Dissertation presented to the Instituto Tecnológico de Aeronáutica, in partial fulfillment of the requirements for the degree of Master of Science in the Graduate Program of Electronics Engineering and Computing, Field of Informatics.

Luckeciano Carvalho Melo

IMITATION LEARNING AND META-LEARNING FOR OPTIMIZING HUMANOID ROBOT MOTIONS

Dissertation approved in its final version by signatories below:

Prof. Dr. Adilson Marques da Cunha
Advisor

Prof. Dr. Marcos R. O. de A. Máximo
Co-advisor

Prof. Dr. Pedro Teixeira Lacava
Pro-Rector of Graduate Courses

Campo Montenegro
São José dos Campos, SP - Brazil
2019

Cataloging-in Publication Data
Documentation and Information Division

Carvalho Melo, Luckeciano
Imitation Learning and Meta-Learning for Optimizing Humanoid Robot Motions / Luckeciano Carvalho Melo.
São José dos Campos, 2019.
193f.

Dissertation of Master of Science – Course of Electronics Engineering and Computing. Area of Informatics – Instituto Tecnológico de Aeronáutica, 2019. Advisor: Prof. Dr. Adilson Marques da Cunha. Co-advisor: Prof. Dr. Marcos R. O. de A. Máximo.

1. Deep Reinforcement Learning. 2. Robotics. 3. Artificial Intelligence. I. Instituto Tecnológico de Aeronáutica. II. Imitation Learning and Meta-Learning for Optimizing Humanoid Robot Motions

BIBLIOGRAPHIC REFERENCE

CARVALHO MELO, Luckeciano. **Imitation Learning and Meta-Learning for Optimizing Humanoid Robot Motions**. 2019. 193f. Dissertation of Master of Science – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Luckeciano Carvalho Melo
PUBLICATION TITLE: Imitation Learning and Meta-Learning for Optimizing Humanoid Robot Motions.
PUBLICATION KIND/YEAR: Dissertation / 2019

It is granted to Instituto Tecnológico de Aeronáutica permission to reproduce copies of this dissertation and to only loan or to sell copies for academic and scientific purposes. The author reserves other publication rights and no part of this dissertation can be reproduced without the authorization of the author.

Luckeciano Carvalho Melo
St. Vereador Edmundo G. Oliveira, 144
64.055-030 – Teresina–PI

IMITATION LEARNING AND META-LEARNING FOR OPTIMIZING HUMANOID ROBOT MOTIONS

Luckeciano Carvalho Melo

Thesis Committee Composition:

Prof. Dr.	Paulo Marcelo Tasinaffo	President	-	ITA
Prof. Dr.	Adilson Marques da Cunha	Advisor	-	ITA
Prof. Dr.	Marcos R. O. de A. Máximo	Co-advisor	-	ITA
Prof^{a}.Dr^{a}.	Esther Luna Colombini	External Member	-	UNICAMP
Prof. Dr.	Filipe Alves Neto Verri	Internal Member	-	ITA

ITA

To my mother, Fátima, whose courage and strength motivates me to always be better.

Acknowledgments

First, I would like to thank the support of my family: my mother, for the unconditional love and for the most beautiful heart I have ever seen; my father, for the discipline and perfectionism taught to me, essential for a work like this; my brother, companion at all hours, always listening to my animated monologues about this work and many others; my wife, Jade, for the patience and support during this one-year journey, in which I often had to sacrifice our moments.

Secondly, I thank my teachers during this long walk, at school and at college. Many of you motivated me and made me believe that this would be possible. Many of you have done so much more than the role of teaching, and on account of that I have created an immense respect for the profession. You are the foundation of society.

Thirdly, I would like to thank all my friends who have made this journey ever more fun. There were countless moments in which you rejoiced in my journey. In particular, my roommates, for the coexistence over the last few years. Many of the happiest memories of this time are due to you and our friendship will continue for the rest of our lives, regardless of any distance.

Fourthly, I thank the ITAndroids team for all the technical teachings and friendships I have formed. In fact, I assign to this team a large part of my training as an engineer and motivation to face complex problems. Our competitions, evenings and weekends of working are worth it and I will miss all of it.

Finally, I thank those who gave support for this work to happen: ITAndroids, for access to the code and proposition of the problem; Professor Marcos Máximo and Professor Adilson Marques da Cunha, for guidance; Intel, for providing the computational support to all the experiments and to make feasible the execution of this research.

"Draco Dormiens Nunquam Titillandus"
— Unknown Wizard

Resumo

O controle de diversos graus de liberdade para robôs humanoides é reconhecido como um dos problemas mais difíceis da robótica. Neste trabalho, primeiramente propõe-se uma estrutura de aprendizado que não apenas imita, mas também otimiza o movimento do robô humanóide de acordo com alguma tarefa usando Aprendizado por Reforço Profundo, sem fazer qualquer suposição sobre a implementação subjacente do movimento. Posteriormente, propõe-se e aplica-se um algoritmo de *Meta-Learning* chamado *Bottom-Up Meta-Policy Search*, que destila algumas políticas especialistas para criar uma meta-política capaz de generalizar para tarefas não vistas durante o treino. Por fim, analisa-se o movimento de caminhada, primeiro propondo e aplicando o algoritmo *Proximal Policy Natural Evolution Strategy* para otimização de parâmetros da *engine* da caminhada e, em seguida, utilizando a metodologia de imitação previamente utilizada para copiar esta *engine* para uma rede neural. Todos os experimentos foram conduzidos no domínio do *RoboCup 3D Soccer Simulation*, culminando no controle do chute com precisão a nível de passe, na melhoria dos parâmetros da caminhada em tarefas específicas e em uma *engine* completamente baseada em uma rede neural.

Abstract

The control of varying degrees of freedom for humanoid robots is recognized as one of the most difficult problems in robotics. In this work, we firstly propose a learning framework that not just imitates but also optimizes humanoid robot movements towards a task using Deep Reinforcement Learning, without making any assumption about the underlying implementation of the movement. Subsequently, we propose and apply a Meta-Learning algorithm called Bottom-Up Meta-Policy Search, which distills few expert policies to create a meta-policy capable of generalizing to unseen tasks during training. Finally, we analyze the walk motion, firstly proposing and applying the algorithm Proximal Policy Natural Evolution Strategy for parameter optimization of the walking engine and then using the imitation methodology previously described to copy such engine to a neural network. All experiments were conducted in the RoboCup 3D Soccer Simulation domain, culminating in kick motions with passing-level precision, in improvement of walk engine parameters on specific tasks, and in a completely neural network-based engine.

List of Figures

FIGURE 1.1 – AlphaStar: Mastering the Real-Time Strategy Game StarCraft II (VINYALS *et al.*, 2019). 26

FIGURE 1.2 – Locomotion of Agent via Deep Reinforcement Learning (HEESS *et al.*, 2017b). 26

FIGURE 1.3 – A Snapshot from the RoboCup 3D Soccer Simulation League. 27

FIGURE 1.4 – Flow chart illustrating the organization of this work. 31

FIGURE 2.1 – Illustration of an actual optimization run with covariance matrix adaptation on a simple two-dimensional problem (KHAN, 2018). . . 33

FIGURE 2.2 – Human level control through deep reinforcement learning in atari games (MNIH *et al.*, 2015). 34

FIGURE 3.1 – A keyframe based kick motion. 39

FIGURE 3.2 – The 3D Linear Inverted Pendulum Model (MAXIMO, 2015). 41

FIGURE 3.3 – The Walking Engine overview. 42

FIGURE 4.1 – An artificial neuron within a feed forward artificial neural network (TANIKIC; DESPOTOVIC, 2012). 43

FIGURE 4.2 – The illustration of gradient descent in a two variables optimization (MAGALHAES, 2017). 45

FIGURE 4.3 – A Recurrent Neural Network (OLAH, 2015). 47

FIGURE 4.4 – Inside a RNN cell (OLAH, 2015). 47

FIGURE 4.5 – Inside an LSTM cell (OLAH, 2015). 48

FIGURE 5.1 – Reinforcement Learning System. 51

FIGURE 5.2 – The agent–environment interaction in a Markov decision process (SUTTON; BARTO, 1998). 56

FIGURE 5.3 – Backup diagram for v_π. The white nodes are states, while the black ones are possible actions. Bellman equation average over the actions in a recursive way. (SUTTON; BARTO, 1998). 58

FIGURE 5.4 – Backup diagram in Bellman Optimality Equation for v_π and q_π. (SUTTON; BARTO, 1998). 60

FIGURE 5.5 – Policy Iteration algorithm. (SILVER, 2015b). 61

FIGURE 5.6 – Actor-Critic model (XU *et al.*, 2007). 68

FIGURE 6.1 – Illustration of MDP context for Meta-Learning. 77

FIGURE 6.2 – RL^2 interaction over MDPs (DUAN *et al.*, 2016). 77

FIGURE 6.3 – Architectural blocks that forms SNAIL (MISHRA *et al.*, 2017). 79

FIGURE 6.4 – Reptile optimization procedure tends to converge to a region that is closest to optimal solution for each task (NICHOL *et al.*, 2018). 83

FIGURE 6.5 – FOMAML vs Reptile: the latter has a more precise gradient because it uses high-order derivative information (LEE, 2018). 83

FIGURE 7.1 – In the hybrid model, we can ensure the starting point is near of the optimal solution (orange arrow); otherwise, the starting point can be bad and harder to optimize (red arrow). 87

FIGURE 7.2 – The architecture of a neural network designed to learn motions. . . 88

FIGURE 7.3 – Intuition behind input normalization (JORDAN, 2018) 90

FIGURE 7.4 – Architecture used by pure Reinforcement Learning models. 91

FIGURE 7.5 – Gaussian noise applied to action space to ensure better exploration in continuous environments. (PLAPPERT *et al.*, 2017) 92

FIGURE 7.6 – Task used to learning kick motion. 93

FIGURE 7.7 – Reinforcement Learning Server Architecture (MUZIO, 2017). 94

FIGURE 7.8 – Data flow graph for PPO algorithm. 96

FIGURE 7.9 – "Pi" data flow node from PPO. 96

FIGURE 7.10 –Policy data flow graph from Figure 7.4 97

FIGURE 7.11 –Policy data flow graph from Figure 7.2 97

FIGURE 7.12 –Master-workers architecture for data parallelism. 98

FIGURE 7.13 –Synchronous and Asynchornous Distributed Training (ABADI *et al.*, 2015). 99

FIGURE 7.14 –Monitoring training metrics with Tensorboard. 100

FIGURE 7.15 –Speedup Test 102

FIGURE 7.16 –Data collection in best scenario 103

FIGURE 7.17 –RNR Reward Curve by learning update 104

FIGURE 7.18 –RNR kick motion sequence. 105

FIGURE 7.19 –RRR Reward Curve by learning update 105

FIGURE 7.20 –RNR+RRR Reward Curves by learning update. We trained in two sessions. The blue curve shows the first session and the gray one the second session. 106

FIGURE 7.21 –RNR+RRR Motion Sequence. In this model, the agent learns to kick with the left leg, as in the reference. 106

FIGURE 7.22 –Tradeoff between RNR and RRR models: when parameter weight for RRR is much greater, it collapses to RRR model. 107

FIGURE 7.23 –RNR+RRR+RISD learning curve in two sessions of training. The blue curve was the first session and the red one was the second. . . . 108

FIGURE 7.24 –RNR+RRR+RISD motion sequence. 108

FIGURE 7.25 –First session of training for reinforcement supervised model. 109

FIGURE 7.26 –Motion sequence from reinforcement supervised learning agent. . . . 109

FIGURE 7.27 –Plots of mean squared error and mean absolute error, during training.111

FIGURE 7.28 –The kick motion. The first row of figures shows the original kick motion. The second row shows the learned kick motion. Both motions are visually indistinguishable. 112

FIGURE 7.29 –Joint values for comparing original and learned kicks. The neural network was able to fit the joint trajectories with small errors. . . . 113

FIGURE 7.30 –Joints positions, during a period of the walking motion for the original walk, and the learned walk and the joints positions effectively attained, during the learned walking motion. 114

FIGURE 7.31 –The walking motions comparison. Figure (a) shows our agent in its regular walk, Figure (b) shows the same agent mimicking UT Austin Villa walk, and Figure (c) shows the UT Austin Villa agent itself performing his own walking motion. 115

FIGURE 7.32 –HLM+RNR reward curve. 116

FIGURE 7.33 –Motion sequence of HLM+RNR model. 117

FIGURE 7.34 –The reward curve from the first session of HLM+RNR+RET model. 118

FIGURE 7.35 –HLM+RNR+RET motion sequence after first session of training. . . 118

FIGURE 7.36 –The reward curve from the second session of HLM+RNR+RET model.119

FIGURE 7.37 –HLM+RNR+RET motion sequence after second training session. . . 119

FIGURE 8.1 – Reward curve of baseline train. 125

FIGURE 8.2 – Baseline versus tanh Architecture model. 126

FIGURE 8.3 – Comparison between Imitation Learning models with different errors during RL training. 127

FIGURE 8.4 – Evaluation of Critic Network's weights initialization, in terms of Value Function prediction error (Figure 8.4a) and reward curve (Figure 8.4b). 128

FIGURE 8.5 – Evaluation of reward scaling. 130

FIGURE 8.6 – Value Function prediction error curve when scaling reward using 0.01 factor. 131

FIGURE 8.7 – Evaluation of entropy term. 132

FIGURE 8.8 – Hyperparameter random search results. 133

FIGURE 8.9 – Evaluation of best hyperparameters sets and comparison with baseline.134

FIGURE 8.10 –Reward curve when conducting a second RL training phase with exploration restarter. 136

FIGURE 8.11 –Training a LSTM network for kick benchmark. 138

FIGURE 8.12 –Reward curve when conducting a second RL training phase with exploration restarter. 140

FIGURE 8.13 –Kick distribution from baseline model. 141

FIGURE 8.14 –Kick distribution from final model. 142

FIGURE 8.15 –Bottom-Up Meta-Policy Search. 145

FIGURE 8.16 –Reptile meta-gradient when outer learning rate is 1. 146

FIGURE 8.17 –BUMPS and Reptile Meta-Training. 148

FIGURE 8.18 –Smoothed error curves (and bootstrap confidence intervals) of RL optimization of 8 m, 10 m, and 12 m, respectively. The rewards modeled by gaussian and linear transformations results in approximately same performance. 150

FIGURE 8.19 –Reward Curve for Precise Kick of 12m. 151

FIGURE 8.20 –Distribution of 12m kick. 152

FIGURE 8.21 –Distribution of 12m kick after a second training session. 152

FIGURE 8.22 –Error curves from the second generation of precise kicks. 153

FIGURE 8.23 –Error statistics for each meta-training task. 154

FIGURE 8.24 –Distribution of 12m kick in second generation. 155

FIGURE 8.25 –Reward curve during direct RL training of meta-policy. 157

FIGURE 9.1 – Illustration of high level behaviors and their relationship. 162

FIGURE 9.2 – PP-NES Network Architecture. 164

FIGURE 9.3 – Forward walk evaluation for baseline and sprint parameters, respectively. 167

FIGURE 9.4 – Conduct walk evaluation for baseline (Figure 9.4a) and sprint parameters (Figure 9.4b). 168

FIGURE 9.5 – Fitness curve in experiments conducted for Sprint Task optimization, where we see catastrophic steps. In the case of Figure 9.5a, the agent was not even able to recover initial fitness value. 170

FIGURE 9.6 – Fitness curves of experiments conduct using the same hyperparameters to Conduct and Sprint Tasks, respectively. Despite the difference in confidence intervals, the optimize performed well in both cases. 171

FIGURE 9.7 – LSTM network for neural walk engine. The ten previous timesteps are used to compute the next joints. 172

FIGURE 9.8 – Evaluation of neural walk engine in sprint and conduct tasks, respectively. 173

List of Tables

TABLE 7.1 – The Network Summary. 89
TABLE 7.2 – The Reinforcement Learning Network Summary. 91
TABLE 7.3 – The Kick Comparison 113
TABLE 7.4 – Walk Comparison - Forward Walk 115
TABLE 7.5 – Kick Comparison - General Evaluation 120
TABLE 7.6 – Kick Comparison - Effective Evaluation 120
TABLE 7.7 – Kick Evaluation - Nao with Toe 121

TABLE 8.1 – Kick Benchmark Comparison - Effective Evaluation 141
TABLE 8.2 – Meta-Policy Generalization Performance 157
TABLE 8.3 – Final results for BUMPS algorithm 158

TABLE 9.1 – Seed Parameters - Walk System 166
TABLE 9.2 – Final Parameters - Sprint Task 167
TABLE 9.3 – Final Parameters - Conduct Task 169
TABLE 9.4 – Neural Walk Engine Network Summary. 172

TABLE A.1 – Hyperparameters - RNR, RRR, RNR+RRR, RNR+RRR+RISD. . . 192
TABLE A.2 – Hyperparameters - Reinforcement “supervised” - First Session. . . . 192
TABLE A.3 – Hyperparameters - HLM+RNR, HLM+RNR+RET - First Session. 193
TABLE A.4 – Hyperparameters from Kick Benchmark after random search. 193
TABLE A.5 – Hyperparameters - Precise Kick. 193

List of Abbreviations and Acronyms

ACER	Actor-Critic with Experience Replay
A2C	Advantage-Actor-Critic
API	Application Programming Interface
BUMPS	Bottom-Up Meta-Policy Search
CoM	Center of Mass
CPU	Central Processor Unit
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
CEM	Cross-Entropy Method
DDPG	Deep Deterministic Policy Gradient
DAG	Directed Acyclic Graph
DPPO	Distributed Proximal Policy Optimization
DP	Dynamic Programming
ES	Evolution Strategies
FOMAML	First-Order Model-Agnostic Meta-Learning
GAE	Generalized Advantage Estimation
GAIL	Generative Adversarial Imitation Learning
GUI	Graphical User Interface
GPU	Graphic Processor Unit
HC	Hill Climbing
HER	Hindsight Experience Replay
HLM	Hybrid Learning Model
HTTP	Hypertext Transfer Protocol
IK	Inverse Kinematics
KL	Kullback-Leibler
LSTM	Long Short-Term Memory
MAML	Model-Agnostic Meta-Learning
MDP	Markov Decision Process
MP	Markov Process
MRP	Markov Reward Process
ML	Machine Learning

MuJoCo	Multi-Joint dynamics with Contact
NAG	Nesterov's Accelerated Gradient
NES	Natural Evolution Strategies
ODE	Open Dynamics Engine
POMDP	Partially Observable Markov Decision Process
PP-NES	Proximal Policy Natural Evolution Strategy
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RISD	Reinforcement with Initial State Distribution
RET	Reinforcement with Early Termination
RNN	Recurrent Neural Network
RNR	Reinforcement with Naive Reward
RRR	Reinforcement with Reference Reward
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
SGD	Stochastic Gradient Descent
SNAIL	Simple Neural Attentive Meta-Learner
TPU	Tensor Processor Unit
TCP	Transmission Control Protocol
TRPO	Trusted Region Policy Optimization
ZMP	Zero Moment Point

List of Symbols

Sets

K	Keyframe set
$\mathbb{R}$	The set of real numbers
$\mathbb{A}$	A set

Numbers and Arrays

$\boldsymbol{x}$	A vector $\boldsymbol{x}$
A	Matrix A
$\boldsymbol{I}$	Identity matrix
$\boldsymbol{X}$	Tensor $\boldsymbol{X}$

Linear Algebra Operations

$\boldsymbol{A}^T$	Transpose of matrix $\boldsymbol{A}$
$\|\boldsymbol{x}\|$	L^2 norm of $\boldsymbol{x}$
$\boldsymbol{a} \odot \boldsymbol{b}$	Element-wise multiplication of vectors a and b

Calculus

$\frac{dy}{dx}$	Derivative of y w.r.t x
$\frac{\partial J}{\partial x}$	Partial derivative of J w.r.t x
$\nabla_{\boldsymbol{X}} z$	Gradient of z w.r.t $\boldsymbol{X}$
$\lim_{k \to b} a$	Limit of a when k tends to b

Probabily and Information Theory

$a \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathbf{x} \sim P} f(x)$	Expectation of $f(x)$ with respect to $P(x)$
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\sigma})$	Gaussian distribution over $\boldsymbol{x}$ with parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$
$\sigma^2(W)$	Variance of W
$\mathbb{P}(X = x \| Y = y)$	Probability of $X = x$ given that $Y = y$
$KL[P, Q]$	Kullback-Leibler divergence of P and Q
$\hat{x}$	Prediction of function x

$A^{(i)}$	Estimator A of ith order

Functions

$\mathcal{C}$	Class of a function
f	A function f
$f(\boldsymbol{x};\boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parameterized by $\boldsymbol{\theta}$
$g(f(\boldsymbol{x}))$	Composition of functions g and f
$f : A \rightarrow B$	Function f defined from set A to set B
$tanh(z)$	Hyperbolic tangent function applied to z
$max(a, b)$	Max function applied to element a and b
$min(a, b)$	Min function applied to element a and b

Deep Learning

$\sigma(\boldsymbol{x})$	Sigmoid function applied do vector $\boldsymbol{x}$
$a_j^{[i]}$	j^{th} neuron from i^{th} layer in a neural network
$W^{[i]}$	Neural network parameters from i^{th} layer
$b^{[i]}$	Bias parameters from i^{th} layer
$\hat{y}$	Prediction of a neural network
$J(\boldsymbol{\theta})$	Cost function of $\boldsymbol{\theta}$
W^*, b^*	Optimal values for parameter weights W and b
$\mathcal{L}(\hat{y}, y)$	Loss function between prediction $\hat{y}$ and y
α	Learning rate hyperparameter
β	Decay rate hyperparameter
ϵ	Fuzz factor
v_{dW}, v_{db}	First order momentum
u_{dW}, u_{db}	Second order momentum
L	Lipschitz constant

Reinforcement Learning

a	Action a
s	State s
R	Reward R
H	Horizon H
γ	Discount factor
π	Policy π
v_π	Value-function of policy π
$\mathcal{P}$	Transition matrix
$\mathcal{A}$	Set of actions
$\mathcal{S}$	Set of states

$\mathcal{R}$	Reward function
G	Return function
q_π	Action-value function of policy π
A_π	Advantage function of policy π
δ^t	Temporal-Difference error
λ	GAE factor

Contents

1 Introduction 25

1.1 Motivation 25

1.2 Contextualization 26

1.3 Objective and Hypothesis 27

1.3.1 Imitation Hypothesis 28

1.3.2 Meta-Learning Hypothesis 28

1.3.3 Walk Analysis 28

1.4 Scope 29

1.5 Contributions of this work 29

1.6 Organization of this work 29

2 Literature Review 32

2.1 Optimization Techniques 32

2.2 Reinforcement Learning for Control 33

2.3 Imitation Learning 35

2.4 Meta-Learning 36

3 RoboCup Soccer 3D Simulation 38

3.1 Domain Description 38

3.2 Kick Motion 38

3.3 Keyframe Movements 39

3.4 Walk Engine 40

4 Deep Learning 43

4.1 Neural Networks 43
4.2 Maximum Likelihood Estimation 44
4.3 Gradient Descent and Backpropagation 45
4.4 Recurrent Neural Networks 47
4.4.1 Long Short-Term Memory 48

5 Reinforcement Learning 49
5.1 Concepts of a Reinforcement Learning System 49
5.2 Reinforcement Learning System 50
5.2.1 Reward 51
5.2.2 State 51
5.2.3 Policy 52
5.2.4 Value Function 52
5.2.5 Model 53
5.3 Markov Decision Process 53
5.3.1 Markov State 54
5.3.2 State Transition Matrix 54
5.3.3 Markov Decision Process 54
5.3.4 State-Value and Action-Value Function 57
5.4 Bellman Equation 57
5.4.1 Bellman Expectation Equation 57
5.4.2 Bellman Optimality Equation 59
5.5 Exact Solution Methods 61
5.5.1 Policy Iteration 61
5.5.2 Value Iteration 63
5.5.3 Limitations of Exact Solution Methods 64
5.6 Policy Gradient Methods 65
5.6.1 Policy Gradient Theorem 66
5.6.2 Actor-Critic Models 67
5.6.3 Advantage Function and GAE Algorithm 69

5.7 Advanced Policy Gradient Methods 70
5.7.1 Optimization Loss 70
5.7.2 Trust Region Policy Optimization – TRPO 71
5.7.3 Proximal Policy Optimization – PPO 72

6 Meta-Learning 75
6.1 Meta-Learning Formulation 76
6.2 Meta-Learning in Architecture Level 76
6.2.1 RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning . 76
6.2.2 Simple Neural Attentive Meta-Learner (SNAIL) 78
6.3 Meta-Learning in Algorithmic Level 79
6.3.1 MAML: Model-Agnostic Meta-Learning 79
6.3.2 First Order Meta-Learning 81

7 The Imitation Hypothesis 84
7.1 The Kick Motion Problem 84
7.2 Experimentation Setup 85
7.2.1 Reinforcement with Naive Reward - RNR 85
7.2.2 Reinforcement with Reference Reward - RRR 85
7.2.3 Reinforcement with Initial State Distribution - RISD 86
7.2.4 Reinforcement with Early Termination - RET 86
7.2.5 Hybrid Learning Model – HLM 86
7.3 Supervised Learning Setup 87
7.3.1 The Dataset 87
7.3.2 Neural Network Architecture and Hyperparameters 88
7.3.3 The Training Procedure 89
7.3.4 The Deployment in the Soccer 3D Environment 89
7.4 Reinforcement Learning Setup 90
7.4.1 Policy Representation 90
7.4.2 Task Description 91
7.5 Infrastructure 93

7.5.1 Reinforcement Learning Server . . . 93
7.5.2 Neural Network Deployment . . . 95
7.5.3 Distributed Training . . . 97
7.5.4 Metrics . . . 99
7.5.5 Monitoring via Tensorboard . . . 100
7.6 Experimentations - Imitation Learning Hypothesis . . . 100
7.7 Distributed Training Results . . . 102
7.8 Pure Reinforcement Learning Methods Results . . . 104
7.8.1 RNR . . . 104
7.8.2 RRR . . . 105
7.8.3 RNR+RRR . . . 106
7.8.4 RNR+RRR+RISD . . . 107
7.8.5 Reinforcement "Supervised" . . . 108
7.8.6 Other ideas for pure RL techniques . . . 110
7.9 Hybrid Learning Models Results . . . 110
7.9.1 Supervised Learning Results . . . 111
7.9.2 HLM+RNR . . . 116
7.9.3 Final Model: HLM+RNR+RET . . . 117
7.9.4 Kick Behavior: Numerical Results . . . 119

8 The Meta-Learning Hypothesis . . . 122
8.1 Kick Benchmark . . . 122
8.1.1 Baseline Training . . . 124
8.1.2 Analysis #1: Neural Network Architecture . . . 125
8.1.3 Analysis #2: Imitation Learning Model Error . . . 126
8.1.4 Analysis #3: Critic Network's Weights Initialization . . . 127
8.1.5 Analysis #4: Reward Scale . . . 127
8.1.6 Analysis #5: Entropy Term . . . 129
8.1.7 Hyperparameter Search . . . 129
8.1.8 Exploration Restarter . . . 135

8.1.9 LSTM Model 137
8.1.10 Final Model 139
8.2 Bottom-Up Meta-Policy Search (BUMPS) 142
8.2.1 Explaining BUMPS: Meta-Gradient through Contexts 146
8.2.2 Training Single-Task Expert Policies 149
8.2.3 Training Meta-Policy and Policy Filtering 154
8.2.4 Results 156

9 Walk Analysis 160
9.1 Parameter Optimization for Engine-Based Walk 160
9.1.1 Walk System Description 160
9.1.2 Combining Evolution Strategies and Reinforcement Learning: Proximal Policy Natural Evolution Strategy (PP-NES) 163
9.1.3 Walk Optimization via PP-NES 165
9.2 Neural Walk Engine 171
9.2.1 Walk Behavior Cloning 172

10 Conclusions, Recommendations, and Future Works . . 175
10.1 Conclusions 175
10.2 Future Work 178

Bibliography 180

Appendix A – Hyperparameters 192

1 Introduction

In this section, we present the motivation of this work and describe the context in which it is inserted. Additionally, we define the objective, scope, and the organization of this work.

1.1 Motivation

Robotics is understood as an important area of research within Engineering and Computer Science, optimizing and automating various areas of industry. One of the ways in which research is developed in this area is in robot soccer, since it comprises challenges of machine perception, environment modeling, planning and reasoning, control, and multi-agent strategy.

Over the years, several techniques have been developed to address each of the problems related to robot soccer, based on the theory of Signal Processing, Control, Trajectory Planning, and classical Artificial Intelligence. These techniques have proved to be functional for maturing the challenge. However, such techniques still perform worse as compared to humans in these activities.

In recent years, however, with the development of processing and memory architectures, Machine Learning techniques have been able to achieve or even have surpass the human performance in machine perception activities (Computer Vision (LU; TANG, 2014) and Speech Recognition (XIONG *et al.*, 2016)), by planning and reasoning (SILVER *et al.*, 2017a), and by developing multi-agent strategy (OPENAI, 2018) and real-time strategy (VINYALS *et al.*, 2019) (as shown in Figure 1.1). It also achieved surprisingly results in controlling agent locomotion in simulation (HEESS *et al.*, 2017b) (as shown in Figure 1.2), or in real world (XIE *et al.*, 2019). In this way, the learning field of combining techniques from Deep Learning and Reinforcement Learning appears as a great candidate in the search of General Artificial Intelligence.

FIGURE 1.1 – AlphaStar: Mastering the Real-Time Strategy Game StarCraft II (VINYALS *et al.*, 2019).

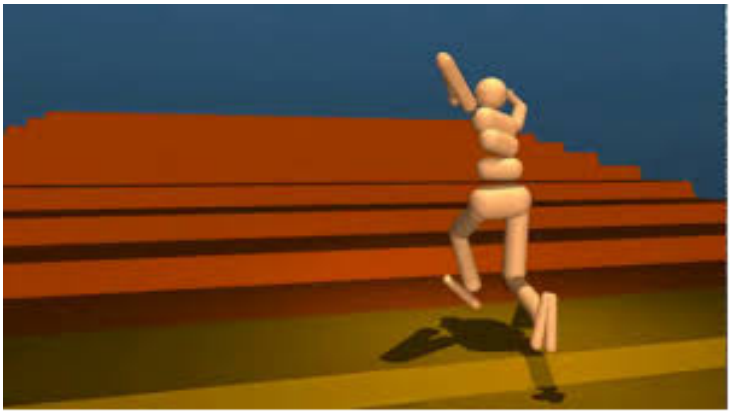


FIGURE 1.2 – Locomotion of Agent via Deep Reinforcement Learning (HEESS *et al.*, 2017b).

1.2 Contextualization

RoboCup is an international scientific community aiming to advance the state of the art of intelligent robots. Its mission is that a team of robots will be able to beat the human team champion of the World Cup until the year 2050 (KITANO *et al.*, 1998). Since this is a goal with a range of different challenges to be solved, there are several categories of competition within the community.

In RoboCup 3D Soccer Simulation League, as shown in Figure 1.3, there is a robot soccer competition in which each team has eleven Nao robots in a physical simulation environment called Simspark (OBST; ROLLMAN, 2005). The purpose in this case is to create and improve algorithms and physical models for the perception, locomotion, and strategy of the humanoid robot, before actually shipping them into hardware for real-world evaluation.

Over the last few years, team efforts have been seen in three different areas, layers, or strands. The first area is considered more basic, meaning the construction of the agent

FIGURE 1.3 – A Snapshot from the RoboCup 3D Soccer Simulation League.

that can model the environment and interact with it – involving both the construction of a solid software architecture and the application of traditional techniques of localization and control of the humanoid robot (MACALPINE *et al.*, 2011).

The second layer, explored at the highest level, involves creating behaviors to perform actions within the game, given the modeled environment – from the creation of models and heuristics for navigation to the creation of multi-agent strategies for positioning and marking (MACALPINE; STONE, 2016).

The third strand, to be approached in this work, is the creation and optimization of models based upon learning for activities such as robot walking and kicking. Simulated categories have great value for learning test, mainly because they provide a benchmark for comparison and do not involve physical robots. Historically, teams with the best performance in these two issues are usually the best positioned within category competitions, due to the fact that they are able to maintain greater possession of the ball and are more offensive to the opposing goal.

1.3 Objective and Hypothesis

Inspired from the context of the RoboCup 3D Soccer Simulation League and based upon some results from the recent techniques of Deep and Reinforcement Learning, this work aims to develop and evaluate some new learning models, based on Deep Reinforcement Learning, for the task of making a humanoid robot to kick the soccer ball and walk in RoboCup 3D Soccer Simulation environment.

In this work, we have two main hypothesis regarding of kick motion and performed an analysis about walk motion.

1.3.1 Imitation Hypothesis

We hypothesize that by representing the kick motion by a neural network with thousands of parameters and following a 2-step training procedure where:

- The first step is a Supervised Learning training, where we transfer the knowledge from a keyframe representation to a neural network; and
- The second step is a Reinforcement Learning training, where we optimize the motion from the neural network using rewards related to a specific performance task,

thus, we are able to develop better policies for kick motion that the initial ones.

1.3.2 Meta-Learning Hypothesis

We also hypothesize that we are able to "learn to learn" how to precisely kick the ball for several distances within a range, by the following procedure:

- First, learn single-task expert policies for tasks sampled from a distribution of tasks - in this case, learn how to precisely kick the ball for several different distances; and
- Using an appropriate neural network architecture, train a meta-policy using Imitation Learning with data collected from such expert policies.

By following this procedure, we expect to generalize such knowledge to unseen tasks (i.e, kick distances that are different from those used to train the meta-policy) or at least be able to quickly adapt to them.

1.3.3 Walk Analysis

In this analysis, there are two main lines of experiments to conduct:

- **"Black Box" Reinforcement Learning** - the objective is to optimize the parameters of a engine-based walk system merging techniques of RL and Evolution Strategies and compare with the initial behavior; and
- **Walk Behavior Cloning** - the objective is to follow the same principles from Imitation Hypothesis to learn the engine-based walk entirely in a single neural network and compare performances.

1.4 Scope

In this work, we will apply advanced Reinforcement Learning techniques, such as Imitation Learning and Meta-Learning, in models based on Deep Neural Networks, in order to find, through gradient-based optimization techniques, optimal policies and parameters for kick and walk control of humanoid robot. These will be contrasted with classic control techniques of humanoid robots coupled with evolutionary optimization strategies, widely used in the context of the RoboCup 3D Soccer Simulation League.

1.5 Contributions of this work

We consider that this work has several key findings and scaled Imitation Learning, Reinforcement Learning and Meta-Learning to problems with high degrees of freedom with complex dynamics involved. As major contributions, we can cite:

- A learning framework to copy motions and optimize torwards a task, which is very useful to transfer knowledge and the fine-tune to specific contexts;
- We analyzed and pinpointed a consistent methodology to how approach a Reinforcement Learning problem using Proximal Policy Optimization (PPO);
- A Meta-Learning algorithm (BUMPS), which is simpler than the existing methods but still very effective and sample efficient;
- We introduced the idea of black-box Reinforcement Learning, proposing and applying Proximal Policy Natural Evolution Strategy (PP-NES), which complements policy gradient methods with evolutionary strategies; and
- We learned a mathematically complex walk engine using a simple neural network and Behavioral Cloning, presenting the first neural-network based walk engine in the context of RoboCup Soccer 3D Simulation.

1.6 Organization of this work

The remaining of this work is organized as follows:

- Chapter 2 will review the literature of Optimization Techniques, Reinforcement Learning for Control, Imitation Learning, and Meta-Learning;

- Chapter 3 will describe the RoboCup Soccer 3D Simulation League and the traditional methods used for the kick and walk motions;
- Chapter 4 will cover the theory behind Deep Learning used in this work;
- Chapter 5 will cover the theory behind Reinforcement Learning;
- Chapter 6 will cover the theory behind Meta-Learning techniques;
- Chapter 7 will describe methodology and experiments regarding the Imitation Hypothesis;
- Chapter 8 will propose a Meta-Learning algorithm and describe methodology and experiments about the Meta-Learning Hypothesis;
- Chapter 9 will describe methodology and experiments about the Walk Analysis conducted in this work; and
- Finally, Chapter 10 will share some conclusions and suggestions for future work.

Figure 1.4 shows a flow chat that illustrates the whole content of this work.

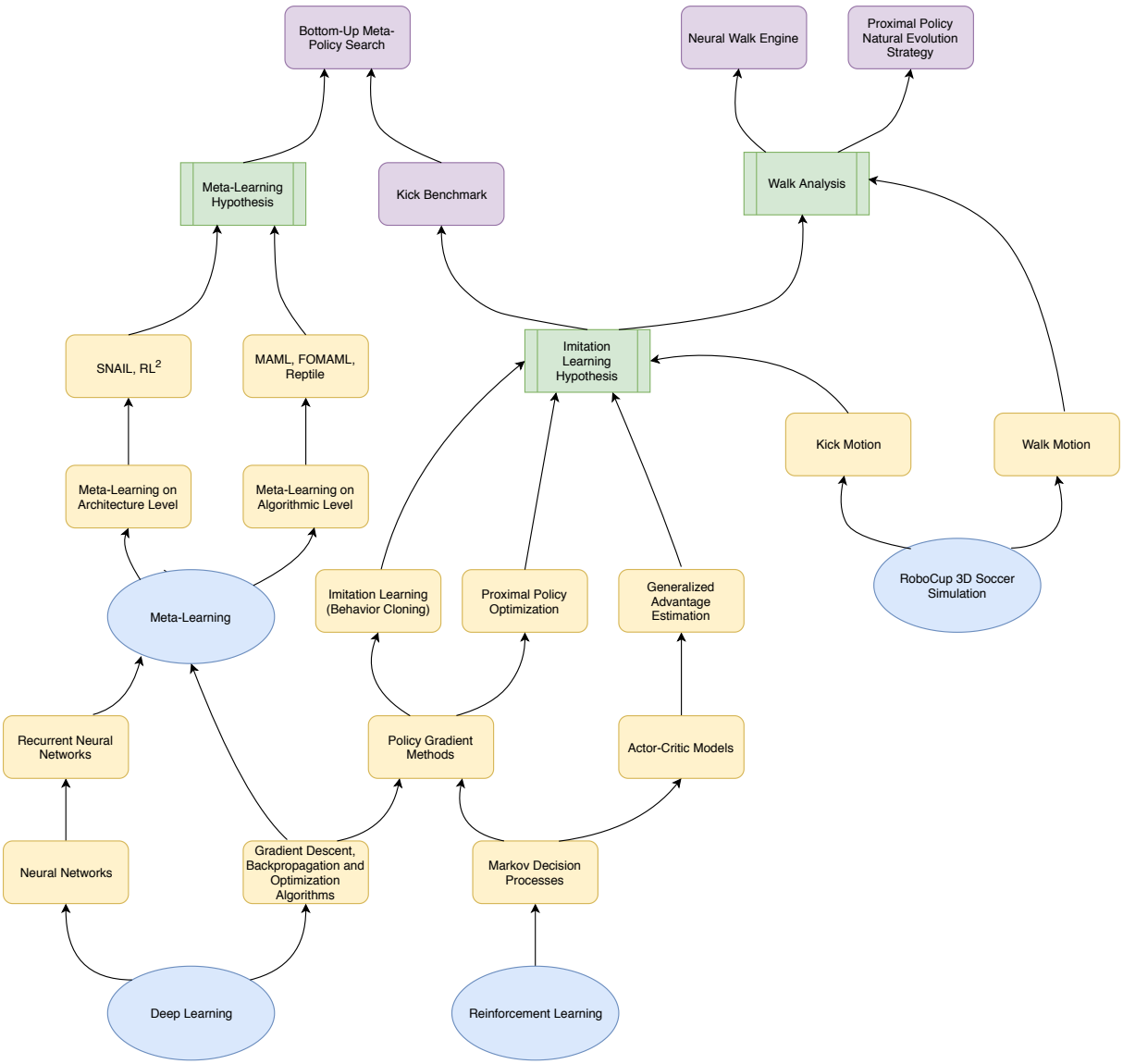


FIGURE 1.4 – Flow chart illustrating the organization of this work.

2 Literature Review

In this chapter, we will review the literature regarding Optimization Techniques, Reinforcement Learning for control tasks, Imitation Learning, and Meta-Reinforcement Learning.

2.1 Optimization Techniques

In Soccer 3D environment, optimization techniques plays an important role to achieve competitive behaviors, because they are used to obtain faster and more robust motions.

In this context, motion optimization means finding the best values for each joint in the time instant during the whole motion. There are lots of parameters involved, which makes manual tuning laborious. The common way to do this is parameterizing the motion and using optimization algorithms to find the best parameters for this model. For the kick motion, it means finding the best keyframe and interpolator values (DEPINET *et al.*, 2015). In the case of walking motion, there is an analytical control algorithm based upon Zero Moment Point (ZMP) and we optimize the walk parameters used in this algorithm, such as the center of mass height, the walking period and step height (MACALPINE *et al.*, 2012a).

In the robotics simulation world, we commonly use meta-heuristics optimization algorithms to conduct optimization. Maximo (2012) used Particle Swarm Optimization to optimize walk parameters. (URIELI *et al.*, 2011) compared the performance of several algorithms in the Soccer 3D context, such as Hill Climbing (HC), Cross-Entropy Method (CEM) (RUBINSTEIN; KROESE, 2004), and Genetic Algorithm and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (HANSEN; OSTERMEIER, 2001). The latter proved to be the most successful within the Soccer 3D environment in optimizing walking and kicking (MACALPINE *et al.*, 2012b; URIELI *et al.*, 2011), therefore being the most common in this environment.

The CMA-ES algorithm is a population-based policy search - with samples from a multivariate Gaussian distribution. These samples are evaluated with respect to a fitness

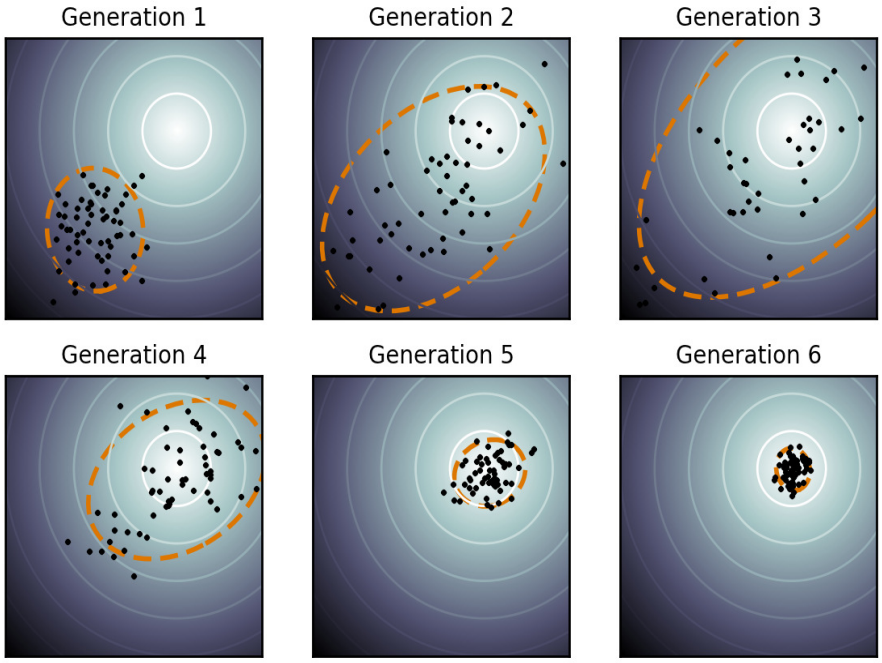


FIGURE 2.1 – Illustration of an actual optimization run with covariance matrix adaptation on a simple two-dimensional problem (KHAN, 2018).

measure. When all the candidates in the group are evaluated, the mean of the multivariate Gaussian distribution is recalculated as a weighted average of the candidates with the highest fitnesses (URIELI *et al.*, 2011), as shown in Figure 2.1. Even though the algorithm works well for kick and walk motions, it does not scale well for hundreds or thousands of parameters (MACALPINE; STONE, 2017).

2.2 Reinforcement Learning for Control

In this work, we will use recent techniques based on policy gradients rather than Evolution Strategies. Recent experimental results demonstrated that they are able to optimize in this scale (SCHULMAN *et al.*, 2015a; SCHULMAN *et al.*, 2017). These algorithms are based on Reinforcement Learning theory applied for control tasks.

Classic reinforcement learning algorithms are called tabular solution methods and use dynamic programming or sampling techniques to map state to actions. The most famous algorithms use Monte-Carlo methods, Temporal-Difference learning, SARSA (RUMMERY; NIRANJAN, 1994), and Q-Learning (WATKINS, 1989). These algorithms are good to solve small problems, such as grid worlds and multi-armed bandits. However, they may be impractical for large MDPs (SUTTON; BARTO, 1998).

However, these methods were improved using the idea of function approximation in-

stead of tabular solutions. The new algorithms started to use other learning representations, such as linear combination of features, radial basis functions (PAPIEROK *et al.*, 2008), and neural networks. The last can be exemplified by the rise of Deep Q-Networks (Figure 2.2), which were able to surpass human-level performance in several Atari games (MNIH *et al.*, 2015).

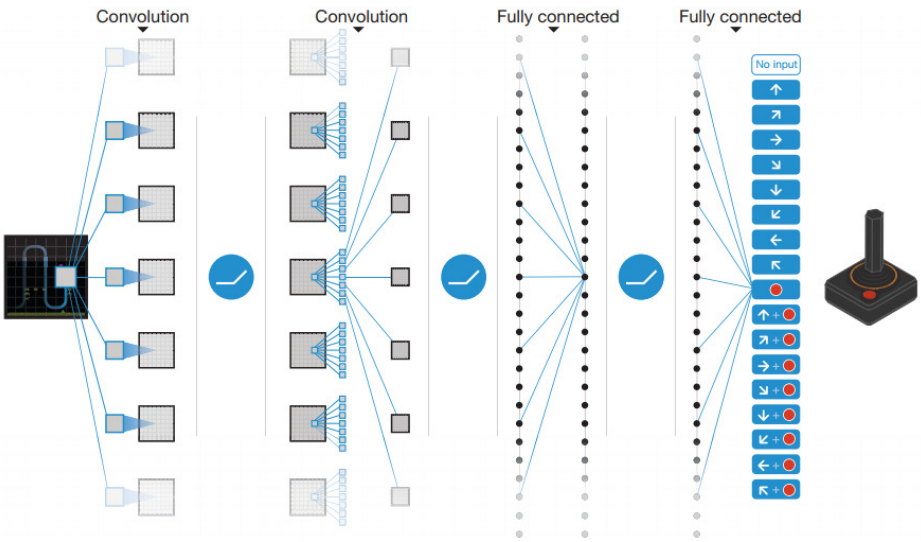


FIGURE 2.2 – Human level control through deep reinforcement learning in atari games (MNIH *et al.*, 2015).

With the advent of Deep Learning and its new techniques and the development of new computer architectures such as GPUs and TPUs, reinforcement learning has been able to scale as well, surpassing top players from the best tabletop games. For example, the AlphaGo (SILVER *et al.*, 2017c) won against the human top player of Go, considered the hardest tabular game. It used a Monte-Carlo Tree Search with a Deep Neural Network to find an optimal policy through self-play. The evolution of AlphaGo, named AlphaZero (SILVER *et al.*, 2017b) was able to learn not just Go, but also Chess and Shogi within a couple of hours.

In the last years, several algorithms arose in order to improve the performance in control tasks. These algorithms are commonly evaluated in the MuJoCo environment (TODOROV *et al.*, 2012), the most famous benchmark for continuous control tasks. Among such methods, we highlight A2C (MNIH *et al.*, 2016), ACER (WANG *et al.*, 2016b), DDPG (LILLICRAP *et al.*, 2015), GAIL (HO; ERMON, 2016), HER (ANDRYCHOWICZ *et al.*, 2017), TRPO (SCHULMAN *et al.*, 2015a) and PPO (SCHULMAN *et al.*, 2017). The TRPO and PPO algorithms use trust-region optimization to apply learning updates and then avoid catastrophic steps; the first considers a constrained optimization, while the second uses a clip parameter strategy. They will be covered in deep later, and PPO will be used in the learning framework proposed in this work, since it has proved to perform better in the MuJoCo benchmark.

In terms of applications, Heess *et al.* (2017b) applied a distributed variation of PPO in MuJoCo and were able to learn parkour movements and run in a model free fashion. These motions, however, are weird, asymmetric and, sometimes, unstable. Peng *et al.* (2018) applied the same algorithm but with key modifications in the reward function and optimization process, in order to conduct Imitation Learning to mimic reference motions, resulting in better and more human movements. Finally, Xie *et al.* (2019) developed an interative RL based design of locomotions skills for a real world Cassie robot, resulting in robust policies that closes the simulation-to-real loop effectively.

The most recent and impressive application of PPO was in OpenAI Five (OPENAI, 2018). It consists of five independent neural networks that form a team to play the online game Dota 2. This team was trained in the equivalent of 10000 years of continuous self-play, using 256 GPUs and 128 thousand of CPUs, and won from the 99.95^{th} percentile human team in a restricted environment. This is impressive not just because the environment is challenging – due to the partial observability, long time horizon and high-dimensional, continuous spaces – but also because of the multi-agent strategy layer learned. Some months later, the same OpenAI Five was able to beat the world champion Dota 2 team after some model modifications and longer train (it experienced about 45000 years of continuous self-play) (OPENAI, 2019).

2.3 Imitation Learning

Imitation Learning, sometimes named Learning by Demonstration or Programming by Example, is a technique where the agent learns a policy by acquiring skills from observing demonstrations (DUAN *et al.*, 2017). These demonstrations come from an expert policy that generates such state-action trajectories (Liu *et al.*, 2017), therefore we need an "oracle" that provides this data. Survey articles about Imitation Learning include (SCHAAL, 1999; BILLARD *et al.*, 2008; ARGALL *et al.*, 2009).

Imitation Learning has great successful stories about its applications in complex domains: Abbeel *et al.* (2010) elaborated apprenticeship learning algorithms for learning trajectory-based task specification from demonstrations and for modeling of helicopter's dynamics, in order to design autonomous flight controllers; Kober e Peters (2009) applied this technique to make a robot learn how to play tennis; and Billard e Mataric (2000) made an agent performs human-like reaching motions.

Looking at the Imitation Learning literature from a more methodological standpoint, their approaches can basically be divided into two classes: Behavioral Cloning and inverse reinforcement learning (TORABI *et al.*, 2018).

Behavioral Cloning (BAIN; SAMMUT, 1995) is a technique that dates back to Pomerleau

(1989) and models imitation learning as a supervised learning problem, where the policy is learned by state-action tuples provided by the expert and used to train a classifier or regressor. It works well for many practical problems, yielding excellent results in practice (LECUN *et al.*, 2005; POMERLEAU, 1989; MüLLING *et al.*, 2013). However, sometimes this technique suffers from distributional shift (ROSS; BAGNELL, 2010), where the training objective of imitating expert behavior along the imitation trajectories is not perfectly aligned with the true objective of performing the task correctly. Recent works combine Behavioral Cloning and model-based learning (TORABI *et al.*, 2018) or address the distributional shift problem (HAAN *et al.*, 2018)

Inverse Reinforcement Learning (IRL) is the second category of Imitation Learning. IRL learns a reward function from expert demonstrations (NG; RUSSELL, 2000); this reward function can then be used to recover a policy by running standard reinforcement learning (Liu *et al.*, 2017). Successful applications of IRL includes Generative Adversarial Imitation Learning (HO; ERMON, 2016).

In this work, we aim to use Behavior Cloning to mimic walk and kick motions. Although considering high-dimensional problems in a complex environment, we are able to apply this technique due to specific reasons: in the case of walk, we can generate a reasonable large amount of data running the agent, thus decreasing the compound error caused by distributional shift (HO; ERMON, 2016). On the other hand, in the case of kick, due to the simple formulation as a keyframe motion, we can fix initial state in our test scenarios, mitigating such problem in this context. However, all the methods proposed in this work that applies Behavioral Cloning could use other algorithms that address this problem, such as the DAgger algorithm (ROSS *et al.*, 2010).

2.4 Meta-Learning

Meta-Learning is the idea of learn how to learn efficiently, and dates back to Schmidhuber (1987), Bengio *et al.* (1991), Naik e Mammone (1992), Thrun e Pratt (1998). There are several ideas regarding how to conduct the Meta-Learning. Earlier studies have investigated automatic tuning of hyperparameters, such as the learning rate (ISHII *et al.*, 2002). However, the recent approaches rely on two main ideas and focus on the balance between performance and generality (MISHRA *et al.*, 2017).

The first idea is to find, via learning, in a called "meta-training" step, a set of parameters that encodes a structure of tasks and easily adapt via gradient descent at "meta-testing" time (RAVI; LAROCHELLE, 2017; FINN *et al.*, 2017; MUNKHDALAI; YU, 2017; GUPTA *et al.*, 2018; STADIE *et al.*, 2018). Although this approach is agnostic to the model architecture, it forces the meta-learner to follow the gradient descent and there may exist better

strategies that exploits the structure of the task in a particular domain.

A second idea is to find a meta-learner representation by using a specific model architecture. Duan *et al.* (2016) and Wang *et al.* (2016a) represented the meta-learner as recursive/recurrent networks, while Mishra *et al.* (2017) proposes a model based on temporal convolutions and an attention mechanism named Simple Neural AttentIve Meta-Learner (SNAIL). Accordingly to Mishra *et al.* (2017), this generic architecture has the capacity to learn an algorithm that exploits domain-specific task structure.

Other recent ideas of Meta-Learning, specifically to Reinforcement Learning context, is to use a learned critic network that provides gradients to the policy (SUNG *et al.*, 2017) or using a planner and an adaptable model for model-based RL (HOUTHOOFT *et al.*, 2018).

Mendonca *et al.* (2019), in a recent work done parallel to ours, proposed an algorithm that use similar ideas of leveraging supervised learning rather than relying on high-variance algorithms such as policy gradient or evolutionary strategies, decoupling the problem of obtaining expert trajectories for every task from the problem of learning a fast RL algorithm. In Meta-Policy Guided Search, they combined the Model-Agnostic Meta-Learning (MAML) from Finn *et al.* (2017) with the Imitation Learning approach to improve sample efficiency in the outer loop of optimization. In contrast, our approach is a simplified version that relies on a contextual policy to learn a rich representation of tasks and then be able to quickly adapt to unseen ones by using a meta-gradient between contexts, instead of minimizing a meta-objective. In this way, we improve even more the sample efficiency because we do not need interaction with the environment during meta-training, yet being effective to learn several kicks for a humanoid robot.

