View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UT Digital Repository

Copyright

by

Alon Farchy

2012

The Thesis Committee for Alon Farchy
certifies that this is the approved version of the following thesis:

Learning in Simulation for Real Robots

Committee:

____
Peter Stone, Supervisor

____
Dana Ballard

Learning in Simulation for Real Robots

by

Alon Farchy, B.S.C.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2012

Acknowledgments

I would like to give thanks to the members of the UT Austin Villa team for their help and support on this project. Special thanks to Samuel Barrett for his help in understanding the team code base and the HTWK walk, to Patrick MacAlpine for his assistance in using the simulation optimization framework, and to Todd Hester for his help in understanding the M5 algorithm. Finally, I give thanks to Peter Stone for his guidance and support throughout this project.

ALON FARCHY

The University of Texas at Austin
May 2012

Learning in Simulation for Real Robots

Alon Farchy, MSCompSci
The University of Texas at Austin, 2012

Supervisor: Peter Stone

Simulation is often used in research and industry as a low cost, high efficiency alternative to real model testing. Simulation has also been used to develop and test powerful learning algorithms. However, optimized values in simulation do not translate directly to optimized values in application. In fact, heavy optimization in simulation is likely to exploit the simplifications made in simulation. This observation brings to question the utility of learning in simulation.

The UT Austin Villa 3D Simulation Team developed an optimization framework on which a robot agent was trained to maximize the speed of an omni-directional walk. The resulting agent won first place in the 2011 RoboCup 3D Simulation League.

This thesis presents the adaptation of this optimization framework to

learn parameters in simulation that improved the forward walk speed of the real Aldebaran Nao. By constraining the simulation with tree models learned from the real robot, and manually guiding the optimization based on testing on the real robot, the Nao's walk speed was improved by about 30%.

Contents

Acknowledgments **iv**

Abstract **v**

Chapter 1 Introduction **1**

Chapter 2 Background Information **3**

2.1 Nao 3
2.2 HTWK Walk Engine 4
2.3 SimSpark 5
2.4 Optimization Framework 6
2.4.1 CMA-ES 8

Chapter 3 The Search for Optimized Parameters **10**

3.1 Predicting Joint Angles 11
3.1.1 Description of M5P 14
3.2 Narrowing the Optimization 15
3.3 Selecting the Right Parameters 17

Chapter 4 Discussion on Simulation Learning **21**

4.1 Assessment of the Approach 21
4.2 Related Work 22
4.3 Failed attempts 24
4.3.1 Predicting Joint Commands 24

4.3.2 Predicting Joints on the Real Robot 25
4.3.3 Matching the Frame Rate 25

Chapter 5 Lessons and Conclusion **27**

Bibliography **28**

Chapter 1

Introduction

In 2004, the UT Austin Villa team published a paper [7] showing how machine learning can be used on a quadrupedal robot called the Sony Aibo to improve its walk speed significantly. These robots were used in the RoboCup soccer competition and clearly outmatched the best hand-tuned walk routines. Since then, the RoboCup legged competition has moved to using the Aldebaran Nao robots. In contrast to the four legged Aibos, these robots are bipedal and come with all of the challenges expected from walking on two legs. Chief among these challenges is the observation that these robots are unstable, fall easily, and break frequently. While the Aibos were able to try a large range of parameters to improve their walk, the same could not be done on the Naos, as a poor choice of parameters could cost several thousands of dollars in damage.

In 2011, the UT Austin Villa team once again used machine learning to outdo the competition in the RoboCup 3D simulation league [9] Using the CMA-ES evolutionary learning algorithm and using Condor to perform large-scale distributed computing, the team was able to run thousands of simulated trials to find parameter values that transformed a relatively slow walk into the fastest simulated walk for the Nao in the world. In simulation there is little cost to running and testing an instance of parameter values, and these tests can be run in much less time. In addition, by utilizing Condor, it was possible to run up to 150 robust tests simultaneously and have them complete in less

than twenty minutes. It is therefore tempting to utilize the same infrastructure to improve the walk of the real robot.

Unfortunately, to do so is no simple task. Even when using the same code base, to simply take the same parameters that are optimized for simulation and run them on the robot would achieve little more than a broken robot. The simulated model of the robot is imperfect, and the simulation environment poorly represents the robotics laboratory. More importantly, the physics engine in the simulator, while considered quite good, simply does not produce the same results as the real world.

The work of this thesis is an attempt compensate for these deficiencies in order to use simulation to find parameter values that improve the walk speed of the Nao. By constraining the simulation sufficiently and manually guiding the optimization framework, this project has succeeded in producing parameters that create a walk that is as stable as the original and walks forward about 30% faster.

This thesis is organized as follows: The next chapter describes the background information required to understand the major work of this thesis. Chapter 3 gives describes the changes that were required to achieve the previously mentioned results. Chapter 4 provides a discussion on these changes and the concept of learning in simulation, relating to similar work. Finally, chapter 5 concludes.

Chapter 2

Background Information

2.1 Nao

Figure 2.1: Aldebaran Nao

The Aldebaran Nao is built in the likeness of a small human and stands a little more than half a meter tall. It has twenty five degrees of freedom, eleven of which are in the pelvis and legs, with which this project is concerned. There is one camera on the Nao's forehead and another where the mouth would

be. Additionally, the Nao has foot pressure sensors, two gyro-meters, and an accelerometer.

The UTCS robotics laboratory is in the basement of a building on campus and is shielded from most outside lighting and disturbances. The robot walks on a thin green carpet field with white duct taped lines marking the typical soccer boundaries.

