To appear in *AAMAS Adaptive Learning Agents Workshop* (ALA 2013),
Saint Paul, Minnesota, USA, May 2013.

Simultaneous Learning and Reshaping of an Approximated Optimization Task

Patrick MacAlpine, Elad Liebman, Peter Stone
Department of Computer Science
The University of Texas at Austin
Austin, TX 78701, USA
{patmac, eladlieb, pstone}@cs.utexas.edu

ABSTRACT

For many target optimization and learning tasks the sample cost of performing the task is very expensive or time consuming such that attempting to directly employ a learning algorithm on the task becomes intractable. For this reason learning is instead often performed on a less expensive task that is believed to be a reasonable approximation of the actual target task. This paper serves to present and motivate the challenging open problem of simultaneously performing learning on an approximation of the true target task, while at the same time shaping the task used for learning to be a better representation of the true target task. Our work, which is still in progress, is performed in the RoboCup 3D simulation environment where we attempt to learn walk parameters for an omnidirectional walk engine used by humanoid robot soccer playing agents.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Parameter learning*; I.2.9 [**Artificial Intelligence**]: Robotics—*Kinematics and Dynamics*

General Terms

Algorithms, Design, Experimentation

Keywords

Bipedal walking, Robot soccer, Machine learning, CMA-ES

1. INTRODUCTION

When trying to optimize a parameter set for some objective task, the question of how to evaluate and compare the value of different parameter configurations is critical. Ideally, we would like to know how to evaluate the value of any single parameter set directly. More often than not, however, there is no straightforward way to obtain a "magic number" that signifies the value of a configuration. In such cases one may resort to Monte Carlo methods in order to estimate the value. The most common and practical way to do this in many domains is by simulating the objective task repeatedly. However, there are cases in which even a limited scale simulation as a means of evaluating the value of different parameter configurations is infeasible computationally. In such cases the only course of action would be to use some surrogate function that is believed to be strongly correlated to actual success in the objective task.

One way to achieve this is to learn a mapping from the value of some surrogate function, that is more tractable computationally, to the amount of success in the objective task, based on a small empirical sample. This approach, albeit enticing, is dangerous, because such global mapping may not exist - the mapping may change as we transition from one part of the parameter space to another. For example, let us assume we are trying to optimize the parameter set of an autonomous vehicle based on its performance in several racetracks. We can estimate how the performance on the racetracks correlates to the performance on a larger-scale problem, for instance, driving in a small town. However, as parameters change the behavior of the vehicle changes, and we will inevitably see different mappings from racetrack performance to larger-scale performance as the autonomous vehicle changes its policy.

For this reason, in this paper we propose ideas on how to optimize for a given task while simultaneously learning how to reshape it as we traverse different parts of the parameter space. We focus our investigation on the RoboCup 3D simulation environment. In the simulation environment, one of the most crucial aspects of agent gameplay is the simulated robot walk. Optimizing the parameter set that governs the walk has been one of the key challenges in this domain [10]. Ideally, we would want to evaluate any parameter set directly on full 11v11 gameplay. However, that is not computationally tractable. For this reason, in the past we have trained an agent directly on an obstacle course, comprised of 11 different activities. It has been empirically proven that doing well on the obstacle course is correlated with gameplay success. However, other approaches, such as learning from infant walk trajectories [4], and learning to optimize based on trajectories observed in real gameplay, have proven less successful than the obstacle course. What is it then about the obstacle course that makes it effective? More exactly, which of the 11 different walking activities is more significant in learning a successful walk, and could it be that weighting the tasks differently would result in a better parameter set? More interestingly, is it possible that in different stages of the learning, different weighting schemes would result in a better learning rate, and a better final walk? In this paper, we address these issues and present several approaches to tackling them.

The rest of the paper is structured as follows. Section 2 gives a domain description. Sections 3 and 4 describe our agent's omnidirectional walk engine and associated parameters for optimization respectively. Section 5 details our walk optimization tasks. In Section 6 we discuss and analyze the weighting of activities in the context of our optimization task. In Section 7 we present initial approaches and results to the optimization task we have set forth. Section 8 details approaches that we are currently in the process of exploring, and Section 9 summarizes.

2. DOMAIN DESCRIPTION

Robot soccer has served as an excellent platform for testing learning scenarios in which multiple skills, decisions, and controls have to be learned by a single agent, and agents themselves have to cooperate or compete. There is a rich literature based on this domain addressing a wide spectrum of topics from low-level concerns, such as perception and motor control [5, 11], to high-level decision-making problems [8, 12].

The RoboCup 3D simulation environment is based on SimSpark [3], a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine [2] (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints used in the humanoid agents.

The robot agents in the simulation are homogeneous and are modeled after the Aldebaran Nao robot [1], which has a height of about 57 cm, and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. In order to monitor and control its hinge joints, an agent is equipped with joint perceptors and effectors. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time. Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, agents can communicate with each other every other simulation cycle (40 ms) by sending messages limited to 20 bytes. Figure 1 shows a visualization of the Nao robot and the soccer field during a game.