Both approaches are also related to few-shot imitation learning (DUAN *et al.*, 2017), where supervised learning is used for meta-optimization. However, in contrast to such method, our work learn using only reward signals and does not require demonstrations for unseen tasks.

Finally, our approach is also related to multi-task RL, specifically with works where there is a training procedure for each training task to provide expert policies (LEVINE *et al.*, 2016; GHOSH *et al.*, 2017). However, we use those policies to train a meta-learner that not just generalizes to unseen tasks, but also could be used to quickly adapt for new ones.

3 RoboCup Soccer 3D Simulation

In this Chapter, we will introduce the RoboCup Soccer 3D domain, where all experiments will be conducted. Furthermore, we will introduce the principles of a keyframe movement and ZMP-based walk engine.

3.1 Domain Description

The RoboCup Soccer 3D Simulation League (Soccer 3D) is a particularly interesting challenge concerning humanoid robot soccer. It consists of a simulation environment for a soccer match with two teams, each one composed by up to 11 simulated Nao robots (GOUAILLIER *et al.*, 2009), the official robot used for the RoboCup Standard Platform League since 2008. The Soccer 3D is interesting for robotics research, since it involves high level multi-agent cooperative decision making, while providing a physically realistic environment, which requires control and signal processing techniques for robust low level skills.

The RoboCup 3D simulation environment is based on SimSpark (XU; VATANKHAH, 2014), a generic physical multi-agent system simulator. SimSpark uses the Open Dynamics Engine (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. The Nao robot has height of approximately 57 cm and 4.5 kilograms. The agent sends speed commands to the simulator and receives perceptual data. Each robot has 22 joints with perceptors and effectors. Communication between agent and server happens in the frequency of 50 Hz (MACALPINE *et al.*, 2012b).

3.2 Kick Motion

In the current level of the Soccer 3D evolution, motion control is a key factor in team's performance. Indeed, controlling a high degrees of freedom humanoid robot is acknowledged as one of the hardest problems in Robotics. Much effort has been devised to humanoid robot walking, where researchers have been very successful in designing

control algorithms which reason about reduced order mathematical models based upon the Zero Moment Point (ZMP) concept, such as the linear inverted pendulum model (KAJITA *et al.*, 2001). Nevertheless, these techniques restrict the robot to operate under a small region of its dynamics, where assumptions of the simplified models are still valid (COLLINS *et al.*, 2005; MUNIZ *et al.*, 2016).

Therefore, model-based techniques are hard to use for designing highly dynamic movements, such as long distance kicks and goalkeeper dives to defend goals from fast moving balls. In the robot soccer domain, a common approach for these movements is to employ keyframe movements (Figure 3.1), where motions are composed by sequences of robot postures. In this case, movements are designed off-line and executed in an open-loop fashion in execution time.

Due to the lack of mathematical models, an approach frequently employed is to rely on human intuition to design keyframe movements by hand, usually aided by graphical tools. However, this process is difficult, time consuming, and is often unable to obtain high performance motions given the high dimensionality of the search space. Other possible solution is to use motion capture from humans (SHON *et al.*, 2005), which has its own challenges due to the fact that the kinematic and dynamic properties of a humanoid robot differs greatly from those of a human.

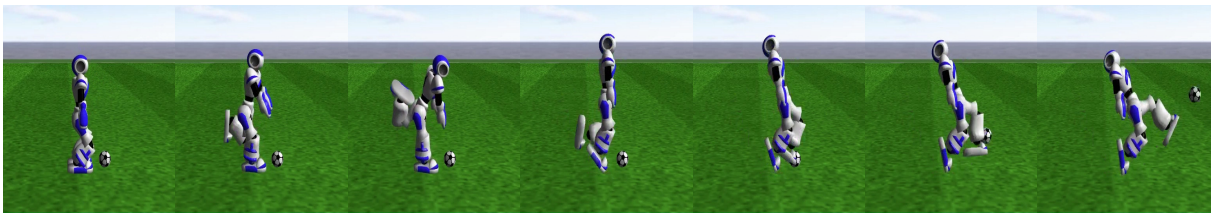


FIGURE 3.1 – A keyframe based kick motion.

3.3 Keyframe Movements

Definition 1 *A* keyframe $\mathbf{k} = [j_1, j_2, \ldots, j_n]^T \in K \subseteq \mathbb{R}^n$ *is an ordered set of joint angular positions, where K and n are the joint space and the number of degrees of freedom of the robot, respectively.*

Definition 2 *A* keyframe step *is an ordered pair* $\mathbf{s} = (\mathbf{k}, t) \in S = K \times \mathbb{R}$*, where* $\mathbf{k}$ *is a keyframe and t is the time when the keyframe must be achieved with respect to the beginning of the movement.*

Definition 3 *A* keyframe movement*, or simply a movement, is defined as the tuple* $\mathbf{m} = (\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_\gamma, r) \in M = S^\gamma \times \mathbb{R}$*, where γ and r are the number of keyframe steps and*

the speed rate of the movement. In this representation, we assume the movement starts at time 0 and the first keyframe step represents the robot posture at the beginning of the movement. Therefore, $t_1 = 0$ *and each time* $t_i, \forall i \geq 2$ *is a time since the beginning of the movement.*

Keyframe movements are executed in an open-loop fashion, where joint positions are computed through interpolation of keyframe steps based upon the current time. If the interface to the robot joints is not position-based, local controllers may be used to track the position references issued by the keyframe. For example, in the Simspark simulator, the simulated Nao Robot has speed-controlled joints, therefore, we use simple proportional controllers for each joint to track desired joint positions. To obtain smooth joint trajectories, we interpolate keyframe steps, by using cubic splines (BARTELS *et al.*, 1987), which are functions of class $\mathcal{C}^2$.

Finally, when a keyframe movement is requested, joint positions are often far away from movement's initial joint positions. Hence, simply executing the keyframe movement in this case would result in high joints accelerations, which would probably make the robot to fall. To avoid this from happening, a transition movement based on linear interpolation is first employed to bring the joint positions to the initial joint positions required by the keyframe movement.

3.4 Walk Engine

The agent's initial walk engine is a model-based controller that relies on Zero Moment Point (ZMP) concept, which says that a biped is dynamically stable at the given time instant if the ZMP – the dynamic version of the Center of Mass (CoM) – is inside the support polygon (VUKOBRATOVIĆ; BOROVAC, 2005). We will describe here how it works. For further details, refer to Maximo (2015), Maximo e Ribeiro (2016).

Firstly, the complex multibody dynamics of a humanoid robot is modeled as a inverted pendulum, where the pendulum mass represents the robot's CoM, in a model called 3D-Linear Inverted Pendulum Model (3D-LIPM) (KAJITA *et al.*, 2001), illustrated by Figure 3.2). The input to the algorithm is the desired velocity $\boldsymbol{v} = [v_x, v_y, v_\varphi]^T$ with respect to the local coordinate system of the robot, meaning the speed in forward, sideways, and rotational directions, respectively. The output, on the other side, is a vector containing the angular position of each joint.

The walking engine consists of some modules, as shown in Figure 3.3. First, at the beginning of each step, a footstep planner decides the torso and the swing foot poses that the robot must achieve at the end of the step. In order to achieve these planned poses

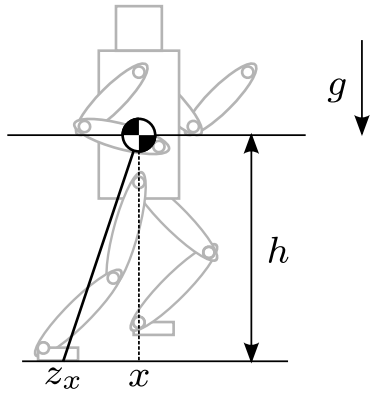


FIGURE 3.2 – The 3D Linear Inverted Pendulum Model (MAXIMO, 2015).

while keeping the robot balance, the algorithm determines trajectories for the CoM and the swing foot. The trajectory for the center of mass (CoM) that keeps the Zero Moment Point (ZMP) at the center of the support foot is computed by using an analytic solution of the 3D-LIPM equation:

$$\boldsymbol{x}_{ZMP} = \boldsymbol{x}_{CoM} - \frac{z_{CoM}}{g}\ddot{\boldsymbol{x}}_{CoM}, \tag{3.1}$$

where $\boldsymbol{x}_{ZMP} = [x_{ZMP}, y_{ZMP}]^T$ is the ZMP position, $\boldsymbol{x}_{CoM} = [x_{CoM}, y_{CoM}]^T$ is the CoM position, z_{CoM} is the CoM height, and g is the acceleration of gravity.

Additionally, the humanoid robot must satisfy dynamical constraints to remain stable and thus the walking algorithm also needs to consider robot geometry. Then, at the beginning of a step k, the footstep planner uses heuristics to select torso and swing foot poses at the beginning of the next step that satisfies these geometric constraints while following the omnidirectional model (given by Equation (3.2)) as closely as possible.

$$\begin{bmatrix} x[k+1] \\ y[k+1] \\ \varphi[k+1] \end{bmatrix} = \begin{bmatrix} x[k] + 2\frac{v_x}{v_\varphi}\sin(\frac{v_\varphi T}{2})\cos(\varphi[k] + \frac{v_\varphi T}{2}) - 2\frac{v_y}{v_\varphi}\sin(\frac{v_\varphi T}{2})\sin(\varphi[k] + \frac{v_\varphi T}{2}) \\ y[k] + 2\frac{v_x}{v_\varphi}\sin(\frac{v_\varphi T}{2})\sin(\varphi[k] + \frac{v_\varphi T}{2}) + 2\frac{v_y}{v_\varphi}\sin(\frac{v_\varphi T}{2})\cos(\varphi[k] + \frac{v_\varphi T}{2}) \\ \varphi[k] + v_\varphi T \end{bmatrix} \tag{3.2}$$

Finally, the Inverse Kinematics (IK) translates the CoM and the swing foot poses into joints positions. Finally, a feedback controller modifies the joints positions to stabilize the walk.

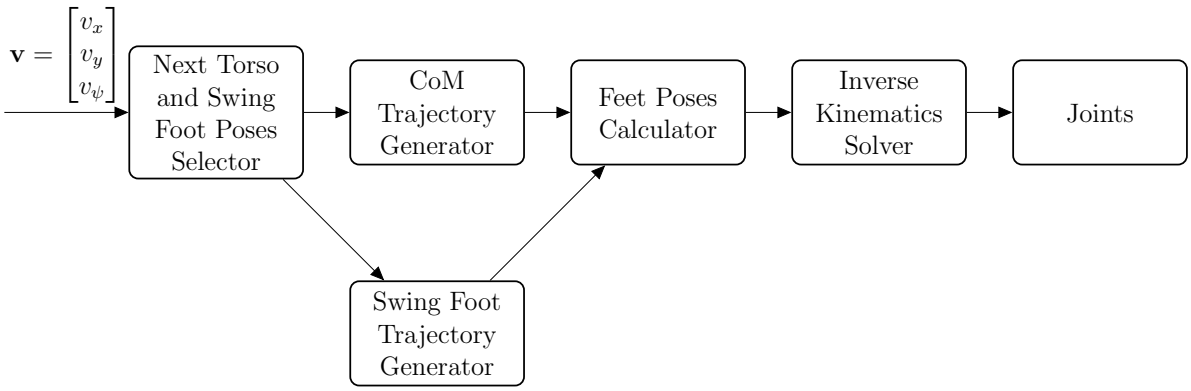


FIGURE 3.3 – The Walking Engine overview.

4 Deep Learning

In the previous chapter, we saw that the most recent Reinforcement Learning algorithms use neural networks as learning representation. In this chapter, we will describe in detail the neural network structure and its components. Additionally, we describe its learning process mathematically as an optimization problem.

4.1 Neural Networks

Neural Networks are a learning representation which goal is to approximate some function f^*. Data collected from an environment encodes an underlying function $\mathbf{y} = f^*(\mathbf{x})$ that maps an input $\mathbf{x}$ to an output $\mathbf{y}$, which may be a category from a classifier or a continue value in regression problems. The neural network defines an approximate mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$, by learning the values of the parameters $\boldsymbol{\theta}$, which result in the best function approximation. Figure 4.1 shows a neural network and an artificial neuron in detail.

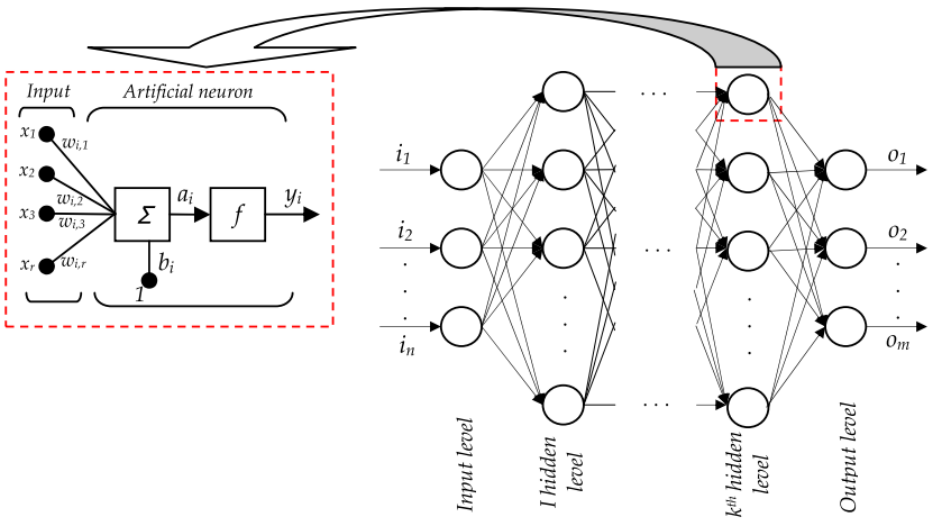


FIGURE 4.1 – An artificial neuron within a feed forward artificial neural network (TANIKIC; DESPOTOVIC, 2012).

These networks are typically represented by composing together many different func-

tions, which are associated with a directed acyclic graph, by describing a computational model. For example, we might have three layers (each of them representing a function $f^{(1)}, f^{(2)}$, and $f^{(3)}$), connecting in a chain and resulting in a final representation $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$.

During a neural network training, the objective is to adjust $f(\mathbf{x})$ to match $f^*(\mathbf{x})$, by using the training dataset, which provides noisy examples of $f^*(\mathbf{x})$ evaluated in different points. The training examples directly specify what the output layer must do at each point $\mathbf{x}$, but the learning algorithm must decide how to use all layers to produce this desired output (GOODFELLOW *et al.*, 2016).

4.2 Maximum Likelihood Estimation

In the context of neural networks, gradient-based algorithms are broadly used, especially those based on the backpropagation idea (RUMELHART *et al.*, 1988). The purpose of these algorithms are to propagate the gradient of a cost function through the whole network, in order to minimize the cost function. Most modern neural networks perform this optimization strategy, by using maximum likelihood, i.e. the cross-entropy between the training data and the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x}). \tag{4.1}$$

In this work, we used the mean squared error loss function, in order to fit data distributions. Indeed, we may show that both cost functions are closely related. Let us consider normally distributed errors:

$$p_{model}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x};\boldsymbol{\theta}), \sigma^2\mathbf{I}). \tag{4.2}$$

where $f(\mathbf{x};\boldsymbol{\theta})$ and $\sigma^2\mathbf{I}$ are the mean and covariance of this distribution. By substituting Equation (4.2) in Eq. (4.1):

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}}\|\mathbf{y} - f(\mathbf{x};\boldsymbol{\theta})\|^2 + const. \tag{4.3}$$

The constant term does not depend on $\boldsymbol{\theta}$ and may be dropped. By explicitly evaluating the expectation in Equation (4.3), we arrive at the mean squared error cost function:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i}^{m}\|y_i - f(\mathbf{x};\boldsymbol{\theta})\|^2. \tag{4.4}$$

It can be shown that maximum likelihood estimator is the best estimator asymptotically, when the number of samples m tends to infinity, in terms of its rates of convergence as m increases. Furthermore, this estimator has the property of consistency, under certain conditions that must be satisfied by the data distribution, and efficiency, in terms of achieve lower generalization errors.

4.3 Gradient Descent and Backpropagation

The objective of a neural network training is to find the parameters W and b that approximates the function represented by itself and the data generator distribution, by minimizing the cost function established. In mathematical terms, it means:

$$(W^*, b^*) = \arg\min_{W,b} J(W, b, x). \tag{4.5}$$

The Gradient Descent technique is an iterative method that calculates the derivative of the loss function and applies them as learning updates according to a learning rate factor, α, previously set. Figure 4.2 illustrates the application of this technique in the context of two variables.

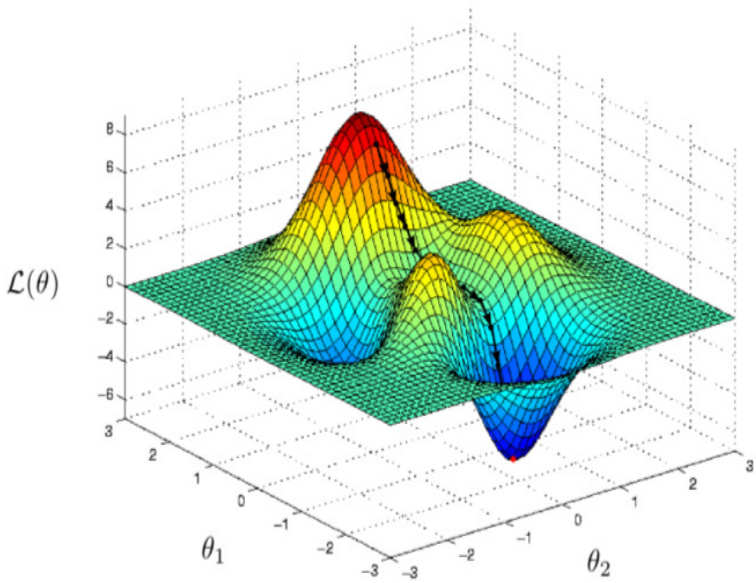


FIGURE 4.2 – The illustration of gradient descent in a two variables optimization (MAGALHAES, 2017).

In neural networks, we have to compute the gradients that correspond to all layers and not just to the output layer. For this, we need to use the chain rule of calculus. Let x be a real number, and let f and g be both functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states

that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{4.6}$$

We can generalizes to vectorial case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^m$ to $\mathbb{R}^n$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $\boldsymbol{z} = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{4.7}$$

It can be generalized in a way that can be applied to tensors:

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j)\frac{\partial z}{\partial Y_j}. \tag{4.8}$$

The backpropagation takes Equation (4.8) recursively, until the gradient is propagated to all layers of the neural network. Algorithm 1 (GOODFELLOW *et al.*, 2016) shows how to apply backpropagation to a deep neural network.

Algorithm 1 Backward computation for a deep neural network

After the forward computation, compute the gradient of output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} \mathcal{L}(\hat{\boldsymbol{y}}, y)$
for $k = l, l-1, \ldots, 1$ **do**
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases:
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$
end for

It is important to mention that there are no guarantees the optimization will converge to a global optimum, i.e, it is possible to get stuck in local optimum. However, in practical applications, this is not a huge problem (GOODFELLOW; VINYALS, 2014). Additionally, in this work, we use an improved version of Gradient Descent, called Adam (KINGMA; BA, 2014), which uses a technique called Momentum (SUTSKEVER *et al.*, 2013) to normalize the gradient in order to improve learning stability.

4.4 Recurrent Neural Networks

Until this point, we considered a neural network representation as a feed-forward network. In this Section, we will present a different architecture called Recurrent Neural Networks (RNN). It is worth to mention that the content previously presented is also valid for this kind of network.

Basically, an RNN is a neural network that process sequential data. We input a sequence $(x_0, x_1, \ldots, x_n)$, which process each timestep and uses the output from a timestep as feature to the subsequent one. Figure 4.3 illustrates this scheme.

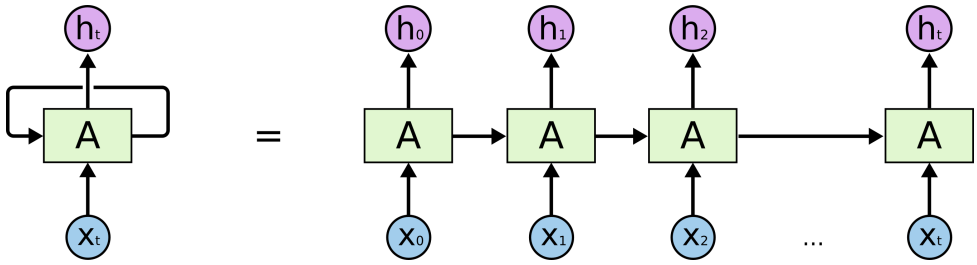


FIGURE 4.3 – A Recurrent Neural Network (OLAH, 2015).

As observed, the RNN has several sequential cells. Figure 4.4 details the operations inside each cell. Basically, we combine the actual hidden state h_t and the input x_t to compute the next hidden state h_{t+1}. Hidden states are also used to compute the outputs using another non-linearity.

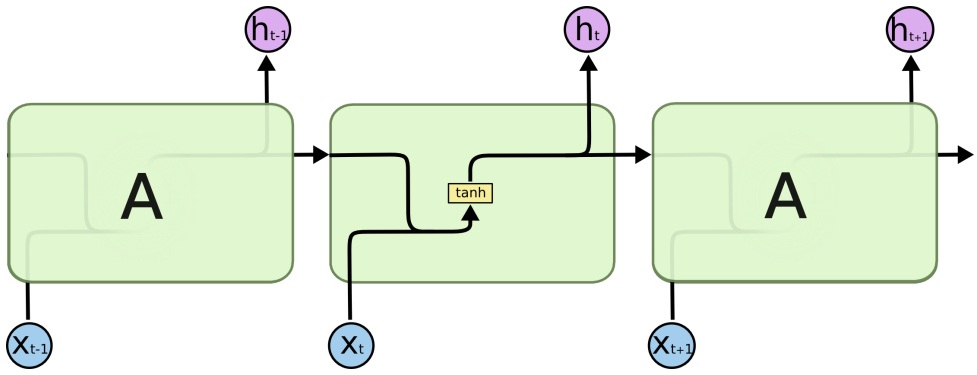


FIGURE 4.4 – Inside a RNN cell (OLAH, 2015).

This architecture is very important to sequential problems, which include Reinforcement Learning optimizations, because we can use past information to compute an action, rather then relying only in the actual information.

4.4.1 Long Short-Term Memory

In this work, we focus on Long Short-Term Memory (LSTM) architecture (HOCHREITER; SCHMIDHUBER, 1997), which is a sophisticated type of RNN. Basically, standard RNN are hard to train because they suffer from long-term sequences (Bengio *et al.*, 1993). In terms of optimization, the gradient is sequentially scaled through RNN cells, in a way that it suffers from problems called vanishing and exploding gradients, where, respectively, they scale and are not able to alter weights or change them drastically.

LSTM networks, on the other side, are specially design to lead with long-term dependencies by using a memory element. Thus, the default behavior of such architecture is to remember long periods of information. A LSTM cell is presented in Figure 4.5.

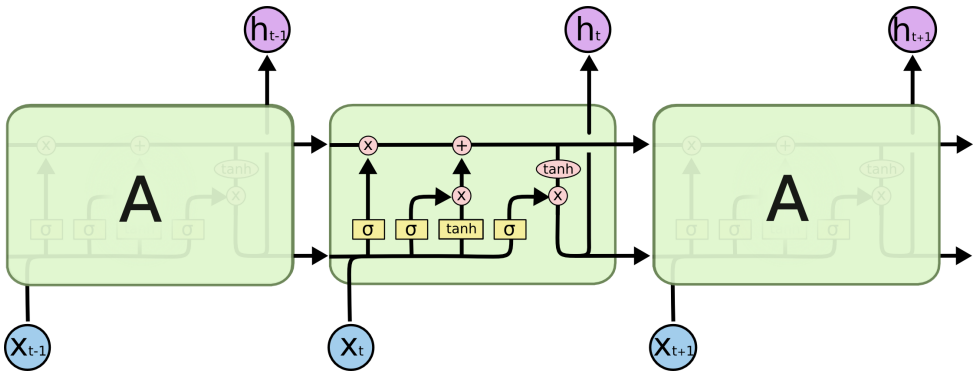


FIGURE 4.5 – Inside an LSTM cell (OLAH, 2015).

As we can see, there are several additional details inside an LSTM cell, composed by combinations of sigmoid/tanh non-linearities. These components form the LSTM gates (namely forget, input, and output gates), which is basically a sigmoid function that decides which information flows in each part of the cell. Additionally, there is an internal cell state, used as memory element.

The forget gate layer is a sigmoid layer composed by previous hidden state h_{t-1} and input timestep x_t, which decides what information will be thrown away from cell state. On the other hand, the input gate layer is composed by a multiplication between sigmoid and tanh layers, both applied to previous hidden state h_{t-1} and input timestep x_t. Its purpose is to decide what new information will be stored in cell state. The cell state is updated by using such information.

Finally, the output gate layer has sigmoid layer again applied to hidden state and input timestep and then combined with a tanh layer applied to cell state. The result of this operation is the actual hidden state h_t.

5 Reinforcement Learning

In this Chapter, we define an RL system and its components mathematically as a Markov Decision Process. Then, we introduce some methods to solve an MDP: Exact Solution Methods and Policy Gradient Methods. Finally, we present the Advanced Policy Gradient algorithms, which will be used in this work.

5.1 Concepts of a Reinforcement Learning System

Reinforcement Learning (RL) is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards (SUTTON; BARTO, 1998).

This learning process happens by interacting with the environment where the agent lives and therefore chooses its actions. It is basically a "trial and error" approach as humans or other animals do, but in this case using a computational approach and mathematical modeling.

The origin of modern Reinforcement Learning has two main threads developed outside Artificial Intelligence research. One concerns about trial and error search inside psychology of animal learning, based on Behaviorism theory (SKINNER, 1953) and "The Law of Effect" (THORNDIKE, 1933). The other thread concerns the problem of optimal control and its solution using value functions and dynamic programming. Both of them contributed for the modern ideas in reinforcement learning, either with the abstractions of learning or mathematical modeling for this problem.

Reinforcement Learning is different from other kinds of Machine Learning, for several reasons:

- **There is no supervisor, only a reward signal** – in Supervised Learning, there is a set of pairs (x, y) and we optimize a cost function in a way to minimize the error

from prediction and the ground truth. Therefore, there is a knowledgable external supervisor and a clear specification of mapping between situation and action to take - which doesn't exist in the context of reinforcement;

- **Feeback is delayed, not instantaneous** – in Reinforcement Learning, the reward is a consequence of a sequence of actions, rather than a direct mapping from actions to reward. For example, during the learning of a kick motion, the final reward of ball movement only occurs after all the agent motion;
- **Data is not independent and identically distributed (i.i.d)** – in this kind of learning, there is a intrinsic sequentiality, i.e, the data we collect at some point depends on the state we are; and
- **Agent's actions affect the subsequent data it receives** – more than being i.i.d data, the learning process itself changes the data distribution, differently from supervised learning where the dataset doesn't change during the course of training.

Therefore, the optimization problem in Reinforcement Learning has new challenges and needs a different analysis when compared with other kinds of Machine Learning. In this chapter, we will explore these ideas and state-of-art techniques to sustain our experimentations.

5.2 Reinforcement Learning System

Figure 5.1 shows how a Reinforcement Learning system works and some of the core concepts involved. As described in section 5.1, there are two main entities in Reinforcement Learning: an *agent*, which takes actions with the object of accumulate reward; and an *environment*, the place where happens this interaction and whose dynamics affects the decision about these actions.

During learning process, the agents gets an *observation* from the environment and maps to a *state*. Accordingly to that state, the agent then executes an *action* in the environment, based on the current *policy* of actions, which has a consequence in the environment and causes a transition in that state. Furthermore, the agent receives a *reward* signal about how that action affected the environment in terms of accomplishing the task goals. This cycle will happen iteratively and the policy will be adjusted in a way that maximize the cumulative reward.

In the next subsections, we will describe in deep the main concepts of an RL system.

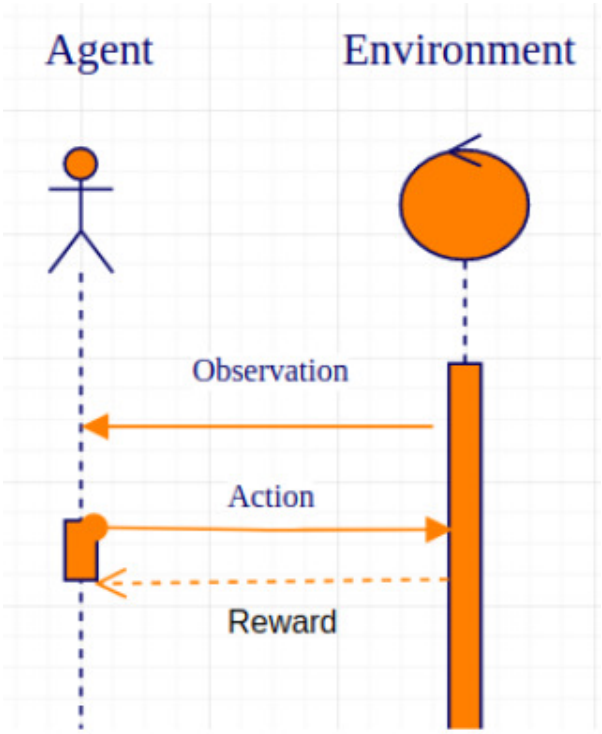


FIGURE 5.1 – Reinforcement Learning System.

5.2.1 Reward

A reward signal is a scalar number received by the agent and drives the goal in an RL problem. It indicates how well is the action taken in that state. In the learning process, the reward signal guides the policy modification: when an action has high reward, it is a case of positive reinforcement and the policy will be adjusted to take this action more likely in that state; on the other side, if the reward is low, it is a case of negative reinforcement and policy will try to act in this way less times.

Finally, we can formalize this idea stating that RL is based on the **Reward Hypothesis**:

Definition 4 *All goals can be described by the maximization of expected cumulative reward.*

5.2.2 State

Basically, a state is a function of the sequence observations, actions, and rewards. There are two different concepts of State:

- **Environment State** – Environment's internal representation - the way it understands its feature at that point and uses to pick next observation or reward; and

- **Agent State** – The information that agent perceives and understand from the observation and uses to pick a new action.

In RL problems, we commonly use the **Agent State** concept – the information provided by the environment and interpreted by the agent in order to decide which action to take next. More specifically, we will use the concept of Markov state, which will be discussed later in Subsection 5.3.1.

5.2.3 Policy

A policy defines agent's behavior. It maps states to actions. It corresponds to what in psychology would be called a set of stimulus–response rules or associations (SUTTON; BARTO, 1998).

The policy is the core element of an RL system. It is sufficient to define the behavior of an agent and therefore is the piece we need after the learning process. There are two kind of policies:

- **Deterministic Policy**, where there is a direct map from state to action. The action taken in each state is always the same:

$$a = \pi(s). \tag{5.1}$$

  In Equation (5.1), the action a is taken by policy π as function of the state s.

- **Stochastic Policy**, where the map happens from the state to a probability distribution between actions:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]. \tag{5.2}$$

5.2.4 Value Function

Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run. The value function is an estimate of reward the agent can expect to accumulate over the future, until a terminal state.

The value function can be formalized in Equation (5.3):

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s]. \tag{5.3}$$

In Equation (5.3), v_π represents the value function related to policy π, as the expectation of cumulative sums of rewards R. We also consider a discount factor γ, which defines how many terms will be considered as infinite sum.

In RL, we focus on choosing actions that bring about states of highest value, not just highest reward. Therefore, it is common to sacrifice immediate rewards to maximize future rewards.

Furthermore, it is much harder to determine value than rewards, because while this is given directly from the environment, that one must be estimated accordingly to data the agent receives. We will describe state-of-art algorithms to predict value function later.

5.2.5 Model

A model is a component that predicts environment's behavior. Basically, it will estimate the next state given the agent's action, as formalized in Equation (5.4):

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a], \tag{5.4}$$

where $\mathcal{P}$ is a transition matrix which calculates the probability of transit between state s and s' given the action a.

Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced (SUTTON; BARTO, 1998). There are two methods based on model classification:

- **Model-based RL**, methods that uses model and planning to solve the learning challenge; and
- **Model-free RL**, which are methods that are direct trial and error learners.

Sometimes there is no model of the environment or it is very hard to create it – in that cases, we commonly use model-free methods. In this work, we do not know about the dynamics of the Soccer 3D simulator, thus we adopt model-free RL.

5.3 Markov Decision Process

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards (SUTTON; BARTO, 1998). Thus, MDPs

formally describe an environment for Reinforcement Learning. Almost all RL problems can be formalized as MDPs (SILVER, 2015a).

In the next subsections, we will define mathematically what is an MDP, its major components and properties.

5.3.1 Markov State

A state S_t is considered *Markov State* if and only if:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]. \tag{5.5}$$

Intuitively, Equation (5.5) tells that in a Markov State, the future is independent of the past given the present. It means that the state has all the relevant information from the history and therefore it is sufficient statistic for the future.

As an example, consider a chess game: given a specific state – a set of pieces in the board – all the information a player needs to play is in the board state, regardless of how the game came in that circumstance. The history may be thrown away.

5.3.2 State Transition Matrix

Other major component of an MDP is the State Transition Matrix. Given a set of Markov States, this component defines the probabilities of transition between states. More formally:

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \ldots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \ldots & \mathcal{P}_{nn} \end{bmatrix}, \tag{5.6}$$

where $\mathcal{P}_{ss'}$ is the transition probability:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]. \tag{5.7}$$

5.3.3 Markov Decision Process

Using the concepts from Subsections 5.3.1 and 5.3.2, we can define a *Markov Process* (MP):

Definition 5 *A Markov Process, or Markov Chain, is a tuple* $(\mathcal{S}, \mathcal{P})$*, where:*

- $\mathcal{S}$ *is a set of states; and*
- $\mathcal{P}$ *is the state transition probability matrix.*

Considering the definition of Markov Process and the concept of Reward described in section 5.2.1, we can define a *Markov Reward Process* (MRP):

Definition 6 *A Markov Reward Process, is a tuple* $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$*, where:*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{P}$ *is the state transition probability matrix;*
- $\mathcal{R}$ *is a reward function, i.e,* $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$*; and*
- γ *is a discount factor, where* $\gamma \in [0, 1]$*.*

A Markov Reward Process (MRP) describes mathematically an environment in an RL problem. Furthermore, we can differentiate an MRP from an MP by the concept of *Return* that is directly related to the reward over time:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{5.8}$$

The idea around cumulative reward was described earlier in section 5.2. However, we need to analyze the discount factor γ – it will weight the trade-off between immediate and delayed rewards.

Basically, if γ is close to zero, the MRP will consider more immediate rewards; on the other side, if γ is close to one, the MRP will consider a more delayed reward. Furthermore, the idea of discount is mathematically convenient and avoids infinite returns in cyclic Markov processes (SILVER, 2015a).

Finally, we can extend the idea of MRP by adding a decision making procedure execute by the agent – which results in the concept of *Markov Decision Process*:

Definition 7 *A Markov Decision Process, is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$*, where:*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{A}$ *is a set of actions;*
- $\mathcal{P}$ *is the state transition probability matrix;*
- $\mathcal{R}$ *is a reward function, i.e,* $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$*; and*

- γ *is a discount factor, where* $\gamma \in [0, 1]$.

Thus, MDP is an MRP with decisions. It is worth to mention that the state transition probability $\mathcal{P}$ in MDP also takes in consideration the action executed by the agent:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]. \tag{5.9}$$

We can also extend the idea of MDP to partially observable environments:

Definition 8 *A Partially Observable Markov Decision Process (POMDP), is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma)$*, where:*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{A}$ *is a set of actions;*
- $\mathcal{O}$ *is a set of observations;*
- $\mathcal{P}$ *is the state transition probability matrix;*
- $\mathcal{R}$ *is a reward function, i.e,* $\mathcal{R}_s = \mathbb{E}[R_{t+1} | S_t = s]$*;*
- $\mathcal{Z}$ *is an observation function,* $\mathcal{Z}^a_{s',o} = \mathbb{P}[O_{t+1} = o \mid S_{t+1} = s, A_t = a]$*; and*
- γ *is a discount factor, where* $\gamma \in [0, 1]$.

Basically, an MDP is a POMDP, when the agent does not have all information about a state, but only an observation that encodes what it knows about the state.

Now, we have a formal definition of an RL problem by using MDP concept. Figure 5.2 shows the agent-environment interaction in an MDP. In the next section, we will formalize other RL components in an MDP context.

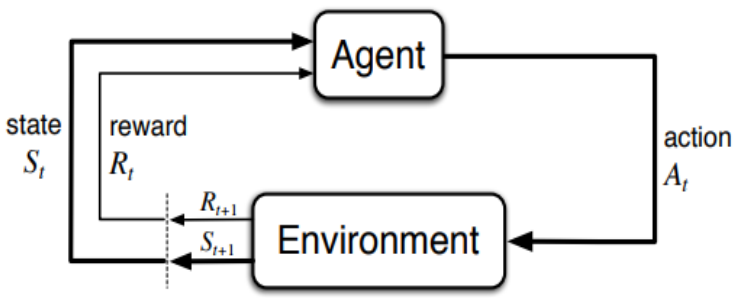


FIGURE 5.2 – The agent–environment interaction in a Markov decision process (SUTTON; BARTO, 1998).

5.3.4 State-Value and Action-Value Function

In an MDP context, we can explore deeper the idea of Value Function, and thus define *State-Value* and *Action-Value functions.*

Definition 9 *The state-value function $v_\pi(s)$ of an MDP is the expected return starting from s and following a policy π:*

$$v_\pi(s) = \mathbb{E}_\pi[G_t \big| S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right]. \tag{5.10}$$

The only difference between this definition and the one explained in Section 5.2.4 is the fact that we consider state-value function as a component of an MDP and in terms of return. However, a new concept arises in MDPs: Action-Value function:

Definition 10 *The action-value function $q_\pi(s, a)$ of an MDP is the expected return starting from state s, taking action a, and then following policy π afterwards:*

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]. \tag{5.11}$$

From definitions 9 and 10, we can see that the action-value function is a granulated version of state-value, considering the new dimension of action space. It is worth to mention that, by definition, the action-value evaluates a specific action in a state, and then follows the policy π, which means that action a is not chosen by that π.

5.4 Bellman Equation

5.4.1 Bellman Expectation Equation

Given definition 9, it is possible to develop state-value function equation and come to a fundamental property of value functions: they satisfy a recursive relationship called *Bellman Equation*:

$$
\begin{aligned}
v_\pi &= \mathbb{E}_\pi[G_t \mid S_t = s] && (5.12)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] && (5.13)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] && (5.14)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (5.15)\\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \middle| S_{t+1} = s' \right]. && (5.16)
\end{aligned}
$$

Equation (5.16) can be simplified in terms of reward and transition matrix:

$$
v_\pi(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v_\pi(s'). \tag{5.17}
$$

The Bellman equation expresses relationship between a state and successor states. Figure 5.3 illustrates an agent-environment interaction. This equation average over all the possibilities, weighting by the probability of occurring.

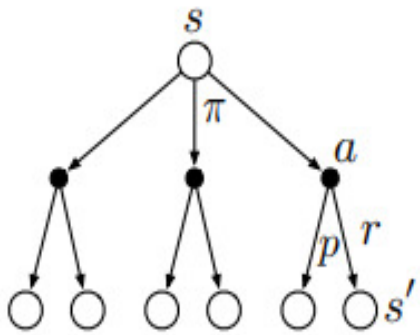


FIGURE 5.3 – Backup diagram for v_π. The white nodes are states, while the black ones are possible actions. Bellman equation average over the actions in a recursive way. (SUTTON; BARTO, 1998).

The Bellman equation can also be derived for action-value function. Given:

$$
q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]. \tag{5.18}
$$

It is possible to develop Equation (5.18), resulting in a version analogous to Equation (5.17):

$$
q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q_\pi(s', a'). \tag{5.19}
$$

The Bellman equation forms the basis of algorithms to learn v_π. The version described here is called *Bellman Expectation Equation* and, as a linear equation, has exact solution. However, there are variations whose purpose is to find optimality in policies and value functions – and the solution for this methods is not straightforward.

5.4.2 Bellman Optimality Equation

Solving an RL problem means find a policy that its actions results in the maximum cumulative reward. It means find the best policy during learning. We can define optimality for policies based on optimal value functions.

Definition 11 *The optimal state-value function $v_*(s)$ is the maximum state-value function over all policies:*

$$v_*(s) = \max_\pi v_\pi(s). \tag{5.20}$$

Similarly, the optimal action-value function $q_*(s, a)$ is the maximum action-value function over all policies:

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{5.21}$$

The optimal value function specifies the best possible performance in the MDP. Therefore, an MDP is considered solved when we know the optimal value function (SILVER, 2015a).

Based on the idea of value function optimality, it is possible to define policy ordering:

Definition 12 *Given policies π, π', we can define a partial ordering in the way that*

$$\forall s \in \mathcal{S}, v_\pi(s) \geq v_{\pi'}(s) \Rightarrow \pi \geq \pi'.$$

Based on this definition, it results in the *Policy Optimality Theorem*:

Theorem 1 *For any MDP:*

- *There exists an optimal policy π_* such that $\pi_* \geq \pi, \forall \pi$; and*
- *All optimal policies achieve the optimal state-value and action-value functions: $v_{\pi_*}(s) = v_*(s)$ and $q_{\pi_*}(s, a) = q_*(s, a)$.*

Also from optimal value function definition, it is possible to define *Bellman Optimality Equation*:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) && (5.22)\\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (5.23)\\
&= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] && (5.24)\\
&= \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s'). && (5.25)
\end{aligned}
$$

Similarly, for action-value function:

$$
\begin{aligned}
q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] && (5.26)\\
&= \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] && (5.27)\\
&= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a). && (5.28)
\end{aligned}
$$

Figure 5.4 illustrates the Bellman Optimality equation for both state-value and action-value functions, similarly to Figure 5.3. Instead of averaging over actions, the optimality equation chooses the path of greedy action.

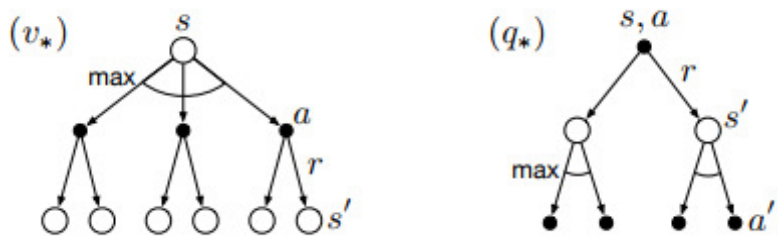


FIGURE 5.4 – Backup diagram in Bellman Optimality Equation for v_π and q_π. (SUTTON; BARTO, 1998).

Solving the Bellman Optimality Equation gives the optimal policy for a specific MDP. Therefore, we can conclude that the solution of an RL problem is fundamentally solve this equation. However, as it is non-linear, there is no closed form solution, so we need to explore ways to achieve it.

5.5 Exact Solution Methods

The first algorithms created to solve the Bellman Optimality Equation in RL problems are based in Dynamic Programming (DP) methods. They require a known MDP model and are, in general, computationally expensive. However, it is important to understand and know their limitations before dive into modern approaches.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. Thus, the exact solution methods are obtained turning Bellman equations into update rules for improving approximations of the desired value functions (SUTTON; BARTO, 1998).

5.5.1 Policy Iteration

The first method described here is the Policy Iteration. Figure 5.5 illustrates how it works. This technique is composed by two main steps: policy evaluation and policy improvement.

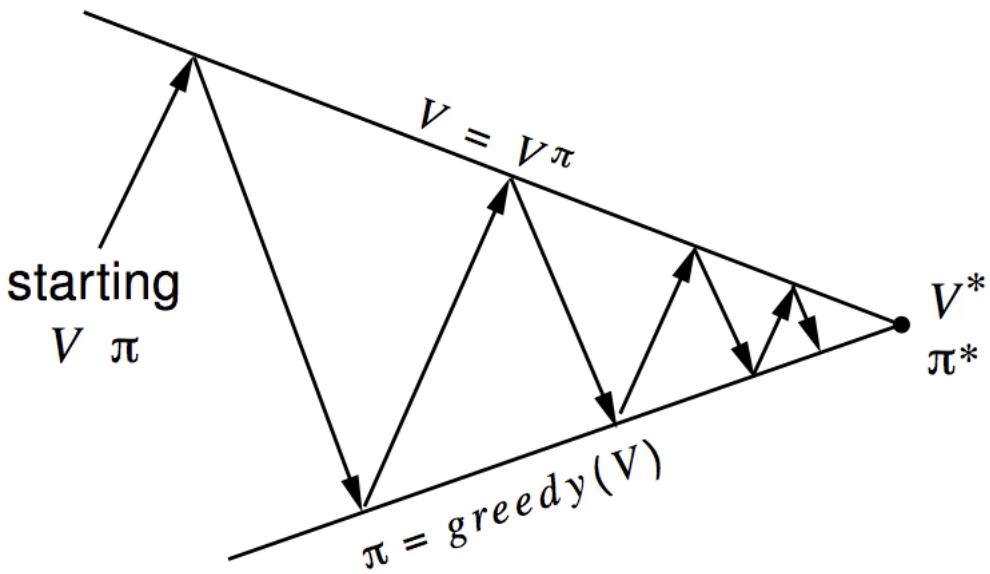


FIGURE 5.5 – Policy Iteration algorithm. (SILVER, 2015b).

In evaluation, the value function v_π from the current policy π will be estimated. As result, we will have a value function that predicts well the real v_π. Then, it is possible to modify the current policy by taking greedy actions based on the value function, during policy improvement. Basically, we would like to generate π' in a way that $\pi' \geq \pi$.

After generating this new policy, we go back to evaluation. This cycle occurs until we find the optimal value function and policy. Therefore, there are two challenges inside policy iteration. We will formulate each of them.

5.5.1.1 Policy Evaluation

During evaluation, we may just apply Bellman updates using the data collected by the agent through the episodes:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s]. \tag{5.29}$$

Additionally, it can be shown that:

$$\lim_{k\to\infty} v_k = v_\pi. \tag{5.30}$$

Algorithm 2 shows the iterative policy evaluation pseudocode.

Algorithm 2 Iterative Policy Evaluation

Require: π, current policy
Require: $V(s)$, an array for storing value function
  Initialize $V(s) = 0$, $\forall s \in \mathcal{S}^+$
  while δ is large enough **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
    $\delta \leftarrow \max(\delta, |v - V(s)|)$
  end while

5.5.1.2 Policy Improvement

During policy improvement, the value function evaluated before will be used to find better policies.

This step is based on *Policy Improvement Theorem*:

Theorem 1 *Let π' and π be any pair of deterministic policies such that, $\forall s \in \mathcal{S}$:*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{5.31}$$

Then the policy π' must be as good as or better than π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{5.32}$$

The proof of this theorem can be found in (SUTTON; BARTO, 1998). Thus, we can

improve the policy by acting greedily:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a) \tag{5.33}$$

Acting this way, it is possible to satisfy the condition to apply the theorem:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s) \tag{5.34}$$

If the improvement stops, it means the equality has been achieved. Therefore, the Bellman optimality equation has been satisfied:

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a) = v_*(s) \tag{5.35}$$

5.5.1.3 Policy Iteration algorithm

The policy iteration, as described earlier, is a sequence of evaluations and improvements, until convergence to optimal policy. The Algorithm 3 shows the pseudocode (SUTTON; BARTO, 1998);

Algorithm 3 Policy Iteration

1. Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily , $\forall s \in \mathcal{S}$
2. Run policy evaluation described in Algorithm 2
3. Policy Improvement
   policy_stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old_action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
      if *old_action* $\neq \pi(s)$, then *policy_stable* $\leftarrow$ *false*
   if *policy_stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2.

5.5.2 Value Iteration

In the Value Iteration, instead of applying two cyclic steps of evaluation and improvement, the algorithm directly applies Bellman optimality backups. Therefore, this method effectively combines the steps from Policy Iteration, which often results in faster convergence.

Algorithm 4 presents the pseudocode for this technique.

Algorithm 4 Value Iteration

Initialize $V(s)$ arbitrarily, $\forall s \in \mathcal{S}^+$
while δ is large enough **do**
  $v \leftarrow V(s)$
  $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
  $\delta \leftarrow \max(\delta, |v - V(s)|)$
end while
Output a deterministic policy $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$

5.5.3 Limitations of Exact Solution Methods

During this section, we present the core ideas from Exact Solution Methods. They have convergence proofs and are reasonably simple algorithms. However, there are limitations in their uses, which turns impracticable in the context of this work.

The first limitation is that these algorithms require knowledge about the dynamic models. In other words, we need to know about the transition matrix $\mathcal{P}^a_{ss'}$, which turns the usage of these methods very limited. For example, in the case of controlling the kick motion, we need the dynamics of the simulation server and there is not such model.

To solve this problem, many **sampling-based methods** have been developed. These are model-free RL algorithms and are applied replacing the expectation from Bellman equations by samples of agent executing actions within the environment. From this idea, some methods have arisen, such as **Monte-Carlo** and **Temporal-Difference methods**, which resulted in classic algorithms like SARSA (RUMMERY; NIRANJAN, 1994) and Q-Learning (WATKINS, 1989). We will not describe them here for the sake of conciseness. However, the idea of sampling based approximations will be used during experimentation.

The second limitation is about storing and iterating over all states and actions for large MDPs. As described during this Section, using Dynamic Programming requires saving values for each state or action-state. It does not scale well, especially for high dimensional continuous domains like the environment used in this work. For example, in the game of Go, there are 10^{170} possible states. If we represent each state with a single bit, we would use approximately 10^{160} gigabytes – there is no capacity in the world for it. Furthermore, there will be a large number of iterations to learn all values individually.

To overcome this problem, the solution is to approximate the value function or even the policy itself using a learning representation:

$$\hat{v}(s, \boldsymbol{\theta}) \approx v_\pi(s), \tag{5.36}$$

$$\hat{q}(s, a, \boldsymbol{\theta}) \approx q_\pi(q, a). \tag{5.37}$$

$$\pi_{\boldsymbol{\theta}}(s, a) \approx \pi(s, a). \tag{5.38}$$

The idea behind this is that we can represent all states in a compact way and, during training, generalize from seen to unseen states (SILVER, 2015d).

We will use neural networks, for some reasons. First, it is differentiable, thus it is possible to use gradient methods for learning. Second, it allows to use data that is not stationary, independent or identically distributed. Lastly, we can use Deep Learning – shown in Chapter 4 – to scale up learning, both in capacity and convergence velocity.

In the next sections, we will define policy gradient methods and describe the major algorithms used in this work.

5.6 Policy Gradient Methods

In Policy Gradient Methods, the objective is to learn a parameterized policy that can select actions without consulting a value function, which can be used just for training purposes. These methods compute the gradient for some cost function $J(\theta)$ with respect to policy parameters, in a way to maximize the performance.

The usage of policy-based algorithms has some advantages. Firstly, parameterize a policy π is often simpler than value functions. Furthermore, $v_\pi(s)$ does not prescribe actions (thus would need dynamics modeling and Bellman updates) and $q_\pi(s, a)$ need to efficiently solve $\arg\max_a q_\pi(s, a)$, which can be challenging for continuous problems (ABBEEL *et al.*, 2017).

Computing gradients ensures better convergence properties and is effective in continuous action spaces. Additionally, we can learning stochastic policies. The disadvantages are sub-optimality convergence and policy evaluation becomes challenging because is typically inefficient and have high variance (SILVER, 2015c).

Finally, policy direct optimization is useful, because it is possible to optimize what we effectively care about and in a compatible way with richer architectures (such as deep neural networks).

5.6.1 Policy Gradient Theorem

There is a excellent theoretical answer which ensures that in parameterized policies, the gradient does not depends on the dynamics model, i.e, it does not involve the derivative of state distribution.

Theorem 1 *Suppose a differentiable parameterized policy that will be optimized using gradient by some performance measure $J(\theta)$. Then:*

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}). \tag{5.39}$$

Where μ means the on-policy distribution under π. The proof for episodic case can be found in (SUTTON; BARTO, 1998). This result is very useful for model-free RL methods.

To apply Policy Gradient Theorem, we need a performance measure $J(\theta)$. In a first moment, let us consider the value function over an episode under policy π. Let τ denote a sequence of states and actions in an episode of length H under policy π. Thus:

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{H} R_t(s,a) \right] = \sum_\tau p(\tau, \boldsymbol{\theta}) R(\tau). \tag{5.40}$$

For estimating this value, we can use the **importance sampling method** (JIE; ABBEEL, 2010) from statistics as "surrogate loss". This technique works as a variance reduction method where we pick samples from a parameter distribution (denoted here as $\boldsymbol{\theta}_{old}$), but computes the performance measure within respect to a distribution θ applying a "importance correction" to the reward:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \boldsymbol{\theta}_{old}} \left[\frac{p(\tau \mid \boldsymbol{\theta})}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau) \right]. \tag{5.41}$$

Applying the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \boldsymbol{\theta}_{old}} \left[\frac{\nabla_{\boldsymbol{\theta}} p(\tau \mid \boldsymbol{\theta})}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau) \right]. \tag{5.42}$$

When $\boldsymbol{\theta} = \boldsymbol{\theta}_{old}$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{old}} = \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\frac{\nabla_{\boldsymbol{\theta}} p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{old}}}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau)\right] \tag{5.43}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}[\nabla_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}_{old}} R(\tau)]. \tag{5.44}$$

Decomposing this last result in terms of states and actions:

$$\mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}[\nabla_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}_{old}} R(\tau)] = \tag{5.45}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\nabla_{\boldsymbol{\theta}} \log \Big[\prod_{t=0}^{H} p(s_{t+1}|s_t, a_t)\pi_{\boldsymbol{\theta}}(a_t|s_t)\Big] R(\tau)\right] \tag{5.46}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\nabla_{\boldsymbol{\theta}} \Big[\sum_{t=0}^{H} \log p(s_{t+1}|s_t, a_t) + \sum_{t=0}^{H} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\Big] R(\tau)\right] \tag{5.47}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) R(\tau)\right]. \tag{5.48}$$

From Equation (5.48), we finally conclude that computing the gradient from the surrogate loss based on importance sampling estimate we have the same result of Policy Gradient Theorem – which means we have a unbiased estimate of gradient that do not require the dynamics model.