2.2 HTWK Walk Engine

Parameter	**Description**
angleXScale	Scale for sensor value of body roll.
angleYOffset	Offset for sensor value of body pitch.
angleYScale	Scale for sensor value of body pitch.
startLength	Used in calculating initial ramp up.
numFrames	Number of frames to take two steps.
swing	Amplitude of the swing calculation.
knee	Base of the leg lifting calculation.
vShort	Factor for the leg lifting calculation.
aShort	Amplitude of the leg lifting calculation.
oShort	Offset of the leg lifting calculation.
balanceGyro1	Factor for body pitch in calculating hip pitch.
balanceGyro2	Factor for body pitch in calculating knee pitch.
balanceGyro3	Factor for body roll in calculating hip roll.
balanceGyro4	Factor for body roll in calculating ankle roll.
vSwing	Factor for the swing calculation.
oSwing	Offset for the swing calculation.
speed	Minimum speed of the walk.

Table 2.1: The HTWK walk parameters examined this project.

The Naos come equipped with Aldebaran's own closed-source walk engine. However, there have been several projects and papers over the last few years which have produced faster and more robust walks for the RoboCup competitions. This is a challenging feat, and while the UT Austin Villa team

also created a customized walk for the Nao, it was deemed too unstable, and the default walk was used in RoboCup 2011.

This year, the team has ported a walk from the German HTWK team and improved on it. A detailed description of this walk is described in [2]. Seventeen parameters from this walk were examined (Table 2.1). A phase of the walk is the time it takes the robot to take two steps, and is measured by the *numFrames* parameter. A step consists of three components. The first component is shifting the center of mass. Then, the back leg is lifted by bending according to the *knee*, *vShort*, *oShort*, and *aShort* parameters. Finally, the lifted leg is swung forward according to the *swing*, *vSwing* and *oSwing* parameters. The *speed* parameter determines the minimum speed of the walk. The rest of the parameters scale and offset the sensor values of the gyro-meters, which are used as the closed loop component during the calculations of the three components.

2.3 SimSpark

RoboCup 3D simulation league uses the SimSpark multi-agent simulator, which was designed by the RoboCup initiative. The physics engine that the simulator uses is called the Open Dynamics Engine (ODE) which has both rigid body dynamics and collision detection. While ODE is quite good at these tasks, it has several shortcomings. For example, there is no friction model on hinges in ODE, and so no friction acts on the simulated robot's joints [1].

Every 20 milliseconds, the physics simulator takes a step and sends sensor information to the simulated agent. At that point, the behavior code can use the sensor information to determine the robot's next action. To walk, the behavior sends the walk engine a walk request, which uses the sensor information to determine the next desired joint commands. To achieve these joint angles, torques values are calculated on each joint, which are then sent back to the simulator.

The simulated agent was originally based on an older model of the Nao,

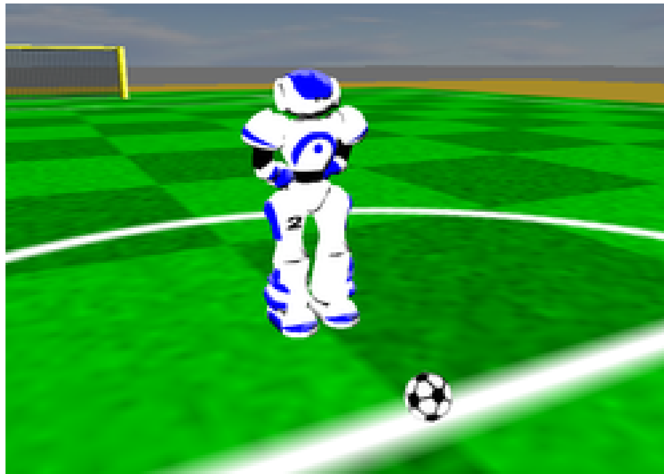

Figure 2.2: Simulated Nao agent

and so some of the dimensions and masses were incorrect. Also, the details in the model of the Nao is greatly approximated. For example, in the feet, where an accurate model would be the most important to accurately simulated a walk, the foot model looks more like perfectly rounded shoes, in comparison to the oddly shaped feet of the real robot. The simulated laboratory is a completely flat surface on which the robot walks, which is a poor representation of the green carpet used in the lab.

2.4 Optimization Framework

This is a brief summary on how the UT Austin Villa 3D Simulation team was able to use machine learning to create the fasted simulated walk for the Nao in the world. For a complete paper on the topic, see [9].

The walk engine for the agent has a component which parses parameter values from a file. These parameters correspond to the seventeen parameters described for the HTWK walk. However, the machine learning algorithm is blind as to what these parameters actually do.

By some manual trial and error, a robust optimization routine called OmniWalk was created for the simulation. The routine consists of a series of

targets to which the agent attempts to walk towards, with a time limit on each target. The targets are always relative to the agent's position and orientation, so failing to reach a previous target does not affect the next target. To get to a target, the agent is ordered to move in the forward and side directions at a constant speed towards the target. Since the agent's walk is omni-directional, it is not evaluated on which way it is facing, only how close it comes as to the target. However, it was assumed that the agent would walk faster facing forward, and so the agent is also made to always turn towards the target.