3. WALK ENGINE

The UT Austin Villa 2012 team used an omnidirectional walk engine based on one that was originally designed for the real Nao robot [6]. The omnidirectional walk is crucial for allowing the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach continually changing destinations (often the ball) more smoothly and quickly than the team's previous set of unidirectional walks [13].

We began by re-implementing the walk for use on physical Nao robots before transferring it into simulation to compete in the RoboCup 3D simulation league. Many people in the past have used simulation environments for the purpose of prototyping real robot behaviors; but to the best of our knowledge, ours is the first work to use a real robot to prototype a behavior that was ultimately deployed in a simulator. Working first on the real robots lead to some important discoveries. For example, we found that decreasing step sizes when the robot is unstable increases its chances of catching its balance. Similarly, on the robots we discovered that the delay between commands and sensed changes is significant, and this realization helped us develop a more stable walk in simulation.

The walk engine, though based closely on that of Graf et al. [6], differs in some of the details. Specifically, unlike Graf et al., we use a sigmoid function for the forward component and use proportional control to adjust the desired step sizes. Our work also differs from Graf et al. in that we optimize parameters for a walk in simulation while they do not. For the sake of completeness and to fully specify the semantics of the learned parameters, we present the full technical details of the walk in this section. Readers most interested in the optimization procedure can safely skip to Section 4. The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. The walk engine processes desired walk velocities chosen by the behavior, chooses destinations for the feet and torso, and then uses inverse kinematics to determine the joint positions required. Finally, PID controllers for each joint convert these positions into torque commands that are sent to the simulator.

The walk first selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. We use x as the forwards dimension, y as the sideways dimension, z as the vertical dimension, and θ as rotating about the z axis. The trajectory is chosen using a double linear inverted pendulum, where the center of mass is swinging over the stance foot. In addition, as in Graf et al.'s work [6], we use the simplifying assumption that there is no double support phase, so that the velocities and positions of the center of mass must match when switching between the inverted pendulums formed by the respective stance feet.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 walk engine parameters were used but only the ones we optimize are listed in Table 1.

To smooth changes in the velocities, we use a simple proportional controller to filter the requested velocities coming from the behavior module. Specifically, we calculate $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta_{\text{step}}(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$.

Figure 1: A screenshot of the Nao humanoid robot (left), and a view of the soccer field during a 11 versus 11 game (right).

Notation	Description
$\text{maxStep}_{\{x,y,\theta\}}$	Maximum step sizes allowed for x, y, and θ
y_{shift}	Side to side shift amount with no side velocity
z_{torso}	Height of the torso from the ground
z_{step}	Maximum height of the foot from the ground
f_{g}	Fraction of a phase that the swing foot spends on the ground before lifting
f_{a}	Fraction that the swing foot spends in the air
f_{s}	Fraction before the swing foot starts moving
f_{m}	Fraction that the swing foot spends moving
ϕ_{length}	Duration of a single step
δ_{step}	Factor of how fast the step sizes change
x_{offset}	Constant offset between the torso and feet
x_{factor}	Factor of the step size applied to the forwards position of the torso
$\delta_{\text{target}\{\text{tilt,roll}\}}$	Factors of how fast tilt and roll adjusts occur for balance control
$\text{ankle}_{\text{offset}}$	Angle offset of the swing leg foot to prevent landing on toe
err_{norm}	Maximum COM error before the steps are slowed
err_{max}	Maximum COM error before all velocity reach 0
$\text{COM}_{\text{offset}}$	Default COM forward offset
$\delta_{\text{COM}\{x,y,\theta\}}$	Factors of how fast the COM changes x, y, and θ values for reactive balance control
$\delta_{\text{arm}\{x,y\}}$	Factors of how fast the arm x and y offsets change for balance control

Table 1: Optimized parameters of the walk engine.

In addition, the value is cropped within the maximum step sizes so that $-\text{maxStep}_i \leq \text{step}_{i,t+1} \leq \text{maxStep}_i$.

The phase is given by $\phi_{\text{start}} \leq \phi \leq \phi_{\text{end}}$, and $t = \dfrac{\phi - \phi_{\text{start}}}{\phi_{\text{end}} - \phi_{\text{start}}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{\text{length}}$, until $\phi \geq \phi_{\text{end}}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{\text{start}} = -0.5$ and $\phi_{\text{end}} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

$$\begin{aligned}
k &= \sqrt{9806.65/z_{\text{torso}}} \\
\alpha &= 6 - \cosh(k - 0.5\phi) \\
\phi_{\text{start}} &= \begin{cases} \dfrac{\cosh^{-1}(\alpha)}{0.5k} & \text{if } \alpha \geq 1.0 \\ -0.5 & \text{otherwise} \end{cases} \\
\phi_{\text{end}} &= 0.5(\phi_{\text{end}} - \phi_{\text{start}})
\end{aligned}$$