However, even with importance sampling, the gradient estimate is very noisy. We will reduce this variance by using three fixes: using a *Critic*, computing the advantage function instead of value function, and changing the surrogate loss to consider KL-divergence trust region.

5.6.2 Actor-Critic Models

During section 5.5, we focused on value-based methods, i.e, we chose actions accordingly to value function estimations. On the other side, the current section detailed the usage of Policy Gradient Methods, where we parameterize a policy and apply gradient updates directly. The problem is the high variance, as described in the last subsection.

It is possible to join both ideas in order to reduce variance for Policy Gradient Methods.

We use a *Critic* to estimate the action-value function:

$$Q(s, a, \boldsymbol{w}) \approx Q_{\pi_{\boldsymbol{\theta}}}(s, a). \tag{5.49}$$

Where $\boldsymbol{w}$ represents the parameters of action-value function. Therefore, there are two entities in a Actor-Critic model:

- **Critic** updates action-value function, with parameters $\boldsymbol{w}$; and
- **Actor** updates policy parameters $\boldsymbol{\theta}$, in direction suggested by critic.

In this context, Actor-Critic methods apply a generalization of Policy Iteration algorithm, illustrated by Figure 5.5: while the Critic solves policy evaluation, Actor executes policy improvement.

Furthermore, with the usage of Critic estimating action-value function, Equation (5.48) replaces instantaneous for long-term action-value, which also reduces variance:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_t \left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \right]. \tag{5.50}$$

Figure 5.6 illustrates how Actor-Critic works.

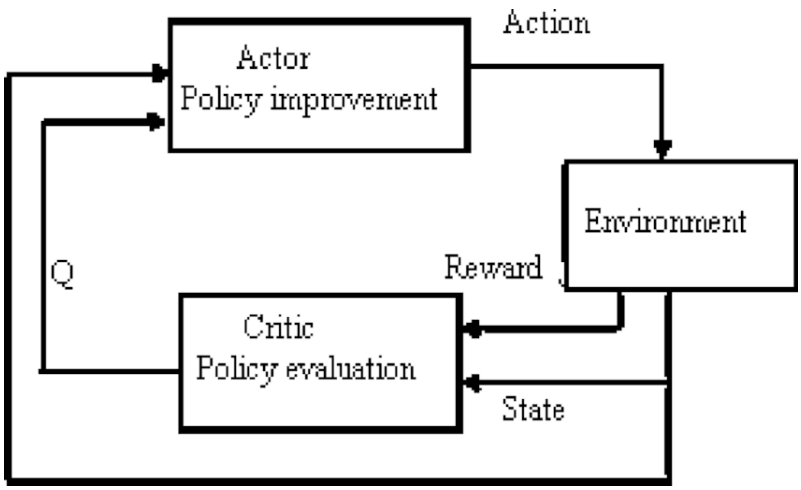


FIGURE 5.6 – Actor-Critic model (XU *et al.*, 2007).

5.6.3 Advantage Function and GAE Algorithm

Another idea aiming to reduce variance in Policy Gradient methods is to estimate the Advantage Function instead of action-value. By definition:

$$A_{\pi_{\boldsymbol{\theta}}}(s,a) = Q_{\pi_{\boldsymbol{\theta}}}(s,a) - V_{\pi_{\boldsymbol{\theta}}}(s). \tag{5.51}$$

Using Advantage Function, we estimate both action-value and value-function values. Thus, the performance measure given by Equation (5.48) changes again:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_t \left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) A_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \right]. \tag{5.52}$$

In practice, in order to estimate Advantage function, it is possible to use Temporal-Difference error:

$$\delta^t_{\pi_{\boldsymbol{\theta}}} = R_{s_t s_{t+1}} + \gamma V_v(s_{t+1}) - V_v(s_t). \tag{5.53}$$

Where $V_{\boldsymbol{v}}$ is a estimation of $V_{\pi_{\boldsymbol{\theta}}}$, parameterized by v. This is possible because Temporal-Difference error estimates Advantage function without bias when $V_v = V_{\pi_{\boldsymbol{\theta}}}(s)$:

$$\begin{aligned}
\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\delta^t_{\pi_{\boldsymbol{\theta}}} \mid s_t, a] &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[R_{s_t s_{t+1}} + \gamma V_{\pi_{\boldsymbol{\theta}}}(s_{t+1})] - V_{\pi_{\boldsymbol{\theta}}}(s_t) && (5.54)\\
&= Q_{\pi_{\boldsymbol{\theta}}}(s_t, a) - V_{\pi_{\boldsymbol{\theta}}}(s_t) && (5.55)\\
&= A_{\pi_{\boldsymbol{\theta}}}(s_t, a). && (5.56)
\end{aligned}$$

However, we cannot guarantee the condition $V_v = V_{\pi_{\boldsymbol{\theta}}}(s)$. Thus, this estimation induce bias. To solve this problem, let's define the Advantage Estimator in terms of discounted Temporal-Difference errors of higher orders, e.g:

$$\begin{aligned}
A^{(1)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t}. && (5.57)\\
A^{(2)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t} + \gamma \delta_{\pi_{\boldsymbol{\theta}},t+1}. && (5.58)\\
A^{(3)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t} + \gamma \delta_{\pi_{\boldsymbol{\theta}},t+1} + \gamma^2 \delta_{\pi_{\boldsymbol{\theta}},t+2}. && (5.59)
\end{aligned}$$

The Generalized Advantage Estimator GAE is defined as the exponentially-weighted average of these estimators (SCHULMAN *et al.*, 2015b). In mathematical terms:

$$A^{GAE(\gamma,\lambda)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) = (1-\lambda)(A^{(1)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \lambda A^{(2)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \lambda^2 A^{(3)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \dots) \tag{5.60}$$

$$= \sum_{t=0}^{\infty} (\gamma\lambda)^t \delta_{\pi_{\boldsymbol{\theta}},t+1} \tag{5.61}$$

The steps from Equation (5.60) to (5.61) can be found in (SCHULMAN *et al.*, 2015b). The λ term refers to trace decay parameter, which define how many steps will be considered in estimation. GAE with $\lambda = 0$ is the case of Equation (5.53) and, as mentioned earlier, can induce bias. On the other side, GAE with $\lambda = 1$ does not induce bias regardless of the accuracy of $V_v(s)$, but has high variance. Therefore, GAE with $0 \leq \lambda \leq 1$ makes a compromise between bias and variance, controlled by λ (SCHULMAN *et al.*, 2015b).

In this work, we use a Advantage Actor-Critic model, where this Advantage function is estimated using GAE. In next section, we finalize Reinforcement Learning Background by adding the last piece in the optimization algorithm to reduce variance: consider KL-divergence trust region.

5.7 Advanced Policy Gradient Methods

There is a third method to reduce variance in parameterized policies: consider it as an optimization problem.

Much of modern ML reduce learning to numerical optimization problem. For example, in supervised learning, the object is to minimize training error (SCHULMAN, 2017).

So, in this section we will present modern policy gradient techniques that represents learning as an optimization problem that allows small updates in the policy using the data sampled.

5.7.1 Optimization Loss

In Equation (5.41), we used importance sampling weighted by model probability distributions and derived the Policy Gradient Theorem result. Then, modified the "target function" from instantaneous reward to advantage estimation.

However, if we consider Equation (5.52) again, we may derive the same gradient from the following loss:

$$\mathcal{L}^{PG}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big], \tag{5.62}$$

and, equivalently, we can differentiate the same gradient from this following loss, where we apply importance sampling to policy distribution:

$$\mathcal{L}^{PG}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big]. \tag{5.63}$$

Schulman *et al.* (2015a) and Kakade e Langford (2002) analyze the usage of Equation (5.63) as surrogate loss and concluded that in practice the results are not much different if compared to Equation (5.62), for small policy changes.

The problem of Equation (5.63) is because it leads to destructively large policy updates. Thus, we need a way to ensure these step sizes are small enough so a bad update does not collapse policy.

5.7.2 Trust Region Policy Optimization – TRPO

From Equation (5.63), the first method to avoid large policy updates is to define a trust region update and reduce learning to a optimization problem:

$$\begin{aligned} &\max_{\boldsymbol{\theta}} \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big], \\ &\text{subject to } \mathbb{E}_t[KL[\pi_{\boldsymbol{\theta}_{old}}, \pi_{\boldsymbol{\theta}}]] \leq \delta. \end{aligned} \tag{5.64}$$

This is a constrained optimization problem. The constrain here is a **Kullback-Leibler divergence**, which measures how a probability distribution is different from a second. In this case, we ensure the value of this divergence is small. Therefore, the policy change is not big, avoiding steps too far away which could collapse learning.

Finally, based on what have been shown in Section 5.6 and this one, we present the TRPO algorithm pseudocode.

The technique proposed to efficiently approximately solve this optimization problem can be found in (SCHULMAN *et al.*, 2015a) and is based on conjugate gradient method (HESTENES; STIEFEL, 1952).

TRPO is a scalable method with strong theoretical foundations, being able to learn

Algorithm 5 TRPO Algorithm

while stopping criteria not met **do**
Run policy π for H timesteps
Estimate Advantage Function at all timesteps
Solve the numerical optimization problem:

$$\max_{\boldsymbol{\theta}} \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big]$$
$$\text{subject to } \mathbb{E}_t[KL[\pi_{\boldsymbol{\theta}_{old}}, \pi_{\boldsymbol{\theta}}]] \leq \delta$$

end while

controllers for robotics locomotion from scratch for continuous tasks, using a generic policy search method and general purpose policy representation due to its results in classical benchmarking environments.

However, as a constrained optimization problem, it is difficult to implement a solver method based on conjugate gradients. Additionally, it is not compatible with architectures that include noise or parameter sharing.

5.7.3 Proximal Policy Optimization – PPO

The Proximal Policy Optimization (PPO) algorithm have some of the benefits of TRPO, but it is much simpler to implement, more general, and have better sample complexity empirically (SCHULMAN *et al.*, 2017).

5.7.3.1 Clipped Surrogate Loss

As we saw, the trust region method ensures small policy updates by constraining the problem. However, it is really hard to solve the underlying optimization problem efficiently.

What PPO does is to try to ensure small changes by clipping the surrogate loss, penalizing big steps. Let us define:

$$r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)}. \tag{5.65}$$

The clipped surrogate loss is defined as:

$$\mathcal{L}^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\min(r_t(\boldsymbol{\theta})A_{\pi_{\boldsymbol{\theta}}}, clip(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)A_{\pi_{\boldsymbol{\theta}}})\Big]. \tag{5.66}$$

The Equation (5.66) modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving $r_t(\boldsymbol{\theta})$ outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, it takes the minimum of clipped and unclipped objective, so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective. With this scheme, it only ignores the change in probability ratio when it would make the objective improve, and we include it when it makes the objective worse (SCHULMAN *et al.*, 2017).

Therefore, PPO tries the same idea of delimiting a small region for updates but treats the problem as an unconstrained optimization – in a way that is possible to apply stochastic gradient descent again.

5.7.3.2 PPO Algorithm

Finally, let us join everything described in this chapter to detail the PPO algorithm used in this work. First of all, we use an Actor-Critic model, where there are two deep neural networks as parameterized policy and value-function. These neural networks are trained using jointly and using the following loss function:

$$\mathcal{L}^{CLIP+VF+S}(\boldsymbol{\theta}) = \mathbb{E}_t[\mathcal{L}^{CLIP}(\boldsymbol{\theta}) - c_1\mathcal{L}^{VF}(\boldsymbol{\theta}) + c_2 S[\pi_{\boldsymbol{\theta}}](s_t)]. \tag{5.67}$$

$\mathcal{L}^{CLIP}(\boldsymbol{\theta})$ denotes the clipped surrogate loss detailed earlier. The second term, $\mathcal{L}^{VF}(\boldsymbol{\theta})$ is the squared loss from prediction of value function and its value computed by the samples collected. Thus, this term aims to train the Critic network. The third term, denotes an entropy term to ensure sufficient exploration (MNIH *et al.*, 2016).

In order to estimate the advantage function of $\mathcal{L}^{CLIP}(\boldsymbol{\theta})$ term, we use the GAE algorithm, as explained in Subsection 5.6.3. We use a truncated version which consider a T-length trajectory-segment, i.e, we collect T timesteps from an episode and apply Equation (5.61) in a finite sum.

Lastly, the algorithm runs as a generalized policy iteration technique: we evaluate the policy by using the data collected and perform improvement applying optimization updates from Equation (5.67) using the Adam algorithm. Algorithm 6 shows the pseudocode.

Algorithm 6 PPO Algorithm

```
while stopping criteria not met do
    foreach agent do
        Run policy π_{θ_old} in environment for T timesteps
        Estimate Advantage Function at all timesteps using truncated GAE algorithm
    end for
    Apply ∇L^{CLIP+VF+S}(θ) to networks using Adam algorithm
    θ_old ← θ
end while
```

6 Meta-Learning

The ability to learn quickly is a key characteristic that distinguishes human intelligence from its artificial counterpart. Humans effectively utilize prior knowledge and experiences to learn new skills quickly. However, artificial learners trained with traditional supervised learning or reinforcement learning methods generally perform poorly when only a small amount of data is available or when they need to adapt to a changing task (MISHRA *et al.*, 2017). Furthermore, although RL has achieved impressive results and surpassed human performance in many tasks, they have high sample complexity, requiring much more data if compared to humans.

The main hypothesis to explain this difference is because humans have a good prior experience to learn a new task, either by the skills learning throughout life and by information encoded in the brain and genetic material. On the other side, deep RL agents need to rebuild knowledge from scratch (DUAN *et al.*, 2016).

Meta-Learning is an area inside machine learning whose objective is to solve these problems by learning how to learn. In this perspective, rather than learn from a single dataset, meta-learning considers as scope a distribution over interrelated tasks and tries to improve generalization across them. Therefore, the learning process of the model itself is seen as an objective of meta-learning algorithms.

In recent works, meta-learning has two main applications. In the supervised learning setup, this technique is related to Few-Shot Learning, a problem where the classifier or regressor needs to predict a label based on a very small amount of data. In RL context, meta-learning is used to train agents in order to generalize across tasks that have closely related goals.

As described in 2.4, there are several formulations to address the problem of meta-learning. In this chapter, we will cover two main approaches related to our experiments. The first, in an architectural level, seeks to find a meta-learner networks which encodes in their weights contextuality or a multi-system learning procedure. The second, in an algorithmic level, seeks to learn a fast adaptation strategy during meta-training.

Before diving into such approaches, we will formulate theoretically the meta-learning setup.

6.1 Meta-Learning Formulation

In meta-learning formulation, there is a distribution over tasks $p(\mathcal{T})$, where each task $\mathcal{T}_i \sim p(\mathcal{T})$ is defined by inputs s_t, outputs a_t, a loss function $\mathcal{L}_i(s_t, a_t)$ and an episode length H_i. The meta-learning model, namely meta-learner, with parameters $\boldsymbol{\theta}$, has the objective to minimize the expected loss of distribution $p(\mathcal{T})$ w.r.t $\boldsymbol{\theta}$:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_i \sim p(\mathcal{T})} \Big[\sum_{t=0}^{H_i} \mathcal{L}_i(s_t, a_t) \Big] \tag{6.1}$$

In the RL context, s_t and a_t are, respectively, the states and actions at each timestep.

The meta-learner is trained in a set of meta-training tasks $\mathcal{T}_{tr}$, solving the optimization problem from Equation (6.1). During test phase, a new set of meta-test tasks $\mathcal{T}_{val}$ were sampled to evaluate generalization.

In RL context, it is possible to formulate meta-learning in terms of Markov Decision Processes, which is closely related to the experiments of this work. Formally, we have a distribution over a set of MDPs $\mathcal{D}$ that we assume knowledge. An agent then optimize a meta-policy $\pi_{\boldsymbol{\theta}}$ by interacting with a set of MDP environments (the tasks), expecting to perform well on average and thus generalize to unseen MDPs sampled from the same distribution $\mathcal{D}$ during testing phase. The objective is averaged across all training MDPs, which reflects the prior that we would like to distill into the agent (DUAN *et al.*, 2016). Figure 6.1 illustrates such formulation. It is also valid in the context of POMDPs.

In the next Sections, we will describe recent and well established meta-learning algorithms.

6.2 Meta-Learning in Architecture Level

In this Section, we will detail few Meta-Learning algorithms which proposes a neural network design, often recurrent-based, aiming to be versatile enough to learn a range of strategies in a variety of domains.

6.2.1 RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning

RL^2 (DUAN *et al.*, 2016) is one of the first algorithms in the Deep RL era that view the learning process of the agent itself as an objective. It structures the policy as a Recurrent Neural Network (RNN) which receives past rewards, actions, and termination flags in

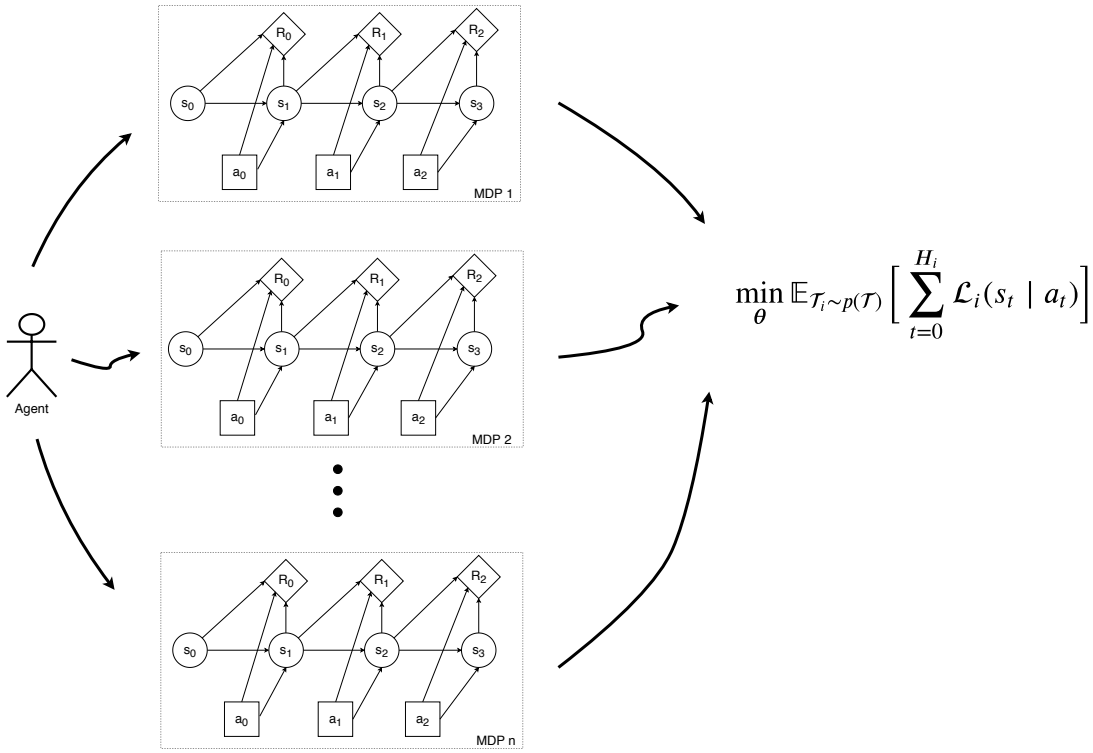


FIGURE 6.1 – Illustration of MDP context for Meta-Learning.

addition to the observed state as inputs. The RNN internal state is preserved across episodes, so that it has the capacity to perform learning in its own hidden activations.

The interaction between agent and environment occurs sequentially: we sample an MDP and perform n episodes (called "trial"). Each episode consists of several cycles of state-action-reward that are collected jointly to episode termination flag. Such data is concatenated as input to the policy, which, conditioned to hidden state h_{t+1}, generates next hidden state h_{t+2} and action a_{t+1}. This hidden state is preserved between episodes, but not between trials. Figure 6.2 illustrates the interaction.

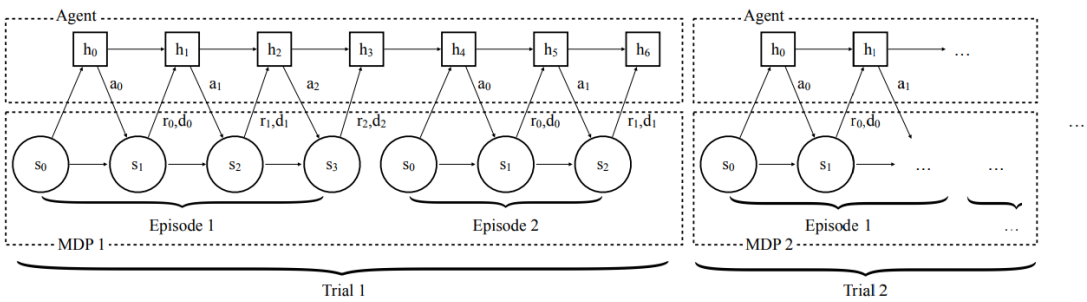


FIGURE 6.2 – RL^2 interaction over MDPs (DUAN *et al.*, 2016).

This problem formulation seeks to maximize cumulative reward throughout a trial, rather than a single episode. The intuition is that since the MDP changes across trials and different strategies are needed for different MDPs, the agent must act accordingly to the belief over the MDP, using all information about past rewards, actions, termination

flags and states to adapt continually – this is like to learn a "fast" RL algorithm.

The policy is represented as a general RNN, where at each timestep the input tuple (s, a, r, d) is embedded using a function $\varphi(s, a, r, d)$ and used as input to the RNN, followed to a dense layer to perform actions. The optimization descried in Duan *et al.* (2016) uses TRPO algorithm with Generalized Advantage Estimation to perform optimization (the "slow" RL procedure).

6.2.2 Simple Neural Attentive Meta-Learner (SNAIL)

The RL^2 has great importance because of how Meta-RL is formulated based on a simple and general RNN architecture. The idea is because RNNs propagate information by keeping it in their hidden state from one timestep to the next. The Simple Neural AttentIve Meta-Learner (SNAIL) (MISHRA *et al.*, 2017) extends the Meta-RL formulation to consider other class of architectures that do not rely on the RNN strategy, because its temporally-linear dependence bottlenecks their capacity to perform sophisticated computation on a stream of inputs. Furthermore, RNN-based architectures are harder to train.

The SNAIL model is based on two main components: temporal convolutions and attention mechanism. In this work, we will superficially describe these components, just to understand the idea behind SNAIL.

Temporal convolutions (OORD *et al.*, 2016) are dilated 1-D convolutions over the temporal dimension. They are causal, thus the generated value at the next timestep are only influenced by past timesteps. Compared to traditional RNNs, they offer more direct, highbandwidth access to past information, allowing them to perform more sophisticated computation over a temporal context of fixed size. However, temporal convolutions does not scale well for long sequences.

The attention mechanism, on the other side, allows the model to pinpoint a specific piece of information from a potentially large context. However, it lacks of positional dependence.

In light of this, temporal convolutions and attention both have problems, but complement each other: while the covolutions provide high-bandwidth access to past information (positional dependence) in a finite context size, the attention mechanism pinpoint access of an infinite large context. The SNAIL architecture interleave both blocks (illustrated in Figure 6.3.)

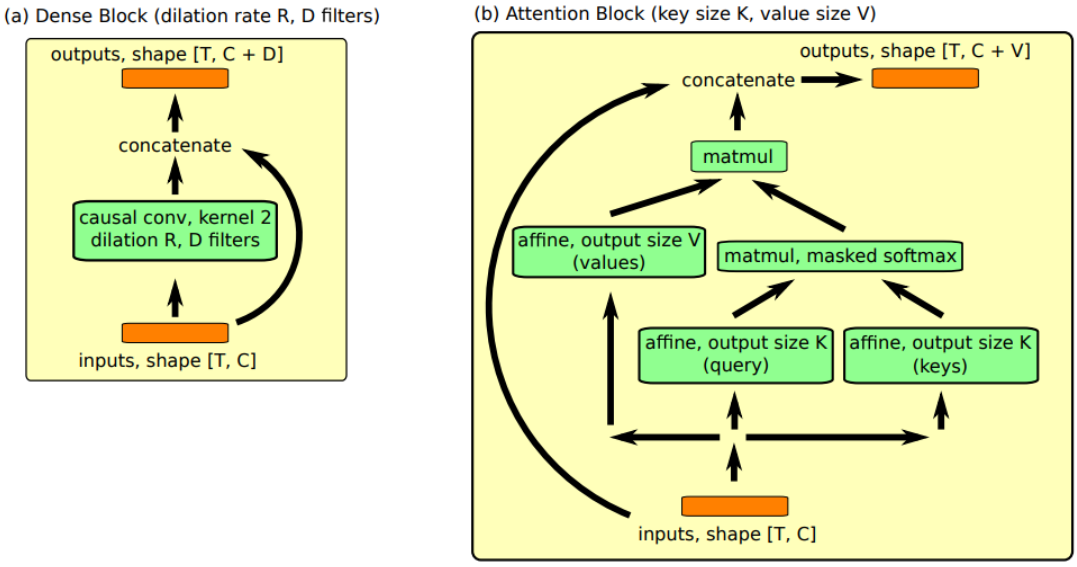


FIGURE 6.3 – Architectural blocks that forms SNAIL (MISHRA *et al.*, 2017).

6.3 Meta-Learning in Algorithmic Level

In the last Section, we explored the Meta-Learning problem in architectural-level, i.e, the learning algorithm is encoded into the weights of a specific network. Those methods are very creative and do not explicitly encodes a particular strategy for the meta-learner. Additionally, the optimization procedure remains the same. A drawback of the architecture level is because they place constraints on the model (which can not work well in a particular domain and is limited to neural network representations) and – which we consider as the main problem – needs reward as input to the model. In this context, we need access to reward after deployment, which can be impossible.

In this Section, we explore another viewpoint of the Meta-Learning problem, where rather than proposing a model architecture, we change the optimization procedure to find a set of parameters $\boldsymbol{\theta}^*$ that can quickly adapts to unseen tasks via gradient descent.

6.3.1 MAML: Model-Agnostic Meta-Learning

Model-Agnostic Meta-Learning (MAML) (FINN *et al.*, 2017) is a method to learn parameters of any standard model and prepare them for fast adaptation. The idea is to find model parameters that are sensitive to changes in the task, such that small updates in the parameters will produce large improvements on loss function of any task sampled from the distribution over them. These small changes and sensibility are related to a gradient-based learning rule, which is needed to take in consideration in meta-training.

The intuition is that some internal representations are more transferable between tasks than others – a neural network can learn "meta-features" that are more shareable than

other ones applicable just for individual tasks.

Formally, MAML consider a model with parameters $\boldsymbol{\theta}$. When adapting to a new task, the model parameters become $\boldsymbol{\phi}$ by steps of gradient descent:

$$\boldsymbol{\phi} = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}) \tag{6.2}$$

The model parameters are trained for the performance of the model parameterized by $\boldsymbol{\phi}$ w.r.t $\boldsymbol{\theta}$ across tasks sampled from the distribution over them. Therefore, the meta-objective is mathematically described as:

$$\min_{\boldsymbol{\theta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\phi}) = \min_{\boldsymbol{\theta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta})) \tag{6.3}$$

In RL context, the loss is given by the negative of cumulative reward's expectation:

$$\mathcal{L}_{\mathcal{T}_i} = -\mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \Big[\sum_{i}^{H} R_i(s_t, a_t) \Big]. \tag{6.4}$$

The meta-optimization across tasks is performed via stochastic gradient descent using the meta-objective from Equation (6.3). The MAML thus involve meta-gradient updates (gradient through gradients), which involves second-order derivatives. MAML for Reinforcement Learning is shown in Algorithm 7.

Algorithm 7 MAML for RL (FINN *et al.*, 2017)

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize $\boldsymbol{\theta}$
2: **while** not done **do**
3: Sample batch of meta-training tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **foreach** meta-training task $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D}_{\mathcal{T}_i} = \{(s_1, a_1, ..., a_H)\}$ using $\pi_{\boldsymbol{\theta}}$ in $\mathcal{T}_i$
6: Evaluate $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta})$ using $\mathcal{D}_{\mathcal{T}_i}$ in Equation (6.2)
7: Update parameters with gradient descent $\boldsymbol{\phi} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta})$
8: Sample K trajectories $\mathcal{D}'_{\mathcal{T}_i} = \{(s_1, a_1, ..., a_H)\}$ using $\pi_{\boldsymbol{\phi}}$ in $\mathcal{T}_i$
9: **end for**
10: Update parameters with gradient descent $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \beta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\phi})$ using each $\mathcal{D}'_{\mathcal{T}_i}$ and $\mathcal{L}_{\mathcal{T}_i}$ from Equation (6.4)
11: **end while**

6.3.2 First Order Meta-Learning

As described in previous Subsection, the MAML algorithm relies on second-order derivatives, which is expensive computationally because we need to deal with Hessian matrices. To address this, there is a cheaper implementation which omits second derivatives, called First Order MAML (FOMAML) (NICHOL *et al.*, 2018).

Basically, if we consider k inner gradient steps, we have the following updates

$$\boldsymbol{\theta_0} = \boldsymbol{\theta_{meta}} \tag{6.5}$$

$$\boldsymbol{\theta_1} = \boldsymbol{\theta_0} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}_0) \tag{6.6}$$

$$\boldsymbol{\theta_2} = \boldsymbol{\theta_1} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}_1) \tag{6.7}$$

$$\dots \tag{6.8}$$

$$\boldsymbol{\theta_k} = \boldsymbol{\theta_{k-1}} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}_{k-1}) \tag{6.9}$$

The learning rule for outer optimization is:

$$\boldsymbol{\theta_{meta}} = \boldsymbol{\theta_{meta}} - \beta \nabla_{\boldsymbol{\theta}} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \tag{6.10}$$

In the outer optimization, the gradient step can be written as:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) = \tag{6.11}$$

$$= \nabla_{\boldsymbol{\theta}_k} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \cdot \nabla_{\boldsymbol{\theta}_{k-1}} \boldsymbol{\theta}_k \dots \nabla_{\boldsymbol{\theta}_0} \boldsymbol{\theta}_1 \cdot \nabla_{\boldsymbol{\theta}} \boldsymbol{\theta}_0 \tag{6.12}$$

$$= \nabla_{\boldsymbol{\theta}_k} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \cdot \prod_{j=1}^{k} \nabla_{\boldsymbol{\theta}_{j-1}}(\boldsymbol{\theta}_j) \tag{6.13}$$

$$= \nabla_{\boldsymbol{\theta}_k} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \cdot \prod_{j=1}^{k} \nabla_{\boldsymbol{\theta}_{j-1}}(\boldsymbol{\theta}_{j-1} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}_{j-1})) \tag{6.14}$$

$$= \nabla_{\boldsymbol{\theta}_k} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \cdot \prod_{j=1}^{k} (\boldsymbol{I} - \alpha \nabla_{\boldsymbol{\theta}_{j-1}}(\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(\boldsymbol{\theta}_{j-1}))) \tag{6.15}$$

If we just ignore second-order derivatives, the final gradient is:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \approx \nabla_{\boldsymbol{\theta}_k} \mathcal{L}'_{\mathcal{T}_i}(\boldsymbol{\theta}_k) \tag{6.16}$$

Which is less expensive computationally and easier to calculate – we just need the new

loss w.r.t to parameters after inner learning steps. The FOMAML algorithm has comparable results to standard MAML (NICHOL *et al.*, 2018), but its performance is generally worse because the lack of information provided by second-order derivatives.

The last algorithm described here is Reptile, which is also presented in the same work of FOMAML. The idea of Reptile is to use higher-order derivative information (not just second-order), but actually computing just first-order derivatives. Algorithm 8 presents the idea.

Algorithm 8 Reptile (NICHOL *et al.*, 2018)

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize $\boldsymbol{\theta}$
2: **while** not done **do**
3: Sample meta-training task $\mathcal{T}_i \sim p(\mathcal{T})$
4: Perform $k > 1$ steps of stochastic gradient descent on task $\mathcal{T}_i$, resulting in parameters $\boldsymbol{W}$
5: Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta(\boldsymbol{W} - \boldsymbol{\theta})$
6: **end while**

The optimization assumption and intuition is that by performing gradient w.r.t model parameters but different tasks we will learn a representation that is near to the optimal solution of all tasks. Mathematically, we assume that task $\mathcal{T}_i \sim p(\mathcal{T})$ has a manifold of optimal policy, $\mathcal{W}^*_{\mathcal{T}}$. The optimization will find a parameter set that is close to all optimal manifolds of all tasks:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{T}_i \sim p(\mathcal{T})}\Big[\frac{1}{2} d(\boldsymbol{\theta}, \mathcal{W}^*_{\mathcal{T}})\Big], \tag{6.17}$$

where $d(x, y)$ is a distance metric between x and y. Using the squared euclidean distance, we can compute the gradient as):

$$\nabla_{\boldsymbol{\theta}}\Big[\frac{1}{2} d(\boldsymbol{\theta}, \mathcal{W}^*_{\mathcal{T}})^2\Big] = \tag{6.18}$$

$$= \nabla_{\boldsymbol{\theta}}\Big[\frac{1}{2} d(\boldsymbol{\theta} - \boldsymbol{W}^*_{\mathcal{T}})^2\Big] \tag{6.19}$$

$$= \boldsymbol{\theta} - \boldsymbol{W}^*_{\mathcal{T}}. \tag{6.20}$$

In the last equation, we assumed that the distance from parameters to optimal configuration is equal to the distance from such parameters to the optimal manifold closes to $\boldsymbol{\theta}$. As result, we have the update rule used in Reptile. Figure 6.4 illustrates the intuition

about this optimization assumption.

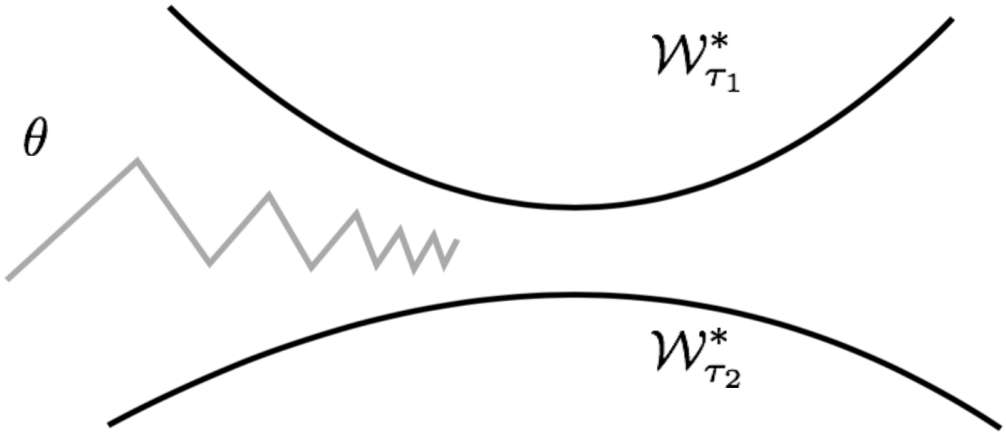


FIGURE 6.4 – Reptile optimization procedure tends to converge to a region that is closest to optimal solution for each task (NICHOL *et al.*, 2018).

Figure 6.5 has an illustrative comparison between FOMAML and Reptile updates. As we can observe, FOMAML basically uses information from the last inner gradient step, while Reptile uses information from all update steps (i.e, consider high-order derivative information that comes from older steps).

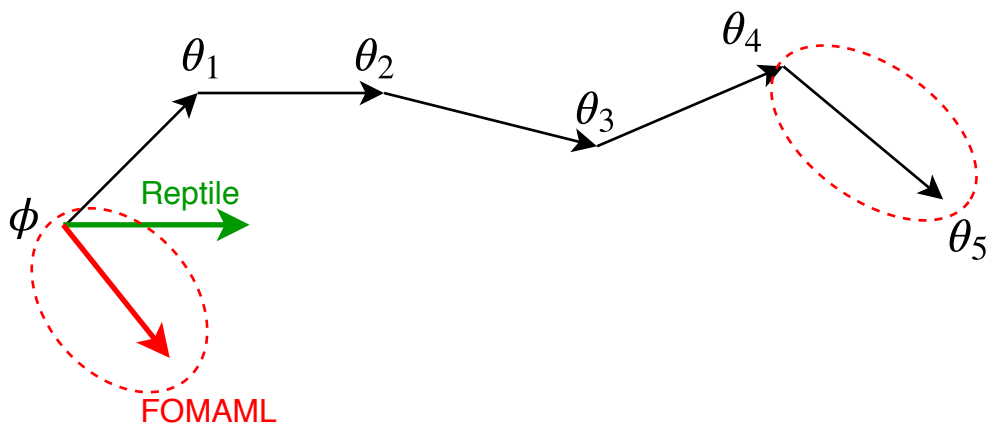


FIGURE 6.5 – FOMAML vs Reptile: the latter has a more precise gradient because it uses high-order derivative information (LEE, 2018).

We will limit our explanation about first-order methods on Meta-Learning here, for the sake of conciseness. For further details (including Taylor series expansions to analyze gradient information) we refer to Nichol *et al.* (2018).

7 The Imitation Hypothesis

In previous chapters, we presented the motivation of this work, the domain description and the review from literature of Reinforcement Learning algorithms for control tasks. Posteriorly, we described in detail the concepts of Deep Learning and Reinforcement Learning. It culminates in the algorithm used for controlling the kick motion in this work: Proximal Policy Optimization. In this chapter, we will describe the problem, the whole experimentation setup and the computational infrastructure used to conduct experiments regarding the Imitation Hypothesis.

7.1 The Kick Motion Problem

First of all, we need to understand why the kick motion is a hard task for learning models, which justifies the use of scalable techniques such as Deep Reinforcement Learning. The main reasons are:

- **Long Time Horizons** – The simulation server runs 50 cycles per second. Each action has minor impact individually, but can collapse the motion in the future. Furthermore, the final objective is to inject velocity in the ball and it happens only after the whole motion. So, the real reward is very delayed;
- **Partially-observed state** – The agent doesn't have full information from the environment and the perception data is noisy. Therefore, all information used in training is liable to have errors that can make learning harder; and
- **High-dimensional, continuous action space** – In the kick motion, at each timestep, the agent have to act in each of 22 joints. Each joint can assume any possible angle. If we consider the agent could assume only two possible actions for each joint, then it has $2^{22} = 4.194.304$ possible combinations of actions. Just to compare, the average number of actions: in chess is 35, and in Go, 250.

7.2 Experimentation Setup

In order to find the best method for learning the Kick Motion, we tested several techniques to compare them and find out that one which converges faster and results in the better and more robust kick. In the next subsections, we will describe each of the methods evaluated in this work.

7.2.1 Reinforcement with Naive Reward - RNR

In this learning model, we just use reinforcement learning with a simple and directly reward. As our task is related to kicking the ball, the "naive" reward is composed by its velocities after the kick motion, as shown in Equation (7.1), where s, v, and u are the actual state, the vector of velocity, and weight parameters, respectively. We call it naive because we do not pass to the learning model any idea of how the motion has to be performed - just what we expect to have as final objective.

It is important to highlight that the techniques described in Section 7.2 could be used jointly as building blocks of a more labored model.

$$R(s) = u^T v. \tag{7.1}$$

7.2.2 Reinforcement with Reference Reward - RRR

In this learning model, we add a term that compares the actual performance with a reference motion, as shown in Equation (7.2):

$$R(s) = w_{ref}^T(\pi - r) \tag{7.2}$$

In this equation, w_{ref} are weight parameters for each joint and r is the vector of absolute values regarding the difference between joint values in that state and the reference (Equation (7.3)).

$$r(\theta, \theta_0) = |\theta - \theta_0|. \tag{7.3}$$

The intuition behind this weight parameters is because some joints are more important than others for kicking. For example, some joints from the leg that kicks the ball themselves could collapse the whole motion if the error is high. On the other hand, joints from the neck are not so important for the kick. Therefore, it is natural to penalize differently in each case.

Lastly, the reference motion used was the keyframe kick we have access from ITAndroids code, previously mentioned in Subsection 3.3 and also used in HLM.

7.2.3 Reinforcement with Initial State Distribution - RISD

The Initial State Distribution is a technique related to where the episode starts to collect data to use in the learning. Without this technique, all episodes start in the beginning of the motion. However, the problem of this strategy is because the policy is forced to learn the motion in a sequential manner, first learning the early phases of the motion, and then incrementally progressing towards the later phases. This can be problematic for the kick motion. Another disadvantage of a fixed initial state is the resulting exploration challenge. The policy only receives reward retrospectively, once it has visited a state. Therefore, until a high-reward state has been visited, the policy has no way of learning that this state is favorable (PENG *et al.*, 2018).

To implement Initial State Distribution, we used a uniformly random variable that could assume any value in the set of states from the reference motion previously described. Suppose we pick the value s. Then, we start the episode running the reference motion, from its first state until s. Finally, we start to collect data and perform the actions from the running policy.

7.2.4 Reinforcement with Early Termination - RET

The Early Termination technique is related to where the episode ends. Without early termination, all motions stops in the same state, which means that the episode length is fixed. The problem of this is because if the motion fails somehow during its execution, all the data collected after the failure will harm learning and evaluation of later stages.

In the kick motion task, the early termination will be triggered when the robot falls during the motion. This behavior will ensure two things: first, that we do not collect wrong data, as mentioned before; and second, to gain more reward and have longer episodes, the agent will be forced to learn a kick motion that does not fall – which will be very good in game situation.

7.2.5 Hybrid Learning Model – HLM

In the Hybrid Learning Model (HLM), the training process occurs in two phases. The first one, a supervised learning phase, we learn the keyframe used by using a neural network, reproducing almost the same motion that we already had previously. The intuition behind this is if the agent starts from a good initial point in the optimization problem,

it could be easier to get better results applying the gradient in the neighborhood of that point. The reward starts with a good value and the starting motion is well defined. Otherwise, if the optimization starts in a random point, it could be impossible to reach a good motion and probably the agent will get stuck in a local optima. This idea is shown in Figure 7.1.

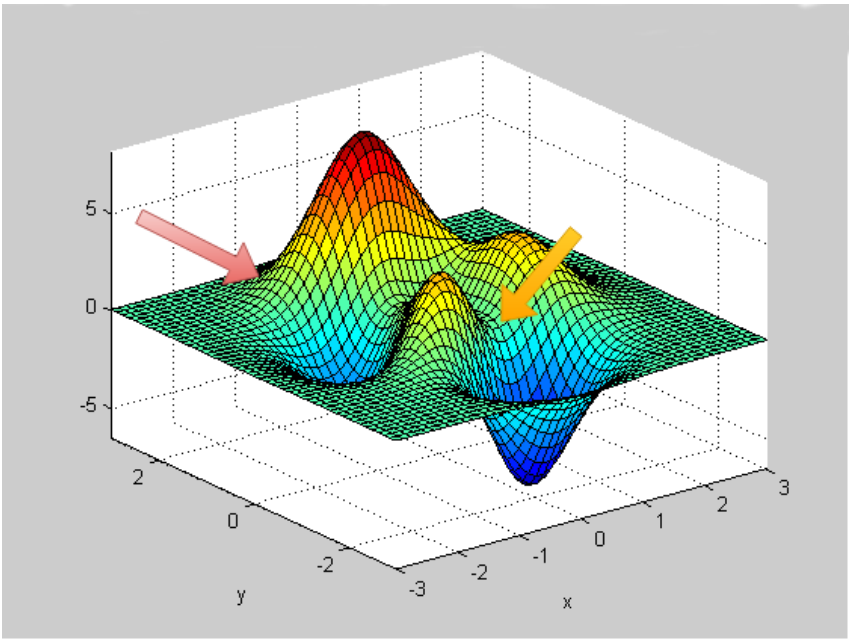


FIGURE 7.1 – In the hybrid model, we can ensure the starting point is near of the optimal solution (orange arrow); otherwise, the starting point can be bad and harder to optimize (red arrow).

7.3 Supervised Learning Setup

In this section, we describe the methodology used in Supervised Learning experiments, which includes the dataset description and collection, the neural network architecture and hyperparameters, the training procedure, and the model deployment in Soccer 3D environment.

7.3.1 The Dataset

In order to use supervised learning for learning keyframe motions with neural networks in the HLM model, we first need to construct a dataset. A dataset consists of samples of keyframe steps. Samples were collected within the Soccer 3D environment with a frequency of 50 Hz. We acquired these samples in two different ways.

In the first one, we commanded an agent of our team to execute specific motions and sampled the reference joint positions computed by our code. In this case, we sampled the

kick and get up keyframe motions (MUNIZ *et al.*, 2016). Notice that, for this approach to be successful, one needs access to the source-code.

The second approach involved changing the Soccer 3D server source-code to provide current joint positions of a given robot, in a similar way as described in (MACALPINE *et al.*, 2013). This allowed us to acquire motion datasets from other teams, without any knowledge of how these movements are implemented. In this case, we collected two types of kicks based on keyframes and sampled joint values of the walking engine (MACALPINE *et al.*, 2012b).

7.3.2 Neural Network Architecture and Hyperparameters

The neural network has to be able to learn how to interpolate between samples, which actually happens. The architecture that performed best – in terms of mean absolute error minimization and simplicity – is shown in Figure 7.2. A deep neural network with 2 hidden, fully connected layers of 75 and 50 neurons was used. The output layer has 23 regression neurons, which represent the 22 joint angles and a neuron which output indicates if the motion has ended or not. The neurons in each hidden layer use the LeakyReLU activation function (XU *et al.*, 2015):

$$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

where α is a small constant. This activation function was used to improve the representation capacity of the neural network, adding support for non-linear functions.

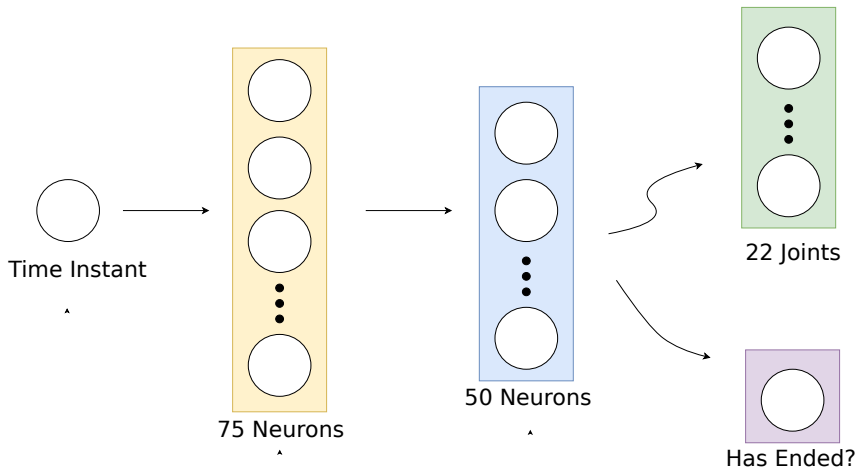


FIGURE 7.2 – The architecture of a neural network designed to learn motions.

This architecture resulted in thousand of parameters to optimize, as exposed in Table 7.1. A very high number, when compared to more traditional optimization approaches (MACALPINE *et al.*, 2012b). Notice that increasing the number of parameters usually allows

representing better movements.

TABLE 7.1 – The Network Summary.

Layer	Neurons	Activation	Parameters
Dense	75	LeakyReLU	150
Dense	50	LeakyReLU	3800
Dense	23	Linear	1173

Total Parameters	5123

7.3.3 The Training Procedure

Since keyframe motions are executed in an open-loop fashion, the sequence of joint positions are always the same for different repetitions, independently of robot's state. Therefore, by adding samples of multiple executions of the same motion would not make our dataset richer. So, we decided to use only one repetition for each movement for faster training. In case of walking motion, we collected samples within one walking period.

During the training, we used 50 thousands epochs divided into 5 training phases, where the learning rate was decreased between phases, in order to achieve better performance. First, we executed 30000 epochs, by using the learning rate of 0.001. The other phases had 5000 epochs each, and we decreased the learning rate by 0.0002 in each phase.

Furthermore, we used Adam optimization (KINGMA; BA, 2014), during the whole training. The loss function used was the mean squared error, as explained in Subsection 4.1. We decided this loss function is adequate for this problem, mainly because it strongly penalizes large errors, which can collapse the whole motion.

7.3.4 The Deployment in the Soccer 3D Environment

In order to perform the network design and the training procedure, we used the Keras (CHOLLET *et al.*, 2015) framework coupled with Tensorflow (ABADI *et al.*, 2015) as backend. After training, the weights were frozen and converted to a specific format, which was readable, by using the Tensorflow C++ API integrated within our agent's code. Hence, the training was performed outside the environment, but the agent actually computes network inferences, during the simulation execution.

7.4 Reinforcement Learning Setup

To use a Reinforcement Learning model, we need to define a policy representation and a task which the agent will follow during the process.

7.4.1 Policy Representation

The policy is represented by a neural network similar to the supervised learning model. A state is described as the stage of kick motion. An action is a set of joint values that has to be applied by the agent in the state of the kicking motion.

In the Supervised Learning stage of HLM, the architecture from both value function and policy networks is that described by Section 7.3. In the Reinforcement Learning stage, we add a Gaussian action space noise. On the other cases, we also add an input normalization filter.

7.4.1.1 Input Normalization Filter

The input normalization filter, also know as feature scaling, has the objective of normalizing the input to avoid the problem of vanishing or exploding gradients by applying updates through all dimensions in equal proportions, as illustrated in Figure 7.3. This normalization happens at each pass of the network, re-calculating mean and standard deviation using the new examples. Hence, this method helps during the convergence.

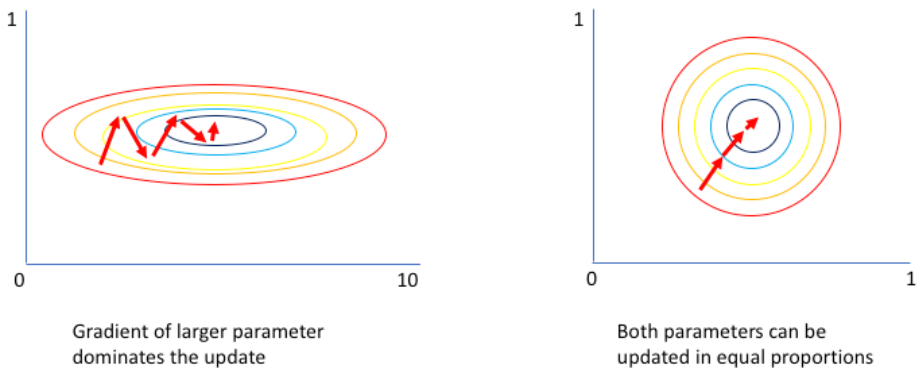


FIGURE 7.3 – Intuition behind input normalization (JORDAN, 2018)

.

7.4.1.2 Neural Network

As an actor-critic model, we have two networks: one for the value function and other for the policy itself. They have the same architecture: two layer, fully connected, with

64 neurons in each hidden layer and hyperbolic tangent as activation function. The architecture is illustrated in Figure 7.4. Table 7.2 summarizes the parameters in this new architecture, also considering the parameters from the Gaussian Action Space Noise, described in 7.4.1.3.

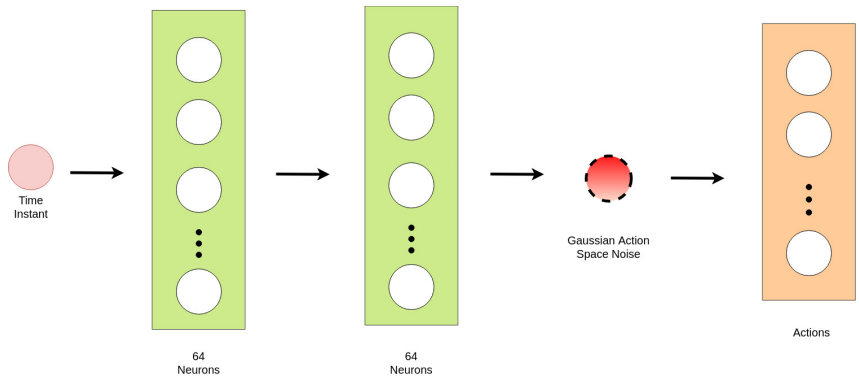


FIGURE 7.4 – Architecture used by pure Reinforcement Learning models.

TABLE 7.2 – The Reinforcement Learning Network Summary.

Layer	**Neurons**	**Activation**	**Parameters**
Dense	64	*tanh*	128
Dense	64	*tanh*	4160
Output	23	Linear	1495
Noise	23	Linear	23

Total Parameters	5806

7.4.1.3 Gaussian Action Space Noise

The last idea explored in this policy representation is the Gaussian action space noise for better exploration. It adds adaptive noise to the action space of the neural network. Discrete RL uses ϵ-greedy (WATKINS, 1989) to confer exploration. Gaussian action space noise injects normal randomness directly into the actions of the agent, altering the types of decisions it makes and helping algorithms explore continuous environments more effectively. Figure 7.5 illustrates this technique.

7.4.2 Task Description

The learning task consists in the activity the agent will execute in order to achieve the expected behavior through the policy. In this experimentation, we would like to learn the kick motion. Therefore, it is straightforward that the scenario is based on kicking the ball.

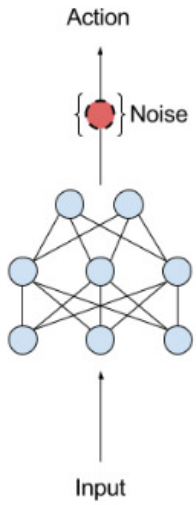


FIGURE 7.5 – Gaussian noise applied to action space to ensure better exploration in continuous environments. (PLAPPERT *et al.*, 2017)

The agent is started in a fixed position of the soccer field. The ball, on the other hand, starts in a position α_{ball} meters from the agent, in front of him. The parameter α_{ball} can be optimized in order to decide the best position the agent has to stop to kick the ball better. However, for this experimentation, we fixed its value.

We repeat the following flow: in the beginning of the episode, all the joints are initialized in a certain value that correspond to the joints the robot assume when it stops to walk and will start the kick motion. Then, we start the learning steps, applying the joints from the neural network to the robot. When we use RISD, there is an intermediate phase executing the reference motion, as described in Subsection 7.2.3.

After the execution of those steps, we finalize the episode turning all the joints to zero. When there is no Early Termination, this happens after a fixed number of steps that we can ensure the motion is over. Otherwise, this occurs when the early termination condition is reached. Figure 7.6 shows the tasking running.