A trial is seven consecutive runs of the same optimization routine. At the end of each trial a fitness score is assigned, which is used to rank how good one instance of parameter values is compared to another. The team decided to use the covariance matrix adaptation evolution strategy (CMA-ES) algorithm, implemented in Java, to perform the optimization. A full description of CMA-ES is provided at the end of this section. For each generation, CMA-ES produces a population size of parameter samples to evaluate and waits for the results, which it then uses to determine what to try in the next generation.

Condor is used to distribute the task of evaluating the population across different machines on the department clusters. Up to 150 jobs can be run simultaneously, and so the population size is always fixed to 150. The framework will rerun each sample if they fail (for whatever reason) until some minimum number of trials completed successfully, and the algorithm can move to the next generation.

This learning framework is very good at finding the parameter values that will exploit the fact that the agent is in simulation. For example, to move it's joints, the agent calculates a torque values for each joint that will achieve the desired joint command. However, there is no upper bound on this torque value, and there is no friction model in the joints. Therefore, the learning framework will exploit the fact that it can move the joints much faster than would be possible on the real robot. Worse, while the agent is penalized for falling during the optimization, the real robot is far more unstable than the simulated one. Subsequently, it is difficult to force the learning framework to

find a walk optimized for speed that will not make the real robot break its neck.

2.4.1 CMA-ES

The covariance matrix adaptation evolution strategy (CMA-ES) is described in detail by Nikolaus Hansen in [5]. This subsection provides a brief summary of its operation.

Then goal of this algorithm is to optimize a set of parameters by searching the parameter space. CMA-ES is an evolutionary algorithm. It begins with user-defined initial values and variances for each parameter. Additionally, the user provides a fixed number of generations and a fixed population size. For each generation, CMA-ES samples a population from the parameter space using a multivariate normal distribution, which is defined by a mean and a covariance matrix. Initially, the mean of the distribution is the initial parameters and the covariance matrix is derived directly from the initial variances. Each of these samples then needs to be evaluated, and a fitness is returned to CMA-ES.

In order to proceed to the next generation, CMA-ES needs to determine the new mean and covariance matrix from which to sample. The mean is shifted from its previous position towards the weighted average of some number of the top valued samples (usually half). The new covariance matrix is also derived from these top samples, but is adapted based on the trajectory of the mean over the past few generations. The new distribution is then re-sampled for the next generation.

One reason this algorithm is appropriate for this project is that a large portion of its computation, the evaluation of a population, can be parallelized. Evaluating each sample of a population can be accomplished using a different simulation process. Using Condor (see [8]), up to 150 of these processes can be distributed across the machines in the department cluster. Additionally, CMA-ES accommodates for parameter spaces which are discontinuous or have

local optima, as is likely the case for the walk parameters.

Chapter 3

The Search for Optimized Parameters

The following subsections each explain the changes made either to the robot or to the simulator in order to use the parameters learned in simulation to improve the Nao's walking speed. For example, the simulated model used in the previous year's competition was based on an old version of the Nao, and so the masses of each component of the robot were updated. However, most of this work does not attempt to the repair the physics simulation nor the simulated models and environment. To repair the former would require a thorough investigation of the Open Dynamics Engine to determine in which meaningful way it is different from the real world in relation to this problem. Then, enough expertise would have to be developed about the ODE code base to correct it and add the missing models. This is certainly doable, as the ODE is open source, but unfortunately was not in the time constraints of this project.

Instead, for the most part, a black box approach was taken for the physics simulator. Given that the physics simulator is inaccurate and cannot be changed, how can artificial forces be sent to the simulator that will compensate for its inaccuracies enough to learn a usable optimized walk?

3.1 Predicting Joint Angles

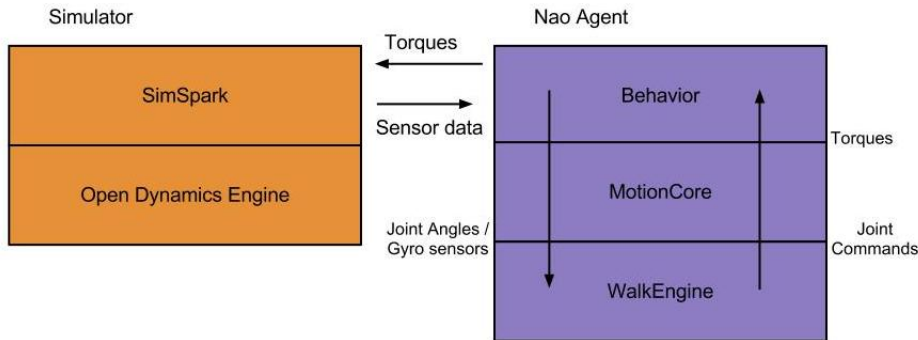

Figure 3.1: This greatly simplified diagram represents the actions that occur during a frame in simulation. First, the physics engine (ODE) updates its state and informs the RoboCup simulator SimSpark. SimSpark notifies the agent of its sensor information. During optimization, the agent is also given its ground truth location to calculate its fitness. The agent's behavior uses this sensor information to determine its next action and issues a walk request to the motion core. The motion core passes the walk request, along with the sensor data to the walk engine, which responds with joint commands. Those joint commands are converted into torques which will achieve those commands, and finally those torques are sent back to the physics engine. It is possible to manipulate the joint commands after the walk engine issues them in order to simulate what would happen on the real robot via prediction.

When the robot is issued a joint command, it is desirable that both the simulated agent and actual robot end up with the same joint configuration. That way, if the physics simulation is at least accurate enough to determine if a joint configuration is stable or not, then it should be able to determine if the robot will stand or not during optimization.