The stance foot remains fixed on the ground, and the swing foot is smoothly lifted and placed down, based on a cosine function. The current distance of the feet from the torso is given by

$$\begin{aligned}
z_{\text{frac}} &= \begin{cases} 0.5(1 - \cos(2\pi \dfrac{t - f_{\text{g}}}{f_{\text{a}}})) & \text{if } f_{\text{g}} \leq t \leq f_{\text{a}} \\ 0 & \text{otherwise} \end{cases} \\
z_{\text{stance}} &= z_{\text{torso}} \\
z_{\text{swing}} &= z_{\text{torso}} - z_{\text{step}} * z_{\text{frac}}
\end{aligned}$$

It is desirable for the robot's center of mass to steadily shift side to side, allowing it to stably lift its feet. The side to side component when no side velocity is requested is given by

$$\begin{aligned}
y_{\text{stance}} &= 0.5 y_{\text{sep}} + y_{\text{shift}}(-1.5 + 0.5\cosh(0.5k\phi)) \\
y_{\text{swing}} &= y_{\text{sep}} - y_{\text{stance}}
\end{aligned}$$

where y_{sep} is the distance between the feet. If a side velocity is requested, y_{stance} is augmented by

$$y_{\text{frac}} = \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ 0.5(1 + \cos(\pi \frac{t - f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ 1 & \text{otherwise} \end{cases}$$

$$\Delta y_{\text{stance}} = \text{step}_y * y_{\text{frac}}$$

These equations allow the y component of the feet to smoothly incorporate the desired sideways velocity while still shifting enough to remain dynamically stable over the stance foot.

Next, the forwards component is given by

$$\begin{aligned}
s &= \text{sigmoid}(10(-0.5 + \frac{t - f_{\text{s}}}{f_{\text{m}}})) \\
x_{\text{frac}} &= \begin{cases} (-0.5 - t + f_{\text{s}}) & \text{if } t < f_{\text{s}} \\ (-0.5 + s) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ (0.5 - t + f_{\text{s}} + f_{\text{m}}) & \text{otherwise} \end{cases} \\
x_{\text{stance}} &= 0.5 - t + f_{\text{s}} \\
x_{\text{swing}} &= \text{step}_x * x_{\text{frac}}
\end{aligned}$$

These functions are designed to keep the robot's center of mass moving forwards steadily, while the feet quickly, but smoothly approach their destinations. Furthermore, to keep the robot's center of mass centered between the feet, there is an additional offset to the forward component of both the stance and swing feet, given by

$$\Delta x = x_{\text{offset}} + -\text{step}_x x_{\text{factor}}$$

After these calculations, all of the x and y targets are corrected for the current position of the center of mass. Finally, the requested rotation is handled by opening and closing the groin joints of the robot, rotating the foot targets. The desired angle of the groin joint is calculated by

$$\text{groin} = \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ \frac{1}{2}\text{step}_\theta(1 - \cos(\pi \dfrac{t - f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ \text{step}_\theta & \text{otherwise} \end{cases}$$

After these targets are calculated for both the swing and stance feet with respect to the robot's torso, the inverse kinematics module calculates the joint angles necessary to place the feet at these targets. Further description of the inverse kinematic calculations is given in [6].

To improve the stability of the walk, we track the desired center of mass as calculated from the expected commands. Then, we compare this value to the sensed center of mass after handling the delay between sending commands and sensing center of mass changes of approximately 20ms. If this error is too large, it is expected that the robot is unstable, and action must be taken to prevent falling. As the robot is more stable when walking in place, we immediately reduce the step sizes by a factor of the error. In the extreme case, the robot will attempt to walk in place until it is stable. The exact calculations are given by

$$\begin{aligned}
\text{err} &= \max_i(\text{abs}(\text{com}_{\text{expected},i} - \text{com}_{\text{sensed},i})) \\
\text{stepFactor} &= \max(0, \min(1, \frac{\text{err} - \text{err}_{\text{norm}}}{\text{err}_{\text{max}} - \text{err}_{\text{norm}}})) \\
\text{step}_i &= \text{stepFactor} * \text{step}_i \; \forall i \in \{x, y, \theta\}
\end{aligned}$$

This solution is less than ideal, but performed effectively enough to stabilize the robot in many situations.

4. OPTIMIZATION OF WALK ENGINE PARAMETERS

As described in Section 3, the walk engine is parameterized using more than 40 parameters. We initialize these parameters based on our understanding of the system and by testing them on an actual Nao robot.