FIGURE 7.6 – Task used to learning kick motion.

7.5 Infrastructure

In this section, we describe in detail the whole infrastructure in which the experiments are conducted. It includes the RL server description, the deployment of models (from supervised learning to RL algorithm and from RL algorithm to soccer agent), the distributed training setup, the evaluation metrics definition, and the tool for monitoring.

7.5.1 Reinforcement Learning Server

Figure 7.7 shows the architecture used to integrate the learning algorithm, agent, and simulation environment. The simulation server exposes a state to the soccer agent, which model this information and passes to the learning algorithm that chooses an action accordingly and returns to the agent. Then, it applies this action in the environment, modifying its state, completing the cycle.

7.5.1.1 Simulation Server

The simulation environment used, as mentioned before, is the Simpark. It uses a TCP Socket to communicate with the agent and exposes an interface for querying the ground truth state for many features from the agent – which can be helpful to state evaluation and avoids to use agent's perceived data that has noise.

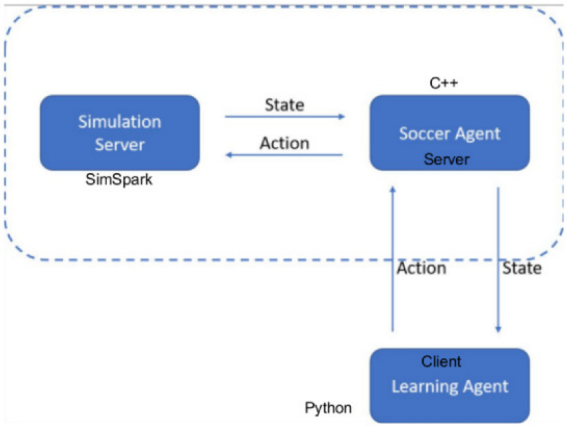


FIGURE 7.7 – Reinforcement Learning Server Architecture (MUZIO, 2017).

7.5.1.2 Soccer Agent

The agent used is from ITAndroids Soccer 3D team and it leads with the perception information sent from the server, model the world with this information and make decisions accordingly. Additionally, there is an entity, called Wizard, whose purpose is querying the ground truth state. All this code is written in C++ and the task described in Section 7.6 was implemented in the agent itself.

Other tool that worth mention is RoboViz, a publicly available visualization tool that allows a few improvements over the default visualization tool. We used it to verify if the task was implemented successfully.

7.5.1.3 Learning Agent

We created another interface to integrate the Soccer Agent with the algorithms used in this experimentation. These were implemented by OpenAI Baselines (DHARIWAL *et al.*, 2017), a repository of high-quality implementations of reinforcement learning algorithms whose purpose is to make easier for the research community to replicate, refine, and identify new ideas, and create good baselines to build research on top of it. The community developed those algorithms using Python language with the Tensorflow (ABADI *et al.*, 2015) framework.

This integration was done using gRPC (GRPC, 2018), an open-source remote procedure call (RPC) system that generates cross-platform client and server bindings. It uses HTTP/2 for transport and Protocol Buffers (VARDA, 2008) as the interface description language, serializing structured data.

7.5.2 Neural Network Deployment

When we use the architecture from Subsection 7.5.1, two problems related to the neural network deployment arise.

The first one is related to the HLM. We have a supervised learning model trained using the framework Keras outside the environment and with no relation to OpenAI Baselines. Therefore, we need a way to import this model from Keras into the learning algorithms.

The second problem is to deploy the network from the learning algorithm to the soccer agent. When the network is trained using OpenAI Baselines, there's no contact between the soccer agent and the model. As described in Subsubsection 7.5.1.3, there is an interface to communicate the actions between them. All calculations happens in only one side. However, during a game, it is not possible to maintain this dependency, so we need a way to run the network inside the soccer agent and without the whole training structure.

7.5.2.1 Importing Supervised Model into OpenAI Baselines

To solve this problem, it is important to understand how Keras works and which is the challenge behind this import.

Keras works on top of lower level machine learning frameworks. Its purpose is to facilitate model prototyping and training using few lines of code. With Keras, developers do not need to understand in deep how computational graphs works and some math behind the models. Actually, this framework wraps code from the lower level ones. Therefore, even being easier to use, all the computation and data structures remains the same.

As the back-end framework that Keras runs on top of, we used Tensorflow. As described previously, it creates a computational graph with math tensors as inputs that "flow" inside and returns the expected result.

Given that context, the idea is straightforward: we create a "Kick Policy" class, that creates the model using Keras, loads its weights, gets the output tensor, and injects in the rest of Tensorflow's computational graph related to the learning algorithm. The Algorithm 9 explains sequentially this idea.

Algorithm 9 Import Keras model into Tensorflow

Require: Keras output model
  $model \leftarrow createKerasModelStructure()$
  $model.load_weights(keras_output_model)$
  $outputTensor \leftarrow model.output$
  Inject $outputTensor$ in the rest of computational graph

It is important to mention that the *load_weights* operation already initializes the graph variables and if we initialize them again, we lose the pre-trained weights.

The biggest problem of this integration is we can't use Input Normalization Filter and Gaussian Action Space Noise, described in Subsubsections 7.4.1.1 and 7.4.1.3, respectively. Therefore, we can face gradient and exploration problems during learning.

7.5.2.2 Exporting OpenAI Baselines onto Soccer Agent

As mentioned earlier, Tensorflow creates a data flow graph with all operations executed during the algorithm. Figure 7.8 shows the whole graph for PPO used in this experimentation. However, we need to export only the policy network itself.

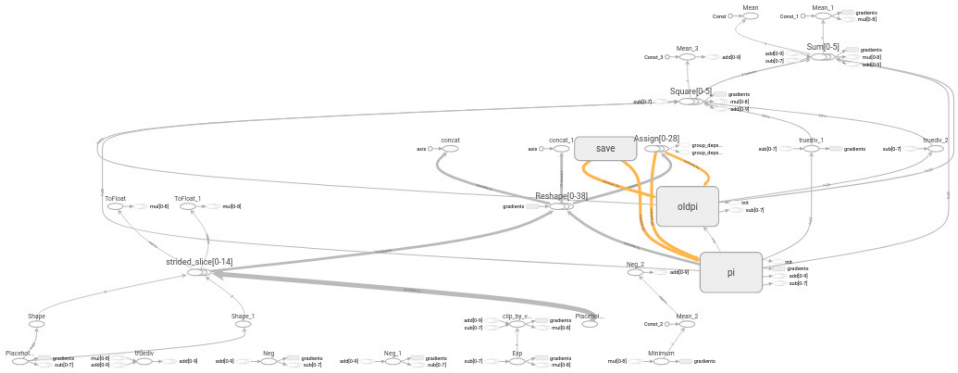


FIGURE 7.8 – Data flow graph for PPO algorithm.

It is important to mention that "policy network" means only the neural network that maps state to actions, and not the whole policy node "pi" in Figure 7.8. The "pi" node also have the value function network and gaussian noise in action space, only needed during the learning process. Figure 7.9 shows the whole "pi" node.

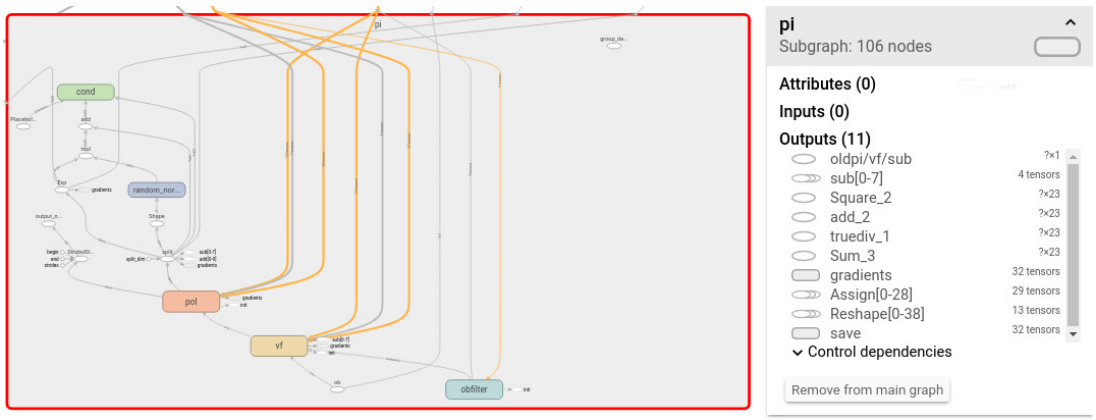


FIGURE 7.9 – "Pi" data flow node from PPO.

Therefore, to export this policy network, we need to add a new "identity" node in the output from that. This node has the same resultant tensor, but it is possible to label

it and we can freeze the subgraph whose output is the labeled node itself. Internally, Tensorflow will start freezing the output node and its dependencies, and, consecutively, until input nodes. Then, we save this data structure in a proto file – a method of serializing structured data – that we read using the C++ API in the soccer agent. Figure 7.10 and 7.11 show the policy data flow graphs from Figures 7.4 and 7.2, respectively.

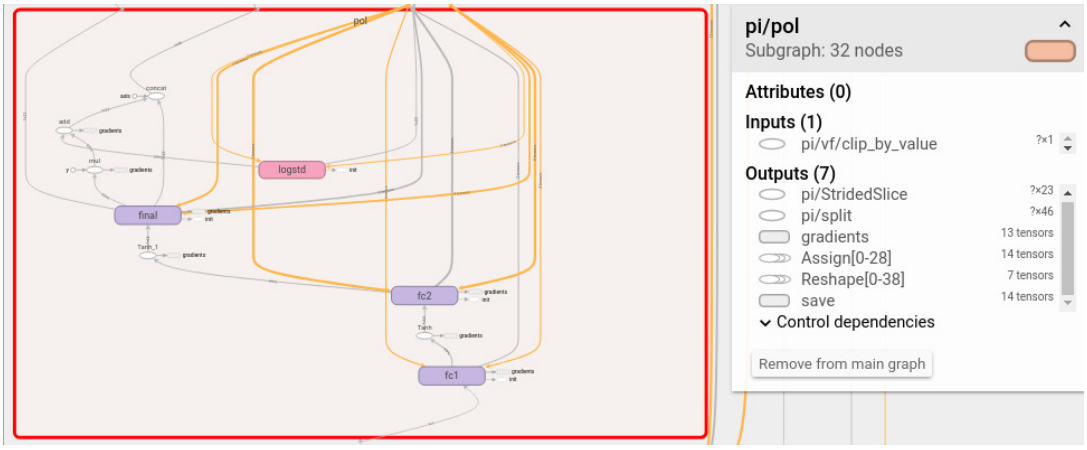


FIGURE 7.10 – Policy data flow graph from Figure 7.4

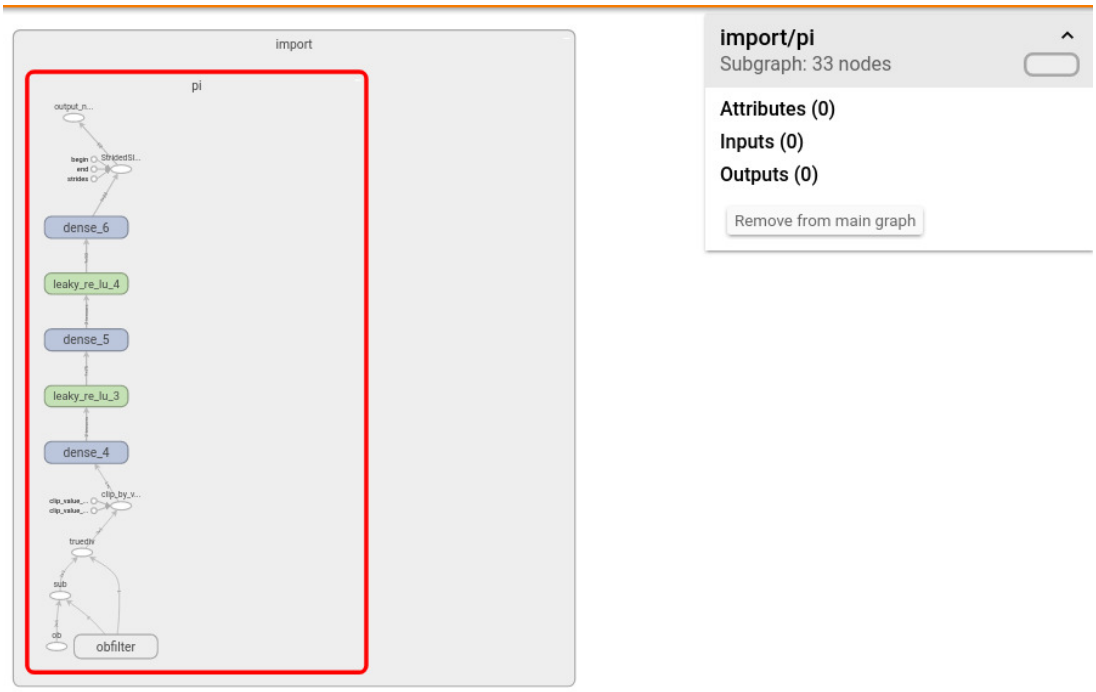


FIGURE 7.11 – Policy data flow graph from Figure 7.2

7.5.3 Distributed Training

Reinforcement Learning training is much more costly computationally when compared to the supervised one described in Section 7.3. Additionally to the costs of applying gradients in the network, we need to gather all the data from agent simulations. Furthermore,

since the data distribution changes during the learning, it is harder to converge, so it needs large amounts of data.

In this context, using a single agent for gathering the data and training the algorithm takes too long to converge and, depending of the problem, it is not viable. We need a way to get more data faster and accelerate the training.

7.5.3.1 Data Parallelism

The solution for this challenge is to distribute the training using several agents. Each agent interacts with its own environment, but the results are applied to the same network. This technique is called Data Parallelism, because the parallelization happens in data gathering. It opposes the idea of Model Parallelism, where the learning model itself is distributed, but this technique is not interesting in reinforcement learning training.

In Data Parallelism, we applied a master-workers architecture, as shown in Figure 7.12. It has several worker nodes, whose job is to gather data by simulating the agent, and a master node, which controls the learning algorithm. This architecture works very well in this experiment because the computation from workers are very similar so we can conduct several independent processes in those. It can also be fault tolerant: if one worker fails, the learning process can keep running.

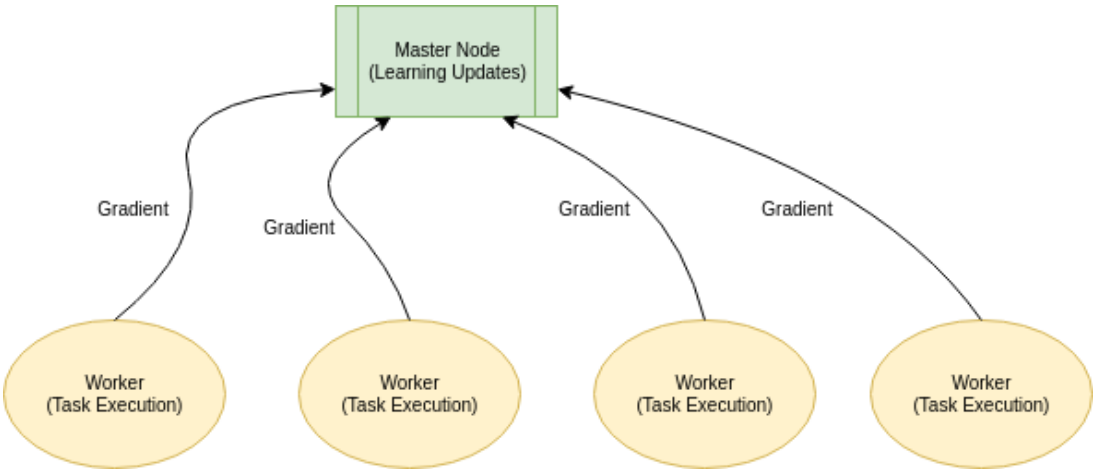


FIGURE 7.12 – Master-workers architecture for data parallelism.

7.5.3.2 Synchronous vs. Asynchronous Distributed Training

There are two ways to apply updates from a master-workers architecture in the learning algorithm. Both ways are illustrated in Figure 7.13.

In asynchronous training, each worker executes the task, gathers data, calculate the gradient, and apply it in the neural network. There's no interaction between workers so

the learning updates can occur freely and, therefore, faster. However, the downside is that each worker has its own version of the network which can cause a discrepancy between them and slows the convergence.

On the other side, in the synchronous training, each worker executes the same job as the asynchronous one. However, the master node collects the gradients and applies an average from them. In this way, the updates occurs in the same frequency, but they are much more precise (and therefore we can use large learning rates).

Heess *et al.* (2017a) implemented a variation of PPO called Distributed PPO (DPPO) and their experiments show that synchronous updates worked better in practice. Hence, in this experimentation we also used this type of distributed training.

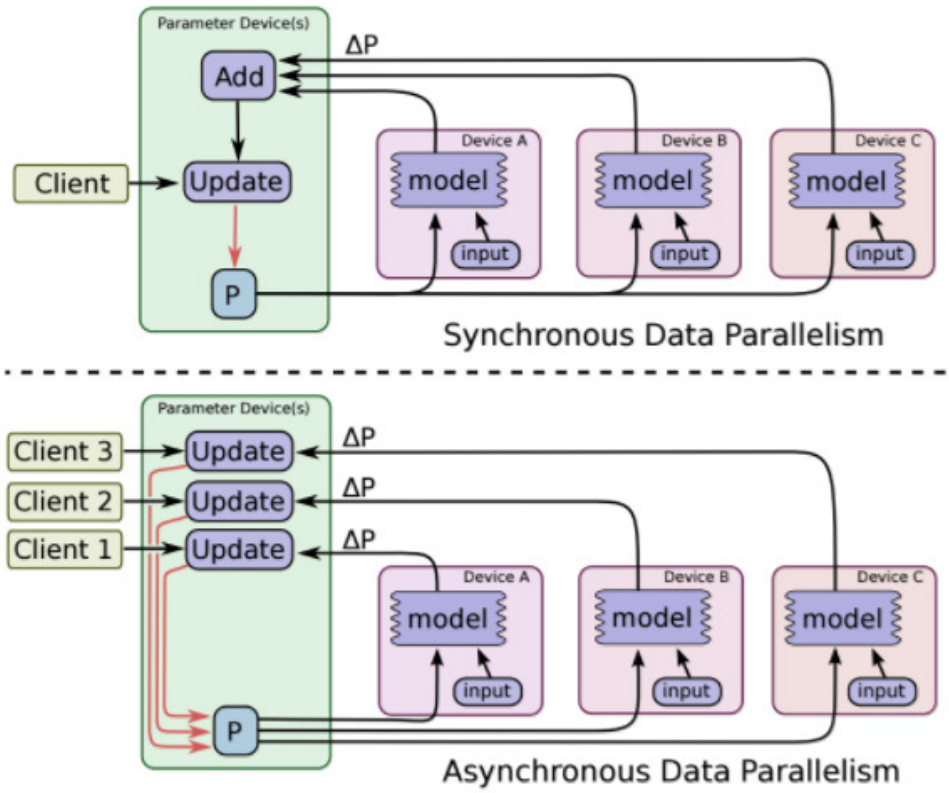


FIGURE 7.13 – Synchronous and Asynchornous Distributed Training (ABADI *et al.*, 2015).

7.5.4 Metrics

We need evaluation metrics to compare the techniques in Section 7.2. Several of them were collected and described below:

- **Episode Reward Mean** – this metric is the mean of cumulate reward during the whole episode. It is directly related to the motion performance. Then, it is the main metric in the comparison;

- **Value Function Loss** – says how good is the prediction of value function by the network;
- **Timesteps So Far** – it means how many cycles (observation-action-reward) have been executed. This metric was used to compare the speedup from distributed training; and
- **Episode Length Mean** – in the case of models with no fixed length, this metric will allow us to understand how the learning worked.

7.5.5 Monitoring via Tensorboard

Lastly, we need a tool whose purpose is to monitor the metrics collected during the training. We used Tensorboard (WONGSUPHASAWAT *et al.*, 2017), a graphic visualizer tool from Tensorflow. Figure 7.14 shows the GUI with some metrics. Using that, we can compare several models and their metrics, visualize computational graphs, and debug the learning process.

FIGURE 7.14 – Monitoring training metrics with Tensorboard.

7.6 Experimentations - Imitation Learning Hypothesis

Table 7.6 synthesizes all the experiments presented in this Chapter.

Experiment	**Section**	**Description**
Distributed Training	Section 7.7	Evaluation of distributed RL training
Pure RL Training – RNR	Subsection 7.8.1	Evaluation of pure RL training using just the ball's velocity in reward.
Pure RL Training – RRR	Subsection 7.8.2	Evaluation of pure RL training using just the reference motion in reward.
Pure RL Training – RNR+RRR	Subsection 7.8.3	Evaluation of pure RL training using ball's velocity and reference motion in reward.
Pure RL Training – RNR+RRR+RISD	Subsection 7.8.4	Evaluation of pure RL training using ball's velocity, reference motion and the initial state distribution technique
"Supervised" Reinforcement	Subsection 7.8.5	Evaluation of a RL training using reference motion as reward but without interaction with environment.
Supervised Imitation Learning	Subsection 7.9.1	Results from the Supervised Imitation Learning training.
HLM Training – HLM+RNR	Subsection 7.9.2	Evaluation of hybrid model, with supervised prior and subsequent RL training, using ball's velocity reward.
HLM Training – HLM+RNR+RET	Subsection 7.9.3	Evaluation of hybrid model, with supervised prior and subsequent RL training, using ball's velocity reward and Early Termination techinque.
Kick Behavior: Numerical Results	Subsection 7.9.4	Evaluation of best final model in a kick test scenario.

7.7 Distributed Training Results

The first results we show here are regarding data parallelism by the adoption of distributed training. In RL setup, we used a master-worker model (as described in Subsection 7.5.3) with **108 agents** collecting data in parallel.

We used Intel AI DevCloud (INTEL, 2018). It is a cluster of Intel Xeon Scalable Processors that assists with machine learning and deep learning research, with several optimizations for this domain. We had available 10 computation nodes, each of them with 24 cores for processing. Therefore, we trained using 9 nodes for workers (each of them with 12 agents) and one node for master computation. All the scripts used for training configuration are available in the github repository[1] of this work.

In Figure 7.15, we see how the data collection had improved by using distributed training in AI DevCloud. In this graph, we have the number of samples collected by learning update in a training using the distributed configuration and other in a local training with a single agent.

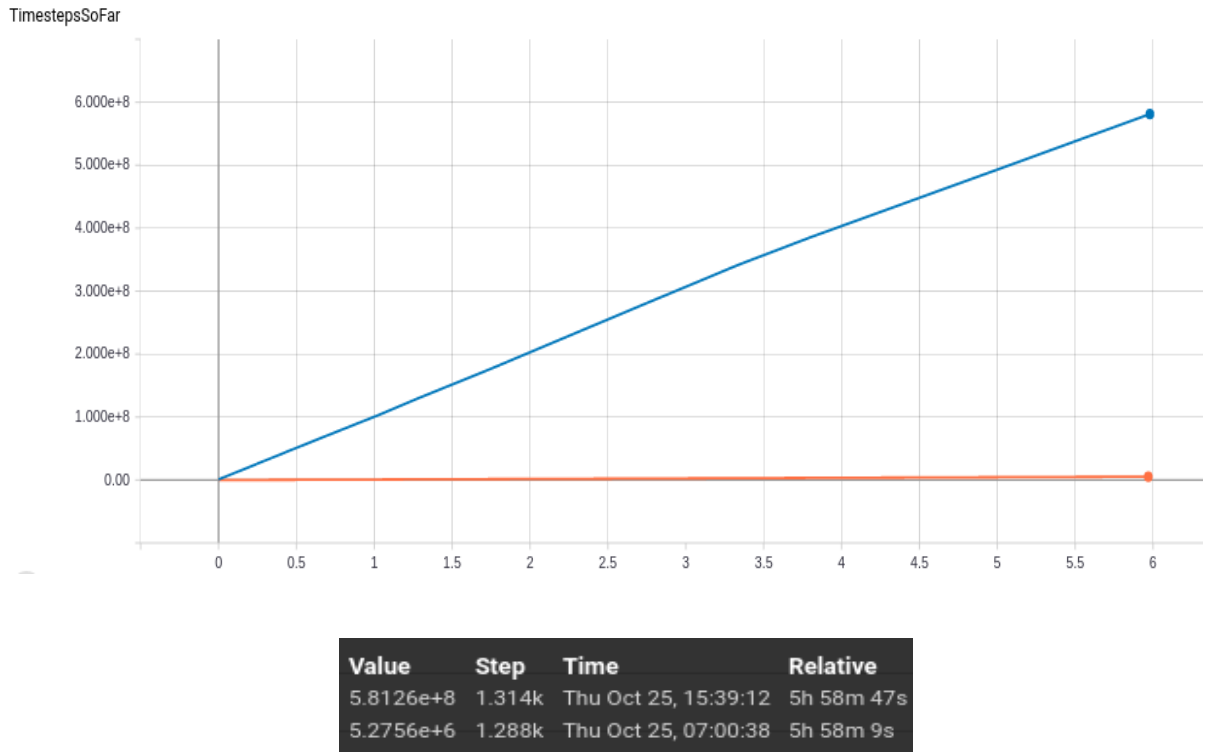


FIGURE 7.15 – Speedup Test

If we consider the same instant at each experimentation (i.e, after the same period of

[1] `https://github.com/luckeciano/deep-rl-humanoid-kick`

time), we can quantify the speedup:

$$SpeedUp \approx \frac{5.81 * 10^8}{5.27 * 10^6} \approx \mathbf{110}. \tag{7.4}$$

In this comparison, we have a speedup of 110. It is interesting the fact that the speedup is greater than the number of agents. It is worth mentioning that this number can vary depending on the training itself, because the computation will change over time. For example, if during training the episode length tends to grow, the time of simulating agents have the same behavior. Therefore, the reason of this speedup is that the training may have been conducted for a computation that collects more data rather than computing more network updates.

In Figure 7.16, we consider the best scenario for data collection from those experiments. In the graph, there is the number of samples collect by the time. In this example, we have collected approximately **740 millions of samples** in eight hours. It means **92.5 millions of samples by hour**. Each of them means a cycle of 20 milliseconds in the simulation server. Therefore, we collected approximately **21.4 days of real time training in each hour of simulation**. The training from Figure 7.16 collected approximately **171 days of uninterrupted training** – almost half year.

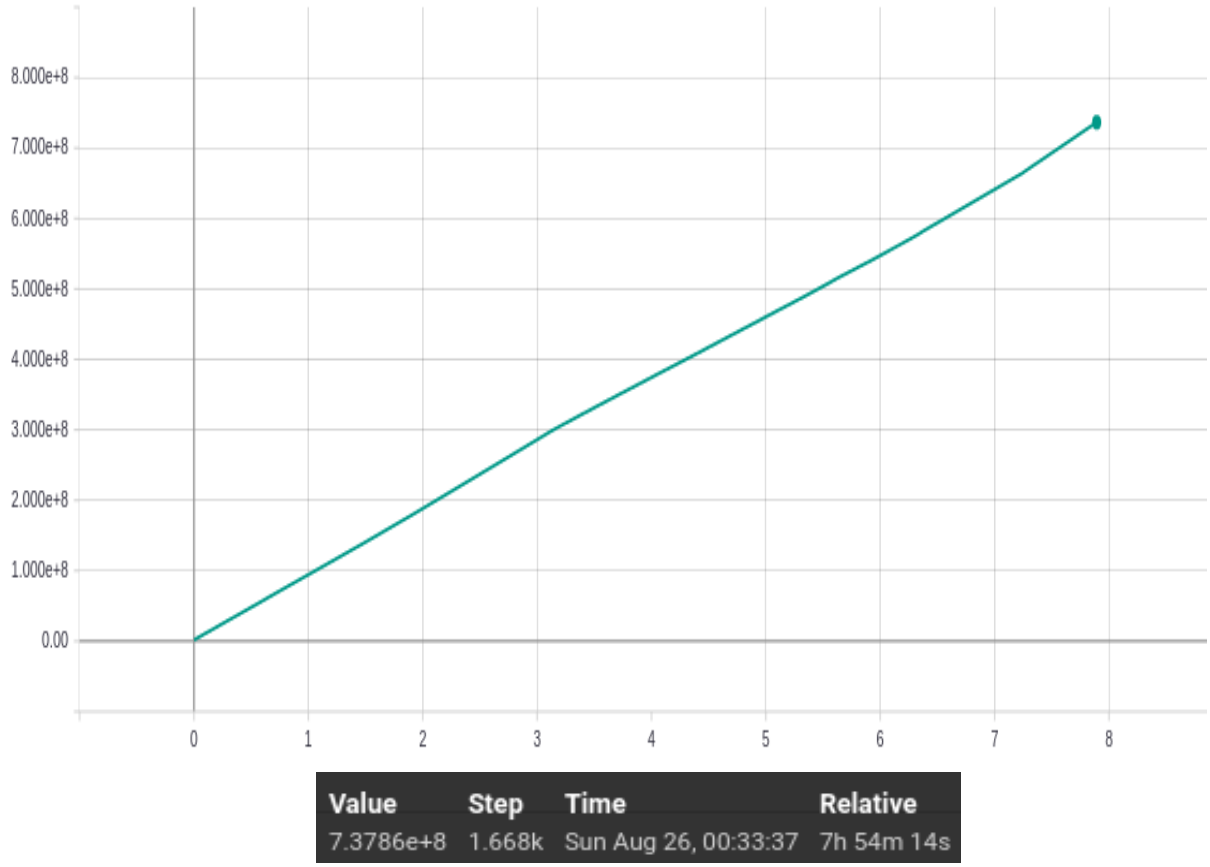


FIGURE 7.16 – Data collection in best scenario

7.8 Pure Reinforcement Learning Methods Results

In this section, we show the results regarding RL experiments. We tested several scenarios using the experimentation setup described in Section 7.2. All the hyperparameters from each experiment can be found in the Appendix A.

7.8.1 RNR

The first model we experimented was the RNR. In this, we started from randomly distributed weights in a neural network and, by using RL, we achieved a motion that not follows a well-defined human kick behavior, but is able to inject some velocity to the ball. Figure 7.17 shows how the reward improves over the learning process.

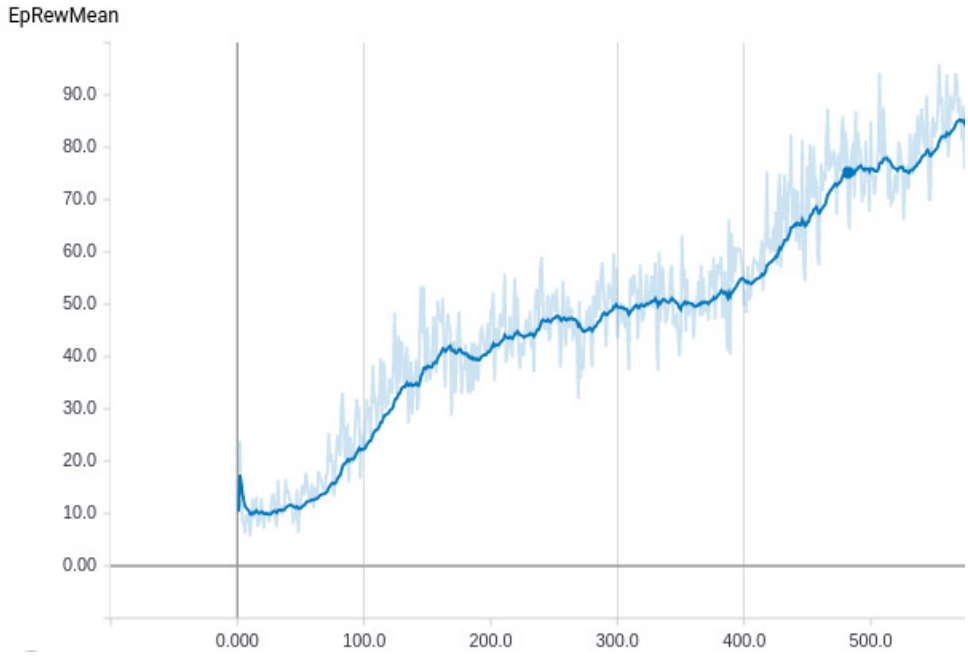


FIGURE 7.17 – RNR Reward Curve by learning update

Other experimentation sessions shown us that RNR tends to converge to the maximum value presented in Figure 7.17.

We also encountered a problem during the optimization process: Simspark simulation tends to drastically drop the reward to zero after approximately 4 or 5 hours of training in certain experiments. We did not find a reason but when we restart training from the previous session, the problem just disappear. We opted to not show this issue in results to not compromise the focus of this research.

Figure 7.18 shows the RNR resultant motion. It is possible to see that the behavior is different from the human kick pattern. Basically, the robot jumps and push its right leg

in ball direction. This kick runs between two to four meters straight. A video showing the animation may be seen in the repository.

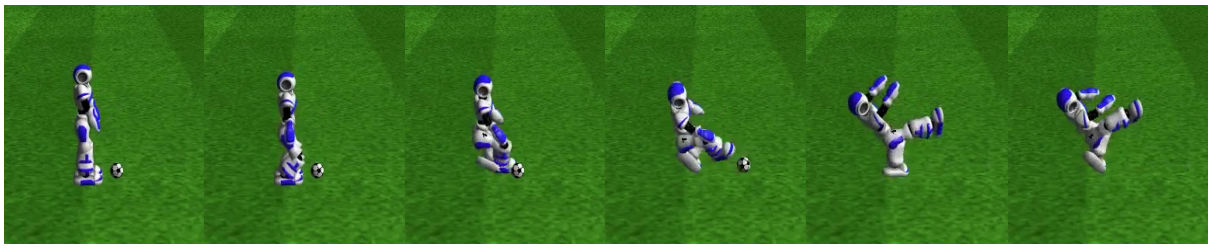


FIGURE 7.18 – RNR kick motion sequence.

7.8.2 RRR

In the context of optimization, due to the large search space, it is difficult to mimic a human behavior of kicking learning from scratch. To address this challenge, we used the reference motion also used in supervised learning section.

When we train just using this reward – without any reward to kick the ball itself – the robot achieves an interesting behavior: it remains stopped, even growing the reward, as in Figure 7.19.

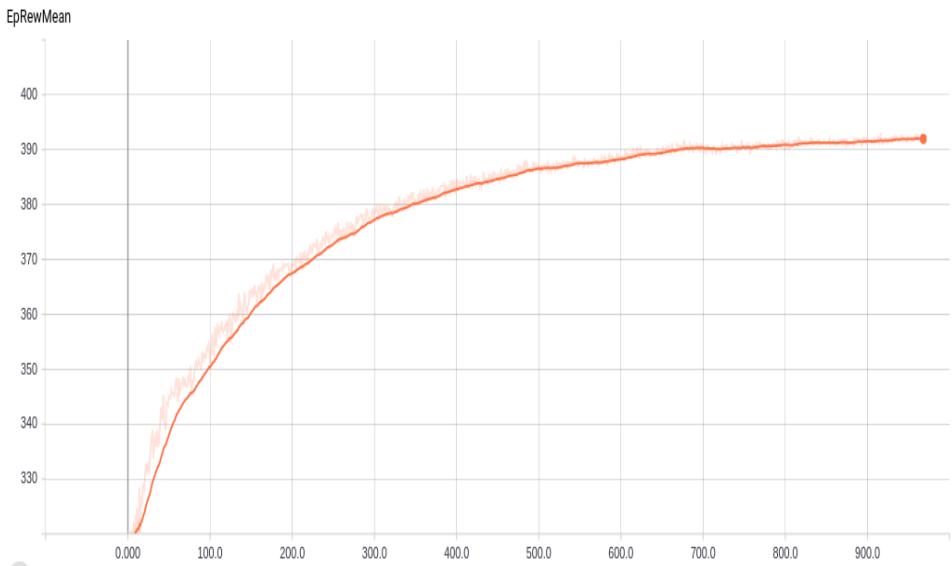


FIGURE 7.19 – RRR Reward Curve by learning update

Actually, we hypothesize that this behavior happens because it is better for the agent remains in the same position with a stabilized error than tries to reduce this error and possibly fall – which will result in a much worse reward if compared to just stand still.

7.8.3 RNR+RRR

A possible solution for RNR and RRR models is to join both in one single solution. It will try to avoid weird motion from RNR configuration by using a reference and will also reinforce positively the task of moving the ball.

Indeed, the resultant motion captures these two ideas and generates a new behavior, which looks more like a human pattern for kicking. We present the training curve for the best experiment in Figure 7.20 and the motion sequence in Figure 7.21.

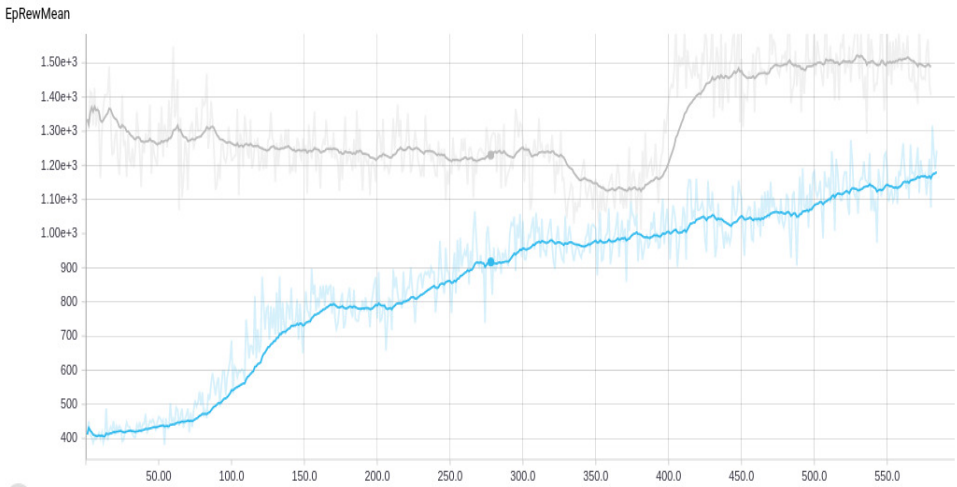


FIGURE 7.20 – RNR+RRR Reward Curves by learning update. We trained in two sessions. The blue curve shows the first session and the gray one the second session.

One fact that is interesting to mention: in RNR motion (without reference), the agent learns to kick with the right leg. In this model, on the other side, the agent learns to kick with the left leg, as in the reference motion.

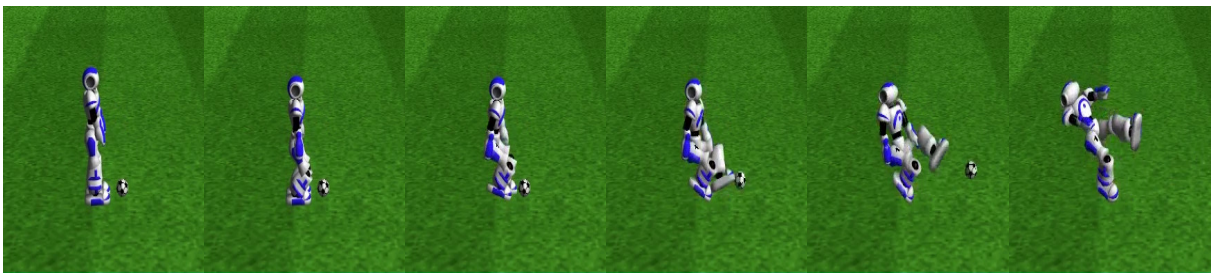


FIGURE 7.21 – RNR+RRR Motion Sequence. In this model, the agent learns to kick with the left leg, as in the reference.

The big challenge of this model is to tune the weight parameter for each reward. If we use much greater reward for moving the ball, the model collapses to the RNR configuration. On the other side, if the reference reward is much greater, it collapses to the RRR. It is a trade-off and is very difficult to tune this kind of parameter manually. Figure 7.22 shows a collapse case.

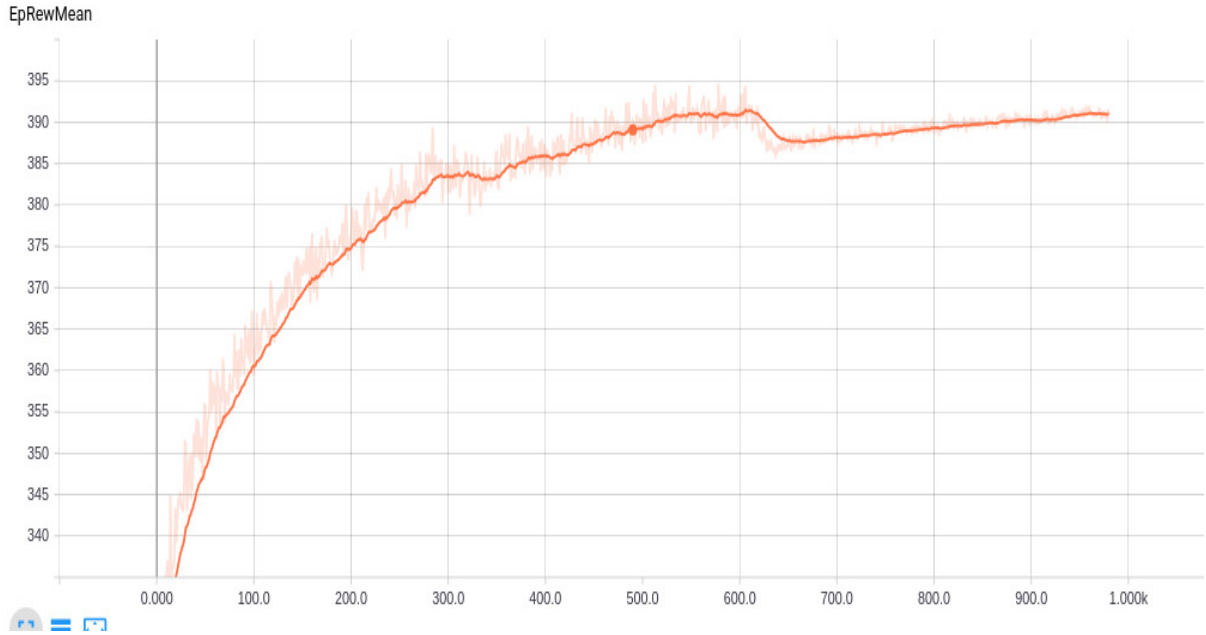


FIGURE 7.22 – Tradeoff between RNR and RRR models: when parameter weight for RRR is much greater, it collapses to RRR model.

7.8.4 RNR+RRR+RISD

In our experiments, even joining RNR and RRR, the agent was not able to learn the main patterns from the kick motion. It is difficult to learn, for example, that the leg should go back before making the move.

The main reason for this challenge – even when we pass the reference motion – is because the training is intrinsically sequential. It means that the later stages is achieved only if all previous states are satisfied. Otherwise, the agent will probably fall and not complete the episode.

Using the RISD model, we can train the whole motion "in parallel". We show the training curve from RNR+RRR+RISD model in Figure 7.23.

During the first session of training, the first thousand updates were related to reduce the error from the reference motion. It is similar to the curve from RRR model. After this, the model grows drastically until a new baseline. It corresponds to the moment where the agent, by exploration, achieves the situation of completing the whole motion.

The second session starts from the first one. However, the first part of the training is comparable to the result before the drastic grow from first session. Then, by exploration, the reward drastically grows again, but for a second baseline, higher.

Comparing the result from both training, the second baselines is able to kick the ball following the reference motion; the first is not.

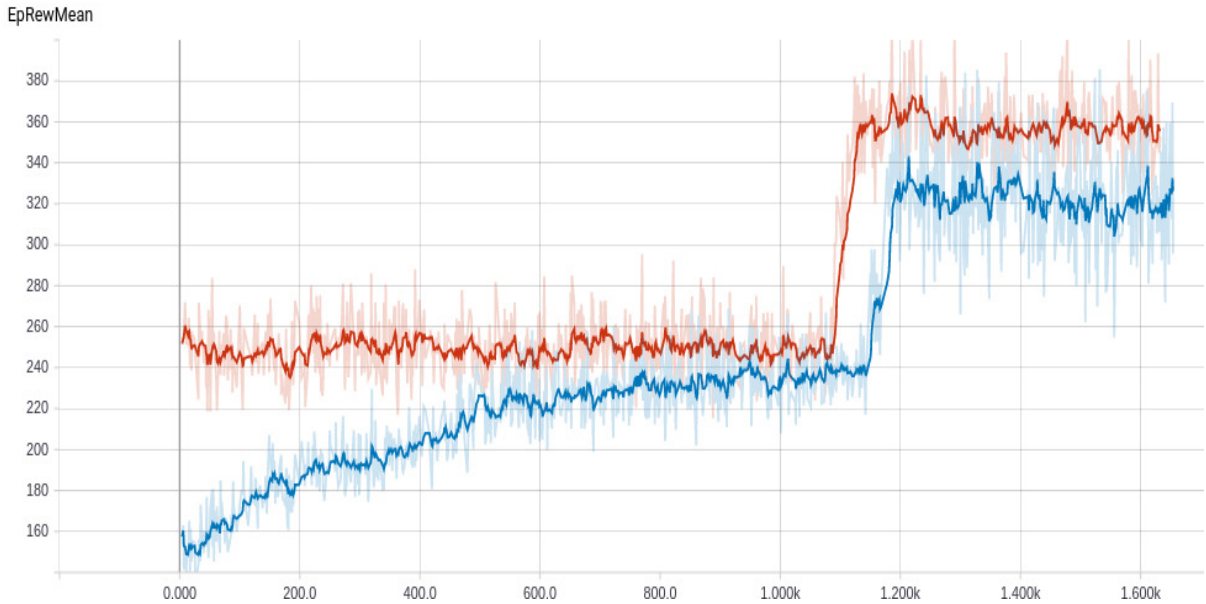


FIGURE 7.23 – RNR+RRR+RISD learning curve in two sessions of training. The blue curve was the first session and the red one was the second.

Figure 7.24 shows the motion sequence from RNR+RRR+RISD model. It is a complex and detailed motion. Although it is able to kick the ball, it is a rare situation and hardly ever achieves 3 meters. The big problem of this motion is the beginning: the agent fails in maintain a support foot. Nevertheless, the pure reinforcement method was able to mimic the human behavior of kicking.

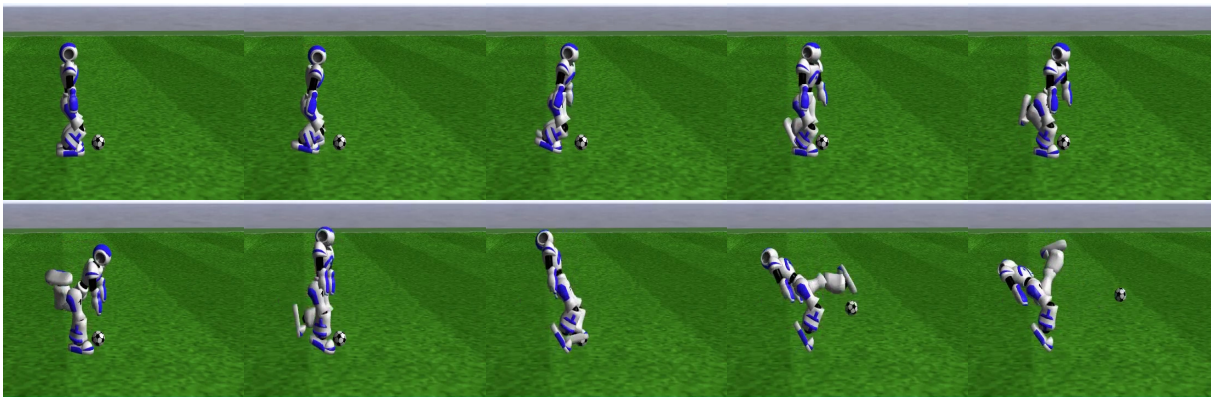


FIGURE 7.24 – RNR+RRR+RISD motion sequence.

7.8.5 Reinforcement "Supervised"

As a mid-term between pure RL techniques and HLM model, we propose a reinforcement "supervised" learning model, where we use only the PPO algorithm but as initial stage we use the reference motions as a dataset (x, y) where $x \in \mathcal{S}$ and $y \in \mathcal{A}$ and the

reward is the opposite of mean absolute error:

$$R(x) = -|\hat{y}(x) - y(x)|. \tag{7.5}$$

In this way, we reduce RRR model to a "supervised" problem without the simulation. This model breaks the sequentially of data, since the states are used in the training in isolation and not as a reflection of the horizon prior to this, satisfying Markov property.

Figure 7.25 shows the first of six training sessions for a reinforcement supervised configuration, using mean absolute error as penalization. It shows to be a bit difficult to converge – the optimization is very sensible to learning rate value. In each training session, we needed to tune this hyperparameter again.

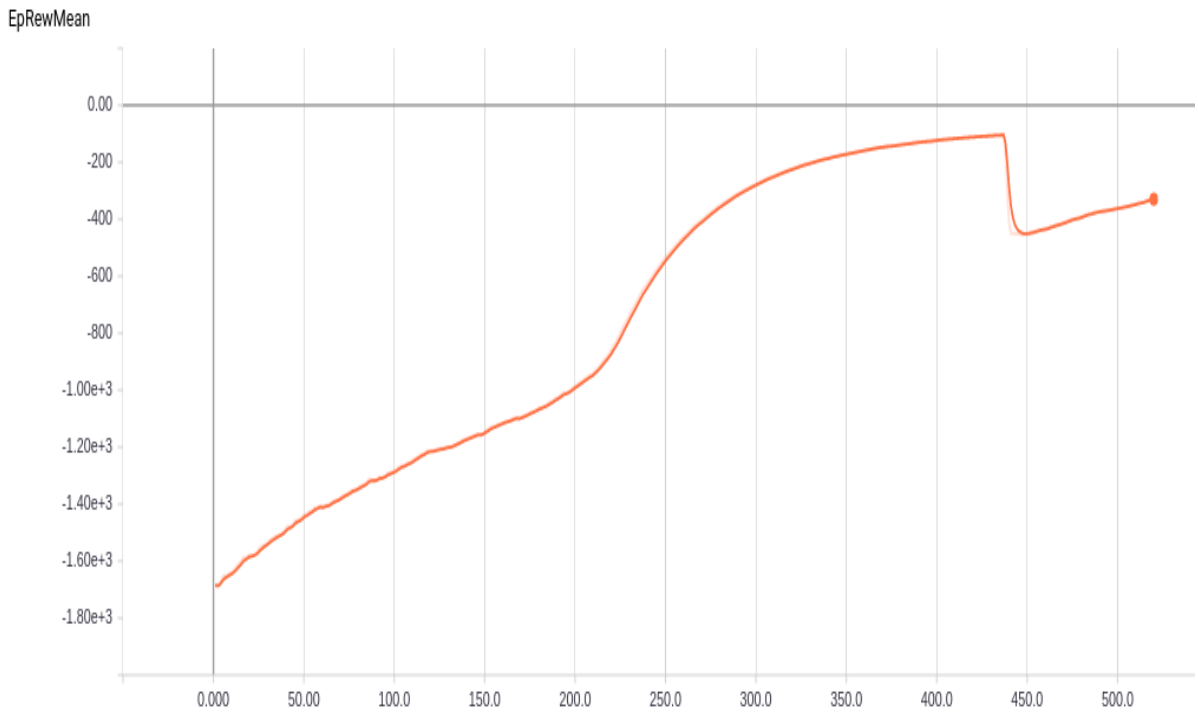


FIGURE 7.25 – First session of training for reinforcement supervised model.

Although the learned motion mimics the reference, it accumulates error at the end of the motion and ends up kicking the floor, as shown in Figure 7.26. For this reason, it could not be used as seed for new kick policies.

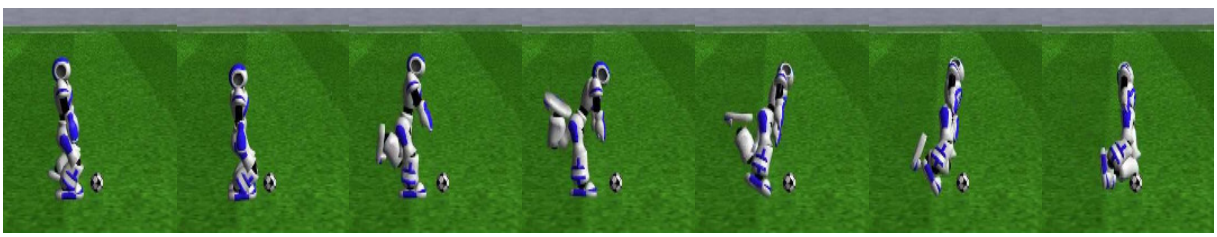


FIGURE 7.26 – Motion sequence from reinforcement supervised learning agent.

7.8.6 Other ideas for pure RL techniques

There are other possible ideas to explore, in order to obtain a reasonable copy from the reference motion to use as seed for optimization. Analyzing the experiments described in this section until now, we found out we have two main problems:

- The agent is not able to learn a stable support foot, compromising the later stages of kicking; and
- The kicking foot sometimes hits the ground and not the ball.

To address these problems, we need domain-specific rewards to ensure stability, such as:

- Consider the center of pressure that the support foot does on the ground; the more centralized is this pressure, the more stable the kick should become (MAXIMO, 2017);
- Consider the curve that kicking foot and leg do in relation to torso during reference motion; and
- Consider torso coordinates from reference motion during training (PENG *et al.*, 2018).

We preferred not to explore these ideas because as we will see in next section, hybrid learning models worked much better and easier. Furthermore, in this work, we aim to explore non domain-specific techniques that could be generalized for other kinds of motions.

It is worth mentioning that the techniques explored until now worked well,even without a seed motion for bootstrapping the learning. Fitting this specific reference motion is really difficult because it has high performance and, therefore, the robot operates close to the limit of stability. However, in other motions, they are good tooling for mimicking.

7.9 Hybrid Learning Models Results

In this section, we will explore an alternative for pure RL methods – what we called Hybrid Learning Models. In this configuration, there is a pre-training step where we used supervised learning algorithms to fit the reference motion before using the PPO algorithm.

7.9.1 Supervised Learning Results

The first results from Hybrid Learning Models are related to the supervised learning step where we copy the motion from keyframe representation to neural network one. We evaluate this model not just in kick motion, but also learning the walking engine and other keyframe motions, such as the get up motion.