The real robot receives a new frame on which to perform calculations and send joint commands every 10 milliseconds, while the frame rate of the simulation is 20 milliseconds. Special code is added to the simulated agent to account for this discrepancy. At each frame, the walk engine determines the next set of joint angles that it would like the robot to achieve and requests that these angles be achieved in the minimal amount of time.

It was clear from the beginning that the joint angles achieved between each frame is different on the simulator from the real robot. The walk developed by the UT Austin Villa team for the previous year's competition was quite stable in simulation, while the same code and parameters on the real robot did not walk nearly so well. If nothing else, the real world physics appears much more random than in the ODE.

The approach taken to resolve this issue was to discover a mapping from the robot's current state and joint commands to the resulting joint angles using machine learning. Theoretically, if this mapping was accurate enough, it could be forcefully simulated during optimization. After optimization, since the walk would change as parameters were optimized, the mapping may have to be relearned using the current best parameter values. The walk would improve iteratively, first improving the simulator by walking on the robot, and then improving the robot by optimizing parameters on the simulator. A diagram of this process is provided at the end of this chapter.

For these experiments, the head and arm angles of the robot were fixed, and so only the 12 joints of the hips and legs were of interest (There are only 11 degrees of freedom since the hip joints move together). The robot was ordered to walk forward for some period of time. At every other frame (to compensate for the difference in frame rate), the robot would record its current state, which is its joint angles and center of mass, and the next set of joint commands. These make 27 inputs and 12 output joints.

The open source machine learning framework Weka was used to estimate this mapping. Various algorithms were tried, and the M5P tree regression algorithm was chosen as a tradeoff between runtime and prediction error. This algorithm produces a binary tree model. Details on how this model is constructed and evaluated are provided at the end of this section.

The algorithm for evaluating the models on a set of inputs was rewritten from scratch and incorporated into the simulated agent's code. After the walk engine determined which joint commands to send, the model for each joint is applied to learn which joint angles would be expected on the real robot. With

Class	Runtime (s)	RAE (%)	RRSE (%)
SimpleLinearRegression	0.35	21.7	25.5
LinearRegression	1.18	16.0	17.6
LeastMedSq	169.0	16.9	19.2
PaceRegression	2.21	15.9	17.5
MultilayerPerceptron	376	12.9	14.2
DecisionStump	1.82	56.8	57.5
M5P	37.3	11.6	16.3
REPTree	3.68	12.4	14.5
IBk	128.9	12.3	16.6
RegressionByDiscretization	20.2	16.7	18.1

Table 3.1: These regression classes were tested with the joint command to joint value data using ten fold cross validation. Here only one joint, HipYawPitch, is presented since this joint was consistently the most inaccurately predicted. RAE stands for relative absolute error, and RRSE is root relative squared error.

this information, it was easy to estimate the difference between the model's prediction and the output joint angles calculated by the physics engine. Every joint angle had at least a 0.1 radian difference from the expected value at nearly every frame.

The first attempt at correcting this error was simply to forcefully change the joint commands to the predicted joint angles, with the expectation that the torques calculated to achieve the requested joint commands were relatively accurate. This expectation turned out to be true. Many of the expected values were achieved much more closely than before, though still with some significant error.

Still, the predicted joint angles were noisy in comparison to the smooth joint commands from the walk engine. The noise level of the real robot's movement fell somewhere between these two values. The solution was to find a balance between trying to simulate the real robot's movements and keeping to the walk engine's original plan. By trial and error, a componentization of the original commands and expected joint angles was found that was both smoother and had a minimal error between the expected and actual output

angles. The new commands were the sum of 70% of the original commands, and 30% of the expected angles.

While this method worked well for simply walking forward, it was necessary to have the robot perform actions similar to whatever the simulated agent might perform. As discussed in the next section, the simulation's optimization routine was required to be able to walk forward and turn. Turning in simulation incorrectly caused the robot to fall since the models were only based on forward walking data. To get data relevant to diverse types of movements, the walk requests issued while the simulation was running the OmniWalk optimization routine were recorded and re-run on the robot. The models produced proved to be more robust, and did not cause the agent to fall while turning in any direction.

3.1.1 Description of M5P

It is worthwhile to look at how M5 trees are created and evaluated in order to understand why they are the appropriate choice for this data set. For each joint, a database of training data is provided from the robot, which has 27 inputs (12 joint commands + 12 joint angles + 3 center of mass), and a single output (the observed joint angle in the next frame). A detailed description on how M5 builds its models is available in its introductory 1992 paper [11], as well as in [14]. What follows is a brief summary of those description.

Conceptually, the M5 tree is a discrete decision tree that utilizes linear regressions at the leaves instead of constant values. Given a set of inputs, the output value can be predicted by traversing the tree to its leaves and then applying the linear function present at that leaf. To traverse the tree, at each node, the value of a selected input is compared to a learned constant, and the tree is traversed to the left if the input is less than that constant, otherwise to the right.

Building the model tree consists of three steps: Building the tree, pruning the tree, and smoothing the leaves. To build the tree, start with a root

node, which covers all of the training data, and split the node on the input which will reduce the error by the greatest amount. The split value for each input is the value which would minimize the variation of the training data down each branch. The error is defined as the resulting standard deviation, and the reduction in error can be calculated using the formula:

$$reduction = sd(T) - \sum_i \frac{|T_i|}{|T|} * sd(T_i)$$

Here, T is the training set at the current node and T_i is the training set that would remain if input i were split. M5 stops splitting when either the variance of the training set at an input is sufficiently small, or when there are too few remaining data instances.

