well, we point out that the learned policy had some bias towards the center of the race track and the speed average was not very high.

7.2 Future Work

This work could be extended by: enhancing the performance of the racing behavior or learning a more complex behavior in the simulation server.

To enhance the performance of the racing behavior we could:

- Incorporate more dimensions in the state space, for example, we could try using feet pressure to measure how stable the robot is.
- Try learning some parameters of the walk engine instead of treating them as hyperparameters. For example, step duration or maximum acceleration.
- Experiment further with the RL hyperparameters, e.g. the discount factor γ.

- Experiment with different DRL algorithms. For example Actor-Critic with Experience Replay (ACER) (WANG *et al.*, 2016).

Finally, we could try learning high level soccer behaviors like dribbling or stealing the ball. We could try something similar to what (HAUSKNECHT; STONE, 2016) did in a 2D environment. This would be a novel approach applied to humanoid soccer robots.

Bibliography

ZIYU Wang and Victor Bapst and Nicolas Heess and Volodymyr Mnih and Rémi Munos and Koray Kavukcuoglu and Nando de Freitas.

BARTO, A. G.; SUTTON, R. S.; ANDERSON, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. 1983.

BROCKMAN, G.; CHEUNG, V.; PETTERSSON, L.; SCHNEIDER, J.; SCHULMAN, J.; TANG, J.; ZAREMBA, W. **OpenAI Gym**. 2016.

DHARIWAL, P.; HESSE, C.; PLAPPERT, M.; RADFORD, A.; SCHULMAN, J.; SIDOR, S.; WU, Y. **OpenAI Baselines**. [S.l.]: GitHub, 2017. `https://github.com/openai/baselines`.

DUAN, Y.; CHEN, X.; HOUTHOOFT, R.; SCHULMAN, J.; ABBEEL, P. Benchmarking deep reinforcement learning for continuous control. **CoRR**, abs/1604.06778, 2016. Disponível em: <http://arxiv.org/abs/1604.06778>.

GABEL, T.; RIEDMILLER, M.; TROST, F. A case study on improving defense behavior in soccer simulation 2d: The neurohassle approach. In: _____. **RoboCup 2008: Robot Soccer World Cup XII**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 61–72. ISBN 978-3-642-02921-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02921-9_6>.

GATYS, L. A.; ECKER, A. S.; BETHGE, M. A neural algorithm of artistic style. **CoRR**, abs/1508.06576, 2015. Disponível em: <http://arxiv.org/abs/1508.06576>.

GOODFELLOW, I. J.; VINYALS, O.; SAXE, A. M. Qualitatively characterizing neural network optimization problems. **arXiv preprint arXiv:1412.6544**, 2014.

GOODFELLOW, Y. B. I.; COURVILLE, A. Deep learning. Book in preparation for MIT Press. 2016. Disponível em: <http://www.deeplearningbook.org>.

HAUSKNECHT, M.; STONE, P. Deep reinforcement learning in parameterized action space. In: **Proceedings of the International Conference on Learning Representations (ICLR)**. [S.l.: s.n.], 2016.

LEVNER, D. **Is Brute Force Backgammon Possible?** 1976. Disponível em: <www.bkgm.com/articles/Levner/BruteForceBackgammon/>.

LILLICRAP, T. P.; HUNT, J. J.; PRITZEL, A.; HEESS, N.; EREZ, T.; TASSA, Y.; SILVER, D.; WIERSTRA, D. Continuous control with deep reinforcement learning. **CoRR**, abs/1509.02971, 2015. Disponível em: <http://arxiv.org/abs/1509.02971>.

LIN, H. W.; TEGMARK, M.; ROLNICK, D. Why does deep and cheap learning work so well? **Journal of Statistical Physics**, Springer, v. 168, n. 6, p. 1223–1247, 2017.

LIN, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. 1992.

MAXIMO, M. R. O. A. **Omnidirectional ZMP-Based Walking for a Humanoid Robot**. Dissertação (Mestrado) — Instituto Tecnológico de Aeronáutica, 2015.

MAXIMO, M. R. O. A.; RIBEIRO, C. H. C. Zmp-based humanoid walking engine with arms movement and stabilization. **Congresso Brasileiro de Automáticas**, 2016. Disponível em: <https://ssl4799.websiteseguro.com/swge5/PROCEEDINGS/PDF-/CBA2016-0048.pdf>.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; HASSABIS, D. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved., v. 518, n. 7540, p. 529–533, Feb 2015. ISSN 0028-0836. Letter. Disponível em: <http://dx.doi.org/10.1038/nature14236>.

NAIR, V.; HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In: **Proceedings of the 27th International Conference on International Conference on Machine Learning**. USA: Omnipress, 2010. (ICML'10), p. 807–814. ISBN 978-1-60558-907-7. Disponível em: <http://dl.acm.org/citation.cfm?id=3104322.3104425>.

ODE. **Open Dynamics Engine (ODE)**. 2004. Disponível em: <http://www.ode.org/>. Acesso em: 29 outubro de 2017.

PROTOCOL Buffers. 2017. Disponível em: <https://developers.google.com/protocol-buffers/>. Acesso em: 06 novembro de 2017.

PYTHON. **What is Python? Executive Summary**. 2017. Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 06 novembro de 2017.