7.9.1.1 Training Results

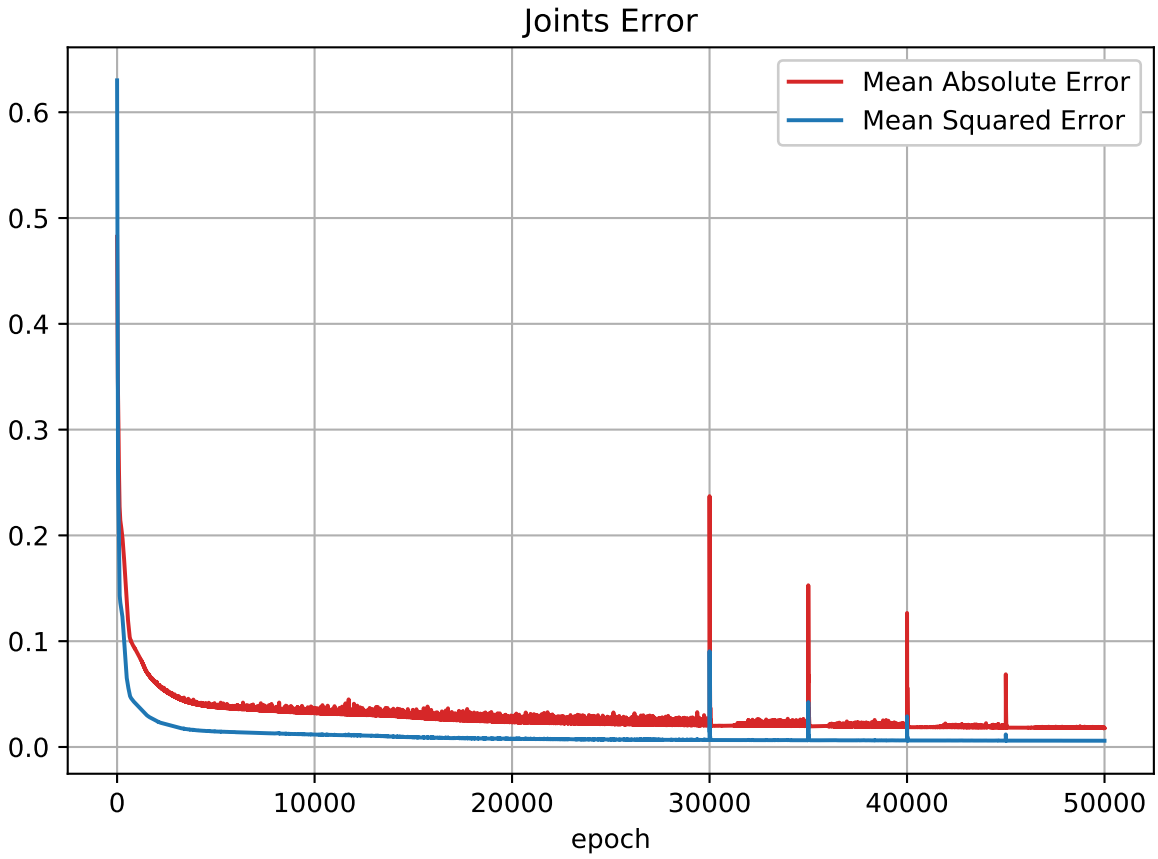


FIGURE 7.27 – Plots of mean squared error and mean absolute error, during training.

The initial results came from the training procedure, outside the simulation environment. Figure 7.27 presents training curves for the kicking keyframe dataset. In this case, plots show the mean squared error and the mean absolute error metrics. In both metrics, the value drastically decreases in the first epochs. This same behavior was present in other training procedures as well. However, only after thousands of epochs, the network has achieved a low error that successfully reproduced the motion, which demonstrates some sensibility of keyframe motions. The peaks, during the training has happened at the learning rate transition instants, but they did not hurt the training procedure itself.

7.9.1.2 The Learned Kick Motion

The final mean absolute error was **0.018** radians and the motion was visually indistinguishable from the original one, as can be seen in Figure 7.28. In this figure, snapshots from both motions were taken. Figure 7.29 shows several plots of joint angles, by comparing the original and learned kick motions. As we may see, the learned motion has fitted the movement with minor errors[2].

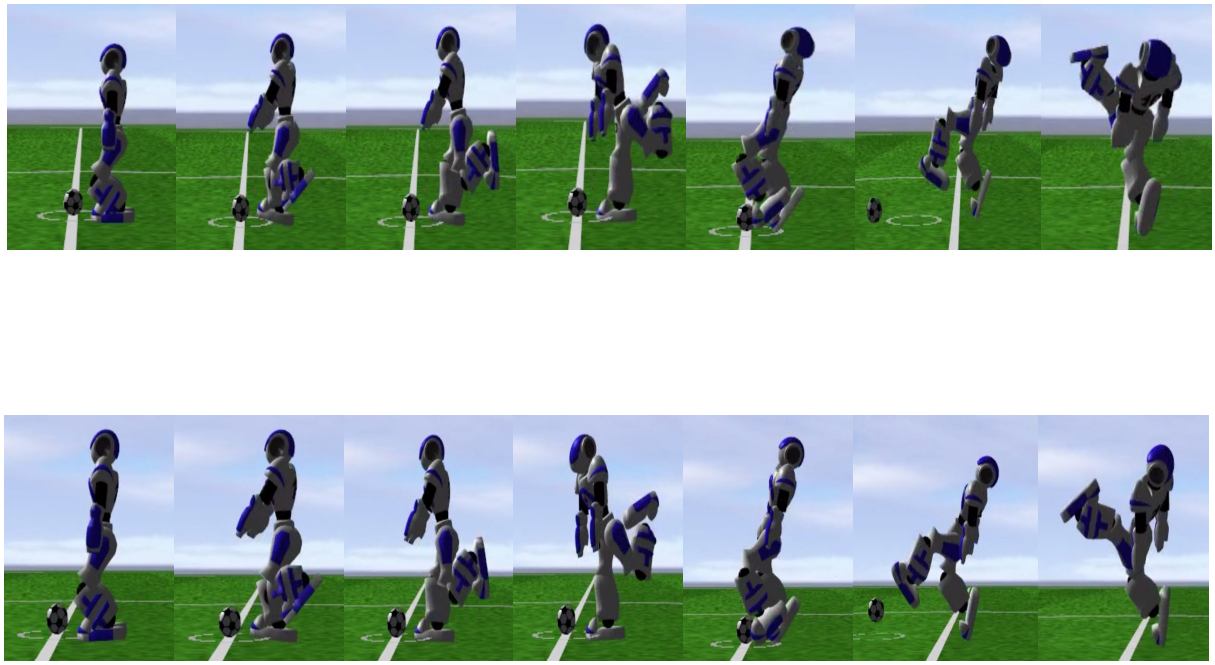


FIGURE 7.28 – The kick motion. The first row of figures shows the original kick motion. The second row shows the learned kick motion. Both motions are visually indistinguishable.

In order to evaluate the learned kick motion in the RoboCup 3D Soccer 3D Simulation domain, we created a statistical test. Inside the test scenario, the ball was initially placed in the center of the field with an agent near to it. The only action of the agent was to kick the ball in the goal direction. After the kick, the agent run until reaching the ball and kicked it again, repeating this process till scoring a goal. When the goal has occurred, this same scenario was repeated. The whole test was conducted, during thirty minutes in clock time and the following data was collected: total number of kicks, number of successful kicks, mean distance that the ball has traveled, and the standard deviation of this measure.

Table 7.3 shows the results from original and learned kicks. In this test, we define accuracy in terms of binary classification of kicks, considering a "right" kick when the ball's initial velocity after kick is greater than 0.5 meters per second.

[2] Kick results video: `https://youtu.be/UAbqQLUnvDo`

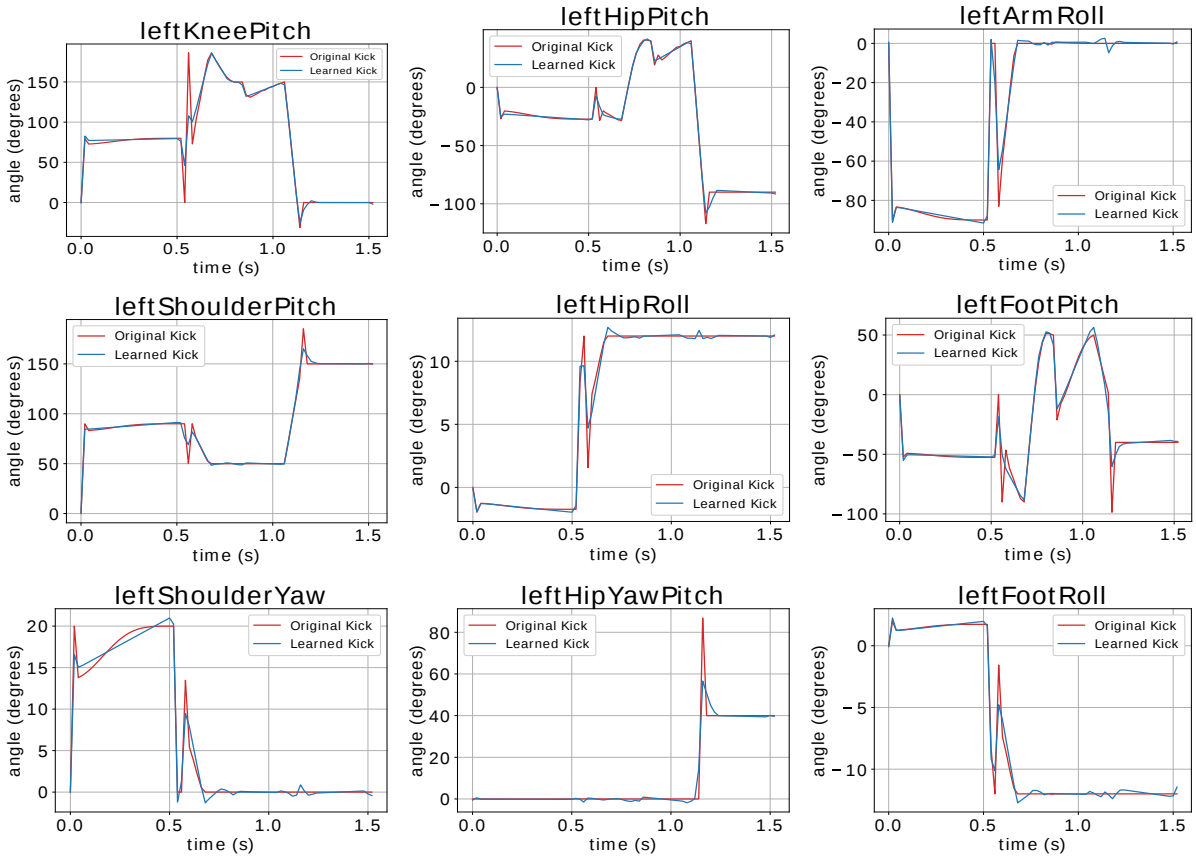


FIGURE 7.29 – Joint values for comparing original and learned kicks. The neural network was able to fit the joint trajectories with small errors.

TABLE 7.3 – The Kick Comparison

Kick Type	**Statistics**		
	Accuracy (%)	**Distance** (m)	
		Mean	***Std***
Original Kick	64.5	8.92	3.82
Neural Kick	52.6	7.16	4.06

Although both kicks had similar results, the original kick was slightly better in this scenario. By confronting Figure 7.29, we can conclude that even with an almost equal representation, the kick lost part of its efficiency due to this residual but cumulative error.

7.9.1.3 Learning Walk Motion as a Keyframe

By using the modified server described in Subsection. 7.3.1, a dataset with samples of the UT Austin Villa's walking motion (MACALPINE *et al.*, 2013) was acquired. This team is the current champion of the RoboCup Soccer 3D competition (MACALPINE; STONE, 2018).

The objective was to mimic the walk motion as a keyframe and use this motion in our agent. The previously described framework for learning our own kick motion was used in this training, by including the neural network architecture and its hyperparameters.

The results from this training are shown in Figure 7.30. Similarly to Figure 7.29, it shows the joint angles throughout the walking motion period for the original and learned walk. Additionally, it shows the real joints values from the movement in the server. These joints were chosen because they were the most dynamic in the walk motion and the hardest to learn.

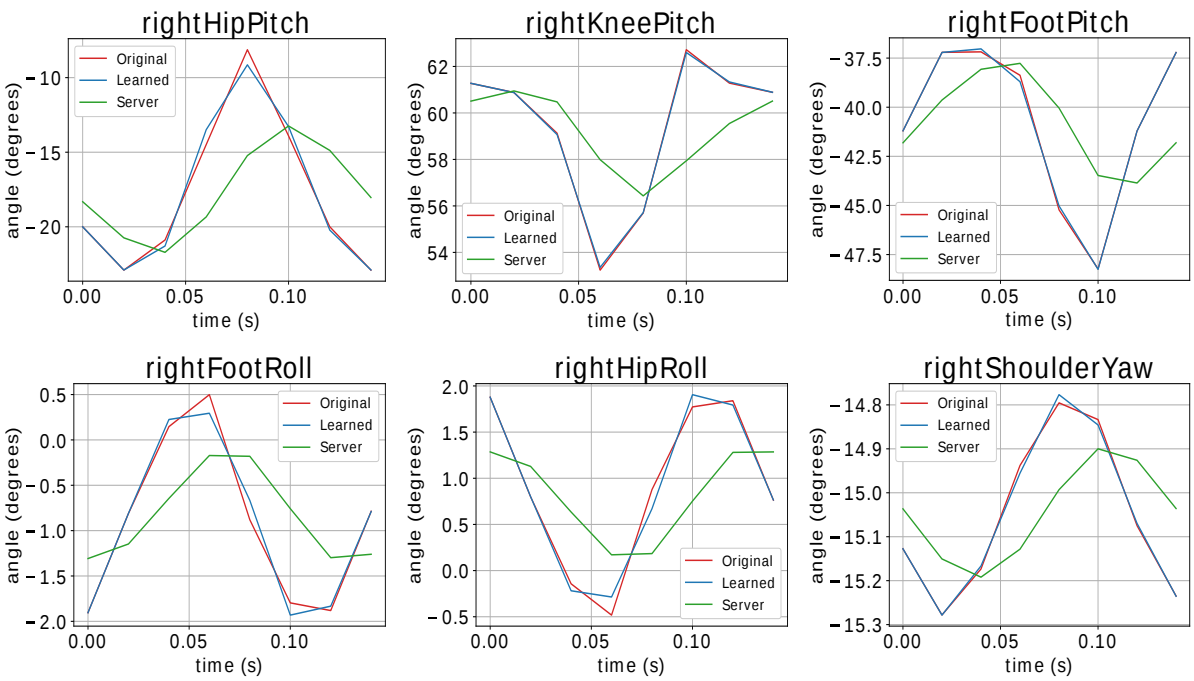


FIGURE 7.30 – Joints positions, during a period of the walking motion for the original walk, and the learned walk and the joints positions effectively attained, during the learned walking motion.

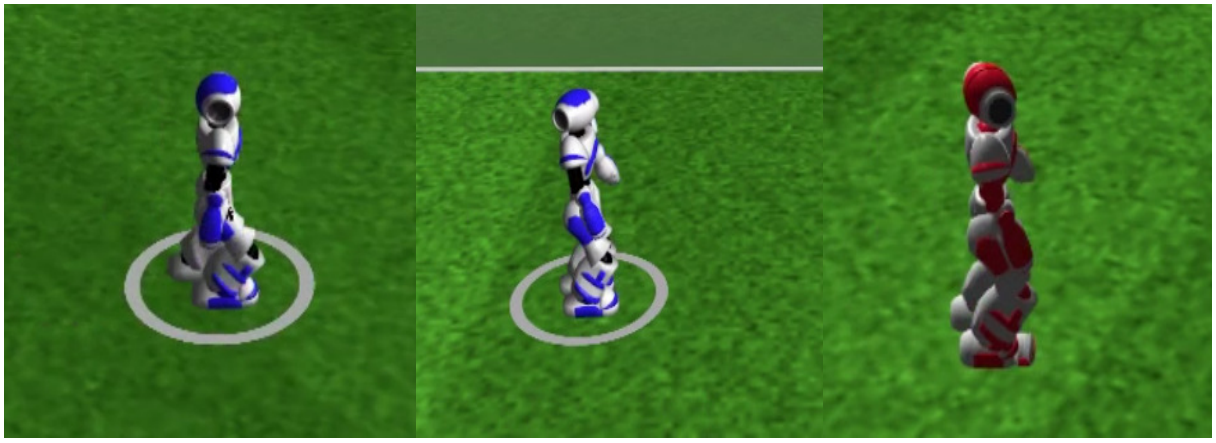
The learned motion has fitted the dataset very well. However, these values were just desired joints. In fact, these values were used as references to joint controllers and were

also attenuated due to joint dynamics. Furthermore, this motion was operated in an open-loop fashion, so the agent was not able to correct its own trajectory, and this walks got biased within the simple task of walking straight forward.

Despite the facts previously described, the motion has worked well in a non-competitive scenario[3], which was shown in the metrics collected from the Forward Walk test scenario – agent walking forward from the goal post until the center line of the field – in Table 7.4 and the visual representation in Figure 7.31.

TABLE 7.4 – Walk Comparison - Forward Walk

Walk Type	**Statistics**			
	Velocity (m/s)		**Y Error** (m)	
	Mean	***Std***	***Mean***	***Std***
Original Walk	0.87	0.01	-	-
Learned Walk	0.23	0.01	0.96	2.63



(a) (b) (c)

FIGURE 7.31 – The walking motions comparison. Figure (a) shows our agent in its regular walk, Figure (b) shows the same agent mimicking UT Austin Villa walk, and Figure (c) shows the UT Austin Villa agent itself performing his own walking motion.

7.9.1.4 Other motions

This same framework was used to learn other keyframe motions originated from our agent itself, such as the get up motion. As the cases previously described, the resultant neural network was capable of mimicking the keyframe, by including its interpolation.

[3] Walk results video: https://youtu.be/-pHxTrxllyY

Hence, all of our keyframe motions could be replaced by neural motions with similar performance.

However, the huge improvement of this method was about mimicking other teams motions. In the Soccer 3D environment, movements like kick and walking have giant impact in team's performance. With this learning framework, our agent was able to mimic multiples movements from several teams.

As an example, we have collected data from UT Austin Villa kick, which was originally optimized by using Deep Reinforcement Learning techniques (MACALPINE; STONE, 2017). Our agent has learned this kick without any additional optimization strategy: we just have used samples collected from the modified server.

7.9.2 HLM+RNR

In this model, we used the supervised learning model obtained and optimized it towards the task of kicking the ball using RL. The idea is to fine-tune the imitated motion to outperform the performance of original one.

Figure 7.32 shows the reward curve for HLM+RNR model. We can see that the reward double during the course of training.

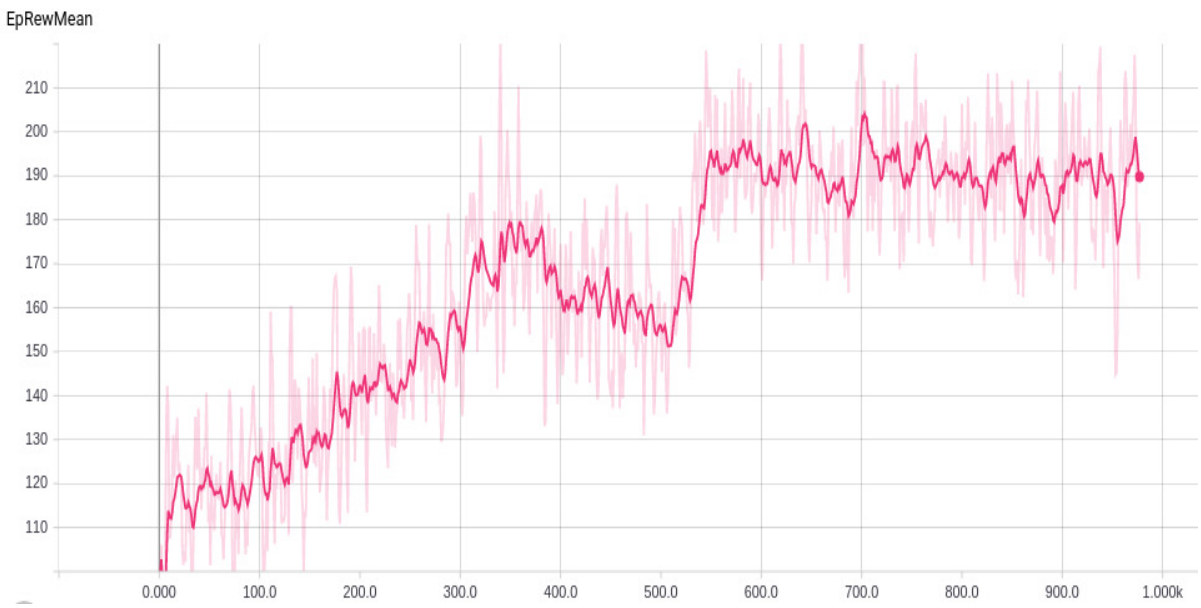


FIGURE 7.32 – HLM+RNR reward curve.

Figure 7.33 shows the motion sequence for HLM+RNR model. We see that this motion is very similar to those ones represented in Figure 7.28, even the performance being so different.

The only problem of this training is the integration with the kick behavior of the

agent. Here, we optimize just the kick motion. However, the behavior also includes walk to the ball and stops in the right place with the right initial joint values (what we call ball approaching). Then, the performance in the optimization environment is a bit different from actual game situation.

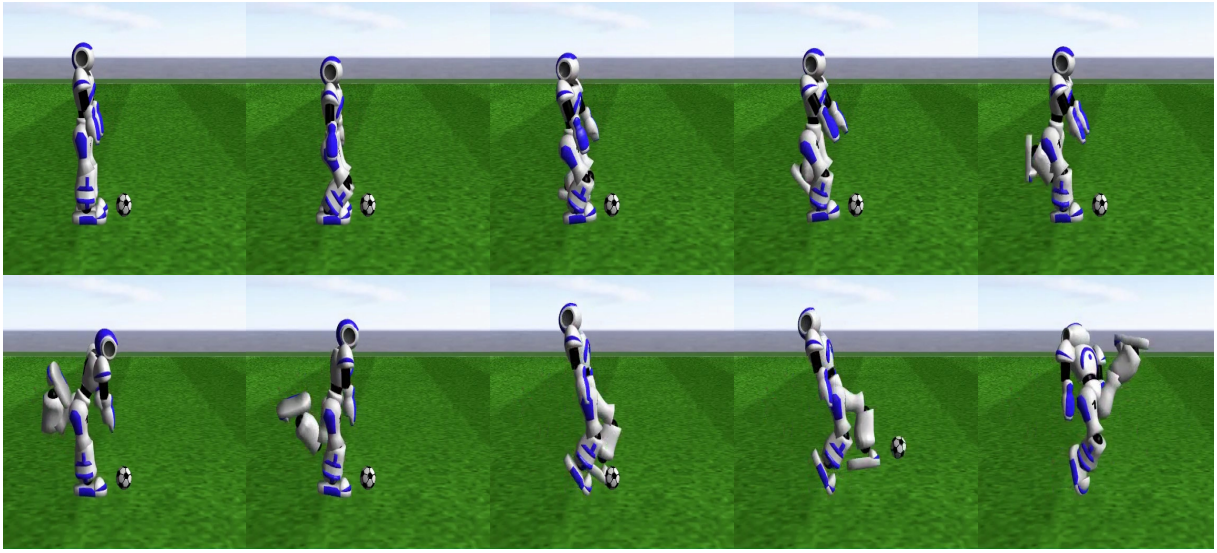


FIGURE 7.33 – Motion sequence of HLM+RNR model.

7.9.3 Final Model: HLM+RNR+RET

As final model to explore, we addressed HLM+RNR with Early Termination technique. As described earlier in this work, without it, we collect data that potentially harm training.

In this case, Early Termination consists in end up the episode in two scenarios:

- When the robot falls and, therefore, it cannot recover itself; and
- When robots kicks the ball, all actions after it will not affect the reward; it is a way to make this reward "less delayed".

In terms of RL training, we did two sessions. In the first one, we parameterized RNR in a way to mainly improve the velocity in frontal direction. The parameters can be found in Appendix A. Figure 7.34 shows the reward curve. It results in a kick that optimizes the main problem from the original kick: accuracy. As you can see in Table 7.3, the keyframe kick has low accuracy and the learned one is worse, due to the intrinsic error from the learning process. At the end of this first session of RL training, the agent is able to improve its accuracy, although the mean distance being low. Numeric results will be described in next subsection.

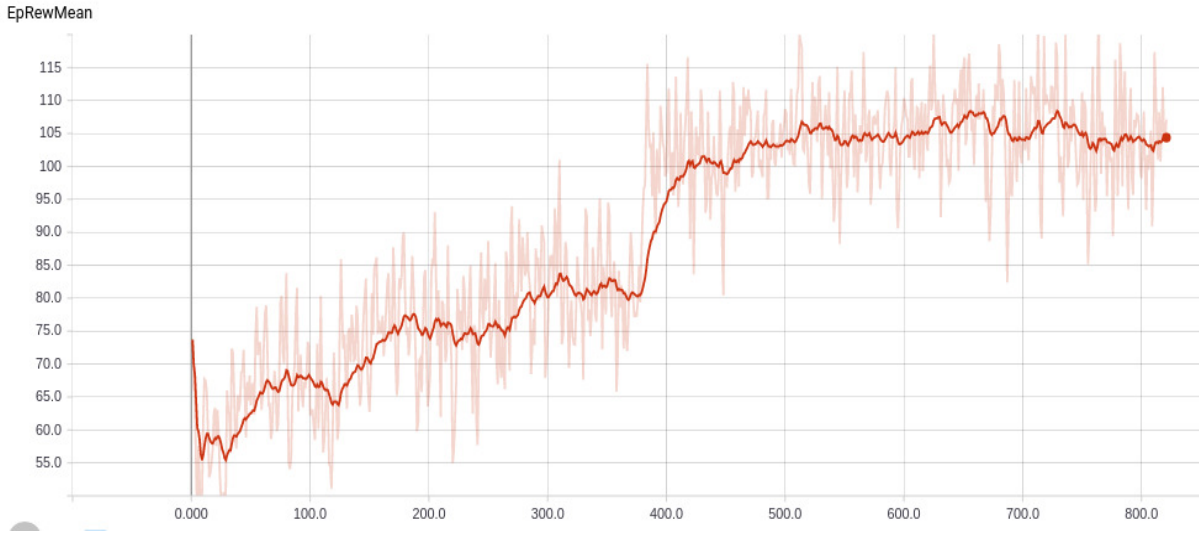


FIGURE 7.34 – The reward curve from the first session of HLM+RNR+RET model.

Figure 7.35 shows the motion sequence after this first session. Although the agent kicks the ball, there is no gain in height. This is a problem for two reasons: first, in a game situation, the ball probably will hits an opponent; second, there is friction between the ball and the ground, decreasing the final reach of it.

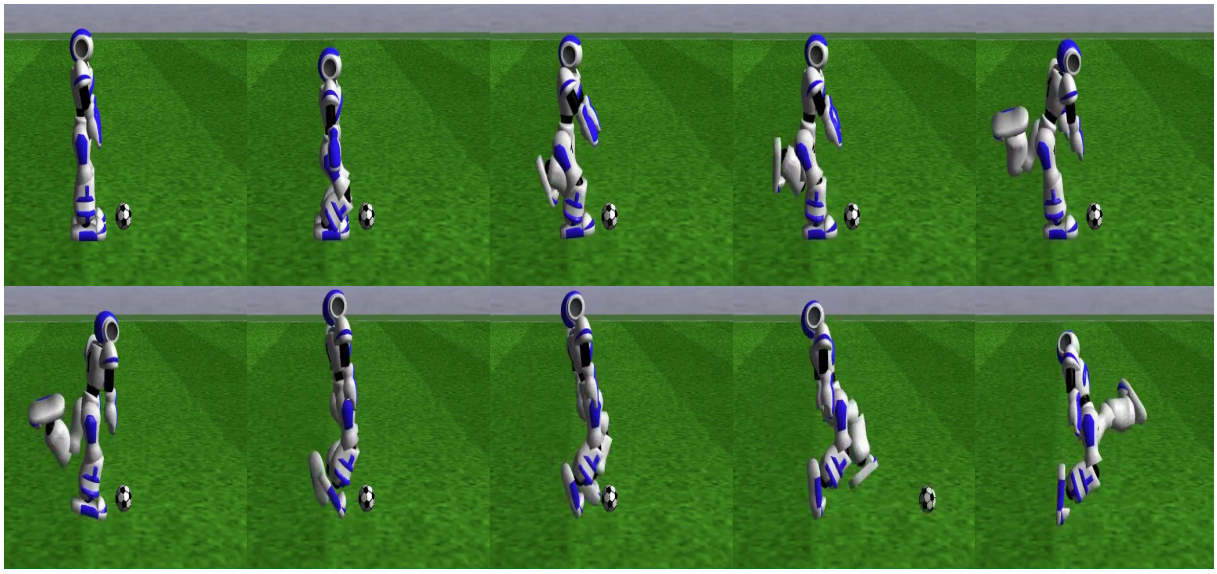


FIGURE 7.35 – HLM+RNR+RET motion sequence after first session of training.

In the second session, on the other side, we gave preference to kicks with height, positively reinforcing z-axis velocity. Figures 7.36 and 7.37 shows the reward curve and motion sequence, respectively. Even not being so different from motion the sequence of Figure 7.35, this training made the agent hits the ball in a way that it gains height, as shown if Figure 7.37, resulting in a better general performance than the original keyframe kick.

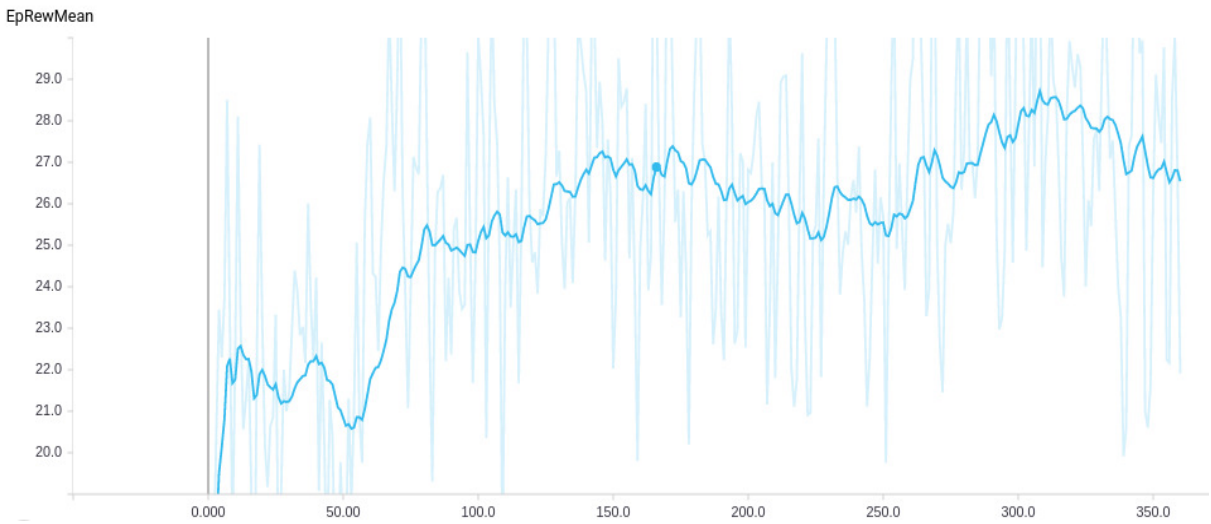


FIGURE 7.36 – The reward curve from the second session of HLM+RNR+RET model.

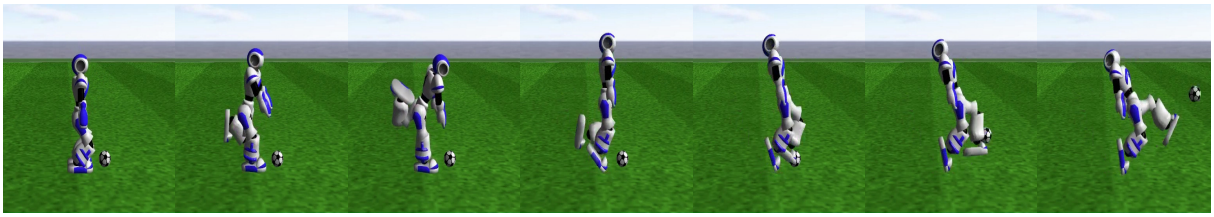


FIGURE 7.37 – HLM+RNR+RET motion sequence after second training session.

7.9.4 Kick Behavior: Numerical Results

We created a last test scenario, similar to that from Table 7.3, where the agent started 2 meters from the ball, approaches to it and finally kick the ball. This process is conducted by the Kick Behavior, a high-level agent behavior that integrates the walk and kick motions. The idea behind this is to try to simulate better in-game situation and check if the model obtained by these optimizations generalizes well to high-level behaviors, given that we did not optimize any parameter regarding to ball approaching or waking motion.

We collected the distance traveled by ball in axis x and y, and the maximum height achieved by the ball during its travel. The concept of accuracy is the same used in Subsubsection 7.9.1.2. We ran a hundred of samples and calculated accuracy and mean and standard deviation from other features.

In Table A.3, we compare four different kicks:

- **Original Kick** – the keyframe kick used as reference motion;
- **Learned Kick** – the neural network we learned by supervised learning and using as dataset the Original Kick;
- **HLM+RNR Kick** – the Hybrid Learning Model proposed in Subsection 7.9.2; and

- **HLM+RNR+RET Kick** – the Hybrid Learning Model proposed in Subsection 7.9.3.

TABLE 7.5 – Kick Comparison - General Evaluation

Kick Type	**Statistics**				
	Accuracy (%)	**Distance X(m)**		**Distance Z (m)**	
		Mean	***Std***	***Mean***	***Std***
Original Kick	69.0	6.27	5.03	0.16	0.41
Learned Kick	63.0	3.06	4.22	0.09	0.17
HLM+RNR Kick	92.0	6.52	3.89	0.33	0.55
HLM+RNR+RET Kick	**92.0**	**7.60**	**3.71**	**0.45**	**0.49**

First, we can see that the usage of Early Termination improved the performance results in this test scenario, even the motion being very similar.

As Table 7.5 shows, Hybrid Learning Models outperform both original and learned ones, in this test scenario. General Evaluation means considering the distance traveled by all kicks, even the ones considered "wrong".

However, it is worth mentioning the Effective Evaluation: consider the distance just from "successful" kicks. We show this evaluation in Table 7.6.

TABLE 7.6 – Kick Comparison - Effective Evaluation

Kick Type	**Statistics**			
	Distance X(m)		**Distance Z (m)**	
	Mean	***Std***	***Mean***	***Std***
Original Kick	**9.05**	**3.44**	0.21	0.49
Learned Kick	4.82	4.46	0.12	0.21
HLM+RNR Kick	7.07	3.55	0.36	0.57
HLM+RNR+RET Kick	**8.26**	**3.09**	**0.48**	**0.49**

We can see from Table 7.6 that when considering just successful kicks, the performance from Original Kick is slightly better. We hypothesize that it happened because there is a trade-off between performance and stability: a better stability sacrifices a portion of performance.

As final experiment, we applied the HLM+RNR+RET model to a different kind of robot, the Nao with Toe. This one differs from the standard Nao used in the rest of the experiments by having an additional joint in the foot, making it larger. When using the keyframe motion, the Nao with Toe was not able to even kick the ball - very often it kicks the ground because of its large foot.

TABLE 7.7 – Kick Evaluation - Nao with Toe

Kick Type	**Statistics**				
	Accuracy (%)	**Distance X(m)**		**Distance Z (m)**	
		Mean	***Std***	***Mean***	***Std***
Nao with Toe Kick	**95.0**	**9.47**	**3.43**	**0.66**	**0.63**

We then applied the framework proposed in this work, which resulted in the values presented in table 7.7. It effectively shows the strength of this framework. The RL algorithm, by exploration, was able to adjust the initial motion to this kind of robot, resulting in a even better motion than the original keyframe based on standard Nao.

8 The Meta-Learning Hypothesis

In previous chapter, we presented the methodology, experiments, and results regarding of Imitation Hypothesis, concluding that, in the context of our experimental setup, a hybrid model with Imitation Learning and posterior RL optimization presented better results. In this chapter, we firstly exploit this model, analyzing hyperparameter sensibility and developing an approach in order to find the best ones for each task. Posteriorly, we conduct a study regarding precise kicks, aiming to find policies that are able to kick the ball to a desired distance. Finally, based on these precise kicks results, we propose a Meta-Reinforcement Learning algorithm called Bottom-Up Meta-Policy Search, in order to solve the problem of learn how to learn to kick.

8.1 Kick Benchmark

In the previous Chapter, we presented a model that successfully learned to kick and optimized it towards the presented task. In this Section, we further analyze some of the main components, such as the neural network architecture, the reward scaling, the exploration rate, and the PPO optimization hyperparameters.

In this Kick Benchmark, we fixed the task of previous chapter's final model – which includes the same reward function, state and action spaces, and termination states. We did a minor modification in the agent by making it run agent's high level behavior for kick during 30 timesteps before starting to execute the policy. This modification avoided a huge change in joints from the initial state (where all joints has zero value) to the first action given by the policy. Finally, we conducted all experiments using single agent training, instead of the distribution settings exposed in last chapter.

Due to competition reasons, we are unable to release agent's code. However, we released the binary used to conduct the training procedure. Additionally, we released all the code used to the PPO server as our OpenAI Baselines fork[1], with necessary documentation to such experiments. We are also releasing all the logs of each experiment conducted

[1] `https://github.com/alexandremuzio/baselines/tree/kick_policy`

to ensure the reproducibility of all experiments performed in this section[2]. Finally, we are also releasing the final code and datasets used to the Imitation Learning step in the main repository of this work[3].

It is well known that policies based on Deep Reinforcement Learning suffers from high variance, which turns to be hard to compare and perform analysis about the training procedure and final performance (COLAS *et al.*, 2018; HENDERSON *et al.*, 2017). To overcome this, we ran several seeds for each experiment and used **95% bootstrap confidence interval** (EFRON; TIBSHIRANI, 1986). This approach provides us more consistent and comparable results. The number of random seeds used has varied depending on the experiment, but they can be found in logs released. We used the boostrapped package (BEECHER DON VAN DER DRIFT; LANGNER, 2018) and we also released the scripts used to generate reward curves based on it.

Finally, for each experiment, we downsampled reward curves to consider only one thousand points.

Table 8.1 shows all experiments presented regarding of kick benhmark.

[2] `https://bit.ly/2WhagKe`
[3] `https://github.com/luckeciano/deep-rl-humanoid-motions-masters`

Experiment	**Section**	**Description**
Baseline Training	Subsection 8.1.1	Establishing a baseline for kick benchmark evaluation.
Analysis #1: Neural Network Architecture	Subsection 8.1.2	Evaluation of neural network architecture in the kick benchmark.
Analysis #2: Imitation Learning Model Error	Subsection 8.1.3	Evaluation of imitation error in the RL training.
Analysis #3: Critic Network's Weights Initialization	Subsection 8.1.4	Evaluation of weights initialization in predictor of value function in kick benchmark.
Analysis #4: Reward Scale	Subsection 8.1.5	Evaluation of reward scale during RL training in kick benchmark.
Analysis #5: Entropy Term	Subsection 8.1.6	Evaluation of how entropy term affects the RL training in kick benchmark.
Hyperparameter Search	Subsection 8.1.7	Evaluation of how a hyperparameter search improves RL training in kick benchmark.
Exploration Restarter	Subsection 8.1.8	Evaluation of Exploration Restarter technique in kick benchmark.
LSTM Model	Subsection 8.1.9	Evaluation of a LSTM architecture in kick benchmark.
Final Model	Subsection 8.1.10	Evaluation of final model that consider all the improvement from last experiments in kick benchmark, with numerical evaluation

8.1.1 Baseline Training

Before conducting the baseline training, we did some improvements in the Behavioral Cloning training. Firstly, we merged the five training phases described in Section 7.3.3 into a single one, using learning rate step decay callback. This change improved our final

error and avoided peaks in learning curve. Additionally, we refactored the code, extracting all the hyperparameters for facilitating tuning.

In the baseline training, we have used the methodology as in Section 7.9, and the hyperparameters of Table A.3, resulting in the reward curve of Figure 8.1. We trained using the Standard Nao robot. In this curve, we do not use smoothing techniques, which shows how RL training can be noisy. Furthermore, we can observe a drastic decrease near to step 600. We are not able to explain such decrease, but as we will see throughout this Section, all experiments in this benchmark have the same behavior, not being influential to the analysis. Despite of this detail, we observe a common behavior to successful optimizations in RL context, better shaped than those from Figures 7.34 and 7.36, which can be justified by our modifications described previously.

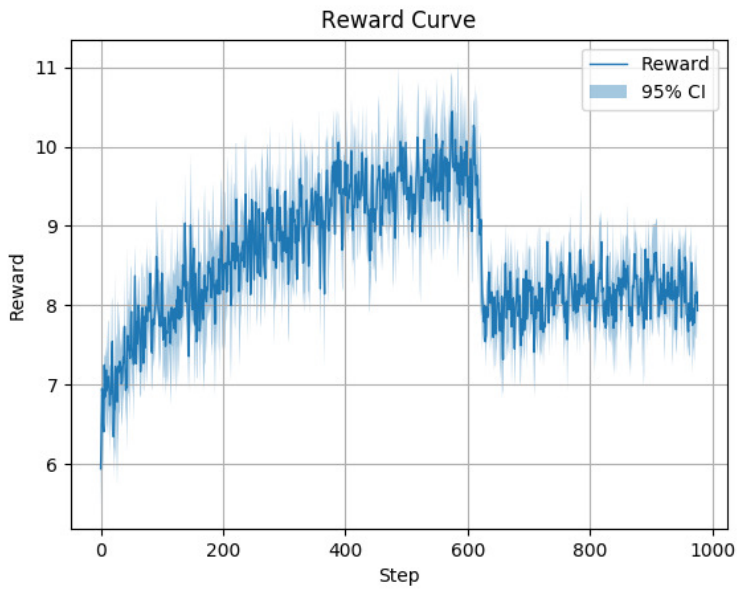


FIGURE 8.1 – Reward curve of baseline train.

8.1.2 Analysis #1: Neural Network Architecture

We conducted a first analysis in the neural network architecture. The idea was to make some changes about the layer size and mainly in activation function, maintaining the number os trainable parameters near from the baseline. We decided to use the standard network from OpenAI Baselines PPO implementation and used in the pure RL methods from Section 7.4. The main idea was to test how the hyperbolic tangent activation function works in the problem. The comparison between this model and the baseline is shown in reward curves of Figure 8.2.

We can observe that the model based on tanh activation performed slightly better than baseline during the first half of training, but practically converging to the same point. We

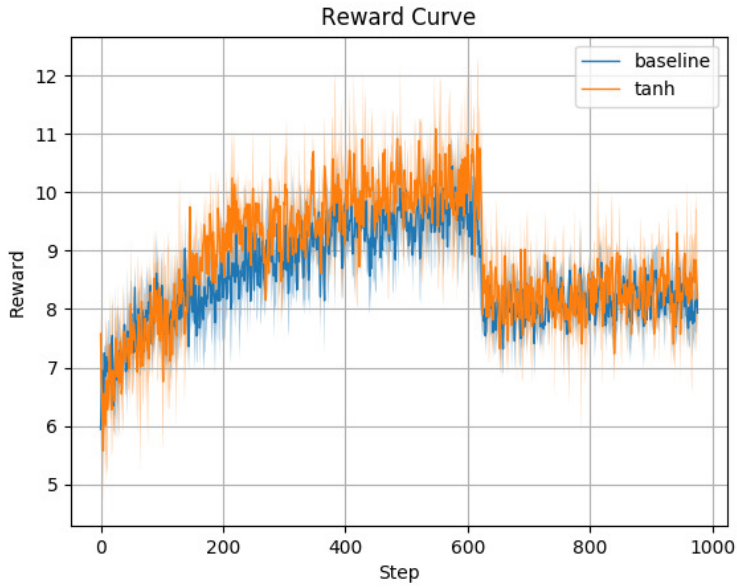


FIGURE 8.2 – Baseline versus tanh Architecture model.

hypothesized that this difference is caused mainly because tanh is more robust to "outliers" (data points that where very different from the mean) due to its saturation for very large absolute values. LeakyReLU activations, on the other hand, does not saturate, which can lead a poor initial representation, affecting initial stages of training. We decided to keep tanh activation, despite of the minimal difference at the end.

8.1.3 Analysis #2: Imitation Learning Model Error

Other question that arises in the methodology followed by this work is regarding overfitting. As we are conducting a training procedure that fits as best as possible the training data during Behavioral Cloning, our results could be possibly leading to an overfitting that could harm the posterior phase of optimization via RL due to a poor initial representation of the kick motion. We then consider two Imitation Learning models, where the only difference is in the final error. We picked the model with larger error in a way that the initial reward was approximately the same as the other model. The comparison is in Figure 8.3.

From Figure 8.3, we can conclude that a more specialized model improves substantially the RL training. Hence, there has no poor representation in such cases. In the context of this work, minimize training error in Behavioral Cloning training often leads to better motions.

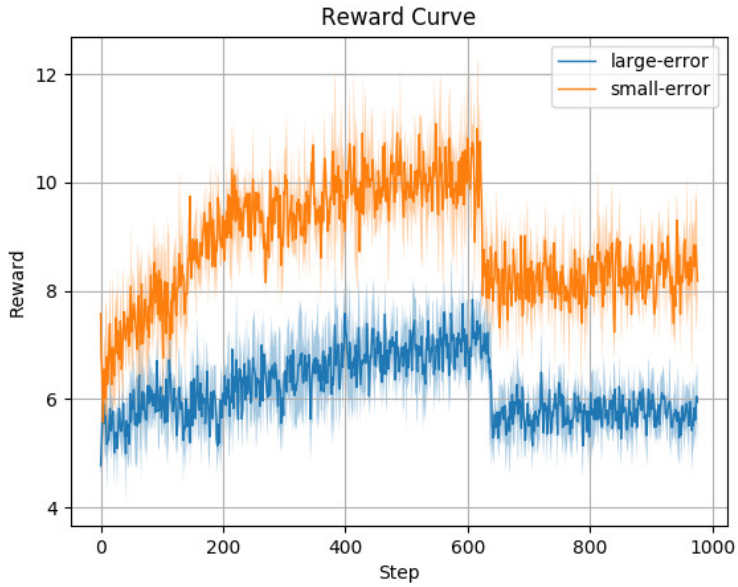


FIGURE 8.3 – Comparison between Imitation Learning models with different errors during RL training.

8.1.4 Analysis #3: Critic Network's Weights Initialization

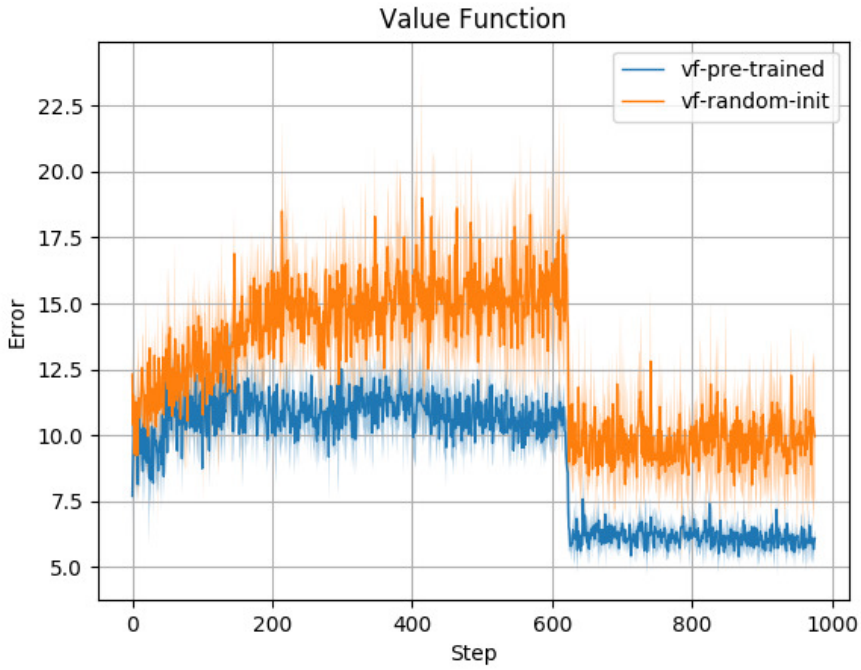
During the experiments of the previous Chapter, in case of models with initial learning by imitation, we loaded network weights in both policy and value function networks. We hypothesized that both networks share common low-level features given the same input, as exploited in other works (SILVER *et al.*, 2017c). We then conducted an ablation study to check the effectiveness of this idea.

Basically, we compare RL training with and without this initialization to the critic network, fixing all other properties. The result can be viewed in the Figures 8.4a and 8.4b.

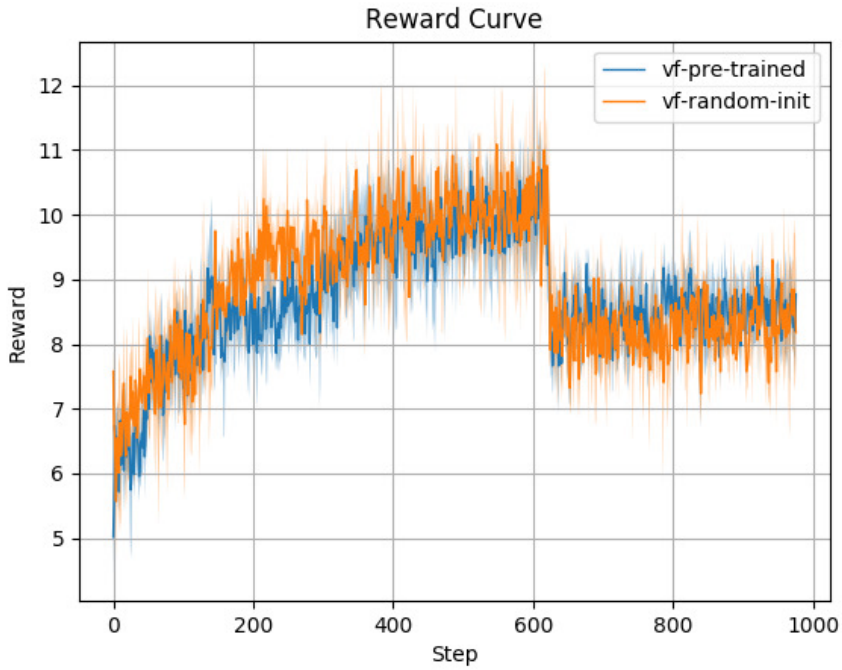
Figure 8.4a shows that, in fact, the loss has been always less in the case with initialization. It encourage us to maintain this initialization. However, in terms of reward, there is no valuable difference in this experiment. Actually, the curves of prediction error explicits a critic network unable to learn a better representation during training, which leads us to analyze the reward scale.

8.1.5 Analysis #4: Reward Scale

In the context of Deep RL algorithms, the reward scale plays a very important role. A large and sparse output scale can result in problems regarding saturation and inefficiency in learning. However, the scale varies across environments and scale values (ILYAS *et al.*, 2018). Therefore, we need to conduct another analysis to evaluate this factor in our work.



(a)



(b)

FIGURE 8.4 – Evaluation of Critic Network's weights initialization, in terms of Value Function prediction error (Figure 8.4a) and reward curve (Figure 8.4b).

In this experiment, we use two kinds of scaling. In the first one, we just multiplied the reward by a scaling factor, and experimented 0.1 and 0.01 – values that commonly appear in other works. In the second scaling approach, we normalize the reward, experimenting just standard deviation normalization and the full normalization using running statistics. The comparison of these models are in Figure 8.5a.

Figure 8.5a shows that a simple scaling factor is slightly better than normalization in terms of this benchmark. Additionally, Figure 8.5b shows that either 0.1 and 0.01 scaling factors result in better reward curves than the baseline.

To address this analysis, Figure 8.6 shows the critic network prediction error curve in case of 0.01 scaling factor. Comparing with errors from Figure 8.4a, we see a very different behavior, where the error decrease through optimization, which shows critic network effectively learning. This is a strong reason to reward increases when scaling reward.

8.1.6 Analysis #5: Entropy Term

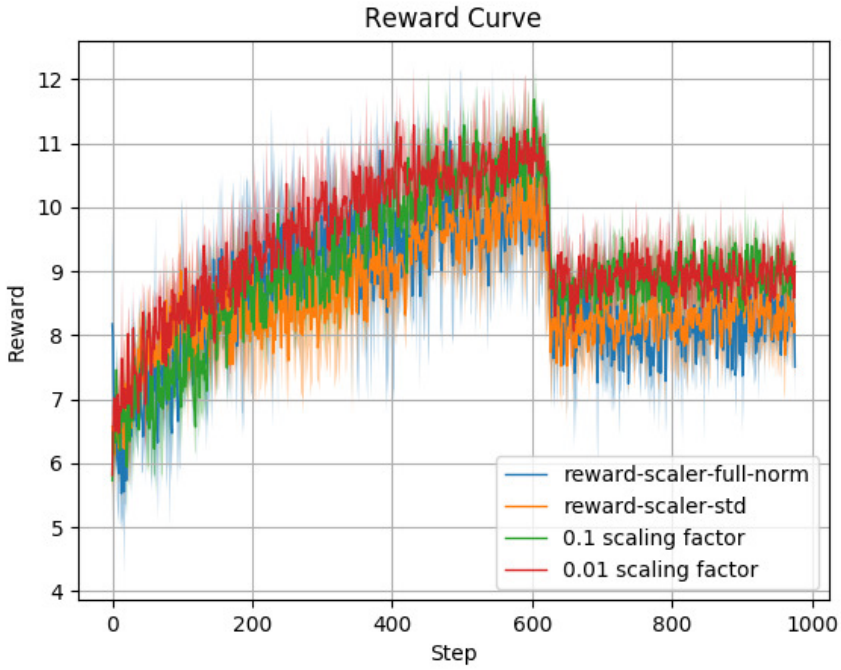
The last parameter of our analysis is the entropy term in the PPO surrogate loss. Accordingly to Mnih *et al.* (2016), this term should encourage exploration by adding more cost to actions that dominates, then making the agent choose more diverse actions during the learning process. In case of these experiments, we are scaling this entropy term by a multiplication factor. In the baseline model, there is no entropy (factor is zero). The values chosen here to conduct the experiments are based on previous works (SCHULMAN *et al.*, 2017).