To prune the tree, M5 begin by building linear regressions on every node of the tree, and compares the estimated error of the node with that of the linear model. If the error of the linear model is reduced, the tree is pruned at that node.

Finally, the linear models are smoothed to correct for the discontinuities between the leaves. The Weka implementation of smoothing is different from that described by Quinlan, and is more similar to that described in [6], though a detailed explanation is missing.

3.2 Narrowing the Optimization

The original routine used in optimizing the walk parameters in simulation was called the OmniWalk routine. The simulated agent was ordered to walk to a series of targets relative to its current position for some fixed time. After the time elapses, the parameters were assigned a fitness depending on the agent's distance to the target. If the agent's distance increased, the parameters were assigned a negative fitness.

Having the real robot fall was costly and highly undesirable. Once the port of the HTWK walk had matured, it was possible have the robot walk

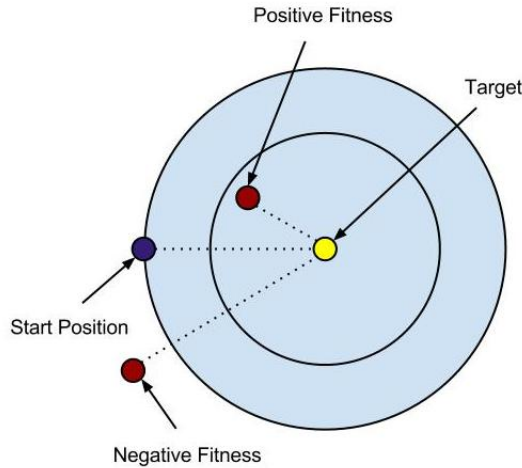

Figure 3.2: The simulated agent is ordered to go to a relative target for a fixed time, and then is assigned a fitness based on the distance to that target.

forwards and turn at half speed without falling. Therefore, both the real and simulated robots were capped at half speed.

Still, the real robot could not move sideways without falling, and yet the OmniWalk optimization routine required an omni-directional walk. So the optimization routine was changed to remove this requirement. Given a target, the robot was ordered to always turn towards the target at a speed proportional to the rotational distance to the target. If the robot was less than 45 degrees from the target, it was then also ordered to walk forward while still turning.

Now the robot could perform the two consecutive iterations of the OmniWalk routine and gather sufficient data for the M5P algorithm to perform well. The generated models forced the movement in simulation to have a falling rate more similar to that of the robot. At the same time, as will be explained in the following section, this method allowed the optimization to select several parameter sets which improved the walking speed of both the simulated agent and the robot.

The original OmniWalk routine required the robot to move to many

different types of targets with the hopes of developing a robust walk for playing soccer. For example, it was desirable for the robot to be able to go to a target, and then quickly change to another target behind it without falling due to its momentum. However, after several trials, it was found that the complete OmniWalk routine was inappropriate for this experiment. The robot was measured in how fast it could walk forward, and it would only add noise to attempt to learn additional actions. The routine was stripped down to only a single target directly in front of the robot at a farther distance than could be possibly reached.

Interestingly, significantly increasing the time limit for this target had a negative effect. The likely reason is that as the simulation runs longer, the cumulative error between the simulation and robot will increase until the simulation can no longer reflect what will happen on the robot. The total walk time used was about fifteen seconds.

3.3 Selecting the Right Parameters

To test its actual walk speed, the robot was placed in front of the goalie box and ordered to walk towards a ball directly in front of it. The forward speed was set to 50%, and the angular speed to the percentage bearing of the ball. If the robot did not see the ball at any particular frame, he was ordered to continue the previous motion. Once the robot stepped with either foot onto the center line of the field, it was ordered to stop and record the time it had been walking. The forward distance to walk was 238 cm. The speed of each parameter set reported here is the average of five trials. If during a trial robot fell or skewed off course, that trial was retried. The original walk speed using this method was measured to be 11.9 cm/s.

The work described in the previous sections successfully constrained CMA-ES to select some parameters that could be run on the real robot. Many of these parameters were generated within the first ten generations of optimization. However, even within these ten generations, many parameters still caused