The initial parameter values result in a very slow, but stable walk. Therefore, we optimize the parameters using the CMA-ES (Covariance Matrix Adaptation Evolution Strategy) algorithm [7], which has been successfully applied previously to a similar problem in [13]. CMA-ES is a policy search algorithm that successively generates and evaluates sets of candidates sampled from a multivariate Gaussian distribution. Once CMA-ES generates a group of candidates, each candidate is evaluated with respect to a *fitness* 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. The covariance matrix of the distribution is also updated to bias the generation of the next set of candidates toward directions of previously successful search steps. As CMA-ES is a parallel search algorithm, we were able to leverage the department's large cluster of high-end computers to automate and parallelize the learning. This allowed us to complete optimization runs requiring 420,000 evaluations in less than two days. This is roughly a 150 times speedup over not doing optimization runs in parallel which would have taken over 100 days to complete.

As optimizing 40 real-valued parameters can be impractical, a carefully chosen subset of 25 parameters (shown in Table 1) was selected for optimization while fixing all other parameters. The chosen parameters are those that seemed likely to have the highest potential impact on the speed and stability of the robot.

5. WALK OPTIMIZATION TASKS

Ideally we would like to be able to optimize walk parameters for the omnidirectional walk engine directly over the task of playing 11v11 soccer. Unfortunately the task of playing a soccer game takes too much time, and requires too much in the way of computational resources to have many games being run in parallel on our computing cluster, for this to be a feasible option. Instead we created an approximated task of playing soccer in which the agent is asked to move to a series of target positions on the field in the form of an obstacle course. This optimization task, which we refer to as the goToTarget task, is described in Section 5.1. The goToTarget task only takes about three minutes to run many instances in parallel, as opposed to the several hours it takes to run full games in parallel on our computing cluster.

Additionally we created a 4v4 task of playing soccer, described in Section 5.2, which gives a better approximation of playing a full 11v11 soccer game, but still takes around 20 minutes to run instances of this task in parallel on our computing cluster. Because the 4v4 task takes so long we only run it sparingly as a reference point to see how walk parameters being learned might fair in a full game.

The goToTarget task is broken up into eleven different activities shown in Figure 2. We would like to learn a set of weights for the rewards given by each of these individual activities, where the sum of the rewards of the activities multiplied by their respective weights determines the overall fitness of the agent on the goToTarget task, such that the agent is able to learn walk parameters that perform better on the 4v4 task.

$$\text{reward}_{\text{goToTarget}} = \sum_{i \in [1,11]} w_i \cdot r_i$$

where r_i is the reward from the i-th activity and w_i is its weight.

5.1 Go to Target Optimization Task

In order to simulate common situations encountered in gameplay, the walk engine parameters are optimized for a goToTarget task.[1] This task consists of an obstacle course in which the agent tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the agent is rewarded based on its distance traveled toward the active target. If the agent reaches an active target, the agent receives an extra reward based on extrapolating the distance it could have traveled given the remaining time on the target. In addition to the target positions, the agent has stop targets, where it is penalized for any distance it travels. To promote stability, the agent is given a penalty if it falls over during the optimization run.

In the following equations specifying the agent's rewards for targets, *Fall* is 5 if the robot fell and 0 otherwise, d_{target} is the distance traveled toward the target, and d_{moved} is the total distance moved. Let t_{total} be the full duration a target is active and t_{taken} be the time taken to reach the target or t_{total} if the target is not reached.

$$\text{reward}_{\text{target}} = d_{\text{target}} \frac{t_{\text{total}}}{t_{\text{taken}}} - \mathit{Fall} \quad (1)$$

$$\text{reward}_{\text{stop}} = -d_{\text{moved}} - \mathit{Fall} \quad (2)$$

The goToTarget optimization includes quick changes of target/direction for focusing on the reaction speed of the agent, as well as targets with longer durations to improve the straight line speed of the agent. The stop targets ensure that the agent is able to stop quickly, while remaining stable. The trajectories that the agent follows during the optimization are described in Figure 2.

5.2 4v4 Optimization Task

Instead of playing a full 11v11 game we instead play a single half of a 4v4 game. A team is rewarded for both scoring goals and also for moving the ball toward the opponent's goal. The reward function used for this task is

$$\begin{aligned}\text{reward}_{\text{4v4}} = {} & \mathit{goalsFor} * \frac{1}{2}\text{Field\_Length} \\ & - \mathit{goalsAgainst} * \frac{1}{2}\text{Field\_Length} \\ & + \mathit{avgBallXPosition}\end{aligned}$$

where *avgBallXPosition* is the average position of the ball in the X (forward/backward) direction since the last goal was scored or, if neither teams scores, the average position of the ball from the beginning of the game. The position values are relative to the distance from the midline of the

[1] Video of the agent performing the goToTarget optimization task can be found at www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/walk.html

1. Long walks forward/backwards/left/right
2. Walk in a curve
3. Quick direction changes
4. Stop and go forward/backwards/left/right
5. Alternating moving left-to-right & right-to-left
6. Quick changes of target to simulate a noisy target
7. Weave back and forth at 45 degree angles
8. Extreme changes of direction to check for stability
9. Quick movements combined with stopping
10. Quick alternating between walking left and right
11. Spiral walk both clockwise and counter-clockwise

Figure 2: GoToTarget Optimization walk trajectories

field with negative position values in a team's defensive half of the field and positive values in the offensive half.