Figures 8.7a and 8.7b show the reward curves. Comparing the multiplication factors, the values of 0.01 and 0.1 are approximately the same and slightly better than 1.0. Comparing the case of 0.01 and the baseline, we also see a minor increase in the reward curve.

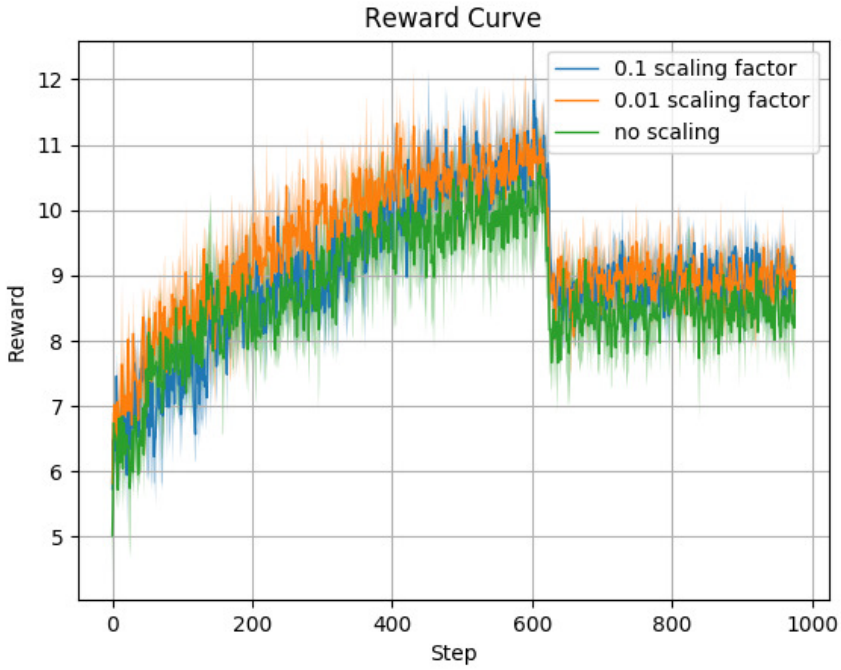
8.1.7 Hyperparameter Search

Those analyses conducted from previous sections are strong reasons to conclude that the design choices could change dramatically how much the agent can learn. However, there is no general rule about how to perform such decision, since some of these parameters have their sensitivity depending of environment and task. This is specially true when dealing with hyperparameters of RL algorithms. Therefore, it is often helpful to search the best values for specific task.

Generally, a manual tuning is performed. Nevertheless, the size of search space and the



(a)



(b)

FIGURE 8.5 – Evaluation of reward scaling.

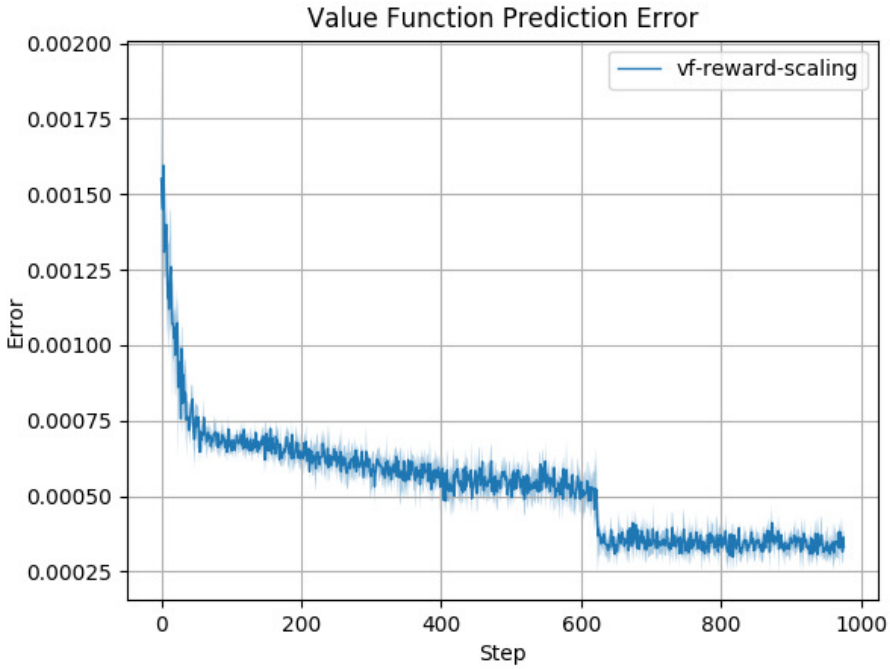


FIGURE 8.6 – Value Function prediction error curve when scaling reward using 0.01 factor.

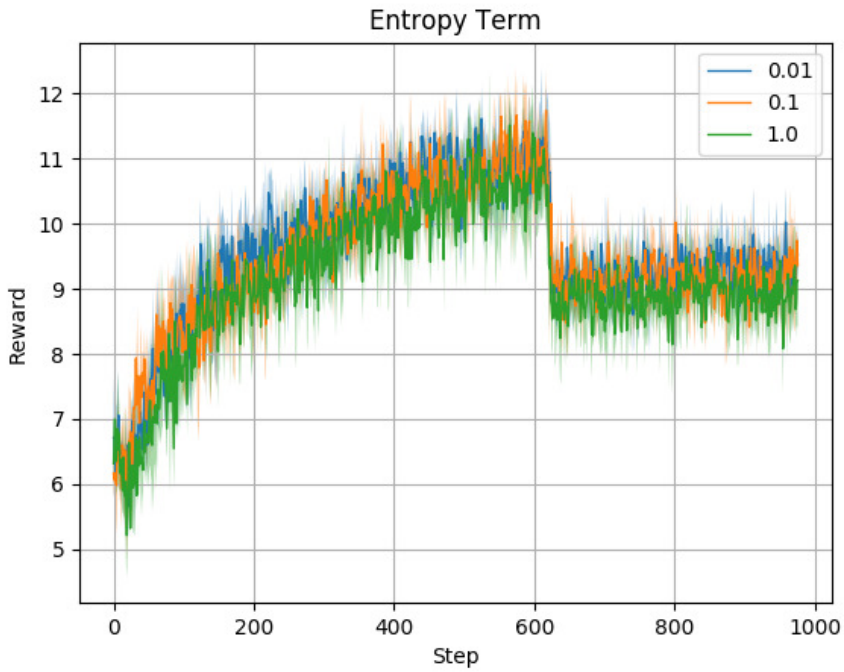
time required to run a single sample make this approach very costly and unlikely to result in a good hyperparameter set, encouraging the usage of automated search. The most simple approach to conduct it is random search, which is surprisingly effective (BERGSTRA; BENGIO, 2012). As an exercise, let us consider that the "close-to-optimal" region of hyperparameters occupies at least 5% of search space. We can calculate how many random seeds we need to ensure with 95% of probability that the best set lies in this region. In fact, each seed has 5% of chance to be in this region, so the probability to all seeds miss this region is $(1 - 0.05)^n$. Thus, to ensure 95% of probability, we need to compute n such as:

$$1 - (1 - 0.05)^n > 0.95, \tag{8.1}$$

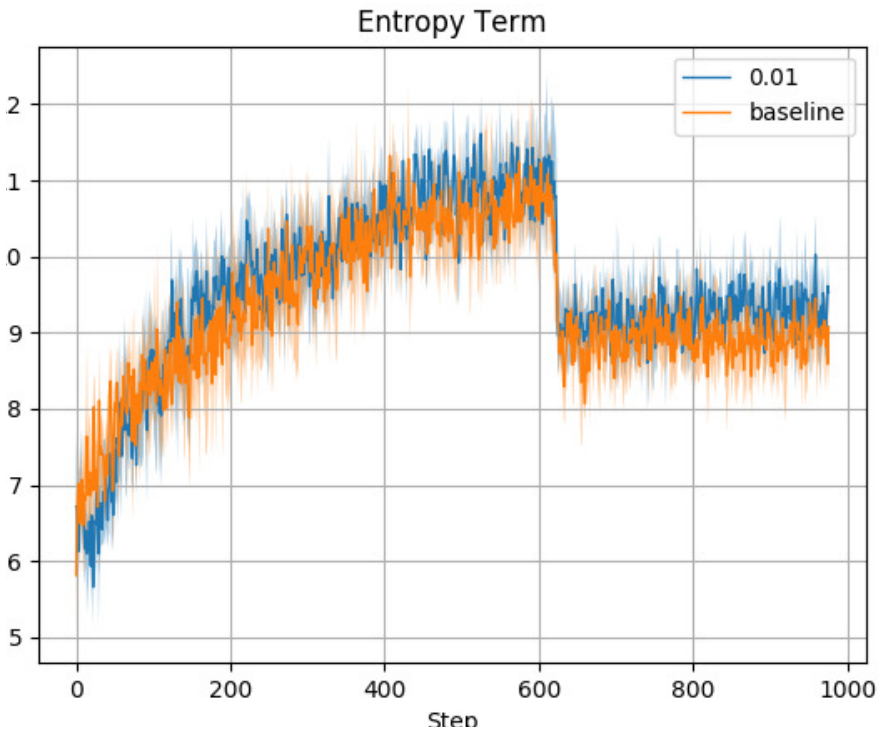
which results in $n \geq 60$. Therefore, a reasonable small number of samples may result in a good hyperparameter set (ZHENG, 2015).

We performed a hyperparameter random search in the benchmark task, optimizing the following hyperparameters:

- **Timesteps per actorbatch** – The number of timesteps collected to perform learning updates. Possible values: [128, 256, 512, 1024, 2048, 4096, 8192];
- **Batch Size** – The number of timesteps used to compute a gradient step. Possible



(a)



(b)

FIGURE 8.7 – Evaluation of entropy term.

values: [64, 128, 256, 512, 1024, 2048, 4096, 8192], constrained to be less than the number of timesteps per actorbatch;

- **Epochs** – Number of gradient steps performed in each learning update. Possible values: [5, 10, 15, 20, 25, 30];
- **Clip Param** – The clipping parameter of PPO Surrogate Loss (Subsubsection 5.7.3.1). Possible values: [0.1, 0.11, 0.12, ..., 0.28, 0.29, 0.3];
- $\boldsymbol{\gamma}$ – Cumulative reward's discount factor (Subsection 5.3.3). Possible values: [0.8, 0.9, 0.99, 0.999]; and
- $\boldsymbol{\lambda}$ – Generalized Advantage Estimation parameter (Subsection 5.6.3). Possible values: [0.9, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.0]

Therefore, we pruned the search space to only a few values for each hyparparameter, resulting in 310464 possible combinations, which is feasible to run a random search. Other hyperparameters, such as learning rate and gaussian exploration rate drastically changes the reward curve and are easier to optimize manually - which has been done for the experimentation described in Chapter 7. We fixed an amount of 1 million timesteps and ran 60 random sets, with 3 seeds of each one, in order to compute a confidence interval. All sets are in our repository of logs. Figure 8.8 shows all the reward curves through running time.

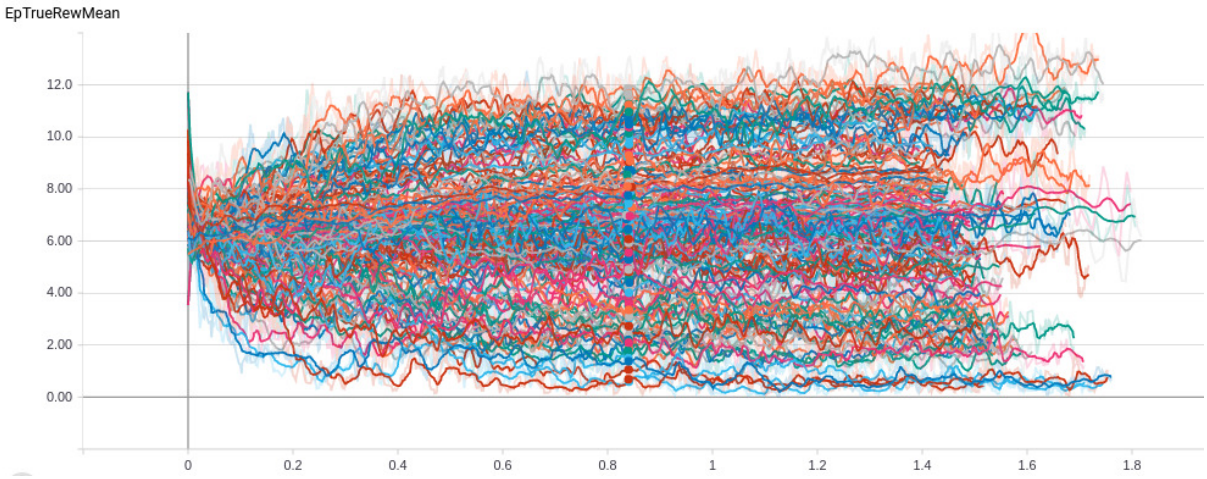
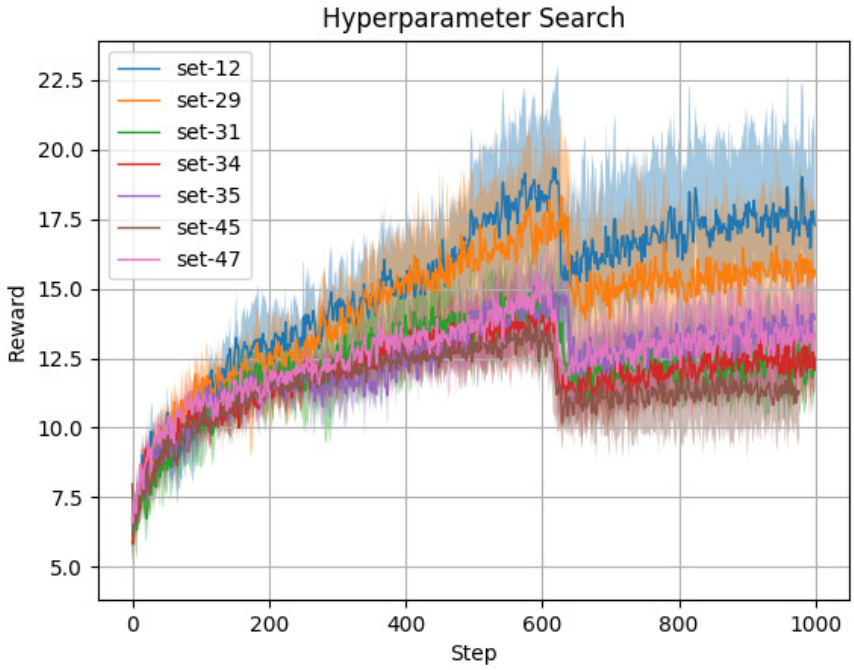
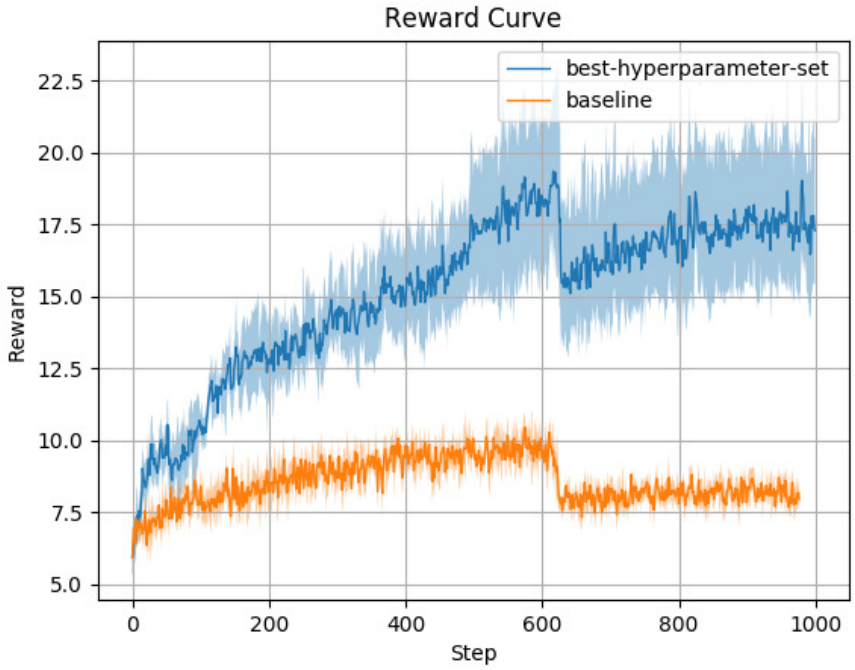


FIGURE 8.8 – Hyperparameter random search results.

Figure 8.8 clearly states how sensible the PPO algorithm is regarding of its hyperparameters. To choose the best sample, we filter 7 sets accordingly to the reward curves and training for a longer time. Results are shown in Figure 8.9a. We also compared the best set with the baseline model in Figure 8.9b.



(a)



(b)

FIGURE 8.9 – Evaluation of best hyperparameters sets and comparison with baseline.

Unlike the previous analyses where the improvement was slight, the hyperparameter search resulted in a much better reward curve, more than doubling the final reward. We adopted this best set, which is shown in Table A.4.

8.1.8 Exploration Restarter

In continuous action space, PPO can prematurely shrink the exploration variance, which leads to slow progress and may make the algorithm prone to getting stuck in local optima (HÄMÄLÄINEN *et al.*, 2018). In our case, we observed that the variance weights were stuck in a region with high values. In our experimentations, the gaussian exploration starts with a very low value to not harm the weights initialization setup and because our reward is very delayed (we just receive the signal if the robot is able to perform the whole motion kick and precisely kicks the ball). Thus, we need a way to ensure consistent exploration during RL training.

To overcome this challenge, we implemented a exploration restarter. Basically, when retraining RL models, we reinitialize the gaussian exploration weights randomly, instead of using the learned values. Firstly, we trained Imitation Learning's policy during 10 million timesteps. We then experimented several exploration rates in the subsequent RL training phase. Figure 8.10 shows the results. We considered only the 600 samples to build the curves in order to facilitate visualization.

Firstly, we observed that conducting a second phase of RL training with the trained exploration and a restarter of rate $1e-05$ lead to similar reward curves, where we conclude that the PPO barely changes the scale of exploration. This can avoid premature shrinks, but slow progress in cases of policies with low stochasticity.

Furthermore, we observed that the lower the exploration restarter rate, the better the initial reward becomes. This is straightforward: after a first session of RL training, our policy already has a good reward, and higher exploration "sacrifices" initial reward aiming to converge to a near better parameter set. The problem is that the time to converge can be longer and could stuck on worse local optima. However, lower exploration will lead to higher exploitation and a stagnant reward curve.

Our conclusion in this experiment is that the exploration restarter helped our training, but its rate needs to be tune accordingly to exploration-exploitation trade-off. In case of this experiment, we consider a great exploration rate between $1e-6$ and $1e-7$. Higher values will basically fit the same reward of the first training phase and lower values will basically stuck in initial reward.

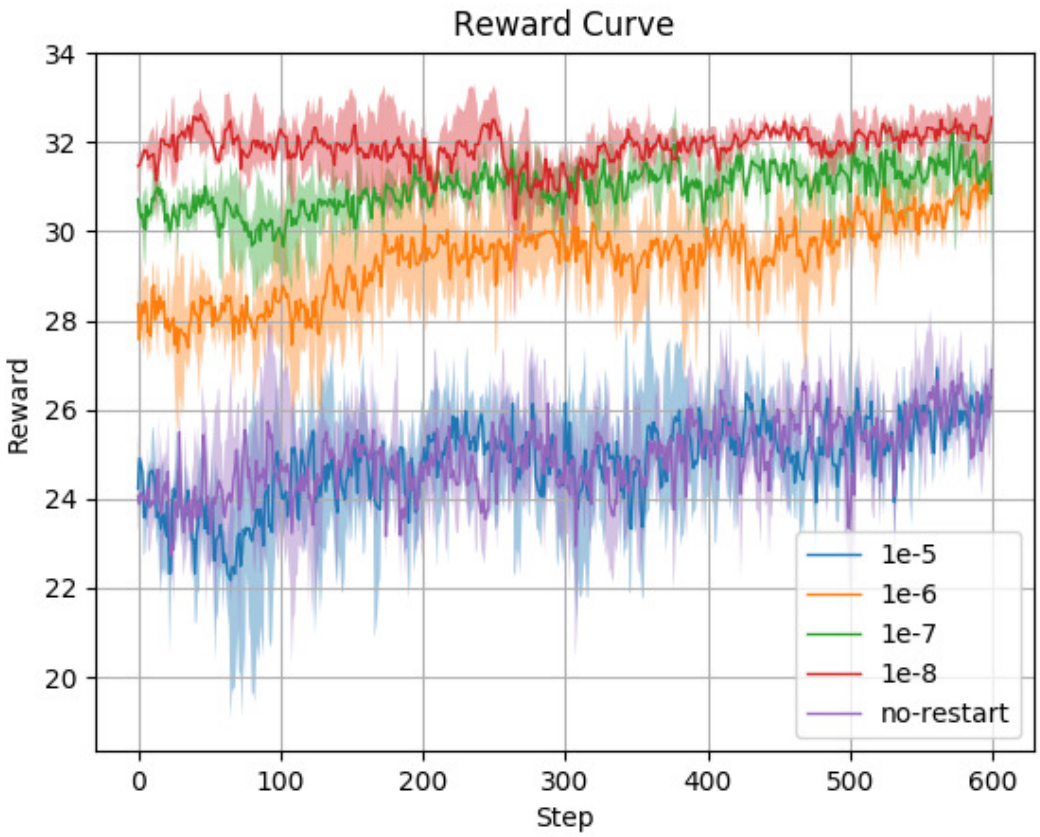


FIGURE 8.10 – Reward curve when conducting a second RL training phase with exploration restarter.

8.1.9 LSTM Model

A common architecture used in sequential problems is recurrent networks. In our case, we have a time-series composed by an action (set of desired joints) for each timestep. Hence, it is natural to propose a network architecture that uses recurrent models.

Theoretically, if we consider a Markov Decision Process, the recurrence is not necessary, given the Markovian property which says that a particular state encodes all information independently of the past set of states and actions – in other words, the future is independent of the past given the present (see Subsection 5.3.1). Thus, we would not need previous information to perform better training.

However, in practical situation, we are dealing with Partially Observed Markov Decision Processes (POMDPs), where the Markovian property is just "softly" accepted. In this case, using past information could lead us to better results in optimization. As illustration for our problem, the actual state does not have information regarding the joints velocity, which is very important to a physics model (PENG *et al.*, 2018). Using previous actions could help the network to infer such values by training.

We decided to try an LSTM model, as a recurrent architecture deeply studied in the context of Reinforcement Learning (BAKKER, 2001) and used in several recent approaches (OPENAI, 2018; VINYALS *et al.*, 2019).

In this new approach, we considered a neural network with an LSTM layer and a posterior dense layer with the actions. In this network, the input is the 10 previous actions, with the same output as the previous cases. The code is released in the main repository of this work.

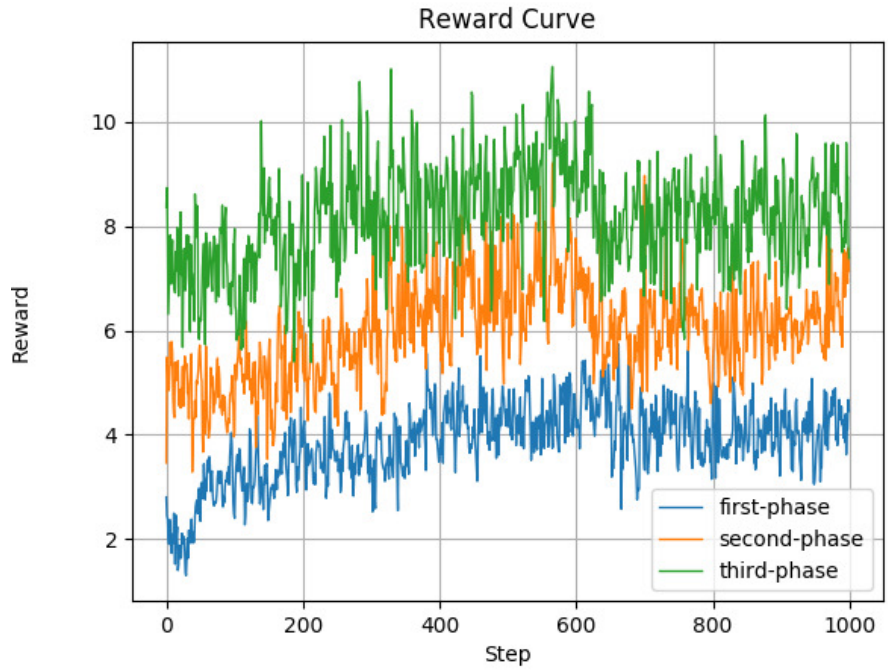
These design choices reasonably change our problem, particularly in the approach used to Imitation Learning. In other words, the problem now is harder to train. Firstly because we have state space much higher (10 observations of 22-dimensional joints set). Furthermore, we are training an LSTM network, which is known as by being harder to train due to its natural sequentiality.

Finally, considering joints as inputs leads us to accumulate error during the kick, which causes stronger covariate shift on the policy distribution.

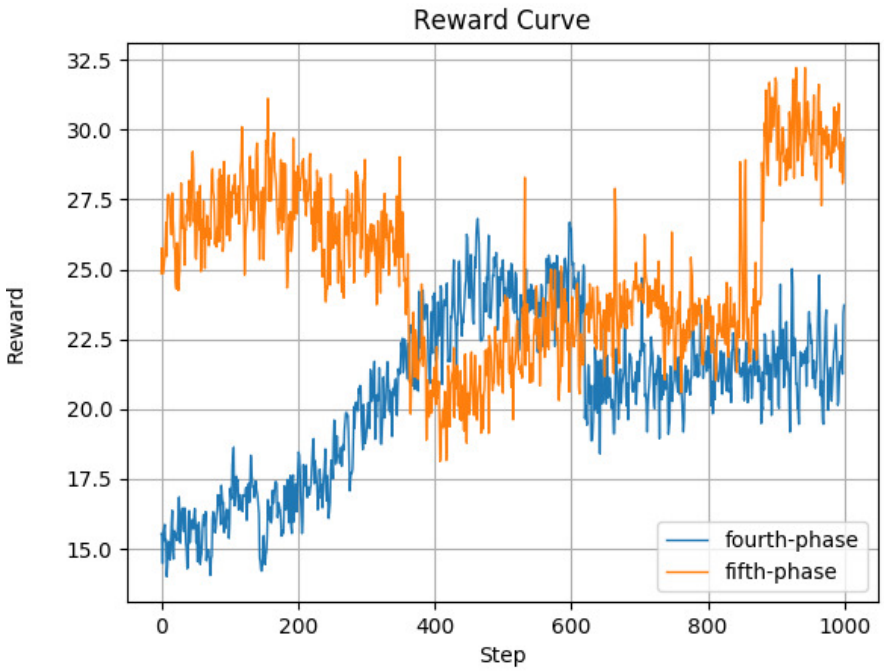
Despite of all these challenges, we have trained the network using the same dataset. After tuning the hyperparameters, the Imitation Learning step resulted in a policy that has lower initial reward. This fact compromised a fair comparison between both models.

We then performed several RL trainings in order to improve our policy. First, we performed three sequential cycles of RL training (Figure 8.11a) with the same hyperparameters, where the curves increase softly. At the end, the reward was better than the initial one from the other architecture used in previous experiments. We then controlled

the exploration using the restarter in two additional training phases (Figure 8.11b).



(a)



(b)

FIGURE 8.11 – Training a LSTM network for kick benchmark.

If we look to the final policy, we observe a reward near to the best result from Figure 8.10. Therefore, despite the harder process to train, the LSTM policy achieve similar results. However, we opt to not use it for some clear reasons: first, we use much more RL

training to achieve such value, being much more costly; furthermore, as the reward curves evidence, there is no gain to use such model. So, we prefer to keep the simplest one.

Finally, we do not consider that LSTM network performed worse. Actually, we changed the problem to use this model. Another way to conduct this experiment is to consider the timesteps values as inputs of the recurrent model, not the joints. This will keep the model smaller and avoid the additional covariate shift caused by the joints as inputs. We leave this as future work.

8.1.10 Final Model

Wrapping up this Section, we describe the final model based on the improvements achieved by the long list of experiments conducted.

Our final model has actor and critic networks as 2-hidden, fully connected layers, with 64 neurons in each, and hyperbolic tangent activation. Moreover, the Imitation Learning step resulted in the mean absolute error as small as possible (10^{-4} of order of magnitude); we used weight initialization in both networks, and scale reward by multiplication factor of 0.01. We added the entropy term scaled by the multiplication factor of 0.01 as well. We used the best results of hyperparameter randoms search (described in Table A.4). We conducted three phases of RL training, using exploration restart with rate of $10^{-6.5}$ and $10^{-7.0}$, respectively. We ran approximately 10 million timesteps in each training phase. The result is shown in Figure 8.12.

As we can see, the reward curve from first phase grows surprisingly. Restart exploration also give us some reward, resulting in a final model with almost five times more reward than initial values and approximately 3.5 times more reward than the best result of baseline model.

As final comparison, we picked a final policy from baseline and final models. We then considered a test scenario, similar to that presented in Subsection 7.9.4: we ran a hundred of kicks, and considered a "wrong" kick when the ball does not achieve at least 0.5 m/s. However, we made some modifications to agent's behavior. Firstly, we started the agent 0.2m from the ball. Secondly, we executed its high level kick behavior to adjust its joints before perform the keyframe motion. The idea is to approximately fix the initial state for both cases and perform a fair comparison.

Table 8.1 shows the results in the same way as Table 7.6 and using effective evaluation. Nevertheless, we cannot make a fair comparison due to changes in the test scenario. Between baseline and final models, we can see a huge difference in such evaluation both in forward and height distances.

The accuracy of baseline and final models are 63% and 85%, respectively. The huge

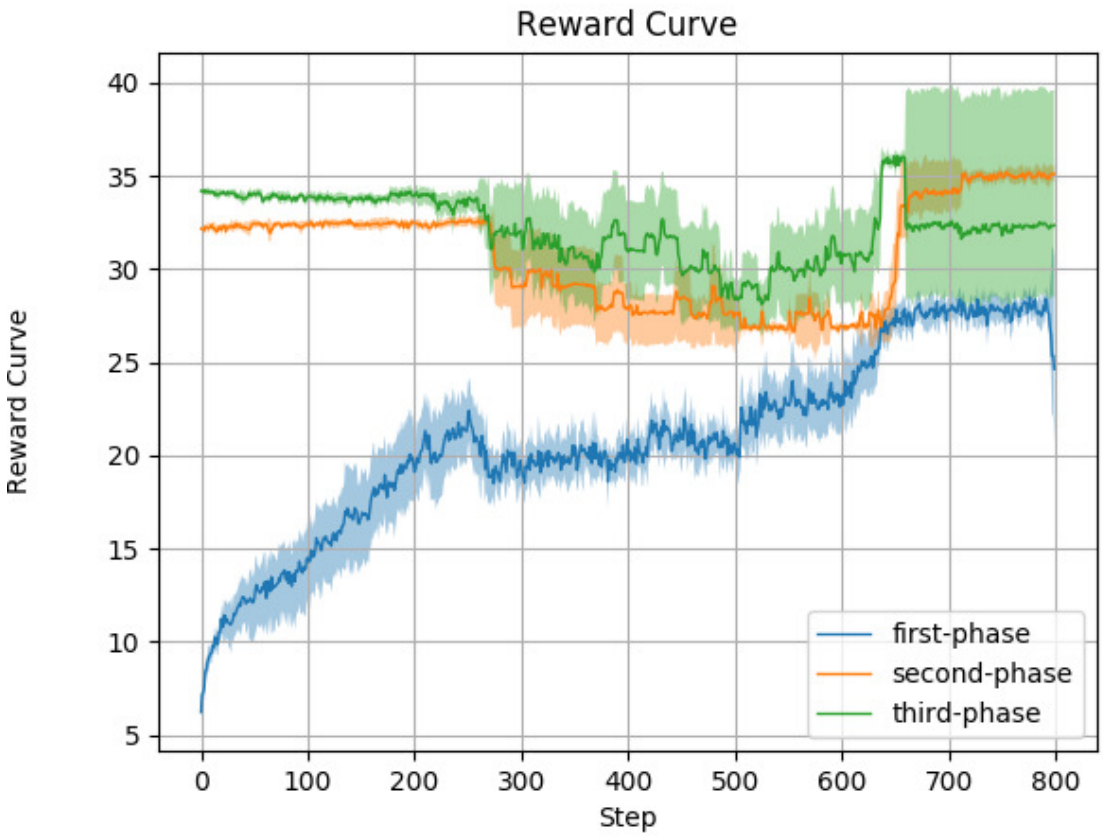


FIGURE 8.12 – Reward curve when conducting a second RL training phase with exploration restarter.

drop, especially for the case of baseline model if compared to results in Table A.3, is justified by the change on test scenario. This causes a covariate shift between training and test distributions, harming both models. However, as we are comparing the models from this Section, we also see improvement in the accuracy numbers.

TABLE 8.1 – Kick Benchmark Comparison - Effective Evaluation

Kick Type	**Statistics**			
	Distance X(m)		**Distance Z** (m)	
	Mean	***Std***	***Mean***	***Std***
Baseline Model	7.51	2.17	0.04	0.001
Final Model	**12.38**	**4.75**	**1.19**	**1.04**

Finally, we are comparing kick results that comes from distributions that can not be represented sufficiently by simple statistics. Therefore, we show in Figures 8.13 and 8.14 the distributions from baseline and final models, respectively. The graphs in the left are scatter plots where we sorted kicks results. In the right side we have the histograms.

The distributions reconfirmed that models suffers with this covariance shift and the huge increase in distance in X and Z axis. However, the final model is highly biased to left (positive in Y axis), as result of weak penalization in comparison to the other factors in reward curve. We concluded that this task scenario and reward shaping could be improved to search effective policies. On the other hand, we also concluded that the analysis and techniques proposed by this Section were fundamentals to improve the model substantially – and they will make part of our methodology in the next sections.

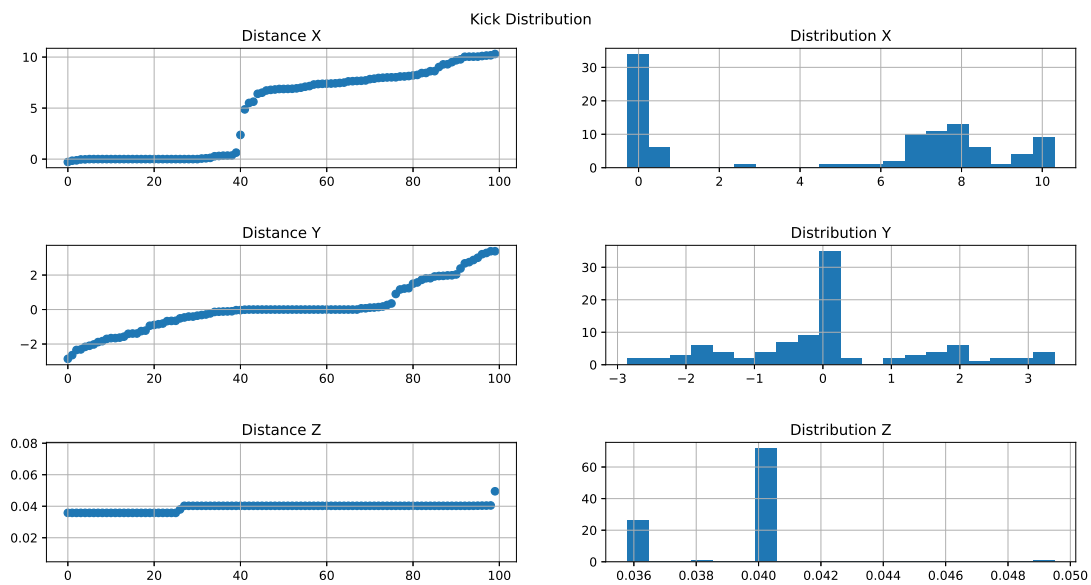


FIGURE 8.13 – Kick distribution from baseline model.

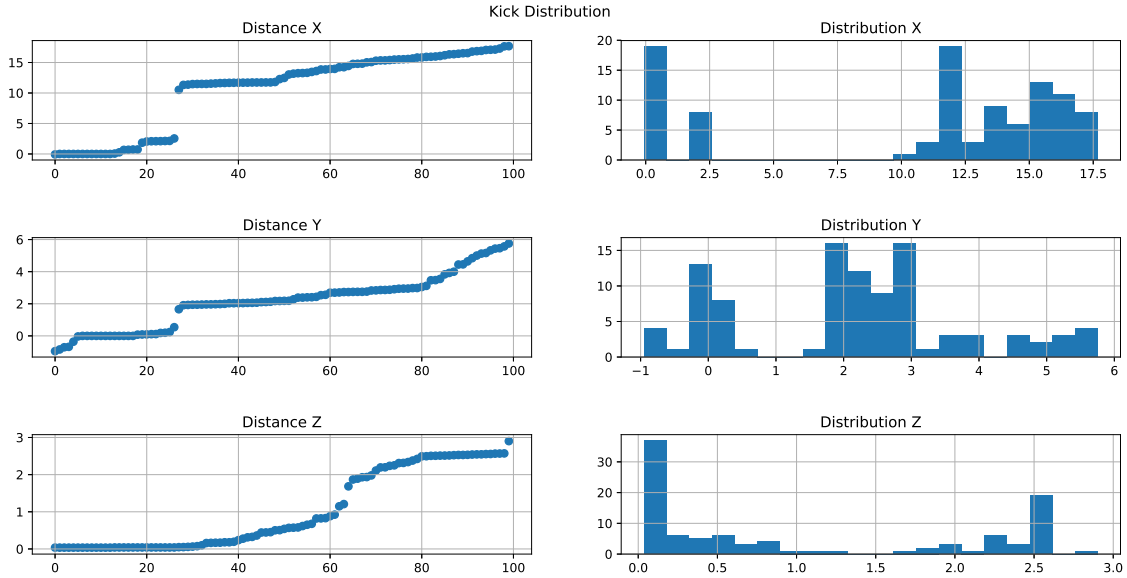


FIGURE 8.14 – Kick distribution from final model.

8.2 Bottom-Up Meta-Policy Search (BUMPS)

In previous experiments, we used RL training – and shaped the reward – to obtain the longest and highest kick possible. However, in game situation, it is better to have precise kicks for several distances, in order to obtain a passing-level control. Therefore, we now change our focus to this more challenging problem.

More than just design a framework to learn each kick distance in isolation, we propose a meta-learning approach that uses the knowledge from some precise kicks previously trained (namely meta-training tasks) and learn a meta-leaner that quickly adapts to other unseen kick distances (meta-test tasks). In this way, we do not need to learn task by task, but only some expert policies and then generalize them to other ones.

We called our approach "Bottom-Up Meta-Policy Search" (BUMPS), because it uses the knowledge of expert policies in single tasks to train a contextual meta-policy that generalizes well for unseen tasks. The method is based on the idea of "Bottom-Up Learning" (SUN, 2012), where learning happens firstly in implicit knowledge and then explicit knowledge (i.e, through "extracting" implicit knowledge).

Before describe the proposed algorithm, it is important to explain why we consider BUMPS as a meta-learning algorithm rather than as a multitask or transfer learning technique. Accordingly to Chen e Liu (2015), in multitask learning, given a set of tasks $\{\mathcal{T}\}$, the objective is to co-learn all tasks simultaneously through some shared knowledge – but not necessarily generalizing to unseen tasks. On the other hand, transfer learning tries to learn well a target task $\mathcal{T}_{target}$ given a set of source tasks $\{\mathcal{T}\}$. Note that its goal

is to learn well only for the target task, while the learning of the source tasks is irrelevant. Finally, in contrast to previous techniques, meta-learning seeks to learn a representation from a set of source tasks $\{\mathcal{T}\}$ that is able to generalize to unseen tasks or at least to quickly adapt to learn them – which is the purpose of BUMPS.

The idea of contextual meta-policy comes from the fact we express a policy as $\pi_\theta(a_t \mid s_t, \omega)$, i.e, we give the task context as input. The meta-training phase then explore the task structure of humanoid kick motion to create a rich representation of shared features based on contextualization. Intuitively, we have the following problem: if we learn to kick to distances a and b (such as $a < b$), then we implicitly knows that the representation of kick distance c is something approximately "between" a and b. This meta-policy is more robust as it learns MDPs with other contexts, better creating the geometric idea of contextualization.

During meta-training, instead of optimizing a meta-objective such as in MAML (FINN *et al.*, 2017), we use a simple first other Behavior Cloning loss within respect to aggregated dataset $\mathcal{D}$:

$$J(\boldsymbol{\theta}, \mathcal{D})_{BC} = -\mathbb{E}_{(s_t,a_t)\sim\mathcal{D}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \tag{8.2}$$

In this way, we reduce the agent-environment interaction, thus improving sample efficiency. Additionally, we use gradients from supervised learning, which have lower variance and therefore are more stable than RL gradients (NOROUZI *et al.*, 2016). Our fast adaptation strategy comes from an alternative interpretation of Reptile algorithm, described in Section 6.3.2. Basically, during meta-training, we optimize a set of parameters that is near optimal to solve all meta-training tasks. The generalization comes from the fact that such parameters can also solve unseen tasks due to proximity between optimal policies with near contexts.

However, we can not guarantee that the best policy for a task will be in the right context, but we can sample some policies in the neighborhood and evaluate them accordingly to a metric, and then copying the best policy to a specific context. We call such approach as policy filtering. We present Bottom-Up Meta-Policy Search Meta-Training in Algorithm 10 and the Policy Filtering as Meta-Testing in Algorithm 11.

Algorithm 10 Bottom-Up Meta-Policy Search Algorithm (Meta-Training)

Require: Distribution over tasks $p(\mathcal{T})$

1: Sample batch of meta-training tasks $\mathcal{T}_i \sim p(\mathcal{T})$
2: Initialize $\mathcal{D} = \{\}$
3: **foreach** meta-training task $\mathcal{T}_i$ **do**
4: Train an expert policy $\pi^*_{\mathcal{T}_i}$ for $\mathcal{T}_i$ using RL
5: Sample K trajectories $\mathcal{D}_{\mathcal{T}_i} = \{(s_1, a_1, ..., a_H)\}$ using $\pi^*_{\mathcal{T}_i}$ in $\mathcal{T}_i$
6: Contextualize trajectories using $\mathcal{T}_i$ goal $\mathcal{D}_{\mathcal{T}_i}(\omega) = \{\mathcal{T}_i : \mathcal{D}_{\mathcal{T}_i}\}$
7: Aggregate $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_{\mathcal{T}_i}(\omega)$
8: **end for**
9: Initialize $\boldsymbol{\theta}$ randomly
10: **while** not done **do**
11: Sample mini-batch of expert policies dataset $\mathcal{D}^* \sim \mathcal{D}$
12: Evaluate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \mathcal{D}^*)_{BC}$ using Equation (8.2)
13: Update parameters with gradient descent $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \mathcal{D}^*)_{BC}$
14: **end while**

Algorithm 11 BUMPS Policy Filtering (Meta-Testing)

Require: Meta-Policy $\Pi_{\boldsymbol{\theta}}$

Require: Meta-Testing task $\mathcal{T}_{test}$

1: Sample single-task policies $\{\pi^i_{\boldsymbol{\theta}}\} = \Pi_{\boldsymbol{\theta}}(\omega)$, where $\omega \sim p(\mathcal{T})$
2: **foreach** policy $\pi^i_{\boldsymbol{\theta}}$ **do**
3: Evaluate $\pi^i_{\boldsymbol{\theta}}$ accordingly to a metric loss $\mathcal{L}^i_{test}(\boldsymbol{\theta})$
4: $\mathcal{L}_{test}(\boldsymbol{\theta}) \leftarrow \mathcal{L}_{test}(\boldsymbol{\theta}) \bigcup \mathcal{L}^i_{test}(\boldsymbol{\theta})$
5: **end for**
6: Filter $\pi^*_{\boldsymbol{\theta}} = \arg\min_{\pi_{\boldsymbol{\theta}}} \{\mathcal{L}_{test}(\boldsymbol{\theta})\}$

We consider separate algorithm into meta-training/testing phases due to the possibility of running online adaptation. Additionally, in these cases, we can optimize meta-testing phase by sampling policies and keeping the tuples (s_t, a_t) just once, and then evaluating the metric loss for each meta-test task when needed. Finally, it is worth to mention that the filter operation can be conducted in two ways: first, we can map from required task to the best context; second, by sampling trajectories using optimal policy and then conducting Behavior Cloning to a new neural network.

Figure 8.15 illustrates how BUMPS works. Before detailing how experiments are conduct in each step of this algorithm, we will explain why BUMPS works.

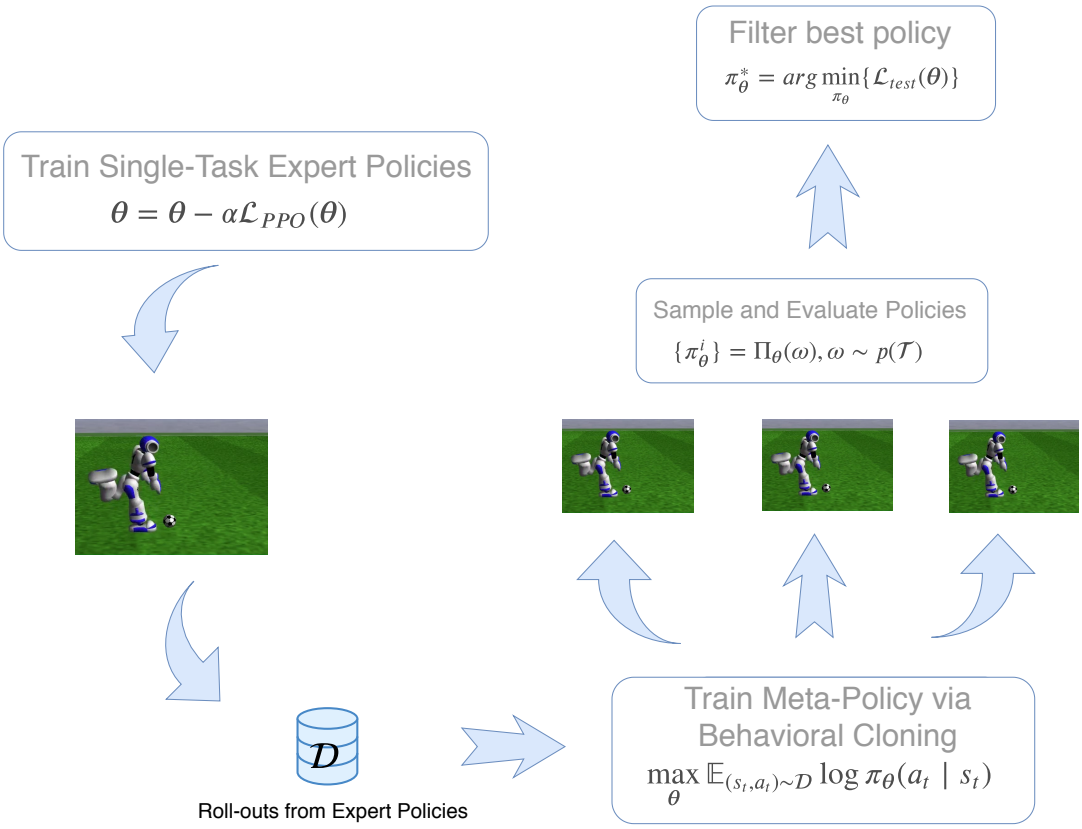


FIGURE 8.15 – Bottom-Up Meta-Policy Search.

8.2.1 Explaining BUMPS: Meta-Gradient through Contexts

As we described previously, BUMPS adaptation is inspired in Reptile algorithm, which relies on first-order derivatives to minimize a loss over all meta-training tasks and then results in a weight initialization that can quickly adapts to each task by using few gradient steps. Its advantage is due to simplicity – it is surprisingly that it works as a meta-learning algorithm – with a solid theoretical explanation.

As we can observe in Algorithm 8, we can superficially describe Reptile as a strategy where we conduct few steps towards a task and then calculate a specie of "meta-gradient" from the parameters before and after such steps. In this way, it uses information of higher order derivatives. This process keeps happening across tasks, which leads to a final representation that is close (in Euclidean distance) to the optimal solution manifolds of training tasks. For the sake of clarity, an optimal solution manifold is a geometric concept which we can describe as a surface in parameter set space which leads to an optimal policy configuration for the task.

The meta-gradient learning rate, as well as the number of inner gradient steps, are hyperparameters. Therefore, we can analyze some important sets to conclude significant insights. Firstly, if we consider the meta-gradient learning rate as one (this value is already used in paper experimentations (LEE, 2018)), Reptile is just replacing the parameters from before inner gradient steps to after. Figure 8.16 illustrates this situation. As we can observe, it is like a huge (but precise) meta-gradient step towards new parameters.

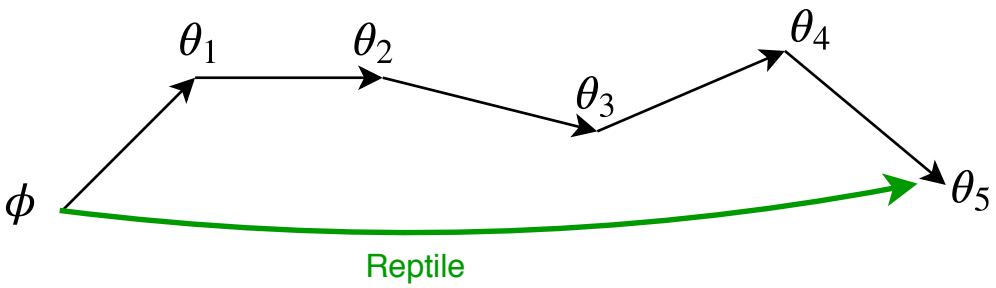


FIGURE 8.16 – Reptile meta-gradient when outer learning rate is 1.

Furthermore, if we consider the number of inner gradient steps as infinitely large (i.e, enough to converge to task optimal manifold), we basically have a standard training procedure with several phases, one for each task. This probably will not converge (it should overfit the last task), but we can train jointly all tasks by considering a contextualization input.

We then join this two interpretations to result in the fast adaptation strategy to BUMPS. Firstly, we will simply train a properly network using all meta-training tasks

roll-outs, using task contextualization. This basically solves meta-training tasks using the meta-policy in each meta-training task context. Furthermore, we can assume that a geometric representation of contextualization closely relates to geometric representation of optimal manifolds (i.e, we can assume the representation of a 7.5m kick is something "between" 7.0m and 8.0m). Thus, by optimizing meta-training tasks jointly we will also have good representations for meta-testing tasks that are close to them.

However, as stated previously, the representation of meta-testing tasks will not necessarily converge to its optimal manifold, then requiring the policy filtering approach. Although we assume this close relationship between context and manifold geometry, there is no linearity in the latter. But we then can sample several contexts in the neighborhood and evaluate them to check if they are inside such optimal manifold.

When we found the context that is closer to such manifold (i.e, better solves the task), we then apply the idea of meta-training gradient from reptile, but instead using it from parameter sets before and after few training steps, we use between to context representations.

The major drawback of BUMPS is because it relies on generalization of tasks, where is likely to happen just in situations where we can assume a geometric representation of contexts. However, if we compare to Reptile, we can also conclude that the performance of Reptile is assured only on meta-training tasks. In the case of meta-testing tasks, this algorithm also relies in the idea of near or overlapped manifolds.

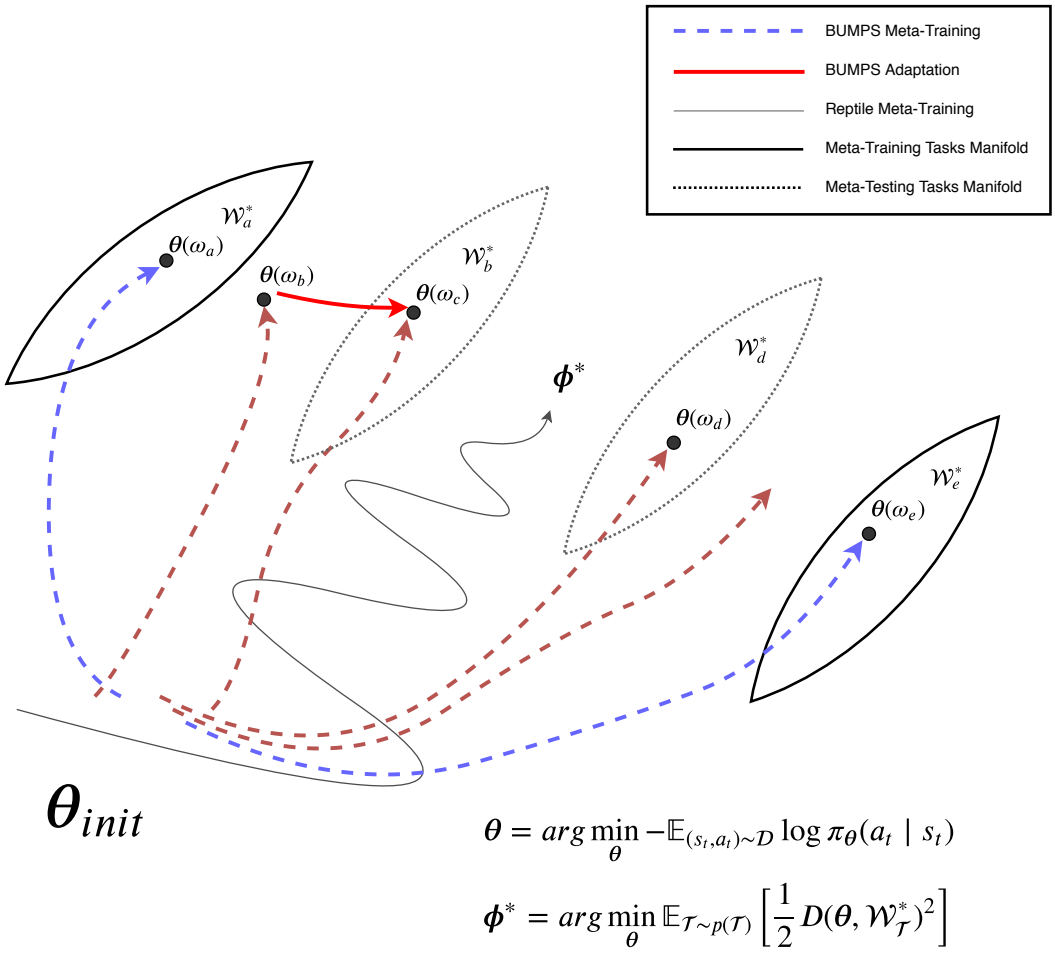


FIGURE 8.17 – BUMPS and Reptile Meta-Training.