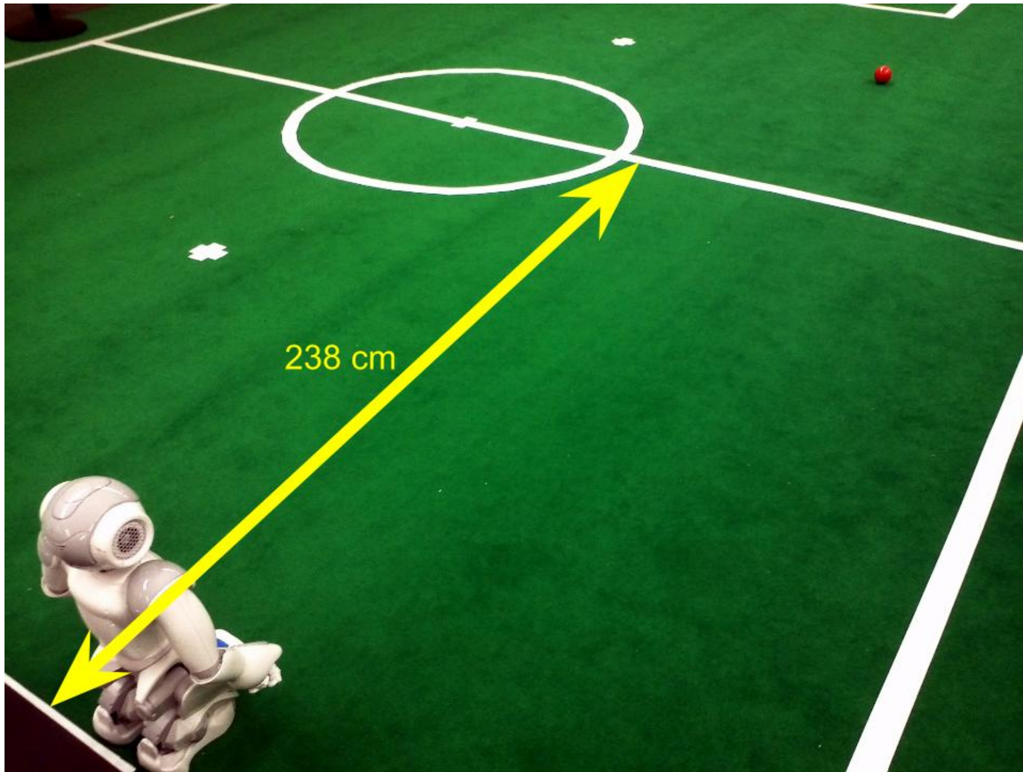

Figure 3.3: To evaluate the robot, it was placed in the position shown in this image and was ordered to walk towards the orange ball. Once it reached the center line of the field, it was ordered to stop, at which point it recorded its own walking time.

the robot to become unbalanced, and most of them did little to actually improve the robot's walk speed. Even though the search space was constrained, it was still too large. What the algorithm needed now was guidance.

The top one or two parameters from each of the first ten generation were tested, and fortunately, more than one were found to perform faster than the original walk. After manually analyzing the parameters, it was found that seven of the parameters, namely the four *balancegyro* parameters and the three *angle* parameters, left the robot unbalanced when changed. This is likely due to the inaccuracies in the simulation's gyro-meter sensor model. Therefore, these parameters were removed from the optimization.

One parameter in particular appeared to increase the robot's walk

speed, which was the *swing* parameter, which controls the amplitude of the swinging foot. Visually, it could be observed that increasing this parameter caused the robot to swing its lifted foot further and take larger steps. To bias the optimization towards using this parameter the initial variance for this parameter was doubled in the CMA-ES algorithm. Finally, the parameters that performed the best (13.8 cm/s) were set to be the new initial parameters for CMA-ES.

The optimization was re-run for ten generations with these changes. Now resulting parameters were more often than not balanced, and often notably better than the original walk. The best actual parameters (14.6 cm/s) were selected from the top valued parameters that the optimization generated, and were used as the new initial parameters for another round of optimization. Unfortunately, while this produced many parameters that performed similarly, none were found that performed significantly better.

The joint learning module in Weka was re-run using data collected while using the current best parameters. Using the new joint models, several better parameter sets were found (15.9 cm/s). This process was repeated with the new best parameters. Unfortunately this did not generate faster parameters. However, since many of the parameters generated were similar to the best, one was found that performed a little slower than the previous best (15.6 cm/s), but appeared as stable as the original. Since these parameters were particularly stable, it was possible to increase the robot's walk speed to 75%, which is the speed used during games with the original parameters. At this speed, the original parameters were measured at 13.5 cm/s. The final optimized parameters were measured at 17.1 cm/s.