SCHULMAN, J.; LEVINE, S.; MORITZ, P.; JORDAN, M. I.; ABBEEL, P. Trust region policy optimization. **CoRR**, abs/1502.05477, 2015. Disponível em: <http://arxiv.org/abs/1502.05477>.

SCHULMAN, J.; WOLSKI, F.; DHARIWAL, P.; RADFORD, A.; KLIMOV, O. Proximal policy optimization algorithms. **CoRR**, abs/1707.06347, 2017. Disponível em: <http://arxiv.org/abs/1707.06347>.

SILVER, D.; LEVER, G.; HEESS, N.; DEGRIS, T.; WIERSTRA, D.; RIEDMILLER, M. Deterministic policy gradient algorithms. In: **Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32**. JMLR.org, 2014. (ICML'14), p. I–387–I–395. Disponível em: <http://dl.acm.org/citation.cfm?id=3044805.3044850>.

SIMSPARK. 2004. Disponível em: <http://simspark.sourceforge.net/wiki/index.php/Main_Page>. Acesso em: 30 outubro de 2017.

STOECKER, J. **RoboViz**. 2011. Disponível em: <https://github.com/magmaOffenburg/RoboViz>.

STROUSTRUP, B. **The C++ Programming Language**. 4th. ed. [S.l.]: Addison–Wesley, 2013.

SUTTON, R. S.; BARTO, A. G. **Introduction to Reinforcement Learning**. 1st. ed. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262193981.

SWIRSZCZ, G.; CZARNECKI, W. M.; PASCANU, R. Local minima in training of neural networks. **stat**, v. 1050, p. 17, 2017.

TENSORFLOW. 2017. Disponível em: <https://www.tensorflow.org>. Acesso em: 06 novembro de 2017.

TROMP, J.; FARNEBäCK, G. Combinatorics of go. 2016. Disponível em: <https://tromp.github.io/go/gostate.pdf>.

WANG, Z.; BAPST, V.; HEESS, N.; MNIH, V.; MUNOS, R.; KAVUKCUOGLU, K.; FREITAS, N. de. Sample efficient actor-critic with experience replay. **CoRR**, abs/1611.01224, 2016. Disponível em: <http://arxiv.org/abs/1611.01224>.

WATKINS, C. J. C. H. **Learning from Delayed Rewards**. Tese (Doutorado) — King's College, 1989.

WILIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. 1992.

XU, R. H. B.; LI, M. Revise saturated activation functions. 2016.

Appendix A - Experiment Parameters

Table A.1 shows the experimental parameters for the two tasks.

Hyperparameter	Value
Horizon (T)	2000
Discount (γ)	0.99
Num. steps	10^6

TABLE A.1 – Experiments parameters.

Table A.2 describes the hyperparameters used for DDPG in both domains.

Hyperparameter	Value
Batchsize	64
Actor learning rate	10^{-4}
Critic learning rate	10^{-3}
Adaptive param. noise	0.2

TABLE A.2 – DDPG hyperparameters used for both experiments.

Table A.3 describes the hyperparameters used for PPO for both domains. We use the hyperparameters from (SCHULMAN *et al.*, 2017).

Hyperparameter	Value
Adam stepsize	3×10^6
Num. epochs	10
Minibatch size	64
GAE parameter (λ)	0.95
Timesteps per batch	2048

TABLE A.3 – PPO hyperparameters used for both experiments.

Table A.4 describes the hyperparameters used for TRPO for both domains.

Hyperparameter	Value
Stepsize (D_{KL})	0.01
GAE parameter (λ)	0.98
Timesteps per batch	1024

TABLE A.4 – TRPO hyperparameters used for both experiments.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	21 de novembro de 2017	DCTA/ITA/TC-125/2017	61

5. TÍTULO E SUBTÍTULO:

Deep Reinforcement Learning Applied to Humanoid Robots

6. AUTOR(ES):

Alexandre Ferreira Velho Muzio

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – Divisão de Ciência da Computação – ITA/IEC

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Deep Reinforcement Learning; Deep Learning; Humanoid Robots; Robotics.

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Dinâmica de robôs; Robôs humanoides; Locomoção por pernas; Controle de robôs; Inteligência artificial; Robótica; Controle.

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Graduação. Programa de Graduação em Engenharia de Computação. Orientador: Prof. Dr. Takashi. Coorientadora: Profª. Msª. Marcos Ricardo Omena de Albuquerque Máximo. Defesa em 21/11/2017. Publicada em 25/11/2017.

11. RESUMO:

Humanoid robot soccer is a very traditional competitive task that aims to push the boundaries of state of the art robotics. One of the many challenges of playing soccer is walking and running while not losing balance. Modern Deep Reinforcement Learning (DRL), that combines Deep Learning algorithms to Reinforcement Learning (RL), has been frequently used to solve complex continuous control problems such as those in robotics. Model-free DRL is very effective for robot locomotion since it avoids directly dealing with the complex dynamics of a humanoid robot. In this work, we focus on the problem of humanoid racing. Using DRL Policy Gradients model-free algorithms (Proximal Policy Optimization, Trust Region Policy Optimization and Deep Deterministic Policy Gradients) we effectively develop a robust and stable Follow-Line controller that allows humanoid robots to race. Finally, the controller was evaluated on a simulated Nao robot from the RoboCup 3D Soccer Simulation League using classic performance metrics from Reinforcement Learning.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **CONFIDENCIAL** ( ) **SECRETO**