Figure 8.17 illustrates BUMPS and a Reptile standard training in parameter set space, considering five tasks a, b, c, d and e and optimal manifolds for a, b, d and e. The meta-policy trained by BUMPS and Reptile are parameterized, respectively, by $\boldsymbol{\theta}^*$ and ϕ^*. $\boldsymbol{\theta_{init}}$ represents an initial parametrization of policies before training and the notation $\boldsymbol{\theta(\omega_{task})}$ represents the parameterization $\boldsymbol{\theta}$ in the context of a specific task. As we can observe, Reptile training will lead to a parameter set that is approximately equally close to optimal manifold of meta-training tasks, but not necessarily inside any one.

On the other hand, BUMPS will conduct their parameters to different regions depending on the context. In the case of meta-training tasks, it will solve them (represented by blue dashed arrows). In the case of meta-testing tasks, however, it will lead to a variety of regions (red dashed arrows): in some cases inside optimal manifold (task d), in others only near. To address this problem, we can just sample some contexts and then perform such "meta-gradient" towards the context inside optimal manifold (as represented by red full arrow).

In light of this overview, we can then proceed to detail how each part of the algorithm was implement in experiments.

8.2.2 Training Single-Task Expert Policies

In our context, the first step of BUMPS is to train precise kicks for the meta-training tasks sampled. We chose as task interval kicks with distance between 7 and 18 meters. Any task can assume a distance goal inside this interval with precision of decimeters, i.e, 7.1, 7.2, ..., 17.9, 18.0.

We sampled kicks goals, as meta-training tasks, uniformly distributed in this interval and spaced half a meter, i.e, 7.0, 7.5, ..., 17.5 and 18.0. The other possible distances are used as meta-test tasks.

To train such expert policies, we need to formulate an optimization task to reinforce precision after the ball stops. The first difference is that we need a longer horizon to evaluate the ball position error at the terminal state, thus using velocities to shape reward is not ideal. Additionally, we can not use a varied-length horizon because it enforces to keep ball moving to get more reward, rather the being more precise.

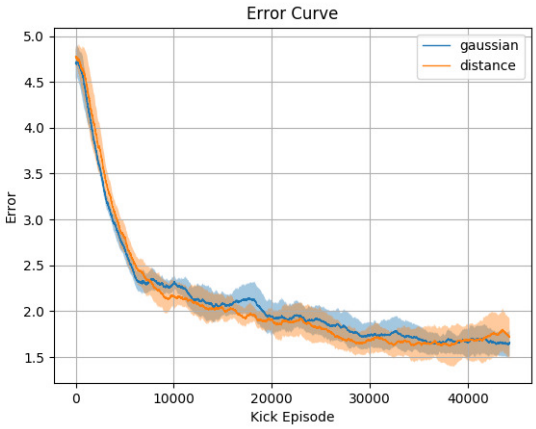
Therefore, we fixed the task horizon to 300 steps and the reward is calculated by reinforce precision of target distance at each timestep:

$$R(\boldsymbol{\theta}) = a \times f(targetDistance - distanceError), \tag{8.3}$$

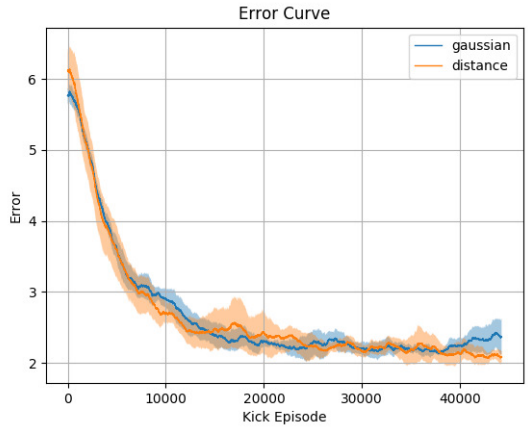
where a, f, $targetDistance$, and $distanceError$ are, respectively, a scaling factor, a function transformation that shapes reward space, the target position and the error between target position and ball's final position. In our case, we tried a simple linear function (which means reward is modeled by Euclidean distance) and a gaussian density, centered in zero and with standard deviation of $targetDistance/3$. We conducted some experiments and found that both transformations conduct to similar results, as show in Figures 8.18a, 8.18b and 8.18c, where we optimized for distances of 8m, 10m and 12m, respectively. We decided to keep the simpler approach of linear function.

A major drawback of this reward is that the agent tends to surpass the target distance, because a high velocity will provide best rewards for initial steps, which can lead to better cumulative rewards at the end. This situation can be mitigated considering a longer horizon. Another solution is to reward just at the end of each episode, which can harm optimization due to it being sparse.

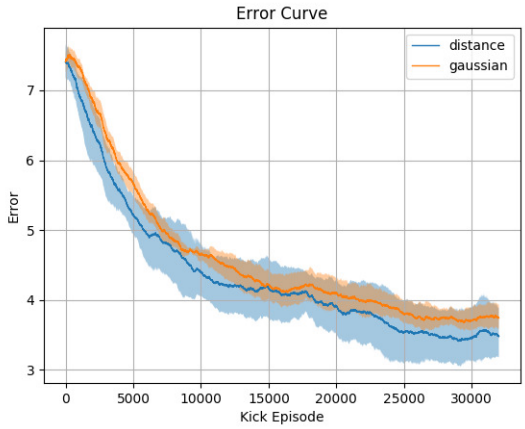
The previously described changes also affects how hyperparameters perform optimization, which lead us to conduct another random search. We chose a set of hyperparameters



(a)



(b)



(c)

FIGURE 8.18 – Smoothed error curves (and bootstrap confidence intervals) of RL optimization of 8 m, 10 m, and 12 m, respectively. The rewards modeled by gaussian and linear transformations results in approximately same performance.

to conduct all precise kick experiments, and the values are shown in Table A.5.

We conducted two generations of training. In the first, we analyze the performance of RL optimization in terms of precise kick. The second, on the other side, is used to collect dataset for next step of BUMPS.

In the first generation, we started by optimizing to 12 m of target distance, from the Behavior Cloning policy used in previous sections. The optimization resulted in the reward curve of Figure 8.19. We observe a well shaped curve, that increases monotonically and have a reasonable small confidence interval, which shows our optimization is consistent.

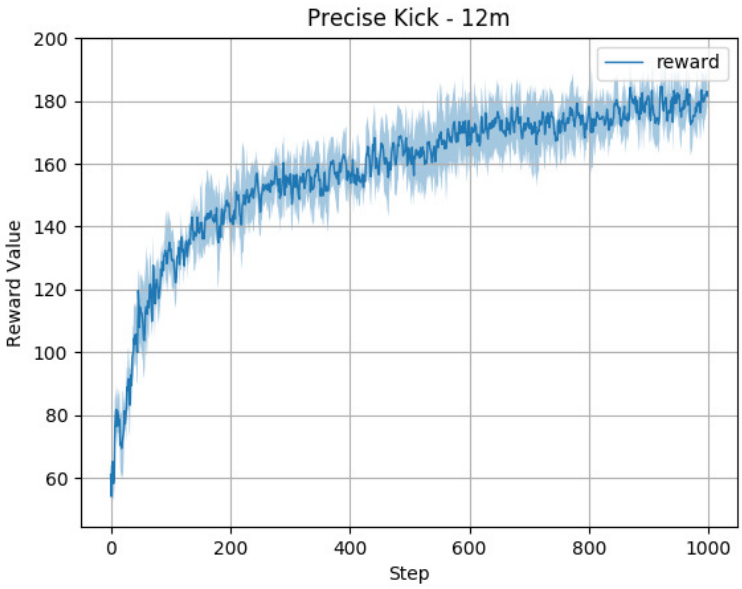


FIGURE 8.19 – Reward Curve for Precise Kick of 12m.

Only the reward curve is not enough to evaluate how precise is the final kick. Thus, we conduct kick evaluation, where we repeat 500 times the kick motion and calculate the distributions in Figure 8.20. The forward distribution has two components: one has low value and describes failed kicks due to distributional shift; and the second one, which interests us, and that is near centered on the target distance.

The previous kick resulted in a final mean error of **2.47 m**. The concern is on the y axis, where there is a huge bias. In order to adjust it and decrease this error, we ran another RL training, with the same context as the previous one, which resulted in the distribution of Figure 8.21, which has final mean error of **1.36m**.

These two training sessions show that when the agent starts from a policy that is near to the goal, the training can exploit the initial high reward and with less exploration achieve better performance – this thinking is the base of our meta-learning idea.

These results are not reasonable to consider as expert policies. Actually, we found

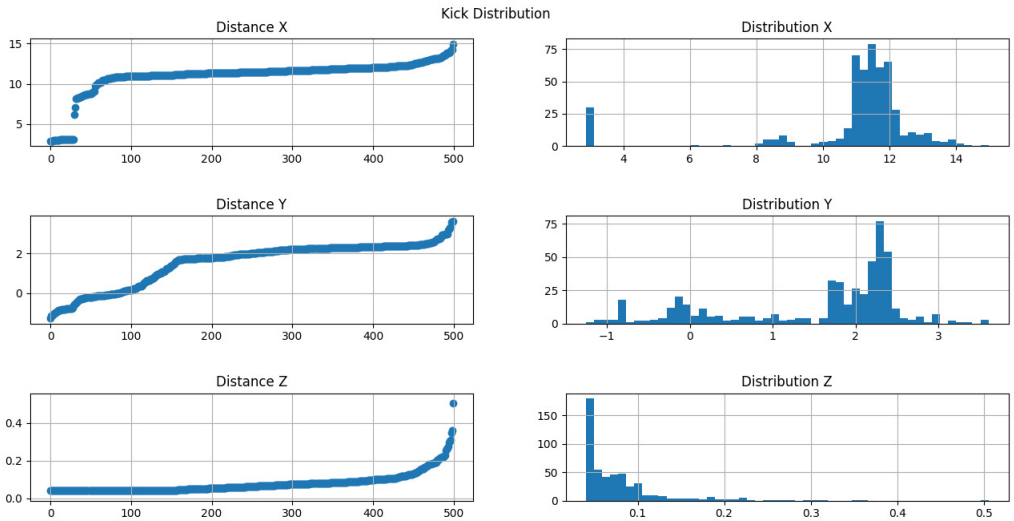


FIGURE 8.20 – Distribution of 12m kick.

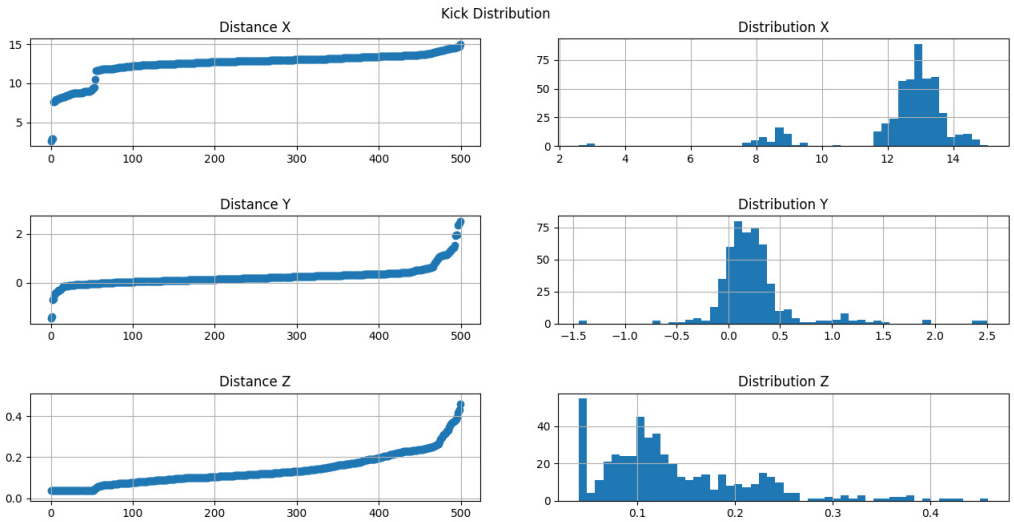


FIGURE 8.21 – Distribution of 12m kick after a second training session.

particularly challenging to obtain good expert policies to the interval between 12m and 14m. By experimentation, we hypothesized that this is because such interval is a transition between a kick that moves close to the ground to a kick that performs an oblique trajectory until a distant target.

In this first generation, we also optimized other kicks, from 4m to 18m. We found that our optimization procedure is limited inside the task interval previously defined. All the experiments conducted, policy configuration files and final distributions are released in our repository of logs[4]. For the sake of conciseness, we will not detail all experiments here.

During the second generation, we used the policies previously trained as initializers to train our meta-training tasks. In order to improve kicks' precision, we also considered a longer horizon of 500 cycles. Other training details we kept from last generation.

Figure 8.22 shows the error curves of all optimizations. For clarity, we omitted the legend describing each curve. Despite of this, we can see that all final errors were lower than 1 meter, which we consider reasonable as expert policy for passing level control and to use for meta-policy training.

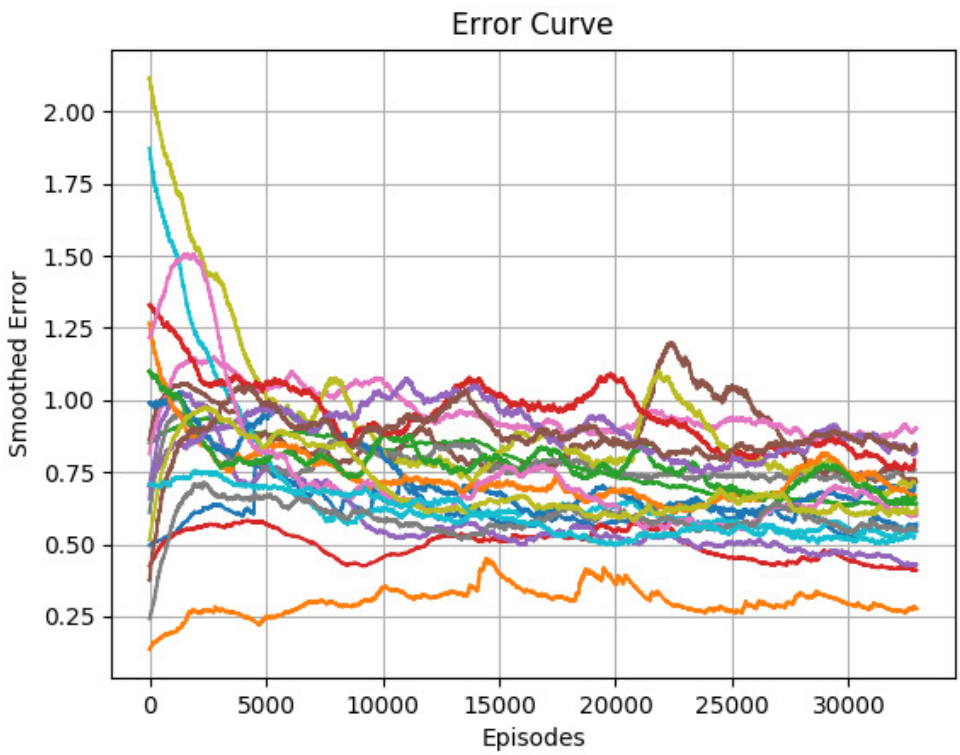


FIGURE 8.22 – Error curves from the second generation of precise kicks.

Finally, to establish a baseline of expert policies, we ran a kick evaluation with a

[4] https://bit.ly/2WhagKe

hundred of repetitions. Figure 8.23 shows the kick error (and 95% bootstrap confidence interval) from each case. We calculate a metric of accuracy which considers a "right" kick if it has target error lower than 2.5 meters. Otherwise, it is considered as a "wrong" kick. Considering the expert policies accuracy, we find a mean of **0.89** and standard deviation of **0.04**.

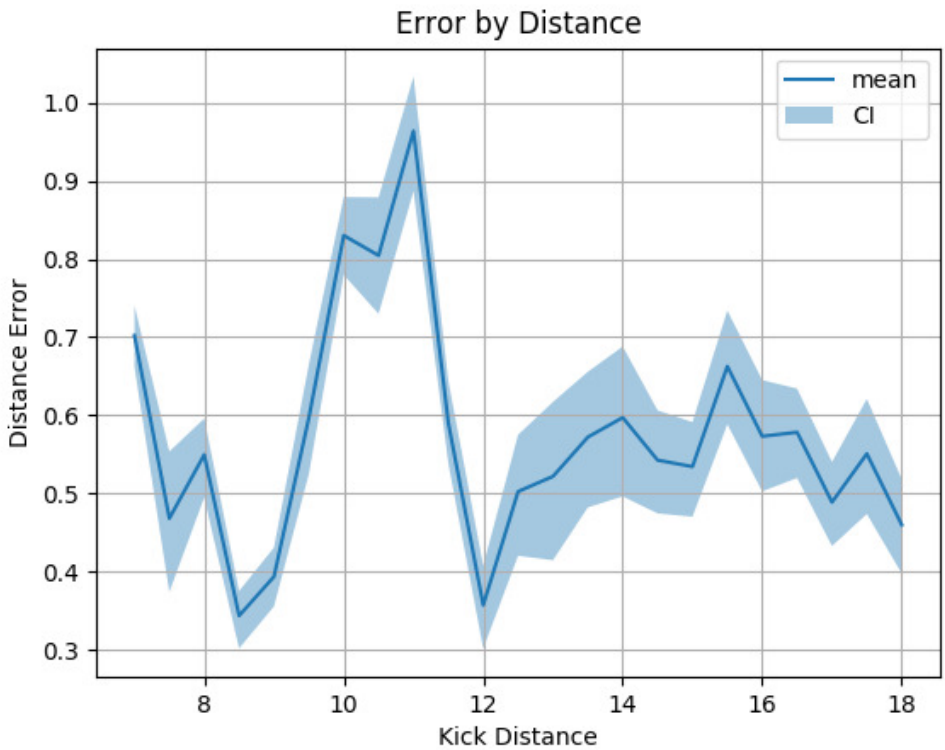


FIGURE 8.23 – Error statistics for each meta-training task.

To wrap up this Subsection, we evaluated the distribution of 12m from this second generation, represented by Figure 8.24. Comparing with Figures 8.20 and 8.21, there is a visible improvement in precision of main component of x axis.

8.2.3 Training Meta-Policy and Policy Filtering

During meta-policy training, we firstly collected trajectories from each expert policy and then added the contextualization, as and additional input that represented the kick desired distance. We trained during 150k epochs, using initial learning rate of 3×10^{-4}, decaying by step factor of 0.8 and minimizing mean absolute loss by using Adam. Additional hyperparameters can be found in released code[5].

In terms of neural network, we trained two feed-forward models: one with 4 hidden layers of 256 units each and other with 11 hidden layers with 128 units each. The idea

[5] `https://github.com/luckeciano/deep-rl-humanoid-motions-masters/tree/master/BUMPS`

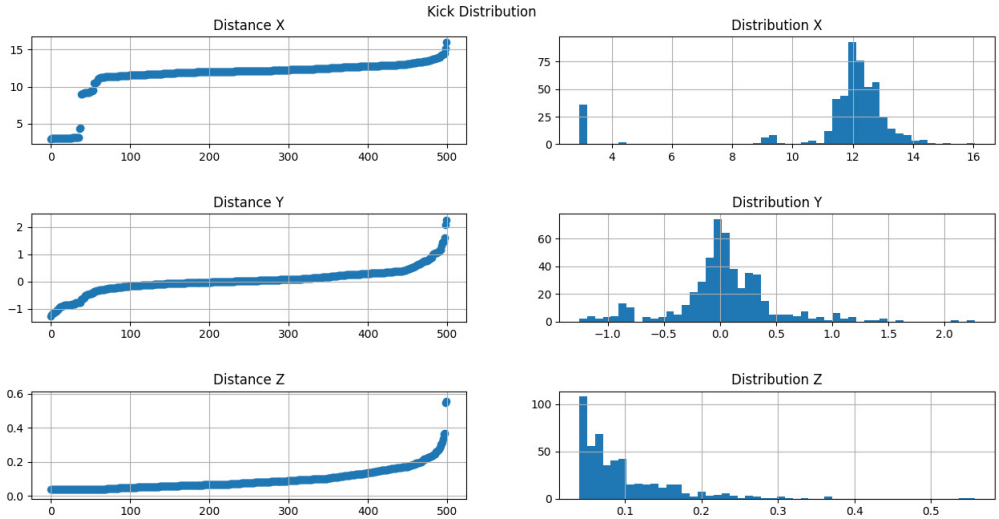


FIGURE 8.24 – Distribution of 12m kick in second generation.

of training two models instead a single one is just to emphasize that it is also possible to ensemble meta-policies, in order to conduct policy filtering and then improve final results. This architecture were found by some search regarding of hidden layers and units, evaluating accordingly to a metric we designed to take in account accuracy and error:

$$M_{\pi_{\boldsymbol{\theta}}} = \frac{1}{N} \sum_{i=0}^{N} acc_i \times err_i + (1 - acc_i) \times 2.5, \tag{8.4}$$

where N is the number of tasks, acc_i and err_i are, respectively, the accuracy and mean error of task i. It is worth to remember that 2.5 is the threshold value to consider a kick right or wrong to accuracy metric.

We tried an LSTM model with joints inputs, in the same way as described in Section 8.1.9. However, we did not find any improvement, at the cost of harder training. Our hypothesis is because we do want to generalize over context, and not only a better way to temporal modeling, which can be achieve by the depth of our chosen models.

Finally, we also tried a SNAIL-like architecture, with temporal convolutions and attention mechanism, but simplified to consider only the sequence of timesteps and desired distances as input, but without rewards and termination flags. Although easier to training in comparison to LSTM model, we have no improvements. We think that, by considering other necessary inputs, we can learn by reinforcement learn the policies and then improve representations – we let as future work.

To conduct Policy Filtering, we sampled all tasks in the previously established interval, spaced by 0.1 (i.e, $7.0, 7.1, \ldots, 17.9, 18.0$). The evaluation scenario of sampled policies is

the same conducted to evaluate single-task expert policies: evaluate kick error mean across a hundred repetitions. As metric loss, we consider the negative of mean absolute error between target and the final distance from kick. In terms of filtering operation, we just considered the map from required task to best context previously described.

8.2.4 Results

In order to evaluate BUMPS algorithm in terms of efficiency to achieve passing-level control, we need to answer three questions:

- **How BUMPS algorithm is compared with model-free RL to learn all tasks?** – This question validates the initial idea of treat precise kick as a meta-learning problem;
- **How much meta-training is able to generalize to unseen tasks?** – This question validates our assumptions and hypotheses described in Subsection 8.2.1; and
- **How much policy filtering improves meta-policy generalization?** – This question validates the idea of contextual gradients and, jointly to second question, validates our explanation from Subsection 8.2.1.

To start answering first question, we performed a RL training using PPO where for each episode we sampled randomly a task then conduct the kick for this desired distance. The reward was the same as for the case of single task expert policy, considering the sampled distance. We also used the same hyperparameters (see Table A.5) except for the network architecture, where we used the same model for meta-policy with 11 hidden layers with 256 units each. We firstly trained this network with 12m kick, just to give an initialization as a Hybrid Learning Model, necessary as stated by Imitation Hypothesis in Chapter 7.

We ran six seeds and presented all reward curves in Figure 8.25. As we can observe, in the context of our experiments, the RL training performed poorly, being not able to improve reward, but actually reducing it.

We hypothesize several reasons for it. Firstly, we do not have a kick representation for all contexts, just for once, which is not generalized across them. Secondly, each task has a different scale for reward, because the reward happens at each step and kicks with higher velocity tends to have higher reward in general. If we always consider the same sequence of tasks, we would need an incredibly longer horizon (or number of episodes before learning update) to consider all kicks during updates. Therefore, for the scale of

our task interval, RL training is not enough. As we will see later, BUMPS is able to learn all meta-testing tasks.

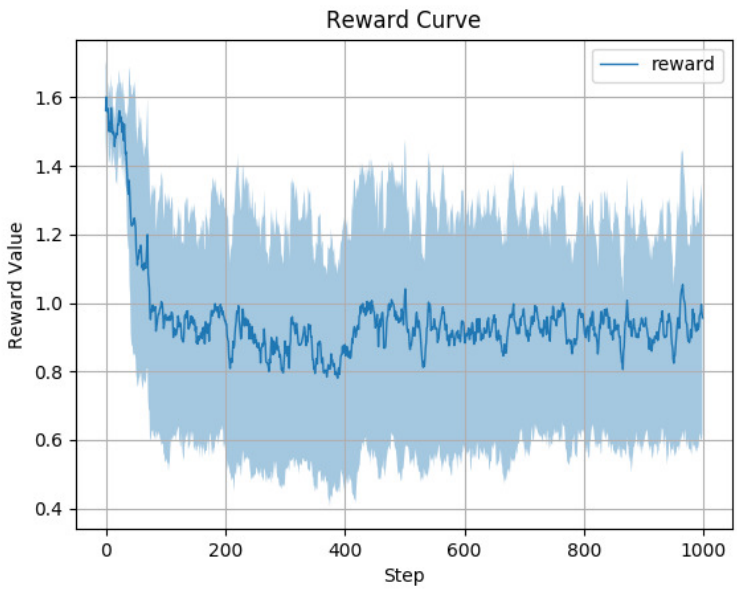


FIGURE 8.25 – Reward curve during direct RL training of meta-policy.

In terms of second question, we can confront accuracy and mean error from single-task expert policies and meta-policy, considering the same evaluation scenario of previous Sections. Table 8.2 shows such values (the accuracy metric is the same as defined in Subsection 8.2.2).

TABLE 8.2 – Meta-Policy Generalization Performance

Model Type	**Statistics**			
	Accuracy (%)		**Error** (m)	
	Mean	***Std***	***Mean***	***Std***
Single-Task Expert Policies	0.89	0.04	0.57	0.14
Meta-Policy (4, 256)	0.84	0.10	0.72[6]	0.25
Meta-Policy (11, 128)	0.84	0.17	0.68	0.28

As observed, the meta-policies are able to not only learn a kick representation for all meta-tasks, but also maintain a reasonable mean error and accuracy that are comparable to single-task expert policies. In fact, approximately 90% of meta-policy kicks in both networks achieved more than 0.7 of accuracy (among them, approximately 90% has error

[6]In the case of meta-policies, for calculating mean error we considered only kicks with accuracy higher than 70%. For Single-Task Expert Policies, all kicks have accuracy higher than such threshold, thus we use all of them.

less than one meter). For all accuracies and distance errors, we released the logs of kick evaluation.

Therefore, our experiments shows that a properly network can generalize to unseen tasks in the context of BUMPS algorithm – but we can improve those values using Policy Filtering.

To answer the last question, we present the final results of BUMPS algorithm, comparing with single-task expert policies. We presented four models:

- **Filtered Meta-Policy (4, 256)** – Final model where we sampled policy contexts from network with 4 hidden layers of 256 units after Policy Filtering;
- **Filtered Meta-Policy (11, 128)** – Final model where we sampled policy contexts from network with 4 hidden layerd of 128 units after Policy Filtering;
- **Filtered Ensemble** – Final model where where we sampled policy contexts from both previous meta-policy models; and
- **Filtered Meta-Policy High Sample Rate (4, 256)** – Final model where we sample policy contexts from network with 4 hidden layers of 256 units after Policy Filtering, but sampling 1091 tasks (approximately 10x more tasks than previous models).

Final results are presented in Table 8.3.

TABLE 8.3 – Final results for BUMPS algorithm

Model Type	**Error (m)**		**Rel. Error (%)**	
	Mean	**Std**	**Mean**	**Std**
Filtered Meta-Policy (4, 256)	0.45	0.10	3.8	1.1
Filtered Meta-Policy (11, 256)	0.43	0.13	3.6	1.3
Filtered Ensemble	**0.40**	**0.11**	**3.3**	**1.1**
Filtered Meta-Policy High Sample Rate (4, 256)	**0.40**	**0.09**	**3.3**	**0.9**

As we observe, there is a visible improvement from meta-policy, especially when considering ensemble and a high sample rate. In all models, BUMPS achieved mean error less than half, which is better than initial performance for meta-training tasks.

Finally, to wrap up this Chapter, we can highlight advantages of BUMPS algorithm:

- **Simplicity** – BUMPS relies on first-order derivatives to train a meta-policy in a standard way, by exploring the relationship between the geometric representation of task contextualization and optimal manifolds;

- **Sample Efficiency** – In contrast to other methods for meta-policy search where meta-training also performs RL, BUMPS only needs to interact with environment during RL training of single tasks expert policies and contexts' evaluation; and

- **Model Exploiter** – While BUMPS is model agnostic (i.e, it does not rely on a specific architecture to work), it can improve generalization accordingly to model architecture, which open doors for exploiting task structure.

In the next chapter, we change the focus of our experiments to explore walking control.

9 Walk Analysis

In Chapters 7 and 8, we conducted experiments regarding the kick motion, from behavior cloning of a simple kick with keyframe representation until a robust meta-learning system with fast adaptation for unseen kicks. To wrap up the experiments of this work, we conducted similar experiments for another challenging problem: walk motion.

Table 9 synthesizes all experiments conducted in this chapter.

Experiment	**Section**	**Description**
Walk Optimization via PP-NES – Sprint Task	Subsubsection 9.1.3.1	Evaluating walk optimization, via PP-NES, in the case of sprint task.
Walk Optimization via PP-NES – Conduct Task	Subsubsection 9.1.3.2	Evaluating walk optimization, via PP-NES, in the case of conduct task.
Walk Behavior Cloning	Subsection 9.2.1	Evaluation of Neural Walk Engine obtained via Behavior Cloning of the model-based walk.

9.1 Parameter Optimization for Engine-Based Walk

9.1.1 Walk System Description

As described in Section 3.4, the initial walking engine of our agent is a model-based controller that relies on ZMP concept and is modeled as a 3D-LIPM model. Actually, several teams from RoboCup Soccer 3D Simulation League use similar models (MACALPINE *et al.*, 2012a; REIS *et al.*, 2017).

In our engine, there are six parameters we need to tune, in order to achieve a faster and more robust walking:

- Period of a walk step (`Period`);

- Height of CoM (`zCoM`);
- Height that the swing foot is lifted off the ground (`zStep`);
- Separation between the feet (`ySeparation`);
- Double Support Ratio (`doubleSupportRatio`); and
- Arms compensation factor (`armsCompensationFactor`).

For more details about these parameters, we refer to Maximo (2015).

Agent's walk system is also composed by high level behaviors that use environment's information (such as the position of other agents and the ball) to perform navigation through the field, by computing the desired forward, sideway and rotational velocities. Figure 9.1 illustrates some behaviors and their relationship.

To improve the walking system as a whole, we need to optimize not just the engine parameters, but also the parameters for navigation behaviors, especially the transition between them. Each behavior has some parameters to define, and if we consider all of them, our search space becomes very large and hard do optimize via interaction. Therefore, we just consider the parameters from the basic navigation system, namely Navigate To Position Behavior, which combines some internal controllers as components and it is used for several other behaviors.

In the Navigate To Position Behavior, we have the following parameters:

- Max Forward Speed;
- Max Backward Speed;
- Max Side Speed;
- Max Angular Speed;
- Angle bandwidth; and
- Close Distance Threshold.

These parameters are related to the saturations of controllers and the "Close Distance Threshold" defines a transition between two internal navigation components depending on the target distance.

Therefore, the walking system has twelve chosen parameters to optimize. Additionally, we need to compute parameters that result in robust and fast walk for all behaviors, which needs to be considered during tasks' modeling.

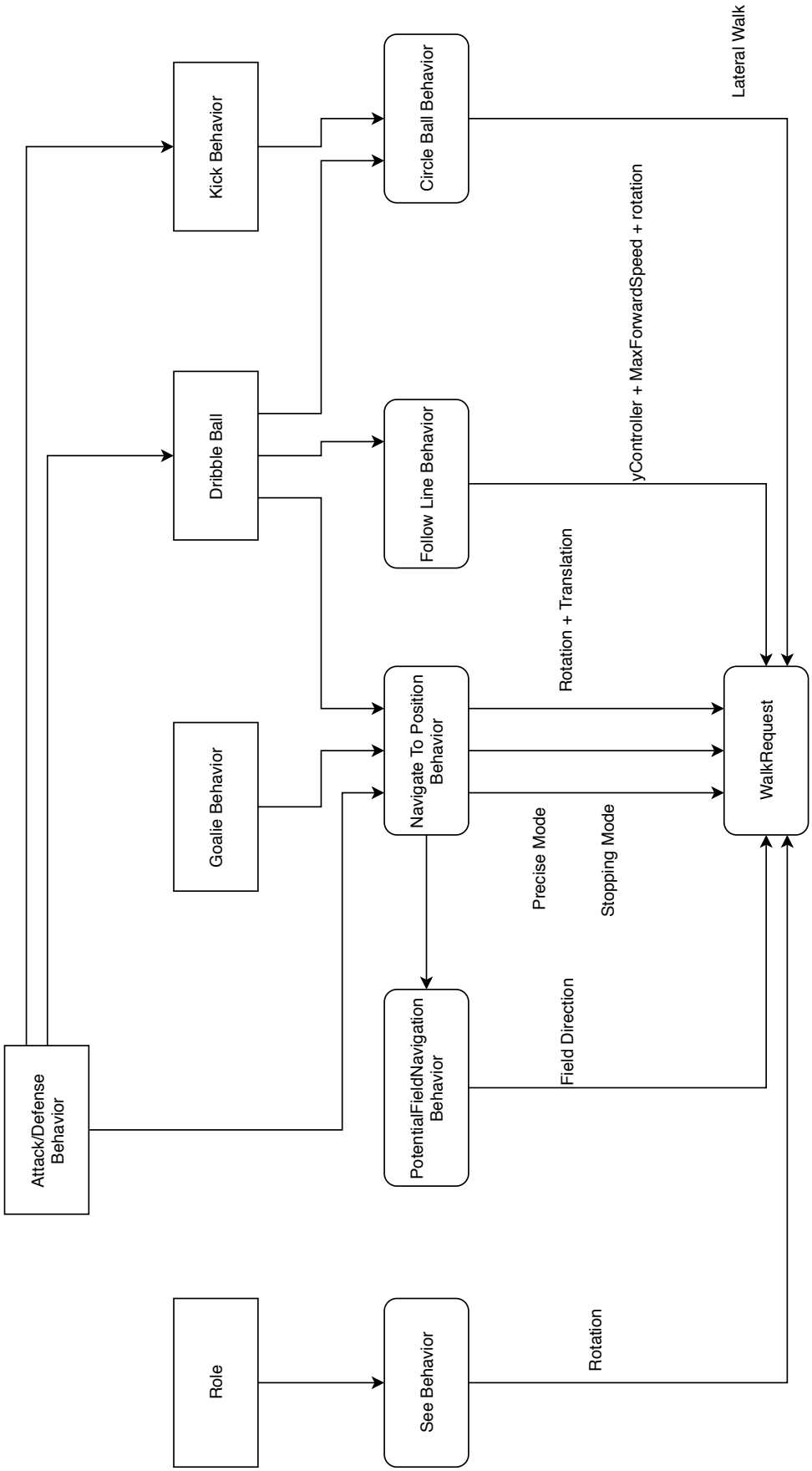


FIGURE 9.1 – Illustration of high level behaviors and their relationship.

Finally, our purpose in this work is not to address the problem of Model-Based Reinforcement Learning, but just to consider the optimization of such parameters as an RL problem and adapt the policy gradient strategy to this case, analyzing the performance of a model-free RL algorithm in this context.

9.1.2 Combining Evolution Strategies and Reinforcement Learning: Proximal Policy Natural Evolution Strategy (PP-NES)

An alternative approach to solving RL problems is using black-box optimization. Evolution Strategies (ES) is a particular set of optimization algorithms in this class. The most widely known member of ES is CMA-ES (SALIMANS *et al.*, 2017), which is valid for the RoboCup Soccer 3D League community as well. The CMA-ES was briefly described in Section 2.1.

Salimans *et al.* (2017) proposes the idea of ES as an alternative to Reinforcement Learning (even for neural network policies), due to strong reasons of its black-box nature:

- Invariance to action frequency and delayed rewards;
- Indifference to the distribution of rewards;
- Tolerance to extremely long horizons; and
- No need to temporal discounting or value function approximation.

Furthermore, in their work, they show:

- ES methods can be highly parallelizable;
- The data efficiency of ES can be surprisingly good;
- ES exhibited better exploration than policy gradient methods; and
- ES are robust in terms of hyperparameters.

Such key findings and properties encourage us to merge Policy Gradients and Evolution Strategies in a practical algorithm to parameter optimization, which we call "Proximal Policy Natural Evolution Strategy" (PP-NES). With PP-NES, we can take advantage of trust-region optimization in a black-box procedure. But not only that: in contrast to PPO – where we train a parameterized policy – PP-NES is designed to optimize the parameters of a model-based controller.

We started from PPO algorithm used in previous sections. Firstly, we define our learning representation as a very simple network whose input is an unit signal and the output is the values of each parameter, with no hidden layers. In the output layer, we apply the gaussian noise to perform exploration of parameter space. The output is a factor that we use to multiply the seed parameters. Figure 9.2 illustrates the network.

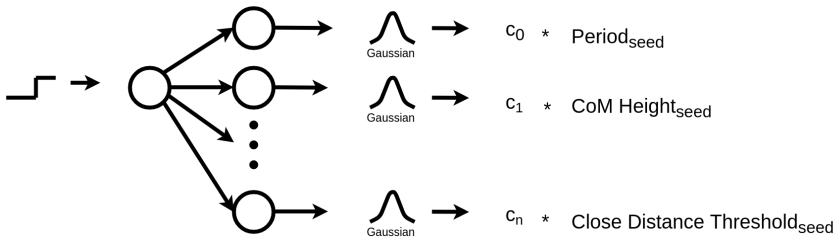


FIGURE 9.2 – PP-NES Network Architecture.

As weight initialization strategy, we start all weights as ones and all bias as zeros. In this way, we ensure that the mean of initial parameters are equals to seed, and we can control exploration by the variance of gaussian distributions. Basically, this network structure means a set of parameters that explore search space by following a gaussian distribution. Therefore, we can consider as a distribution over parameters $p_{\boldsymbol{\theta}}$ where we sample the population $k \sim p_{\boldsymbol{\theta}}$:

$$p_{\boldsymbol{\theta}}(\mathbb{1}) \sim \mathcal{N}[\boldsymbol{\theta}_{\mu}, \boldsymbol{\theta}_{\sigma}] \tag{9.1}$$

The terms $\boldsymbol{\theta}_{\mu}$ and $\boldsymbol{\theta}_{\sigma}$ refers to the output mean and standard deviation, where the mean is simply the sum of bias and weights. As notation we will consider M_k as the dynamics model parameterized by the sample k from the distribution $p_{\boldsymbol{\theta}}$ parameterized by $\boldsymbol{\theta}$.

In terms of optimization, we conduct the whole PPO algorithm with some changes. Firstly, we consider the episode as a single cycle state-action-reward. The state comes from the interaction between the model with specific parameters and the environment. The "action" is related to the set of decisions the model takes using such parameters. Finally, the reward is a fitness function that evaluates the model during the episode.

If we collapse the cumulative reward to a single episode context, Equation (5.8) will simplify to:

$$G = F(k), \tag{9.2}$$

where F is the fitness function evaluation using a parameter set, Using this simplification into GAE estimation, we have the following Advantage function:

$$
\begin{aligned}
A_{M_k}(s,a) &= Q_{M_k}(s,a) - V_{M_k}(s) && (9.3)\\
&= \mathbb{E}_{k\sim p_{\boldsymbol{\theta}}}[R_s + \gamma V_{M_k}(s_{t+1})] - V_{M_k}(s) && (9.4)\\
&= \mathbb{E}_{k\sim p_{\boldsymbol{\theta}}}[R_s] - V_{\boldsymbol{\theta}}(s) && (9.5)\\
&= F(k) - V_{\boldsymbol{\theta}}(s), && (9.6)
\end{aligned}
$$

which means that our advantage function is just the difference between the estimated fitness function and its prediction. This simplification occurs because there is no subsequent state, therefore its value function prediction is zero. Furthermore, the concept of value function can be viewed as the expected fitness function's mean in that update, and the critic network will point the gradient to the direction of the parameters set that has largest positive difference from such mean:

$$
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{k\sim p_{\boldsymbol{\theta}}}\Big[\nabla_{\boldsymbol{\theta}} \log M_k(a|s)\Big(F(k) - V_{\boldsymbol{\theta}}(s)\Big)\Big]. \tag{9.7}
$$

Equation (9.7) is very similar to the version of ES used in Salimans *et al.* (2017). Indeed, we can consider PP-NES as an "actor-critic" natural evolution strategy. Natural Evolution Strategies (NES) (WIERSTRA *et al.*, 2014) are a class of ES algorithms that maximize the average fitness value over the a population sampled from a parameterized distribution performing stochastic gradient descent.

Finally, it is worth to mention that the idea of critic network in PP-NES is quite rudimentary, because it does not "critic" a sample, but only the distribution. Basically it is a prediction of the running mean fitness value – a straightforward baseline to reduce policy's variance.

9.1.3 Walk Optimization via PP-NES

In order to analyze PP-NES algorithm and the optimization of the model-based controller, we developed some experiments in the walking system. Firstly, we define as seed parameters the actual values used in the agent (Table 9.1). This value has extreme importance because defines the initial region of search, which highly affects the optimum values to where the training converges.

As stated previously, the walking system has multiple behaviors, as a multi-task problem – we need to find a parameter set that performs well in all tasks. We defined three tasks to conduct optimization: Sprint Task and Conduct Task.

TABLE 9.1 – Seed Parameters - Walk System

Parameter	Seed Value
`Period`	0.1
`zCoM`	0.32
`zStep`	0.05
`ySeparation`	0.05
`doubleSupportRatio`	0.2
`armsCompensationFactor`	0.8
`maxForwardSpeed`	0.9
`maxBackwardSpeed`	0.9
`maxSideSpeed`	0.4
`maxAngularSpeed`	5.23
`angularBandwidth`	3.14
`closeDistanceThreshold`	1.0

All the training procedures, videos, evaluation results and binaries are released[1]. Furthermore, we also released the modifications to transform PPO in PP-NES[2].

9.1.3.1 Sprint Task

The first task is basically to run in forward direction during a fixed time or until it falls down. The idea here is to evaluate PP-NES performance while to analyze robot's behavior with new parameters. The fitness function is just the forward distance.

As final parameters, we considered the set that performed best in this task. We also conducted a hyperparameter search and defined some sets that leads to better performance. We ran several seeds of each depending on the experiment (the exactly numbers can be found on the logs released).

In the first attempt, we tried to optimize all parameters, but the agent proceeded to decrease aggressively the ySeparation parameter, which resulted in a faster but very unstable engine. We then fixed some of parameters (namely, ySeparation, and all navigation parameters except max forward speed), because they could lead either to a bad local optimum or are not interesting to the task. We also considered a longer episode with 800 server cycles, instead of 300 from previous experimentation. We ran five repetitions in each evaluation, to avoid an outlier behavior.

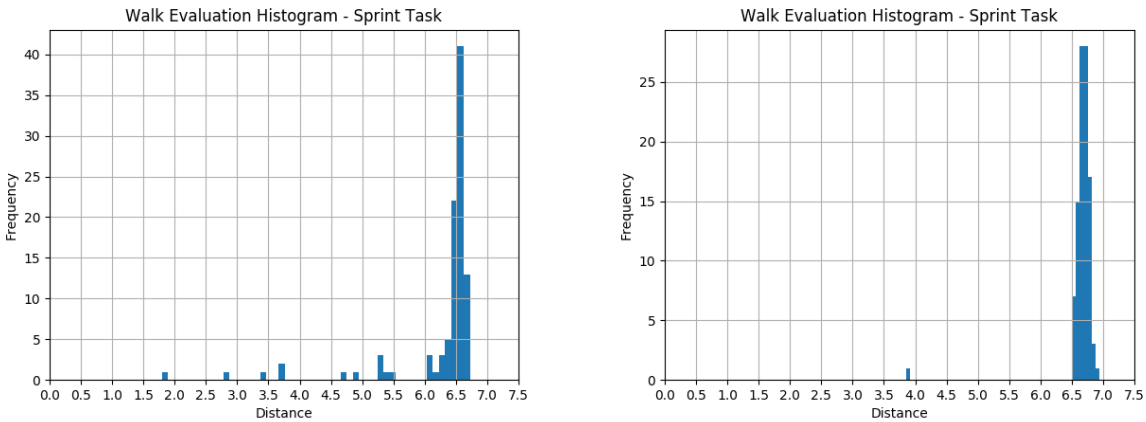
The resultant parameters of sprint task are in Table 9.2. We evaluated them in a test scenario similar to the task itself, and compared with baseline parameters in Figures 9.3a and 9.3b, with the histograms of a hundred repetitions of this test. We also published

[1] `https://bit.ly/2WhagKe`
[2] `https://github.com/alexandremuzio/baselines/tree/walk_policy`

videos about the evaluation of each parameter set.

TABLE 9.2 – Final Parameters - Sprint Task

Parameter	Value
`Period`	0.0832775
`zCoM`	0.336567
`zStep`	0.10164
`ySeparation`	0.05
`doubleSupportRatio`	0.00922754
`armsCompensationFactor`	1.04674
`maxForwardSpeed`	1.29035
`maxBackwardSpeed`	0.9
`maxSideSpeed`	0.4
`maxAngularSpeed`	5.23
`angularBandwidth`	3.14
`closeDistanceThreshold`	1.0



(a) (b)

FIGURE 9.3 – Forward walk evaluation for baseline and sprint parameters, respectively.

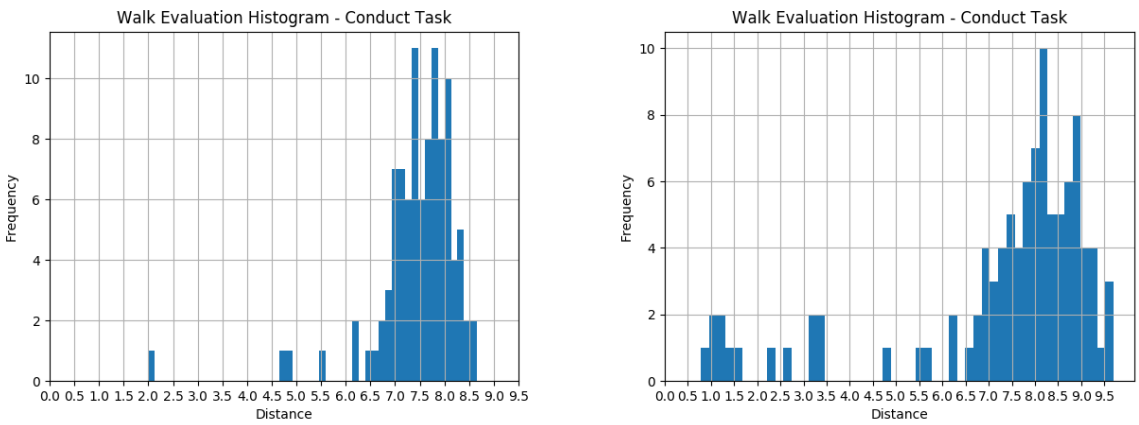
We observed from histograms and evaluation videos an improvement on forward walk. However, in game situation, the parameters did not perform well, for two main reasons: firstly, the optimization leads to a faster walk but it compromised stability. In fact, the task has no external disturbances, hence the parameters could explore in the direction of "sacrifice" stability to gain velocity. This idea leads to the second reason: the task itself does not represent the game situation, where there are several perturbations and the commanded speed varies substantially, either in direction and absolute value.

9.1.3.2 Conduct Task

In order to address the problem of Sprint Task not representing game situation, we developed the Conduct Task, where the agent have to go to the ball and conduct it towards the goal. Both elements start in fixed positions. The reward is the distance that ball traveled in the goal direction. Episode's length remains 800 cycles.

The first difference from the previous task is that in Conduct Task there is freedom to optimize ySeparation parameter. The ySeparation does not converge to zero because the task itself requires more stability to perform well. Furthermore, all the velocity saturation parameters need to be tuned, because in this task the agent can follow varied directions (its navigation can change depending on the way it touches the ball).

In this case, starting from Sprint Task parameters leaded to better results in Conduct Task. They provided a better initial forward performance while satisfying initial stability requirements. The resultant parameters are shown in Table 9.3. Similar to previous task, we developed a test scenario analogous to the task. Figures 9.4a and 9.4b show the histograms from a hundred repetitions of such test. We also released videos in the logs repository.



(a) (b)

FIGURE 9.4 – Conduct walk evaluation for baseline (Figure 9.4a) and sprint parameters (Figure 9.4b).

From histograms, we observe a slightly increase of "main component" mean, although there are more failed samples. When we analyze these examples, we observed a particular situation where the agent needs to circle the ball, which is rare during conduct task – and probably did not occur during training time for such parameters.

The problem from conduct task can be generalized to the fact that a walking system is a multi-task problem, where tasks are defined by how high level behaviors command speed. Both tasks analyzed here describes specific tasks, and in both cases the resultant

TABLE 9.3 – Final Parameters - Conduct Task

Parameter	Value
`Period`	0.113302
`zCoM`	0.338307
`zStep`	0.0870786
`ySeparation`	0.0490317
`doubleSupportRatio`	0.0023986
`armsCompensationFactor`	1.29023
`maxForwardSpeed`	0.889164
`maxBackwardSpeed`	0.716367
`maxSideSpeed`	0.475553
`maxAngularSpeed`	5.08302
`angularBandwidth`	2.44133
`closeDistanceThreshold`	0.904493

parameters are better to perform them than baseline, but does not generalize to unseen tasks.

This is a case of overfitting: the model tends to over specialize in the task. This problem has also been observed using CMA-ES in the same context (MACALPINE *et al.*, 2012a). An alternative solution is to train several models, each for a specific walk situation, and design a state machine that implements the transition rules throughout them. During optimization, we implement specific tasks (or a task with varied challenges), and fix the parameters that we are not interested in optimize that moment. The tasks have to cover all possible states and transitions as well, to avoid incompatible parameter sets. This is an example of layered learning (STONE, 2000).

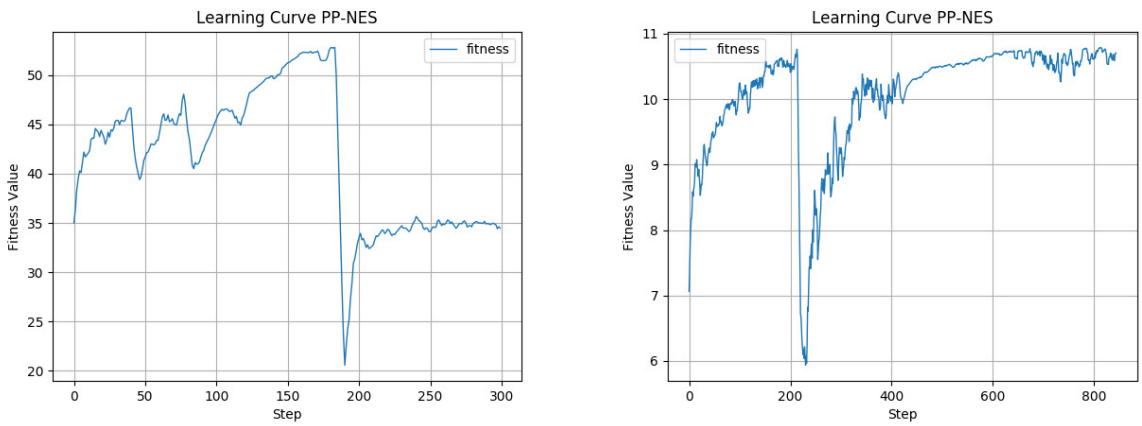
This approach worked well in previous works using the context of robot soccer. However, design the state machine previously described can be very challenging, especially when we have a complex and hierarchical behavior system as in Figure 9.1, where the transitions between behaviors are complex.

We propose an alternative analysis for walk optimization in next section, where we clone the model-based controller to a parameterized policy.

9.1.3.3 PP-NES Evaluation

To wrap up this Subsection, we conducted an analysis of PP-NES performance. As our objective here is to optimize the model-based controller, we limit our work to just analyze its performance in the problem, rather than comparing with other ES or Policy Gradient algorithms, which we leave as future work.

In terms of optimization, we found the common logarithmic learning curves, but with some catastrophic steps that slowed learning. Figures 9.5a and 9.5b illustrate this effect. These bad steps occurred less frequently with better hyperparameters, which lead us to relate with bad gradient estimation. Additionally, the learning rate was kept approximately constant during such experiments, and therefore the exploration rate have not been shrunk properly to later stages of training, similarly to described in Subsection 8.1.8. With higher exploration and a reasonable small number of samples, the gradient estimation got worse, resulting in such bad steps.



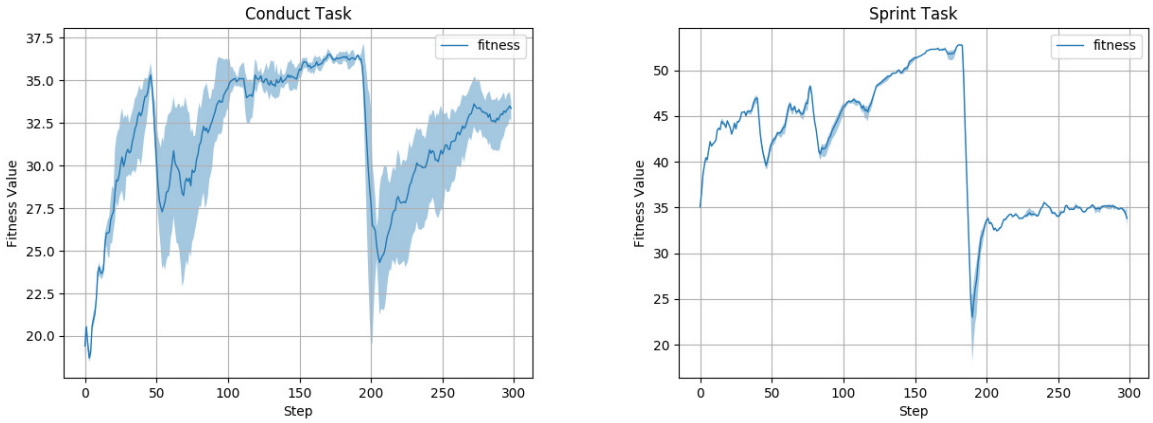
(a) (b)

FIGURE 9.5 – Fitness curve in experiments conducted for Sprint Task optimization, where we see catastrophic steps. In the case of Figure 9.5a, the agent was not even able to recover initial fitness value.