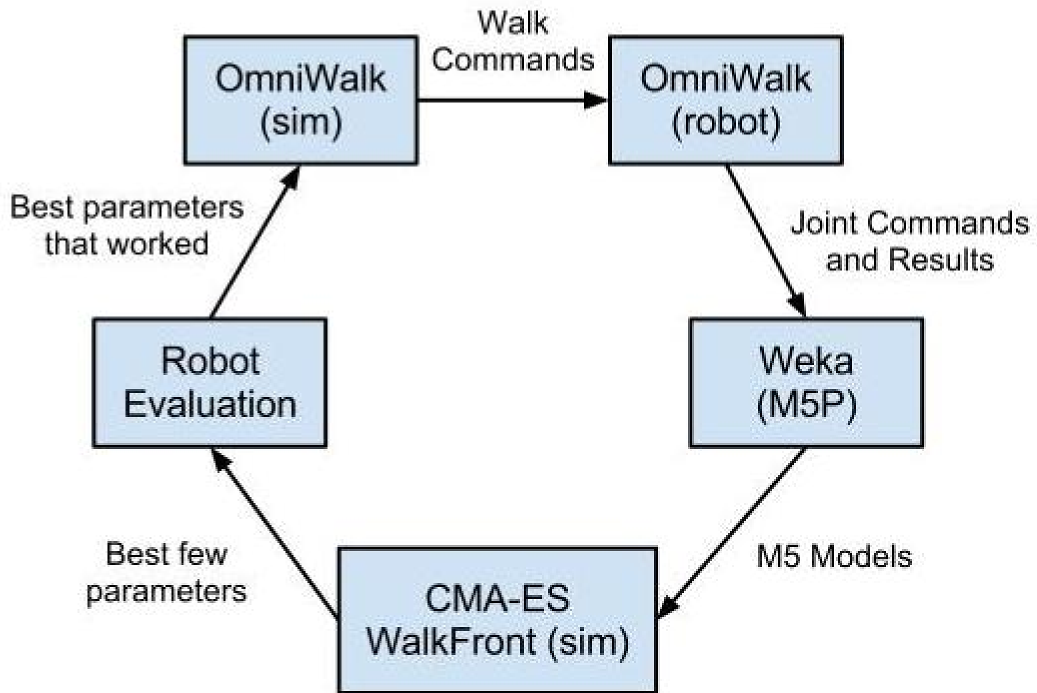

Figure 3.4: The above loop was completed 4 times (though one iteration skipped the Weka step). The loop started with the OmniWalk in simulation. The simulation walked to a series of targets and wrote down each of its received walk requests. Those requests were rerun on the robot, which collected data on its joint commands and the resulting angles. That data was passed to a module build on Weka which built tree models using the M5P algorithm. These models are used to predict joint angles in the simulation. The optimization algorithm CMA-ES was run in simulation, and the computation was distributed to 150 machines using Condor. The best few optimized parameters were tested on the actual robot. The best of those parameters was loaded back into simulation, and the process was repeated.

Chapter 4

Discussion on Simulation Learning

This chapter discusses the fundamental differences between the simulation and the robot, and examines the validity of the approach taken in this thesis. Additionally, similar work in the field of learning in simulation and using simulation to predict real applications is discussed. Finally, this section gives a brief summary of other approaches taken in this project and attempts to explain why they did not succeed.

4.1 Assessment of the Approach

There are thousands of ways in which the simulation performs differently from reality. However, the interest of this thesis is to improving the forward walk speed of the robot by exclusively changing the walk engine parameters. Therefore, the only differences which are of concern fall into the following two categories.

There are differences that produce parameters that make the robot fall, but not the simulation. Given a command, the simulation will produce different joint angles than the real robot. Even if these angles were the same, the physics simulator may determine that a particular joint configuration does

not make the robot fall, when in reality it does. The robot used in these experiments had one leg weaker than the other, which is not modeled in simulation and yet may cause it to fall. To avoid such differences, the simulator can be constrained to perform only actions which are known to be stable in both mediums. This was done by reducing the walk to 50%, removing sideway movement from the OmniWalk optimization, and removing parameters from the optimization which unbalanced the robot. Using the learned joint models in simulation also succeeded in adding noise, which constrained the optimization from following some paths that exploit a particular feature of simulation.

The other category of differences are those which improve the walk speed of the simulation, but not of the robot. The purpose of the learned joint models was to force the simulation to act more similarly to the real robot. This was seen to have had some effect when better parameters could not be found until these models were used and subsequently updated. However, this method still produced quite a few parameters that performed well or poorly in simulation, but had the opposite effect on the robot. Manual intervention was necessary to discover which parameters improved both walks, and guide the optimization to use these parameters on a subsequent run of the optimization.

The change presented which produced the least positive result was to match the masses of the simulated parts of the agent to those of the robot. While this may have increased the accuracy of simulation by some amount, it did not directly attack either of the two previously mentioned differences.

4.2 Related Work

Simulation has been used as a tool for modern research and development. Stefan H. Thomke [13] describes how simulation has been used for crash testing in the automobile industry for over a decade. Building real models for crash testing is much more expensive and time consuming than building a model in simulation. Additionally, real models are limited in value since they are often

built after the automobile's design can no longer be changed, and due to the large number of possible crash scenarios. Using simulation for the Nao provides the same benefits. However, in industry *humans* typically learn by iterating in simulation in order to modify their design on the product. In contrast, this project does not make any manual changes to the Nao, but seeks to guide the simulation to improve it automatically.

Simulation has been recently used in robotics research to develop and test new algorithms such as for path planning models [15]. In particular, learning has been used in the development of learning algorithms for applications such as for modeling robots with many sensors and actuators [10], and in 1990 for learning evasive maneuvers in flight simulation [4]. Intuitively, learning in simulation lends itself well towards active learning, when a database of training data in unavailable. However, there is fewer literature examining the use of results obtained from simulation learning in the real world.

The price of the convenience of simulation is its inaccuracy. As Erann Gat puts it, "You cant do science about robots without firing up a robot" [3]. In his 1995 paper, Gat claims that in order for simulation to be useful, the results of simulation must be tested on the real robot. This is similar to the iterative approach of this thesis. The simulation's optimization must be somehow guided by performance on the real robot, and in this case the guidance is manual.

This manual guidance can be seen as an ad-hoc approach to demonstration learning, described by Stefan Schaal in 1997 [12] Schaal makes the observation that unlike classical reinforcement learning algorithms, humans and animals use prior knowledge to learn a new task. Schaal's description of "extracting a policy" for Q-learning is similar in nature to manually selecting a parameter sample which works well on both the robot and simulation and feeding it to CMA-ES as the initial parameters.

4.3 Failed attempts

There are certainly other methods to accomplish the goals of this thesis. This section is reserved for methods that did not.

4.3.1 Predicting Joint Commands

To constrict the joint angles to those on the simulator, the joint commands were set to some fraction of the expected output angles from the M5 models and the original commands. The resulting angles were closer to the prediction. Still, they were not quite as accurate as could be desired.

An attempt was made to fix this by adding an inverse mapping from output joint angles to input joint commands. The same 27 inputs and 12 outputs were recorded on the simulator as on the real robot, but the output joints were switched with the input joint commands. This data was run through Weka's M5P algorithm to produce inverse models.

The new walking algorithm went as follows: Given the current state and walk request from the behavior, the walk engine produces a set of joint commands. The expected joint angles that would be produced on the real robot are predicted from models built from data from the real robot. Then, to achieve those joint angles, the final joint commands were predicted from the inverse models built from data from the simulation.