When running the 4v4 task we used a common fixed opponent: a baseline agent optimized with the goToTarget optimization task during which all activity rewards were weighted equally.

6. THE WEIGHTING PROBLEM

In order to learn an effective walk, we thus far focused on having the agent train on an obstacle course comprising 11 different walk trajectories (see Figure 2). The obstacle course serves as a computationally tractable surrogate for the 4v4 game task, by which we measure what we consider to be the true fitness of a solution. Given that the different walk trajectories are fixed, the most crucial aspect of the obstacle course fitness approximation is the weighting. By default, the aggregate reward obtained by each walk sample is just the sum of rewards for each trajectory, meaning they are all equally weighted.

It is extremely unlikely, however, that the 11 trajectories are indeed equally important, implying that this is indeed the optimal weighting. In other words, assuming each walk trajectory task (or "activity", as we refer to it from this point onwards) contributes differently towards an effective walk, than a better weighting scheme should produce a superior result.

Table 2 presents the fitness of walks obtaining training on single activities. They are all degraded compared to the walk learned from combining the rewards, implying that indeed at least a mix of different activities is required to learn an effective walk. The fitnesses vary quite heavily, however, implying that each activity does contribute differently to the final result. We note in Section 7.1 that a baseline agent trained on the entire set of equally weighted activities should have an expected 4v4 game fitness of 0.

Another crucial observation is that different weighting schemes for the various activities do matter in terms of the fitness of the walk learned. To illustrate this point, we first compare the fitnesses of walks trained on all activities except one (i.e. "knocking-out" one activity at a time and combining the rest - a $[1, 1, \ldots, 0, 1, \ldots, 1]$ weighting scheme) in Table 3. While removing some activities (3 and 4) harms the fitness considerably, in two cases (activities 1 and 8) it seems that removing the activities actually contributes to the fitness. This further strengthens our intuition that the current weighting scheme is suboptimal, and that we would like to find a good way of learning better weights as we optimize for the original task.

Activity	Fitness	StdErr
1	-26.9605623276	1.29619603248
2	-31.250364697	1.08828864978
3	-26.2453386308	1.15218299104
4	-23.7791771111	1.07371662994
5	-65.9514994776	1.28455077906
6	-66.0047726619	0.91150218967
7	-44.4248067669	1.15508047473
8	-79.6944283186	0.940911592582
9	-80.1612411765	0.815991750977
10	-68.7431571429	0.958372832705
11	-82.8619171642	0.928203179673

Table 2: The fitness obtained from learning on each activity in the obstacle course separately (# generations = 400). All the fitnesses are negative, meaning they are inferior compared to the walk trained on a combination of all activities. It is clear however that different activities lead to much better walks compared to others (1, 3 and 4 compared to 8 and 9, for instance). This leads us to the conclusion that they contribute differently to the overall walk fitness.

Removed Activity	Fitness	StdErr
1	5.14204210992	1.34800057635
2	1.52892855814	1.38832807929
3	-23.0764260492	1.46600109146
4	-12.437155757	1.54833200304
5	0.180571428689	1.46645622415
6	1.80133529091	1.52815390984
7	-0.996885825397	1.23685369536
8	4.26231683333	1.75793256622
9	-7.97918691071	1.29305579673
10	2.47325064844	1.50828940856
11	2.40324721429	1.42510368531

Table 3: The fitness obtained from learning on all activities except one (# generations = 400). Different activities detract differently from the overall fitness once removed. In two cases (activities 1 and 8) it seems that removing the activities actually contributes to the fitness.

Lastly, we note that changing the weights does matter even when they are all positive. To prove this point, we compare the fitnesses of walks learned on all the activities when one of the activities is given a double weight compared to the others (i.e. a $[1, 1, \ldots, 2, 1, \ldots, 1]$ weighting scheme) in Table 4. In certain cases doubling the weight of an activity can have a negative effect on the overall fitness (for instance, activity 10), whereas it sometimes improves fitness somewhat (as is the case for activities 2 and 4). Interestingly, while removing activity 9 harmed the fitness (it's value after removal was -7.98), doubling it also harms the fitness somewhat (-3.07).

Doubled Activity	Fitness	StdErr
1	1.12573153279	1.53372958915
2	5.2379902459	1.2398576484
3	-0.372906246032	1.37630012203
4	4.71968851008	1.44316311066
5	-3.65889347287	1.52761938361
6	-1.32128969672	1.41529373379
7	5.32546797479	1.45144844538
8	-6.35841603361	1.26643719415
9	-3.07685847934	1.30071004926
10	-18.182396626	1.32485039788
11	4.20284932422	1.51572553484

Table 4: The fitness obtained from learning on all activities with a double weight on one of them (# generations = 400). Different activities affect the overall fitness differently once doubled. In certain cases doubling the weight of an activity has adverse affects (for instance, activity 10) and in some cases it improves the fitness (activities 2 and 4, for instance).