We found sensibility to hyperparameters similar to PPO algorithm, which we resolve by conducting hyperparameter random search. However, the resultant sets performed consistently in both Sprint and Conduct Tasks (See Figures 9.6a and 9.6b). This robustness is an advantage of this algorithm: as long as the optimization task provides an accurate estimate of fitness function, optimization hyperparameters will similarly perform well in several tasks. This happens for two main reasons: firstly, as we are optimizing parameters to the same model, we can expect similar search spaces; secondly, as there is only a single cycle state-action-reward, the algorithm is indifferent to the distribution of rewards.

Additionally, the algorithm is highly parallelizable by design – it is possible to evaluate all samples of a learning cycle at the same time, and perform gradient synchronously. With enough resources, we can explore higher learning rates, which reduce optimization time.

Finally, we can conclude that PP-NES was satisfactory as an optimization algorithm for our problem of optimizing a model-based walk engine, despite the problems described previously. In the experiments presented, we were able to find parameter sets that per-



(a) (b)

FIGURE 9.6 – Fitness curves of experiments conduct using the same hyperparameters to Conduct and Sprint Tasks, respectively. Despite the difference in confidence intervals, the optimize performed well in both cases.

formed better than initial values in the specified tasks.

9.2 Neural Walk Engine

In the previous Section, we optimized a model-based controller using PP-NES, a proposed algorithm that joins trust region policy gradient method with black-box optimization. Although we considered the optimization method satisfactory, the problem of improving a walk system by optimizing few parameters of such engine is challenging and require complex frameworks to produce better results, such as layered learning, as described in Subsubsection 9.1.3.2.

We then hypothesized that an easier approach to conduct this optimization is to imitate the walk engine by using a neural network and then proceed an RL training, following similar methodology as used in Chapter 7. As the walk engine is a hierarchical multi-task learning, we would adopt a curriculum-based RL training. In that way, we would have a single parameterized policy, instead of several models that overfit the optimization task.

In this final experimental section, we wrap up this work by applying Behavior Cloning in the walk system, resulting in a completely neural network based walk engine, which we called neural walk engine. We present the methodology and comparisons with the model-based controller.

9.2.1 Walk Behavior Cloning

We conducted walk behavior cloning in a similar way to the one described in Chapter 7. Although simple, this supervised imitation technique was enough to ensure good performance and did not suffer from distributional shift.

Firstly, we collected a dataset in a game situation (which reduces distributional shift effects), gathering information about requested velocity, current joints and then desired joints for next step. We ran such collector during half game, totalizing 14424 samples. In this way, we train the model based only in the data that high level behaviors provides, rather than all possible velocity space. Additionally, we chose only legs' joints (14 in total), once the others has constant values during walk.

In terms of neural network architecture, we designed an LSTM-based model, where the input is ten previous timesteps tuples composed by commanded speed and current joints in each timestep. As output, the 14 legs joints. Table 9.4 shows model summary. Figure 9.7 sketches the network.

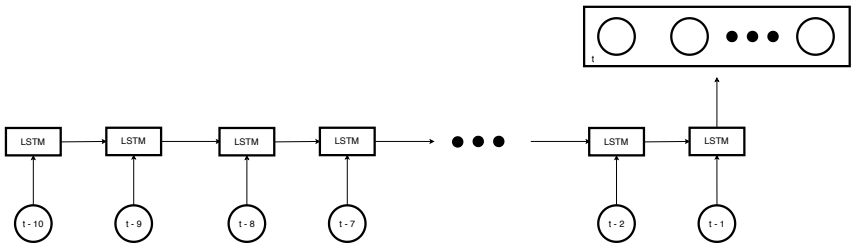


FIGURE 9.7 – LSTM network for neural walk engine. The ten previous timesteps are used to compute the next joints.

TABLE 9.4 – Neural Walk Engine Network Summary.

Layer	**Neurons/Units**	**Parameters**
LSTM	30	5750
Dense	14	434

Total Parameters	6194

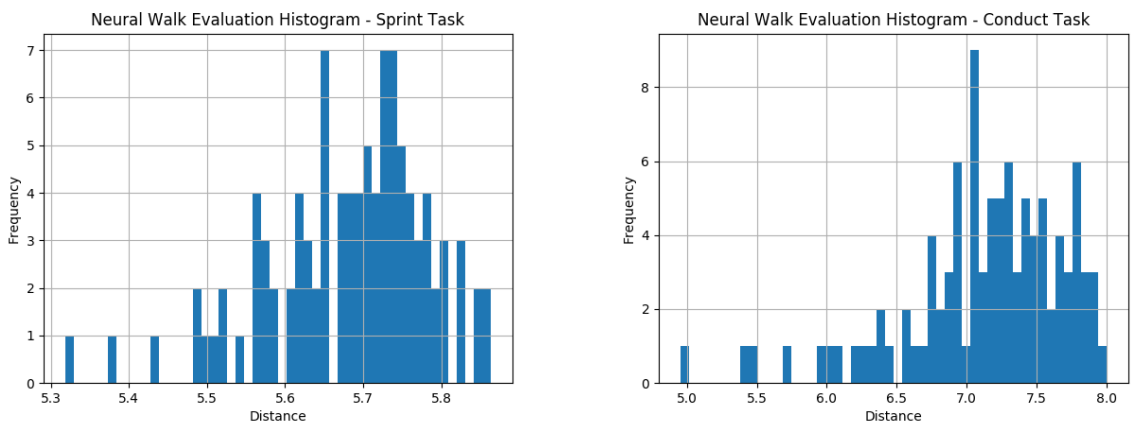
In this architecture, the LSTM layer is responsible to temporal modeling of the walk system, while the dense layer use resultant features to conduct regression task. We expect two advantages of this temporal modeling: first, it uses previous timesteps to compute next desired joints, which means that we have a feedback system that helps in stability and quick adaptation in case of external perturbations. Secondly, due to the LSTM layer, the network can infer hidden states more easily, such as joint velocities.

Additionally, the usage of recurrence decrease the number of necessary parameters and layers to learn good walking skills. Indeed, a fully dense network will need to perform temporal modeling across layers, which leads to a much deeper network.

In terms of training, we minimize mean absolute error during 1000 epochs and use an initial learning rate of 3×10^{-4} with step decay every 50 epochs, using 0.8 as decay factor. The batch size was 128. The final model has approximately 0.013 of mean absolute error, which is enough to conduct all navigation behaviors.

Figures 9.8a and 9.8b shows the evaluation of neural walk engine in sprint and conduct evaluations, respectively. These test scenarios are the same as in the last section and the objective is just to show that the Neural Walk Engine has similar performance. Comparing Figures 9.3a and 9.3b, we observe that the learned engine is slightly slower, but did not fall down during evaluation. The same is valid for Figures 9.4a and 9.4b. Out interpretation of this result is that it reflects of the feedback loop that corrects the regime error and thus improves stability of a constant motion.

Additionally to previous comparisons, we released videos about the agent conducting ball towards the goal using both engines[3], where we can see that both engines work almost indistinguishable.



(a) (b)

FIGURE 9.8 – Evaluation of neural walk engine in sprint and conduct tasks, respectively.

As presented by the results, modeling the engine as an LSTM policy can be helpful in terms of stability and inferring hidden states. Additionally, we are able to optimize towards a objective using policy gradient methods, which can boost its performance. The major drawback, however, is that we do not have specific parameters to modify high level features, such as period, CoM, or velocity saturations. Actually, we learn a specific set of features depending on the data. This makes a bit harder to learn new high level features like these in order to improve performance.

We conducted some RL experiments in a forward walk task and using distance as reward signal, but the optimization basically maintained the same performance. In cases

[3] `https://bit.ly/2WhagKe`

with high exploration rate, there is improvement, but the initial performance is much worse and, at the end, there is no improvement from the initial neural engine. From such experiments, we observed that in RL optimization of walk engine the agent will prefer to maintain stability than assume higher risk of exploration and drops reward significantly by falling down.

We limit our work here, leaving this deep research as future work. In this section, we were able to model a walk engine that works without a dynamics model regarding of agent-environment interaction. We only used the data from an expert teacher, but without details of underlying implementation.

10 Conclusions, Recommendations, and Future Works

In this final chapter, we wrap up this work sharing conclusions and some research lines that emerge from it.

10.1 Conclusions

In this work, we presented several techniques that combines Imitation Learning and Reinforcement Learning to improve humanoid robot motions.

In chapter 1, we have shown a brief introduction, presenting the motivation and objective of this work. Chapter 2 reviewed the literature that closely relates to our work, in the domains of Optimization Techniques, Reinforcement Leraning for control, Imitation Learning, and Meta-Learning.

Chapter 3 described the RoboCup Soccer 3D Simulation League domain. It also presented the idea of keyframed motions applied to kick and the ZMP-based walk engine used in our soccer agent.

Chapter 4 described mathematically the Deep Learning techniques and the underlying optimization problem, presenting the widely used methods of this domain. Chapter 5, on the other hand, described a Reinforcement Learning system, its representation as an MDP and the algorithms used in this domain, culminating in those ones used in this work. Chapter 6 presented the new ideas behind Meta-Learning, and some of the algorithms that inspired our experiments, either in architecture and algorithmic levels.

Chapter 7 presents the kick motion problem and all the methodology and experiments to evaluate the Imitation Hypothesis. Accordingly to it, by representing the kick motion by a neural network with thousands of parameters and following a 2-step training procedure where:

- The first step is a Supervised Learning train, where we transfer the knowledge from

a keyframe representation to a neural network; and

- The second step is a Reinforcement Learning train, where we optimize the motion from the neural network using rewards related to a specific performance task,

thus, we are able to develop better policies for kick motion that the initial ones.

For mimicking motions, we presented pure RL techniques – starting from randomly policies until well defined kick movements – that are agnostic to the task itself and hence could be applied in other motions, such as walk. However, in our experimental setup, these techniques has some disadvantages:

- If we do not expose a reference motion, agent will learn a sub-optimal policy which is very different from human behavior, as shown in subsection 7.8.1;
- If we just pass a reference motion, agent prefers to stand still because exploration leads to higher penalizations, as shown in subsection 7.8.2; and
- Even if we have a reference motion and several techniques for reducing sampling bad data, the agent was not able to conduct the mimicked motion properly – the residual error collapsed the motion, as show in subsections 7.8.4 and 7.8.5.

Therefore, **from the perspective of the experimental setup and all constraints induced by it – which includes the choice of state and action spaces, computational resources restrictions, and other design decisions – pure RL techniques by themselves leaded to suboptimal policies and did not achieve high performance motions**.

We also presented a Hybrid Learning Model, which initially trains a neural network policy using supervised learning from the reference motion and optimizes it towards a task using Deep RL techniques. In this model:

- The Supervised Learning model is able to mimic the reference motion with a residual error. However, by itself, it is not able to generate better policies, as shown in subsection 7.9.1; and
- When we inject this pre-trained model in the RL algorithm and then optimizes it towards a performance task, **it proved to work well and achieved the objective of improve policy performance, which was, in this case, to gain accuracy**, as presented in subsections 7.9.2, 7.9.3 and 7.9.4.

The supervised learning step provided high initial reward and a good point to start exploration and the RL algorithm could absorb such benefits and find a better policy than the original motion.

Therefore, we conclude that, in some problems and experimental setups, pure RL techniques can have greater difficulty due to the fact that, starting from an initially randomized parametrized policy, we have a much more complicated optimization problem, where it should be considered several aspects for modeling the reward to at least mimic the reference motion. On the other hand, when using a supervised pre-train, it is possible to obtain an initial policy that replicates well the reference motion and is able to be optimized using RL and, finally, outperforms this reference when we consider a performance task.

Chapter 8 extends the learning framework from the previous chapter in order to consider precise kicks. Firstly, in Section 8.1, we conducted several experiments to understand the sensibility of some hyperparameters and some optimization procedures, in order to improve the performance from a baseline. We observed that **PPO algorithm was very sensitive to hyperparameter changes**, which is undesirable to an RL algorithm. We then addressed such problem by **conducting a random search on some hyperparameters, culminating in a much better reward curve**. We have also tested an LSTM model and a proposed technique called exploration restarter, which helped to improve more our reward in subsequent training sessions.

We then consider our learning framework feasible to learn precise kicks. More than that, we conducted experiments in order to address the Meta-Learning Hypothesis, which states that:

- By learning few similar tasks from a distribution over them, and
- Using a meta-training procedure from the experience of the emergent expert policies,

we are able to generalize the knowledge to unseen tasks and also facilitate the adaptation to those that were not learned properly.

In order to confirm our hypothesis, we introduced Bottom-Up Meta-Policy Search (BUMPS), an algorithm that firstly train single-tasks expert policies using RL training, then collect data from them to train a meta-policy using supervised learning and finally performs a fast adaptation using meta-gradient through contexts. We conducted a theoretical explanation of BUMPS in Section 8.2.1, comparing with Reptile algorithm.

In terms of training single-task expert policies, we modeled a task with fixed episode length and using a reward that reinforces precision to target at each cycle, **culminating in precise kicks with error less than 1 meter**, as described in Section 8.2.2. We then performed meta-policy training, exploring some network architectures based on the metric of Equation (8.4). Finally, we conducted Policy Filtering and also explored some possible of architectures, task sample rate and ensemble.

As results, we presented that each part of BUMPS algorithm improves final results,

that **achieved precision error less than half meter, while maintained kicks accuracy in approximately 85%**. Furthermore, we validate our explanations and hypotheses about meta-policy training and meta-gradient through contexts.

We wrapped up our experiments by analyzing the challenge of walk optimization. We considered two viewpoints: a model-based and a model-free. In the model-based, we used the actual walk engine and optimized its parameters with an algorithm that we proposed in Section 9.1.2, called **Proximal Policy Natural Evolution Strategy**. This algorithm merges advantages of PPO and Evolution Strategies, by considering the policy as a gaussian distribution over the parameters to be optimized by proximal policy method. Rather than considering an episode as a sequence of timesteps (and then maximize cumulative reward), we consider an episode as a single timestep (and then maximize a fitness function that describe agent's behavior).

In our optimization, despite of some catastrophic steps common to on-policy algorithms during training, **we were able to find better parameter sets to forward walk and conduct ball scenarios**. However, as happened in previous works, **the optimization tends to overfit the task**, which is bad for game situation, where **the walk engine is actually a multi-task policy, composed by the tasks generated by each high-level navigation controller**. The solution to this problem is to consider several models with distinct parameterizations. This does not solves the overfitting problem and also requires a complex hand-designed state machine to manage transitions between each model.

We then consider the model-free approach in Section 9.2. Firstly, this viewpoint has the purpose to exploit the imitation learning approach to a harder problem, and check if we can transfer the knowledge of a complex mathematical model to a neural network. But after the results from model-based optimization, we also hypothesized that such viewpoint can solve the problem of task overfitting, by conducting a curriculum-based learning. For the sake of conciseness, we limited our work to apply Behavioral Cloning, and **we learned the whole walk engine in a single neural network**. Due to the fact we used an LSTM model,**we got a some improvements in the steady state**, but we were not able to optimize forward speed using the RL methodology of this work.

Finally, this work opened several research lines to conduct and improve machine learning techniques for control. We cast all of them in next Section.

10.2 Future Work

In terms of kick motion, we aim to further analyze RL training to improve kicks' precision. One of the best hypothesis we have is to improve reward shaping, by considering

a reward that nearly predicts final distance error, or at least a training procedure able to optimize using a single reward at the end of episode.

We also want to solve a great source of distributional shift by fixing joint's initial state, performing smoothing between agent's joints before kick and the fixed initial state. It will facilitate the comparison between kick motions and deploy to game situation.

In terms of BUMPS algorithm, we plan to apply it in heavily-benchmarked environments in order to compare with other meta-policy search methods, in terms of adaptation and sample efficiency. Furthermore, we aim to apply in some real-world problems and online setting, to extend BUMPS validation to such cases. Additionally, try tasks with goals in a higher dimensional geometric space (2-D or 3-D) and harder problems where geometric representation is not as clear.

Finally, we objective to conduct BUMPS with meta-learner architectures, especially SNAIL, by collecting expert policies training data, including past rewards and termination flags, and then evaluating generalization across unseen tasks.

In terms of parameter optimization of model-based walking engine, the objective is to confront PP-NES with CMA-ES inside RoboCup Soccer 3D environment, in a variety of tasks that includes kick, walk, and getup motions.

Finally, we aim to further develop Neural Walk Engine to understand how to optimize its performance for a faster and more stable walk and then evaluate if a curriculum-based learning for it can be better than using parameter optimization for model-based controller using layered learning. .

Bibliography

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANé, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIéGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

ABBEEL, P.; COATES, A.; NG, A. Y. Autonomous helicopter aerobatics through apprenticeship learning. **Int. J. Rob. Res.**, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 29, n. 13, p. 1608–1639, November 2010. ISSN 0278-3649. Disponível em: <http://dx.doi.org/10.1177/0278364910371999>.

ABBEEL, P.; SCHULMAN, J.; CHEN, X. P. Lecture, **Policy Gradients**. 2017. Disponível em: <https://drive.google.com/file/d/0BxXI_RttTZAhY216RTMtanBpUnc/view>.

ANDRYCHOWICZ, M.; WOLSKI, F.; RAY, A.; SCHNEIDER, J.; FONG, R.; WELINDER, P.; MCGREW, B.; TOBIN, J.; ABBEEL, P.; ZAREMBA, W. Hindsight experience replay. **CoRR**, abs/1707.01495, 2017. Disponível em: <http://arxiv.org/abs/1707.01495>.

ARGALL, B. D.; CHERNOVA, S.; VELOSO, M.; BROWNING, B. A survey of robot learning from demonstration. **Robot. Auton. Syst.**, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, v. 57, n. 5, p. 469–483, May 2009. ISSN 0921-8890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2008.10.024>.

BAIN, M.; SAMMUT, C. A framework for behavioural cloning. In: **Machine Intelligence 15**. [S.l.: s.n.], 1995.

BAKKER, B. Reinforcement learning with long short-term memory. In: **Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic**. Cambridge, MA, USA: MIT Press, 2001. (NIPS'01), p. 1475–1482. Disponível em: <http://dl.acm.org/citation.cfm?id=2980539.2980731>.

BARTELS, R. H.; BEATTY, J. C.; BARSKY, B. A. **An Introduction to Splines for Use in Computer Graphics &Amp; Geometric Modeling**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. ISBN 0-934613-27-3.

BEECHER DON VAN DER DRIFT, D. M. L. V. S. G. B. L. S.; LANGNER, M. **Bootstrapped - confidence intervals made easy**. [S.l.]: GitHub, 2018. `https://github.com/facebookincubator/bootstrapped`.

Bengio, Y.; Bengio, S.; Cloutier, J. Learning a synaptic learning rule. In: **IJCNN-91-Seattle International Joint Conference on Neural Networks**. [S.l.: s.n.], 1991. ii, p. 969 vol.2–.

Bengio, Y.; Frasconi, P.; Simard, P. The problem of learning long-term dependencies in recurrent networks. In: **IEEE International Conference on Neural Networks**. [S.l.: s.n.], 1993. p. 1183–1188 vol.3.

BERGSTRA, J.; BENGIO, Y. Random search for hyper-parameter optimization. **J. Mach. Learn. Res.**, JMLR.org, v. 13, p. 281–305, February 2012. ISSN 1532-4435. Disponível em: <http://dl.acm.org/citation.cfm?id=2188385.2188395>.

BILLARD, A.; CALINON, S.; DILLMANN, R.; SCHAAL, S. Robot programming by demonstration. In: _____. **Springer Handbook of Robotics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 1371–1394. ISBN 978-3-540-30301-5. Disponível em: <https://doi.org/10.1007/978-3-540-30301-5_60>.

BILLARD, A.; MATARIC, M. Learning human arm movements by imitation: Evaluation of a biologically-inspired connectionist architecture. **Robotics and Autonomous Systems**, v. 41, 06 2000.

CHEN, Z.; LIU, B. **Lifelong Machine Learning in the Big Data Era**. 2015. `https://www.cs.uic.edu/~liub/IJCAI15-tutorial.html`. [Online; accessed at Aug 02, 2019].

CHOLLET, F. *et al.* **Keras**. 2015. Https://keras.io.

COLAS, C.; SIGAUD, O.; OUDEYER, P.-Y. How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments. **arXiv e-prints**, p. arXiv:1806.08295, Jun 2018.

COLLINS, S.; RUINA, A.; TEDRAKE, R.; WISSE, M. Efficient Bipedal Robots Based on Passive Dynamic Walkers. **Science Magazine**, v. 307, p. 1082–1085, February 2005.

DEPINET, M.; MACALPINE, P.; STONE, P. Keyframe sampling, optimization, and behavior integration: Towards long-distance kicking in the robocup 3d simulation league. In: AKIN, H. L.; BIANCHI, R. A. C.; RAMAMOORTHY, S.; SUGIURA, K. (Ed.). **RoboCup-2014: Robot Soccer World Cup XVIII**. Springer Verlag, 2015. Disponível em: <http://www.cs.utexas.edu/users/ai-lab/?depinet:robocup14>.

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

DUAN, Y.; ANDRYCHOWICZ, M.; STADIE, B. C.; HO, J.; SCHNEIDER, J.; SUTSKEVER, I.; ABBEEL, P.; ZAREMBA, W. One-shot imitation learning. **CoRR**, abs/1703.07326, 2017. Disponível em: <http://arxiv.org/abs/1703.07326>.

DUAN, Y.; SCHULMAN, J.; CHEN, X.; BARTLETT, P. L.; SUTSKEVER, I.; ABBEEL, P. Rl^2: Fast reinforcement learning via slow reinforcement learning. **CoRR**, abs/1611.02779, 2016. Disponível em: <http://arxiv.org/abs/1611.02779>.

EFRON, B.; TIBSHIRANI, R. Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy. **Statist. Sci.**, v. 1, n. 1, p. 54–75, 1986. Doi:10.1214/ss/1177013815. Disponível em: <http://projecteuclid.org/euclid.ss/1177013815>.

FINN, C.; ABBEEL, P.; LEVINE, S. Model-agnostic meta-learning for fast adaptation of deep networks. **CoRR**, abs/1703.03400, 2017. Disponível em: <http://arxiv.org/abs/1703.03400>.

GHOSH, D.; SINGH, A.; RAJESWARAN, A.; KUMAR, V.; LEVINE, S. Divide-and-conquer reinforcement learning. **CoRR**, abs/1711.09874, 2017. Disponível em: <http://arxiv.org/abs/1711.09874>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. `http://www.deeplearningbook.org`: MIT Press, 2016.

GOODFELLOW, I. J.; VINYALS, O. Qualitatively characterizing neural network optimization problems. **CoRR**, abs/1412.6544, 2014. Disponível em: <http://arxiv.org/abs/1412.6544>.

GOUAILLIER, D.; HUGEL, V.; BLAZEVIC, P.; KILNER, C.; MONCEAUX, J.; LAFOURCADE, P.; MARNIER, B.; SERRE, J.; MAISONNIER, B. Mechatronic design of nao humanoid. In: **2009 IEEE International Conference on Robotics and Automation**. Kobe, Japan: IEEE, 2009. p. 769–774. ISSN 1050-4729.

GRPC. 2018. `https://grpc.io/`. Accessed: 2018-08-20.

GUPTA, A.; MENDONCA, R.; LIU, Y.; ABBEEL, P.; LEVINE, S. Meta-reinforcement learning of structured exploration strategies. **CoRR**, abs/1802.07245, 2018. Disponível em: <http://arxiv.org/abs/1802.07245>.

HAAN, P. de; JAYARAMAN, D.; LEVINE, S. Causal confusion in imitation learning. **Conference on Neural Information Processing Systems**, 12 2018.

HÄMÄLÄINEN, P.; BABADI, A.; MA, X.; LEHTINEN, J. PPO-CMA: proximal policy optimization with covariance matrix adaptation. **CoRR**, abs/1810.02541, 2018. Disponível em: <http://arxiv.org/abs/1810.02541>.

HANSEN, N.; OSTERMEIER, A. Completely derandomized self-adaptation in evolution strategies. **Evolutionary Computation**, v. 9, n. 2, p. 159–195, 2001. Disponível em: <https://doi.org/10.1162/106365601750190398>.

HEESS, N.; TB, D.; S, S.; LEMMON, J.; MEREL, J.; WAYNE, G.; TASSA, Y.; EREZ, T.; WANG, Z.; ESLAMI, S. M. A.; RIEDMILLER, M.; SILVER, D. Emergence of locomotion behaviours in rich environments. **arXiv**, Jul 2017.

HEESS, N.; TB, D.; SRIRAM, S.; LEMMON, J.; MEREL, J.; WAYNE, G.; TASSA, Y.; EREZ, T.; WANG, Z.; ESLAMI, S. M. A.; RIEDMILLER, M. A.; SILVER, D. Emergence of locomotion behaviours in rich environments. **CoRR**, abs/1707.02286, 2017. Disponível em: <http://arxiv.org/abs/1707.02286>.

HENDERSON, P.; ISLAM, R.; BACHMAN, P.; PINEAU, J.; PRECUP, D.; MEGER, D. Deep reinforcement learning that matters. **CoRR**, abs/1709.06560, 2017. Disponível em: <http://arxiv.org/abs/1709.06560>.

HESTENES, M. R.; STIEFEL, E. Methods of conjugate gradients for solving linear systems. **Journal of research of the National Bureau of Standards**, v. 49, p. 409–436, 1952.

HO, J.; ERMON, S. Generative adversarial imitation learning. **CoRR**, abs/1606.03476, 2016. Disponível em: <http://arxiv.org/abs/1606.03476>.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 9, n. 8, p. 1735–1780, November 1997. ISSN 0899-7667. Disponível em: <http://dx.doi.org/10.1162/neco.1997.9.8.1735>.

HOUTHOOFT, R.; CHEN, R. Y.; ISOLA, P.; STADIE, B. C.; WOLSKI, F.; HO, J.; ABBEEL, P. Evolved policy gradients. **CoRR**, abs/1802.04821, 2018. Disponível em: <http://arxiv.org/abs/1802.04821>.

ILYAS, A.; ENGSTROM, L.; SANTURKAR, S.; TSIPRAS, D.; JANOOS, F.; RUDOLPH, L.; MADRY, A. Are deep policy gradient algorithms truly policy gradient algorithms? **CoRR**, abs/1811.02553, 2018. Disponível em: <http://arxiv.org/abs/1811.02553>.

INTEL. **Intel DevCloud**. 2018. Disponível em: <https://software.intel.com/en-us/ai-academy/devcloud>.

ISHII, S.; YOSHIDA, W.; YOSHIMOTO, J. Control of exploitation-exploration meta-parameter in reinforcement learning. **Neural networks : the official journal of the International Neural Network Society**, v. 15, n. 4-6, p. 665—687, 2002. ISSN 0893-6080. Disponível em: <https://doi.org/10.1016/S0893-6080(02)00056-4>.

JIE, T.; ABBEEL, P. On a connection between importance sampling and the likelihood ratio policy gradient. In: LAFFERTY, J. D.; WILLIAMS, C. K. I.; SHAWE-TAYLOR, J.; ZEMEL, R. S.; CULOTTA, A. (Ed.). **Advances in Neural Information Processing Systems 23**. Curran Associates, Inc., 2010. p. 1000–1008. Disponível em: <http://papers.nips.cc/paper/3922-on-a-connection-between-importance-sampling-and-the-likelihood-ratio-policy-gradient.pdf>.

JORDAN, J. Blog, **Normalizing your data (specifically, input and batch normalization).** 2018. Disponível em: <https://www.jeremyjordan.me/batch-normalization/>.

KAJITA, S.; KANEHIRO, F.; KANEKO, K.; YOKOI, K.; HIRUKAWA, H. The 3D Linear Inverted Pendulum Mode: A simple modeling for a biped walking pattern generation. In: **In Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems**. Hawaii, USA: IEEE, 2001.

KAKADE, S.; LANGFORD, J. Approximately optimal approximate reinforcement learning. In: **IN PROC. 19TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING**. [S.l.: s.n.], 2002. p. 267–274.

KHAN, N. A parallel implementation of the covariance matrix adaptation evolution strategy. **CoRR**, abs/1805.11201, 2018. Disponível em: <http://arxiv.org/abs/1805.11201>.

KINGMA, D. P.; BA, J. Computer science > learning adam: A method for stochastic optimization. **arXiv**, Dec 2014.

KITANO, H.; ASADA, M.; KUNIYOSHI, Y.; NODA, I.; OSAWAI, E.; MATSUBARA, H. Robocup: A challenge problem for ai and robotics. In: KITANO, H. (Ed.). **RoboCup-97: Robot Soccer World Cup I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 1–19. ISBN 978-3-540-69789-3.

Kober, J.; Peters, J. Learning motor primitives for robotics. In: **2009 IEEE International Conference on Robotics and Automation**. [S.l.: s.n.], 2009. p. 2112–2118. ISSN 1050-4729.

LECUN, Y.; MULLER, U.; BEN, J.; COSATTO, E.; FLEPP, B. Off-road obstacle avoidance through end-to-end learning. In: **Proceedings of the 18th International Conference on Neural Information Processing Systems**. Cambridge, MA, USA: MIT Press, 2005. (NIPS'05), p. 739–746. Disponível em: <http://dl.acm.org/citation.cfm?id=2976248.2976341>.

LEE, Y. **On First-Order Meta-Learning Algrithms**. 2018. `https://www.slideshare.net/YoonhoLee4/on-firstorder-metalearning-algorithms`. [Online; accessed at May 30, 2019].

LEVINE, S.; FINN, C.; DARRELL, T.; ABBEEL, P. End-to-end training of deep visuomotor policies. **J. Mach. Learn. Res.**, JMLR.org, v. 17, n. 1, p. 1334–1373, January 2016. ISSN 1532-4435. Disponível em: <http://dl.acm.org/citation.cfm?id=2946645.2946684>.

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>.

Liu, Y.; Gupta, A.; Abbeel, P.; Levine, S. Imitation from Observation: Learning to Imitate Behaviors from Raw Video via Context Translation. **arXiv e-prints**, p. arXiv:1707.03374, Jul 2017.

LU, C.; TANG, X. Surpassing human-level face verification performance on LFW with gaussianface. **CoRR**, abs/1404.3840, 2014. Disponível em: <http://arxiv.org/abs/1404.3840>.

MACALPINE, P.; BARRETT, S.; URIELI, D.; VU, V.; STONE, P. Design and Optimization of an Omnidirectional Humanoid Walk: A Winning Approach at the RoboCup 2011 3D Simulation Competition. In: **In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)**. Toronto, Ontario, Canada: AAAI, 2012.

MACALPINE, P.; BARRETT, S.; URIELI, D.; VU, V.; STONE, P. Design and optimization of an omnidirectional humanoid walk: A winning approach at the RoboCup 2011 3D simulation competition. In: **Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)**. Toronto, Ontario, Canada: AAAI, 2012.

MACALPINE, P.; COLLINS, N.; LOPEZ-MOBILIA, A.; STONE, P. Ut austin villa: Robocup 2012 3d simulation league champion. In: CHEN, X.; STONE, P.; SUCAR, L. E.; ZANT, T. van der (Ed.). **RoboCup 2012: Robot Soccer World Cup XVI**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 77–88. ISBN 978-3-642-39250-4.

MACALPINE, P.; STONE, P. Prioritized role assignment for marking. In: BEHNKE, S.; LEE, D. D.; SARIEL, S.; SHEH, R. (Ed.). **RoboCup 2016: Robot Soccer World Cup XX**. Berlin: Springer Verlag, 2016, (Lecture Notes in Artificial Intelligence). p. 306–18.

MACALPINE, P.; STONE, P. UT Austin Villa 3D Simulation Soccer Team 2017. In: . Nagoya, Japan: RoboCup Symposium, 2017.

MACALPINE, P.; STONE, P. UT Austin Villa: RoboCup 2017 3D simulation league competition and technical challenges champions. In: SAMMUT, C.; OBST, O.; TONIDANDEL, F.; AKYAMA, H. (Ed.). **RoboCup 2017: Robot Soccer World Cup XXI**. Japan: Springer, 2018, (Lecture Notes in Artificial Intelligence).

MACALPINE, P.; URIELI, D.; BARRETT, S.; KALYANAKRISHNAN, S.; BARRERA, F.; LOPEZ-MOBILIA, A.; STIURCA, N.; VU, V.; STONE, P. **UT Austin Villa 2011 3D Simulation Team Report**. Austin, Texas, December 2011.

MAGALHAES, G. I. **Reconhecimento de gestos da Linguagem Brasileira de Sinais**. Bachelor's Thesis — ITA, 2017.

MAXIMO, M. R. O. A. **Omnidirectional ZMP-based walking for a humanoid robot**. Dissertação (Master's Thesis) — Instituto Tecnológico de Aeronáutica, São José dos Campos, SP, Brazil, 2015.

MAXIMO, M. R. O. de A. **Otimizacao de Caminhada de Robos Humanoides**. Bachelor's Thesis — ITA, 2012.

MAXIMO, M. R. O. de A. **Automatic walking step duration through model predictive control**. Tese (Doutorado) — Aeronautics Institute of Technology, 2017.

MAXIMO, M. R. O. de A.; RIBEIRO, C. H. C. Zmp-based humanoid walking engine with arms movement and stabilization. In: **In Proceedings XXI Congresso Brasileiro de Automática**. Vitória, Brasil: [s.n.], 2016.

MENDONCA, R.; GUPTA, A.; KRALEV, R.; ABBEEL, P.; LEVINE, S.; FINN, C. Guided meta-policy search. **CoRR**, abs/1904.00956, 2019. Disponível em: <http://arxiv.org/abs/1904.00956>.

MISHRA, N.; ROHANINEJAD, M.; CHEN, X.; ABBEEL, P. Meta-learning with temporal convolutions. **CoRR**, abs/1707.03141, 2017. Disponível em: <http://arxiv.org/abs/1707.03141>.

MNIH, V.; BADIA, A. P.; MIRZA, M.; GRAVES, A.; LILLICRAP, T. P.; HARLEY, T.; SILVER, D.; KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. **CoRR**, abs/1602.01783, 2016. Disponível em: <http://arxiv.org/abs/1602.01783>.

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, February 2015. ISSN 00280836. Disponível em: <http://dx.doi.org/10.1038/nature14236>.

MUNIZ, F.; MAXIMO, M. R. O. A.; RIBEIRO, C. H. C. Keyframe movement optimization for simulated humanoid robot using a parallel optimization framework. In: **2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)**. Recife, Brazil: IEEE, 2016. p. 79–84.

MUNKHDALAI, T.; YU, H. Meta networks. In: PRECUP, D.; TEH, Y. W. (Ed.). **Proceedings of the 34th International Conference on Machine Learning**. International Convention Centre, Sydney, Australia: PMLR, 2017. (Proceedings of Machine Learning Research, v. 70), p. 2554–2563. Disponível em: <http://proceedings.mlr.press/v70/munkhdalai17a.html>.

MUZIO, A. F. V. **Deep Reinforcement Learning Applied to Humanoid Robots**. Bachelor's Thesis — ITA, 2017.

MüLLING, K.; KOBER, J.; KROEMER, O.; PETERS, J. Learning to select and generalize striking movements in robot table tennis. **The International Journal of Robotics Research**, v. 32, n. 3, p. 263–279, 2013. Disponível em: <https://doi.org/10.1177/0278364912472380>.

Naik, D. K.; Mammone, R. J. Meta-neural networks that learn by learning. In: **[Proceedings 1992] IJCNN International Joint Conference on Neural Networks**. [S.l.: s.n.], 1992. v. 1, p. 437–442 vol.1.

NG, A. Y.; RUSSELL, S. J. Algorithms for inverse reinforcement learning. In: **Proceedings of the Seventeenth International Conference on Machine Learning**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. (ICML '00), p. 663–670. ISBN 1-55860-707-2. Disponível em: <http://dl.acm.org/citation.cfm?id=645529.657801>.

NICHOL, A.; ACHIAM, J.; SCHULMAN, J. On first-order meta-learning algorithms. **CoRR**, abs/1803.02999, 2018. Disponível em: <http://arxiv.org/abs/1803.02999>.

NOROUZI, M.; BENGIO, S.; CHEN, Z.; JAITLY, N.; SCHUSTER, M.; WU, Y.; SCHUURMANS, D. Reward augmented maximum likelihood for neural structured prediction. **CoRR**, abs/1609.00150, 2016. Disponível em: <http://arxiv.org/abs/1609.00150>.

OBST, O.; ROLLMAN, M. Spark – a generic simulator for physical multiagent simulation. **Computer Systems Science and Engineering**, v. 5, p. 347–356, 2005.

OLAH, C. **Understanding LSTM Networks**. 2015. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Online; accessed at May 30, 2019].

OORD, A. van den; DIELEMAN, S.; ZEN, H.; SIMONYAN, K.; VINYALS, O.; GRAVES, A.; KALCHBRENNER, N.; SENIOR, A.; KAVUKCUOGLU, K. Wavenet: A generative model for raw audio. In: **Arxiv**. [s.n.], 2016. Disponível em: <https://arxiv.org/abs/1609.03499>.

OPENAI. **OpenAI Five**. 2018. `https://blog.openai.com/openai-five/`.

OPENAI. **How to Train Your OpenAI Five**. 2019. `https://openai.com/blog/how-to-train-your-openai-five/`.

PAPIEROK, S.; NOGLIK, A.; PAULI, J. Application of reinforcement learning in a real environment using an rbf network. In: CITESEER. **Proceedings of the International Workshop on Evolutionary Learning for Autonomous Robot Systems**. [S.l.], 2008. p. 17–22.

PENG, X. B.; ABBEEL, P.; LEVINE, S.; PANNE, M. van de. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. **ACM Transactions on Graphics (Proc. SIGGRAPH 2018 - to appear)**, v. 37, n. 4, 2018.

PLAPPERT, M.; HOUTHOOFT, R.; DHARIWAL, P.; SIDOR, S.; ABBEEL, P.; ANDRYCHOWICZ, M.; CHEN, R.; CHEN, X.; ASFOUR, T. Better exploration with parameter noise. In: . [s.n.], 2017. Disponível em: <https://blog.openai.com/better-exploration-with-parameter-noise/>.

POMERLEAU, D. A. Advances in neural information processing systems 1. In: TOURETZKY, D. S. (Ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. cap. ALVINN: An Autonomous Land Vehicle in a Neural Network, p. 305–313. ISBN 1-558-60015-9. Disponível em: <http://dl.acm.org/citation.cfm?id=89851.89891>.

RAVI, S.; LAROCHELLE, H. Optimization as a model for few-shot learning. In: **5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings**. [s.n.], 2017. Disponível em: <https://openreview.net/forum?id=rJY0-Kcll>.

REIS, L. P.; LAU, N.; ABDOLMALEKI, A.; SHAFII, N.; FERREIRA, R.; PEREIRA, A.; SIMõES, D. **FC Portugal 3D Simulation Team: Team Description Paper 2017**. 2017. `https://www.robocup2017.org/file/symposium/soccer_sim_3D/FCPortugal3D_TDP_2017.pdf`. [Online; accessed at May 23, 2019].

ROSS, S.; BAGNELL, D. Efficient reductions for imitation learning. In: TEH, Y. W.; TITTERINGTON, M. (Ed.). **Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics**. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010. (Proceedings of Machine Learning Research, v. 9), p. 661–668. Disponível em: <http://proceedings.mlr.press/v9/ross10a.html>.

ROSS, S.; GORDON, G. J.; BAGNELL, J. A. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. **arXiv e-prints**, p. arXiv:1011.0686, Nov 2010.

RUBINSTEIN, R. Y.; KROESE, D. P. **The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics)**. Berlin, Heidelberg: Springer-Verlag, 2004. ISBN 038721240X.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 533, 1988.

RUMMERY, G. A.; NIRANJAN, M. **On-Line Q-Learning Using Connectionist Systems**. [S.l.], 1994.

SALIMANS, T.; HO, J.; CHEN, X.; SUTSKEVER, I. Evolution strategies as a scalable alternative to reinforcement learning. **CoRR**, abs/1703.03864, 2017. Disponível em: <http://arxiv.org/abs/1703.03864>.

SCHAAL, S. **Is imitation learning the route to humanoid robots?** 1999.

SCHMIDHUBER, J. **Evolutionary Principles in Self-Referential Learning. On Learning now to Learn: The Meta-Meta-Meta...-Hook**. Dissertação (Diploma Thesis) — Technische Universitat Munchen, Germany, 14 May 1987. Disponível em: <http://www.idsia.ch/˜juergen/diploma.html>.

SCHULMAN, J. Lecture, **Advanced Policy Gradient Methods: Natural Gradient, TRPO and More**. 2017. Disponível em: <https://drive.google.com/file/d/0BxXI_RttTZAhMVhsNk5VSXU0U3c/view>.

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.; MORITZ, P.; LEVINE, S.; JORDAN, M. I.; ABBEEL, P. High-dimensional continuous control using generalized advantage estimation. **CoRR**, abs/1506.02438, 2015. Disponível em: <http://arxiv.org/abs/1506.02438>.

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>.

SHON, A. P.; GROCHOW, K.; RAO, R. P. N. Robotic imitation from human motion capture using gaussian processes. In: **In Proceedings of the IEEE/RAS International Conference on Humanoid Robots (Humanoids**. Tsukuba, Japan: [s.n.], 2005.

SILVER, D. Lecture, **Markov Decision Processes**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf>.

SILVER, D. Lecture, **Planning by Dynamic Programming**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf>.

SILVER, D. Lecture, **Policy Gradients**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf>.

SILVER, D. Lecture, **Value Function Approximation**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf>.

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **CoRR**, abs/1712.01815, 2017. Disponível em: <http://arxiv.org/abs/1712.01815>.

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **CoRR**, abs/1712.01815, 2017. Disponível em: <http://arxiv.org/abs/1712.01815>.

SILVER, D.; SCHRITTWIESER, J.; SIMONYAN, K.; ANTONOGLOU, I.; HUANG, A.; GUEZ, A.; HUBERT, T.; BAKER, L.; LAI, M.; BOLTON, A.; CHEN, Y.; LILLICRAP, T.; HUI, F.; SIFRE, L.; DRIESSCHE, G. van den; GRAEPEL, T.; HASSABIS, D. Mastering the game of go without human knowledge. **Nature**, Macmillan Publishers Limited, part of Springer Nature. All rights reserved., v. 550, p. 354–, October 2017. Disponível em: <http://dx.doi.org/10.1038/nature24270>.

SKINNER, B. **Science And Human Behavior**. Free Press, 1953. (A Free Press paperback). ISBN 9780029290408. Disponível em: <https://books.google.com.br/books?id=Pjjknd1HREIC>.

STADIE, B. C.; YANG, G.; HOUTHOOFT, R.; CHEN, X.; DUAN, Y.; WU, Y.; ABBEEL, P.; SUTSKEVER, I. Some considerations on learning to explore via meta-reinforcement learning. **CoRR**, abs/1803.01118, 2018. Disponível em: <http://arxiv.org/abs/1803.01118>.

STONE, P. **Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer**. Cambridge, MA, USA: MIT Press, 2000. ISBN 0262194384.

SUN, R. Bottom-up learning and top-down learning. In: _____. **Encyclopedia of the Sciences of Learning**. Boston, MA: Springer US, 2012. p. 479–481. ISBN 978-1-4419-1428-6. Disponível em: <https://doi.org/10.1007/978-1-4419-1428-6_387>.

SUNG, F.; ZHANG, L.; XIANG, T.; HOSPEDALES, T. M.; YANG, Y. Learning to learn: Meta-critic networks for sample efficient learning. **CoRR**, abs/1706.09529, 2017. Disponível em: <http://arxiv.org/abs/1706.09529>.

SUTSKEVER, I.; MARTENS, J.; DAHL, G.; HINTON, G. On the importance of initialization and momentum in deep learning. In: **Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28**. JMLR.org, 2013. (ICML'13), p. III–1139–III–1147. Disponível em: <http://dl.acm.org/citation.cfm?id=3042817.3043064>.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. MIT Press, 1998. Disponível em: <http://www.cs.ualberta.ca/~sutton/book/the-book.html>.

TANIKIC, D.; DESPOTOVIC, V. Artificial intelligence techniques for modelling of temperature in the metal cutting process. **IntechOpen**, September 2012.

THORNDIKE, E. L. A proof of the law of effect. **Science**, American Association for the Advancement of Science, v. 77, n. 1989, p. 173–175, 1933. ISSN 0036-8075. Disponível em: <http://science.sciencemag.org/content/77/1989/173.2>.

THRUN, S.; PRATT, L. Learning to learn: Introduction and overview. In: _____. **Learning to Learn**. Boston, MA: Springer US, 1998. p. 3–17. ISBN 978-1-4615-5529-2. Disponível em: <https://doi.org/10.1007/978-1-4615-5529-2_1>.

TODOROV, E.; EREZ, T.; TASSA, Y. Mujoco: A physics engine for model-based control. In: **IROS**. [S.l.]: IEEE, 2012. p. 5026–5033. ISBN 978-1-4673-1737-5.

TORABI, F.; WARNELL, G.; STONE, P. Behavioral Cloning from Observation. **arXiv e-prints**, p. arXiv:1805.01954, May 2018.

URIELI, D.; MACALPINE, P.; KALYANAKRISHNAN, S.; BENTOR, Y.; STONE, P. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In: TUMER, K.; YOLUM, P.; SONENBERG, L.; STONE, P. (Ed.). **Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)**. [S.l.]: IFAAMAS, 2011. v. 2, p. 769–776. ISBN 978-0-9826571-5-7.

VARDA, K. **Protocol Buffers: Google's Data Interchange Format**. [S.l.], 6 2008. Disponível em: <http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html>.

VINYALS, O.; BABUSCHKIN, I.; CHUNG, J.; MATHIEU, M.; JADERBERG, M.; CZARNECKI, W. M.; DUDZIK, A.; HUANG, A.; GEORGIEV, P.; POWELL, R.; EWALDS, T.; HORGAN, D.; KROISS, M.; DANIHELKA, I.; AGAPIOU, J.; OH, J.; DALIBARD, V.; CHOI, D.; SIFRE, L.; SULSKY, Y.; VEZHNEVETS, S.; MOLLOY, J.; CAI, T.; BUDDEN, D.; PAINE, T.; GULCEHRE, C.; WANG, Z.; PFAFF, T.; POHLEN, T.; WU, Y.; YOGATAMA, D.; COHEN, J.; MCKINNEY, K.; SMITH, O.; SCHAUL, T.; LILLICRAP, T.; APPS, C.; KAVUKCUOGLU, K.; HASSABIS, D.; SILVER, D. **AlphaStar: Mastering the Real-Time Strategy Game StarCraft II**. 2019. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`.

VUKOBRATOVIĆ, M.; BOROVAC, B. Zero-moment point - thirty five years of its life. **International Journal of Humanoid Robotics**, v. 1, n. 1, p. 157–173, 2005. Disponível em: <http://www.cs.cmu.edu/~cga/legs/vukobratovic.pdf>.

WANG, J. X.; KURTH-NELSON, Z.; TIRUMALA, D.; SOYER, H.; LEIBO, J. Z.; MUNOS, R.; BLUNDELL, C.; KUMARAN, D.; BOTVINICK, M. Learning to reinforcement learn. **CoRR**, abs/1611.05763, 2016. Disponível em: <http://arxiv.org/abs/1611.05763>.

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, Cambridge, UK, May 1989. Disponível em: <http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf>.

WIERSTRA, D.; SCHAUL, T.; GLASMACHERS, T.; SUN, Y.; PETERS, J.; SCHMIDHUBER, J. Natural evolution strategies. **J. Mach. Learn. Res.**, JMLR.org, v. 15, n. 1, p. 949–980, January 2014. ISSN 1532-4435. Disponível em: <http://dl.acm.org/citation.cfm?id=2627435.2638566>.

WONGSUPHASAWAT, K.; SMILKOV, D.; WEXLER, J.; WILSON, J.; MANE, D.; FRITZ, D.; KRISHNAN, D.; VIEGAS, F.; WATTENVERG, M. Visualizing dataflow graphs of deep learning models in tensorflow. In: . [s.n.], 2017. Disponível em: <http://idl.cs.washington.edu/files/2018-TensorFlowGraph-VAST.pdf>.

XIE, Z.; CLARY, P.; DAO, J.; MORAIS, P.; HURST, J.; PANNE, M. van de. **Iterative Reinforcement Learning Based Design of Dynamic Locomotion Skills for Cassie**. 2019.

XIONG, W.; DRO, J.; HUANG, X.; SEIDE, F.; SELTZER, M.; STOLCKE, A.; YU, D.; ZWEIG, G. Achieving human parity in conversational speech recognition. **CoRR**, abs/1610.05256, 2016. Disponível em: <http://arxiv.org/abs/1610.05256>.

XU, B.; WANG, N.; CHEN, T.; LI, M. Empirical evaluation of rectified activations in convolutional network. **arXiv**, Nov 2015.

XU, X.; HU, D.; LU, X. Kernel-based least squares policy iteration for reinforcement learning. v. 18, p. 973–92, 08 2007.

XU, Y.; VATANKHAH, H. Simspark: An open source robot simulator developed by the robocup community. In: BEHNKE, S.; VELOSO, M.; VISSER, A.; XIONG, R. (Ed.). **RoboCup 2013: Robot World Cup XVII**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 632–639. ISBN 978-3-662-44468-9.

ZHENG, A. **Evaluating Machine Learning Models**. 2015. `https://www.oreilly.com/ideas/evaluating-machine-learning-models/page/5/hyperparameter-tuning`. [Online; accessed at May 19, 2019].

Appendix A - Hyperparameters

TABLE A.1 – Hyperparameters - RNR, RRR, RNR+RRR, RNR+RRR+RISD.

Hyperparameter	Value
Learning Rate	10^{-3}
Timesteps per Actorbatch	4096
Batch Size	256
Epochs	10
γ	0.99
λ	0.95
Timesteps (Total)	8×10^8
Clip Parameter	0.2
Entropy Coefficient	0.0
RNR Parameters Vector	$\begin{bmatrix} 1.0 & -0.3 & 2.0 \end{bmatrix}$
RRR Parameters Vector	$\begin{bmatrix} 1.0 & 1.0 & 2.0 & 2.0 & 2.0 & 1.0 & 2.0 & 2.0 & 2.0 \\ 1.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 & 3.0 & \\ 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & & & & \end{bmatrix}$

TABLE A.2 – Hyperparameters - Reinforcement “supervised” - First Session.

Hyperparameter	Value
Learning Rate	10^{-4}
Timesteps per Actorbatch	4096
Batch Size	256
Epochs	10
γ	0.99
λ	0.95
Timesteps (Total)	10^6
Clip Parameter	0.2
Entropy Coefficient	0.0

TABLE A.3 – Hyperparameters - HLM+RNR, HLM+RNR+RET - First Session.

Hyperparameter	Value
Learning Rate	10^{-6}
Timesteps per Actorbatch	4096
Batch Size	2048
Epochs	10
γ	0.99
λ	0.95
Timesteps (Total)	4×10^6
Clip Parameter	0.2
Entropy Coefficient	0.0
RNR Parameters Vector	$\begin{bmatrix} 1.0 & -0.3 & 2.0 \end{bmatrix}$

TABLE A.4 – Hyperparameters from Kick Benchmark after random search.

Hyperparameter	Value
Learning Rate	10^{-6}
Timesteps per Actorbatch	512
Batch Size	128
Epochs	30
γ	0.99
λ	1.0
Timesteps (Total)	4×10^6
Clip Parameter	0.21
Entropy Coefficient	0.01

TABLE A.5 – Hyperparameters - Precise Kick.

Hyperparameter	Value
Learning Rate	10^{-6}
Timesteps per Actorbatch	4096
Batch Size	1024
Epochs	30
γ	0.999
λ	1.0
Timesteps (Total)	12×10^6
Clip Parameter	0.29
Entropy Coefficient	0.01
Reward Multiplicative Factor	0.1

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N	4. N DE PÁGINAS
DM	November 19th, 2018	DCTA/ITA/TC-057/2018	193

5. TÍTULO E SUBTÍTULO:
Imitation Learning and Meta-Learning for Optimizing Humanoid Robot Motions

6. AUTOR(ES):
Luckeciano Carvalho Melo

7. INSTITUIÇÃO(ÕES)/ÓRGAO(S) INTERNO(S)/DIVISÃO(ÕES):
Aeronautics Institute of Technology – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:
Deep Reinforcement Learning; Robotics; Artificial Intelligence

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:
Aprendizagem (inteligência artificial); Dinâmica de Robôs; Redes neurais; Robótica; Computação.

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**
ITA, São José dos Campos, 2018. Trabalho de Graduação.

11. RESUMO:
O controle de diversos graus de liberdade para robôs humanoides é reconhecido como um dos problemas mais difíceis da robótica. Neste trabalho, primeiramente propõe-se uma estrutura de aprendizado que não apenas imita, mas também otimiza o movimento do robô humanóide de acordo com alguma tarefa usando Aprendizado por Reforço Profundo, sem fazer qualquer suposição sobre a implementação subjacente do movimento. Posteriormente, propõe-se e aplica-se um algoritmo de *Meta-Learning* chamado *Bottom-Up Meta-Policy Search*, que destila algumas políticas especialistas para criar uma meta-política capaz de generalizar para tarefas não vistas durante o treino. Por fim, analisa-se o movimento de caminhada, primeiro propondo e aplicando o algoritmo *Proximal Policy Natural Evolution Strategy* para otimização de parâmetros da *engine* da caminhada e, em seguida, utilizando a metodologia de imitação previamente utilizada para copiar esta *engine* para uma rede neural. Todos os experimentos foram conduzidos no domínio do *RoboCup 3D Soccer Simulation*, culminando no controle do chute com precisão a nível de passe, na melhoria dos parâmetros da caminhada em tarefas específicas e em uma *engine* completamente baseada em uma rede neural.

12. GRAU DE SIGILO:
(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**