Unfortunately, while this method did achieve its goal, it did not work as desired. The simulated robot *was* able to match the expected joint angles with an error of less than 0.1 radians for every joint on nine out of ten consecutive frames. However, the original joint commands that the walk produced were very smooth, while the transformed joint commands passed through two layers of prediction were too noisy. The simulated robot fell more often than the real robot, even though its input to output mapping of joints appeared to be more accurate.

4.3.2 Predicting Joints on the Real Robot

Instead of making simulation more like the real robot, there is also the possibility to make the robot move more like the simulation. Since (without prediction), the simulated robot is significantly more stable than the actual robot, it could be beneficial to mimic its movement. Additionally it may be possible to use optimized parameters on the real robot if it can walk more like the simulated robot.

All of the methods previously explained to make the simulation move like the robot were inverted and tried on the real robot. A mapping of joint commands to angles was found from simulation data and used to determine commands for the real robot. Adding an inverse mapping was also attempted, which introduced too much noise and was discarded. Again, it was found that a componentization worked the best (0.6 * original + 0.4 * expected).

However, the noise introduced by using prediction had a much greater impact on the robot than it did in the simulation. While the robot appeared to be as stable as without prediction, the speed of the walk dropped by about 30%. Also, this method failed to make the robot more stable when using parameters optimized in simulation, and so it was discarded.

4.3.3 Matching the Frame Rate

By default, the simulator physics takes 20 millisecond steps and interpolates in between. The agent, therefore, receives a chance to read its sensors and update its joint torques once every 20 milliseconds. In contrast, the actual robot gets the same chance every 10 milliseconds, and consequently has twice the number of frames.

There are several reasons that it would be desirable to have the simulator run at the same frame rate as the real robot. In learning the input to output joint mappings on the real robot, every other frame must be skipped. This causes the joint commands to be mapped two frames ahead, which introduces some error. Additionally, there are checks in the code base ensure

that the sensor and joint command values from the simulator are received at 20 milliseconds instead of 10. These checks would be unnecessary if the simulator ran at the real robot's frame rate, and the increased granularity would put the accuracy closer to that of the real robot.

After some digging in the SimSpark code base, it was possible to increase the frame rate to the desired amount, but unfortunately, it seems that the simulation environment or the physics engine itself somehow uses the assumption that a step occurs at 20 millisecond intervals, and so this method was abandoned.

This change, if successful, may have been a step in the wrong direction. The frame rate is an important component to both the robot and simulation, and repairing this issue could possibly have made the simulation more accurate. Yet, it would have still been inaccurate. The question is whether this change would have made an improvement in simulation become an improvement in reality, and it is not the belief of the author that this would have been the case. While this change does improve the accuracy of the simulation, it does little to constrict it from producing unusable results and does not help to guide the optimization.

Chapter 5

Lessons and Conclusion

In this thesis, it was found that learning can be used in simulation to improve a the values of parameters used on a real robot. However, simulation cannot be expected to simulate most details properly, especially if the simulation is a complex combination of parts and layers, as is SimSpark. The key is to focus on the most abstract notion of what it means to find an improvement. Once that notion is identified, the simulator must be iteratively constricted, learned parameters must be tested, and the optimization must be somehow guided until it is found that improvement in simulation is an improvement in reality.

Improvement in this thesis is to walk faster. In the simulation, it is possible to indirectly measure improvement by measuring the remaining distance to a target. Constriction was implemented by predicting and forcing the agent's joint angles, slowing down the robot, and by stripping down the optimization routine to the bear essentials. Testing the output of the optimization iteratively revealed which parameters were more or less appropriate to optimize. By manually focusing on those parameters and guiding the optimization to use previously successful values, parameters were learned that improved the robot's forward walking speed by about 30%.

Bibliography

[1] Open dynamics engine faq. http://ode-wiki.org/wiki/index.php?title=FAQ.

[2] Sven Behnke. Online trajectory generation for omnidirectional biped walking, 2006.

[3] Erann Gat. On the role of simulation in the study of autonomous mobile robots. In *AAAI-95 Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents.*, Stanford, CA, March 1995.

[4] John Grefenstette, Connie Loggia Ramsey, and Alan C. Schultz. Learning sequential decision rules using simulation models and competition. In *Machine Learning*, pages 355–381, 1990.

[5] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2005.

[6] Geoffrey Holmes, Mark Hall, and Eibe Frank. Generating rule sets from model trees. In *Twelfth Australian Joint Conference on Artificial Intelligence*, pages 1–12. Springer, 1999.

[7] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

[8] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[9] Patrick MacAlpine, Samuel Barrett, Daniel Urieli, Victor Vu, and Peter Stone. Design and optimization of an omnidirectional humanoid walk: A winning approach at the RoboCup 2011 3D simulation competition. In *Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, July 2012.

[10] Josep M Porta and Enric Celaya. Reinforcement learning for agents with many sensors and actuators acting in categorizable environments. *Journal of Artificial Intelligence Research*, 23:79–122.

[11] Ross J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[12] Stefan Schaal. Learning from demonstration. In M.C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, Cambridge, MA, 1997. MIT Press.

[13] Stefan H. Thomke. Simulation, learning and r&d performance: Evidence from automotive development. *Research Policy*, 27(1):55–74, May 1998.

[14] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997.

[15] Simon X. Yang and Max Meng. Real-time collision-free path planning of robot manipulators using neural network approaches. *Auton. Robots*, 9(1):27–39, August 2000.