7. INITIAL APPROACHES AND RESULTS

7.1 Baseline

Our baseline for any attempt to improve on the approximation of game fitness is the standard obstacle course. Using the rewards obtained from the obstacle course, the learning converges after ∼200 generations, leading to a game fitness that is 0 in expectation (this should come as no surprise, as the 4v4 game fitness is estimated against the walk learned using the obstacle course). Figure 3 presents the learning rate for the baseline - the growth of average game fitness over generations.

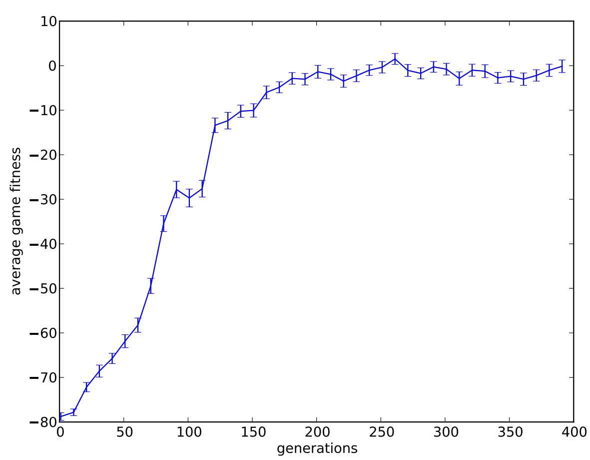

Figure 3: Learning rate for baseline walk. Error bars show the standard error.

7.2 Linear Regression

Perhaps the most straightforward approach to assigning weights to the rewards obtained in each activity is to find how the rewards can best be combined to predict 4v4 game fitness. To do this, we apply multivariate linear regression at each step to find the coefficients which solve $Rw - f = 0$ with R being the $11X population\_size$ reward matrix at each generation, f being the fitness vector, and w being the weight coefficients which we are trying to solve. We can do this whenever we evaluate 4v4 game fitnesses, and adjust our weights accordingly.

Unfortunately, the weights may often be negative. This can be either the result of the inherent noise in our learning process, or because in certain contexts, doing well on a certain task i might be at the expense of doing well on another task, j, and if j is more dominantly correlated to game fitness, a negative coefficient for i might ensue. Negative weights are detrimental because in effect they encourage the optimization to do poorly on certain tasks. For this reason, different variations of this approach were implemented.

7.3 Linear Regression with Negative Weights Elimination

Given that a simple linear regression would return negative coefficients, a straightforward approach would be to cap negatively weighted rewards at 0. Thus the optimization does not obtain a reward for failing in a task but is not directly encouraged to succeed in it, as the aggregate reward is only determined by the rewards gained in other activities.

The growth of fitnesses over generations (as measured on 4v4 games) using this approach is shown in Figure 4. While learning is evident, the algorithm seems to reach convergence after ∼150 generations and remains sub-par compared to the baseline obstacle course approach.

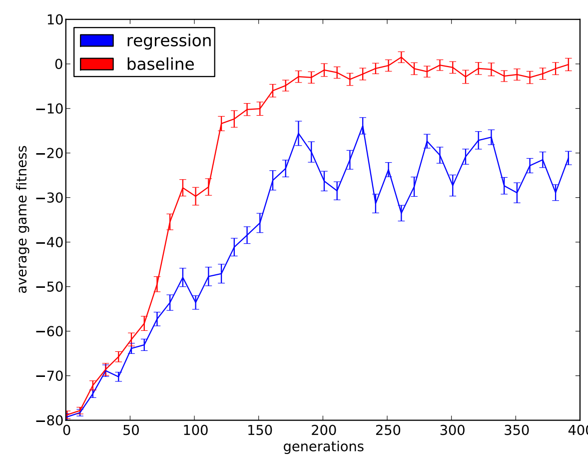

Figure 4: Learning rate for regression based weighting with negative weight elimination, compared to the baseline. Error bars show the standard error.

Let R^* be the subset of the rewards matrix that is only comprised of positively weighted columns. Let w^* be the vector of positive weights for these columns. The drawback of the approach proposed above is that w^* does not solve $R^*w^* - f = 0$ and there is no reason to assume it minimizes $R^*w^* - f$.

7.4 Non-Negative Linear Regression

A different, more rigorous approach to the negative coefficients problem is to directly solve the $Rw - f = 0$ least squares regression problem subject to the constraints that $\forall i.w_i >= 0$. This kind of regression is known as non-negative least squares (NNLS)[9], and can be solved using quadratic programming.

Empirically, however, we discovered that after ~120 generations the regression converges to $\forall i.w_i = 0$ coefficients, essentially terminating the process of learning.

7.5 Bounded Linear Regression

To circumvent the problem of converging to $\forall i.w_i = 0$, a different approach would be to prevent the weights from dropping below a certain positive value. Along the same lines, one might not perhaps want the proportions between the weights to be too extreme, effectively cancelling out some activities compared to the effect of others on the total aggregate reward. This implies solving the least squares problem $Rw - f = 0$ with respect to the constraints

$$\forall i.min_w \leq w_i \leq max_w$$

For any chosen set of parameters max_w, min_i. This, too, can be solved directly using quadratic programming[9].

Empirically, however, after ~150 generations the algorithm converges to $\forall i.w_i = 1$, which is equivalent to the obstacle course default, making any attempts at further optimization uninteresting.

7.6 Caveat in Least Squares Approach

The correlation between game fitnesses and aggregate reward is dependent on three things: the rewards for individual activities, and their weighting. The rewards for individual activities are in turn dependent on the learning of the CMA-ES algorithm. As the population gets better at the task, we expect the correlation to increase. However, any correlation relies on the variance in the population being more dominant than the random noise inherent in any of the estimations. Therefore, as the population converges, the noise becomes increasingly more dominant in the correlation, reducing its value, up to the point of convergence, where we are implicitly correlating random noise, with the expectation of 0. At this point, any least-squares solution would be essentially meaningless. Ironically, the better the population gets, the harder it is to derive meaningful weights for single activity rewards from our data.

This observation is clearly evident in Figure 5, we can see that as the population variance decreases, the correlation between game fitness and the accumulated reward decreases. However, we know that at this point the CMA-ES algorithm has converged to a fairly successful parameter configuration, and that this is an artifact of the per-generation correlation based approach.

8. APPROACHES IN PROGRESS

Following the observations described in the previous section, we have devised several new approaches meant to overcome some of the difficulties posed by simple regression-based approaches.

8.1 Correlation Based Approach

A wholly different approach that has been implemented and is currently being tested is the following. Since the samples at a single generations are extremely noisy both with respect to the game fitness evaluation and the estimation of the rewards, at each game evaluation phase we estimate

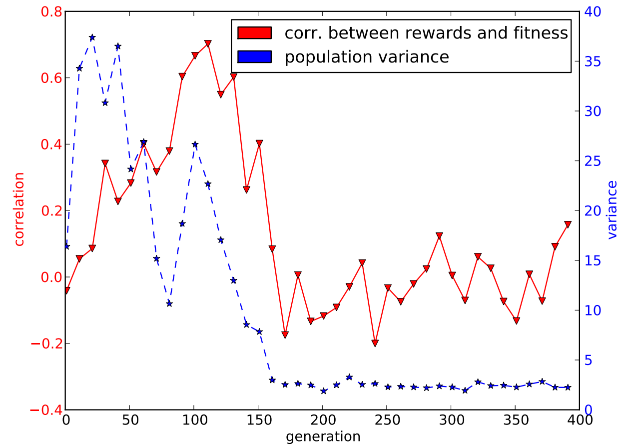

Figure 5: Correlation between game fitness and reward vs. variance

the standard Pearson correlation between the entire set of rewards observed for a given activity i and the entire set of game fitnesses, as observed for all the generations in which game fitness was evaluated. This kind of aggregate method drastically reduces the noise as we progress, and is hinged on the observation that even as population samples change the information obtained in previous generations is still valuable. This approach is also less likely to be susceptible to the phenomenon of population convergence described in 7.6, at least until population convergence is dominant across many generations, which does not happen in the current setting.

Initial results for this approach can be seen in Figure 6, where the growth of fitness over generations (as measured on 4v4 games) is shown. While the results are still preliminary, it seems that learning is still ongoing up to ~200 generations, also somewhat improving on the naive regression-based results in Section 7.3.

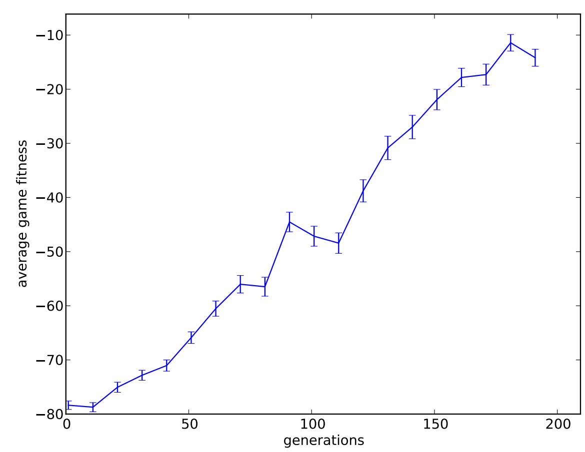

Figure 6: Learning rate for the correlation-based weighting approach. Error bars show the standard error.

8.2 Confidence-Based Weight Update

Another approach, which we are also testing at the moment, relies on the assumption that the weighting from previous generations of the CMA-ES run might still hold valuable information (similar to the key observation in Section 8.1). More importantly, after finding new weight coefficients using linear regression, it only adjusts the current weights to the extent it trusts the correlation between the rewards and the fitnesses. The stronger it is, the more we update. Thus, after each game fitness evaluation, our weight update scheme is $w_i^{new} = max(w_i^{old} + |\rho_i| \cdot (w_i^r - w_i^{old}), 0.1)$, where ρ_i stands for the correlation between obtained rewards and game fitnesses for the i-th activity, and w_i^r is the coefficient for the i-th activity rewards obtained via linear regression. The weights are kept at > 0.1 to avoid their convergence to $\forall i. w_i = 0$ as we've seen for the NNLS approach (see Section 7.4).

8.3 Ranking Approach

One of the risks when working with weighted aggregate rewards is that in some cases the rewards from some activities may completely dominate the aggregate reward in a way that is not mitigated by the weighting scheme, especially if the weighting is done based on correlation. Another key observation is that the CMA-ES algorithm does not rely on specific values, only on the ranking of the population which they induce. Therefore, a possible way to mitigate this would be to use ranks directly, instead of reward values. In this scheme, we take the ranks of the population across the different activities and aggregate them into a ranking in the following way:

$$reward^j = \sum_i w_i \frac{1}{rank_i^j}$$

where $reward^j$ is the reward for the j-th sample in the population, $rank_i^j$ is the reward for this sample in the i-th activity, and w_i is the weighting of this activity.

This essentially obtains the aggregate inverse ranking of the population. The ranking is inverted so that the setting remains a maximization problem.

Several weighting approaches are currently being tested:

- *Pearson correlation* - This approach is essentially similar to that proposed in 8.1.
- *Spearman's rank correlation* - since the measurements we are correlating may be extremely noisy, we are also calculating correlation weights based on Spearman's rank correlation which correlates the rank vectors of the game fitnesses and the activity rewards, obtaining a more robust correlation score that is less sensitive to individual values.
- *Wilcoxon's t-test* - This measure utilizes Wilcoxon's t-test, to test the hypothesis that the population mean ranks of the activity rewards and the game fitnesses are equal, obtaining a p-value that is used to obtain the weight.

9. SUMMARY

We have presented the problem and challenge of simultaneously performing learning on an approximation of a target optimization task, while at the same time shaping the task used for learning to be a better representation of the true target optimization task. In the coming weeks we intend to further explore and analyze the approaches we have outlined for learning walk parameters for an omnidirectional walk in the RoboCup 3D simulation domain. For future work we would like to discover ways to dynamically create useful new trajectories for our obstacle course optimization task in addition to varying how we weight different preexisting parts of the task.

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to Samuel Barrett who wrote the walk engine used during this research. Also thanks to Yinon Bentor and Suyog Dutt Jain for contributions to early versions of the optimization framework employed by the authors. LARG research is supported in part by grants from the National Science Foundation (IIS-0917122), ONR (N00014-09-1-0658), and the Federal Highway Administration (DTFH61-07-H-00030). Patrick MacAlpine is supported by a NDSEG fellowship.

10. REFERENCES

[1] Aldebaran Humanoid Robot Nao. http://www.aldebaran-robotics.com/eng/.

[2] Open Dynamics Engine. http://www.ode.org/.

[3] SimSpark. http://simspark.sourceforge.net/.

[4] K. Adolph, W. Cole, M. Komati, J. Garciaguirre, D. Badaly, J. Lingeman, G. Chan, and R. Sotsky. How do you learn to walk? thousands of steps and dozens of falls per day. *Psychological Science*.

[5] S. Behnke, M. Schreiber, J. Stückler, R. Renner, and H. Strasdat. See, walk, and kick: Humanoid robots start to play soccer. In *Proc. of the 6th IEEE-RAS Int. Conf. on Humanoid Robots (Humanoids 2006)*, pages 497–503. IEEE, 2006.

[6] C. Graf, A. Härtl, T. Röfer, and T. Laue. A robust closed-loop gait for the standard platform league humanoid. In *Proc. of the 4th Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS Int. Conf. on Humanoid Robots*, pages 30 – 37, 2009.

[7] N. Hansen. *The CMA Evolution Strategy: A Tutorial*, January 2009. http://www.lri.fr/~hansen/cmatutorial.pdf.

[8] S. Kalyanakrishnan and P. Stone. Learning complementary multiagent behaviors: A case study. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 153–165. Springer, 2010.

[9] C. L. Lawson and R. J. Hanson. *Solving least squares problems*. 3 edition, 1995.

[10] P. MacAlpine, S. Barrett, D. Urieli, V. Vu, and P. Stone. 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-12)*, July 2012.

[11] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[12] P. Stone. *Layered Learning in Multi-Agent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, USA, December 1998.

[13] D. Urieli, P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, and P. Stone. On optimizing interdependent skills: A case study in simulated 3D humanoid robot soccer. In *Proc. of the Tenth Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pages 769–776, May 2011.